Acumatica Framework Development Guide

Contents

Copyright

4030 Lake Washington Blvd NE, Suite 100
Kirkland, WA 98033

Software Version - 5.0

Last updated: October 21, 2014

Acumatica Framework Overview

This chapter provides a high-level overview of Acumatica Framework architecture and components and highlights the main concepts behind the platform design, in the following topics:

Introduction

Acumatica Framework is a modern web application development platform designed for developing business applications. This document provides a high-level overview of Acumatica Framework architecture and components and highlights the main concepts behind the platform design.

CTOs, Software Architects and Application Developers who are interested in using Acumatica Framework for commercial or internal software development are the target audience of this document.

In addition to delivering traditional features specific to enterprise resource planning (ERP) development platforms, Acumatica Framework introduces advanced features and functionality necessary for the development of web applications, as listed below.

Modern Web Technology

- Desktop-like GUI functionality and accessibility through a web browser
- Security model that eliminates the possibility of browser-side data manipulation
- Excellent application performance, even over latent and unreliable Internet connections
- Cross-platform compatibility at the web browser level

Readiness for Data Center and SaaS Delivery Models

- Ability to scale horizontally and run on server farms behind a load balancer
- High application density, which allows for the maximum number of users per server
- Built-in support for multi-tenancy
- Centralized upgrade and versioning management

Tools for Personalization, Customization, and Integration with External Systems

- Built-in localization and personalization support
- Tools for customizing applications at the graphical user interface (GUI), business logic, and database levels, including the integrated web interface and Acumatica Extensibility Framework
- Tools for developing add-on modules and components
- Generic Web Service application programming interface (API) for accessing the business logic

Acumatica Framework not only enables the development of modern web applications, but also provides application developers with everything they need to develop and maintain applications in a fast and cost-efficient way. This maximum efficiency of application development is achieved through the following items.

Development Environment Built on an Industry-Standard Platform

- Runtime environment built on top of a Microsoft.NET platform
- Development environment built on top of Microsoft Visual Studio IDE
- Ready to host on Microsoft Azure

System Foundation Layer

- Set of low-level components and primitives required for full-cycle application development
- Database access layer and primitives to isolate the application developer from database specific logic
- Set of integrated UI elements to isolate application developers from HTML, HTTP, and JavaScript
- Application programming model that isolates the business logic layer from the presentation and data access layers
- Security model that is transparent to the application developer
- Set of wizards and designers to automate the creation of database access and presentation layers
- Set of extendable templates for creating typical application webpages

Application Foundation Layer

- Common application frameset and site management application
- Built-in security management and user management application
- Integrated report designer and report engine
- Integrated Help management system
- Integrated document management system
- Translation and localization tools

Acumatica Framework and Modern Web Development

The inspiration behind Acumatica Framework was the concept of creating a commodity platform that enables the development of contemporary web applications. To achieve this task two items must be addressed:

- Providing the technologies and runtime architecture that deliver the features and functionality of a modern web application
- Providing the development tools and methodology that make it a commodity product for application development

This section explains the technologies implemented in the runtime design of Acumatica Framework that that address these items. Development tools and development process are covered in Chapter 7 of this document.

What is a Modern Web Application?

In our vision, a modern web application can be differentiated from traditional desktop or web applications by combining the following features:

- The primary client interface is a web browser and can be accessed from anywhere via an ordinary Internet connection
- The application does not require any files or components to be installed on the client's computer
- The application is easy and convenient to use, especially when compared to similar Desktop applications
- The application addresses issues related to slow and unreliable Internet connections without affecting the user experience.
- The application addresses security issues related to the exchange of confidential data over a public Internet connection and eliminates the possibility of client-side data manipulation
- The application can be configured and operated in high-availability mode so that the failure of one of the deployment infrastructure nodes does not result in data loss or prevent the application from its normal operations
- It should be possible to scale the application horizontally, which means there is a nearly linear increase of the application throughput in terms of number of users, number of tenants and number of transactions by adding more computing power
- The application is designed for datacenter deployment and natively supports deployment and operation in a multi-tenant environment
- Operating in a multi-tenant environment does not compromise application density, application performance or application security

Each of the points above can be addressed individually, but when combined, they present quite a challenge to application development and the runtime architecture. The articles below explain how these challenges were addressed during the design of the Acumatica Framework runtime components.

Interactive GUI using an Internet Browser

To provide an interactive GUI through the web browser interface Acumatica Framework exposes a set of advanced web controls through the browser Document Object Model (DOM) and implements a communication layer between these controls and the application server through the XMLHttpRequest object in the web browser. This technology can be referred to as an AJAX application model.

The client-side Acumatica Framework web controls are designed as a set of JavaScripts functions that are downloaded during the initial application load and then cached by the web browser. Each application screen is a standard HTML page that contains the details of the screen layout and references to the client side web controls. When combined, the HTML page and the web controls produce an interactive web page that is similar in functionality and behavior to traditional Desktop applications.

Additionally, this technology only requires a standard web browser and does not require the installation of any client-side software or redistributable components. It also works over HTTP or HTTPS protocol which makes it available virtually everywhere.

Performance over Unreliable and Latent Connections

An application written with Acumatica Framework provides good performance even over unreliable and latent Internet connections. This is achieved through the following techniques:

- JavaScript is moved into static generic classes that are loaded one time, when opening the application, and then cached by the browser.
- The static HTML part of the form is minimized to present only the visible screen area. The rest of the form is loaded on demand.

- After the initial form load, only the modified data is sent between the client and server to minimize network traffic and improve response time.
- Server is optimized for the fastest possible request execution.

Browser Level Cross-Platform Compatibility

Generally, an application written with Acumatica Framework is supported by any browser that is compatible to the Level 2 Domain Object Model standard maintained by W3C.

An application written with Acumatica Framework can be accessed through the following Web browsers:

- Internet Explorer
- Mozilla Firefox
- Apple Safari
- Google Chrome

These browsers are available on Windows, Linux and Mac OS platforms providing cross-platform application compatibility.

The list of supported Internet browsers will be extended in the future.

Prevention of Client Side Data Manipulation

The AJAX programming model assumes the use of browser side JavaScript. The JavaScript executed in the browser is not protected, enabling a user to take control of the executed code using a JavaScript debugger. This means that any application logic written with JavaScript is vulnerable to data manipulation. For business applications this means that any data received from the client cannot be trusted and needs to be re-validated when received by the server[1].

With Acumatica Framework, JavaScript is only used for handling initial data format validation, GUI related logic and synchronizing the browser content with data located on the application server. All business logic is executed exclusively on the application server. All data validation logic is duplicated on the application server to prevent any possibility of data manipulation on the client-side.

[1]This assumption is only valid for applications where data manipulation on the client side is not acceptable. For a large range of applications where data manipulation on the client side is not critical, business logic can be moved to the client browser.

Exchange of Confidential Data over the Internet

Acumatica Framework relies on and supports the HTTPS protocol to provide confidentiality of data transmitted over the Internet. This is the same technology the banking industry uses to provide on-line Internet banking services.

High Performance, Scalability and Availability Support

To achieve horizontal scalability and fault tolerance an application written with Acumatica Framework can be configured to run on multiple application servers behind a load balancer. With this configuration, it is not possible to predict the application server that will receive the next request from the client. In this model, session specific data must be shared between the application servers. The cost of serialization and the amount of data that need to be shared between application servers is often the main challenge to scaling complex business applications horizontally.

Acumatica Framework implements the following techniques to address issues related to session-state management without sacrificing performance, fault tolerance, or scalability:

- Objects on the application server are created on each request and disposed after the request execution. The applicaiton state is preserved in the session through the serialization mechanism.

- Data serialized into the session is minimized[1] to store only modified data (inserted, deleted, or modified records). The rest of the data is extracted from the database on demand[2] and built around the session data.
- A custom serialization mechanism is implemented to serialize only relevant data and reduce the amount of service information.[3]
- Hash tables, constraints, relations, and indexes concerned with the execution of business logic are created strictly on demand. This technique allows the user to avoid execution of these operations on each request if not needed.[4]

[1]Serialization and retrieval times are directly proportional to the size of the serialized data.

[2]A custom algorithm that extracts only the data required for the current request execution from the database is implemented.

[3]The standard serialization mechanism implemented in the Microsoft .NET platform is generic and cannot be optimized when used for a specific task.

[4]Creation of indexes, constrains, hash tables, and relations consumes a significant amount of CPU and runtime memory.

High Application Density

An application created with Acumatica Framework provides an excellent per-user density. In general, a web-based application provides a better per-user density compared with traditional applications deployed through Microsoft Remote Desktop, Citrix, or Virtual Desktop Infrastructure technologies. This is because of lower memory consumption and extensive pooling of shared resources. The use of AJAX technology in Acumatica Framework allows the user to achieve an even better application density[1] compared with standard web-based applications. Two factors take place here:

- Expensive HTML rendering operations are performed only once: on the initial page load. All subsequent requests to the same page do not trigger HTML rendering, which reduces the load on the application server.
- Exchange with only modified data between client and server reduces network traffic.

[1]It should be pointed out that because of the rich GUI functionality a user can generate more requests to the server within the same period of time when compared to traditional web-based applications. This may result in a higher server load generated by a single user within the same time period. But, at the same time, the rich GUI allows the user to execute the same job faster compared to traditional web applications, providing better user experience. Overall the number of transactions per second that could be handled in an AJAX model on the same hardware is higher.

Designed for Datacenter Deployment

Combination of the following factors makes applications created with Acumatica Framework perfect for deployment in datacenters:

- Build-in support for deployment of single instance of application on multiple application servers behind a load balancer. This mean that highly reliable and scalable configurations can be supported.
- An excellent per-user density. This means lower investments into hardware infrastructure.
- Web-based and accessible through HTTP and/or HTTPS protocol, a set of technologies to minimize network traffic. This means simple network configuration and lower requirements for network bandwidth.
- Zero footprint on client computers. This means simple upgrade and update management and lower maintenance costs.
- All the benefits of underlying Microsoft.NET technology in regards to datacenter deployment.

Ability to Scale Up or Down

Scaling an application down is as important as scaling an application up. With minimum deployment an application created with Acumatica Framework can be installed on a single desktop or notebook computer in both production or development environment. With a single code base an application can be scaled up or down.

Built in Multi-tenancy Support

With the development of microprocessor technologies and increasing computing power it becomes possible to host multiple tenants on a single application server. This approach can be referred to as multi-tenancy. The multi-tenant approach allows for the best application density and hardware utilization. In addition, the use of a multi-tenant approach opens the questions related to tenants isolation and quality of the services monitored.

Acumatica Framework has a build-in multi-tenancy architecture and applications created with Acumatica Framework can be configured to operate in multi-tenant mode. Acumatica Framework supports both the execution of a single application instance that hosts multiple tenants and the execution of an individual application instance for each tenant. The following items are addressed on the platform level:

- Isolation of custom code that is submitted by tenants as customization and the quality of service for each of these tenants[1] are addressed by starting the application in a different application domain[2].
- Tenants database isolation is implemented by providing a single tenant identifying field in all database structures[1]. This mechanism is generic, the name and value of the field are linked to the tenant's application domain and are not exposed to application code or logic[3].
- Database isolation can also be achieved by linking the tenant's application domain to the individual tenant database.
- Acumatica Framework provides a set of tools for automated tenant deployment, monitoring of services quality and upgrade management of multi-tenant deployments[4].

Configuring an application, created with Acumatica Framework, to operate in multi-tenant mode creates close to zero overhead compared to running in single tenant mode.

[1]This is a configurable option and can be activated if required.

[2]Application domain is a term specific for Microsoft.NET platform. Please, refer to Microsoft documentation for mode detailed explanation.

[3]This is important, because if the multi-tenancy isolation mechanism is exposed to application logic it becomes vulnerable to mistakes made by application programmers.

[4]These tools are not a part of standard Acumatica Framework and must be purchased separately.

Acumatica Framework and Microsoft Technology

Acumatica Framework is built on top of Microsoft.NET and Microsoft Visual Studio IDE technologies. This choice makes it easy for an application developer who is familiar with Microsoft.NET technology to learn Acumatica Framework and start application development. Also, the use of Microsoft Visual Studio IDE provides an efficient and productive environment for programmers. This section explains the use of Microsoft technologies in Acumatica Framework.

Acumatica Framework and Microsoft.NET Technology

Acumatica Framework is designed and created on top of Microsoft.NET technology. It is written using C# programming language as a managed code. Acumatica Framework extensively uses core services and components of Microsoft.NET technology such as:

- CLR and JIT compilation

- Thread and memory management
- Session Management
- Build Providers
- SOAP Implementation
- C# Programming Language
- Generics and Attributes
- Code Reflection
- Dynamic Methods
- Web Site Code Compiler
- Code Security
- Application Domain model

Additionally, Acumatica Framework does not rely on or use the high level components, primitives or application building blocks provided with Microsoft.NET. Instead, it implements its own stack of primitives and components on top of core Microsoft.NET technologies. This stack includes:

- Application programing API and application event model
- Database access layer and support of multiple database access engines
- Transaction management and thread pooling
- Serialization, searching and indexing primitives
- Caching
- SOAP proxy builder
- Membership and access providers
- Site management
- Localization
- Audit tools
- Help system
- Session splitter
- Web controls

Microsoft.NET technology was selected as a foundation for Acumatica Framework because:

- It fits Acumatica Framework runtime performance and scalability requirements
- It provides all the features and technologies required for Acumatica Framework design
- It provides a complete set of high quality services, components and primitives required to build Acumatica Framework
- Wide acceptance of the technology and programmers familiarity of Microsoft.NET platform
- Microsoft Visual Studio IDE environment
- Support and maintenance from industry leader

The reasons for implementing its own stack of primitives, components, and building blocks instead of one supplied with Microsoft.NET platform are:

- Implementation of functionality that is specific for Acumatica Framework

- Optimization of components and primitives to meet performance requirements of Acumatica Framework
- Elimination of wrappers and additional code layers related to modification of generic components behavior for Acumatica Framework requirements
- Independence from software vendor on possible components and primitives modification[1]

[1]Core features and services of Microsoft.NET platform that are used as a base for Acumatica Framework are stable, reliable and not subjected to significant changes from the vendor. At the same time, high level components, primitives, and services are less generic and subjected to significant functionality and code changes.

Acumatica Framework and Microsoft Visual Studio IDE

The Acumatica Framework development environment is implemented as a set of extensions to Microsoft Visual Studio IDE. These extensions include:

- Template project for Microsoft Visual Studio
- Master pages and a set of Page Templates to create typical application screens
- Web controls integrated with Visual Web Designer
- Wizards for creating data access, business logic, and presentation layers
- Design time libraries and components of Acumatica Framework

The choice of Microsoft Visual Studio IDE is quite natural considering the use of Microsoft.NET technology.

Acumatica Report Designer is implemented as a standalone WinForms application and does not utilize Microsoft Visual Studio IDE.

Acumatica Framework and External Components

Acumatica Framework does not rely, use, or depend on any external non-Microsoft tools or components. This is a principal decision, chosen for the following reasons:

- All Acumatica Framework components are designed to be integrated to provide the best performance and development experience. The use of external components significantly restricts this integrated design.
- Acumatica Framework does not contain any unmanaged code and extensively uses the code security model provided by Microsoft.NET. Most of the external components do not use the same standards.
- Use of external components raises the question of functionality and security issues and at the same time triggers compatibility issues on components, updates, and upgrades.
- Use of external components also increases the cost of software through licensing and royalty fees.

In fact, it is the same set of reasons why the use of Microsoft.NET technology is limited to core services and components.

However, Acumatica Framework does not restrict the use of external components if the developer needs them.

Acumatica Framework and Microsoft Azure

Applications developed with the Acumatica Framework are easily hosted with Microsoft Azure for the following reasons:

- Hosting at Microsoft Azure out of the box with one code base

- Full support of Microsoft SQL Azure
- Unique load-balancing proxy for effective multi-server deployment

Acumatica Framework Components

This section provides an overview of the Acumatica Framework component structure.

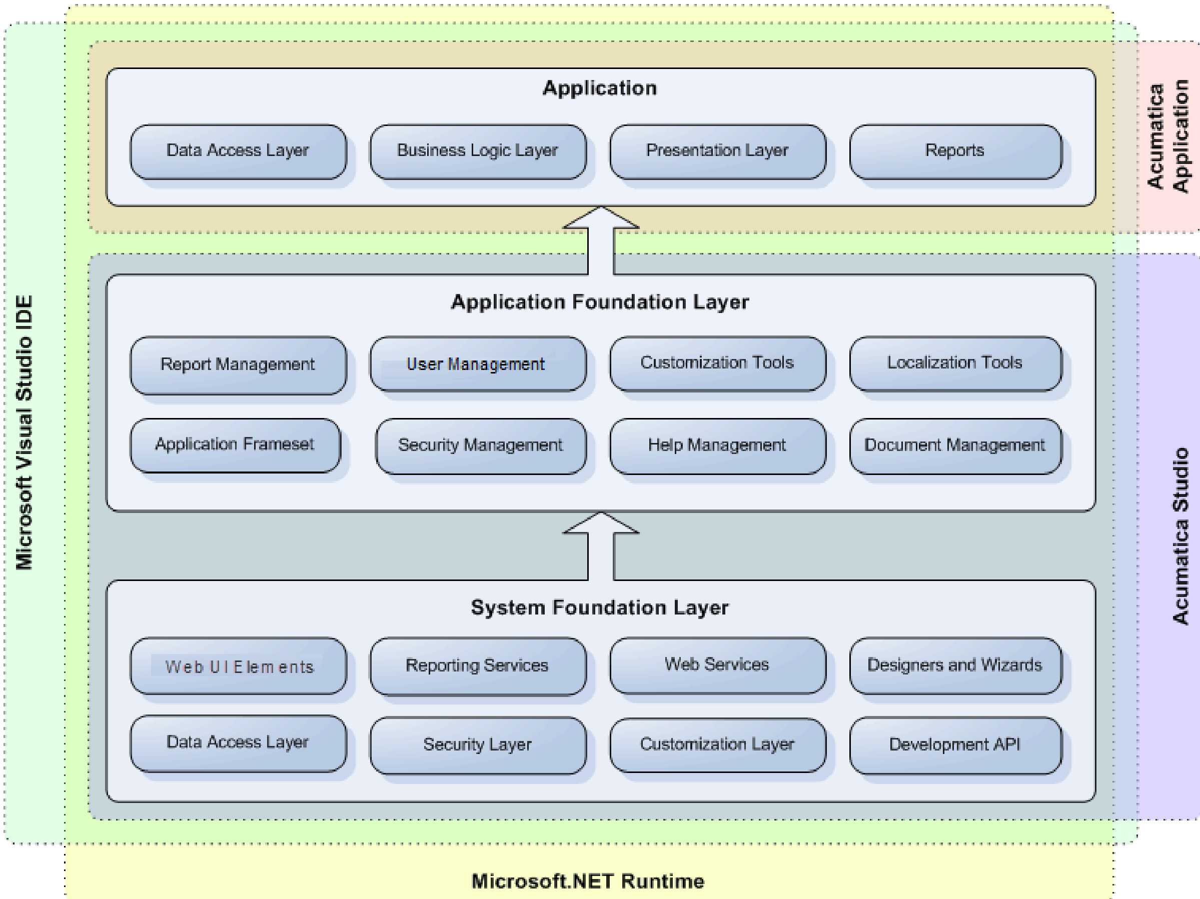


Figure: Acumatica Framework components

Acumatica Framework consists of the *System Foundation Layer* that provides core platform services and the *Application Foundation Layer* that provides a template application and a set of application building blocks.

System Foundation Layer

System Foundation Layer is a set of core components and primitives with functionality required to develop and run an Acumatica Framework-based application.

The primary reasons behind the inclusion of the system foundation layer are:

- Isolate application programmer from complexities related to coding of a web application and from direct use of HTML, CSS, HTTP, and JavaScript.
- Provide the application programmer with a development environment where all pieces of the GUI, business logic, and database access are programmed with the same language and technology.
- Provide the application programmer with development API and methodology to create an application.

- Provide transparent to application programmer runtime services to handle application security, customization, localization, and personalization.
- Provide a set of high level tools and utilities to speed up and automate the creation of business and GUI components and at the same time enforce application integrity.

The *System Foundation Layer* consists of the following main components:

- *Data Access Layer* - set of components responsible for database access, data manipulations, and data persistence management.
- *Security Layer* - set of components responsible for user authorization, access rights verification, and audit on data access and business logic levels.
- *Customization Layer* - set of components responsible for providing runtime customization features on the GUI, database access, and business logic layers.
- *Development API* - set of templates and API for implementing application business logic.
- *Web Controls* - set of web controls implementing access to business logic through the Web GUI interface.
- *Web Services* - the component that provides access to application business logic through the generic Web Service API.
- *Reporting Services - Acumatica Report Designer* and components responsible for runtime report execution.
- *Designers and Wizards* - set of components to automate creation of the application data access classes from the database tables and the GUI (Web Forms) during application development.

Application Foundation Layer

Application Foundation Layer is a set of application building blocks and database structures implemented on top of the system foundation layer components. It provides the application programmer with ready to use components and framework for creating and extending Acumatica Framework-based applications. By using the *System Foundation Layer* components, the programmer will be able to focus on implementing the application business logic and then plug it into the template application, delivering it to the end user as a full functioning business application.

The application foundation layer consists of the following components:

- *Application Frameset*, also referred to as the template application, - application and database structures providing frameset, layout, and navigation services.
- *User Management System* - set of components and database structures for managing users and storing users personal settings and user preferences.
- *Security Management* - set of components and database structures for managing application security, application access policies, and security audit.
- *Help Management System* - the integrated Wiki-based help content editing, management, and search system.
- *Document Management System* - the integrated document storage and management system.
- *Report Management System* - set of tools, components, and database structures that allow registration, listing, and execution of reports created with the Acumatica Report Designer.
- *Customization Tools* - set of tools, components, and database structures for creating, storing, and applying the customization of the standard application on the representation, functional, and database levels.
- *Localization Tools* - the component that allows localization of the application to the different languages.

Application Layer

An application written with Acumatica Framework has the n-tier architecture with a clear separation of the presentation, business, and data access layers. All these layers are implemented by application programmers on top of *System Foundation Layer* and *Application Foundation Layer*.

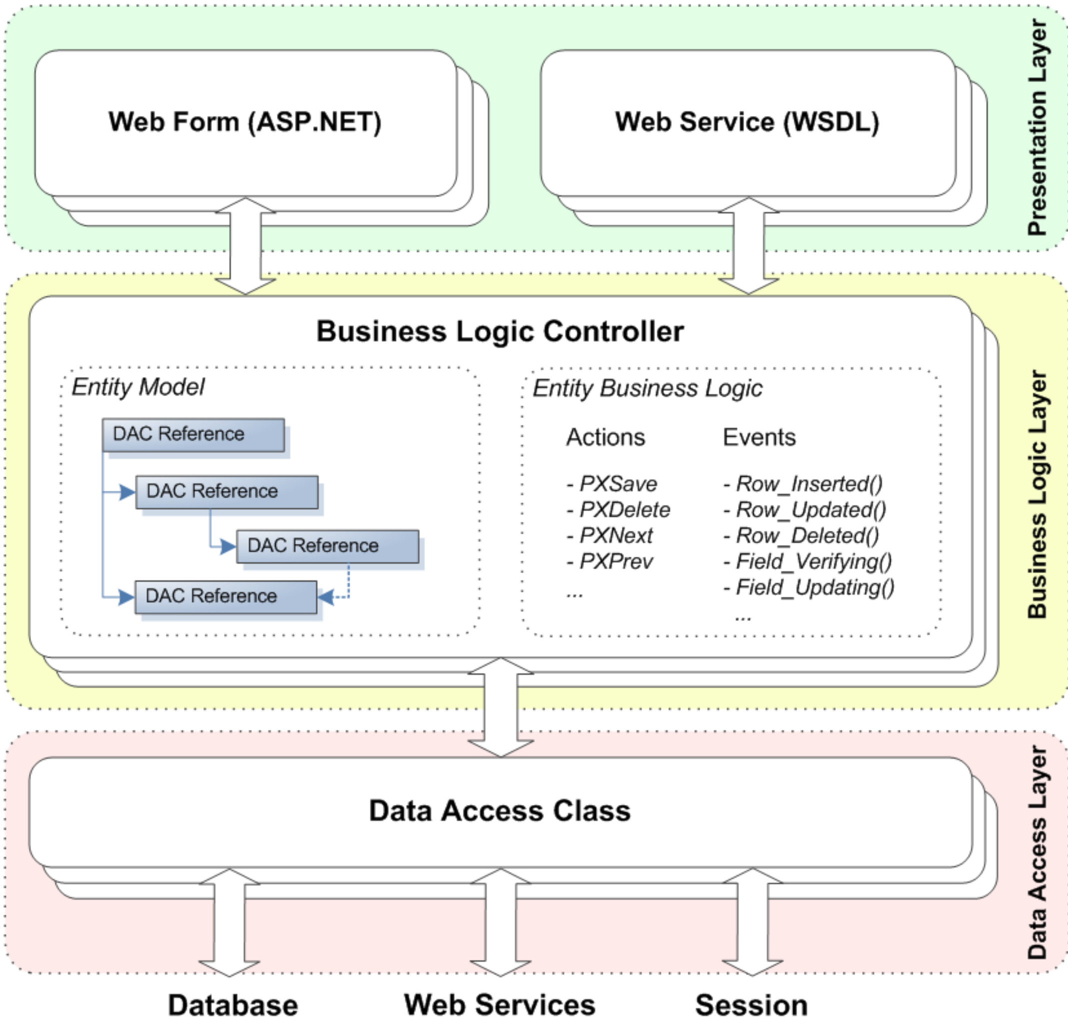


Figure: Application architecture

The picture above illustrates the application component model from the point of view of the application programmer.

Data Access Layer

Data Access Layer is implemented as a set of *data access classes* which wrap data from database tables or data received through other external sources. A data access class associated with a database table may be generated with the help of the *Data Access Class Generator* wizard, which reads database meta data and allows the application developer to select a table and specify columns that should be reflected in the data access class.

Instances of data access classes are maintained by the *Business Logic Layer*. Between request they are stored in the session through a custom optimized serialization mechanism.

Business Logic Layer

The business logic is implemented though the *business logic controller*. These objects are classes derived by the application programmer from the special API class and tied to one or more *data access classes*.

Each business logic controller consists conceptually of two parts: (i) *Object Model*, which includes the required *data access classes*, their relationships, and other meta information, and (ii) *Business Logic* section, which implements the business logic. Each business logic controller could be accessed from *Presentation Layer* or from the application code that is implemented within another business logic controller.

When the business logic controller receives an execution request, it extracts data required for request execution from the data access classes included in the *Object Model*, triggers business logic execution, returns its result to the requesting party, and updates data access classes instances with modified data.

Presentation Layer

Presentation Layer is responsible for providing access to the application business logic through the GUI. It consists of a set of declarative *Web Forms* bound to particular *business logic controllers*. Web Forms are created by the application developer from the templates provided with Acumatica Framework and customized with the help of the *Layout Editor* wizard, which utilizes meta data information extracted from the business logic controller.

When the user requests a new web page, the Presentation Layer is responsible for processing this request. Web Forms are used for generating static HTML page content and providing additional service information required for dynamic configuration of the *Web Controls*. When the user receives the requested page and starts browsing or entering data, the Presentation Layer is responsible for handling asynchronous HTTP requests. During processing, the Presentation Layer submits a request to the Business Logic Layer for execution. Once execution is completed, it analyzes any changes in the business logic container state and generates the response that is sent back to the browser as an XML document.

Business logic can also be accessed through the generic *Web Services* that are part of the Presentation Layer as well. Web Services provide an alternative interface to the application business logic associated with a particular Web Form. From the point of view of the related business logic controller, request from the Web Form and the Web Service are identical and, thus, cause execution of exactly the same business logic. Unlike Web Forms, Web Services are generic and automatically generated by the Acumatica Framework runtime component, based on meta data information extracted from the business logic container and the Web Form.

The Presentation Layer also includes *reports* created with the Acumatica Report Designer. At runtime, reports are loaded and executed through *Reporting Services*, which interface with the Presentation Layer through the special, predefined, *business logic controller* included in the *Application Foundation Layer*.

Runtime Tools

The previous section explained the ability of Acumatica Framework to deliver a set of core services and tools that are important for building and deploying large business applications. All these tools and services are generic and transparent to the application developer. This means that the application developer should not worry about implementing them during the design or application programming stages. In this section, the tools and services used at run time are explained in more detail.

Role-Based Security

Applications created with Acumatica Framework automatically implement role-based security. Access rights can be assigned to:

- A group of screens and reports that have similar logic and are listed under the same namespace
- A screen or report
- Fields used in a particular screen or report
- Methods that can be executed from a particular screen or report

The following access rights can be granted:

- Namespace: Denied, View Only, Granted
- Screen or report: Denied, View Only, Edit, Insert, Delete, Undefined (inherited from the namespace level)
- Field: Denied, View Only, Edit, Undefined (inherited from the screen level)
- Method: Denied, Granted, Undefined (inherited from the screen level)

Assess rights are implemented on the Business Logic Level. Access rights are validated each time the business logic is accessed through both GUI or Web Services.

Personalization

Applications created with Acumatica Framework can be personalized by the user through:

- Adding any application screen or report to the favorites folder
- Saving widgets of an application screen to the personal dashboard
- Preserving the sequence, width, and set of visible columns for grids in any application screen
- Preserving personal filtering settings for any grid and lookup window in any application screen
- Configuring personal export and regional settings

Localization

Applications created with Acumatica Framework can be localized on the presentation, business logic, and database level owing to:

- Standard Microsoft.NET localization mechanism is implemented for localizing the presentation layer.
- All messages returned from the business logic layer can be localized through the dictionary mechanism.
- The runtime environment of Acumatica Framework supports the Unicode standard to store and operate with data in a non-ANSI format.
- Information like addresses or product descriptions can be stored in special, language-specific, database fields and presented in the user selected language.

Acumatica Framework also provides a built-in utility that enables localization of the product by the end user. Once localization is entered and applied, the applicaiton does not require any recompilation or re-installation. Also, localization can also be exported, imported, and merged.

Customization for End Customers

An important feature of Acumatica Framework is the built-in support for end-customer customization, which allows modification of all application layers without recompilation and re-installation of the application and includes:

- Customization of the *Presentation Layer* through:
    - Removing or disabling controls from any application form

    - Changing the form layout by moving controls and changing the tab order of controls
    - Adding new bounded and unbounded controls to any application form
    - Modifying lookup logic by adding more fields to the lookup windows or even by completely replacing the lookup logic
- Customization of the *Data Access Layer* through an extension of the database scheme with new user defined fields
- Customization of the *Business Logic Layer* by submitting a custom application code to the application server

Customization is stored separately from the core application code as meta data. Customization can be modified, exported, or imported. Because customization is stored separately, it is preserved with updates and upgrades of the core application.

Customization for Serial Solutions

Acumatica Extensibility Framework is a part of Acumatica Framework customization plaform that enforces development of third party solutions for multiple customers. Acumatica Extensibility Framework is the key instrument for independent software vendors (ISVs), owing to the following features:

- Customization of the *Data Access Layer* through an extension of the database scheme with new user-defined fields or new user-defined tables that are logical extensions of existing tables
- Customization of *Business Logic Layer* through extension classes built into a separate assembly
- Support for multiple interdependent extensions of both the *Data Access Layer* and *Business Logic Layer* on a single instance of the end-customer application

Generic Web Service API

Applications created with Acumatica Framework expose a generic Web Service application programming interface (API). The API is based on SOAP and WSDL standards and provides programmable access to the same application logic. It is a fast, reliable, and convenient way to perform such operations as:

- Data migration and data import
- Data query and extraction of information for reporting
- Application integration with the external systems
- Execution of long running operations
- Administrative tasks

Each operation made in the API is executed through the same business logic as in the GUI. This ensures functionality and database integrity of the application, regardless of the way it was accessed.

Access to the business logic layer through the API is controlled by the same security mechanism that controls access to the business logic layer through the GUI. In order to perform the API operations, the user must be authorized on the application server and must be granted the appropriate access rights.

The Web Service API is dynamically generated from the application data access and business logic layers and customized metadata. Meaning that if any customization of the data access layer or the business logic layer is made, it will be reflected with the Web Service API as well.

Development Tools

Providing the development tools and the methodology that make a modern web application a commodity is one of the main objectives of Acumatica Framework. This section gives an overview of such development tools and methodologies provided by Acumatica Framework to the application developer and explains on examples of how this increases product quality and the application programmer's productivity.

Visual Web Designer Support

The Acumatica Framework Integrated Development Environment (IDE) is built on top of the Microsoft Visual Studio product. However, it implements its own set of web controls to generate an advanced GUI in a web browser.

The creation of a consistent, professional, and appealingly looking GUI is a complicated task, and special attention was paid in Acumatica Framework to GUI development. All of Acumatica Framework's *Web Controls* have the same rendering and similar appearance in design mode in the IDE and runtime mode in a web browser. This allows the developer to utilize all the facilities of the Visual Web Designer component of Visual Studio. The application developer can use the convenient drag-and-drop mechanism to create an application form layout, to perform form visual editing, and to set control's properties and behavior through an intuitive graphical interface. This approach does not require any knowledge of HTML or Java Script, yet allows the developer to create a professional and appealing web GUI.

The example below illustrates design versus runtime rendering.

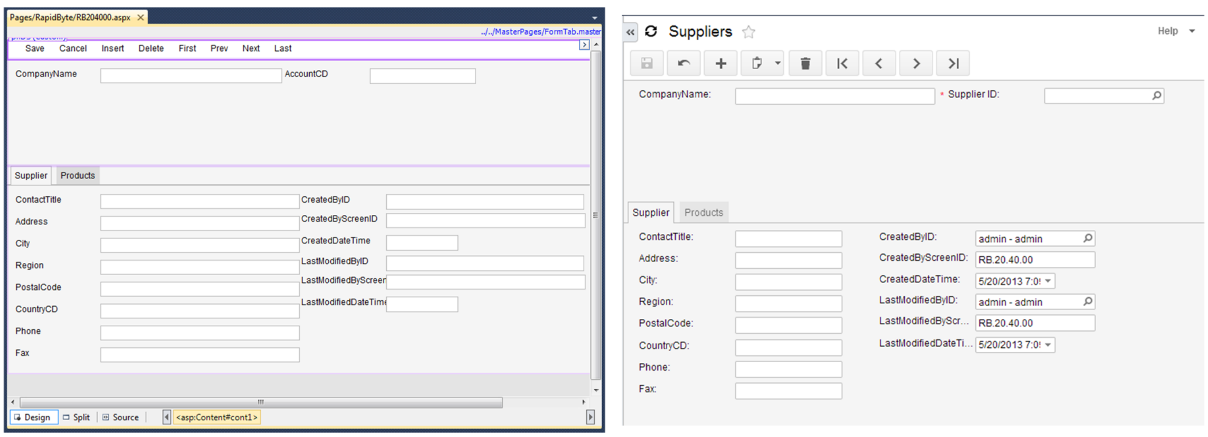


Figure: Web Form in design mode (left) vs. Web Form in runtime mode (right)

Convenient Programming API

In Acumatica Framework, the application programmer is provided with a convenient, event-driven programming API, traditional in rich GUI applications. This model covers database access, business logic, GUI behavior, and error handling. All coding is done with a single language: C#.

This piece of code is written to update the receipt total once one of its related transactions is updated and gives an example of the business logic implemented in the business logic controller:

```
public virtual void DocTransation_RowUpdated(PXCache cache,
                                             PXRowUpdatedEventArgs e)
{
    DocTransaction old = e.OldRow as DocTransaction;
    DocTransaction trn = e.Row as DocTransaction;
    if ((trn != null) && (trn.TranQty != old.TranQty ||
                              trn.UnitPrice != old.UnitPrice))
    {
        Document doc = Receipts.Current;
        if (doc != null)
        {
            doc.TotalAmt -= old.TranQty * old.UnitPrice;
            doc.TotalAmt += trn.TranQty * trn.UnitPrice;
            Receipts.Update(doc);
```

```
            }
        }
    }
```

This code's execution will result in the following behavior:

1. The user selects document transaction in the grid and updates its fields.
2. To complete the row editing, the user presses *Ctrl* + *Enter* on the keyboard. This triggers an event and execution of the code above resulting in update of the receipt total (see the figure below).

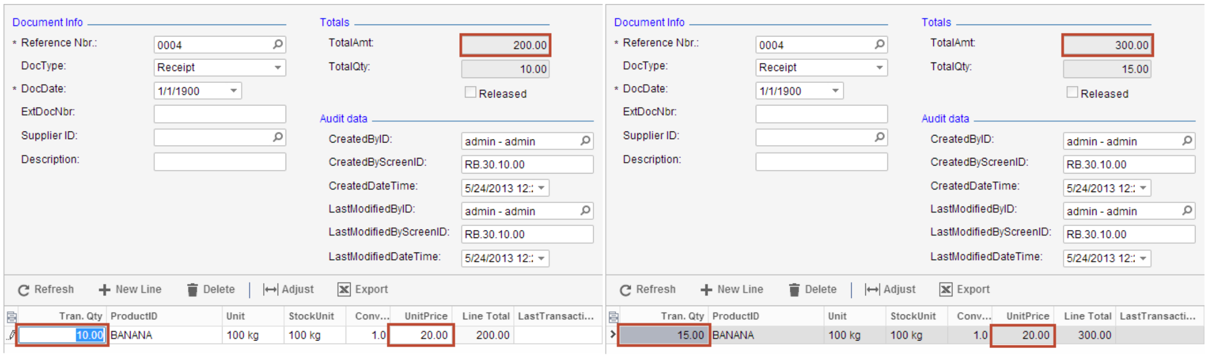


Figure: Example of document transaction details update

BQL and Multiple Database Engine Support

With Acumatica Framework, the application programmer is restricted from direct database access and from writing SQL queries. Database specifics are hidden for the application behind *data access classes*, and the SQL queries are constructed declaratively through *Business Query Language (BQL)*. Through a set of generic classes, the BQL library provides rich syntax for building equivalents of SQL queries of any complexity. Unlike SQL statements, BQL statements operate with data access classes, rather than database tables, and provide compatibility between different database engines. The BQL library supports MS SQL and MySQL database engines as well as access to the database through the ODBC provider.

You can see an example of building BQL queries in the application code below, where BQL queries are declared using generic *PXSelect* and *PXSelectOrderBy* classes and execution of the queries is triggered by invoking static `Select()` methods of these classes:

```
private IEnumerable accInqRecords()
{
    int? ledgerid = ((AccountHistoryFilter)AccInqFilter.Current).Ledger;
    string periodnbr = ((AccountHistoryFilter)AccInqFilter.Current).PeriodNbr;
    if (ledgerid == null && periodnbr == null)
        yield break;

    List<string> fperiods = new List<string>();
    if (periodnbr != null)
    {
        foreach (FiscalPeriod fp in
                 PXSelect<FiscalPeriod,
                          Where<FiscalPeriod.periodNbr,
                                Equal<Current<FiscalPeriod.periodNbr>>>>.
                    Select(this))
        {
            fperiods.Add(fp.FiscalPeriodID);
        }
    }

    foreach (PXResult<AccountHistory, Account> res in
             PXSelectOrderBy<
```

```
                 AccountHistory,
                 LeftJoin<Account,
                          On<AccountHistory.accountID,
                             Equal<Account.accountID>>>,
                 OrderBy<Asc<Account.accountCD>>>.Select(this))
    {
        AccountHistory ah = res;
        if ((ah.LedgerID == ledgerid || ledgerid == null) &&
            (fperiods.Contains(ah.FiscalPeriod) || periodnbr == null))
        {
            yield return res;
        }
    }
}
```

Besides creating abstraction from the database specifics, the BQL library also provides the following benefits to the application programmer:

- Compile time statements verification
- Dynamic query building
- Prevention of SQL infusion
- Intellisense support
- Implemented methods for select, insert, update, and delete
- Intelligent requests execution.

A repeated request does not result in additional query to the database and returns the data cached in the business logic controller, unless the requested collection was changed in the database. The Business Logic Layer can be configured to identify such situations and automatically load and return the latest version of data from the database.

Code Reuse through Attributes

Take a look at the first example of code above. It implements logic of updating receipt total based on updating a document transaction. Such logic is often common for entire applications, not a single screen. This logic can be generalized by having it moved into an *Attribute* class. The attribute is used to annotate a data field in the data access class. Then it can be reused anywhere in the code, as in the example below:

```
public class DocTransaction : PX.Data.IBqlTable
{
    ...
    #region TotalAmt
    public abstract class totalAmt : PX.Data.IBqlField
    {
    }
    [PXDBDecimal(2)]
    [PXDefault(TypeCode.Decimal, "0.00")]
    [PXUIField(DisplayName = "Line Total", Enabled = false)]
    [DeltaMultiply(typeof(DocTransaction.tranQty), typeof(DocTransaction.unitPrice),
                   typeof(Document.totalAmt))]
    public virtual decimal? ExtPrice { get; set; }
    #endregion
    ...
}
```

In this example, the logic of updating receipt total on updating of the transaction is generalized and implemented inside the *DeltaMultiply* attribute. It will be triggered after each update, delete, or insert operation on the *DocTransaction* data access class instance and will update totals on the receipt level, in the the appropriate *Document* data access class instance.

Acumatica Framework provides a wide range of preprogrammed attributes that can be used for defining data types, database mapping, referential integrity, data format validation, and specifying default values for the field, among other things. For example, the logic shown in the above example can be implemented using the preprogrammed *PXFormula* attribute, which is meant exactly for implementing calculations of data fields:

```
public class DocTransaction : PX.Data.IBqlTable
{
    ...
    #region TotalAmt
    public abstract class totalAmt : PX.Data.IBqlField
    {
    }
    [PXDBDecimal(2)]
    [PXDefault(TypeCode.Decimal, "0.00")]
    [PXUIField(DisplayName = "Line Total", Enabled = false)]
    [PXFormula(typeof(Mult<DocTransaction.tranQty, DocTransaction.unitPrice>),
               typeof(SumCalc<Document.totalAmt>))]
    public virtual decimal? ExtPrice { get; set; }
    #endregion
    ...
}
```

As the *data access classes* are shared within an application, formatting, custom logic, and any constraints implemented in attributes will be reused in each *business logic controller* that utilizes this *data access class* . This technique allows the user to move shared application functionality into attributes and avoid code duplication, while still enforcing application integrity.

Error and Message Handling

Acumatica Framework provides the application programmer with a standard mechanism to handle multiple errors and messages in the application code, which transparently passes these errors and messages to the client. The code below gives an example of handling an error triggered by the business logic on an attempt to add a data record:

```
protected virtual void SupplierProduct_RowInserting(PXCache cache,
                                                    PXRowInsertingEventArgs e)
{
    SupplierProduct product = (SupplierProduct)e.Row;
    if ((product != null) && (product.ProductID != null))
    {
        SupplierProduct record =
            PXSelect<SupplierProduct,
                     Where<SupplierProduct.accountID,
                           Equal<Current<Supplier.accountID>>,
                           And<SupplierProduct.productID,
                               Equal<Required<SupplierProduct.productID>>>>>.
                Select(this, product.ProductID);
        if (record != null)
            throw new PXException("Such supplier's product already exists");
    }
}
```

This code will result in the error indication in the GUI if the user attempts to a add a product that already exists for the given supplier account, as illustrated below.

Figure: Error handling example

Managing Advanced GUI Behavior

By using the API provided by Acumatica Framework, the developer has access to special properties of the Business logic controller. Elements such as: visible, disabled, tab stop, color etc. These properties are mapped to the appropriate properties of Web Controls during data binding. Any change to these properties gets propagated back to the browser during the request execution and is reflected in the user GUI. This piece of code illustrates disabling of controls in case the document is not subjected to modifications because of its state:

```
protected void Document_RowSelected(PXCache sender, PXRowSelectedEventArgs e)
{
    Document doc = e.Row as Document;
    if (doc == null || doc.Released == true)
    {
        PXUIFieldAttribute.SetEnabled(sender, doc, false);
        Receipts.Cache.AllowDelete = false;
        Receipts.Cache.AllowUpdate = false;
        ReceiptTransactions.Cache.AllowDelete = false;
        ReceiptTransactions.Cache.AllowUpdate = false;
        ReceiptTransactions.Cache.AllowInsert = false;
    }
    else
    {
        PXUIFieldAttribute.SetEnabled(sender, doc, true);
        PXUIFieldAttribute.SetEnabled<Document.totalAmt>(sender, doc, false);
        PXUIFieldAttribute.SetEnabled<Document.totalQty>(sender, doc, false);
        Receipts.Cache.AllowDelete = true;
        Receipts.Cache.AllowUpdate = true;
        ReceiptTransactions.Cache.AllowDelete = true;
        ReceiptTransactions.Cache.AllowUpdate = true;
        ReceiptTransactions.Cache.AllowInsert = true;
    }
    PXUIFieldAttribute.SetEnabled<Document.docNbr>(sender, doc, true);
    PXUIFieldAttribute.SetEnabled<Document.docType>(sender, doc, true);
}
```

This code's execution will result in the following behavior on the screen:

1. The user selects a document that is not released and can see that the controls on the form and the grid are available for modification.
2. The user navigates to a released document and can see that the data entry controls become disabled. Also, the user cannot insert or update any data in either the document header or the details (see the figure below).

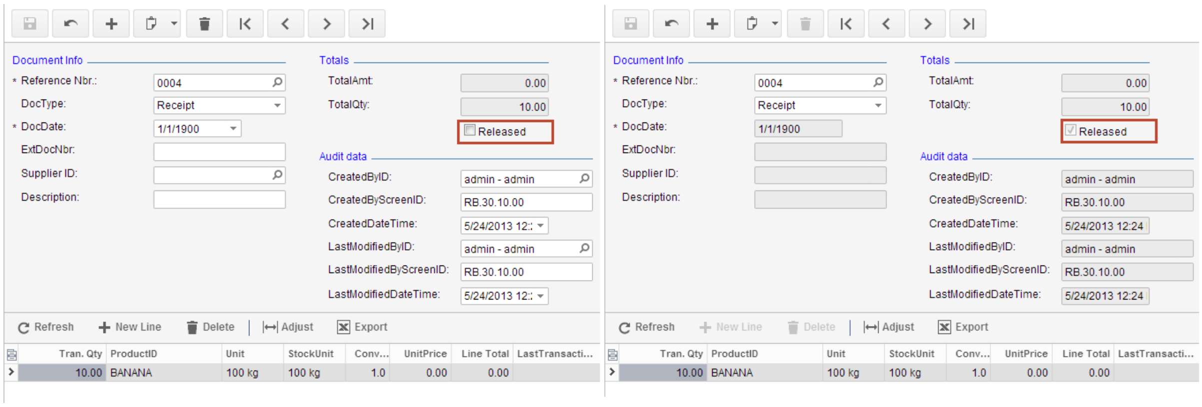


Figure: Example of disabling controls

It is important to mention that changes in the representation logic coded inside the business logic controller are not pushed into the Presentation Layer, but requested by the Presentation Layer if it supports and recognizes this additional information. This technique enables support of an alternative Presentation Layer like Web Services that might not be aware or require such advanced behavior. At the same time, it allows programming of advanced GUI behavior in the same location where the application business logic is coded. This feature is convenient for the programmer, because it reduces the application code base and the possibility of programming mistakes.

Master Pages, Templates, and CSS Support

The Visual Studio project and item templates provide reusable and customizable project and item stubs that accelerate the development process, removing the need to create new projects and items from scratch. Project templates provide the basic files needed for a particular project type, include standard assembly references, and set default project properties and compiler options.

Acumatica Framework distribution includes:

- The project template for the creation of a new application
- A set of page templates that automate the creation of typical page layouts

The master pages mechanism in ASP.NET allows for the creation of an application that looks and feels consistent. Master pages define the standard appearance and behavior that is common in all application pages. The application developer creates individual content pages that refer to the master page. When a content page is requested, it merges with the master page to produce output that combines the layout and base functionality of the master page with the content of the requested page.

Acumatica Framework fully supports the master pages mechanism and provides the application developer with a set of predefined master pages. The application developer can design his own master pages or modify existing ones.

A web application written with Acumatica Framework supports style modification through Cascading Style Sheets (CSS).

The combination of these technologies creates consistent application GUI and behavior.

Application Creation Wizards

Acumatica Framework provides a set of wizards for automating creation of *data access classes* and *Web Forms*. Use of these wizards eliminates the manual job associated with data access class creation and data biding configuration.

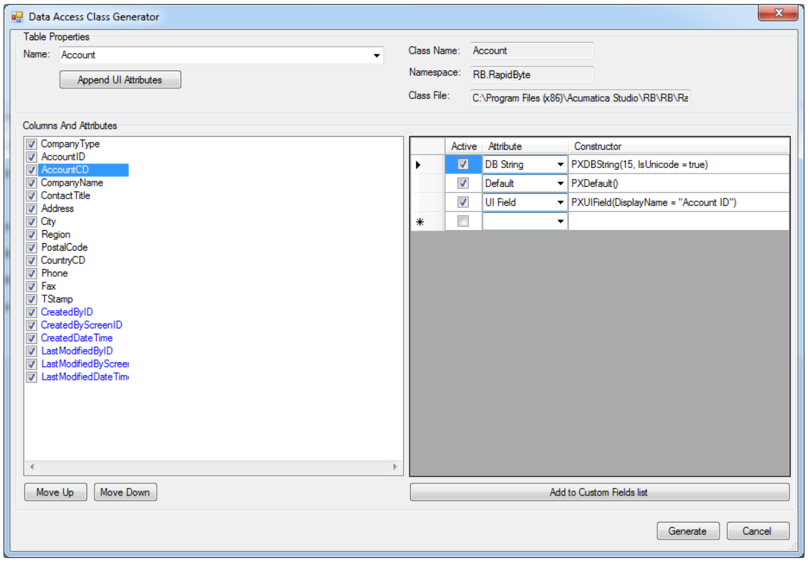


Figure: Data Access Class Generator

The *Data Access Class Generator* wizard provides the application developer with an easy and convenient way to create and modify data access classes. It implements the following functionality:

- Reading data structure from a table, SQL query, or Web Service (referred to as an external data source).
- Creating a data access class based on data structure received from external data source.
- Reading data access class structure from its definition and merging it with data structure received from the external data source.
- Automatical mapping of application-specific attributes based on external data source properties' names.

The *Data Access Class Generator* wizard is a powerful reverse engineering tool, which allows the user to connect to an existing database and extract the information required for building the application *Data Access Layer*.

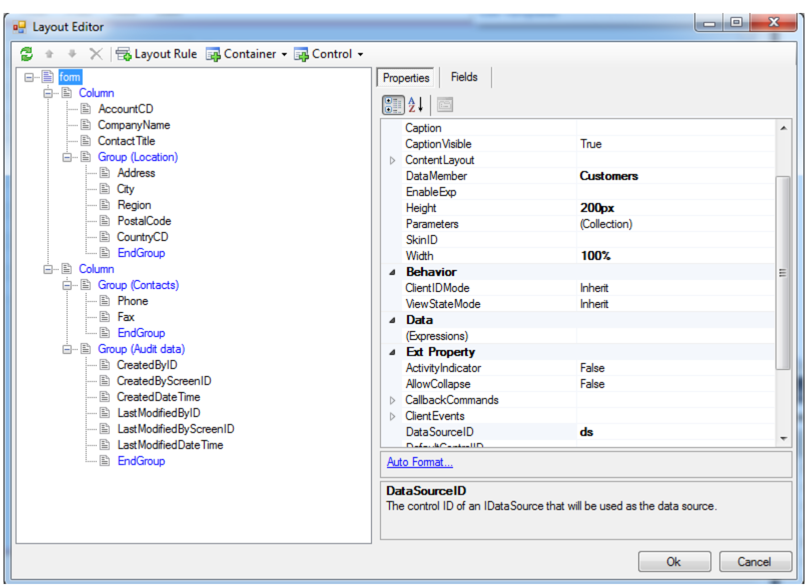


Figure: Web Page Layout Editor

The *Layout Editor* wizard automates creation of new web forms. It uses meta data stored in the business logic controller and data access class to help the application developer create new web forms or to modify existing ones in a fast and efficient manner. The *Layout Editor* wizard implements the following features:

- Reading meta data from the *business logic controller* and the *data access class* and creating a list of controls that could be added to the Web Form.
- Adding controls selected by the programmer to the Web Form.
- Updating Web Form controls with changed *business logic controller* and the *data access class* meta data.

Acumatica Report Designer

Acumatica Framework provides application developers with an integrated report designer.

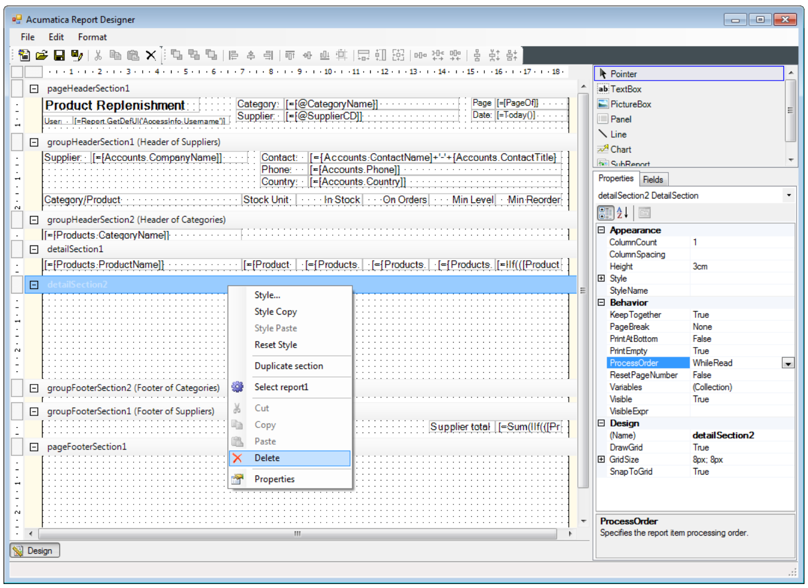


Figure: Acumatica Report Designer

Acumatica Report Designer is implemented as a standalone desktop application. It can be used by both application developers, for developing new reports, and end users, for customizing existing reports.

Acumatica Report Designer is tightly integrated with Acumatica Framework runtime components and provides the following features and services:

- Remote connection to the application server and the ability to browse the application database schema through web services.
- Report's query designer that supports simple selects, sub selects, views, and server-side pre-processing.
- Grouping, sorting, and filtering support.
- Creation of report elements tree with support of drag and drop placement of report elements on the design form.
- Automatic formatting of report elements based on meta data extracted from application database schema.
- Support of basic aggregate expressions and runtime-calculated formulas.
- Support of mass control movement, alignment, editing, and formatting operations with undo functionality.

- Integrated report starting form with report parameters that are dynamically loaded from the report's definition.
- Runtime synchronization of report elements' formatting such as setting masks and decimal values' precision.
- Export to HTML, Excel, and PDF formats.
- Drill down to application forms.

Integrated with *Reporting Services*,Acumatica Report Designer provides a complete reporting solution for the application developer and a complete set of customization tools for the end user.

Conclusion

Acumatica Framework provides the complete suite of components and technologies for developing complex web applications with rich graphical user interface.

Acumatica Framework is generally suitable for creating any kind of application, but the biggest competitive advantage could be achieved on large projects that require the creation of multiple screens, with similar interface and rich business logic functionality, such as:

- Business Support Systems (ERP, CRM, or MRP)
- Large custom solutions implemented by consulting companies
- Custom solutions in large companies implemented by internal development teams

The main advantages Acumatica Framework provides are:

- High speed of application development through the high level of development automation
- Low number of errors in the application code by enforcing code reuse and application integirty
- Simplicity of the platform through a single coding place
- Language and transparency of the platform services to the application developers
- Scalability and high-availability of the created application combined with simple application deployment
- Remote availability of the created application through the common Internet connection
- Rich and consistent GUI

All of this results in:

- Faster time to market
- Lower application development costs
- Lower TCO for customers
- Better user experience and satisfaction

Components and Tools

Acumatica Framework provides a set of development tools and Visual Studio templates for building applications:

- Acumatica Framework Configuration Wizard
- Report Designer
- Acumatica Framework Templates
- Data Access Class Generator
- Layout Editor

Acumatica Framework Configuration Wizard

The wizard helps you to create, deploy and maintain applications built on Acumatica Framework.

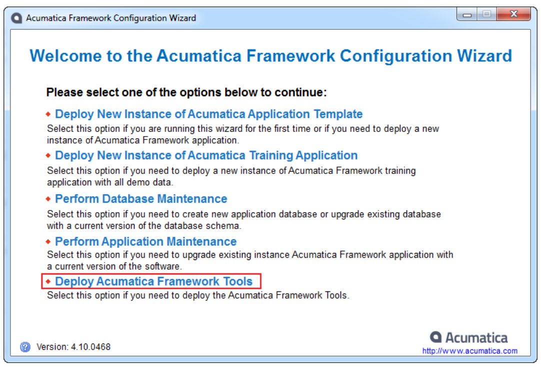


Figure: Acumatica Framework Configuration Wizard

Report Designer

Report Designer is the visual tool for creating report forms and printable pages.

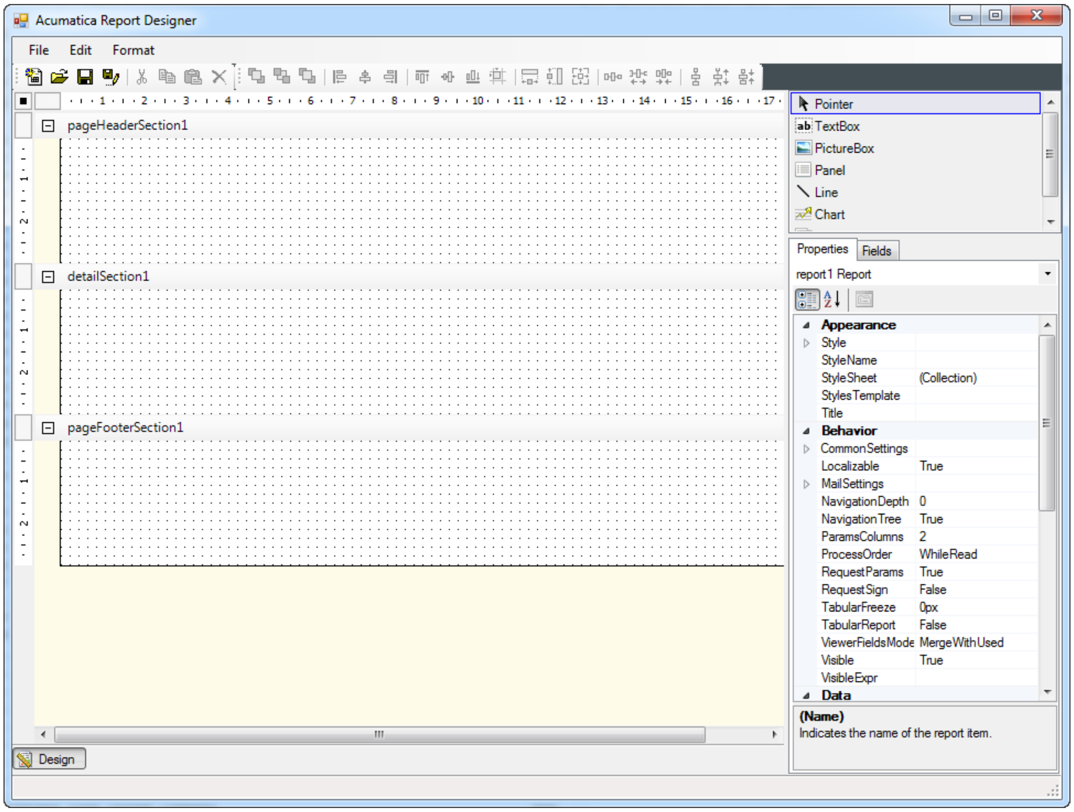


Figure: Report Designer

Acumatica Framework Templates

The Acumatica Framework Templates is a package of default Visual Studio templates that include:

1. PXGraph class template
2. FormDetail page template
3. FormTab page template
4. FormView page template
5. ListView page template
6. TabDetail page template
7. TabView page template

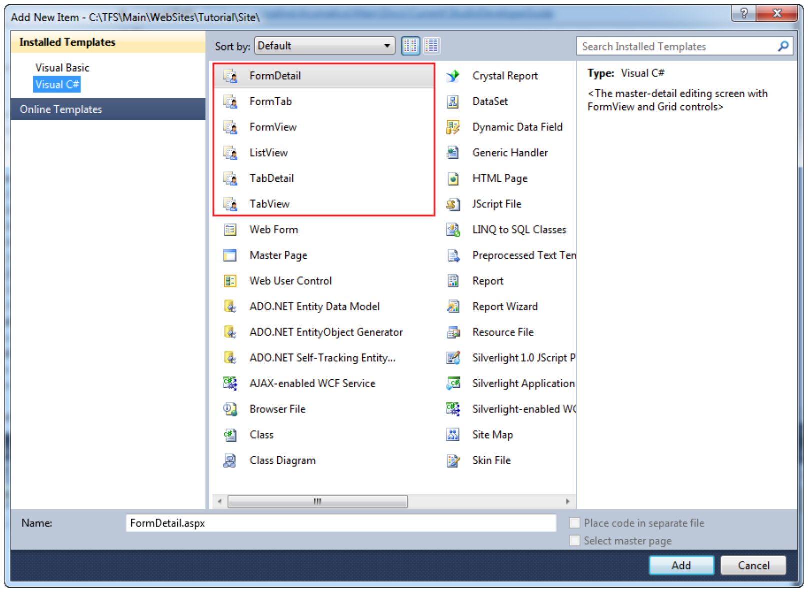


Figure: Acumatica Framework Templates

Data Access Class Generator

The Data Access Class Generator tool is intended for initial automated generation of data access class declaration. The Data Access Class Generator is opened from the data source's smart tag menu from a page opened in the design mode in Visual Studio.

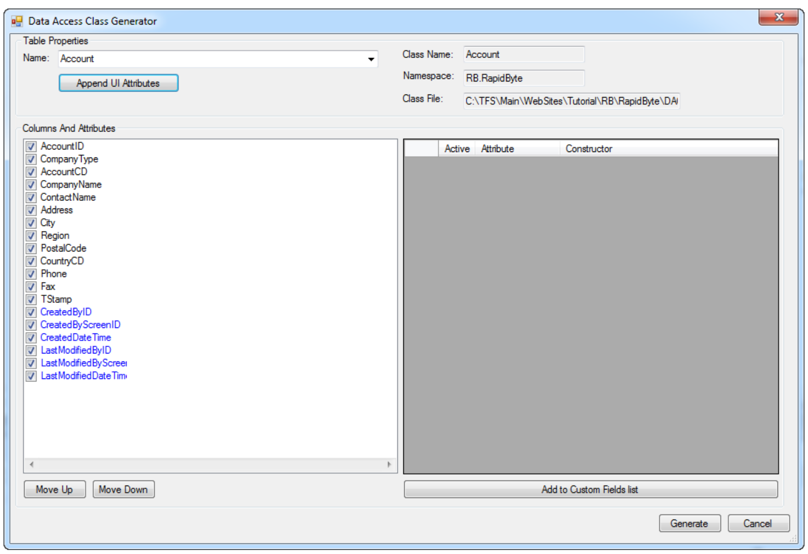


Figure: Data Access Class Generator

Layout Editor

The Layout Editor tool allows you to configure the page layout and adjust positioning of controls on a form.

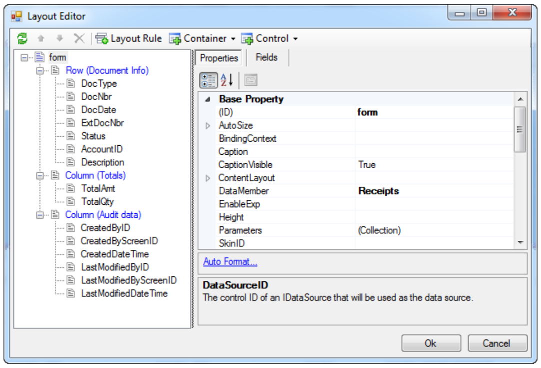


Figure: Layout Editor

Design Guidelines

This section contains the design requirements for the database schema and the application built on Acumatica Framework.

- *Database Design Guidelines*
- *Application Design Guidelines*

Database Design Guidelines

The article covers the following aspects of database design:

- *System and Application Tables*
- *Table and Column Naming Conventions*
- *Typical Columns and Data Types*
- *Primary Key*
- *Foreign Keys and Nullable Columns*
- *Audit Fields*
- *Concurrent Update Control*
- *Support for Attaching Additional Objects to Data Records*
- *Preserving Deleted Records*
- *Multi-Tenancy Support*

System and Application Tables

The database of your Acumatica Framework-based application consists of the following tables:

- System tables: Those that are created by default for the application template and not used to store your application data
- Application tables: Acumatica ERP tables (which exist if you have created an add-on project or implemented customization)
- Application tables: Your own tables

Do not add columns to system tables or modify them in any other way. Such modifications could corrupt the application and would be lost during the next database upgrade. See the *System table list* file for the list of system tables.

Regarding your own application tables, you have to design and create the needed tables that store your application data. You then map these application tables to data access classes (DACs) that define the object model of the application. In one table, you can keep data records of multiple entities, each of which is defined as a separate data access class in the application object model.

Table and Column Naming Conventions

When you are creating a table, you should consider the following suggestions regarding naming conventions:

- Make sure that table and column names are valid C# identifiers, because these names match the names of classes and properties you declare in the application. Do not start a table or column name with a digit.

- Do not use the underscore symbol (_) in table or column names, because it is a reserved symbol in Acumatica Framework. For example, `CompanyType` is a valid column name, while `Company_Type` is invalid.
- Use singular nouns for table names. Typically, a table is mapped to a data access class that represents the entity. For instance, the `SOShipment` table contains data records that represent instances of the SOShipment entity.

  Acumatica Framework generates SQL statements with table and column names in the same letter case as the corresponding data access classes and fields are declared in the application. Also, the DAC Generator tool produces data access class declarations in the same letter case as the tables and columns are defined in the database schema.

- Use two prefixes in table names: a two-letter company name and two-letter application module prefix. For example, the `MCSVAppointment` table can be used in the Services (SV) module for the MyCompany company. These prefixes help to distinguish your application tables from Acumatica ERP tables and tables of other vendors if you create an add-on project or extension library.
- If you add a column to an Acumatica ERP table, start the column name with the *Usr* prefix followed by the two-letter company name. For instance, you could use `UsrMCColumn` for the column of the MyCompany company. In this case, the column will be preserved during upgrades. In your own application tables, there are no strict requirements to start column names with any prefixes.
- Be sure that custom indexes on Acumatica ERP tables start with the *Usr* prefix followed by the two-letter company name, so that the indexes will be preserved during upgrades.

Column Name Suffixes

We recommend that you use the following suffixes in column names:

- *ID* for surrogate keys, including database identity columns—for example, `CustomerID`
- *CD* for natural keys—for example, `CustomerCD`
- *Nbr* for numbering identifiers—for instance, `OrderNbr`
- *Price* for prices, such as `UnitPrice`
- *Cost* for costs—for example, `UnitCost`
- *Amt* for amounts, such as `FreightAmt`
- *Total* for totals, such as `OrderTotal`
- *Qty*, *QtyMin*, and *QtyMax* for quantities—for instance, `OrderQty`
- *Date* for dates, such as `OrderDate`
- *Time* for time points and time spans—for example, `BillableTime`
- *Pct* for percents, such as `DiscountPct`

Typical Columns and Data Types

You should use the following data types for columns. In the **Type Attribute** column in the table below, you can find the most typical type attributes that are added to the corresponding data fields in the data access class declaration.

Typical Data Types

Value	Data Type (SQL Server)	Type Attribute on the Data Field
Database identity	int	`[PXDBIdentity]`

Value	Data Type (SQL Server)	Type Attribute on the Data Field
Natural key (for example, document number)	nvarchar (15)	`[PXDBString(15, IsKey = true, IsUnicode = true)]`
Line number	int	`[PXDBInt]`
Short string (for example, a name or unit of measure)	nvarchar (20), nvarchar (50)	`[PXDBString(20, IsUnicode = true)]`
Long string (such as a description)	nvarchar (255)	`[PXDBString(255, IsUnicode = true)]`
Type or status identifier (for instance, a document type)	int or char (1)	`[PXDBInt]` or `[PXDBString(1, IsFixed = true)]` respectively
Boolean flag (for example, active/inactive)	bit	`[PXDBBool]`
Price or cost, monetary units	decimal (19, 6)	`[PXDBDecimal(6)]`
Amount or total, monetary units	decimal (19, 4)	`[PXDBDecimal(4)]`
Quantity, pieces	decimal (25, 6)	`[PXDBDecimal(6)]`
Maximum, minimum, or threshold quantity, pieces	decimal (9, 6)	`[PXDBDecimal(2)]`
Percent, rate (for example, discount percent)	decimal (9, 6)	`[PXDBDecimal(2)]`
Weight or volume	decimal (25, 6)	`[PXDBDecimal(6)]`
Date	smalldatetime	`[PXDBDate]`
Time span	int	`[PXDBTimeSpan(DisplayMask = "t", InputMask = "t")]`
Coefficient (such as a conversion factor)	decimal (9, 6)	`[PXDBDecimal(1)]`

Primary Key

You have to define the primary key in each application table that you create. The primary key may consist of one column or multiple columns. The primary key must include the *`CompanyID` column* if one is defined in the table.

For each table, you can use one of the following typical primary key variants:

- One key column included in the primary key in the table and set as the key in the data access class
- A pair of columns, with one column included in the primary key in the table and the other one set as the key in the data access class
- Multiple columns that are included in the primary key and set as the compound key in the data access class

In a setup table, the only `CompanyID` column must be included in the primary key.

One Key Column

You may use one key column for rather short dictionaries. For instance, you can use the two-letter country code from ISO 3166 as the key in the `Country` table.

A Pair of Columns With Key Substitution in the UI

If you want to represent a user-friendly key in the user interface (UI) that corresponds to a surrogate key in the database, you may use a pair of columns and the key substitution mechanism provided by Acumatica Framework. You can define two columns in a table, one for the surrogate key (typically the database identity column) and one for the natural key, and set only the surrogate key as primary in the table. In the application object model, you set the key to the only natural key data field. In this case, Acumatica Framework provides the ability to transparently work with different keys at the database and application level. In the UI, users work only with the natural key while the database operates with the surrogate key (see the key substitution scheme below).

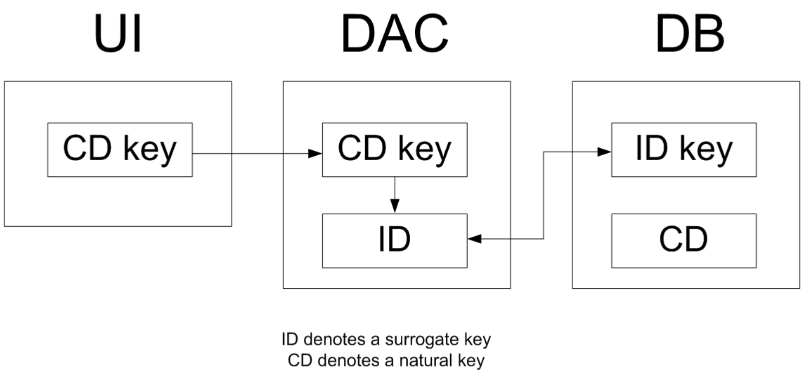


Figure: Key substitution in Acumatica Framework

For instance, you can define two columns in the `Product` table, `ProductID` and `ProductCD`. `ProductID` is the identity column that is the only column included in the primary key of the table. `ProductCD` is the string key of a product instance, which is entered by the user through the UI. The `ProductCD` column isn't included in the primary key and is handled as the unique key column by Acumatica Framework.

Multiple Column Key

The compound key consisting of multiple columns may be used for complex entities. For instance, you can include two columns, `DocType` and `DocNbr`, in the primary key for the `Document` table. In the `DocDetail` table, you may use `DocNbr` and `DocDetailNbr` as the compound primary key. The corresponding data fields should be also set as the key fields in the data access class.

Foreign Keys and Nullable Columns

In the database, you have to define the primary key in each application table that you create. The primary key defines the unique data record identifier, which provides table-level integrity of data.

There are no strict requirements to define column-level constraints and foreign keys in application tables. Whether or not you define the constraints at the database level depends on the design approach you use. At the higher level of the application object model represented by data access classes, you can flexibly define any level of constraints, including default values, nullable fields, and parent-child relationships between data access classes. If you aren't sure whether a column should allow a null value, you can allow null values for it in the database. Later, in the data access class, you can make the data field either required or nullable; you can even make the field required on one page and optional on another.

For boolean and decimal columns, we recommend that you define default values either in the database, or in data access classes. This simplifies the application code by helping to avoid multiple checking of values for nulls.

Audit Fields

Audit fields keep meta information on the creation and last change of a database record. Audit fields are updated automatically by the framework.

To enable tracking of audit data for a particular table, you should add the columns listed below to the table and declare the corresponding audit data fields in the data access class. You have to add the corresponding type attribute to each audit field. If the audit columns are properly created in the database table and the corresponding data fields are declared in the data access class, Acumatica Framework automatically updates audit data in these fields every time a data record is modified from the application. The audit column parameters and DAC attributes are given below.

Audit Columns

Database Column Name	Data Type (SQL Server)	Type Attribute on the Data Field
CreatedByID	uniqueidentifier, not null	`[PXDBCreatedByID]`
CreatedByScreenID	char (8), not null	`[PXDBCreatedByScreenID]`
CreatedDateTime	smalldatetime, not null	`[PXDBCreatedDateTime]`
LastModifiedByID	uniqueidentifier, not null	`[PXDBLastModifiedByID]`
LastModifiedByScreenID	char (8), not null	`[PXDBLastModifiedByScreenID]`
LastModifiedDateTime	smalldatetime, not null	`[PXDBLastModifiedDateTime]`

Concurrent Update Control

You can add the SQL Server timestamp column to a table to make Acumatica Framework able to handle concurrent updates. The corresponding timestamp data field should be declared in the data access class. If the timestamp data field is declared, Acumatica Framework handles the timestamp column automatically. Acumatica Framework checks the row version every time the row is modified. We recommend that you add the timestamp column to all tables of your application (see the table below).

The Timestamp Column

Database Column Name	Data Type (SQL Server)	Type Attribute on the Data Field
TStamp	timestamp, not null	`[PXDBTimestamp]`

Support for Attaching Additional Objects to Data Records

You can attach additional objects to a data record—for instance, add a textual note or upload a file or multiple files to a data record. You enable support for data record attachments for each particular table individually. To enable support for data record attachments, add the column that stores the global data record identifier (typically, `NoteID`) to the table and declare the corresponding field in the data access class. For more information on file upload through an application page, see *Working With Images*. See below for the global identifier column parameters and the attribute that should be added to the corresponding DAC field.

The Global Data Record Identifier Column (NoteID)

Database Column	Data Type (SQL Server)	Type Attribute on the Data Field
Global data record identifier (typically named NoteID)	bigint, null	`[PXNote]`

Preserving Deleted Records

Acumatica Framework provides a low-level mechanism for preserving deleted data records in the database. With this mechanism, when an application initiates deletion of a data record, the data access layer generates the SQL query that marks the data record as deleted but does not permanently remove the data record from the table. On select, the data access layer generates the SQL query that returns only data records that are not marked as deleted. The data records that are preserved in this way can be restored. You can enable the preservation of deleted data records for each table individually. To preserve data records in a particular table, add the `DeletedDatabaseRecord` column to the table and do not declare the data field in the data access class. On deletion of a data record in the table, the framework automatically preserves the deleted data record transparently to the application developer.

The DeletedDatabaseRecord Column

Database Column	Data Type (SQL Server)	Type Attribute on the Data Field
DeletedDatabaseRecord	bit, not null	Not declared in DAC

Multi-Tenancy Support

Multiple companies or *tenants* can work on the same instance of an Acumatica Framework-based application with completely isolated data. The application looks identical to all tenants, but each company has exclusive access to its data only. Data is isolated at the lowest level of the application, in the data access layer that executes SQL queries for the company of the current logged-in user.

Multi-tenancy support is enabled for each particular table individually. To enable multi-tenancy support for a table, add the `CompanyID` column to it and include the column in the primary key (see the column parameters in the table below). The `CompanyID` column is handled automatically by the framework and should not be declared in data access classes. If a table doesn't have the CompanyID column, all data from the table is fully accessible to all companies that exist in the database. For more information, see *Support of Multiple Companies*.

The following scheme illustrates how different logical companies work with the Acumatica Framework-based application in a multi-tenant configuration. They work with the same application but have isolated data access, as if they work with different database instances.

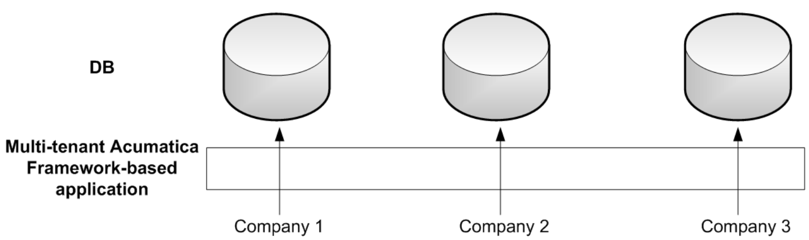


Figure: Multi-tenant Acumatica Framework-based application

The CompanyID Column

Database Column Name	Data Type (SQL Server)	Type Attribute on the Data Field
CompanyID	int, not null, included in primary key	Not declared in DAC

Support for Shared Data Access Between Companies

Acumatica Framework provides shared data access in a multi-tenant configuration. Acumatica Framework supports a hierarchy of logical companies that may work with a combination of shared and individual data. In shared access mode, every company may work with its individual copy of a data record. Copies differ by `CompanyID`. All copies represent the same logical object in the application but different data records in the database. For instance, each company may use individual settings of the application.

Support for shared data access is enabled for each particular table individually. To enable support for shared data access for a table, add the `CompanyMask` column to the table (see the column parameters in the table below). The `CompanyMask` column is handled automatically by the framework and should not be declared in data access classes. If a table doesn't have the CompanyMask column, shared data access is not available for this table.

The scheme below shows a possible multi-tenant configuration with shared data access between Company 1, Company 2, and Company 3. Users of Company 2 have access to the data of all three companies. Users from the other two companies have access to their individual data only. Physically, the data of all three companies is stored in a single database instance.

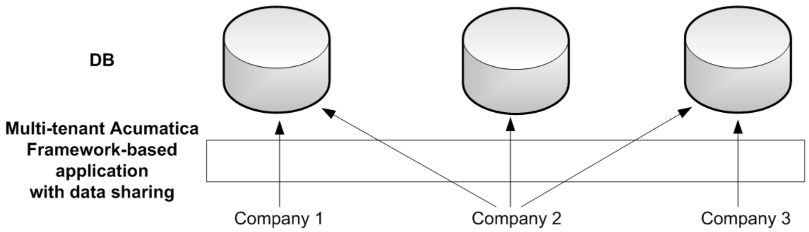


Figure: Shared data access in a multi-tenant Acumatica Framework-based application

The CompanyMask Column

Database Column Name	Data Type (SQL Server)	Type Attribute on the Data Field
CompanyMask	varbinary (32), not null, default 0xAA	Not declared in DAC

Application Design Guidelines

This document summarizes the application design and style conventions used in Acumatica ERP.

Development Environment Options

Acumatica Framework supports Microsoft Visual Studio 2008, 2010, and 2012. See below for additional details about these versions.

For **Microsoft Visual Studio 2008**, you must have Service Pack 1 installed. Also, you must install the following hotfix from Microsoft: *KB967253.*

The following settings are recommended for the MS Visual Studio environment to enforce a uniform webpage appearance:

1. Set the following options under the **Tools** > **Options** > **HTML Designer** > **CSS** section:
    - **Font and text**: *CSS (classes)*
    - **Padding and borders**: *CSS (classes)*
    - **Floating, positioning, and sizing**: *CSS (inline styles)*
    - **Bullets and numbering**: *CSS (classes)*
    - **Background**: *CSS (classes)*
    - **Margins**: *CSS (classes)*
2. Select the following buttons and check boxes under the **Tools** > **Options** > **HTML Designer** > **CSS Styling** section:
    - **Auto Style Application**
    - **Only reuse classes with the prefix "style"**
    - **Use width and height attributes for image instead of CSS**
    - **Use shorthand properties when generating styles**

- **Change positioning to absolute for controls added using Toolbox, paste, or drag and drop**

We also recommended that you modify the following options:

- **View** > **Visual Aids** > **CSS Display:none Elements**: *False* (cleared)
- **View** > **Visual Aids** > **CSS Visibility:hidden Elements**: *False* (cleared)

Captions

Add a caption to the following:

- Each form header
- Each form details grid header

Screen Numbering

When numbering screens in Acumatica ERP, use the following convention:

```
XX.99.99.99
 |  |  |  |_ Sub-Screen Sequential Number
 |  |  |____ Screen Sequential Number
 |  |_______ Screen Type:
 |                    10 - Setup
 |                    20 - Maintenance
 |                    30 - Data Entry
 |                    40 - Inquiry
 |                    50 - Processing
 |                    60 - Reports
 |__________ Two-Letter Module Code
```

Report Numbering

When numbering reports in Acumatica ERP, use the following conventions in addition to those outlined above:

```
XX.6X.99.99
    |
    |____ Report Type:
           61: Review Reports - Reports for document review prior to release
           62: Register Reports - Reports used to print audit information
                                   on processed documents or entities
           63: Balance Reports - Reports reflecting current or historical
                                                  balance information
           64: Forms - Printed webpages
           65: Inquiry Reports - Reports that provides status information
                                  required for operational management
           66: Statistical Reports - Reports that provide statistical or
                                               historical information
```

Application Programming Overview

Acumatica Framework provides the platform and tools for developing cloud business applications. This document explains Acumatica Framework runtime structure, introduces main components, and illustrates their relationships on simple examples.

The chapter is a starting point for application developers who are going to develop and customize applications with the help of Acumatica Framework.

Runtime Structure and Components

An application written with Acumatica Framework has n-tier architecture with a clear separation of the presentation, business, and data access layers. The picture below illustrates the application component model from the point of view of the application programmer.

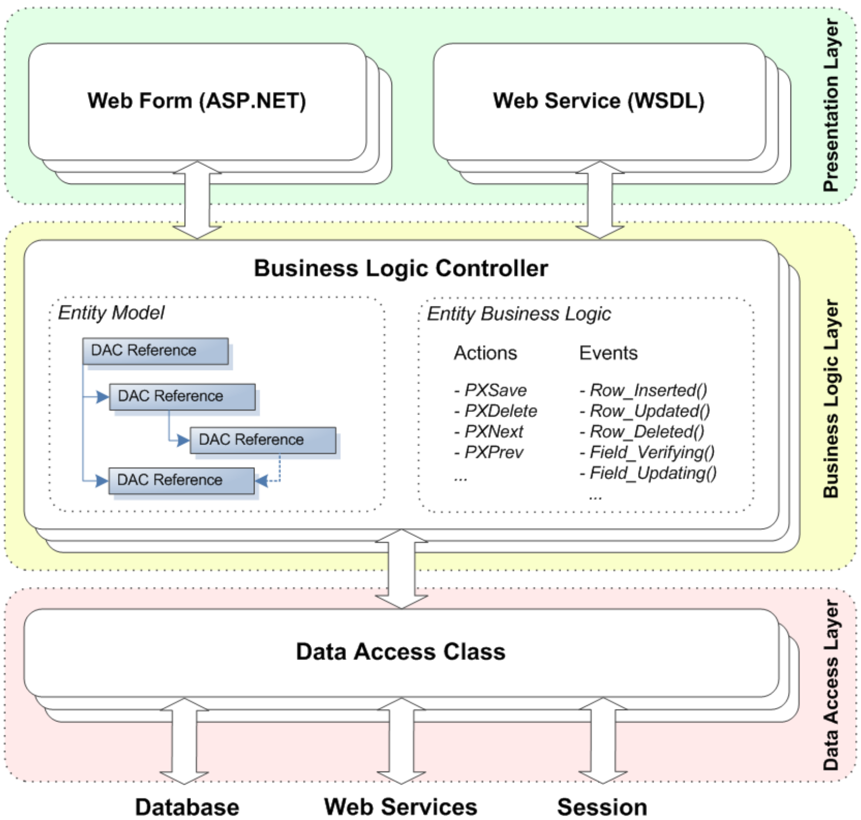


Figure: Application architecture.

Data Access Layer

Acumatica Framework relies on *object relationship mapping (ORM)* technology to access the database from the business logic. Acumatica Framework implements own, proprietary ORM technology. This technology provides an application developer with a set of standard CRUD operations to execute on database tables and methods to execute complex SQL queries.

An important feature of the Acumatica Framework ORM technology is a high-performance serialization mechanism that stores modified but not persisted database records in the session state. Modified data are merged with the result of the query execution to emulate statefull data access behavior for the application developer and minimize the amount of data stored in the session.

Business Logic Layer

Business Logic Layer is implemented as a set of *business logic controllers (graphs)*.

Each business logic controller consists of two parts:

- *Entity Model* that declares *data access classes* the entities are stored in, their relationships, and actions that can be executed over the entities
- *Entity Business Logic* that implements the business logic of the actions and events associated with modifying entity data

Business logic controllers implement the interfaces for *Presentation Layer* to retrieve the entity data and execute the actions over the entity. *Business Logic Layer* relies on *Data Access Layer* to retrieve data from the database and execute CRUD operation.

Presentation Layer

Presentation Layer is responsible for providing:

- The user interface based on the ASPX technology and implemented as a set of declarative *Web Forms*
- The alternative interface for accessing the business logic in the form of auto-generated *Web Service API*

Presentation Layer is completely declarative and contains no business logic.

Querying the Data

This system implements a custom language for writing database queries called BQL (business query language). It is not LINQ and doesn't use it. BQL is written in C# and based on generic classes syntax, but still is very similar to SQL syntax. It has almost the same keywords placed in the order they are used in SQL. For example:

```
PXSelect<Product,
    Where<Product.availQty, IsNotNull,
        And<Product.availQty, Greater<Product.bookedQty>>>>
```

If the database provider is MS SQL Server, the framework will translate this expression into the following SQL query:

```
SELECT * FROM Product
WHERE Product.AvailQty IS NOT NULL
    AND Product.AvailQty > Product.BookedQty
```

BQL gives several benefits to the application developer. It does not depend on database-provider specifics, is object-oriented and extendable. An important benefit is compile-time syntax validation, which helps to prevent SQL syntax errors.

Since BQL is implemented on top of generic classes, you need types that would represent database tables. In the context of Acumatica Framework, they are called *data access classes (DACs)*.

For example, to execute the SQL query from the example above, you should define the `Product` data access class as:

```
using System;
using PX.Data;

// Types used in BQL statements should derive from special interfaces:
// table - IBqlTable, column - IBqlField.
[System.SerializableAttribute()]
public class Product : PX.Data.IBqlTable
```

```
{
    // The type used in BQL statements to reference the ProductID column
    public abstract class productID : PX.Data.IBqlField
    {
    }
    // The property holding ProductID value in a record
    [PXDBIdentity(IsKey = true)]
    public virtual int? ProductID { get; set; }

    // The type used in BQL statements to reference the AvailQty column
    public abstract class availQty : PX.Data.IBqlField
    {
    }
    // The property holding AvailQty value in a record
    [PXDBDecimal(2)]
    public virtual decimal? AvailQty { get; set; }

    // The type used in BQL statements to reference the BookedQty column
    public abstract class bookedQty : PX.Data.IBqlField
    {
    }
    // The property holding BookedQty value in a record
    [PXDBDecimal(2)]
    public virtual decimal? BookedQty { get; set; }
}
```

Each table field is declared in a data access class twice:

- As a *type* to reference a field in the BQL command
- As a *value* to hold the table field data

If the DAC is bound to the database, it must have the same class name as the database table. Fields are bound to the database by means of data mapping attributes (such as `PXDBIdentity` and `PXDBDecimal`), using the same naming convention.

A complete code sample that queries the database is given below:

```
using System;
using System.Collections;
using PX.Data;

public static void Main()
{
    // Select Product records
    PXResultSet<Product> res =
        PXSelect<Product,
            Where<Product.availQty, IsNotNull,
                And<Product.availQty, Greater<Product.bookedQty>>>>
            .Select(new PXGraph());

    // You can iterate through the result set
    foreach(PXResult<Product> rec in res)
    {
        // A record from the result set can be cast to the DAC
        Product p = (Product)rec;
        Console.WriteLine("ID: {0}, available: {0}, booked: {0}",
                          p.ProductID, p.AvailQty, p.BookedQty);
    }
}
```

BQL library also supports such advanced features as:

- DACs that are not bound to the database
- Virtual fields that are not bound to the database
- Scalar sub-selects

- Projections
- Stored procedures execution
- Server-side calculated fields
- Non-blocking updates of statistical data records

Entity Model Declaration

Business Entity or simply *Entity* in Acumatica Framework represents an individual instance of the objects (such as *Product*, *Order*) to which the information pertains. Entity can be simple, where the data are represented with a single database record in a single table, or complex. With the complex entity, data are typically held in multiple tables and associated through a complex hierarchy and relationship rules.

Working with the business entities in Acumatica Framework is implemented through the *business logic controller* object also referred as *graph* (graph is a mathematical term for a set of objects where some pairs of objects are connected by links).

A graph provides the interface for the presentation logic to operate with the business entity and relies on Data Access Layer components to store and retrieve the business entity from the database.

Let’s first take a look at the declaration of a simple business entity:

```
//Declaration of the graph
public class ProductMaint : PXGraph<ProductMaint>
{
    //Declaraion of the data view
    public PXSelect<Product> Products;

    //Declaration of the actions
    public PXCancel<Product> Cancel;
    public PXSave<Product> Save;
}
```

In this example the graph implements the following interfaces:

- **Products** – the *data view* that can be used for querying and modifying entity data
- **Cancel** – the *action* that discard all the changes made to the entity and reloads it from the database
- **Save** – the *action* that commits the changes made to the entity to the database and then reloads the committed data

Handling Entity Data

Data View and Entity Cache

Data views implement the interfaces for querying entity data from the business logic controller and submitting modified data back to the entity.

Data views are declared as public fields of `PXSelect` command type:

```
public PXSelect<Product> Products;
```

Based on this declaration, the system automatically instantiates the DAC *entity cache*.

An *entity cache* object in the Acumatica Framework is the primary interface for working with individual entity records from the graph business logic. It has two components and two primary responsibilities:

- The `Cached` collection – in-memory cache that contains modified entity records. The `Cached` collecton is instantiated based on the corresponding DAC declaration and managed by the cache.

- The controller – the cache component that implements basic CRUD operations on the `Cached` collection and triggers a sequence of data manipulation events when modifying or accessing the data in the `Cached` collection. These events can be later subscribed from the graph to implement the business logic associated with entity data modification.

The diagram below helps to understand the internal graph structure and responsibilities of the data view and the entity cache.

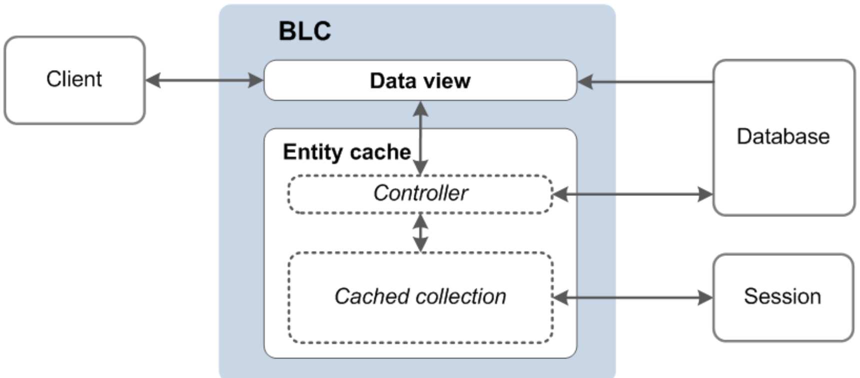


Figure: The graph structure – a data view and an entity cache.

Data Modification Scenarios

Now lets consider basic entitiy data manipulation scenarious that can be executed from the graph business logic or from the user interface. Entity data manupulation through the user interface indirectly invokes the same methods as the direct call from the business logic.

Querying Entity Data for the First Time

Entity data can be requested through the *Products.Select()* method. During this operation, the systems will execute BQL command from the data view declaration. Data returned by the BQL command will be returned to the requestor. See the diagram below.

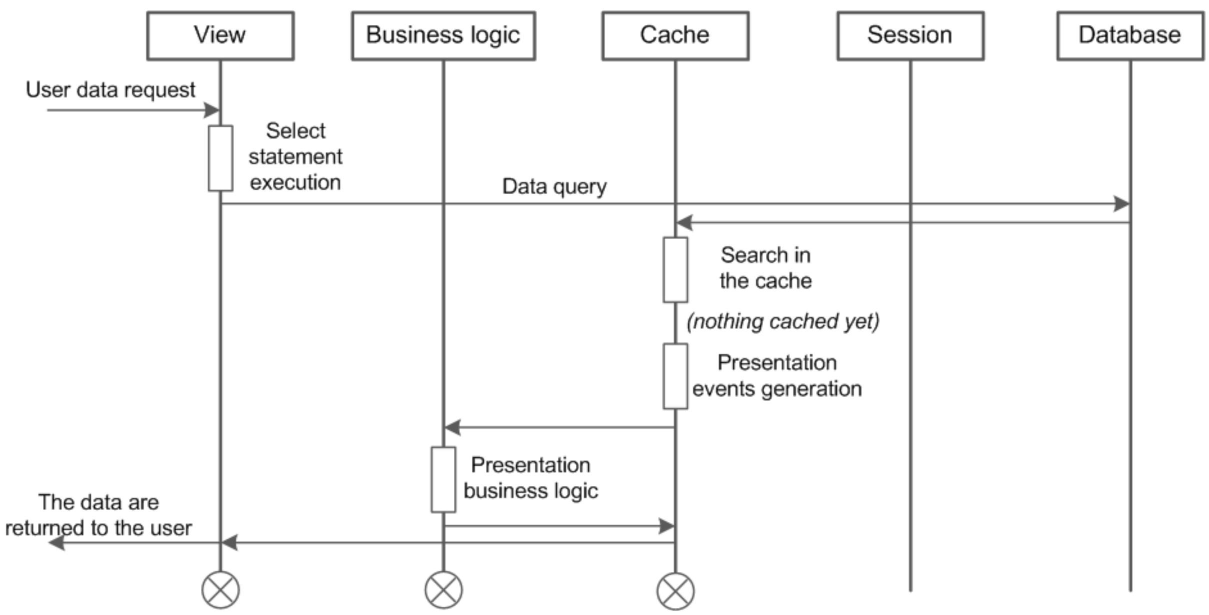


Figure: Querying entity data for the first time.

Updating an Existing Entity Record

An existing business entity record can be updated through the *Products.Update(record)* method. This method places the modified record into the cache.

If the data record is not found in the `Cached` collection, the cache controller will load the data record from the database, add it to the `Cached` collection, mark it as updated, and update it with the new values. The search of the data record in the `Cached` collection and loading of the data record from the database is based on the DAC key fields. The diagram below illustrates this scenario.

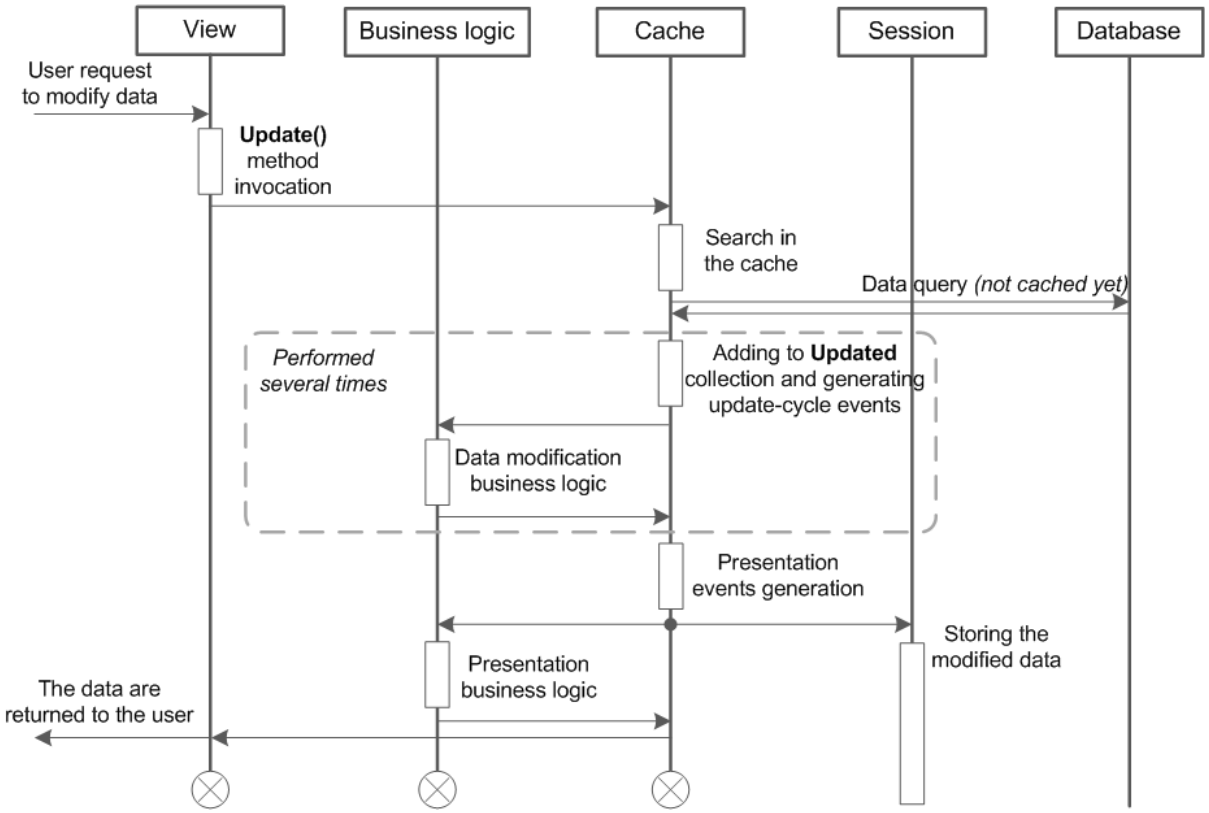


Figure: Updating the entity record for the first time.

If the updated record exists in the `Cached` collection the cache controller will locate it and update it with the new values. The diagram below illustrates this scenario.

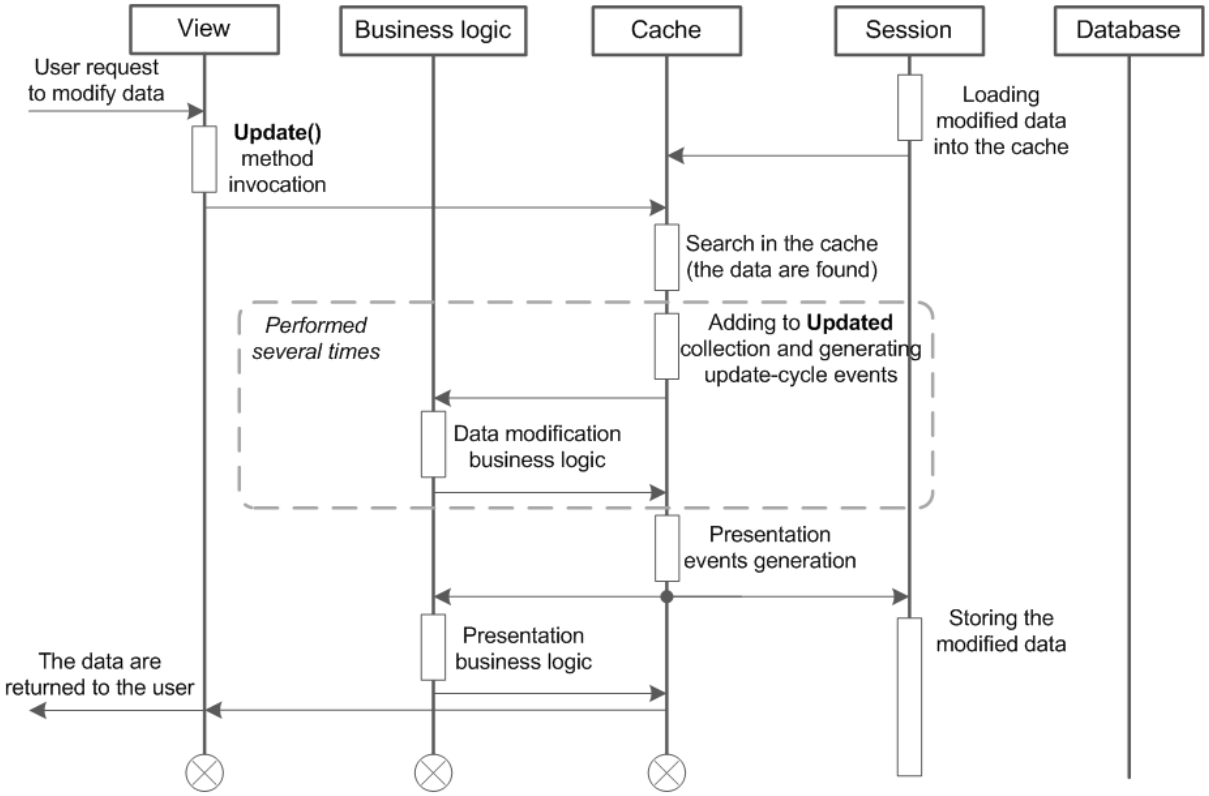


Figure: Updating the cached (previously modified) entity record.

Inserting a New Entity Record

A new record can be inserted into the business entity through the *Products.Insert(record)* method. The new inserted record will be added to the `Cached` collection and marked as inserted. The diagram below illustrates this scenario.

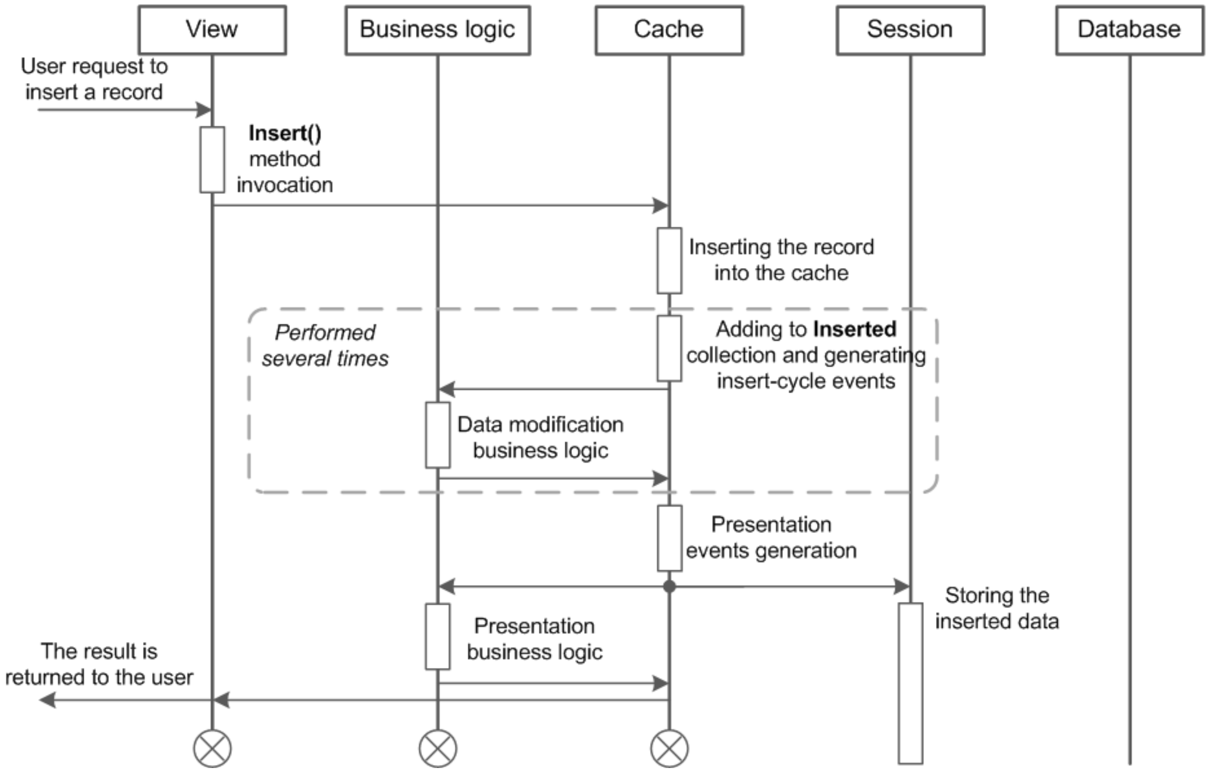


Figure: Inserting the new entity record.

Deleting an Existing Entity Record

An existing record can be deleted from the business entity using the *Products.Delete(record)* method.

If the data record is not found in the `Cached` collection, the cache controller will load the data record from the database, add it to the `Cached` collection, and mark it as deleted. The search of the data record in the Cached collection and loading of the data record from the database is based on the DAC key fields. The diagram below illustrates this scenario.

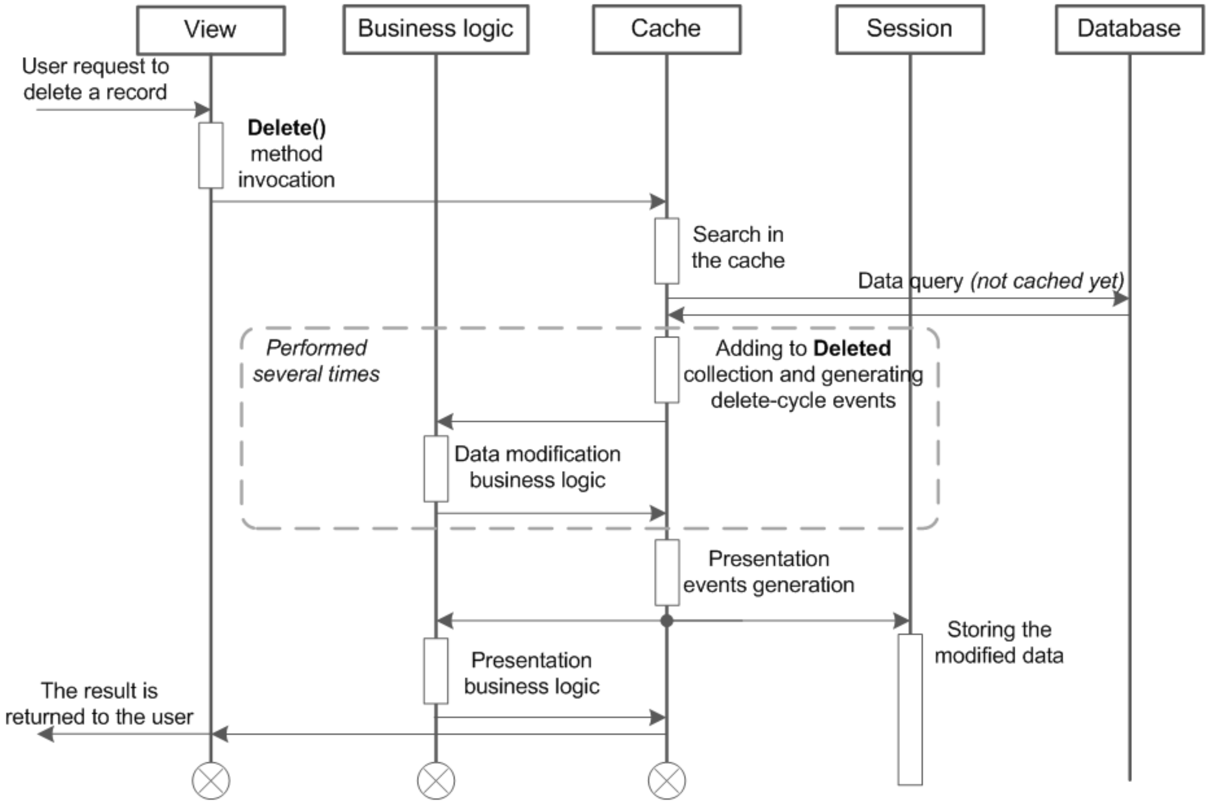


Figure: Deleting the non-cached (unmodified) entity record.

If the deleted record is found in the `Cached` collection, the cache controller will locate it and mark as deleted. The diagram below illustrates this scenario.

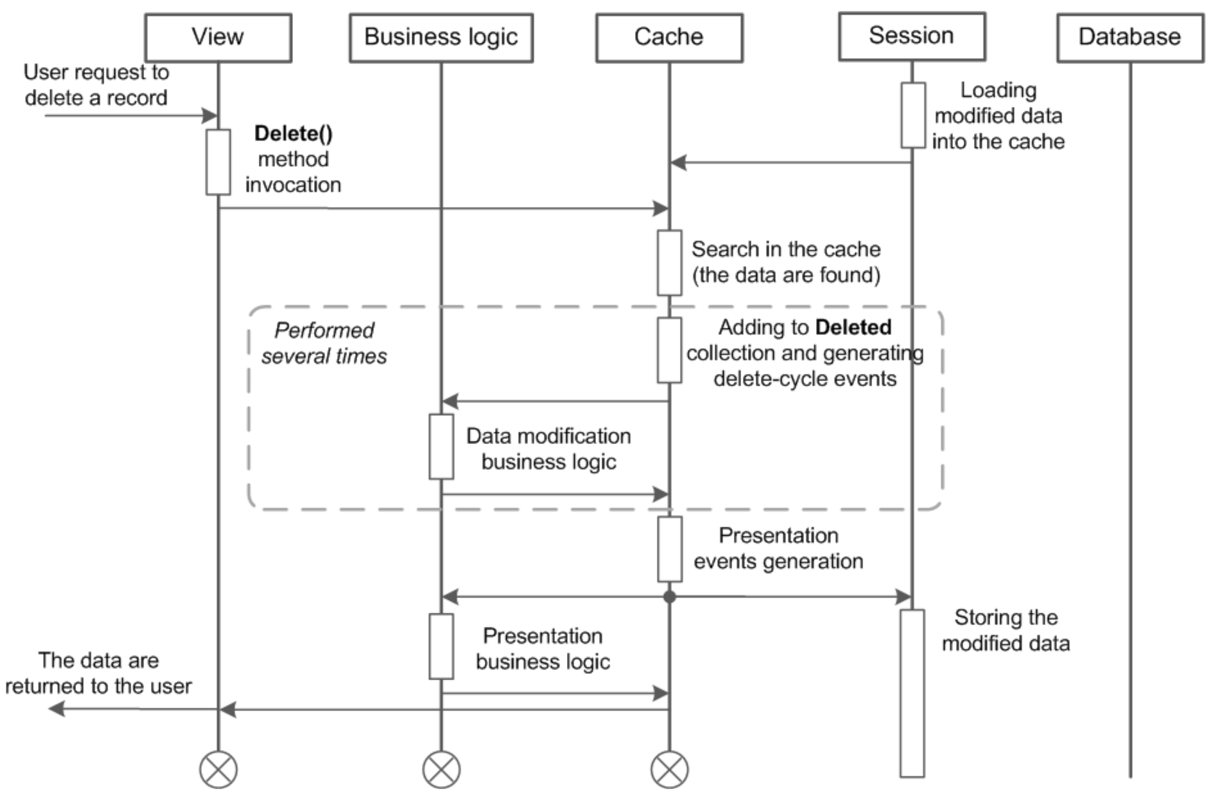


Figure: Deleting of the cached (previously modified) entity record.

Querying an Updated Entity Data

Entity data can be modified and then queried again. In this scenario, the data records stored in the caches memory will be merged with the result of the BQL command execution. Data records merge is based on DAC key fields. The final result of the `Select()` execution will incorporate all the earlier entity records modifications that has not been preserved to the database yet. The diagram below illustrates this scenario.

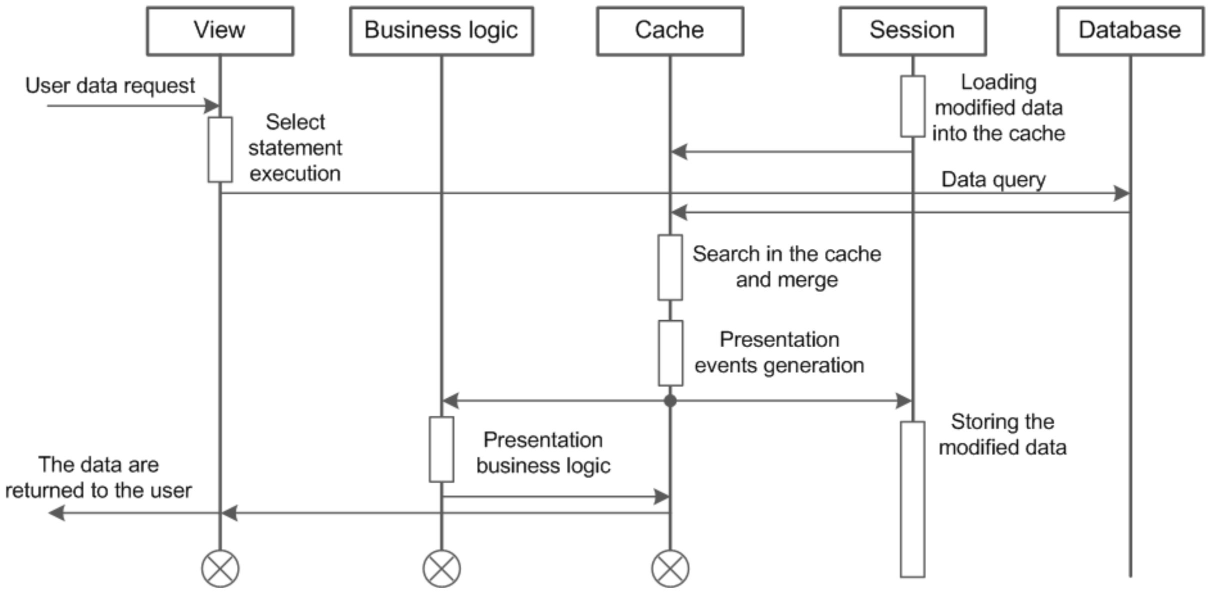


Figure: Querying the modified entity data – reading and merging with the cached data.

Persisting Entity Changes to the Database

When entity data are modified, the system has two different entity versions, the new one stored in the caches memory and the original one persisted in the database. At this point a programmer has two options:

- Save the new entity version to the database using the `Persist()` method of the graph
- Discard all in-memory changes and load the original entity version using the `Clear()` method of the graph

From the Presentation Layer these methods are called by invocation of the `Save` and `Cancel` actions. These actions are predefined and mapped to the `Persist()` and `Clear()` methods.

The diagram below illustrated saving of entity changes to to the database.

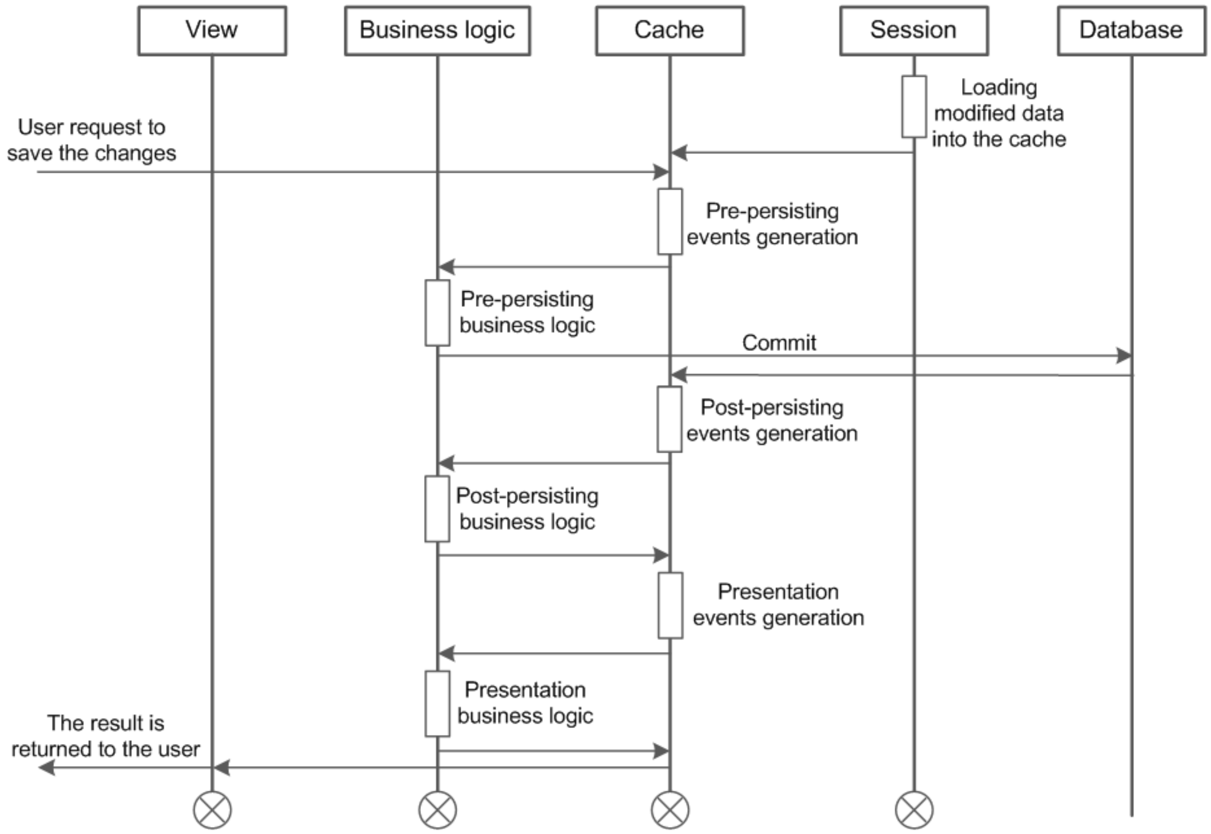


Figure: Saving the entity changes to the database.

The diagram below illustrated discarding of all in-memory entity changes.

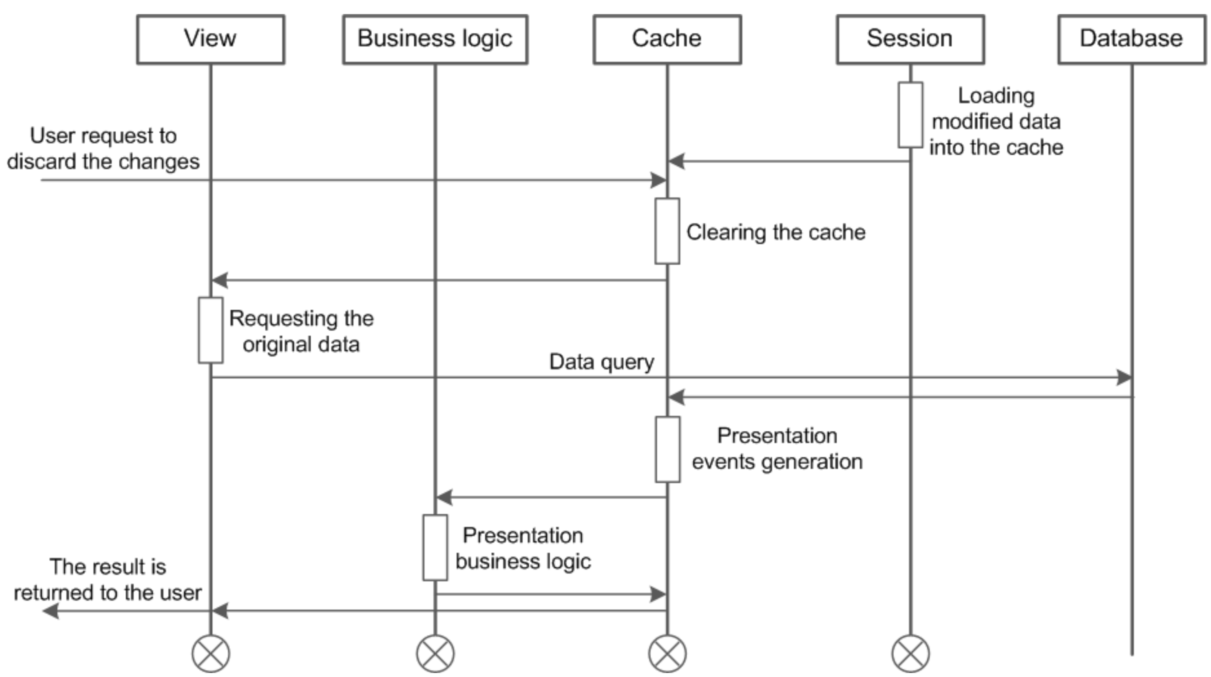


Figure: Discarding the changes and loading the original entity data.

Preserving the Entity Version Between the Round Trips and Handling the Subsequent Selects from the Views

It is important to understand that a graph is a stateless object. It is discarded after each data request. In order to preserve the modified entity version between the requests, the cache controller serializes the `Cached` collection into the session state and restors it later when the graph is instantiated on the subsequent request. In this scenario, it is very important that the cache contains only the modified entity records, not the complete entity record set.

Implementing Business Logic

Business logic is implemented by overloading certain methods invoked by the system in the process of manipulating data. For such procedures as inserting a data record or updating a data record, the `PXCache` controllers generate series of events causing invocation of the methods called *event handlers*. The application is able to interfere in the series of events on different stages. For this purpose, the application impements methods that are executed as event handlers.

There are 18 events raised on all stages of data processing.

Business logic can be divided into common logic relevant to different parts of the application and the logic specific to an application screen (web page). The common logic is implemented through event handler methods defined in attributes, while the screen-specific logic is implemented as methods in the associated graph.

Common Business Logic

The common business logic is implemented by defining event handlers in attributes. If such attribute is added to the declaration of a data access class, attribute logic is applied to the data records of this type for any graph used to access this table.

There are a number of predefined attributes implemented in the framework. For example, in the following declaration of a data field for a column

```
[PXDBDecimal(2)]
public virtual string AvailQty { get; set; }
```

`PXDBDecimal` is an attribute binding this field to a database column of the decimal type. The attributes of this form exist for most database data types.

Another typical example of an attribute is `PXUIField`. It is used to configure the input control for the column in the user interface. This allows having the same visual representation of the column on all application screens (unless a screen redefines it). For example:

```
[PXDBDecimal(2)]
[PXUIField(DisplayName = "Available Qty", Enabled = false)]
public virtual string AvailQty { get; set; }
```

Application can also define its own attributes, in the following way:

```
// Application-defined attribute implementing common business logic
public class MyAttribute : PXEventSubscriberAttribute,
                           IPXEventNameSubscriber
{
    // An event handler
    protected virtual void EventName(PXCache sender,
                                     PXRowEventNameEventArgs e)
    {
        ...
    }
    ...
}
```

Such attributes are also added to the DAC declaration:

```
[PXDBDecimal(2)]
[PXUIField(DisplayName = "Available Qty", Enabled = false)]
[MyAttribute]
public virtual string AvailQty { get; set; }
```

Screen-Specific Business Logic

For a specific screen, the application can redefine the common logic or extend it. For this purpose, you should define event handlers in the graph associated with the screen. Each event hanlder method is tied to a particular table or a table field via the naming convention.

For example, you can verify a value of a column:

```
public class ProductRecalc : PXGraph<ProductRecalc>
{
    ...
    // Event handler verifying that the value of the AvailQty column
    // in Product records is greater than 0.
    // It is triggered when, for instance, a Product record is updated.
    protected virtual void Product_AvailQty_FieldVerifying(
        PXCache sender,
        PXFieldVerifyingEventArgs e)
    {
        Product p = (Product)e.Row;
        if (p != null && p.AvailQty != null)
        {
            if (p.AvailQty < 0)
                throw new PXSetPropertyException<Product.availQty>(
                    "Value must be greater than 0.");
        }
    }
}
```

Programming Tasks

The articles from this section explain how to complete various programming tasks that you may face with while developing a business application on Acumatica Framework.

- *Localizing Applications*
- *Generating a Data Access Class*
- *Working With Images*
- *Adding Widgets to Dashboard*
- *Data Representation*
- *Calculations*
- *Data Input*
- *Interaction With the Server*
- *Creating an Acumatica ERP Add-on Project*
- *Implementing a Credit Card Processing Plug-in*
- *Using Substitute Keys*
- *Calling a New PXSmartPanel*

Localizing Applications

Acumatica Framework provides built-in localization tools that you can use to translate the user interface and application messages to different languages. This topic provides guidelines on how to prepare the Acumatica Framework application for further localization efforts. (See the related link under this topic.)

To get the application ready for localization, you must prepare data access classes (DACs) and the application code.

What Can Be Localized

The system can retrieve the string constants specified in the following items of the application:

- `PXUIField` attributes in DAC fields
- `PXUIField` attributes in business logic container (BLC) DAC override fields and actions
- `PXStringList` and `PXIntList` attributes
- Tooltips for the `PXButton` attribute
- Captions of form, grid, and panel controls and input control labels specified in the ASPX page
- Site Map tree (titles of all sitemap nodes)
- Reports (textbox labels, diagram agenda, etc.)
- Classes marked with the `PXLocalizable` attribute

By using the **System** > **Management** > **Manage** > **Translation Dictionaries** system webpage, you can add translations for the collected string constants and save them to the database. When a user signs in with a specific language, the systems loads the translations and displays translated strings to the user.

If the same string is found in multiple places in the application, the system saves information about all the occurences. You can specify a default translation that applies to all occurences of the same string and separate translations for some or all the occurences.

Preparing DACs

The system can automatically update the translation dictionary of Acumatica ERP with the string constants specified in the `DisplayName` parameter of the `PXUIField` attribute. The translation dictionary is also updated with list attributes of the `PXStringList` attribute or `PXIntList` attribute. Therefore, the declaration of a field in a DAC should meet the following requirements:

- Each visible field in a DAC must include the `PXUIField` attribute.
- The `DisplayName` parameter must be specified for the `PXUIField` attribute, not only to make the name of the user interface element of the webpage clearer than the corresponding field name of the database table, but also to provide the localization capability.

Note the following example of a field declaration with the `PXUIField` attribute applied.

```
#region DocType
public new abstract class docType : PX.Data.IBqlField
{
}
[PXDBString(3, IsKey = true, IsFixed = true)]
[PXDefault()]
//The PXUIField with the DisplayName parameter
[PXUIField(DisplayName = "Document Type")]
public override string DocType { get; set; }
#endregion
```

If you apply the `PXStringList` attribute to the string field, its list attributes will also be collected and placed in the dictionary for localization.

Here is an example of a field declaration with the `PXStringList` attribute and `PXUIField` attribute applied.

```
#region LineSource
public abstract class lineSource : PX.Data.IBqlField
[PXString(1, IsFixed = true)]

//The PXStringListAttribute with its list attributes
[PXStringListAttribute(
    new string[] { "D", "R" },
    new string[] { "Draft", "Request" })]

//The PXUIField with the DisplayName parameter
[PXUIField(DisplayName = "Line Source")]
public virtual string LineSource { get; set; }
#endregion
```

Localizing Application Code

To enable localization of messages in the source code, move all translatable strings from the application to the `public static class` marked with the `PXLocalizable` attribute. (The exceptions to this requirement are field descriptions and list attributes in the data access classes, which are handled separately.) An example of such a class follows.

```
using System;
using PX.Data;

namespace PX.Objects.EM
```

```
{
  [PXLocalizable()]
  public static class Messages
  {
     public const string FieldNotFound = "The field specified is not found.";
     public const string InvalidAddress = "The address is not valid.";
     public const string AdditionalData = "Author's title: {0}, author's name: {1}"
  }
}
```

A string may contain placeholders (as with the last code line in the code above, which contains *{0}* and *{1}* placeholders).

The string from a class marked with the `PXLocalizable` attribute can be collected by the application and added to the translation dictionaries. If you need to receive the translated string within the application code, use the `PXMessages.Localize(...)` method or `PXLocalizer.Localize(...)`, as shown below.

```
string msg = PXMessages.Localize(Messages.FieldNotFound);
```

When you throw an exception of `PXException` type or of a type derived from `PXException`, you should provide a not-localized message. The system will localize the message automatically if the translation dictionaries include a translation for this message. See the example below.

```
if (field == null)
{
        throw new PXException(Messages.FieldNotFound);
}
```

Notice that no hyphenation is provided by the system. During the acquisition process of localizable data, all the new-line symbols (*\n\r*) are to be removed. You can use the reserved symbol (~) to cause insertion of a new line.

Localizing Strings in the Code

To get a localized string at run time, you should use the `Localize(string)` method of the `PXMessages` class or the `Localize(string, string)` method of the `PXLocalizer` class.

The `PXMessages.Localize(string)` method searches for the translation of the provided string in the database and returns the first translation found.

```
string text = PXMessages.Localize(PX.Data.Update.Messages.SiteUnderMaintenance);
```

You should use the `PXMessages.LocalizeFormat(...)` method if the string includes placeholders (such as *{0}* or *{1}*).

The `PXLocalizer.Localize(string, string)` method returns the translation with the given key, which you specify in the second parameter. A string may have multiple translations; one translation for each occurence of the string in the application. For each of the occurences, a key value is created. For example, if the string is declared in a class marked with the `PXLocalizable` attribute, the full qualified name of the class is the key, as the following code shows.

```
string localizedMsg = PXLocalizer.Localize(
    ActionsMessages.ChangesWillBeSaved,
    typeof(ActionsMessages).ToString());
```

When you throw an exception of the `PXException` or derived type, you should provide a non-localized string as the exception message. The system will automatically search for translation and display a localized version of the message.

If you change the `DisplayName` value of the `PXUIField` attribute on the fly, create your own `PXUIFieldState`, you should localize the string independently.

Generating a Data Access Class

Once you have linked the created page to the business logic container (BLC) class, you can generate a data access class (DAC) that implements a communication layer between the BLC and the database. To use the Data Access Class Generator to generate the *Country.cs* DAC file code in the simplest way, do the following steps:

In this topic, we assume that your database includes the simple *Country* table. Although for simplicity this table doesn't include the system attribute **NoteID** and the audit fields **CreatedByID**, **CreatedByScreenID**, **CreatedDateTime**, **LastModifiedByID**, **LastModifiedByScreenID**, and **LastModifiedDateTime**, we recommend that you use all these fields in each database table.

1. Open the page in design mode, point to the **ds** control, click the smart tag associated with this control, and select **Generate Class**, as shown in the screenshot below.

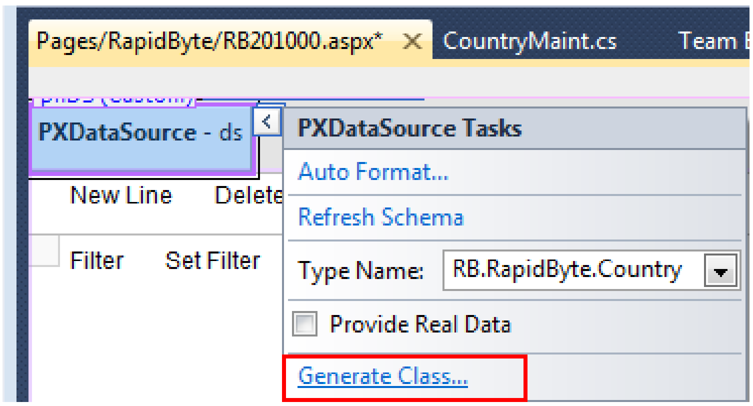


Figure: Starting to generate the DAC

2. In the Data Access Class Generator window that appears (see the screenshot below), type `Country` into the **Name** field under the Table Properties section as the name of the table that will store countries' data, or select **Country** from the drop-down list of database tables. The list of fields from the *Country* table appears.
3. Click **Append UI Attributes** to add the `PXUIField` attribute to the fields.

   If you decide not to display some DAC fields on the webpage, after generating the DAC, you should manually delete redundant `PXUIField` attributes.

4. Click **Generate** to generate the data access class.

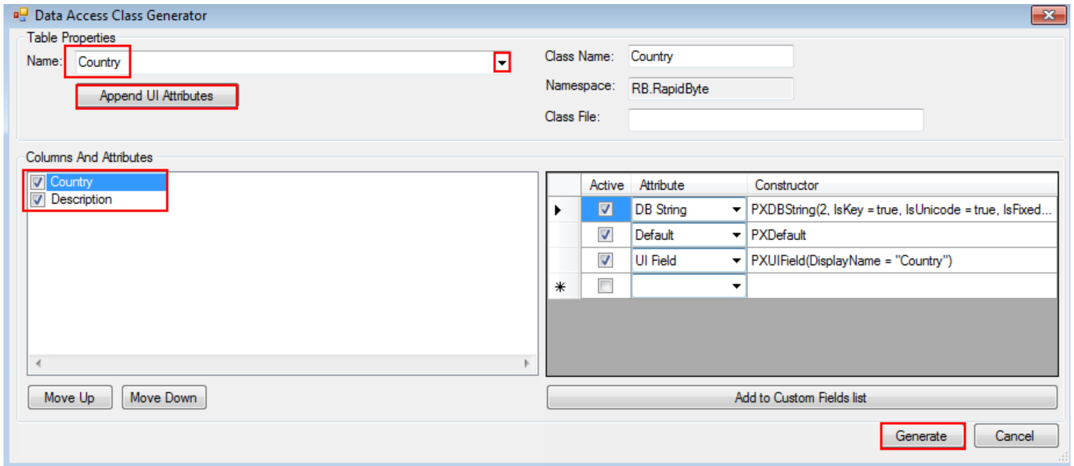


Figure: Generating the DAC by using the Data Access Class Generator window

As a result, Acumatica Framework creates the new file, *Country.cs*, with the generated DAC code and then opens this file

When the list of fields is loaded, the Data Access Class Generator automatically assigns attributes to the audit fields. The settings are stored in the *CustomFields.config* file, which you can update by clicking **Add to Custom Fields List**. If the DAC already exists, the wizard that is built into the DAC Generator loads data from the DAC and replenishes the list of fields with the database fields that are not listed in the DAC. By default, new fields, which are displayed at the end of the list, are not selected.

When you click **Generate**, already existing fields are overridden if you have selected them for generation.

The *CustomFields.config* file has an XML structure and consists of two main sections, called *Config* and *CustomFields*.

In the *config* section, the design class type is annotated, and some necessary default property values are defined.

The *CustomFields* section contains the definitions, type definitions, and constructors of the system attribute `NoteID` and the audit attributes `CreatedByID`, `CreatedByScreenID`, `CreatedDateTime`, `LastModifiedByID`, `LastModifiedByScreenID`, and `LastModifiedDateTime` are defined.

Only Acumatica ERP developers can change the content of this file. You can use this file as a reference manual, for instance, on the stage of constructing the structure of database tables or the generation of multiple DACs.

Working With Images

This topic covers how to upload images to attach them to webpages and how to manage uploaded images. You can attach image and video files to any area of a webpage: upper (form), lower (tab), or lower (tab table). In this topic, attachment of an image file to the form area of a webpage is illustrated.

Preparing a Placeholder to Upload an Image File

To make it possible to upload an image file and attach the uploaded image to the required area of the webpage, you must perform the following actions:

1. Add two mandatory fields—**Image**, having the *nvarchar(256)* data type, and **NoteID**, with the *bigint* data type—to the database table whose fields are to be used for generating the respective data access class (DAC) fields, so that the `Image` and `NoteID` fields in the DAC code are defined as classes.
2. Open an Acumatica Framework solution and generate a new DAC.

3. Create the page.

4. Set the `DataMember` property value as the related business logic container (BLC, also called *graph*) name based on this DAC.

5. Open the source mode and modify the .aspx page code of the created page: Replace the starting and ending *PXTextEdit* tags of the `Image` field with the *PXImageUploader* tags, as shown in the screenshot below.

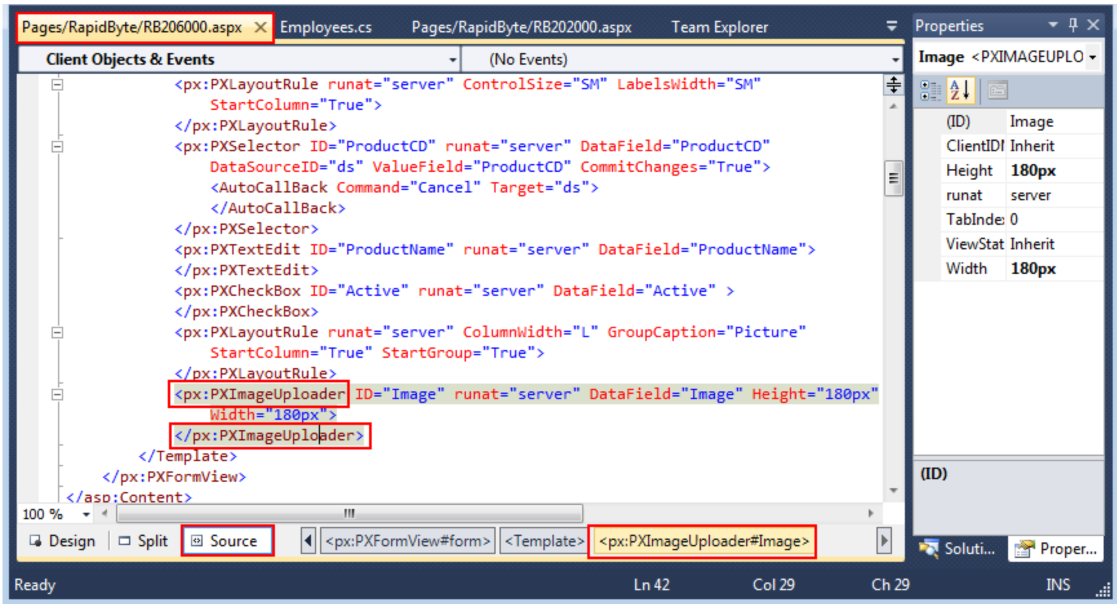


Figure: Modifying the tag name of the .aspx page

6. By using the Layout Editor window, add the **Image** field (after setting optimal default `Height` and `Width` property values), along with all the other required fields, onto the appropriate area of the page. (You shouldn't add the **NoteID** field onto the page.)

Image file extensions of files to be uploaded must be registered on the *File Upload Preferences* (SM.20.25.50) form. Navigate to the **Configuration** > **Document Management** > **Configure** > **File Upload Preferences** form. If the required file types are not defined already, define them and save your changes. On this form, you can also define the maximum size of an uploaded file (in kilobytes), as shown in the following screenshot.

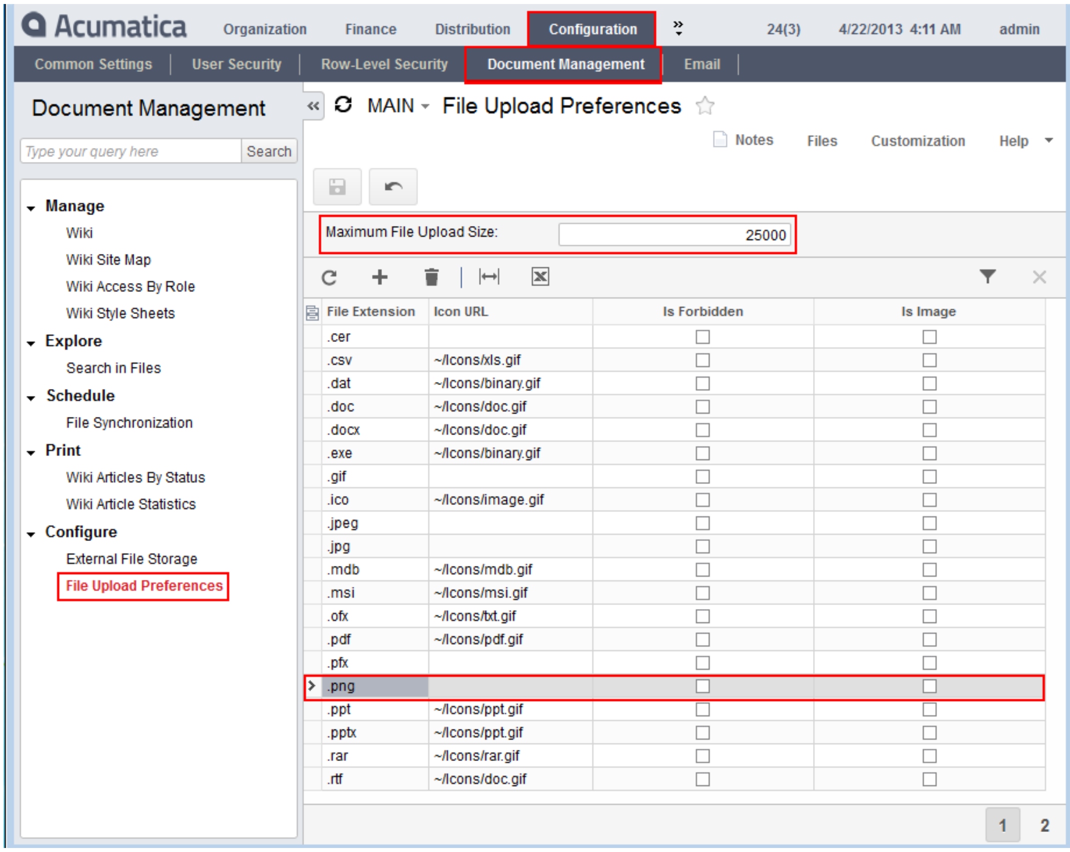


Figure: Making sure image file extensions are registered

Uploading Image Files and Managing Images

This section provides a simple example, by using the *Products* sample webpage, of uploading and managing image files. To upload three images, proceed as follows:

1. Start the application, navigate to the *Products* webpage, and click **Click here to upload image** in the upper webpage area, where you had placed the **Image** field. Click **Browse** and find the required image file.
2. Select the desired file and click **Upload**. Notice the image under **Click here to upload image**, as the screenshot below illustrates.

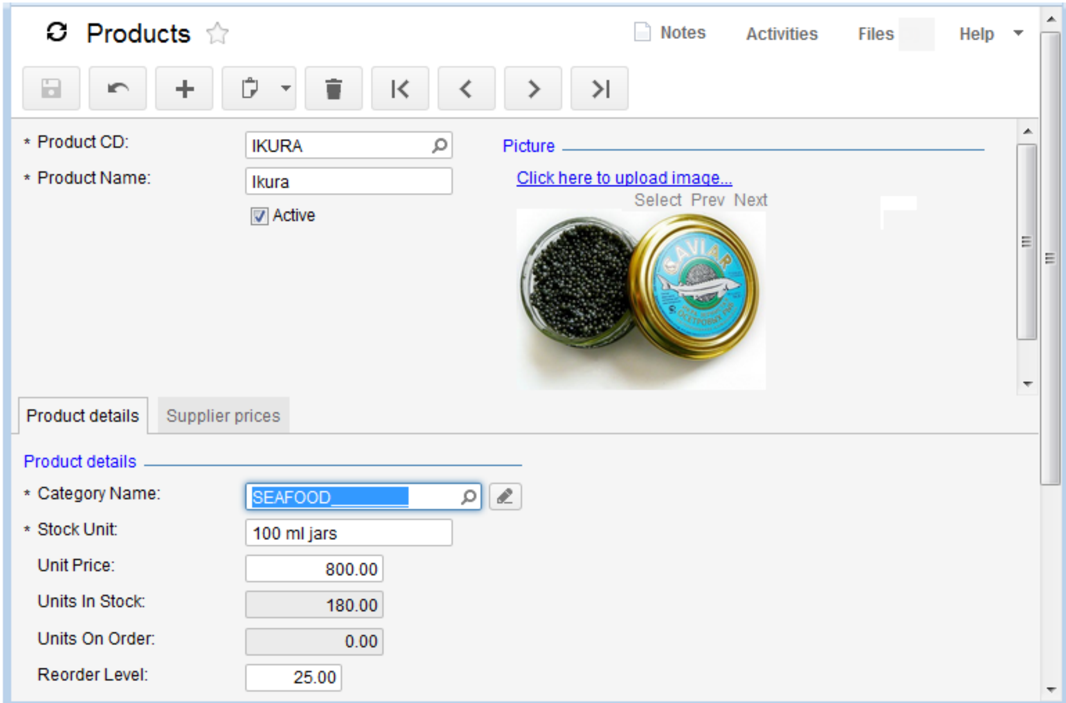


Figure: The first uploaded image

3. To upload a second and third image, repeat the two previous instructions twice.
4. After you have uploaded the third image, ensure that the **Next**, **Prev**, and **Select** navigation buttons in the upper right corner have become available.

   By clicking **Next** or **Prev**, you can scroll through all images—those you uploaded and those that already existed.

5. Select the image to be displayed by default.
6. To adjust the selected image to be displayed by default, click **Select**; then click **Save** on the form toolbar. Open another webpage or select another product, and then open the *Products* webpage and select the product record to which you assigned the default image. Notice that the default image is located where it was earlier.
7. Click the image to see the file image in its original scale.

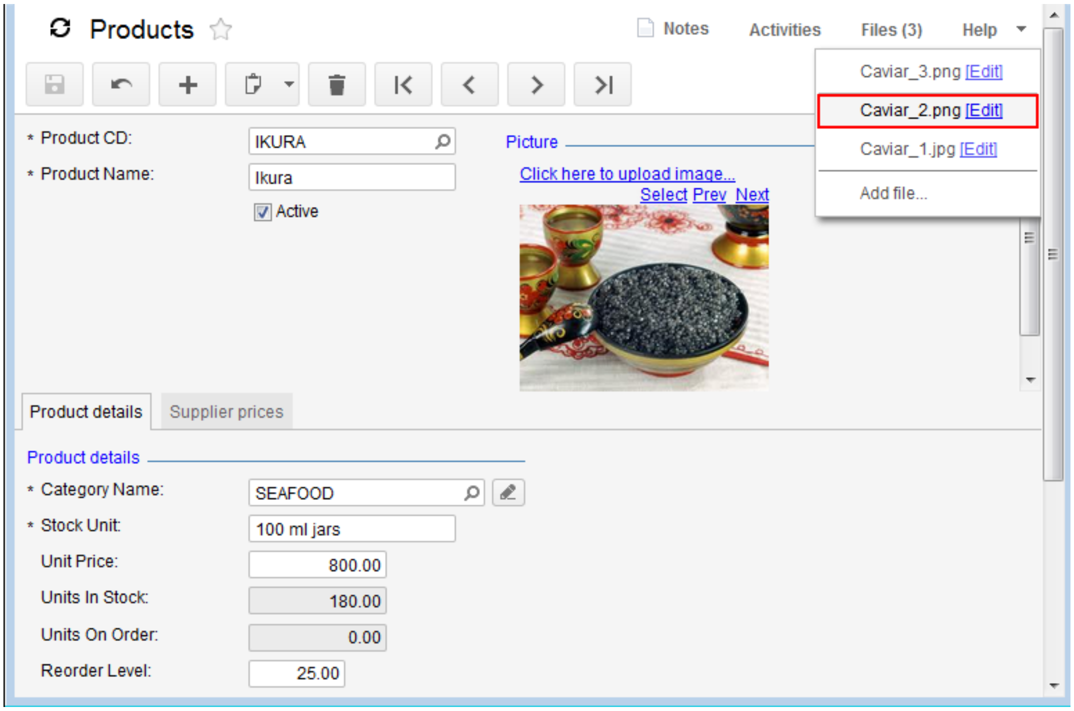


Figure: Opening the image file editor window

8. To replace any attached image file, click **File** and then click the *Edit* link (at the right of the name of the image file, as shown in the screenshot above) to open the *File Maintenance* (SM.20.25.10) form in a window. On the form toolbar of this form, click **Upload New Version** (see the screenshot below), and then attach the file as described above in Instruction 2. After you have replaced the file, you can see the new line in the table on the **Versions** tab; the appearance of the new line means that the full uploading and replacement history data is available for any uploaded image.

To delete the attachment of the image (or any version of the image file), just click **Delete** (to delete the image file attachment) in the upper area or **Delete Row** (to delete a version of the image file attachment) in the lower part of the *File Maintenance* form.

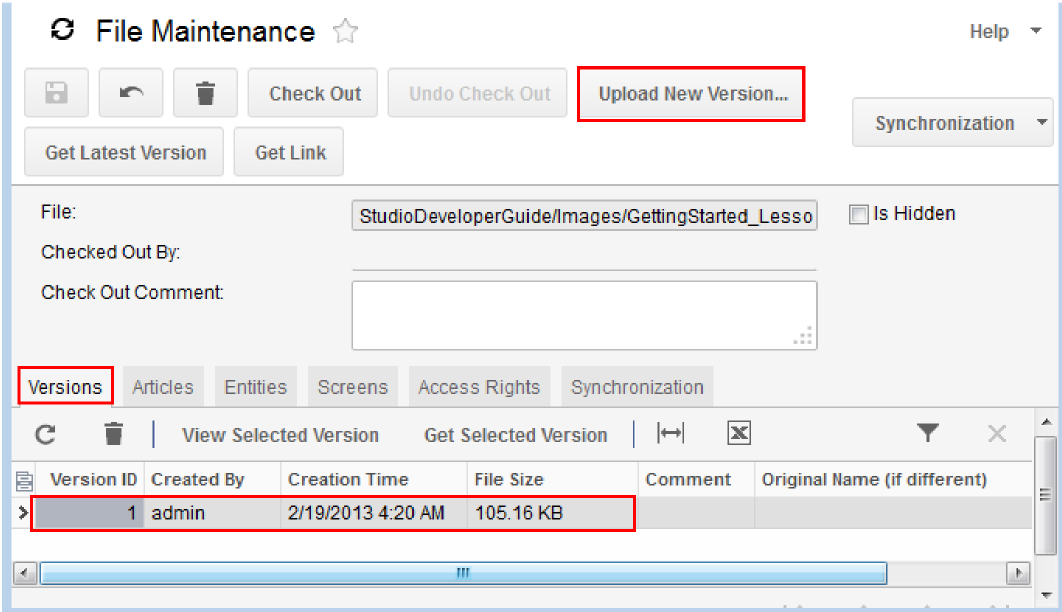


Figure: Replacing the attached image file

Adding Widgets to Dashboard

Possible widget types (parameters of the `DashboardType` attribute):

- 0 - Table (default)
- 1 - Wiki article
- 2 - Task
- 6 - Table with owner and workgroup
- 7 - Calendar
- 8 - Generic Inquiry
- 20 - Chart

Data Representation

In this chapter, you will get acquainted with the various aspects of a webpage representation, such as how to configure and design a webpage layout, adjust lookup fields, filter webpage data, and use status field.

Content

This chapter covers the following topics:

- *Filtering Data on a Webpage*

Filtering Data on a Webpage

This topic describes two filtering methods: setting selection criteria in the top (master) area of a webpage to filter the details, and defining a reusable filter. The topic describes how you would create a special inquiry webpage that enables the filtering of records; such a webpage uses the first filtering method. The second method, defining a reusable filter, can be used with most processing webpages and reports.

We illustrate the implementation of both methods and the appropriate testing steps by using an example with a simple application, *Rapid Byte*. You should not perform any of the actions described in this topic. These actions are provided to show a part of the development process while helping you become acquainted with the filtering methods that can be used in applications developed with Acumatica Framework.

A third filtering method, used for processing pages, is described in the last section of this topic.

Creating a DAC and a BLC for the Inquiry Webpage

In this section, the groundwork is laid for the first filtering method, for which you would create a special inquiry webpage. This section describes the process of creating a data access class (DAC) and a business logic container (BLC, also called a *graph*) for filtering webpage data. You can see the code lines that implement the filtering logic for the first filtering method.

Suppose that you need to create a complex webpage based on the *FormDetail* template to filter and sort products that the company sells or plans to sell. In the upper (master) area of this webpage, the **Category Name** (of the product) and **Supplier ID** fields will be used as the filter conditions, while in the lower (details) area, the table with the filtered products will be displayed.

For this method, first you would create a simple DAC for filtering conditions, and then you would create a BLC to implement the filtering logic. To perform these steps, you would do the following: (Again, you shouldn't perform these actions at this time; just analyze them.)

1. Manually create a new DAC, *ProductFilter*, that includes two DAC fields, `CategoryName` and `SupplierID`, as shown below.

```
// public class ProductFilter : PX.Data.IBqlTable
namespace RB.RapidByte
{
 using System;
 using PX.Data;

 [System.SerializableAttribute()]
 public class ProductFilter : PX.Data.IBqlTable
 {
  #region CategoryName
  public abstract class categoryName : PX.Data.IBqlField
  {
  }
  [PXString(15, IsUnicode = true)]
  [PXUIField(DisplayName = "Category Name")]
  [PXSelector(typeof(Category.categoryName),
                        DescriptionField = typeof(Category.description))]
  public virtual string CategoryName { get; set; }
  #endregion
  #region SupplierID
  public abstract class supplierID : PX.Data.IBqlField
  {
  }
  [PXString(15, IsUnicode = true)]
  [PXUIField(DisplayName = "Supplier ID")]
  [PXSelector(typeof(Search<Account.accountID, Where<Account.companyType,
         Equal<CompanyType.supplier>>>),
          new Type[] {typeof(Account.accountID),
                     typeof(Account.companyName),
                     typeof(Account.country),
                     typeof(Account.contactName),
                     typeof(Account.contactTitle)
                          })]
  public virtual string SupplierID { get; set; }
  #endregion
    }
}
```

Because `PXFilter` contains a single DAC object that is always created during webpage initialization and never saved to the database, there is no need to specify any key field within a DAC exclusively used in the `PXSelector<Table>` data members.

2. Add the *ProductInquiry.cs* BLC file code, based on the *PXGraph* template, and modify it as follows. (The + sign at the left of the code line means that this code line must be added, while the - sign means that you should delete the code line because it is redundant.)

```
using System;
using System.Collections;
-using System.Collections.Generic;
using PX.Data;
using PX.SM;

namespace RB.RapidByte
{
 public class ProductInquiry : PXGraph<ProductInquiry>
 {
+  public PXCancel<ProductFilter> Cancel;
+  public PXFilter<ProductFilter> Filter;
+  [PXFilterable]
+  public PXSelectJoin<Product, LeftJoin<SupplierProduct, On
+  <Product.productID, Equal<SupplierProduct.productID>>>> ProductRecords;


+  public ProductInquiry()
+ {
+   Cancel.SetCaption("Clear Filter");
+   this.ProductRecords.Cache.AllowInsert = false;
+   this.ProductRecords.Cache.AllowDelete = false;
+   this.ProductRecords.Cache.AllowUpdate = false;
+ }

+  protected virtual IEnumerable productRecords()
+  {
+   ProductFilter filter = Filter.Current as ProductFilter;
+   PXSelectBase<Product> cmd = new PXSelectJoinOrderBy<Product, LeftJoin
+     <SupplierProduct, On<Product.productID, Equal
+     <SupplierProduct.productID>>>, OrderBy<
+     Asc<Product.productName>>>(this);

+   if (filter.SupplierID != null)
+   {
+      cmd.WhereAnd<Where<SupplierProduct.supplierID,
         Equal<Current<ProductFilter.supplierID>>>>();
+   }
+   if (filter.CategoryName != null)
+   {
+      cmd.WhereAnd<Where<Product.categoryName,
         Equal<Current<ProductFilter.categoryName>>>>();
+   }
+   return cmd.Select();
+  }
 }
}
```

3. Build the project.

`PXFilter` always contains a single data record, which is created and inserted into an appropriate `PXCache` object when the BLC is retrieving data. The `PXFilterable` attribute is used to allow the end user to filter a **PXGrid** control's data (the records of a tab table or the details table of a webpage).

In the DAC code, the `PXFilter` BQL statement blocks all logic associated with database operations, neither attempting to read from the database nor persisting changed records. You use `PXFilter` for storing and displaying records that are used in business logic and available on the user interface (UI) but that you do not need to preserve. `PXFilter` creates a unique record in a cache, and the values of the record attribute depend on the current filtering conditions.The `PXFilterable` attribute activates the preservable (reusable) filter on the details table so the user can save the current filtering settings as a template filter.

The `PXFilterable` attribute enables the user to work with the second filtering method (described in the next section), while all the other lines of the BLC file code are needed to implement the first filtering method.

The *ProductInquiry* BLC is not parameterized with the primary view type—that is, the BLC class does not have the second parameter, as the following expression shows: `public class ProductInquiry : PXGraph<ProductInquiry>`. The following table describes the programming goals and the way the BLC code accomplishes them.

<table>
<tr><th>Programming Goal</th><th>Description</th></tr>
<tr><td>Add a button and define its name</td><td>Because the standard navigation buttons should not be displayed on the form toolbar for this webpage, you should add your own buttons. To add the Cancel button, which clears the filter, insert the following code line.
<pre>public PXCancel<ProductFilter> Cancel;</pre></td></tr>
<tr><td>Disable the details table</td><td>The following code lines disable the update, insert, and delete functionality for the details table. Because the application is stateless, these access rights must be set each time data is needed for the user.
<pre>this.ProductRecords.Cache.AllowInsert = false;
this.ProductRecords.Cache.AllowDelete = false;
this.ProductRecords.Cache.AllowUpdate = false;</pre></td></tr>
<tr><td>Compose the BQL statement</td><td>The BQL library supports dynamic statement composition. The following code lines set up a new BQL command.
<pre>PXSelectBase<Product> cmd = new PXSelectJoinOrderBy<Product,
LeftJoin
    <SupplierProduct, On<Product.productID, Equal
    <SupplierProduct.productID>>>, OrderBy<
    Asc<Product.productName>>>(this);</pre>When the user inserts the SupplierID or CategoryName value as a filter parameter, the base statement is dynamically modified, based on one or both values of the filter parameters. The following code lines enable the user to receive the filtered records.
<pre>if (filter.SupplierID != null)
{
    cmd.WhereAnd<Where<SupplierProduct.supplierID,
    Equal<Current<ProductFilter.supplierID>>>>();
}
if (filter.CategoryName != null)
{
    cmd.WhereAnd<Where<Product.categoryName,
    Equal<Current<ProductFilter.categoryName>>>>();
}
return cmd.Select();</pre></td></tr>
</table>

Creating an Inquiry Webpage

This section describes the creation of an inquiry webpage based on the DAC and BLC created in the previous section. By using this webpage, an end user could use the first filtering method. Here are the instructions you would perform (again, you shouldn't perform any of these actions at this time) to create and refine an inquiry webpage to filter products:

1. In the Solution Explorer window, right-click **Pages**, select the folder of your solution, and select **Add New Item**. Select the **Visual C#** node of the template tree, select the *FormDetail* template, and enter the page name. Click **Add** to create the page.
2. Open the created page in design mode, refresh it, and specify the following control properties for the **ds** control to link it to the created BLC:
    - `TypeName`: *RB.RapidByte.ProductInquiry*
    - `PrimaryView`: *Filter*
3. Specify the following properties for the **PXFormView** control (**form**):
    - `Datasource`: *ds* (has been automatically set by the system)
    - `DataMember`: *Filter*
4. For the **PXGrid** control (**grid**), specify the following properties:
    - `Datasource`: *ds* (has been automatically set by the system)
    - `DataMember`: *ProductRecords*
    - `SkinID`: *DetailsWithFilter*
5. By using the Layout Editor, generateand adjust two filtering fields and add the fields onto the master area of the page, and then generate, adjust, and add all the necessary columns onto the details table.
6. Build the solution.
7. Start the application and open the *Product Inquiry* webpage.
8. By using the **Category Name** lookup field, select a category name and watch the filtering of the information in the details table (see the screenshot below). You can also select the supplier by using the **Supplier ID** lookup field; again note the filtering of the information in the details table.

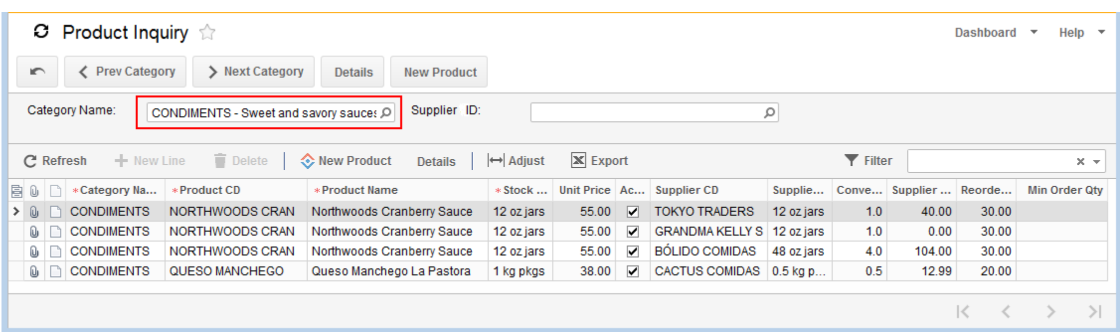


Figure: Analyzing the filtering effect

Filtering Data on the Webpage by Using Two Methods

This section demonstrates how users can filter data on the created webpage by using two methods: specifying selection criteria in the top (master) area of the created page, and defining a reusable filter. (Again, you shouldn't perform these actions.) To analyze both methods, you would proceed as follows:

1. Open the *Product Inquiry* webpage, which shows a variety of information for each product record that already exists in the database, such as the stock and supplier unit of measure (**Stock Unit** and **Supplier Unit**), sales and supplier price (**Unit Price** and **Supplier Price**), conversion factor, and minimum order quantity.
2. To use the first filtering method, in the **Category Name** field, select a category. This filters data by the selected category.
3. In the **Supplier ID** field, select a supplier to see data filtered by the specified category and supplier.
4. Click the **Cancel (Esc)** button in the form toolbar to again display all product records.

   Because these filtering conditions (selection criteria) cannot be saved for later use, the first filtering method can be considered an ad hoc method.

5. To begin using the second filtering method (establishing a reusable filter), click the **Filter** icon to bring up the **Filter settings** dialog. In the condition table, enter two conditions joined by the *AND* logical operator, as shown in the screenshot below. To save this condition as a named filter to make the filter conditions reusable, click **Save**, and enter the name of the filter (for instance, *1*). Select the **Default** check box if you want these filter conditions to be applied automatically when you open this page. (Each time you save a filter as the default for a page, this check box is cleared automatically for any filter that was previously set as the default for the page.)

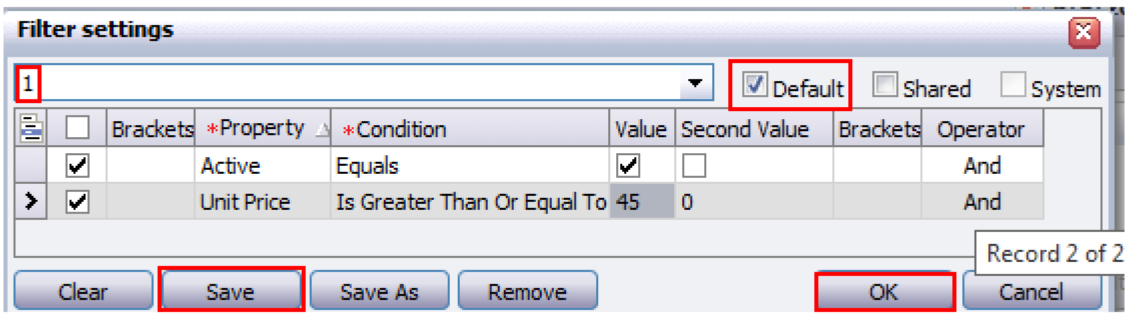


Figure: Adding the filter conditions for the default filter

6. Click **OK** to exit the **Filter Settings** dialog. Notice that records are filtered based on the filter you defined, as the screenshot below illustrates. The system displays only active products (that is, products having the **Active** status) with unit price values that are greater than or equal to $45.

   Now you can use the filter any time you open this page. If you defined the filter as the default filter for the page, the Filter action will be available (with the name of the default filter within the unlabeled field, as the screenshot below shows). If you haven't defined a default filter, the unlabeled field will be blank, and you can click the black arrow to open the list of filters available for this form and select one to apply. To add another filter, click the **Filter** icon; in the **Filter settings** dialog, click **Clear**, and add new condition lines. See also *Using Reusable Filters*.

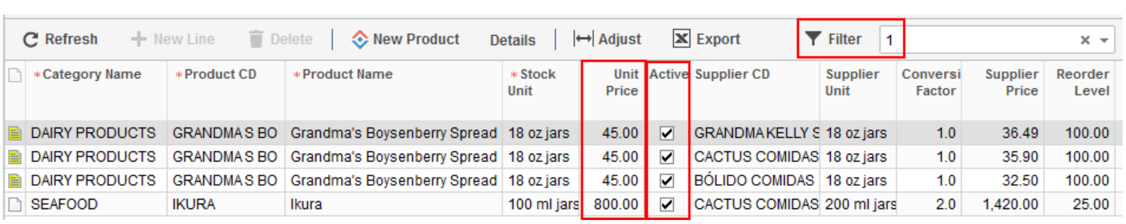


Figure: Viewing the filtered products

7. Select and manually remove the filter name so that the unlabeled field becomes blank. All the product records will again be displayed.

8. Repeatedly click the **Prev Category** button and then the **Next Category** button. Watch how the composition of product records changes in the details table based on the category.

   You can use both filtering methods simultaneously. In this case, the filtering conditions are joined with the *AND* logical operator. That is, you will see the product records that meet both sets of filtering criteria.

Creating a BLC for Implementing Filtering of Processing Webpages

The third filtering method, which provides filtering of processing pages, works within a long-running operation.

Analyze the *RenewContracts* BLC code fragment given below, which illustrates the third filtering method. For the appropriate processing webpage, this code filters the contracts that are to be closed because of expired contract dates. Further, these contracts will be processed to prepare bills for customers and change the status of the contracts. The `PXFilter<ExpiringContractFilter>` expression implements the filter based on expiring contracts that the user has selected for processing.

```
...................................
 public class  RenewContracts :  PXGraph < RenewContracts >
 {
   public PXCancel<ExpiringContractFilter> Cancel;
   public PXFilter<ExpiringContractFilter> Filter;
   public PXFilteredProcessing<ContractsList,
   ExpiringContractFilter> Items;
   public RenewContracts()
   {
     Items.SetSelected<ContractsList.selected>();
   }
   protected virtual IEnumerable items()
   {
     ExpiringContractFilter filter = Filter.Current;
     if (filter == null)
     {
       yield break;
     }
     bool found = false;
     foreach (ContractsList item  in Items.Cache.Inserted)
     {
       found = true;
       yield return item;
     }
     if (found)
     yield break;
     PXSelectBase<Contract>
     select = new PXSelectJoin<Contract, InnerJoin<ContractBillingSchedule,
      On<Contract.contractID, Equal< ContractBillingSchedule.contractID>>,
      InnerJoin<Customer, On<Customer.bAccountID,
      Equal< Contract.customerID>>>>,
      Where<Contract.isTemplate, Equal<boolFalse>,
      And<Contract.baseType, Equal<Contract.ContractBaseType>,
      And<Contract.expireDate, LessEqual<Current<ExpiringContractFilter.
      endDate>>,And<Contract.type, NotEqual<ContractType.ContractUnlimited>,
      And<Contract.status, NotEqual<ContractStatus.
      ContractStatusCanceled>>>>>>>( this);
     if (! string.IsNullOrEmpty(filter.CustomerClassID))
        select.WhereAnd<Where<Customer.customerClassID, Equal<Current
        <ExpiringContractFilter.customerClassID>>>>();
        if (filter.TemplateID !=  null)
        {
           select.WhereAnd<Where<Contract.templateID, Equal<Current
           <ExpiringContractFilter.templateID>>>>();
        }
       /* Expiring Contracts has a hierarchical structure and we
        need to show only the latest expiring node hiding all
        of its original contracts                               */
        foreach (PXResult<Contract, ContractBillingSchedule, Customer>
```

```
        resultSet in select.Select())
        {
           Contract contract = (Contract)resultSet;
           ContractBillingSchedule schedule =
           (ContractBillingSchedule)resultSet;
           Customer customer = (Customer)resultSet;
           bool skipItem =  false;
        if (contract.Type == ContractType.Expiring)
        {
           Contract child =
           PXSelect<Contract, Where<Contract.originalContractID,
           Equal<Required< Contract.originalContractID>>>>.Select
          (this.contract.ContractID);
           skipItem = child != null;
        }
        if (!skipItem)
        {
          ContractsList result new ContractsList();
          result.ContractID = contract.ContractID;
          result.Description = contract.Description;
          result.Type = contract.Type;
          result.ExpireDate = contract.ExpireDate;
          result.CustomerID = contract.CustomerID;
          result.CustomerName = customer.AcctName;
          result.LastDate = shedule.LastDate;
          result.NextDate = schedule.NextDate;
          result.ExpireDate = contract.ExpireDate;
          result.TemplateID = contract.TemplateID;
          result.Status = contract.Status;
          yield return Items.Insert(result);
       }
     }
     Items.Cache.IsDirty = false;
  }
.................................
```

Creating Lookup Fields

A lookup field represents one of the user interface (UI) elements, but unlike a text field and check box, and along with a combo box that has a drop-down list, a lookup field has a pop-up window. This window, called lookup window, is used for quick search of the required item, and may consist of arbitrary number of named columns. Any lookup window is populated with data records retrieved from the database or by using a special method declared in the code (the `PXCustomSelector` derived attribute class).

Before adding the lookup field onto a page, you have to define the structure and content of the lookup window.

You can also modify the type of an existing text or numbering field to make it a lookup field. In this case, you will have to delete and add again this field onto the page after making the appropriate modification in the field's definition code.

You can create the lookup window content through the data access class (DAC) or business logic controller (BLC) code by using the `PXSelector` or your own `PXCustomSelector` derived attribute. Columns and their order in the lookup window is defined as *typeof* parameters in an addition to the special *Search* BQL expression, by using which you can restrict displaying data.

The primary DAC in a *Search* BQL expression is also used in definition of columns' structure and their order. See below the *The Rules for Defining Lookup Columns' Structure and Their Order* section.

.

If the created lookup field is not a key field, after adding it onto the form area of the page, you can set the**CommitChanges** property for this field to *True*, if it's necessary to immediately apply selected value and force appropriate business logic execution.

Creating Lookup Columns by Using the PXSelector Attribute

By using this attribute, you can create a lookup field columns that are bound with a database,

So after choosing a field to change it to a selector field, you need to add the `PXSelector` attribute with appropriate parameters for a DAC field. The first typical selector expression for the column list creation is the following.

```
[PXSelector(typeof(Search<Accounts.accountCD>),
            typeof(Accounts.accountCD),
            typeof(Accounts.companyName),
            typeof(Accounts.country),
            typeof(Accounts.contactName),
            typeof(Accounts.contactTitle)
            )]
```

When you use the direct reference to the DAC class field, the first parameter of the **PXSelector** attribute indicates the referred DAC, and the second one, after the period, indicates the DAC field. You can refer to a DAC class type either directly or through a BQL statement. Only the first member of the `Search` expression is employed as a DAC field. The first DAC in such an expression is named primary DAC.

The simple `Search` BQL expression defines that all the records of the *Accounts* database table will be displayed on the lookup window. By using the additional `typeof()` expressions, we define columns and their order in the lookup window.

- If you are going to use a `Search` statement without any search restriction section, and without any *Join* or *OrderBy* operation, you can replace that `Search` expression with the `typeof(MyDAC.MyField)` expression. In this case, the common expression may be the following. (Notice that the `typeof(Accounts.accountCD)` field is added twice: first, to define the primary DAC name (that is name of the first DAC in the substituted `Search` expression as the first parameter) and its field as the second parameter, and second, to allocate this column as the leftmost. You could place the second `typeof(Accounts.accountCD)` field to the any needed place to change the order of this field's column. Moreover, if you don't add the primary DAC's field in the additional `typeof()` expression, this field anyway will be displayed, but its position will be the rightmost. It doesn't matter, which notation you use—see the code fragment above or below.)

```
[PXSelector(typeof(Accounts.accountCD),
            typeof(Accounts.accountCD),
            typeof(Accounts.companyName),
            typeof(Accounts.country),
            typeof(Accounts.contactName),
            typeof(Accounts.contactTitle))]
```

- If you use only the `Search` selection (or only the first `typeof()` parameter), all the fields that have the*PXUIVisibility.SelectorVisible* value of the `Visibility` parameter for the `PXUIField` primary DAC attribute are automatically included to the list of columns for the lookup window. You can include as lookup columns only fields that are specified with the*PXUIVisibility.SelectorVisible* value of the `Visibility` parameter in the primary DAC. To do so, use only the `Search` parameter or only the first `typeof()` parameter. More details concerning the `Visibility` parameter you can see in *Using the Visibility Parameter*. See also the *The Rules for Defining Lookup Columns' Structure and Their Order* section.

Use a more complicated `Search` expression, when it's necessary to restrict values of a primary DAC field, join values of a few DACs, or change sort order of this field (from *ascended* to *descended*). As the result, you get the restricted and sorted list of items in the pop-up window which can include columns from several DACs. The user can select for the webpage only the attribute value of the field in the `Search` expression, as the webpage's field is based the primary DAC's field.

This way implies mandatory adding the `PXSelector` attribute with the `Search` method as a parameter. The `Search` method gives you possibility to display data records of a lookup window which are restricted by conditions specified in a BQL expression.

For instance, you can see the code fragment of the `Account` DAC below. The condition of displaying companies in the lookup window is that each company must have the *Supplier* company type. (We assume that all companies—suppliers, customers, and other companies—are located in one database table . They are compatible as they have the similar set of fields.)

```
................................
  #region SupplierCD
  public abstract class supplierCD : PX.Data.IBqlField
  {
  }
  [PXDBString(15, IsUnicode = true)]
  [PXUIField(DisplayName = "Supplier CD")]
  [PXSelector(typeof(Search<Account.accountCD, Where<Account.companyType,
                   Equal<CompanyType.supplier>>>),
                  {typeof(Account.accountCD),
                   typeof(Account.companyName),
                   typeof(Account.country),
                   typeof(Account.contactName),
                   typeof(Account.contactTitle)})]
  public virtual string SupplierCD { get; set; }
  #endregion
................................
```

When it's needed to join several DAC data records, the common selector expression, in which the nore compicated BQL statement is used, may be written as follows. (The typeof() additional expression isn't used in this example. but the optional `DescriptionField` parameter is used.)

```
  [PXSelector(typeof(Search2<VendorClass.vendorClassID,
            LeftJoin<EPEmployeeClass, On<EPEmployeeClass.vendorClassID,
            Equal<VendorClass.vendorClassID>>>>),
            DescriptionField = typeof(VendorClass.descr))]
```

As a result, the lookup field with two columns is created, **VendorClassID** and **Description**. If the `VendorClass` primary DAC comprises fields with the *PXUIVisibility.SelectorVisible* `Visibility` parameter value, all these fields will be displayed as columns of the created lookup field along with the aforementioned two columns. Anyway, in this case the **VendorClassID** will be displayed as the leftmost column, while the **Description** field—as the rightmost one. All the selector fields with the *PXUIVisibility.SelectorVisible* **Visibility** value will be displayed as columns located between the **VendorClassID** and the **Description** columns in order of their declaration.

You can create a selector whose columns comprise fied values of several DACs, and also define any other column order. (See the following code fragment.)

```
  [PXSelector(typeof(Search2<VendorClass.vendorClassID,
            LeftJoin<EPEmployeeClass, On<EPEmployeeClass.vendorClassID,
            Equal<VendorClass.vendorClassID>>>>),
            typeof(EPEmployeeClass.paymentMethodID),
            typeof(VendorClass.vendorClassID),
            typeof(VendorClass.cashAcctID),
            typeof(EPEmployeeClass.salesAcctID),
            DescriptionField = typeof(VendorClass.descr))]
```

In this code, fields of two DACs, `VendorClass` and `EPEmployeeClass`, have been included as columns in the selector. The key field **VendorClass.vendorClassID** will be displayed not as the leftmost, but as the second column from the right of the selector pop-up window,

The `DescriptionField` parameter, which is not a mandatory parameter, indicates the hint field associated with the lookup field; this hint provides a description of the selected item, if applicable, in the lookup window and within the field box. (The description field text is displayed both within the webpage field and in a separate column of the lookup window.)

You can use the `SubstituteKey` parameter to replace the surrogate key with natural one to display more informative key value, particularly, in the lookup window: instead of the surrogate key column, the natural key column can be used. See *Using Substitute Keys* for details.

In the code fragment below, the example of usage the `SubstituteKey` parameter (along with the `DescriptionField` parameter) is shown.

```
[PXSelector( typeof(Search2<FABook.bookID, InnerJoin<FABookBalance,
             On<FABookBalance.bookID, Equal< FABook.bookID>>>,
             Where<FABookBalance.depreciate, Equal<boolTrue>>>),
             SubstituteKey = typeof(FABook.bookCode),
             DescriptionField = typeof(FABook.description))]
```

As a result, the lookup field with minimum two columns is created: **BookID** and **Description**.
If the `FABook` primary DAC comprises fields with the *PXUIVisibility.SelectorVisible* `Visibility` parameter value, all these fields will be displayed as columns of the created lookup field along with the aforementioned two columns.

Instead of the surrogate **BookID** key field, the **BookCode** key field will be displayed on the lookup field.

Data in this lookup field is restricted with conditions that only **FABook.bookID** books are displayed, which have the IDs in the *FABookBalance* book database table, and are to be depreciated, while number of items equal the minimum number of the records containing such **BookID** values in the *FABookBalance* or in the *FABook* database table, as we used the *InnerJoin* operator.

See the *Adding Lookup Fields Onto a Form and Onto a Grid*, where the consequent actions of adding lookup fields onto the page are described.

The Rules for Defining Lookup Columns' Structure and Their Order

To properly construct required columns of a lookup field so that all the columns were placed in the needed order and contain only the data necessary for users, you should stick to the following rules:

1. Any `PXSelector` attribute's expression consist of a `Search` statement (the mandatory part) and additional `typeof()` part (the optional part). The mandatory part may be represented by a `Search` BQL statement or by a `typeof(MyDAC.MyField)` expression, where **MyDAC.MyField**—the primary DAC's name (before the dot) and the name of this DAC's field (after the dot).
2. If you are going to use a `Search` statement without any search restriction section, and without any Join and Order operation, you can replace that `Search` BQL statement with a `typeof(MyDAC.MyField)` expression.
3. Don't use the additional `typeof()` part of the selector expression to automatically display the *SelectorVisible* fields of the primary DAC as the lookup field's columns; otherwise, these fields are not displayed. The order of the columns straightly depends on the order of the fields declaration in the primary DAC. The primary DAC's field of the Search expression (or in the first `typeof`), or its suvstitute key field, will be displayed in any case.
4. If you use the additional `typeof()` part of the selector expression, notice that all the columns to be displayed must be listed in this part, including primary DAC's field (or the field in the first `typeof`) . Exception: the primary DAC field (or its substitute field), if this field is not listed in the additional `typeof()` part of the selector expression, will ever be displayed as a lookup field's column.
5. Define the order of columns (from the left to the right) by the corresponding order of the additional `typeof()` part of the selector expression.

6. The primary DAC's field (or the field in the first `typeof`) will be displayed as the rightmost lookup field's column, if it hasn't been listed in the additional `typeof()` of the selector expression. Otherwise, this field will be displayed in order, in which it has been listed.
7. If the `DescriptionField` is defined, and this field is not listed among the *SelectorVisible* fields or in the additional `typeof()` part of the selector expression, the appropriate column will be added to the right side of the lookup window, but as the second column at the right, if the primary DAC's column is to be added as a rightmost column.
8. If the `SubstituteKey` parameter is used. the natural key field replaces the surrogate key value in every case.

Creating Lookup Columns by Using the PXCustomSelector Attribute

By using this attribute, you can also create a lookup field columns. Instead of a `Search` expression, the `GetRecords()` method is used,

After generating the required DAC, you can add the `PXCustomSelector` attribute with appropriate parameters to the DAC field code.

The first example illustrates development and use of the `PXCustomSelector` attribute of the lookup field with an unbound lookup column. (See the code fragments below.)

```
...................
  [AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
  public sealed class DaylightSelectorAttribute : PXCustomSelectorAttribute
  {
      public DaylightSelectorAttribute()
                : base(typeof(Year.nbr), typeof(Year.nbr))
      {
      }
      public IEnumerable GetRecords()
      {
          var currentYear = DateTime.Today.Year;
          const int range = 30;
          var start = currentYear - range;
          var end = currentYear + range;
          for (int i = start; i < end; i++)
                yield return new Year { Nbr = i };
      }
  }
...................
```

The `DaylightSelector` attribute defined as a class that inherits from the `PXCustomSelector` attribute, has been created to provide a lookup field's column with the range of years. This range is defined by using the `for` cycle, `range` constant, and value of the `Year` variable. The `DaylightSelector` class derived from the `PXCustomSelectorAttribute` was created to provide a lookup field populated with a list of years that are less or more by 30 than the current one.

The next code fragment illustrates attaching the `DaylightSelector` attribute to the `Year` field of the `DaylightShiftFilter` DAC.

```
...................
  [Serializable]
  [PXCacheName(Messages.CalendarYear)]
  public partial class DaylightShiftFilter : IBqlTable
  {
     #region Year
     public abstract class year : IBqlField
     {
     }
     [PXInt]
     [PXUIField(DisplayName = "Year")]
     [CurrentYearByDefault]
     [DaylightSelector]
     public virtual int? Year { get; set; }
```

```
     #endregion}
...................
```

The user will be able to select a year, that is less or more by 30 than the current one. In accordance with this code example, the displaying year range will depend on the current client operational system year.

The second example illustrates development and use of the `PXCustomSelector` attribute of the lookup field with bound lookup columns. (See the code fragments below.)

```
...................
  public class CustomerPriceClassAttribute : PXCustomSelectorAttribute
  {
     public CustomerPriceClassAttribute()
             : base(typeof(AR.ARPriceClass.priceClassID))
    {
             this.DescriptionField = typeof(AR.ARPriceClass.description);
    }
    protected virtual IEnumerable GetRecords()
    {
             AR.ARPriceClass epc = new PX.Objects.AR.ARPriceClass();
             epc.PriceClassID = AR.ARPriceClass.EmptyPriceClass;
             epc.Description = Messages.BasePriceClassDescription;
             yield return epc;
             foreach (AR.ARPriceClass pc in PXSelect<AR.ARPriceClass>.
                                                 Select(this._Graph))
                  {
                      yield return pc;
                  }
    }
  }
...................
```

The `CustomerPriceClass` attribute, which is also defined as a class that inherits from the `PXCustomSelector` attribute, has been created to provide a lookup field's columns with the price class and their descriptions, obtained from the `ARPriceClass` DAC by using the `foreach` cycle.

The next code fragment illustrates implementing the `CustomerPriceClass` attribute by adding it to the `SalesPriceFilter` DAC code for the `CustPriceClassID` selector field.

```
...................
  [Serializable]
  public partial class SalesPriceFilter : IBqlTable{
  ...................
  #region CustPriceClassID
  public abstract class custPriceClassID : PX.Data.IBqlField
  {
  }
  [PXDBString(10, InputMask = ">aaaaaaaaaa")]
  [PXDefault(AR.ARPriceClass.EmptyPriceClass)]
  [PXUIField(DisplayName = "Customer Price Class",
       Visibility = PXUIVisibility.SelectorVisible)]
  [CustomerPriceClass]
  public virtual string CustPriceClassID { get; set; }
  #endregion
...................
```

While in the first example the explicitly defined columns are employed, in the second example the SelectorVisible columns will be displayed in the pop-up window.

The user will be able to select the required customer price class from the lookup field after you add this selector field onto the page and compile the project. In accordance with this code example, two columns will be displayed in the selector field: **PriceClassID** and **Description**, as they have the `Visibility` property set to *SelectorVisible*.

Adding Lookup Fields Onto a Form and Onto a Grid

A lookup (or selector) field is a standard user interface (UI) element that is used for quick search of the required item value through a webpage field or details table column element. Searched items are displayed on the popup window that includes one or more columns with data.

Before adding lookup fields, you should create them by modifying the code of the appropriate data access class (DAC) or business logic container (BLC). Creating process of a lookup field and typical selector expression structures are described in details in *Creating Lookup Fields*.

Adding a Lookup Field Onto a Form

Suppose that you have created the lookup field's code for the *Employees* webpage, which already has UI elements, including the **EmployeeCD** simple text field that is to be transformed to a selector field.

In this case, your typical actions may be the following:

1. Open the *Employees* page, right-click any area of the page, and select *Refresh*.

   If this page was already opened, the refresh procedure lets you retrieve the changes you have made during the first adding UI elements onto the page.

2. Point to the **form** control, open the smart tag associated with it, and select **Edit Content Layout**.
3. On the left area of the Layout Editor that appears, delete the **EmployeeCD** field by clicking **Remove active item**.

   First you should do before adding a lookup (selector) field—remove the same field that existed before as a text or numeric field.

4. On the right area of the Layout Editor, select the **Fields** tab, and you can see the **EmployeeCD** field, defined as a *Selector* control (that is, as a lookup field).
5. Select the check box for the **EmployeeCD** field and click **Generate**.
6. On the left area of the Layout Editor, move up by one position the restored **EmployeeCD** field to place it in its original position.
7. On the **Properties** tab, open the drop-down list for the **DisplayMode** property to see the options, but keep the *Hint* default value, as shown in the screenshot below.

   The **DisplayMode** property defines the display format of the lookup field value on the webpage and within the lookup window during run time. The property has the following settings: *Value*: If you use this mode, you can see in the webpage field only the employee CD (the first 15 letters of the employee's last name in this case), and in the lookup window you see two columns—one with the employee CD, and for the other the **DescriptionField** property is used. *Text*: If you use this mode, in the webpage field, you see only the description field's name, and in the lookup window, you see two columns: one with the employee CD, and the other with the employee's description. This mode is used when the field value is calculated, such as a numbered key value (defined as an **Identity** field) or, for instance, the full name of an employee (as the description field). To allow the user to add a calculated value for a non-nullable field, you must also set the **TextMode** property to *Editable*. *Hint*: If you use this mode, on the webpage field box and in the lookup window, you can see two values: the employee CD and the employee's full name.

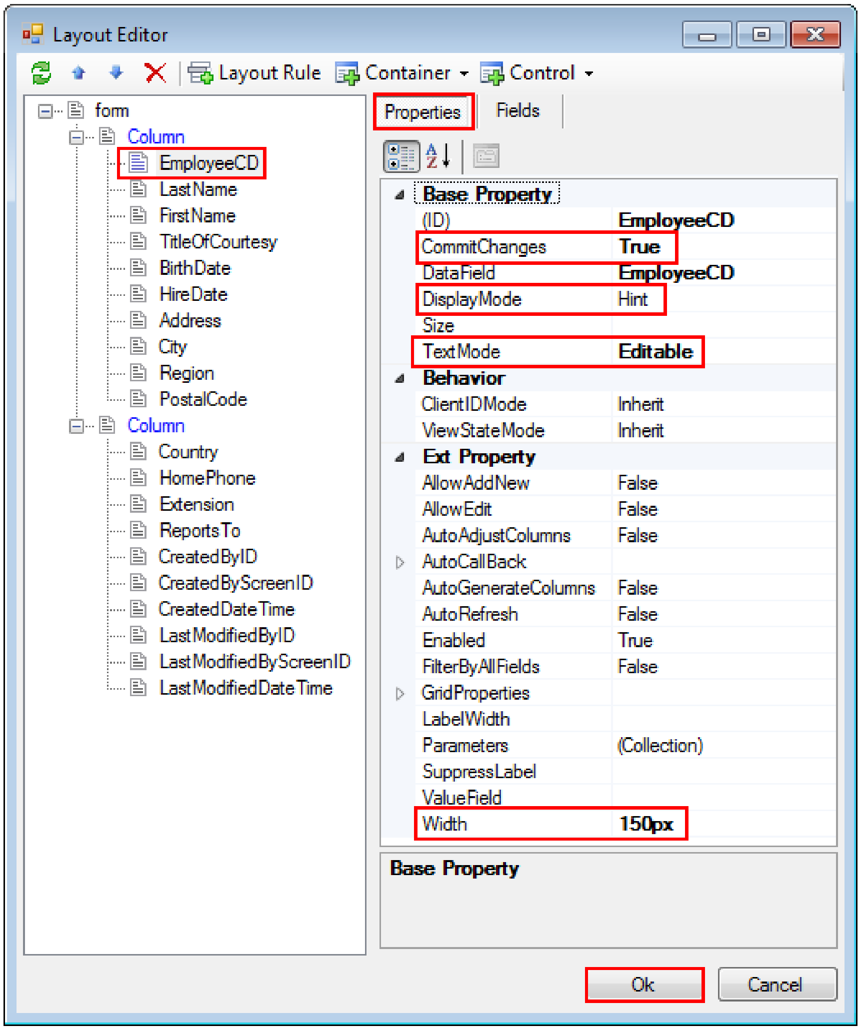


Figure: Adjusting properties of the lookup field

8. For each lookup field, set the value of **CommitChanges** property to *True*.
9. Optional: Enter the optimal **Width** property value.
10. Click **OK** to close the Layout Editor window.
11. Select the source mode to see the .aspx code; notice that the **EmployeeCD** lookup field's tag has been created—*PXSelector*— which contains entered property values. (See the screenshot below.)

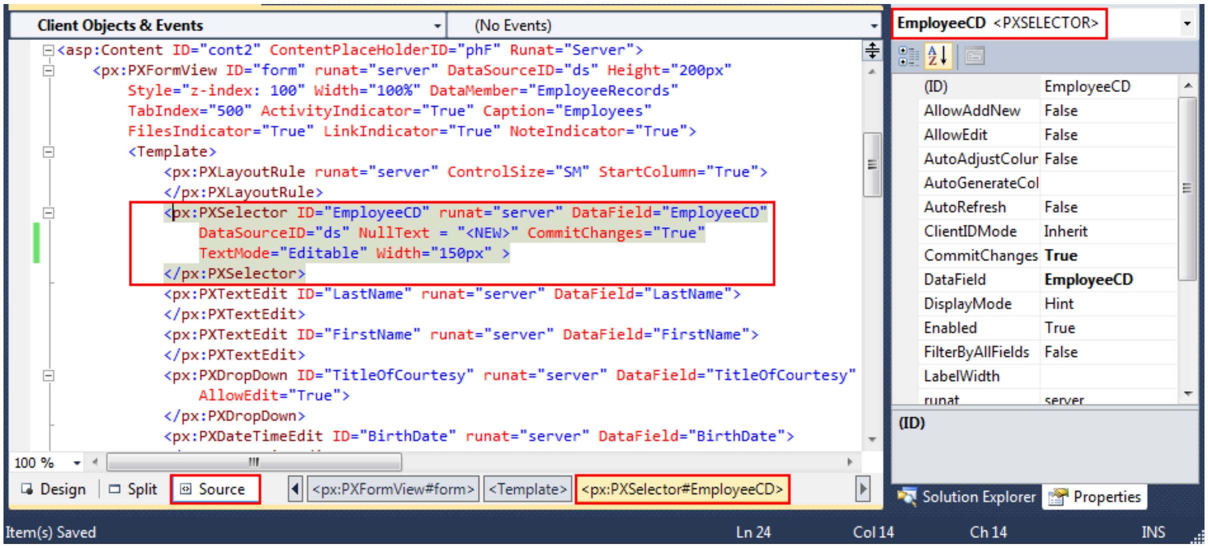


Figure: Analyzing the PXSelector tag's content

12. Start the application with the *Employees* webpage, open (or refresh the page if it's already open), click **Insert**, and add another employee record. Click **Save** to save the entered record. Click navigation buttons to select existing records and watch their attribute values. Notice that in the *Hint* display mode (as in the *Text* mode), in the **EmployeeCD** field box, the employee CD is displayed, followed by a hyphen and the employee's full name (the description field), as shown in the screenshot below.

In the describing example, the system automatically capitalizes all letters entered in the **EmployeeCD** field and trims all letters past the 15th letter on the right. Because blanks on the left are never trimmed, we recommend that you not add blanks left of the **EmployeeCD** value.

13. Click the Magnifier icon of the **EmployeeCD** field. You see the drop-down list with the CDs and full names of employees, as the screenshot below illustrates.

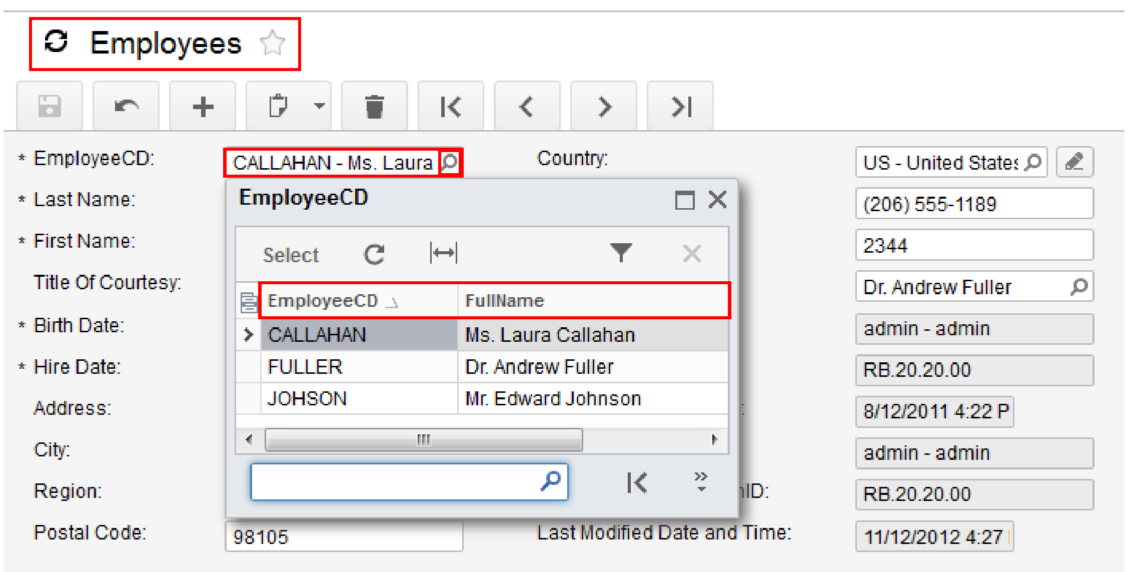


Figure: Viewing the structure of the EmployeeCD lookup field

Adding a Lookup Field Onto a Grid

Calculations

In this chapter, you will get acquainted with the various types of calculations, including calculations by using formulas, autonumbering, and calculation by using accumulator attributes. Topics of this chapter also contain descriptions of how to handle concurrent and frequent field updates.

Content

This chapter covers the following topics:

Calculating Values of UI Elements

To implement the calculation of values, you use the following attributes:

- `PXDBCalced`, which creates an equation in a final T-SQL statement, is used for unbound data access class (DAC) fields.
- `PXDBScalar`, which declares a sub-query in a final T-SQL statement, also is used for unbound DAC fields.
- `PXFormula` performs various types of calculations, including totals, and is used for both database-bound and unbound DAC fields.
- `PXUnboundFormula` is used for unbound DAC fields. It performs aggregate calculations depending on one or more conditions and assigns results to one or more summary fields.

In many cases, the *FieldSelecting* event handler is raised when a DAC field value is being prepared to be displayed on the UI. This event should be used to calculate database-unbound DAC field values whose calculation methods can not be specified declaratively. For detailed information, see *FieldSelecting Event*.

Calculating With PXDBCalced

By using the `PXDBCalced` attribute, you can perform calculations with four standard arithmetical operators: addition (*Add*), subtraction (*Sub*), multiplication (*Mult*), and division (*Div*). The attribute also provides the *Minus* operator, which you can use to change a negative decimal result to a positive one and a positive result to a negative one. You can see the list of all operands in *PXDBCalced Attribute*.

For example, see the following DAC code fragment, where the **Discrepancy** field is used to define the quantity of products to be reordered. The second parameter is used to define the data type of the result.

```
[PXDBCalced(typeof(Minus<Sub<Sub<ProductReorder.unitsInStock,
             ProductReorder.unitsOnOrder>, ProductReorder.reorderLevel>>),
                                                  typeof(Decimal))]
```

Calculating With PXDBScalar

The `PXDBScalar` attribute declares a sub-query, which you can use to obtain the result of a BQL statement.

By using the following DAC code fragment, you can obtain the quantity of the specified product in stock.

```
[PXDBScalar(typeof(Search<StockBalance.unitsInStock,
     Where<StockBalance.productID, Equal<Products.productID>>>))]
```

By using the DAC code fragment that follows, you can get an array of the current product's **Supplier Price** values of different suppliers, sort the values from the lowest to the highest price, and return the value with the lowest price.

```
...............................................
      #region SupplierPrice
      public abstract class supplierPrice : PX.Data.IBqlField
      {
      }
      [PXDecimal(2)]
      [PXUIField(DisplayName = "Supplier Price")]
      [PXDBScalar(typeof(Search<SupplierProduct.supplierPrice,
        Where<SupplierProduct.productID, Equal<ProductReorder.productID>,
            And<SupplierProduct.supplierPrice, Greater<decimal_0>>>,
            OrderBy<Asc<SupplierProduct.supplierPrice>>>))]
      [PXDBDefault(typeof(Search<SupplierProduct.supplierPrice,
         Where<SupplierProduct.productID, Equal<Current
          <ProductReorder.productID>>, And<SupplierProduct.supplierCD,
                          Equal<Current<ProductReorder.supplierCD>>>>>))]
      public virtual decimal? SupplierPrice { get; set; }
      #endregion
...............................................
```

Calculating Column and Total Values With PXFormula

This section illustrates the `PXFormula` calculation attribute by using the *Sales Orders* webpage, which is based on the *FormDetails* template.

`PXFormula` is used to declare various kinds of formulas for calculation of DAC field values, such as discounts, extended prices, line totals, and other values you might need to calculate. The `PXFormula` attribute provides calculations by using four standard arithmetical operators: addition (*Add*), subtraction (*Sub*), multiplication (*Mult*), and division (*Div*). A few aggregate methods can be used by the `PXFormula` attribute as a parameter: *SumCalc*, *CountCalc*, *MinCalc*, and *MaxCalc*.

Three typical code examples with different structures are given below. The second and third examples do not permit the user to add any value to the formula, since all the values are to be calculated. The first example permits the user to enter values to pass them for calculations of aggregates.

The `PXParent` attribute, illustrated below, provides a master-details relationship between the upper and lower areas of the webpage. The total field values in the master area change as lines in the details table are inserted or updated, based on values in the columns of the details table.

```
[PXParent ( typeof(Select<Order,Where<Order.orderID,
          Equal<Current<OrderDetails.orderID>>>>))]
```

It doesn't matter on which field the `PXParent` attribute was declared. The first `PXParent` attribute found will be used with the DAC defined for this aggregate. This attribute works only with the first and second code examples showing the usage of the `PXFormula` attribute.

For the first example, shown below this paragraph, a simple expression with one parameter is illustrated. It calculates only the aggregate value in the **TotalQty** field by using the `PXFormula` attribute; the total quantity of the current receipt is defined each time the user saves inserted or updated data.

`[PXFormula(null, typeof(SumCalc<Documents.totalQty>))].`

The second example (shown below this paragraph) shows a more complicated expression with two parameters. This formula, declared for the **Extended Price** column of the details table, updates the **Lines Total** value in the form area of the webpage with the sum of the **Extended Price** column rows, whose DAC field (`ExPrice`) is used as a parameter of the `PXParent` attribute. (See the screenshot and the note below.) The formula also updates for each row the **Extended Price** value, which is calculated by multiplying the following numbers: the value of the **Unit Price** column, the value of the **Quantity** column, and the result when the **Discount** column value (the percent divided by 100) is subtracted from 1.

```
[PXFormula(typeof(Mult<Mult<OrderDetail.unitPrice, OrderDetail.quantity>,
         Sub<DecimalOne, Div<OrderDetail.discount, DecimalHundred>>>),
           typeof(SumCalc<Order.linesTotal>))]
```

Thus, if the unit price was $55.00, the quantity was 42.00, and the specified discount percent was 10.00, the extended price would be calculated as follows: $55.00 * 42.00 * (1 – 10.00/100) = $2079.00, as the screenshot below illustrates.

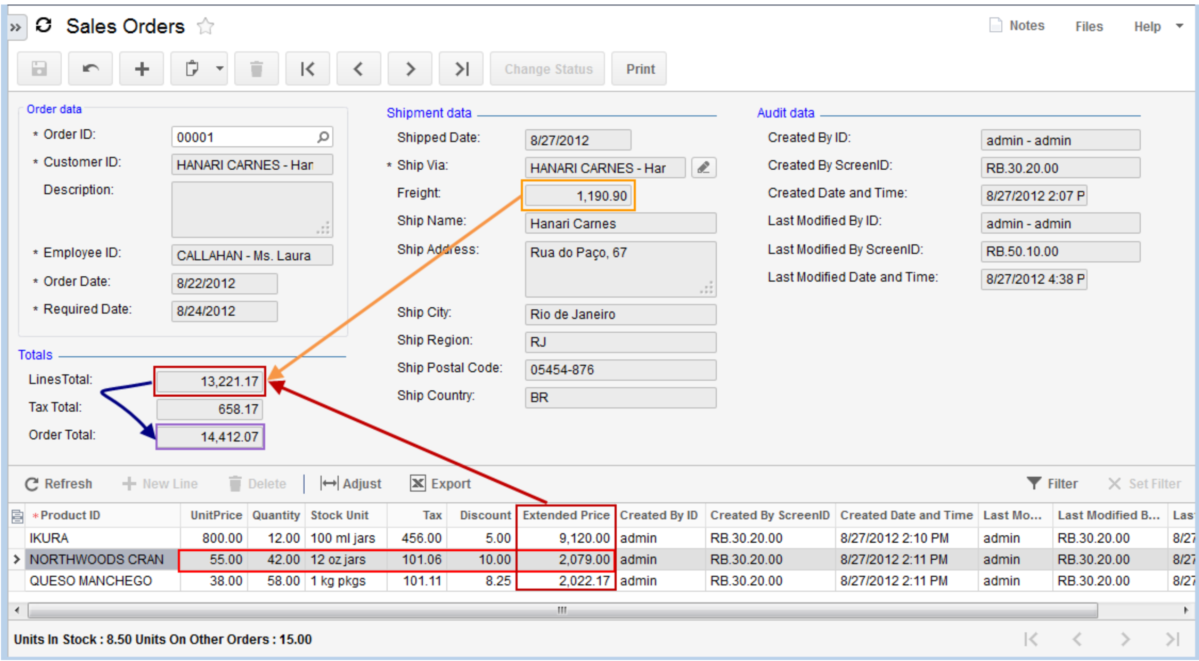


Figure: Calculation of sales order totals

In the code fragment shown in the second example, note the following:

- The `DecimalOne` and `DecimalHundred` classes represent the constants that equal 1 and 100, respectively. These constants, declared earlier, are used in the `PXFormula` expression to calculate the coefficient by which product costs are multiplied. Users enter discount values as percentages; the entered discount percent is then divided by 100.

For the third example (shown below this paragraph), the simplest expression with one parameter is illustrated, with the static formula, declared for the **Order Total** field. This formula updates the order total amount with the sum of **Lines Total** and **Freight**. (See also the screenshot above.)

```
[PXFormula(typeof(Add<Order.linesTotal, Order.freight>))]
```

Calculating Aggregate Values With PXUnboundFormula

The `PXUnboundFormula` attribute, which is mostly used with the `Switch` operator, lets you obtain aggregate results and assign them to the respective summary webpage fields. As a first parameter of this attribute, the BQL expression (usually with the `Switch` operator) is used, while in the second parameter, the *SumCalc* aggregate method is used along with the summary field name. The `PXUnboundFormula` attribute may be added to any DAC field code, since the destination field does not depend on the field chosen for this attribute. The destination summary field is specified in the second parameter of the attribute, which is added after the `SumCalc` aggregate method.

You can see a DAC code fragment that uses the `PXUnboundFormula` attribute below. Note that several **PXUnboundFormula** attributes have been added to the **Taxable Amount** field definition. Also, notice that the **Taxable Amount** field value does not depend on the results of the calculations of the `PXUnboundFormula` attributes. These results will be entered to the summary fields that are defined in the second parameter of each attribute.

```
...............................................
#region CuryTaxableAmt
public new abstract class curyTaxableAmt : PX.Data.IBqlField
{
}
[PXDBCurrency(typeof(APTaxTran.curyInfoID), typeof(APTaxTran.taxableAmt))]
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Taxable Amount", Visibility = PXUIVisibility.Visible)]
[PXUnboundFormula(typeof(Switch<Case<WhereExempt<APTaxTran.taxID>,
                             APTaxTran.curyTaxableAmt>, decimal0>),
                                typeof(SumCalc<APInvoice.curyVatExemptTotal>))]
[PXUnboundFormula(typeof(Switch<Case<WhereTaxable<APTaxTran.taxID>,
                                        APTaxTran.curyTaxableAmt>, decimal0>),
                              typeof(SumCalc<APInvoice.curyVatTaxableTotal>))]
[PXUnboundFormula(typeof(Switch<Case<WhereExempt<APTaxTran.taxID>,
                                          APTaxTran.curyTaxableAmt>, decimal0>),
             typeof(SumCalc<AP.Standalone.APQuickCheck.curyVatExemptTotal>))]
[PXUnboundFormula(typeof(Switch<Case<WhereTaxable<APTaxTran.taxID>,
APTaxTran.curyTaxableAmt>, decimal0>), typeof(SumCalc<AP.Standalone.
                                 APQuickCheck.curyVatTaxableTotal>))]
public override decimal? CuryTaxableAmt { get; set }
}
#endregion
...............................................
```

Data Input

In this chapter, you will get acquainted with the specific singularities of data input support and various types of data manipulation by using Acumatica Framework tools and facilities. Topics of this chapter also contain descriptions of how to import data from external files, validate field values, add input masks.

Content

This chapter covers the following topics:

Managing Visibility of DAC Fields and UI Elements

You can manage visibility of a DAC field in the appropriate section of the Layout Editor window, and a user interface (UI) element—such as a field, combo box, check box—on a webpage.

Using the Visibility Parameter

In this section is described the managing of a data access class (DAC) field visibility in the appropriate segment of the Layout Editor window (on the **Fields** tab).

Layout Editor is used to adjust each UI element properties and append them onto a page while working in design mode. Each visible DAC field must have its `PXUIField`—DAC field attribute. This attribute may have parameters, one of which predefines visibility of a DAC field in one of segments of the Layout Editor window: *Visible*, *Invisible*, or *Selector*. The capability of splitting UI elements into different segments facilitates creation of a webpage and enables the developer to quickly analyze correctness of the DAC code (for instance, not to forget to define a DAC field in the DAC code as a selector (lookup) field).

See below the *Country* DAC code fragment for an example of usage parameters of the `PXUIField` attribute.

```
..........................
   public abstract class country : PX.Data.IBqlField
   {
   }
   [PXDBString(2, IsKey = true, IsUnicode = true, IsFixed = true)]
   [PXDefault()]
   [PXUIField(DisplayName = "Country", Visibility = PXUIVisibility.SelectorVisible)]
   public virtual string Country { get; set; }
...............................
```

The `PXUIField` attribute denotes the appearance of the DAC field within appropriate segment of the Layout Editor. The `DisplayName` parameter specifies the name of the UI element on the interface. The `Visibility` parameter specifies the visibility scope of the UI element and has four possible values:

- *PXUIVisibility.Visible*: Indicates that the DAC field is to be included in the *Visible* segment of the Layout Editor window. If the `PXUIField` attribute is added for a field without the `Visibility` parameter, this DAC field becomes visible by default for Layout Editor.
- *PXUIVisibility.Invisible*: It means that the DAC field is to be included in the *Invisible* segment of the Layout Editor window. If the `PXUIField` attribute is not added for a field, this field also is included in the *Invisible* segment of Layout Editor.
- *PXUIVisibility.SelectorVisible*: Indicates that the DAC field is to be included in the *Selector* segment of the Layout Editor window to use it for generation the selector (lookup) field or column. You can use such fields as columns of a lookup field when this field has no explicit set of columns specified.
- *PXUIVisibility.Dynamic*: It means that a DAC field bound to a grid control is not visible in any section of the Layout Editor window. You can use such DAC fields to automatically display them in a details table or tab table as columns of a webpage, if you add no columns onto the page and set the **AutoGenerateColumns** property value to *AppendDynamic*.

Using the Visible Parameter

This is a static way of the UI element visibility management. The following code fragment of a business logic container (BLC) code illustrates the use of this parameter.

```
..................................
  #region DAC Overrides
  [PXDBString(1, IsKey = true, IsUnicode = true, IsFixed = true)]
  [PXUIField(DisplayName = "Company Type", Visible = false)]
  [PXDefault(CompanyType.Supplier)]
  public virtual void Accounts_CompanyType_CacheAttached(PXCache Sender){}

..................................
```

You made the **Company Type** field invisible by adding `Visible = false` in the *DAC Overrides* region of a BLC code.

The next code fragment of a DAC code illustrates making invisible of a special system grid column, **LastLineNumber**, whose value is used by the appropriate BLC logic, but is not needed for the user's work.

```
..................................
  #region LastLineNbr
  public abstract class lastLineNbr : PX.Data.IBqlField
  {
  }
  [PXDBInt()]
  [PXUIField(Visible = false)]
  public virtual int? LastLineNbr { get; set; }
  #endregion
```

```
  #region NoteID
  public abstract class noteID : PX.Data.IBqlField
 ..................................
```

The `Visible` parameter has an alternative—`Enabled` parameter, which is used when instead of making a UI element invisible, is necessary to make it visible, but non-editable.

Using the SetVisible Method

The `PXUIField` attribute class enables dynamic modification of `PXUIField` attribute parameters. Here, the *SetVisible* method is used by the event handler to override the `Visible` parameter when data is selected from the DAC.

```
..................................
   public class EmployeeList : EmployeeMaint
   {
      public virtual void Employees_RowSelected(PXCache cache,
                                        PXRowSelectedEventArgs e)
    {
       if  (e.Row != null)
      PXUIFieldAttribute.SetVisible<Employees.employeeCD>(cache, e.Row, false);
    }
  }
..................................
```

The `PXUIFieldAttribute.SetVisible` method sets the `Visible` parameter of the appropriate `PXUIField` attribute to *false* at run time. If you don't supply a field name, this method affects all fields of the DAC.

> The `PXUIFieldAttribute.SetVisible` method overrides the default value of the `Visible` parameter specified in the DAC. Therefore, if you apply this method to the entire DAC and must make invisible some fields under certain conditions, you should explicitly make invisible these fields.

The next code fragment of the *APInvoiceEntry* BLC code illustrates making invisible of a form UI elements and grid columns, `CuryOrigDocAmt` and `Box1099`, appropriately in the invoice (if the `RequireControlTotal` property in the AP setup is set to *False* or the document has not been released), and in the *Transactions* grid (if the **Vendor1099** value is *False*).

```
..................................
protected virtual void APInvoice_RowSelected
(PXCache cache, RowSelectedEventArgs e);
{
 APInvoice doc = e.row as APInvoice;
..................................
PXUIFieldAttribute.SetVisible<APInvoice.curyOrigDocAmt>(cache,
doc, (bool)APSetUp.Current.RequireControlTotal || docreleased);
PXUIFieldAttribute.SetVisible<APTran.box1099>
(Transactions.Cache, null, Vendor1099);
..................................
```

> Only the `RowSelected` handler on a PrimaryView DAC's BLC code or a BLC constructor are places where is possibly to modify visibility through the code.

Validating UI Element Values

In this topic, the process of implementing a simple validation logic for user interface (UI) elements is described. Validation logic is necessary to prevent entering wrong or inadmissible values to user interface (UI) elements, as well as values that do not match the conditions that are specified beforehand. As a rule, validation logic is implemented by using various kinds of event handlers.

Implementing a Simple Validation Logic

Suppose that you must restrict UI element values of your *Employees* webpage, whose **General Info** tab includes data sections of more than one data access class (DAC). The **Hire Date** UI element (the date type field) had been included in the *EPEmployee* DAC, while the **Date Of Birth** UI element (also the date type field) had been included in the *CRContact* DAC (see the screenshot below). The **Date Of Birth** field must have not null or empty (blank) values; values of the **Hire Date** must match the condition: the age of the employee cannot be less than 16 years.

It doesn't matter, in a common or in different DACs are allocated UI elements that are to be bound by a condition; the illustrated situation with different DACs is a bit more complicated, and nothing more.

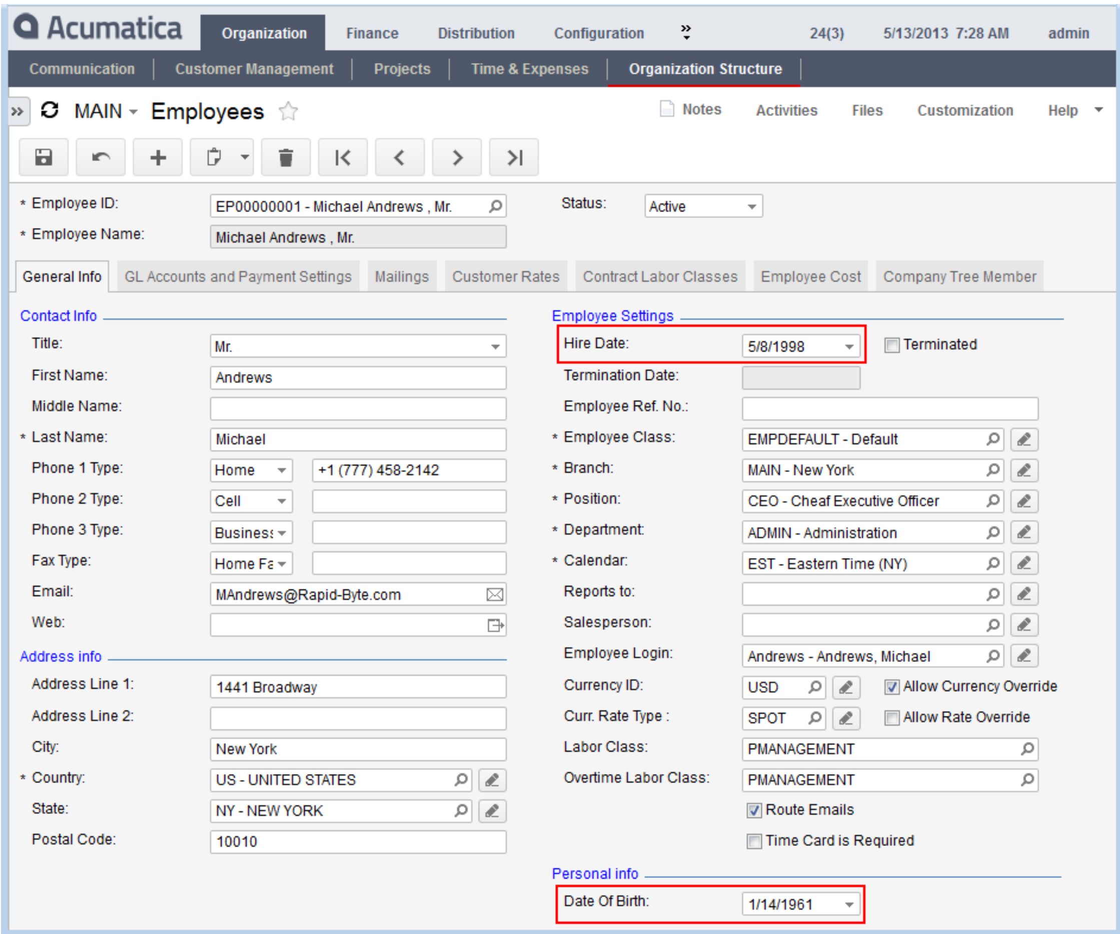


Figure: The UI elements to be validated

(You shouldn't perform these instructions, just analyze the code lines.) To implement this validation logic, proceed as follows.

1. Add to the *EPEmployeeEvents* region of the *EP.EmployeeMaint* business logic container (BLC) code the following code lines.

```
#region EPEmployee Events
.........................................................................
protected override void EPEmployee_RowPersisting(PXCache sender,
                                                PXRowPersistingEventArgs e)
{
  PXDefaultAttribute.SetPersistingCheck<Contact.dateOfBirth>
                            (sender, e.Row, PXPersistingCheck.NullOrBlank);
  DateTime birth = (DateTime)this.Contact.Current.DateOfBirth;
  EPEmployee row = (EPEmployee)e.Row;
  DateTime alloweddate = new DateTime(birth.Year + 16,
                birth.Month, birth.Day);
  DateTime hire = (DateTime)row.HireDate;
  if (hire != null &&((DateTime)hire) < alloweddate)
```

```
  throw new PXSetPropertyException("The employee's hire date must be " +
                               "at least 16 years after his or her
birthdate.");
}
#endregion
..........................................................................
```

Within the *RowPersisting* event code, two methods of a field validation are used: The *PXDefaultAttribute.SetPersistingCheck* method, which is used to remind the user to enter the appropriate date of birth. (You can tweak the validation process by using the **PXPersistingCheck** parameter values (*Null*, *NullOrBlank*, or *Nothing*.) The following code lines, which (along with the *PXSetPropertyException* method) checks the condition to warn the user if the new employee is younger than 16. These validation methods prevent a record from being saved if at least one of the aforementioned conditions is true. If the date of birth is null or empty, the common error message is displayed (such as *Nullable object must have a value*), but you can use the *PXSetPropertyException* method to declare your own detailed error message by using the second validation version.

2. Set the *AutoCallBack* properties for the **Hire Date** field as follows:
    - **Enabled**: *True* (keep default)
    - **Target** : *form*
    - **Command**: *Save*
3. Build the solution.

Testing the Results

Now you can test the results of the implemented validation logic to ensure that the logic works properly.

(You shouldn't perform these instructions, just imagine the testing steps.) Perform the following actions:

1. Return to the *Employees* form and try to add a new employee record without entering the **Date Of Birth** value. Enter values for all the other required fields (allocated by the asterisk at the left of the name).
2. Click **Save**: The error message appears that the not nullable object must have a value, and the record is not saved.

    As was mentioned in the hint in the previous section, to define a more exact error message, you can add on your own a few more customization code lines to the *EP.EmployeeMaint* BLC code lines that contain the appropriate condition check and error message text.
3. Enter the date of birth so that the difference between it and the hire date is less than 16 years, and the second error message appears, as shown in the screenshot below. This is the message text added by you to the event code as a parameter of the *PXSetPropertyException* method.

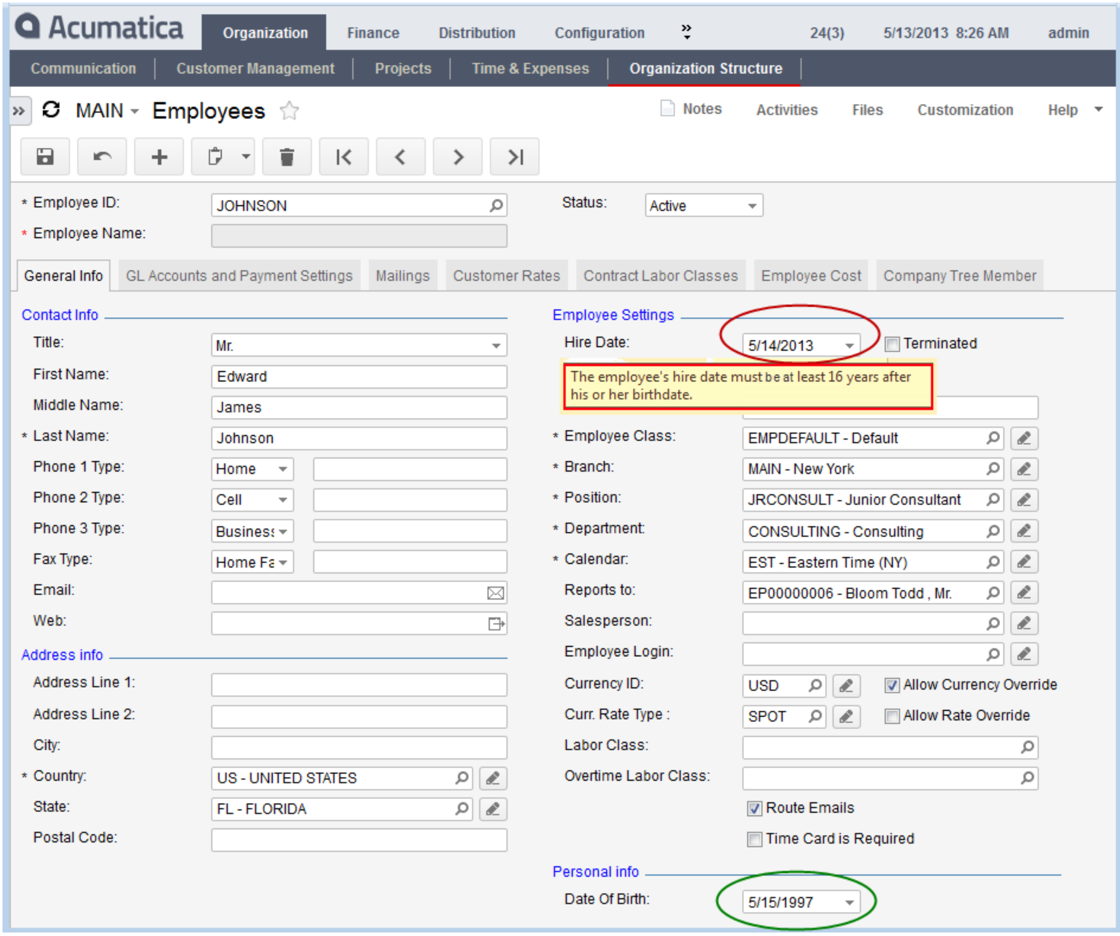


Figure: Entering a record with the not permitted Hire Date value

4. Make the hire date at least 16 years later than the date of birth, and click **Save**. The new record has been saved.

Further in your practice, you will possibly have to implement more complicated validation logic: For instance, logic which provides blocking of the user's data entering (in the multi-user mode) when one or more dynamically changed values of a group of fields can disturb the defined threshold value (such as the minimum number of units in stock). As a rule, you will use the one or more kinds of event handlers to successfully resolve required problems.

Using Input Mask and Display Mask

This topic describes how to use the `InputMask` parameter of the `PXDBString` attribute to restrict entering of text data for specified user interface (UI) elements of webpages. Value restrictions of UI elements can be of two types: content and structure.

In the first section is given the definition of the `InputMask` parameter and described the list of the possible values of this parameter and their usage, while in the second section is given the simple example of adding and using the `InputMask` parameter in the data access class (DAC) code.

You can use also the `DisplayMask` parameter: While the `InputMask` parameter enables the programmer to get or set the value specifying how users will enter data, the `DisplayMask` parameter enables the programmer to specifying how the UI element data will be displayed. The display mask has the same settings.

The InputMask Parameter and Its Possible Values

The `InputMask` parameter is a pattern that indicates the allowed characters in a string value. As a result, the application does not allow the user to enter other characters or more or less number of characters than had been defined for the UI element.

The default value of the `InputMask` parameter for key fields: `>AAAAAA`.

The mask format follows C# conventions, including the following:

- C, &: Any symbol
- A, a: Any letter or digit
- L, ?: Letter only
- #, 0, 9: Digit only
- >: All of the following characters will be in uppercase
- <: All of the following characters will be in lowercase

Example of use:

```
InputMask = ">LLLLL"
InputMask = ">aaaaaaaaaa"
InputMask = ">CC.00.00.00"
```

Static methods to set the parameter at run time:

```
public static void SetInputMask(PXCache cache, Object data, String name, String
 mask)
public static void SetInputMask<Field>(PXCache cache, Object data, String mask)
public static void SetInputMask(PXCache cache, String name, String mask)
public static void SetInputMask<Field>(PXCache cache, String mask)
```

Adding and Using an InputMask Parameter

Instructions below represent a simple example of creating and using the *InputMask* parameter. You shouldn't perform any actions, just analyze them.

To add a mask for validating the home phone number, do the following:

1. Modify the **HomePhone** member of the *Employee* data access class (DAC), as shown below. (*Plus* at the left of a code line means that this code line must be added while *minus* denotes deleting a code line that is to be replaced with the next line marked by the sign of plus.)

```
      ...
      public class Employee : PX.Data.IBqlTable
      {
          ...
          #region HomePhone
          public abstract class homePhone : PX.Data.IBqlField
          {
          }
-         [PXDBString(24, IsUnicode = true)]
+         [PXDBString(24, IsUnicode = true, InputMask = "(###) ###-####")]
          [PXUIField(DisplayName = "Home Phone")]
          public virtual string HomePhone { get; set; }
          #endregion
          ...
     }
      ...
```

2. Build the project.
3. Open the *Employees* page, right-click any area of the page, and select *Refresh*.

   If this page was already opened, you must refresh it to retrieve the changes you have made.

4. Point to the **form** control, open the smart tag associated with it, and select **Edit Content Layout**.
5. In the left area of the Layout Editor window that appears, expand the second column node and delete the **HomePhone** field by clicking the **Remove active** item.
6. In the right window of the Layout Editor, click the **Fields** tab, and notice the **HomePhone** field, which is defined now as a *MaskEdit* control.
7. Select the check box that precedes the **HomePhone** field, and click **Generate**.
8. In the left window of the Layout Editor, move up by one position the restored **HomePhone** field to place it in its original position.

   Formatting characters are not stored in the database or applied on the DAC level. For example, if a phone number is displayed in the UI as *(999) 999-9999*, the number is stored in the database as *9999999999*. As a result, some existing data may be displayed incorrectly if, for instance, imported data contained invalid characters or a different number of digits. In such cases, you need to restore the appropriate value of this phone number manually or change the incorrect input mask.

9. Click **OK** to close the Layout Editor window, and save the page.
10. Start the application with the *Employees* webpage, open the webpage (or perform refresh procedure, if it had been opened before), and explore the functionality of the masked field: Insert a new employee record and add a phone number to ensure that you cannot add more than ten digits to this field, and that the parentheses and hyphen are displayed in the appropriate positions, in compliance with the mask definitions. (See the screenshot below.)

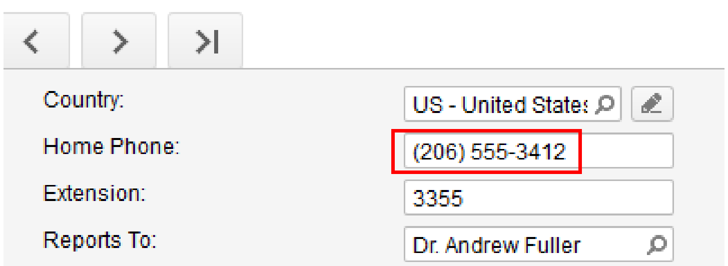


Figure: Exploring the HomePhone field with the InputMask value restrictions

You can specify input masks only for masked text edit fields. However, a simple text edit field has the `ValidateExp` property, for which you can specify a regular expression that will be executed by JavaScript when fields in a browser are validated.

Interaction With the Server

In this chapter, you will get acquainted with the singularity of interaction a webpage with the Server.

Content

This chapter covers the following topics:

Confifuring Webpage UI Elements and Behavior of BLCs

User interface (UI) elements have the `CommitChanges` property for specifying dynamic webpage behavior. This property indicates for the webpage when the client data needs to be sent to the server for processing. The first section of this topic is devoted to the description of the `CommitChanges` property while in the second section is illustrated the use of the `AutoCallback` group of properties, which provides navigation buttons that can be employed for moving from one webpage to another one.

The CommitChanges Property

Navigation between records on the webpage is based on the *key fields* concept. When the user selects key field on the webpage (for instance, to navigate to another product ID), the browser sends the keys to the server to retrieve a new record based on the selected key values.

The some UI element values may need to be sent to the server for processing (for instance, to respecify possible values of the webpage's UI elements that depend on the added or updated field value). To activate the system capability to provide interactive webpage behavior during data entry or update, the developer should set the **CommitChanges** property to *True* for appropriate UI elements. These UI elements can be placed on the form control or in the grid control as table columns.

Depending on the implemented logic, changed values of UI elements (with the **CommitChanges** property that is set to *True*) can be send to the server at the moment of modifying their values or at the moment of losing focus. UI element values are sent and refreshed only for UI elements with the **CommitChanges** property set to *True*,

During execution of the **CommitChanges** property, data the user inserted on the web page is posted to the server and submitted to the BLC to trigger the execution of the associated business logic.

Using AutoCallBack Properties to Add a Navigation Button on a Grid Toolbar

For an example, adding a navigation button on the grid toolbar of the *List of Employees* inquiry webpage is illustrated. Users may click this button to open the *Employees* maintenance webpage, if they want detailed information about the current employee.

Because this example illustrates only the design part of implementation of a navigation button, without logic changes in the business logic controller (BLC) code, to describe the use of the **AutoCallBack** properties, you shouldn't perform the instructions below.

To add the **Employee Details** navigation button, the developer must fulfill the following actions:

1. Open the *Employees* page in design mode and select the **ds** control. Select the **CallbackCommands** property and click the button at the right. On the **Callback Commands** window that appears, select the **openEmployee** command (that was defined in the appropriate BLC code) and change the **DependOnGrid** property value to *grid*. Click **OK**.

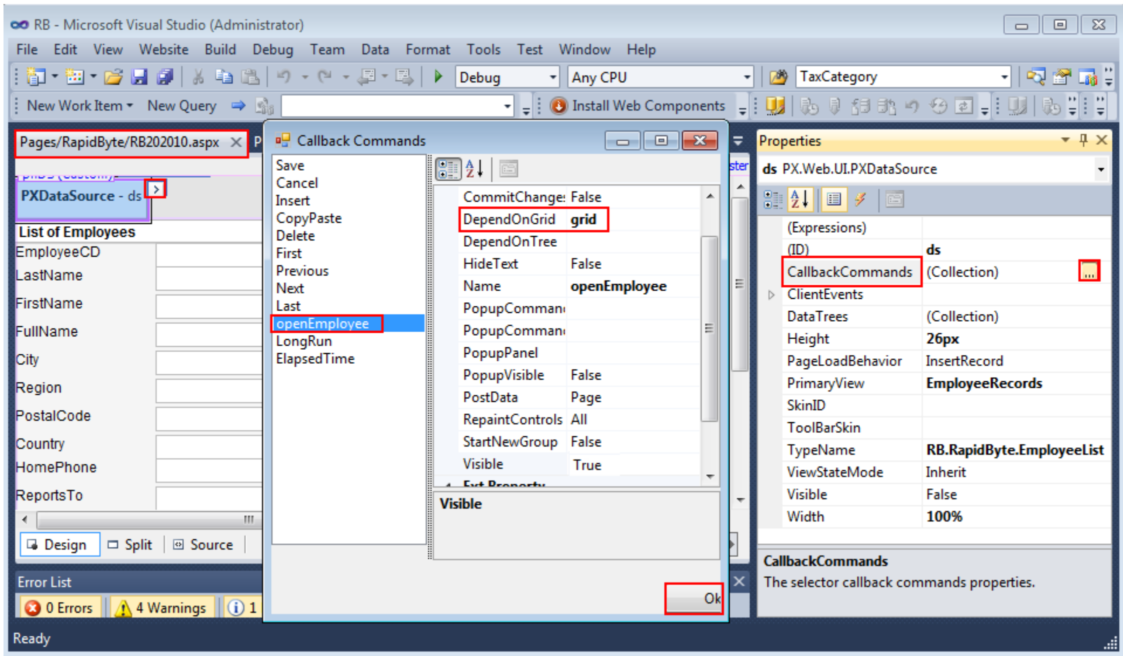


Figure: Setting the DependOnGrid property

The **DependOnGrid** property specifies the **grid** control the action depends on. When the action button is clicked, the data source posts the keys from the active **grid** control row to synchronize the **grid** control column values with the current DAC reference before the action is executed.

2. Add the custom button on the grid toolbar, as the screenshot below illustrates. Select the **grid** control and select the **ActionBar** > **CustomItems** property. On the **PXToolBarItem Collection Editor** window that appears, add a new member by clicking **Add** in the lower left area of the window. Modify the properties of the new button as follows:

   - **Text**: *Details*
   - **AutoCallBackCommand**: *openEmployee*

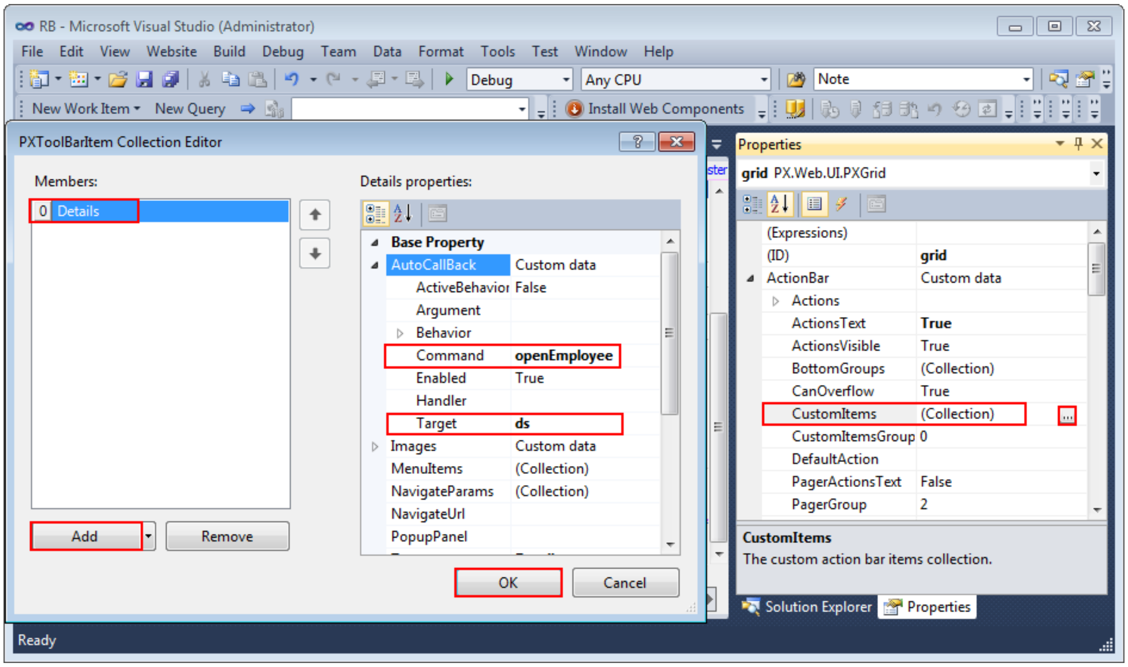


Figure: Adding a custom button

3. After saving the page and building the solution; you can start application, open the *List of Employees* webpage, select any row with an employee, and click the **Employee Details** button. The *Employees* webpage opens, with more detailed information about the selected employee (see the screenshot below).

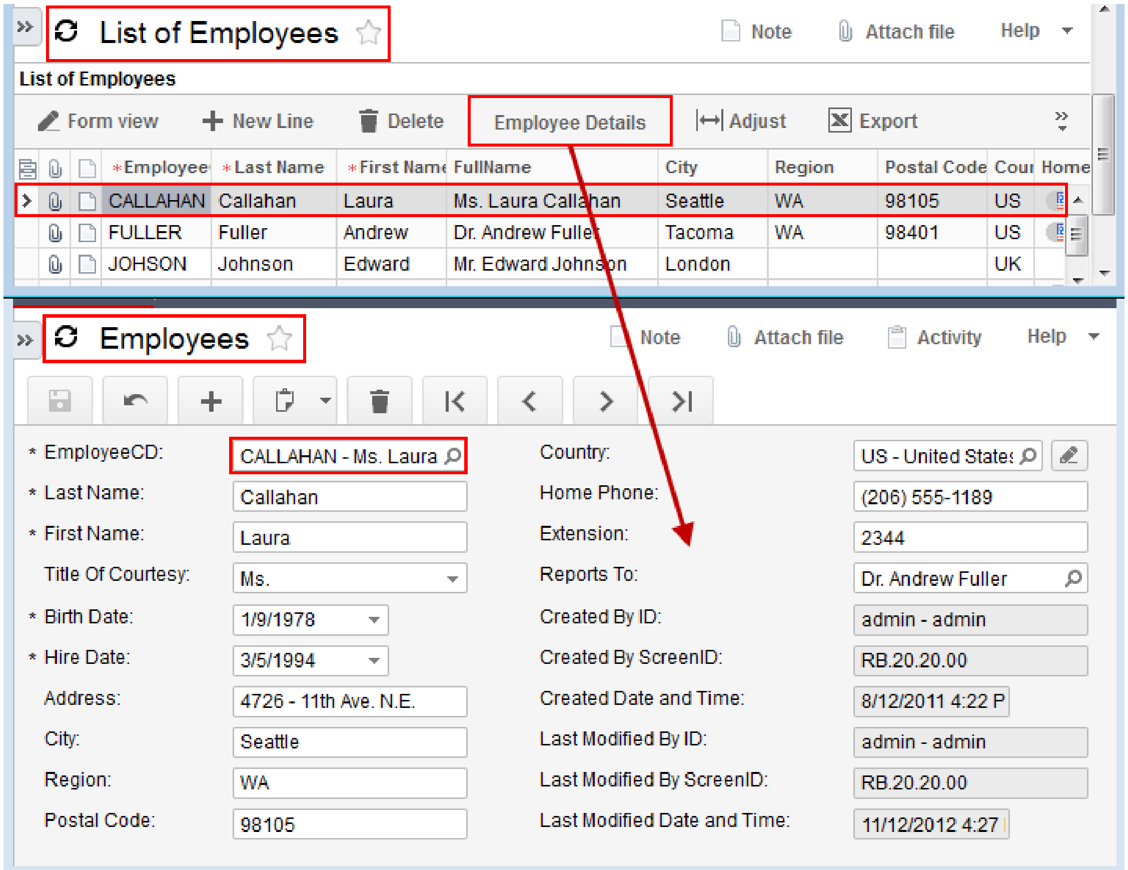


Figure: Using the Employee Details button

Creating an Acumatica ERP Add-on Project

This article explains how to create a new project in Microsoft Visual Studio. You create the project before you start to develop an add-on application integrated with Acumatica ERP.

Upload an Acumatica ERP Website

Before you begin, make sure that Acumatica Framework has been installed on your computer. Then upload an Acumatica ERP website into Microsoft Visual Studio Solution by performing the following actions:

1. Start Microsoft Visual Studio. On the **Files** menu, select *Open* and then *Web Site*, as shown in the screenshot below.

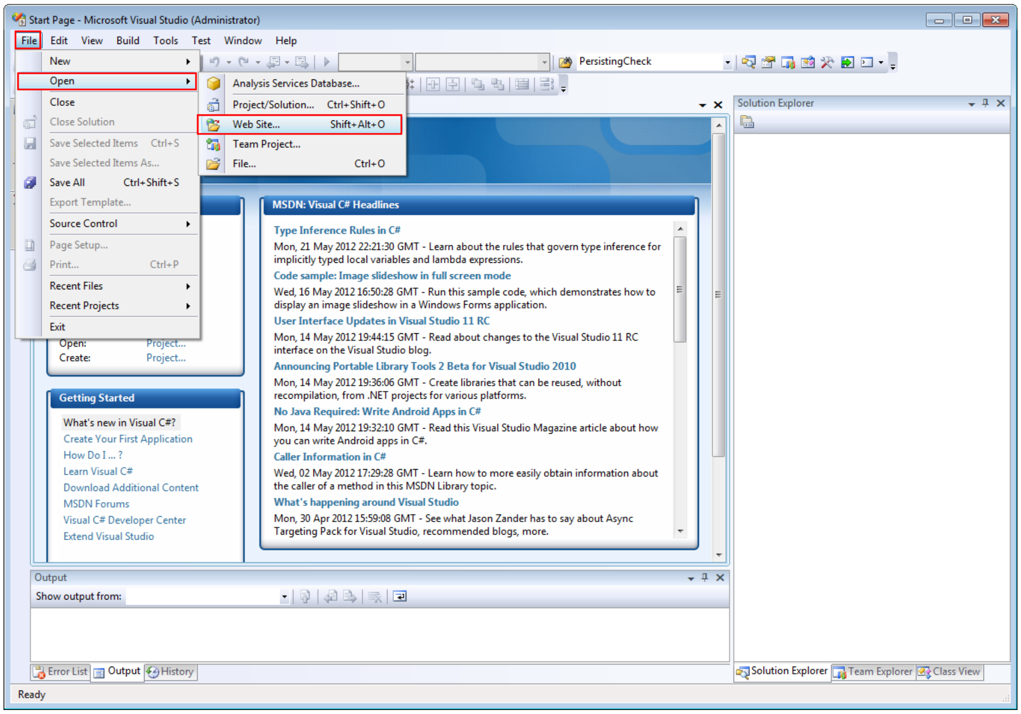


Figure: Starting to import a website

2. On the **Open Web Site** dialog box that appears, select the folder where the original Acumatica ERP application instance had been installed, and click **Open**. The Acumatica ERP site structure is imported into Microsoft Visual Studio as a new solution, as shown in the screenshot below.

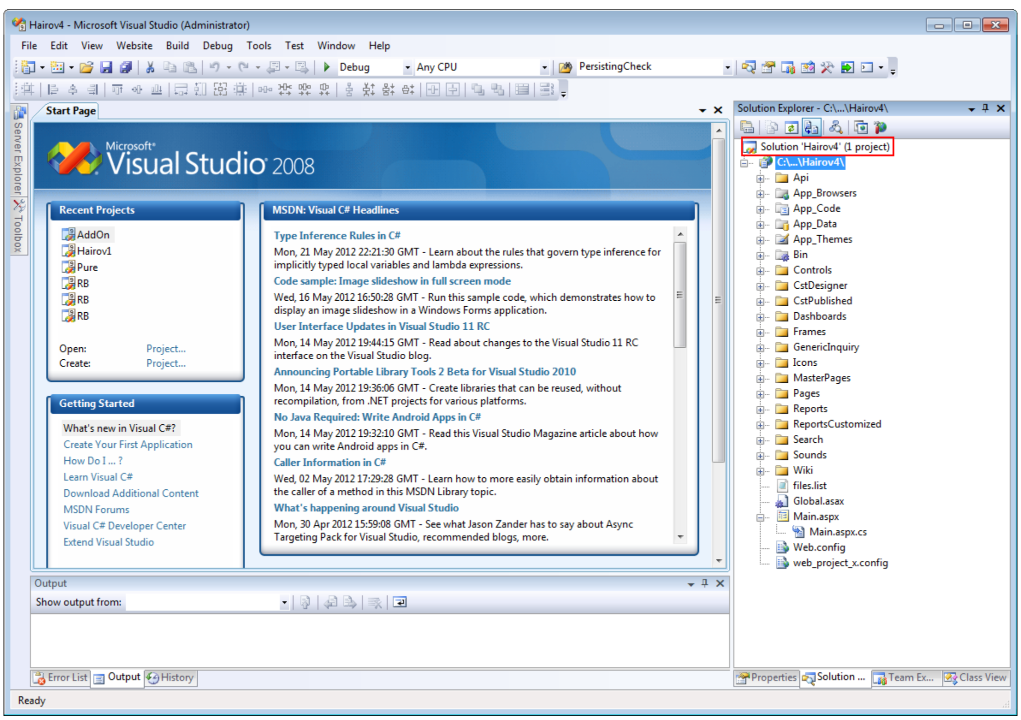


Figure: The imported website

Create an Add-on Project

Now you create a new project within the solution by doing the following:

1. In the Solution Explorer tree, right-click the solution name, and select *Add* and then *New Project*, as shown in the screenshot below.

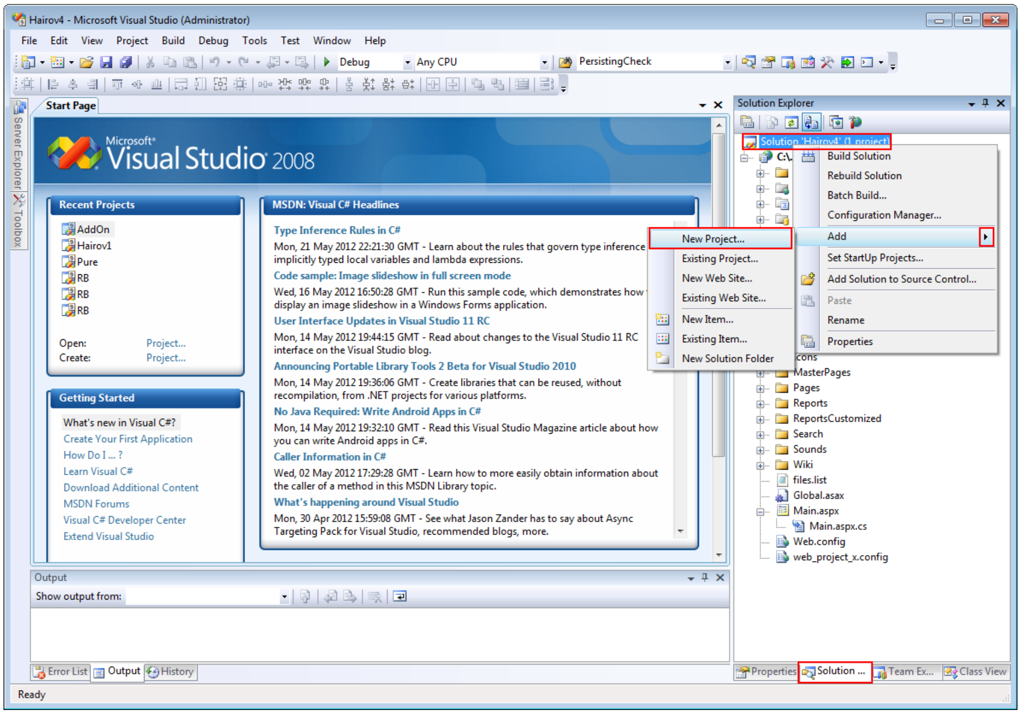


Figure: Adding a new project

2. In the **Add New Project** window that appears, select **Visual C#** as the project type and **Class Library** as the project template. Type the name of the new project and select the folder where the new project must be located, as shown in the second screenshot below. Click **OK**.

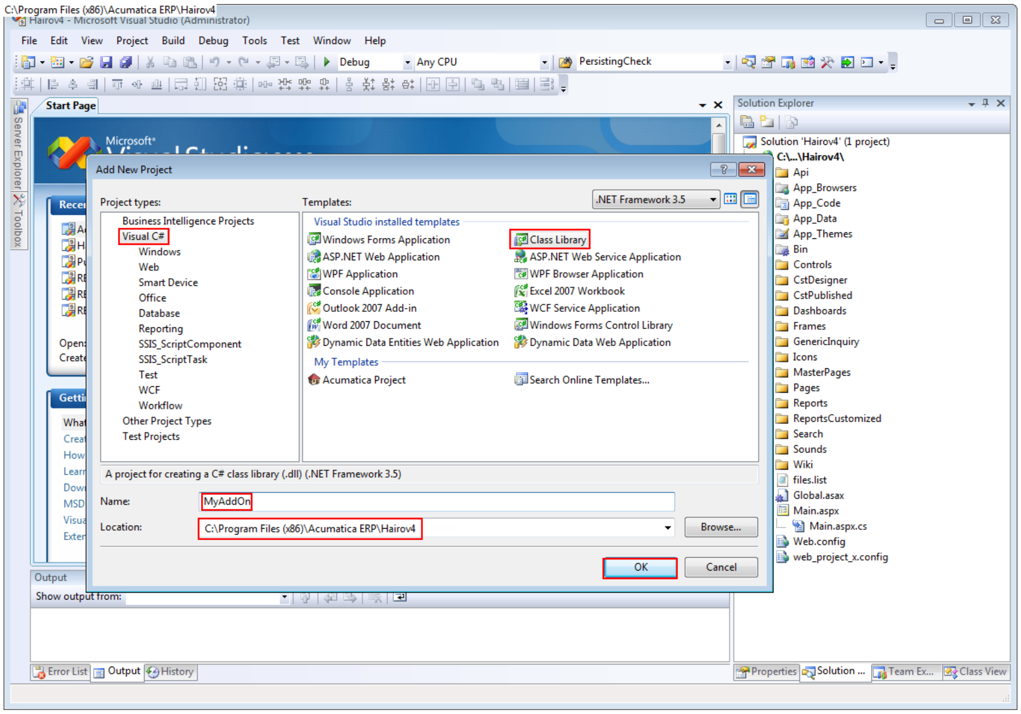


Figure: Defining the project properties

The project name must be unique within the Acumatica ERP installations that exist on the server or on your PC (if you are installing the project locally).

We recommend that you place the files of the new project within the Acumatica ERP application solution folder so that you can easily locate them. (See the example on the screenshot above.)

3. Right-click the created project's name, and select *Add* and then *New Folder*, to create the **DAC** folder within the project. Repeat these steps to create the **Descriptor** folder within the project.

4. Right-click **References** under the project's name and then select *Add Reference*, as the screenshot below illustrates.

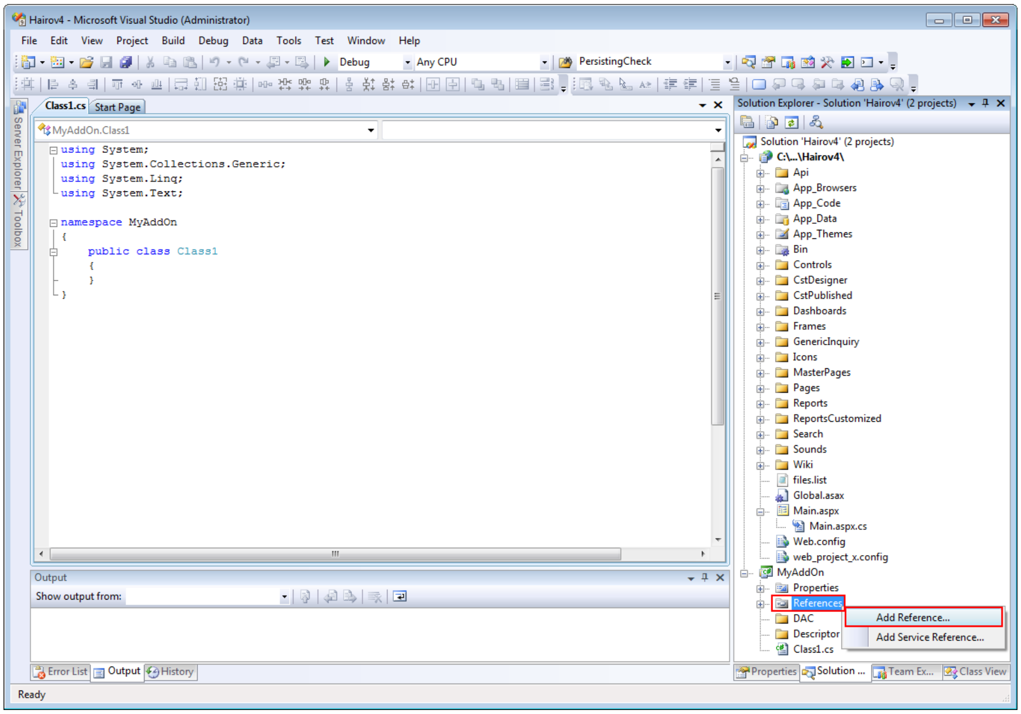


Figure: Starting to get references

5. In the **Add Reference** window that appears, select the **Browse** tab. Via the **Look in** search box, find the folder where the original application is located, select its **Bin** subfolder, and select the *PX.Common.dll*, *PX.Data.dll*, and *PX.Objects.dll* files. Then click **OK** to get references from the original application. (See the screenshot below.)

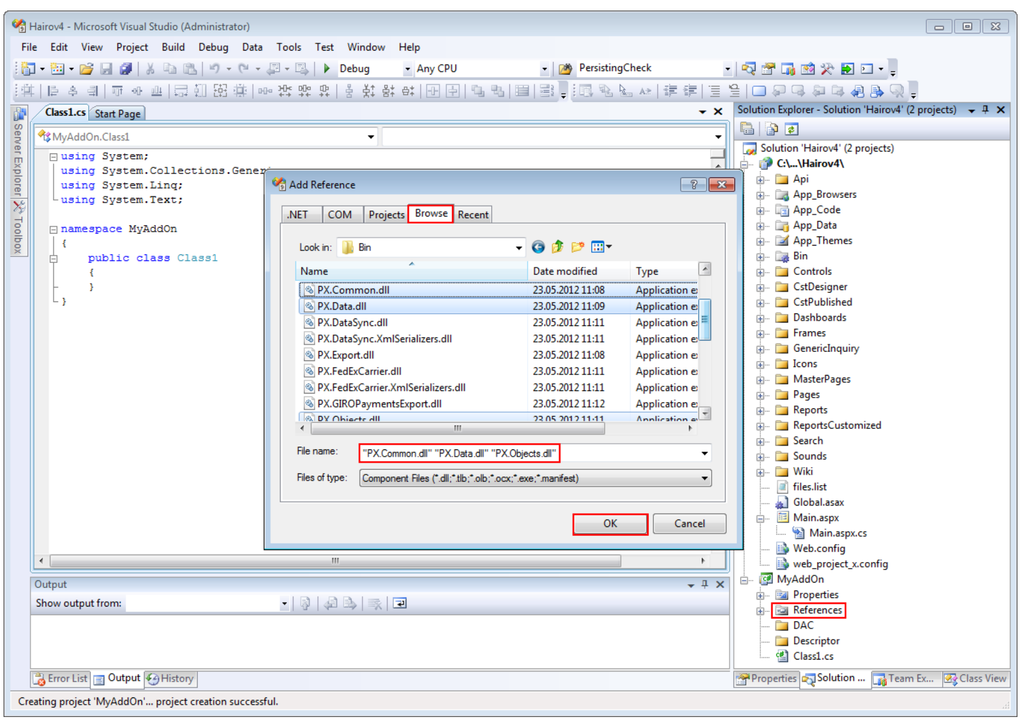


Figure: Getting references

6. Right-click the **Bin** folder and select *Add Reference*, as shown in the screenshot below.

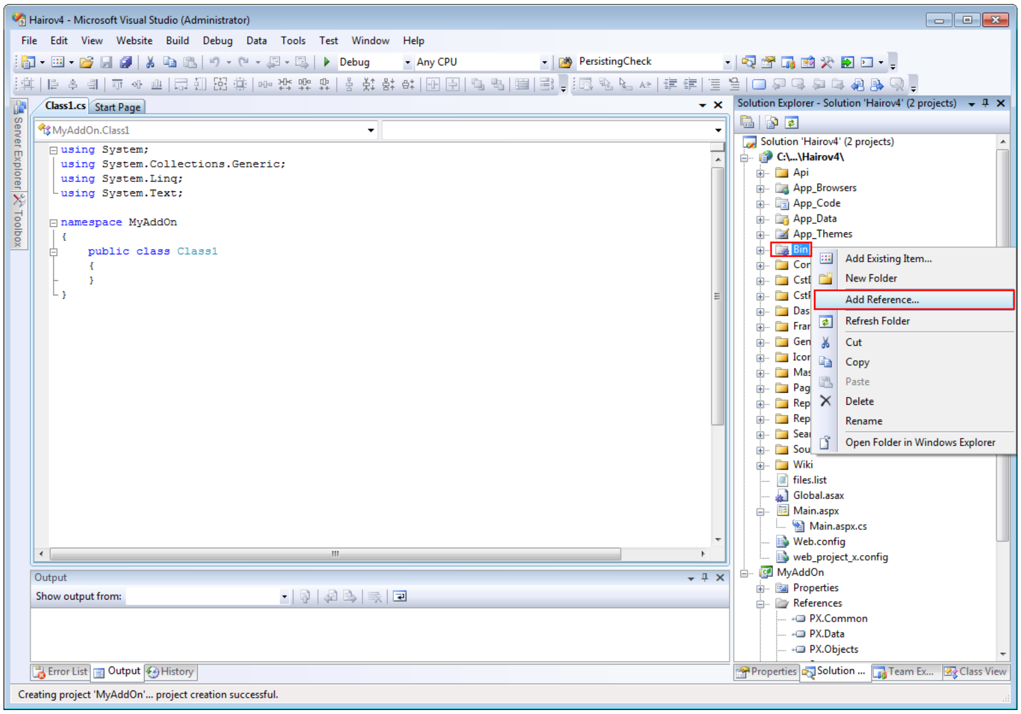


Figure: Preparing to add the reference

7. In the **Add Reference** window that appears again, open the **Projects** tab. Select the automatically created record with the new project's name from the list (which contains one record in the illustrated case), and click **OK**, as shown in the screenshot below. The reference to the created project is added to the Acumatica ERP website.

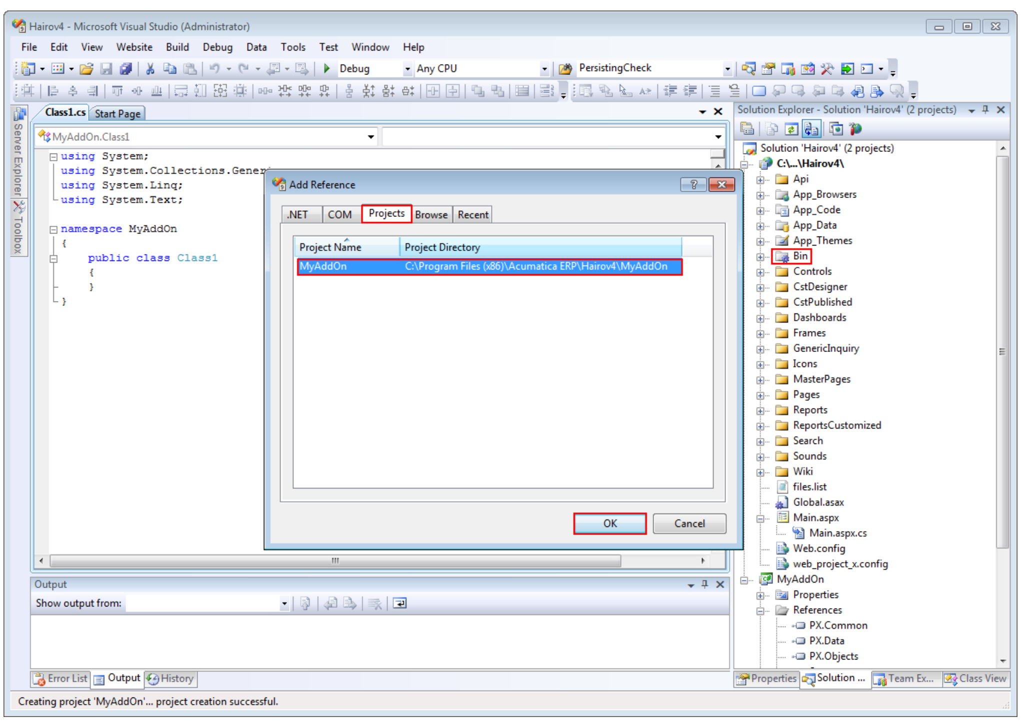


Figure: Adding the reference from the original application

8. In the Solution Explorer, right-click the *Class1.cs* file in the root of the project, and select *Delete* to remove this redundant file, as shown in the screenshot below.

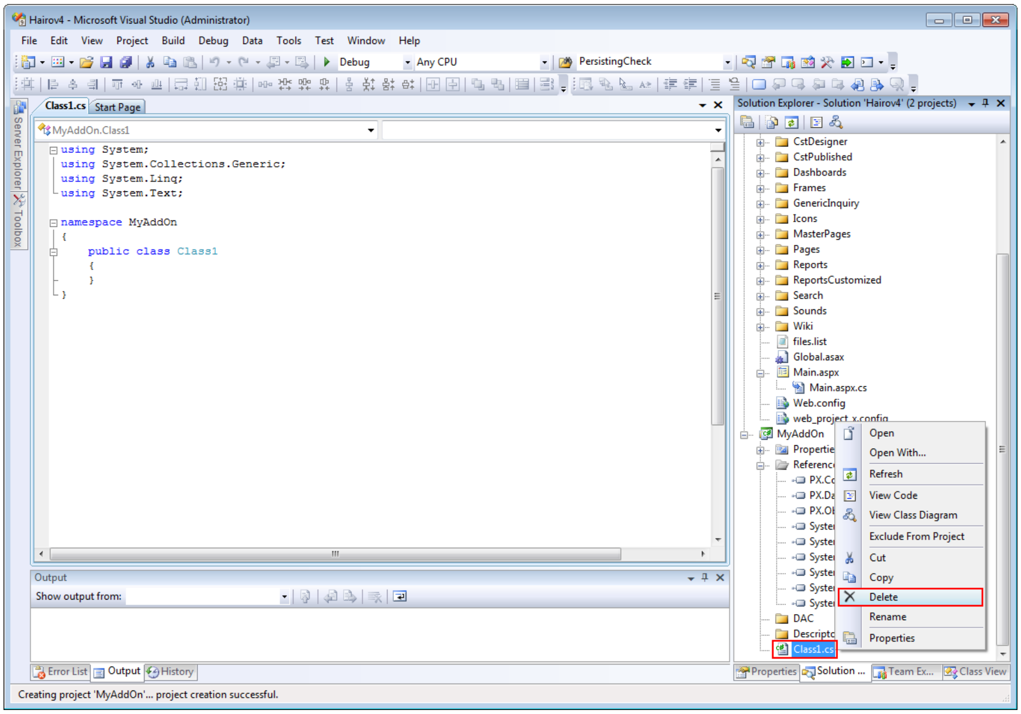


Figure: Deleting the originally created file

9. On the **File** menu, select *Save all*. Select the full path to the new project, and type the name (or keep the default name) of the solution file, as shown in the screenshot below, to save the created project within its solution.

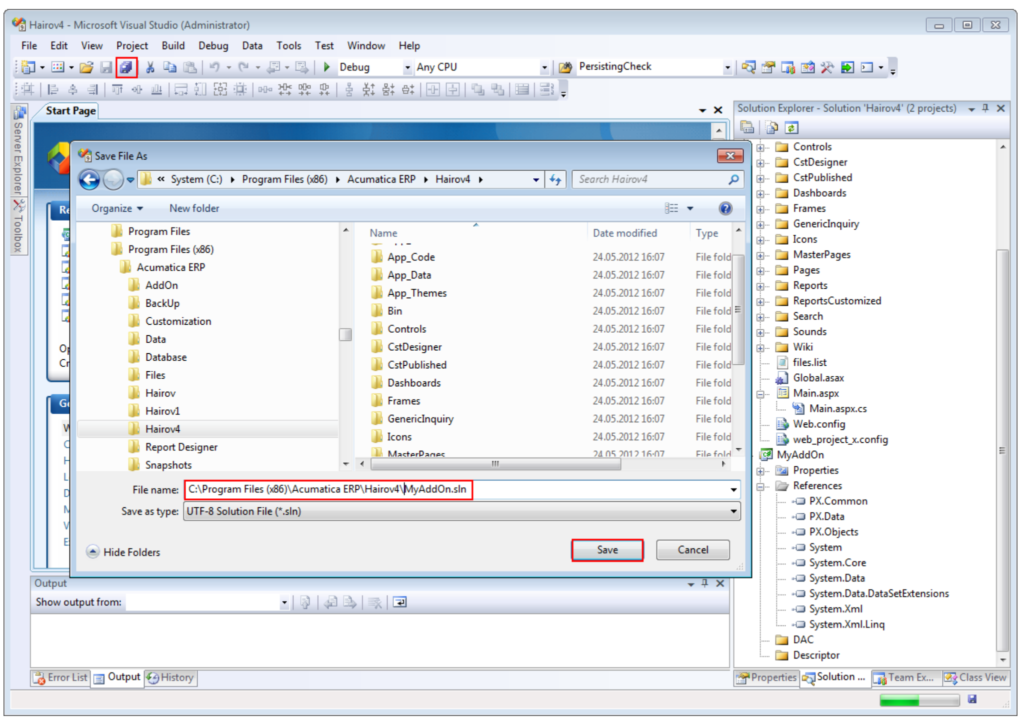


Figure: Saving the add-on project

10. On the **Build** menu, select *Build Solution*. At this point, the new solution (with the new add-on project) should be built without errors. The screenshot below illustrates the build process.

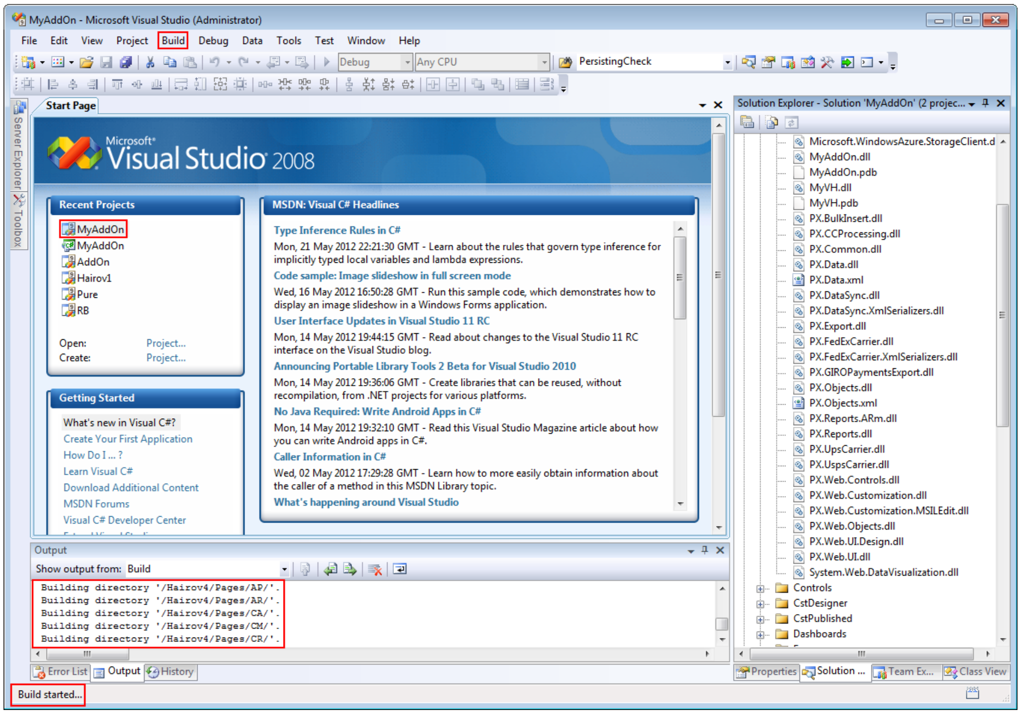


Figure: Building the entire solution

Summary

By executing the instructions in this article, you have learned to do the following:

- Upload the original Acumatica ERP site into Microsoft Visual Studio and create the new solution for developing a new integrated product.
- Create the new project and file structure within this solution for development of an add-on application. The new project area can be used for implementing business logic within that add-on application.
- Create references between the Acumatica ERP website and the new project. This enables the use of Acumatica ERP objects in your project and adds the reference to the new project within the original Acumatica ERP website.
- Add the configuration file to provide automatic mapping of the Acumatica ERP application attributes to the corresponding database fields.

Implementing a Credit Card Processing Plug-in

With Acumatica ERP, you can process credit card payments through third-party authorization centers. In the system core, only the processing through Authorize.Net is supported, but it can be implemented for other authorization service providers. This may be done in the future versions of Acumatica ERP or even by the Acumatica ERP client development team. Usually, access to the authorization service service requires certain prerequisites from the client:

- Must have an Internet Merchant Account (IMA)
- Must provide an SSL connection to the authorization center, so must have valid SSL certificate.
- Must have a contract with the corresponding authorization center.

Implementation of Credit Card Processing

Generally, a credit card authorization center has its own communication protocol: specific rules to send required data (card number, amount, CCV code, and so on) and to receive and interpret its response. Normally, the protocol includes the following functions:

- **Authorize CC Payment**: Checks if the requested sum may be taken from credit card and locks it on the credit card account. Usually, if authorization is not captured or voided, it expires after 30 days.
- **Capture CC Payment**: Actually takes the previously authorized amount from the card.
- **Authorize And Capture** (optional): Performs the previous two actions in one transaction.
- **Void**: Reverses the authorized or captured transaction. This may be done during a certain period of time after the transaction (such as 24 hours).
- **Credit**: Returns money back to the card.
- **Void Or Credit** (optional): Tries a void first and then performs a credit if voiding failed.

So we need only to implement this protocol and the communication with the core of Acumatica ERP.

The object must implement the following interface:

```
// This class implements the interaction with the authorization center
public abstract class ICCPaymentProcessing
{
    abstract public void Initialize(
        IProcessingCenterSettingsStorage aSettingsReader,
        ICreditCardDataReader aCardDataReader,
        ICustomerDataReader aCustomerDataReader,
        IDocDetailsDataReader aDocDetailsReader);
    abstract public void Initialize(
        IProcessingCenterSettingsStorage aSettingsReader,
        ICreditCardDataReader aCardDataReader,
        ICustomerDataReader aCustomerDataReader);
    abstract public bool DoTransaction(CCTranType aType,
                                       ProcessingInput aInputData,
                                       ProcessingResult aResult);
    abstract public bool IsSupported(CCTranType aType);
    abstract public void ExportSettings(IList<ISettingsDetail> aSettings);
    abstract public void ExportSettings(
        IList<ISettingsDetail> aSettings,
        CCProcessingSettingsType settingsType);
    abstract public CCErrors ValidateSettings(ISettingsDetail setting);
    abstract public void TestCredentials(APIResponse apiResponse);
}

// Types of transactions
public enum CCTranType
{
    AuthorizeAndCapture,    //Authorize And Capture as one transaction
    AuthorizeOnly,          //Authorize only
    PriorAuthorizedCapture, //Capture previously authorized transaction
    CaptureOnly,            //Capture manually authorized transaction
    Credit,                 //Return of the previously authorized transaction
    Void,                   //Void the previously authorized transaction
    VoidOrCredit,           //Try to Void, if failed -  Credit previously authorized
 transaction
}

// Supplementary interface to read processing center settings
// from the Acumatica ERP core
public interface IProcessingCenterSettingsStorage
{
    void ReadSettings(Dictionary<string, string> aSettings, string aCenterID);

}
```

```
// Supplementary interface  to read credit card data from the
// Acumatica ERP core
public interface ICreditCardDataReader
{
    void ReadData(Dictionary<string, string> aData);
    string Key_CardNumber { get; }
    string Key_CardExpiryDate { get; }
    string Key_CardCVV { get; }
    string Key_PMCCProcessingID { get; }
}

// Supplementary interface to read customer data from the Acumatica ERP core
public interface ICustomerDataReader
{
    void ReadData(Dictionary<string, string> aData);
    string Key_CustomerCD { get; }
    string Key_CustomerName { get; }
    string Key_Customer_FirstName { get; }
    string Key_Customer_LastName { get; }
    string Key_Customer_CCProcessingID { get; }
    string Key_BillAddr_Country { get; }
    string Key_BillAddr_State { get; }
    string Key_BillAddr_City { get; }
    string Key_BillAddr_Address { get; }
    string Key_BillAddr_PostalCode { get; }
    string Key_BillContact_Phone { get; }
    string Key_BillContact_Fax { get; }
    string Key_BillContact_Email { get; }
}

// Supplementary interface  to read specific document (bill, payment)
// item's data from the Acumatica ERP core
public interface IDocDetailsDataReader
{
    void ReadDate(List<DocDetailInfo> aData);
}

// Supplementary class to store document line information
public class DocDetailInfo
{
    public string ItemID;
    public string ItemName;
    public string ItemDescription;
    public decimal Quantity;
    public decimal Price;
    public bool? IsTaxable;
}

// Supplementary class  to receive data of the specific transaction
// from the Acumatica ERP core.
// Not all the fields may be used, depending on the type of the transaction.
public class ProcessingInput
{
    public int TranID;
    public int PMInstanceID;
    public string CustomerCD;
    public string DocType;
    public string DocRefNbr;
    public string OrigRefNbr;
    public string CuryID; //ISO Code
    public decimal Amount;
    public bool VerifyCVV;
}

// Supplementary class to return the result of authorization
// center transaction to Acumatica ERP
public class ProcessingResult
{
    public int TranID;
```

```
    public CCTranStatus TranStatus;
    public bool isAuthorized;
    public string PCTranNumber;
    public string PCResponseCode;
    public string PCResponseReasonCode;
    public string PCResponse;
    public string PCCVVResponse;
    public string AuthorizationNbr;
    public string PCResponseReasonText;
    public string ErrorText;
    public int? ExpireAfterDays;
    public CcvVerificationStatus CcvVerificatonStatus;
    public CCErrors.CCErrorSource ErrorSource = CCErrors.CCErrorSource.None;
}
```

The central object for the implementation is the `ICCPaymentProcessing` class; the rest just describes interfaces to communicate with the Acumatica ERP core.

```
abstract public bool DoTransaction(CCTranType aType, ProcessingInput aInputData,
 ProcessingResult aResult);
```

This is the main function of the object, which is called by Acumatica ERP to perform a request to the authorization center. So it must implement all of the main functions described above.

```
abstract public bool IsSupported(CCTranType aType);
```

Called by the core to determine if the operation is supported by the authorization center (useful for the optional types).

```
abstract public void Initialize(
    IProcessingCenterSettingsStorage aSettingsReader,
    ICreditCardDataReader aCardDataReader,
    ICustomerDataReader aCustomerDataReader,
    IDocDetailsDataReader aDocDetailsReader);
abstract public void Initialize(
    IProcessingCenterSettingsStorage aSettingsReader,
    ICreditCardDataReader aCardDataReader,
    ICustomerDataReader aCustomerDataReader);
```

These functions are called by the core when the object is created to provide a communication interface for the required data pulling (used in the `DoTransaction()` function) .

```
abstract public void ExportSettings(IList<ISettingsDetail> aSettings);
```

Used to export required for the processing settings keys (such as account login, password, and communication definitions). This function is used in the processing center configuration interface. These settings may be entered manually, but it's more convenient to import the key for them from the object.

Transaction Input and Output

Input	When the `DoTransaction()` method is called, the Acumatica ERP core provides the following information: • `public int TranID` internal unique transaction identifier (in the Acumatica ERP database) • `public int PMInstanceID` internal unique identifier of the credit card in Acumatica ERP. Card information may be obtained using the `ICreditCardDataReader` reference. • `public string CustomerCD;` unique identifier of the customer in Acumatica ERP.

<table>
<tr>
<td></td>
<td>

<code>public string DocType; public string DocRefNbr;</code> - unique internal payment document identifier. Document information may be obtained using <code>IDocDetailsDataReader</code> interface.
<code>public string OrigRefNbr;</code>
<code>public string CuryID;</code> ISO Code for the currency of transaction
<code>public decimal Amount;</code> Amount of the transaction
<code>public bool VerifyCVV;</code> Defines if CCV (credit card verification code) verification is required.

</td>
</tr>
<tr>
<td>Output</td>
<td>
Result of the transaction is returned to the Acumatica ERP core by using the <code>ProcessingResult</code> reference. The fields are as follows:

<code>public int TranID;</code> Internal unique transaction identifier (in the Acumatica ERP database), which must be the same as in input.
<code>public CCTranStatus TranStatus;</code> The status of the transaction, which must be one of the following

<pre><code>public enum CCTranStatus
{
    Approved, //The transaction is approved
    Declined, //The transaction is declined
    Error,   //There is an error in the transaction processing
 (usually, in the processing center)
    HeldForReview, //The transaction is held for review
    Unknown   //Unknown - for example, there is no answer or the
 answer can't be interpreted.
}</code></pre>

<code>public bool isAuthorized;</code> The transaction was authorized, for convenience
<code>public string PCTranNumber;</code> The transaction number assigned by the authorization center. It is needed to reference this transaction, for example, if you want to capture the authorized transaction.
<code>public string PCResponseCode;</code> The raw response code of the authorization center.
<code>public string PCResponseReasonCode;</code> The raw response reason code, a more detailed code from the authorization center.
<code>public string PCResponse;</code> The complete raw response from the authorization center.
<code>public string PCCVVResponse;</code> Additional code of the CCV verification from the authorization center (part of the complete response).
<code>public string AuthorizationNbr;</code>
<code>public string PCResponseReasonText;</code> The text of the response reason from the authorization center (part of the complete response). This text will be displayed in the credit card payment processing interface.
<code>public string ErrorText;</code> The description of the error if it happens in the object itself. For example, some settings are missing or the request to processing center can't be done.
<code>public int? ExpireAfterDays;</code> The period in days after which the transaction is automatically expired (for authorization transactions).

</td>
</tr>
</table>

- `public CcvVerificationStatus CcvVerificatonStatus;` The CCV verification status, which must be one of the following:

```
public enum CcvVerificationStatus
{
    Match,                          //CCV code is correct
    NotMatch,                       //CCV code is wrong
    NotProcessed,                   //CCV code is not processed
    ShouldHaveBeenPresent,          //CCV code was not provided, but is
 required for the authorization
    IssuerUnableToProcessRequest, //Card issuer is not able to verify
 the code
    RelyOnPreviousVerification,     //CCV code has been verified before
 (by the Acumatica ERP core) -
    //this flag is never set by the Credit Card Processing module.
    Unknown                         //Other
}
```

- `public CCErrors.CCErrorSource ErrorSource = CCErrors.CCErrorSource.None;` In the case of error, indicates its source, which may be one of the following:

```
public enum CCErrorSource
{
    None,
    Internal,          //Internal error of object
    ProcessingCenter, //Processing center reported an error
    Network,           //Network error  - for example, request time-
outed
}
```

It is the implementation's responsibility to perform a request to the authorization center and interpret the result of the request. Although Acumatica ERP will receive the AC response, it will rely on the `TransStatus` and `IsAuthorized` in the application payment logic.

How It works

On the Acumatica ERP side, the description of the credit card processing object is configured using the processing center configuration interface:

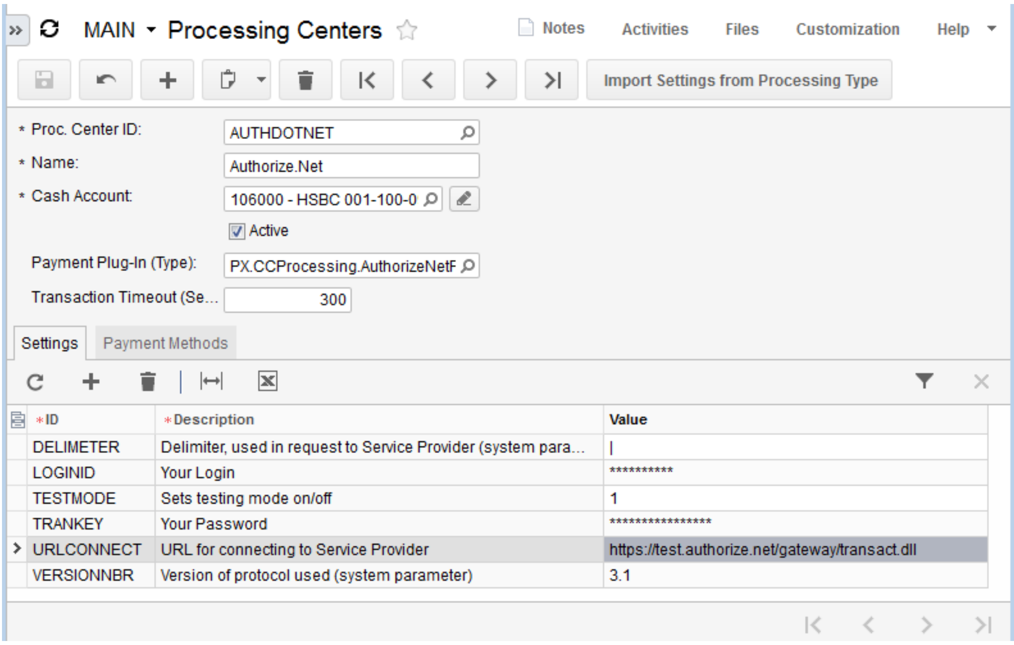


Figure: Configuration Screen

In this interface, the user must provide:

- The ID of the processing center and its description. This ID will be passed to the object when the `Initialize()` method is called.
- The full name of the credit card processing class.
- The set of default parameters for payment methods. This parameters are stored as key-value pairs. Keys may be imported from the objects if the `ExportSettings()` method is implemented properly in the class.
- The processed transaction open period; see the Warning for details.

On the second tab of configuration screen, the user can configure payment method types, which will be processed using the selected processing center. The card must be marked as active and the default in order to be processed through the processing center.

Specific card data is entered in the customer definition screen and stored encrypted in the database (unless tokenized processing is used). Sensitive data, such as the CCV code for the card, is stored encrypted until the first authorization is successfully made. After that, the data is deleted from the database and the following transactions are done without verification of the CCV code in the processing center. So, they will have CcvVerificationStatus = RelyOnPreviousVerification.

To perform actual credit card processing, the user should use the **Finance** > **Accounts Receivable** > **Work Area** > **Payments and Applications** page.

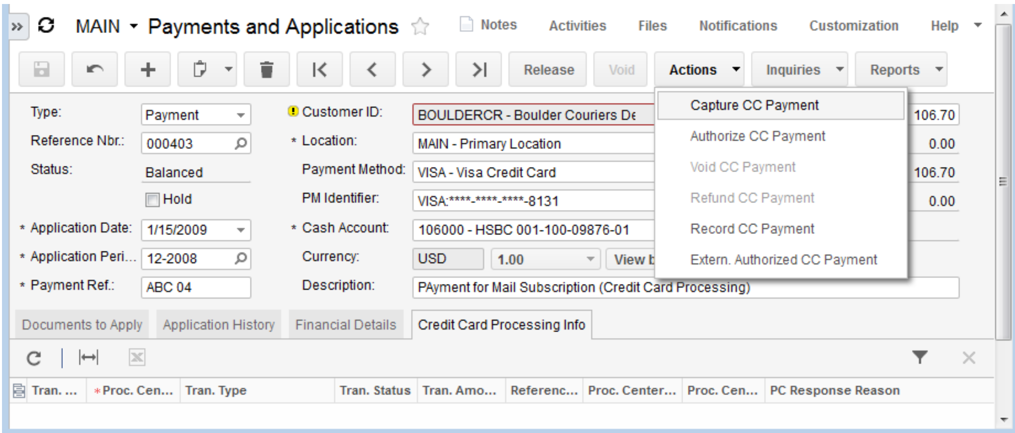


Figure: Credit Card Payment

The payment is entered as usual. If the customer has credit cards configured as the methods of payment, one of them may be selected as the payment method (if one is configured as the default for the customer, it will be selected automatically). In this case, the following options on the **Card Processing** menu will be available:

- Capture: To authorize and capture amount of this payment document. If the authorization center supports **Authorize And Capture**, this will be done in one transaction. Otherwise, two separate transactions will be performed. If the document already has the authorization transaction, only the **Capture** will be done.
- Authorize: To do the **Authorize** transaction only.
- Void: To Void/Credit Authorized or Captured Transaction. In some cases, voiding of the document is required.

If the **Integrated CC Processing** check box on the Accounts Receivable Preferences form is selected, successful capturing of the payment will automatically release the payment document. Otherwise, releasing the document is the user's responsibility.

When a user presses the one of the CC Processing buttons, the system creates an instance of the `CCPaymentProcessing` class, which is responsible for credit card transactions handling.

```
public class CCPaymentProcessing : PXGraph<CCPaymentProcessing>,
                                   IProcessingCenterSettingsStorage,
                                   ICustomerDataReader,
                                   ICreditCardDataReader,
                                   IDocDetailsDataReader
{
    public bool Authorize(int aPMInstanceID, bool aCapture, string aCuryID,
                          decimal aAmount, string aDocType, string aRefNbr,
                          ref int aTranNbr)
    public bool Capture(int aPMInstanceID, int aAuthTranNbr, string aCuryID,
                        decimal aAmount, ref int aTranNbr)
    public bool Void(int aPMInstanceID, int aRefTranNbr, ref int aTranNbr)
    public bool VoidOrCredit(int aPMInstanceID,int aRefTranNbr,
                             ref int aTranNbr)
    public bool Credit(int aPMInstanceID, int aRefTranNbr,string aCuryID,
                       decimal? aAmount, ref int aTranNbr)
}
```

The requested function is then called:

- Does preliminary validation of the credit card, checking the expiration date.
- Finds the authorization center configured to process this card.

- Creates an instance of the card processing object (which implements the `ICCPaymentProcessing` interface).

```
try
{
    Type processorType = BuildManager.GetType(aProcCenter.ProcessingTypeName, true);
    processor = (ICCPaymentProcessing)Activator.CreateInstance(processorType);
}
catch (HttpException)
{
    throw new PXException(Messages.ERR_ProcessingCenterTypeIsInvalid,
                          aProcCenter.ProcessingTypeName,
 aProcCenter.ProcessingCenterID);
}
catch (Exception)
{
    throw new PXException(Messages.ERR_ProcessingCenterTypeInstanceCreationFailed,
                          aProcCenter.ProcessingTypeName,
 aProcCenter.ProcessingCenterID);
}
```

It then calls its `Initialize` function, which does the following:

- Detects if CCV code for the card was verified, and sets the VerifyCVV flag to true, if not.
- Creates and commits to the database a transaction record; its unique identifier will be passed to the card processing object.
- Calls the `DoTransaction()` method of the object.

```
try
{
    hasError = !processor.DoTransaction(aTranType, inputData, result);
}
catch (WebException webExn)
{
    hasError = true;
    result.ErrorSource = CCErrors.CCErrorSource.Network;
    result.ErrorText = webExn.Message;
}
catch (Exception exn)
{
    hasError = true;
    result.ErrorSource = CCErrors.CCErrorSource.Internal;
    result.ErrorText = exn.Message;
    throw new
 PXException(String.Format(Messages.ERR_CCPaymentProcessingInternalError,aTranNbr,
 exn.Message));
}
finally
{
    this.EndTransaction(aTranNbr, result, (hasError ? CCProcStatus.Error :
 CCProcStatus.Finalized));
}
```

- After the transaction completion, it updates (closes) the transaction record based on the returned result (or error handling procedure). Note that errors are stored in a separate field in the database rather than in the `PCResponseText` field (if the error happens on our side). For an authorization transaction, it also stores the expiration date if the processing object provides a value for it in the result.

A protection mechanism prevents the user from starting two transaction for the same document in parallel (for example, from another window or computer). Before starting, the system checks if there is an open transaction for the document and rejects the action if so. In some conditions, such as server crash or hardware malfunction, the result of the transaction processing may be lost by the system so it will open forever. To avoid locking of the system, open transactions are made auto-expiring: when they start, an open period length is defined for them. If the processing result is lost, this transaction is considered as expired after this period and the user can start another one. This period length is defined in the processing center configuration interface as **Open Transaction Timeout (sec)**. Unfortunately, there is no way to synchronize an expired transaction with the authorization server automatically (it may be successful there); it will require user interaction to prevent double-charges.

- If transaction is successful and credit card processing is synchronized with document state handling, it may be released (or voided) after the processing.

Void Transactions Processing

In Acumatica ERP, a released AR document can't be deleted from the system. When you need to void such a document, the system actually creates another one that is reversing the original transaction. This document has the same number as original document, but another DocType, Void. If the original transaction has been paid by credit card, this payment has to be voided or refunded. To do this processing correctly, all of the credit card transactions made for the original document are also attached to the voiding document (so credit card processing transactions are shared between the original and the voiding document). The system tries to void the transaction first and if the transaction is declined by the authorization center (a void is possible after a rather short period of time), it tries to refund it. The transaction is processed the same way as described above.

Using Substitute Keys

This article explains the use of surrogate keys in Acumatica Framework.

In the table defined below:

```
CREATE TABLE [dbo].[Ledger](
    [CompanyID] [int] NOT NULL,
    [LedgerID] [int] IDENTITY(1,1) NOT NULL,
    [LedgerCD] [varchar](10) NOT NULL,
    [BalanceType] [char](1) NOT NULL,
    [BaseCuryID] [varchar](5) NOT NULL,
    [Descr] [nvarchar](60) NULL,
    [tstamp] [timestamp] NULL,
    [CreatedByID] [uniqueidentifier] NOT NULL,
    [CreatedByScreenID] [char](8) NOT NULL,
    [CreatedDateTime] [smalldatetime] NOT NULL,
    [LastModifiedByID] [uniqueidentifier] NOT NULL,
    [LastModifiedByScreenID] [char](8) NOT NULL,
    [LastModifiedDateTime] [smalldatetime] NOT NULL,
 CONSTRAINT [Ledger_PK] PRIMARY KEY CLUSTERED
(
    [CompanyID] ASC,
    [LedgerID] ASC
))

CREATE UNIQUE NONCLUSTERED INDEX [Ledger1] ON [dbo].[Ledger]
(
    [CompanyID] ASC,
    [LedgerCD] ASC
)
```

LedgerID is a surrogate key and *LedgerCD* is a native or natural key associated with this record.

Let's assume that we have *Batch* record that references *Ledger* record by surrogate key *LedgerID*. In this case user expects to see *LedgerCD* value in applicaiton UI. But at the same time *Batch* record stores *LedgerID* value for referencing *Ledger* record. For such situations, Acumatica Framework provides *Substitute Key* feature that substitutes *surrogate key* with *natural key* on presenting data in User Interface.

Use of surrogate allows to significantly reduce the space that is used by database for referencing and at the same time provide user with convenient data entry mechanism and generic functionality for renaming *natural keys* that are presented to the user in interface at a single dictionary.

In order use substitute key functionality following declaration is required:

1: Modify class *Ledger* by removing **IsKey** named parameter from *LedgerID* member, and add **IsKey** named parameter to *LedgerCD* member as below:

```
    [System.SerializableAttribute()]
    public class Ledger : PX.Data.IBqlTable
    {
        #region LedgerID
        public abstract class ledgerID : PX.Data.IBqlField
        {
        }
        protected Int32? _LedgerID;
-              <font color = "red"><b>[PXDBIdentity(), IsKey=true]</b></font>
+              <font color = "green"><b>[PXDBIdentity()]</b></font>
        [PXUIField(DisplayName = "Ledger ID", Visibility = PXUIVisibility.Visible,
 Visible = false )]
        public virtual Int32? LedgerID
        {
            get
            {
                return this._LedgerID;
            }
            set
            {
                this._LedgerID = value;
            }
        }
        #endregion
        #region LedgerCD
        public abstract class ledgerCD : PX.Data.IBqlField
        {
        }
        protected string _LedgerCD;
-              <font color = "red"><b>[PXDBString(10)]</b></font>
+              <font color = "green"><b>[PXDBString(10, IsKey=true)]</b></font>
        [PXUIField(DisplayName = "Ledger", Visibility =
 PXUIVisibility.SelectorVisible)]
        public virtual string LedgerCD
        {
            get
            {
                return this._LedgerCD;
            }
            set
            {
                this._LedgerCD = value;
            }
        }
        #endregion
              ...
        }
```

1: Use parameter *SubstituteKey* in *PXSelector* attribute definition for *LedgerID* member of *Batch* class as specified below:

```
    public class Batch : PX.Data.IBqlTable
    {
                ...
        #region LedgerID
        public abstract class ledgerID : PX.Data.IBqlField
        {
        }
        protected Int32? _LedgerID;
```

```
        [PXDBInt()]
        [PXDefault(typeof(GLSetup.ledgerID))]
        [PXUIField(DisplayName = "Ledger ID", Visibility =
  PXUIVisibility.SelectorVisible)]
        [PXSelector(typeof(Search<Ledger.ledgerID, Where<Ledger.balanceType,
  NotEqual<BudgetLedger>>>),
                        <font color = "green"><b>SubstituteKey =
  typeof(Ledger.ledgerCD))</b></font>]
        public virtual Int32? LedgerID
        {
            get
            {
                return this._LedgerID;
            }
            set
            {
                this._LedgerID = value;
            }
        }
        #endregion
               ....
    }
```

With such declaration *Field Schema Editor Wizard* will replace *LedgerID* with *LedgerCD* on adding *LedgerID* member from *Batch* class on application form. During runtime system will automatically substitute *LedgerID* with *LedgerCD* on providing data to UI and convert it back on passing data from UI to DAC.

With marking *LedgerCD* with **IsKey** parameters in class *Ledger* you must add parameter **SubstituteKey** to all *Data Access Classes* that references class *Ledger* by *LedgerID*.

Calling a New PXSmartPanel

How does the **Copy Order** (or any similar) action know to call the *PXSmartPanel*, that is, for the copy order (or another webpage used as a printable document), that is, how the programmer or customizer can get a new *PXSmartPanel* to display when he or she clicks the **OK** button? (See the screenshot below.)

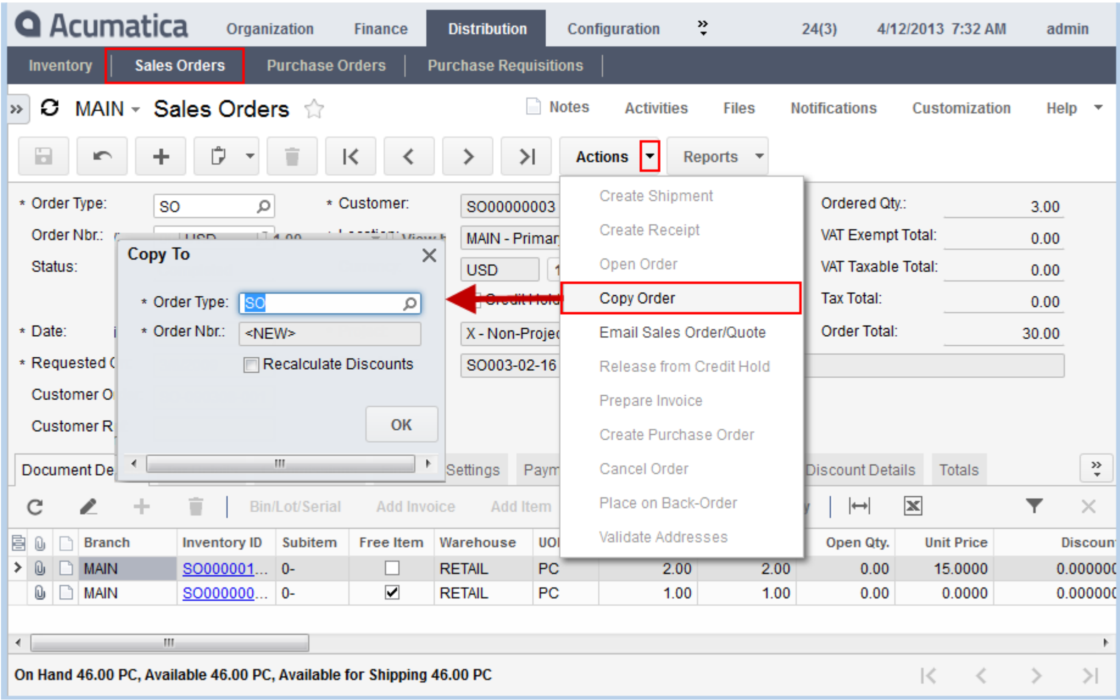


Figure: Calling a Smart Panel

Here is the explanation. To define a smart panel in the .aspx page, you should specify the *Key* property for it making this property equal to one of the view names in your business logic container (BLC, called also as graph). Then you should append a button to the panel with expected dialog result.

```
<px:PXSmartPanel ID="panelCopyTo" runat="server"
Height="135px" Width="300px" Style="z-index: 108; left: 351px;
position: absolute; top: 99px;" Caption="Copy To" CaptionVisible="true"
DesignView="Content" LoadOnDemand="true"
Key="copyparamfilter"
AutoCallBack-Enabled="true"AutoCallBack-Target="formCopyTo"
AutoCallBack-Command="Refresh" CallBackMode-CommitChanges="True"
CallBackMode-PostData="Page">

<px:PXButton ID="PXButton9" runat="server" DialogResult="OK" Text="OK"
Width="63px" Height="20px" TabIndex="102" CommandName="CheckCopyParams"
CommandSourceID="ds"> </px:PXButton>
```

Then in the button delegate, which will process copy order request, perform a call to *AskExt* method of the view specified as a *Key*:

```
public virtual IEnumerable CopyOrder(PXAdapter adapter)
    {
       if (copyparamfilter.AskExt() == WebDialogResult.OK &&
       string.IsNullOrEmpty(copyparamfilter.Current.OrderType) == false)
        {
.......................................................................
```

When the user clicks the **Copy Order** (or another document) menu item, the execution will interrupt on the *AskExt* call and a pop-up window will be displayed. After user clicks the **OK** button in the panel, the system will call the *CopyOrder* method for the second time, and this time *AskExt* will return required dialog result.

Debugging Applications

This article explains how to link the Acumatica Framework application site to the database and start the Acumatica Framework application in the debug mode.

Linking the Acumatica Framework Application Site to the Database

1. Locate the `RB.sln` file on C:\Program Files (x86)\Acumatica Framework\RB\RB.sln and double-click it to open the solution.
2. Locate the `web.config` file inside the website project, and open it for editing.
3. Modify the connection string by specifying the credentials to your development database as shown below.

   Use the credentials database name and company IDs you created. If login fails because of database connection errors, you can verify the connection settings in the `Web.config` file under the connectionStrings section. You can use the following examples as a reference. For a locally installed SQL Server that uses SQL Server authentication:

   ```
   connectionString ="data source=(local);Initial Catalog=Northwind1; User
    Id=USERID; Password=PASSWORD"
   ```

   For a locally installed SQL Server that uses Windows authentication:

   ```
    connectionString="data source=(local);Initial Catalog=Northwind1; Integrated
    Security=yes"/>
   ```

   For a remote SQL Server that uses SQL Server authentication:

   ```
   connectionString ="data source=MSSQLSERVER; Initial Catalog=Northwind1; User
    Id=USERID; Password=PASSWORD"
   ```

4. Set the `Main.aspx` page in the root of the website project as a project starting point.
5. Run the application from the Visual Studio. It will start the development server and run the application in Debug mode.

   If you created a new database, use the credentials below for the first login:

   - **Login**: admin
   - **Password**: setup

   When you run your project in Debug mode, code execution may suspend at certain points with warning or error messages such as *SecurityException was unhandled by user code*. These warnings, artifacts of the debugging environment in which the project is executing, will not occur when the project is deployed to a production IIS server. You can safely ignore them and continue code execution by simply pressing F5 or clicking the **Run/continue** button on the debugging toolbar in Visual Studio. Alternatively, you can avoid the error messages during debugging by commenting out the security restriction section of the Web.config file, as shown below:

   ```
   <!--
       <securityPolicy>
           <trustLevel name="ProjectX" policyFile="web_project_x.config"/>
       </securityPolicy>
       <trust level="ProjectX" originUrl=""/>
    -->
   ```

   The `web.config` file is allowed for check-out but not allowed for check-in.

Debugging the Acumatica Framework Application Under IIS Server

In many cases the developer, instead of running the Acumatica Framework application from the Visual Studio development server, finds it more convenient to run it from IIS. Below are the steps that are required to register Acumatica Framework with the IIS server and attach it to the application with the debugger:

1. Register the Acumatica Framework application site under IIS as follows:
   a. Open the Internet Information Server (IIS) Manager application.

      To locate this application, in the search area of the windows start menu, type `IIS`.

   b. In the IIS Manager, focus on **Default Web Site** and from the content menu, select **Add Application...**. The Add Application menu will appear.
   c. In the Add Application menu, specify the website alias, application pool and physical path of the site. Use the example below as a reference:
      - **Alias**: `RB`
      - **Application Pool**: `DefaultAppPool`
      - **Local Path**: `C:\Program Files (x86)\Acumatica Framework\RB\Site\`
   d. Click **Add** to create the new website.
   e. Go to the created site and set the `Main.aspx` page as a default document for this site.
   f. Make sure that site works by accessing it as `http://localhost/RB`.
2. Open the `C:\Program Files (x86)\Acumatica Framework\RB\RB.sln` solution in Visual Studio.

   If you have user access control activated on your computer, make sure that you run Visual Studio as an Administrator.

3. Edit the `web.config` file like: `<compilation debug="true" ... >...</compilation>`
4. Once the project is opened in the Visual Studio, go to the **Debug->Attach to Process** menu.
5. In the **Attach to Process** pop-up window, select the **Show processes from all users** and **Show processes from all sessions** check boxes. Locate the process named `w3wp.exe` and click **Attach**.
6. Accept the warning and the debug session will start. Now you can access the website from `http://localhost/RB` and intercept the break points set in the code from Visual Studio.

Note that your local path might differ from the path specified if you mapped the solution to the different location on the local file system or building different branch.

API Reference

This reference describes the application programming interface (API) of the Acumatica Framework.

The following sections correspond to the specific components of the framework API:

- *BQL*
- *Event Model Overview*
- *Core Classes*
- *Attributes*

Event Model

The Acumatica Framework provides its own event model. By implementing event handlers, application developers can add business logic for the manipulation of data within business logic controllers (BLCs).

The following chapters describe different ways of adding event handlers, provide detailed diagrams for common data manipulation scenarios, and include complete reference information on all events, including code samples demonstrating common usage and classes and enumerations related to these events:

- *Event Model Overview*
- *Scenarios*
- *Events*

Event Model Overview

The Acumatica Framework provides its own event model. An application developer can define *event handlers*, methods invoked by the Acumatica Framework once the corresponding events are raised, to add business logic related to the manipulation of business logic controller (BLC) data. This business logic includes validation and calculation of field values, management of related data records (inserting, updating, or deleting), checks for duplicate records, and implementation of user interface (UI) presentation logic.

Data Manipulation Scenarios

Events related to the manipulation of data records and data fields are raised in a particular order within certain scenarios. For descriptions of these data manipulation scenarios, see the *Scenarios* section.

All Events

For reference information about all events, see *Events*.

Event Handlers Types

Two types of event handlers are associated with each event:

- *Graph event handlers* are defined as methods in a BLC class for a particular data access class (DAC) or a particular DAC field. See the reference topic of each event for an example of a graph event handler declaration.
- *Attribute event handlers* are defined as methods in attribute classes. The corresponding logic is attached to all DAC objects or data fields annotated with these attributes. The attribute in which an attribute event handler is implemented must be derived from

the `PXEventSubscriberAttribute` class and must implement the interface of the `IPX`*`EventName`*`Subscriber` form, as shown in the following example.

```
// The attribute implements handlers for the FieldVerifying
// and RowPersisting events
public class MyAttribute : PXEventSubscriberAttribute,
                           IPXFieldVerifyingSubscriber,
                           IPXRowPersistingSubscriber
{
     public virtual void FieldVerifying(PXCache sender,
                                        PXFieldVerifyingEventArgs e)
     {
         ...
     }

     public virtual void RowPersisting(PXCache sender,
                                       PXRowPersistingEventArgs e)
     {
         ...
     }
}
```

Event Handlers Execution

All event handlers executed for a particular event share the same `PXCache` instance that has raised this event. A `PXCache` instance is created to control the modified data records of a particular DAC type. The `PXCache` instance is always available as the first argument in an event handler. The second argument provides specific data corresponding to the event.

Once an event is raised, the order in which associated event handlers are executed may differ.

For some events, the chain of graph event handlers is executed before attribute event handlers, which are executed only if the `Cancel` property of the event arguments doesn't equal `true` after execution of the graph event handlers.

For other events, the attribute event handlers are executed first, and the graph event handlers are executed afterwards. The reference topic for each event includes a diagram showing the order in which the system invokes handlers for a particular event..

Adding Event Handlers Dynamically

A BLC includes collections of graph event handlers for all events except `CacheAttached`. Each such collection holds event handlers for a particular event and has the same name as the event . By using the methods of these collections, you can add and remove graph event handlers in code at run time.

A method added as an event handler must have the signature of a graph event handler, but doesn't need to follow the naming convention for graph event handlers. If you want to add a method as an event handler, invoke the `AddHandler<>()` method on the corresponding collection. For example, if the event is related to a row, it is invoked as follows.

```
RowEventName.AddHandler<DACName>(MethodName);
```

The event is invoked as follows if it is related to a field.

```
FieldEventName.AddHandler<DACName.fieldName>(MethodName);
```

To remove a handler, you should invoke the `RemoveHandler<>()` method in exactly the same way.

On invocation of `AddHandler<>()`, event handlers are added to either the beginning or the end of the collection:

- Event handlers are added to the beginning of the collection for any event whose name ends with *ing*, except the `RowSelecting` event.

- Event handlers are added to the beginning of the collection for any event whose name ends with *ed* and for the `RowSelecting` event.

Scenarios

Most events are raised within common scenarios related to the manipulation of data records. The scenarios are invoked by Acumatica Framework on certain user actions in the user interface (UI), on the corresponding requests to the Web Service API, and on the execution of special methods within the business logic controller (BLC).

For details on how Acumatica Framework processes the basic data operations, see the following topics:

- *Inserting a Data Record*
- *Updating a Data Record*
- *Deleting a Data Record*
- *Displaying a Data Record*
- *Saving Changes to the Database*

Inserting a Data Record

The sequence of events raised during the insertion of a data record is illustrated in the figure below.

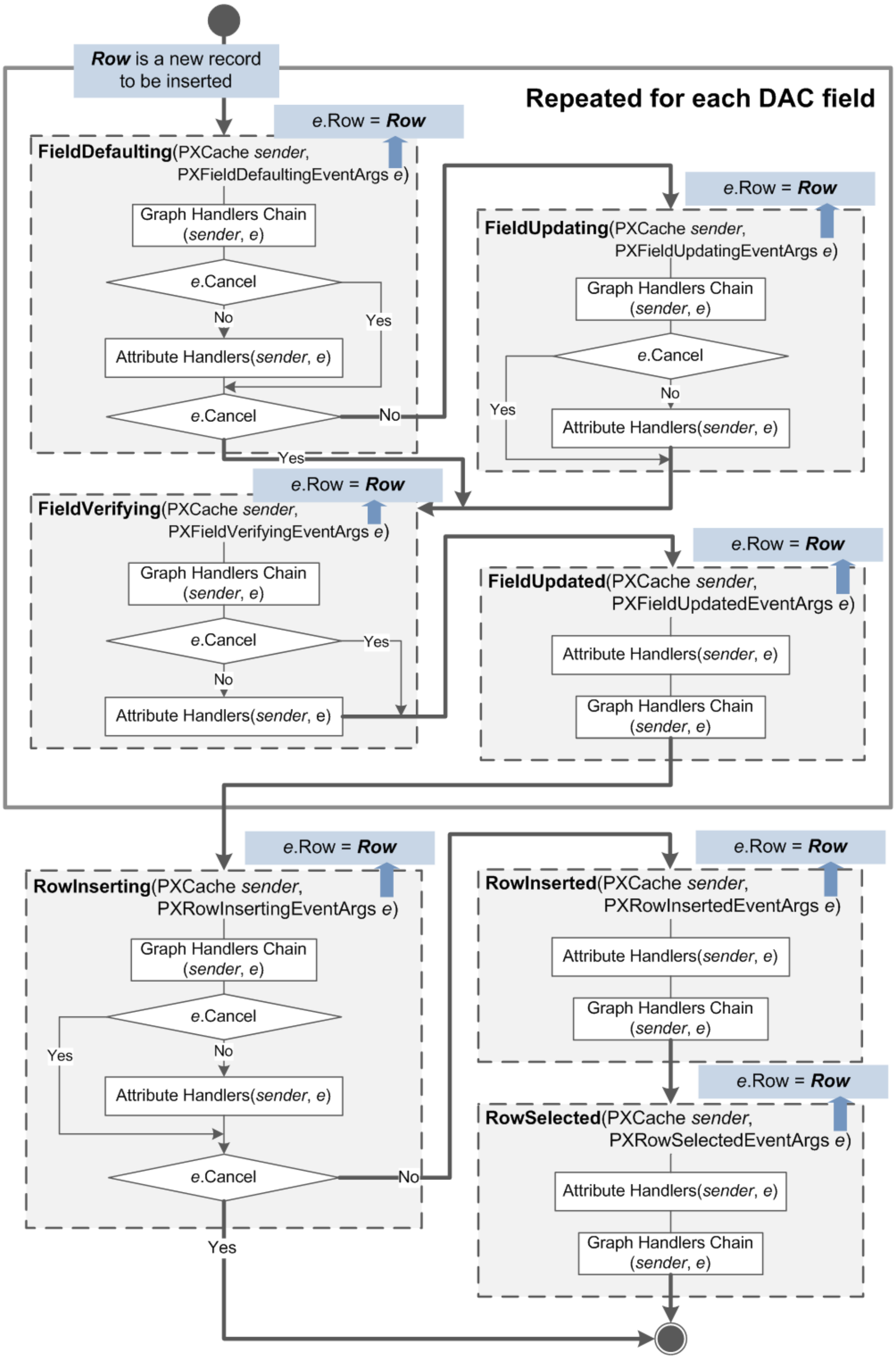


Figure: Inserting a data record

The system inserts a data record—as an instance of a data access class (DAC)—when a user creates a new data record in the user interface (UI), a request is sent to the Web Service API, or, in code, the `Insert()` method of a data view is called. The data record is actually inserted into the `PXCache` object that corresponds to the DAC of the data record. An inserted data record has the `Inserted` status and is available through the `Inserted` and `Dirty` collections of the `PXCache` object.

When a data record is inserted, data field events are raised for each data field in the following order:

- `FieldDefaulting`
- If the `e.Cancel` property equals `true`, `FieldUpdating`
- `FieldVerifying`
- `FieldUpdated`

Next, the following data record events are raised:

- `RowInserting`
- If the `e.Cancel` property dosn't equal `true`:
    - `RowInserted`
    - `RowSelected`

The instance of the inserted data record is available in the `e.Row` property of event arguments.

Updating a Data Record

The sequence of events raised during the update of a data record is illustrated in the figure below.

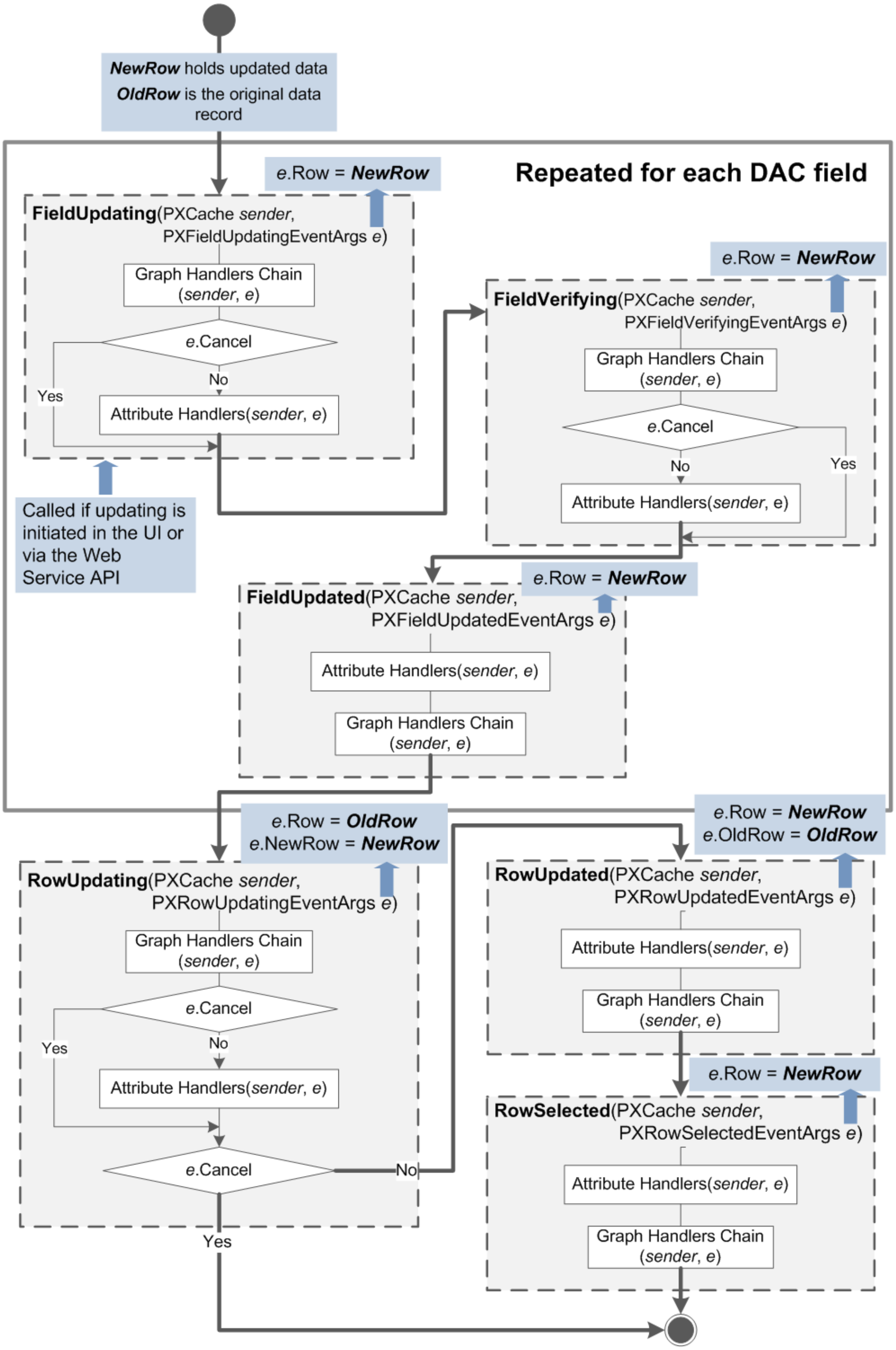


Figure: Updating a data record

A data record is updated when a user modifies the data record on the user interface (UI), the request is sent through the Web Service API, or the `Update()` method is invoked on the data view. Updated data records, which the system gives the `Updated` status, are later available through the `Updated` and `Dirty` collections of the appropriate `PXCache` object.

The `RowUpdating` and `RowUpdated` events are fired before the update happens and after the update happens, respectively. The developer can handle these events and has access to the updated data record and the previous version of the data record that is kept in the PXCache object. The actual update happens between these two events when the data record is copied to the `PXCache` object.

When a data record is updated, the following data field events are raised for each updated data field:

- `FieldUpdating`
- `FieldVerifying`
- `FieldUpdated`

Next, data record events are raised as follows:

- `RowUpdating` is raised. At this moment, in the `e` variable representing event data, `e.Row` holds the data record version from the cache, while `e.NewRow` holds the updated data record. You can still stop updating by throwing a `PXException` instance.
- If `e.Cancel` doesn't equal `true`:
    - `RowUpdated` is raised. `e.Row` now holds the updated instance, while the `e.OldRow` holds a copy of the old data record with old values.
    - `RowSelected` is raised. Only the updated data record can be accessed through `e.Row`.

Deleting a Data Record

The sequence of events raised during the deletion of a data record is illustrated in the figure below.

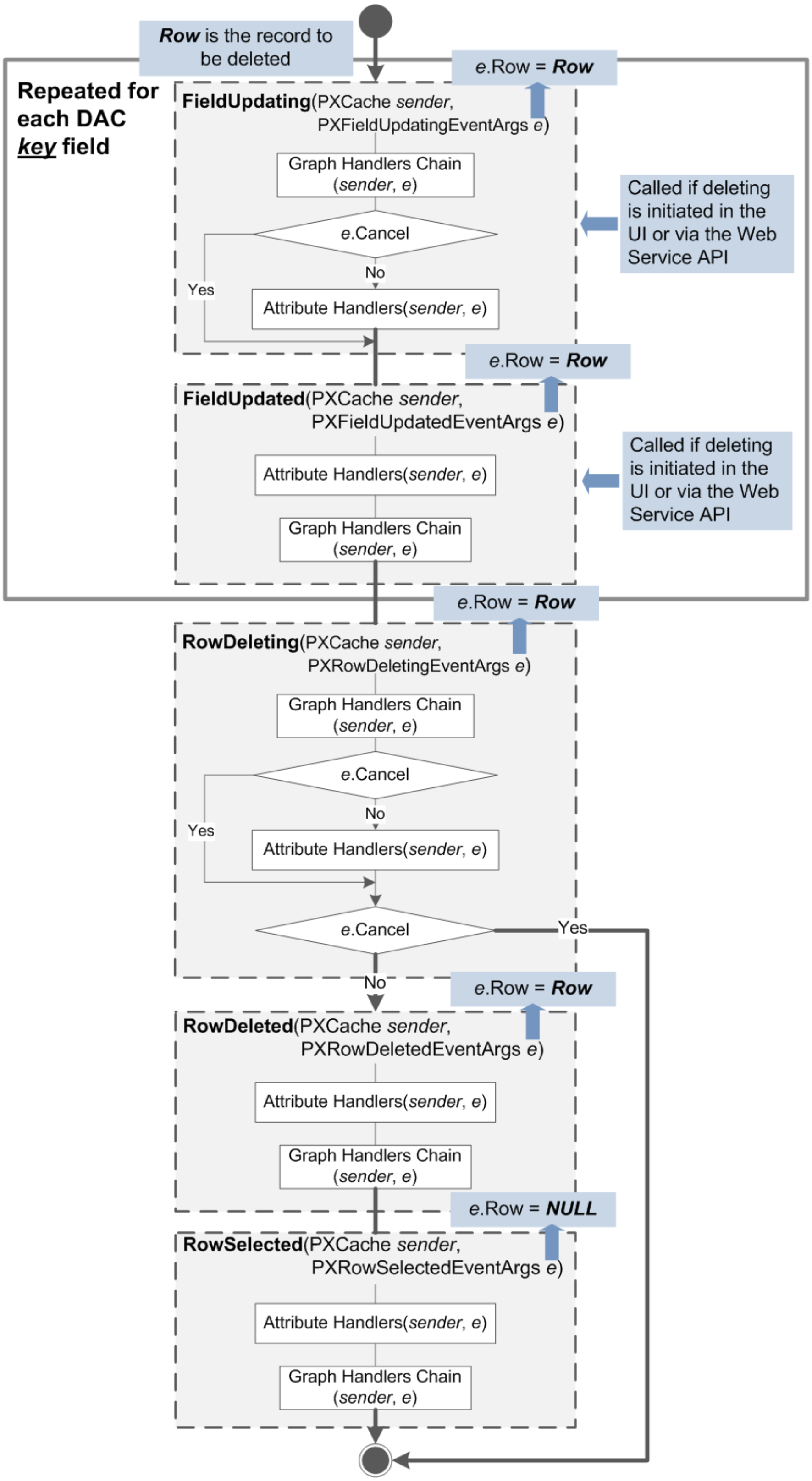


Figure: Deleting a data record

A data record is deleted when a user deletes the record on the user interface (UI), the request is sent through the Web Service API, or the `Delete()` method of a data view is invoked in code. As a result of the deletion, the data record gets the `Deleted` status, if it already exists in the database, or the

InsertedDeleted status, if the record has just been inserted into the PXCache object and the deletion from the database is not required. The data record is later available through the Deleted and Dirty collections of the PXCache object.

If the deletion has been initiated by a user on the UI or through the Web Service API, first, the following field events are raised for each *key* data field:

- FieldUpdating
- FieldUpdated

Next, data record events are raised as follows:

- RowDeleting is raised. At this point, the developer can still stop the deleting by throwing a PXException instance. In the e variable representing event data, e.Row holds the data record being deleted.
- If e.Cancel doesn't equal true:
    - RowDeleted is raised, and e.Row still holds the data record.
    - RowSelected is raised, and e.Row equals NULL.

Displaying a Data Record

Each time a data record is displayed in the user interface (UI) or retrieved through the Web Service API, the RowSelected event is raised, as well as the FieldSelecting event, for each data field. For both events, the e.Row property of event arguments holds the data record that is being displayed or retrieved.

This process is illustrated in more detail in the diagram below.

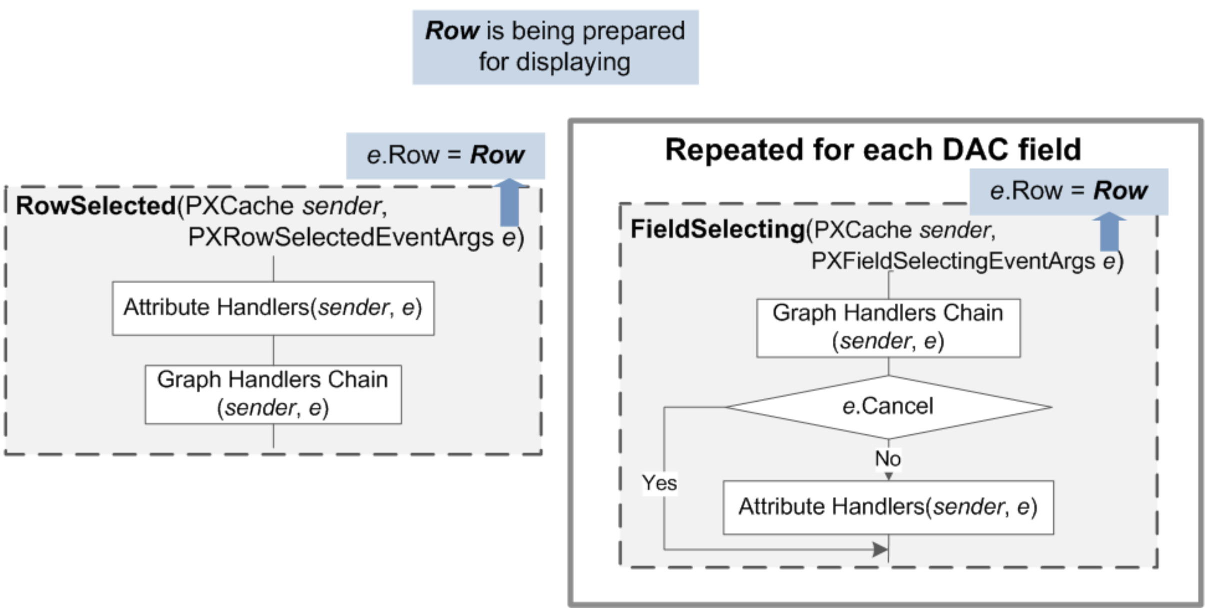


Figure: Displaying a data record

Saving Changes to the Database

The sequence of events raised during the saving of a data record is illustrated in the figure below.

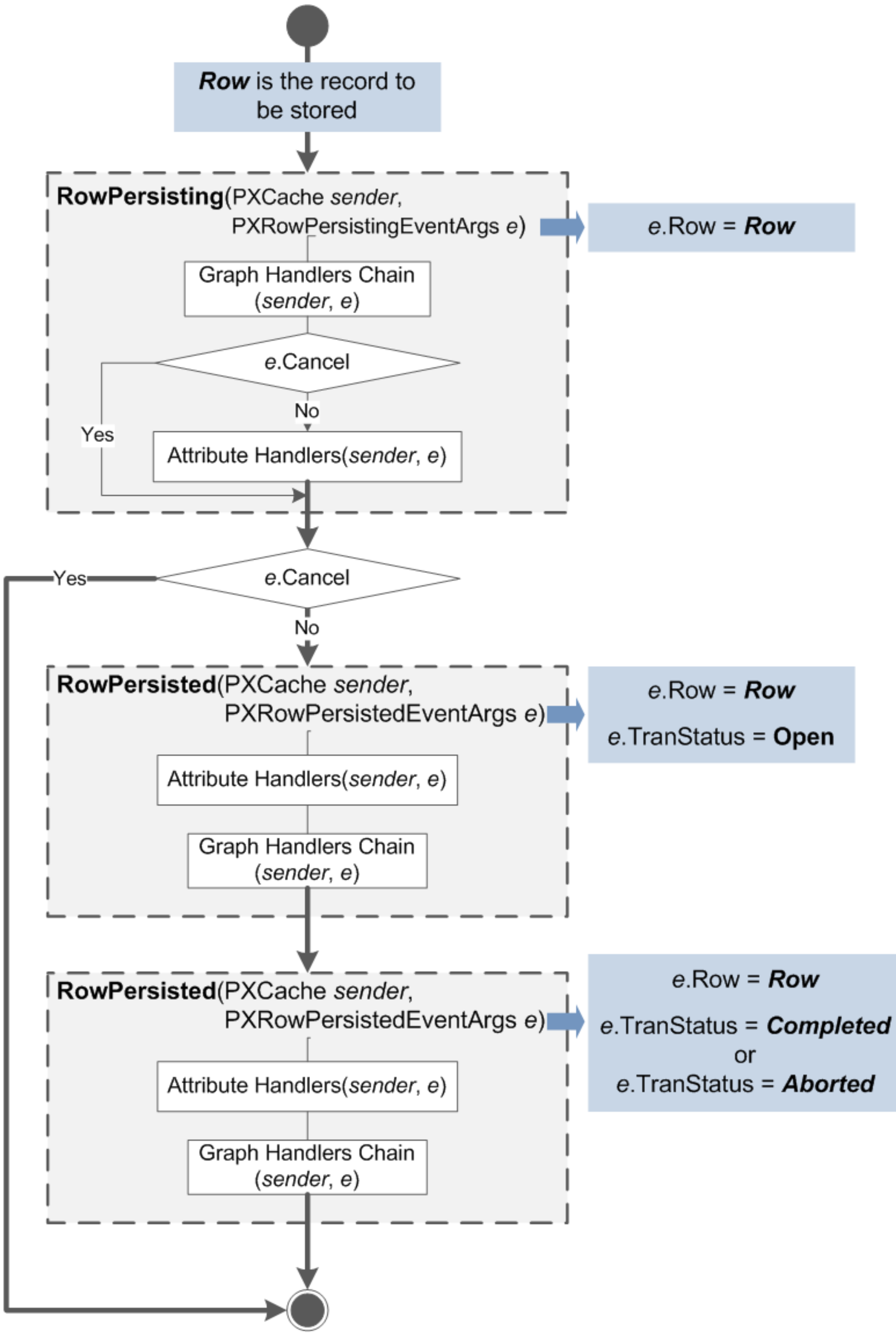


Figure: Committing a data record to the database

While a user is inserting, updating, or deleting a data record, no changes are committed to the database. The system stores the modified data records in the session, and you can access them through the appropriate `PXCache` object. The system commits the changes to the database when the user presses *Save* in the user interface (UI), the request is sent through the Web Service API, or `Actions.PressSave()` is invoked on the business logic controller (BLC) instance.

During the process of saving changes to the database, events are raised as follows:

- `RowPersisting` is raised. By this moment, a database transaction has already been opened. If any of the handlers sets `e.Cancel` to `true`, the process will be canceled for the currently processed data record, without an error being reported to the user. To cancel the whole process of committing changes and indicate the error to the user, you should throw an instance of `PXException`.

- If `e.Cancel` doesn't equal `true`:
    - `RowPersisted` is raised. The committing operation for the current data record (available through `e.Row` in the handler) is completed, but the transaction is still open: `e.TranStatus` equals `Open`.
    - `RowPersisted` is raised one more time, either with `e.TranStatus` equal to `Completed` (if all changes have been saved successfully) or with `e.TranStatus` equal to `Aborted` if an error occurred and all changes have been canceled.

Events

This section includes reference information on all events as well as on classes and enumerations related to only one particular event (such as the event arguments class).

See below for the lists, by categories, of all events:

- Data field events:
    - *FieldDefaulting Event*
    - *FieldVerifying Event*
    - *FieldUpdating Event*
    - *FieldUpdated Event*
    - *FieldSelecting Event*
- Data record events:
    - *RowSelected Event*
    - *RowInserting Event*
    - *RowInserted Event*
    - *RowUpdating Event*
    - *RowUpdated Event*
    - *RowDeleting Event*
    - *RowDeleted Event*
- Database-related events:
    - *CommandPreparing Event*
    - *RowSelecting Event*
    - *RowPersisting Event*
    - *RowPersisted Event*
- Exception-handling event:
    - *ExceptionHandling Event*
- Event for overriding DAC field attributes:
    - *CacheAttached Event*

FieldDefaulting Event

The `FieldDefaulting` event is triggered:

- When a user's action on the user interface (UI) or a Web Service application programming interface (API) call causes insertion of a new record into the PXCache object.
- When any of the following methods of the PXCache class initiates assigning a field its default value:
    - Insert()
    - Insert(object)
    - Insert(IDictionary)
    - SetDefaultExt(object, string)
    - SetDefaultExt<Field>(object)

The FieldDefaulting event handler is used to generate and assign the default value to a data access class (DAC) field.

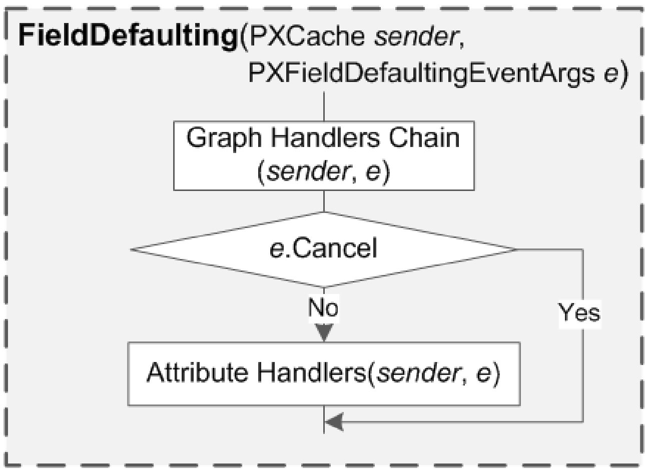


Figure: Execution order for FieldDefaulting event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_FieldName_FieldDefaulting(
    PXCache sender,
    PXFieldDefaultingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* PXCache sender

  The cache object that raised the event
- *(required)* PXFieldDefaultingEventArgs e

  The instance of the *PXFieldDefaultingEventArgs* type that holds data for the FieldDefaulting event

Examples of Use

The code below generates the default value for a DAC field.

```
public class POOrderEntry : PXGraph<POOrderEntry, POOrder>,
                            PXImportAttribute.IPXPrepareItems
{
    ...

    protected virtual void POOrder_ExpectedDate_FieldDefaulting(
        PXCache sender,
        PXFieldDefaultingEventArgs e)
    {
        POOrder row = (POOrder)e.Row;
        Location vendorLocation = this.location.Current;
        if (row != null && row.OrderDate.HasValue)
        {
            int offset = (vendorLocation != null ?
                         (int)(vendorLocation.VLeadTime ?? 0) : 0);
            e.NewValue = row.OrderDate.Value.AddDays(offset);
        }
    }

    ...
}
```

Related Types

- *PXFieldDefaultingEventArgs Class*

PXFieldDefaultingEventArgs Class

Provides data for the *FieldDefaulting* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXFieldDefaultingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the current DAC object.
- `public object NewValue`

  Gets or sets the default value for the DAC field.
- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `FieldDefaulting` event handlers specified within the DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

FieldVerifying Event

The system triggers the `FieldVerifying` event for each data access class (DAC) field of a data record that is inserted or updated in the `PXCache` object in the process of:

- Insertion or update initiated in the user interface (UI) or through the Web Service application programming interface (API).

- Any of the following methods of the `PXCache` class initiates the assignment of the default value to the DAC field:
  - `Insert()`
  - `Insert(object)`
  - `Insert(IDictionary)`
  - `SetDefaultExt(object, string)`
  - `SetDefaultExt<Field>(object)`
- A DAC field update that is initiated by any of the following methods of the `PXCache` class:
  - `Update(object)`
  - `Update(IDictionary, IDictionary)`
  - `SetValueExt(object, string, object)`
  - `SetValueExt<Field>(object, object)`
- Validation of a DAC *key* field value when the validation is initiated by any of the following methods of the `PXCache` class:
  - `Locate(IDictionary)`
  - `Update(IDictionary, IDictionary)`

The `FieldVerifying` event handler is used to:

- Implement the business logic associated with validation of the DAC field value before the value is assigned to the DAC field.
- Cancel the assigning of a value by throwing an exception of `PXSetPropertyException` type—if the value does not fit the requirements.
- Convert the external presentation of a DAC field value to the internal presentation and implement the associated business logic. The internal presentation is the value stored in a DAC instance.

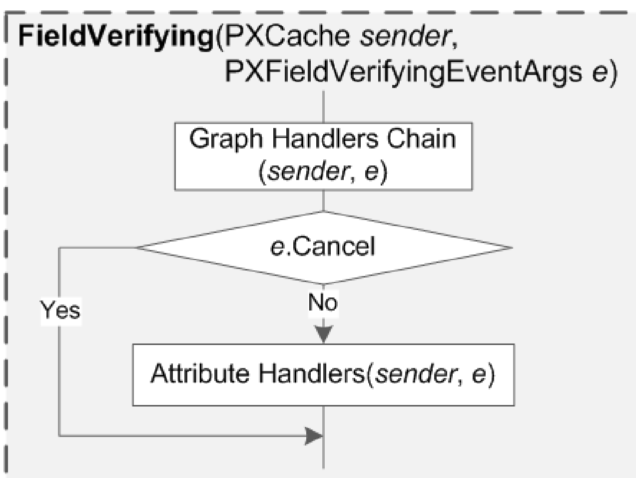


Figure: Execution order for FieldVerifying event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_FieldName_FieldVerifying(
```

```
    PXCache sender,
    PXFieldVerifyingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXFieldVerifyingEventArgs e`

  The instance of the *PXFieldVerifyingEventArgs* type that holds data for the `FieldUpdating` event

Examples of Use

The code below validates the new value of a DAC field.

```
public class APPaymentEntry : APDataEntryGraph<APPaymentEntry, APPayment>
{
    ...

    protected virtual void APPayment_AdjDate_FieldVerifying(
        PXCache sender,
        PXFieldVerifyingEventArgs e)
    {
        if ((bool)((APPayment)e.Row).VoidAppl == false &&
            vendor.Current != null && (bool)vendor.Current.Vendor1099)
        {
            string Year1099 = ((DateTime)e.NewValue).Year.ToString();
            AP1099Year year = PXSelect<
                AP1099Year,
                Where<AP1099Year.finYear,
                      Equal<Required<AP1099Year.finYear>>>>.
                Select(this, Year1099);
            if (year != null && year.Status != "N")
                throw new PXSetPropertyException(
                    Messages.AP1099_PaymentDate_NotIn_OpenYear,
                    PXUIFieldAttribute.
                        GetDisplayName<APPayment.adjDate>(sender));
        }
    }

    ...
}
```

The code below validates the external presentation of a DAC field value and converts it to the internal presentation if it is acceptable.

```
[TableAndChartDashboardType]
public class CAReconEnq : PXGraph<CAReconEnq>
{
    ...

    protected virtual void CashAccountFilter_CashAccountID_FieldVerifying(
        PXCache sender,
        PXFieldVerifyingEventArgs e)
    {
        CashAccountFilter createReconFilter = (CashAccountFilter)e.Row;
        if (!e.NewValue is string) return;
        CashAccount acct =
            PXSelect<CashAccount,
                     Where<CashAccount.accountCD,
                           Equal<Required<CashAccount.accountCD>>>>.
```

```
                    Select(this, (string)e.NewValue);
            if (acct != null && acct.Reconcile != true)
                throw new PXSetPropertyException(Messages.CashAccounNotReconcile);
            e.NewValue = acct.AccountID;
        }

        ...
}
```

Related Types

- *PXFieldVerifyingEventArgs Class*

PXFieldVerifyingEventArgs Class

Provides data for the *FieldVerifying* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXFieldVerifyingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the current DAC object.

- `public object NewValue`

  Gets or sets the new value of the current DAC field.

- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `FieldVerifying` event handlers specified within the DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

- `public bool ExternalCall`

  Gets the value specifying if the new value of the current DAC field has been received from the UI or through the Web Service API.

FieldUpdating Event

In the following cases, the `FieldUpdating` event is triggered for a data access class (DAC) field before the field is updated:

- For each DAC field value received from the user interface (UI) or through the Web Service application programming interface (API) when a data record is being inserted or updated.
- For each DAC *key* field value in the process of deleting a data record when the deletion is initiated from the UI or through the Web Service API.
- While any of the following methods of the `PXCache` class initiates assigning a field its default value:
  - `Insert()`
  - `Insert(object)`
  - `Insert(IDictionary)`

    - `SetDefaultExt(object, string)`
    - `SetDefaultExt<Field>(object)`
- While any of the following methods of the `PXCache` class initiates updating a field:
    - `Update(IDictionary, IDictionary)`
    - `SetValueExt(object, string, object)`
    - `SetValueExt<Field>(object, object)`
    - `SetValuePending(object, string, object)`
    - `SetValuePending<Field>(object, object)`
- During conversion of the external DAC key field presentation to the internal field value, initiated by the following `PXCache` class methods:
    - `Locate(IDictionary)`
    - `Update(IDictionary, IDictionary)`
    - `Delete(IDictionary, IDictionary)` methods

The `FieldUpdating` event handler is used when either or both of the following occur:

- The external presentation of a DAC field (the value displayed in the UI) differs from the value stored in the DAC.
- Value storage is spread among several DAC fields (database columns).

In both cases, the application should implement both the `FieldUpdating` and *FieldSelecting* events.

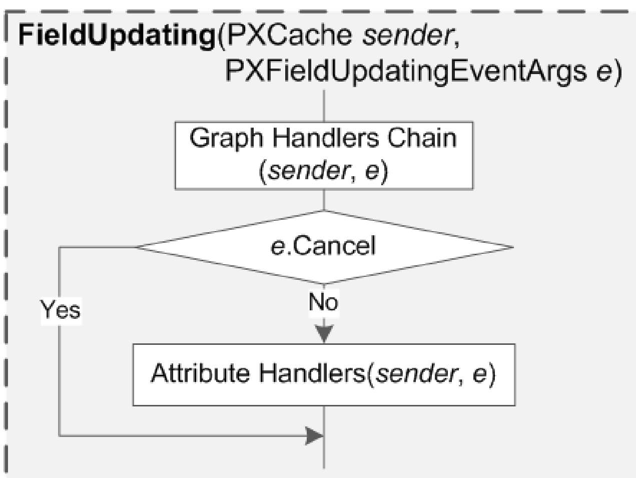


Figure: Execution order for FieldUpdating event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_FieldName_FieldUpdating(
    PXCache sender,
    PXFieldUpdatingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXFieldUpdatingEventArgs e`

  The instance of the *PXFieldUpdatingEventArgs* type that holds data for the `FieldUpdating` event

Examples of Use

The code below spreads the external presentation of a field among multiple DAC fields.

```
protected void Batch_ManualStatus_FieldUpdating(PXCache sender,
 PXFieldUpdatingEventArgs e)
{
    Batch batch = (Batch)e.Row;
    if (batch != null && e.NewValue != null)
    {
        switch ((string)e.NewValue)
        {
            case "H":
                batch.Hold = true;
                batch.Released = false;
                batch.Posted = false;
                break;
            case "B":
                batch.Hold = false;
                batch.Released = false;
                batch.Posted = false;
                break;
            case "U":
                batch.Hold = false;
                batch.Released = true;
                batch.Posted = false;
                break;
            case "P":
                batch.Hold = false;
                batch.Released = true;
                batch.Posted = true;
                break;
        }
   }
}

protected void Batch_ManualStatus_FieldSelecting(PXCache sender,
 PXFieldSelectingEventArgs e)
{
    Batch batch = (Batch)e.Row;
    if (batch != null)
    {
        if (batch.Hold == true)
        {
            e.ReturnValue = "H";
        }
        else if (batch.Released != true)
        {
            e.ReturnValue = "B";
        }
        else if (batch.Posted != true)
        {
            e.ReturnValue = "U";
        }
        else
        {
            e.ReturnValue = "P";
        }
    }
```

```
}
```

Related Types

- *PXFieldUpdatingEventArgs Class*
- *PXEntryStatus Enumeration*

PXFieldUpdatingEventArgs Class

Provides data for the *FieldUpdating* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXFieldUpdatingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the current DAC object.
- `public object NewValue`

  Gets or sets the internal DAC field value.
- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `FieldUpdating` event handlers specified within the DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

FieldUpdated Event

In the following cases, the `FieldUpdated` event is triggered after a data access class (DAC) field is actually updated:

- For each DAC field value received from the user interface (UI) or through the Web Service application programming interface (API) when a data record is inserted or updated in the `PXCache` object
- For each DAC *key* field value in the process of deleting a data record from the `PXCache` object when the deletion is initiated from the UI or through the Web Service API
- While any of the following methods of the `PXCache` class initiates assigning a field its default value:
  - `Insert()`
  - `Insert(object)`
  - `Insert(IDictionary)`
  - `SetDefaultExt(object, string)`
  - `SetDefaultExt<Field>(object)`
- While a field is updated in the `PXCache` object, initiated by any of the following methods of the `PXCache` class:
  - `Update(object)`

    - `SetValueExt(object, string, object)`
    - `SetValueExt<Field>(object, object)`
- During validation of the DAC *key* field value initiated by any of the following `PXCache` class methods:
    - `Locate(IDictionary)`
    - `Update(IDictionary, IDictionary)`
    - `Delete(IDictionary, IDictionary)`

The `FieldUpdated` event handler is used to implement the business logic associated with changes to the value of the DAC field in the following cases:

- Assigning the related fields of the data record containing the modified field their default values or updating them
- Updating any of the following:
    - The detail data records in a one-to-many relationship
    - The related data records in a one-to-one relationship
    - The master data records in a many-to-one relationship

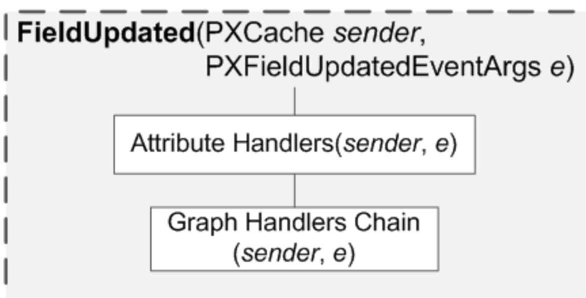


Figure: Execution order for FieldUpdated event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_FieldName_FieldUpdated(
    PXCache sender,
    PXFieldUpdatedEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXFieldUpdatedEventArgs e`

  The instance of the *PXFieldUpdatedEventArgs* type that holds data for the `FieldUpdated` event

Examples of Use

The code below updates the related field values of the current data record, assigns them the default values, or performs both actions.

```
public class APInvoiceEntry : APDataEntryGraph<APInvoiceEntry,
                              APInvoice>,
                              PXImportAttribute.IPXPrepareItems
{
    ...

    protected virtual void APTran_UOM_FieldUpdated(
        PXCache sender,
        PXFieldUpdatedEventArgs e)
    {
        APTran tran = (APTran)e.Row;
        sender.SetDefaultExt<APTran.unitCost>(tran);
        sender.SetDefaultExt<APTran.curyUnitCost>(tran);
        sender.SetValue<APTran.unitCost>(tran, null);
    }

    ...
}
```

The code below updates the related data records.

```
public class ARCashSaleEntry : ARDataEntryGraph<ARCashSaleEntry,
                                                 ARCashSale>
{
    ...

    protected virtual void ARCashSale_ProjectID_FieldUpdated(
        PXCache sender,
        PXFieldUpdatedEventArgs e)
    {
        ARCashSale row = e.Row as ARCashSale;

        foreach (ARTran tran in Transactions.Select())
            Transactions.Cache.SetDefaultExt<ARTran.projectID>(tran);
    }

    ...
}
```

Related Types

- *PXFieldUpdatedEventArgs Class*
- *PXEntryStatus Enumeration*

PXFieldUpdatedEventArgs Class

Provides data for the *FieldUpdated* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXFieldUpdatedEventArgs : CancelEventArgs
```

Properties

- public object Row

  Gets the current DAC object

- public object OldValue

  Gets the previous value of the current DAC field

- public bool ExternalCall

  Gets the value specifying whether the new value of the current DAC field has been changed in the UI or through the Web Service API

FieldSelecting Event

The FieldSelecting event is triggered:

- When the external representation—the way the value should be displayed in the user interface (UI)—of a data access class (DAC) field value is requested from the UI or through the Web Service application programming interface (API).
- When any the following methods of the PXCache class initiates assigning a field its default value:
  - Insert()
  - Insert(object)
  - Insert(IDictionary)
- While a field is updated in the PXCache object, initiated by any the following methods of the PXCache class:
  - Update(object)
  - Update(IDictionary, IDictionary)
- While a DAC field value is requested through any of the following methods of the PXCache class:
  - GetValueInt(object, string)
  - GetValueInt<Field>(object)
  - GetValueExt(object, string)
  - GetValueExt<Field>(object)
  - GetValuePending(object, string)
  - ToDictionary(object)
  - GetStateExt(object, string)
  - GetStateExt<Field>(object)

The FieldSelecting event handler is used to:

- Convert the internal presentation of a DAC field (the data field value of a DAC instance) to the external presentation (the value displayed in the UI).
- Convert the values of multiple DAC fields to a single external presentation.
- Provide additional information to set up a DAC field input control or cell presentation.

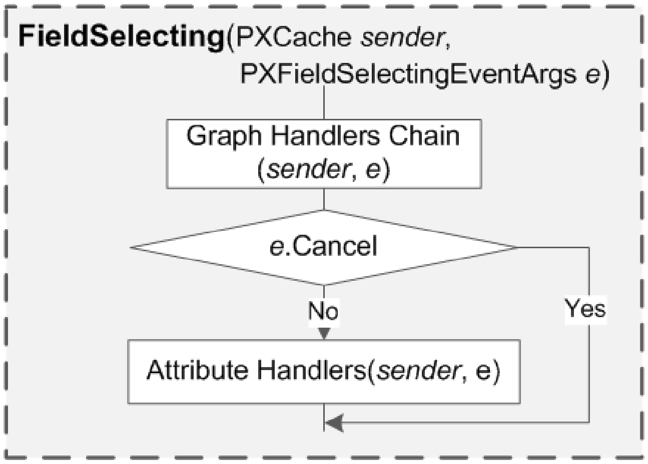


Figure: Execution order for `FieldSelecting` event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_FieldName_FieldSelecting(
    PXCache sender,
    PXFieldSelectingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXFieldSelectingEventArgs e`

  The instance of the *PXFieldSelectingEventArgs* type that holds data for the `FieldSelecting` event.

Examples of Use

The code below converts the DAC field value to its external presentation.

```
public class PXTimeSpanLongAttribute : PXIntAttribute
{
    ...

    public override void FieldSelecting(PXCache sender,
                                        PXFieldSelectingEventArgs e)
    {
        if (_AttributeLevel == PXAttributeLevel.Item || e.IsAltered)
        {
            string inputMask = this.inputMask ??
                                  _inputMasks[(int)this._Format];
            int lenght = this.inputMask != null ? _maskLenght :
                                              _lengths[(int)this._Format];
            inputMask = PXMessages.LocalizeNoPrefix(inputMask);
            e.ReturnState = PXStringState.CreateInstance(
                e.ReturnState,
                lenght,
                null,
```

```
                _FieldName,
                _IsKey,
                null,
                String.IsNullOrEmpty(inputMask) ? null : inputMask,
                null, null, null, null);
        }
        if (e.ReturnValue != null)
        {
            TimeSpan span = new TimeSpan(0, 0, (int)e.ReturnValue, 0);
            int hours =
                (this._Format == TimeSpanFormatType.LongHoursMinutes) ?
                span.Days * 24 + span.Hours :  span.Hours;
            e.ReturnValue = string.Format(_outputFormats[(int)this._Format],
                                          span.Days, hours, span.Minutes);
        }
    }

    ...
}
```

The *example* related to `FieldUpdating` demonstrates the conversion of multiple DAC field values into external presentation in a single field.

The code below calculates the external value of a DAC field.

```
[TableAndChartDashboardType]
public class RevalueAPAccounts : PXGraph<RevalueAPAccounts>
{
    ...

    protected virtual void RevalueFilter_TotalRevalued_FieldSelecting(
        PXCache sender,
        PXFieldSelectingEventArgs e)
    {
        if (e.Row == null) return;

        decimal val = 0m;
        foreach (RevaluedAPHistory res in APAccountList.Cache.Updated)
            if ((bool)res.Selected)
                val += (decimal)res.FinPtdRevalued;
        e.ReturnValue = val;
        e.Cancel = true;
    }

    ...
}
```

The code below defines the mask for the input control or cell presentation of a DAC field.

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Parameter |
                AttributeTargets.Class | AttributeTargets.Method)]
public class PXDBStringWithMaskAttribute : PXDBStringAttribute,
                                           IPXFieldSelectingSubscriber
{
    ...

    public override void FieldSelecting(PXCache sender,
                                        PXFieldSelectingEventArgs e)
    {
        if (e.Row == null) return;

        string mask = this.FindMask(sender, e.Row);
        if (!string.IsNullOrEmpty(mask))
            e.ReturnState = PXStringState.CreateInstance(e.ReturnState,
                                                         _Length,
                                                         null,
                                                         _FieldName,
```

```
                                                          _IsKey,
                                                          null,
                                                          mask,
                                                          null, null, null,
                                                          null);
        else
            base.FieldSelecting(sender, e);
    }

    ...
}
```

The code below defines precision for a DAC field input control or cell presentation.

```
public class LSSOShipLine :
   LSSelect<
       SOShipLine, SOShipLineSplit, SOShipLineSplit.uOM,
       Where<SOShipLineSplit.shipmentNbr,
             Equal<Current<SOShipLine.shipmentNbr>>,
           And<SOShipLineSplit.inventoryID,
               Equal<Current<INLotSerialStatus.inventoryID>>,
           And<SOShipLineSplit.siteID,
               Equal<Current<INLotSerialStatus.siteID>>,
           And<SOShipLineSplit.subItemID,
               Equal<Current<INLotSerialStatus.subItemID>>,
           And<SOShipLineSplit.locationID,
               Equal<Current<INLotSerialStatus.locationID>>,
           And<SOShipLineSplit.lotSerialNbr,
               Equal<Current<INLotSerialStatus.lotSerialNbr>>>>>>>>>
{
    ...

    protected virtual void OrigOrderQty_FieldSelecting(
        PXCache sender,
        PXFieldSelectingEventArgs e)
    {
        e.ReturnState =
            PXDecimalState.CreateInstance(
                e.ReturnState,
                ((INSetup)_Graph.Caches[typeof(INSetup)].Current).DecPlQty,
                _OrigOrderQtyField,
                false,
                0,
                decimal.MinValue,
                decimal.MaxValue);
        ((PXFieldState)e.ReturnState).DisplayName =
            PXMessages.LocalizeNoPrefix(Messages.OrigOrderQty);
        ((PXFieldState)e.ReturnState).Enabled = false;
    }

    ...
}
```

The code below defines lists of values and labels for the `PXDropDown` input control of the DAC field.

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Class |
                AttributeTargets.Parameter | AttributeTargets.Method)]
[PXAttributeFamily(typeof(PXBaseListAttribute))]
public class PXStringListAttribute : PXEventSubscriberAttribute,
                                     IPXFieldSelectingSubscriber
{
    ...

    public virtual void FieldSelecting(PXCache sender,
                                       PXFieldSelectingEventArgs e)
    {
        if (_AttributeLevel == PXAttributeLevel.Item || e.IsAltered)
```

```
        {
            string[] values = _AllowedValues;
            e.ReturnState = PXStringState.CreateInstance(
                e.ReturnState, null, null, _FieldName,
                null, -1, null, values, _AllowedLabels,
                _ExclusiveValues, null);
        }
    }

    ...
}
```

Related Types

- *PXFieldSelectingEventArgs Class*
- *PXFieldState Class*
    - *PXStringState Class*
    - *PXSegmentedState Class*
        - *PXSegment Class*
    - *PXDoubleState Class*
    - *PXFloatState Class*
    - *PXDecimalState Class*
    - *PXDateState Class*
    - *PXIntState Class*
    - *PXGuidState Class*
    - *PXLongState Class*
- *PXUIVisibility Enumeration*
- *PXErrorLevel Enumeration*
- *PXErrorHandling Enumeration*

PXFieldSelectingEventArgs Class

Provides data for the *FieldSelecting* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXFieldSelectingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the current DAC object.
- `public object ReturnState`

  Gets or sets the data used to set up DAC field input control or cell presentation.
- `public bool IsAltered`

Gets or sets the value indicating whether the `ReturnState` property should be created for each data record.

- `public object ReturnValue`

  Gets or sets the external presentation of the value of the DAC field.

- `public bool ExternalCall`

  Gets the value specifying if the current DAC field has been selected in the UI or through the Web Service API.

- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `FieldSelecting` event handlers specified within the DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

PXFieldState Class

Provides data to set up a DAC field input control or cell presentation.

Inherits

`IDataSourceFieldSchema, ICloneable`

Syntax

```
public class PXFieldState : IDataSourceFieldSchema, ICloneable
```

Properties

- `public virtual Type DataType`

  Gets the type of data stored in the field.

- `public virtual bool Identity`

  Gets the value indicating whether the field is mapped to an identity column in a database table.

- `public virtual bool IsReadOnly`

  Gets the value indicating whether the field is read-only.

- `public virtual bool IsUnique`

  Gets the indication of a uniqueness constraint on the field.

- `public virtual int Length`

  Gets or sets the storage size of the field.

- `public virtual string Name`

  Gets the name of the field.

- `public virtual bool Nullable`

  Gets the value indicating whether the field can store the `null` value.

- `public virtual int Precision`

  Gets the maximum number of digits used to represent a numeric value stored in the field.

- `public virtual int Scale`

  Gets the number of digits to the right of the decimal point used to represent a numeric value stored in the field.

- `public virtual bool? Required`

  Gets or sets the value indicating whether the value of the field is required.
- `public virtual object Value`

  Gets or sets the value stored in the field.
- `public virtual string Error`

  Gets or sets the error text assigned to the field.
- `public virtual bool IsWarning`

  Gets or sets the value indicating whether the field is marked with the `Warning` sign.
- `public virtual PXErrorLevel ErrorLevel`

  Gets or sets the error level assigned to the field.
- `public virtual bool Enabled`

  Gets or sets the value indicating whether the current field input control or cell will respond to a user's interaction.
- `public virtual bool Visible`

  Gets or sets the value indicating whether the current field input control or column is displayed.
- `public virtual string DisplayName`

  Gets or sets the display name for the field.
- `public virtual string DescriptionName`

  Gets or sets the name of a DAC field displayed in the `PXSelector` control of the field if the `DisplayMode` property is set to `Text`. If the `DisplayMode` property is set to `Hint`, the name is displayed in the `ValueField` - `DescriptionName` format. By default, `DisplayMode` is set to `Hint`.
- `public virtual PXUIVisibility Visibility`

  Gets or sets the `PXUIVisibility` object for the field.
- `public virtual object DefaultValue`

  Gets or sets the default value that is displayed in the field's cell for a new record that is not yet committed to the `PXGraph` instance.
- `public virtual string ViewName`

  Gets or sets the name for the `PXView` object bound to the `PXSelector` field control.
- `public virtual string[] FieldList`

  Gets or sets the array of DAC fields for the `PXSelector` field control.
- `public virtual string[] HeaderList`

  Gets or sets the array of field display names for the `PXSelector` field control.
- `public virtual string ValueField`

  Gets or sets the name of a DAC field, which is:

  - Displayed in the `PXSelector` field control on focus.
  - Used to locate the selected record in the `PXSelector` field control.
  - Displayed in the `PXSelector` field control when the `DisplayMode` property is set to `Value`.
- `public virtual bool PrimaryKey`

  Gets the value indicating whether the field is marked as a key field.

Methods

- `public void SetFieldName(string)`

  Sets the name of the field.

- `public static PXFieldState CreateInstance(object value, Type dataType, bool? isKey, bool? nullable, int? required, int? precision, int? length, object defaultValue, string fieldName, string descriptionName, string displayName, string error, PXErrorLevel errorLevel, bool? enabled, bool? visible, bool? readOnly, PXUIVisibility visibility, string viewName, string[] fieldList, object value)`

  Creates an instance of the `PXFieldState` class.

- `public PXFieldState CreateInstance(Type dataType, bool? isKey, bool? nullable, int? required, int? precision, int? length, object defaultValue, string fieldName, string descriptionName, string displayName, string error, PXErrorLevel errorLevel, bool? enabled, bool? visible, bool? readOnly, PXUIVisibility visibility, string viewName, string[] fieldList, Type dataType)`

  Creates an instance of the `PXFieldState` class.

- `public static string GetStringValue(PXFieldState state, string fFormat, PXFieldState state)`

  Returns the string representation of the field's value.

  Parameters:

  - `state`

    The `PXFieldState` object of the field.

  - `fFormat`

    The format for a numeric value.

  - `dFormat`

    The format for a `DateTime` value.

- `public static PXFieldState[] GetFields(PXGraph, Type[], PXGraph)`

  Returns the `PXFieldState` objects for the specified `PXGraph` instance and the array of DAC objects.

PXStringState Class

Provides data to set up the `segstringmented` DAC field input control or cell presentation.

Inherits

`PXFieldState`

Syntax

```
public class PXStringState : PXFieldState
```

Properties

- `public virtual string InputMask`

  Gets or sets the value specifying how users enter data and how data is displayed

- `public virtual string[] AllowedValues`

Gets or sets the list of values for the `PXDropDown` field input control

- `public virtual string[] AllowedLabels`

  Gets or sets the list of labels for the `PXDropDown` field input control

- `public virtual string[] AllowedImages`

  Gets or sets the list of images for the `PXDropDown` field input control

- `public virtual bool ExclusiveValues`

  Gets a value that enables or disables editing of the value in the `PXDropDown` field input control

- `public virtual bool IsUnicode`

  Gets or sets a value indicating whether Unicode string content is supported

- `public Dictionary`

  Gets the collection of values and labels for the field PXDropDown input control.

Methods

- `public static PXFieldState CreateInstance(object value, int? length, bool? isUnicode, string fieldName, bool? isKey, int? required, string inputMask, string[] allowedValues, string[] allowedLabels, bool? exclusiveValues, object value)`

  Creates an instance of the PXStringState class

PXSegmentedState Class

Provides data to set up the segmented DAC field input control or cell presentation.

Inherits

`PXStringState`

Syntax

```
public class PXSegmentedState : PXStringState
```

Properties

- `public PXSegment[] Segments`

  Gets or sets the list of segments for the segmented field input control or cell presentation

- `public bool ValidCombos`

  Gets or sets the value indicating whether the segmented field input control displays a single lookup or a separate lookup for each segment

- `public string Wildcard`

  Gets or sets the collection of characters allowed to be specified within each segment in addition to the `Mask` property of `PXSegment`

Methods

- `public static PXFieldState CreateInstance(object value, string fieldName, PXSegment[] segments, string viewName, bool? validCombos, object value)`

  Creates an instance of the `PXSegment` class

PXSegment Class

Provides data to set up a single segment of a segmented field input control or cell presentation.

Syntax

```
public class PXSegment
```

Methods

- `public PXSegment(char editMask, char fillCharacter, short length, bool validate, short caseConverter, short align, char separator, char editMask)`

  Creates an instance of the `PXSegment` class

Fields

- `public readonly char EditMask`

  Gets the input mask for the segment:

  - `C`: `MaskType.Ascii`
  - `a`: `MaskType.AlphaNumeric`
  - `9`: `MaskType.Numeric`
  - `?`: `MaskType.Alpha`
- `public readonly short Length`

  Gets the number of characters in the segment
- `public readonly bool Validate`

  Gets the value indicating whether the new specified segment value should be validated
- `public readonly short CaseConvert`

  Gets the value that specifies whether the letters in the segment are converted to uppercase or lowercase:

  - `0`: `NotSet`
  - `1`: `Upper`
  - `2`: `Lower`
- `public readonly short Align`

  Gets the text alignment type in the segment:

  - `1`: `Left`
  - `2`: `Right`
- `public readonly char Separator`

  Gets the character used to separate the segment from the previous one
- `public readonly bool ReadOnly`

  Gets the value indicating whether the contents of the segment can be changed

PXDoubleState Class

Provides data to set up the `decimal` DAC field input control or cell presentation.

Inherits

`PXFieldState`

Syntax

```
public class PXDoubleState : PXFieldState
```

Properties

- `public virtual double MinValue`

  Gets or sets the minimum value that can be set in the field input control

- `public virtual double MaxValue`

  Gets or sets the maximum value that can be set in the field input control

Methods

- `public static PXFieldState CreateInstance(object value, int? precision, string fieldName, bool? isKey, int? required, double? minValue, object value)`

  Creates an instance of the `PXDoubleState` class

PXFloatState Class

Provides data to set up the `float` DAC field input control or cell presentation.

Inherits

`PXFieldState`

Syntax

```
public class PXFloatState : PXFieldState
```

Properties

- `public virtual double MinValue`

  Gets or sets the minimum value that could be set in the field input control.

- `public virtual double MaxValue`

  Gets or sets the maximum value that could be set in the field input control.

Methods

- `public static PXFieldState CreateInstance(object value, int? precision, string fieldName, bool? isKey, int? required, float? minValue, object value)`

  Creates an instance of the PXFloatState class

PXDecimalState Class

Provides data to set up the `decimal` DAC field input control or cell presentation.

Inherits

`PXFieldState`

Syntax

```
public class PXDecimalState : PXFieldState
```

Properties

- `public virtual double MinValue`

  Gets or sets the minimum value that can be set in the field input control
- `public virtual double MaxValue`

  Gets or sets the maximum value that can be set in the field input control

Methods

- `public static PXFieldState CreateInstance(object value, int? precision, string fieldName, bool? isKey, int? required, decimal? minValue, object value)`

  Creates an instance of the `PXDecimalState` class

PXDateState Class

Provides data to set up the `DateTime` DAC field input control or cell presentation.

Inherits

`PXFieldState`

Syntax

```
public class PXDateState : PXFieldState
```

Properties

- `public virtual string InputMask`

  Gets or sets the value specifying how users enter data
- `public virtual string DisplayMask`

  Gets or sets the value specifying how data is displayed
- `public virtual DateTime MinValue`

  Gets or sets the minimum value that can be set in the field input control
- `public virtual DateTime MaxValue`

  Gets or sets the maximum value that can be set in the field input control

Methods

- `public static PXFieldState CreateInstance(object value, string fieldName, bool? isKey, int? required, string inputMask, string displayMask, DateTime? minValue, object value)`

  Creates an instance of the `PXDateState` class

PXIntState Class

Provides data to set up the `integer` DAC field input control or cell presentation.

Inherits

PXFieldState

Syntax

```
public class PXIntState : PXFieldState
```

Properties

- `public virtual int MinValue`

  Gets or sets the minimum value that could be set in the field input control
- `public virtual int MaxValue`

  Gets or sets the maximum value that could be set in the field input control
- `public virtual string[] AllowedValues`

  Gets or sets the list of values for the field input control of the `PXDropDown` type
- `public virtual string[] AllowedLabels`

  Gets or sets the list of labels for the field input control of the `PXDropDown` type
- `public virtual string[] AllowedImages`

  Gets or sets the list of images for the field input control of the `PXDropDown` type

Methods

- `public static PXFieldState CreateInstance(object value, string fieldName, bool? isKey, int? required, int? minValue, int? maxValue, int[] allowedValues, string[] allowedLabels, Type dataType, object value)`

  Creates an instance of the `PXIntState` class

PXGuidState Class

Provides data to set up the `Guid` DAC field input control or cell presentation.

Inherits

PXFieldState

Syntax

```
public class PXGuidState : PXFieldState
```

Methods

- `public static PXFieldState CreateInstance(object value, string fieldName, bool? isKey, object value)`

  Creates an instance of the `PXGuidState` class

PXLongState Class

Provides data to set up the `long` DAC field input control or cell presentation.

Inherits

PXFieldState

Syntax

```
public class PXLongState : PXFieldState
```

Properties

- `public virtual double MinValue`

  Gets or sets the minimum value that could be set in the field input control
- `public virtual double MaxValue`

  Gets or sets the maximum value that could be set in the field input control

Methods

- `public static PXFieldState CreateInstance(object value, string fieldName, bool? isKey, int? required, long? minValue, long? maxValue, long[] allowedValues, string[] allowedLabels, object value)`

  Creates an instance of the `PXLongState` class

RowSelected Event

The `RowSelected` event is triggered in the process of:

- Displaying a data record in the user interface (UI).
- Execution of the following methods of the `PXCache` class:
  - `Locate(IDictionary)`
  - `Insert()`
  - `Insert(object)`
  - `Insert(IDictionary)`
  - `Update(object)`
  - `Update(IDictionary, IDictionary)`
  - `Delete(IDictionary, IDictionary)`

Avoid executing BQL statements in a `RowSelected` event handler, because this execution may cause performance degradation because of multiple invocations of the `RowSelected` event for a single data record.

The `RowSelected` event handler is used to:

- Implement the UI presentation logic.
- Set up the processing operation on a processing screen (a type of UI screen that allows the execution of a long-running operation on multiple data records at once).

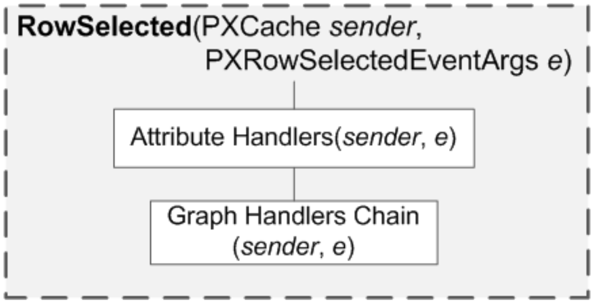


Figure: Execution order for RowDeleted event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowSelected(PXCache sender,
                                           PXRowSelectedEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXRowSelectedEventArgs e`

  The instance of the *PXRowSelectedEventArgs* type that holds data for the `RowSelected` event

Examples of Use

The code below sets UI properties for input controls at run time.

```
public class VendorMaint :
     BusinessAccountGraphBase<VendorR, VendorR,
         Where<BAccount.type, Equal<BAccountType.vendorType>,
             Or<BAccount.type, Equal<BAccountType.combinedType>>>>
{
    ...

    protected virtual void Vendor_RowSelected(PXCache sender,
                                              PXRowSelectedEventArgs e)
    {
        Vendor row = (Vendor)e.Row;
        if (row == null) return;

        bool isNotInserted = !(sender.GetStatus(row) ==
                               PXEntryStatus.Inserted);
        PXUIFieldAttribute.SetVisible<VendorBalanceSummary.depositsBalance>(
            VendorBalance.Cache, null, isNotInserted);
        PXUIFieldAttribute.SetVisible<VendorBalanceSummary.balance>(
            VendorBalance.Cache, null, isNotInserted);
        PXUIFieldAttribute.SetEnabled<Vendor.taxReportFinPeriod>(
            sender, null,
            row.TaxPeriodType != PX.Objects.TX.VendorTaxPeriodType.FiscalPeriod);
        PXUIFieldAttribute.SetEnabled<Vendor.taxReportPrecision>(
            sender, null, row.TaxUseVendorCurPrecision != true);
    }

    ...
```

```
}
```

The code below sets UI properties for actions.

```
public class APAccess : PX.SM.BaseAccess
{
    ...

    protected virtual void RelationGroup_RowSelected(PXCache sender,
                                                     PXRowSelectedEventArgs e)
    {
        PX.SM.RelationGroup group = e.Row as PX.SM.RelationGroup;
        if (group != null)
        {
            if (String.IsNullOrEmpty(group.GroupName))
            {
                Save.SetEnabled(false);
                Vendor.Cache.AllowInsert = false;
            }
            else
            {
                Save.SetEnabled(true);
                Vendor.Cache.AllowInsert = true;
            }
        }
    }

    ...
}
```

The code below sets up the processing operation on a processing screen.

```
[TableAndChartDashboardType]
public class APIntegrityCheck : PXGraph<APIntegrityCheck>
{
    ...

    protected virtual void APIntegrityCheckFilter_RowSelected(
        PXCache sender,
        PXRowSelectedEventArgs e)
    {
        APIntegrityCheckFilter filter = Filter.Current;

        APVendorList.SetProcessDelegate<APReleaseProcess>(
            delegate(APReleaseProcess re, Vendor vend)
            {
                re.Clear(PXClearOption.PreserveTimeStamp);
                re.IntegrityCheckProc(vend, filter.FinPeriodID);
            }
        );
    }

    ...
}
```

Related Types

- *PXRowSelectedEventArgs Class*

PXRowSelectedEventArgs Class

Provides data for the *RowSelected* event.

Inherits

`EventArgs`

Syntax

```
public sealed class PXRowSelectedEventArgs : EventArgs
```

Properties

- `public object Row`

  Gets the DAC object that is being processed

RowInserting Event

The `RowInserting` event is trigged before the new data record is inserted into the `PXCache` object as a result of:

- Inserting initiated in the user interface (UI) or through the Web Service application programming interface (API).
- Invocation of the following methods of the `PXCache` class:
  - `Insert()`
  - `Insert(object)`
  - `Insert(IDictionary)`

The `RowInserting` event handler is used to:

- Evaluate the data record that is being inserted.
- Cancel the insert operation by throwing an exception (see *Examples of Use*).
- Assign the default values to the fields of the data record that is being inserted.

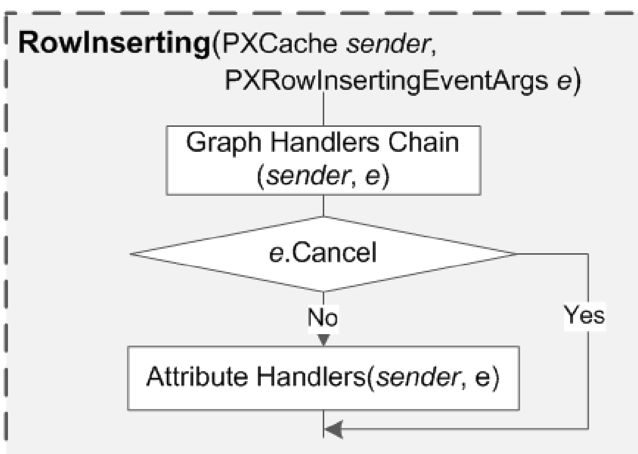


Figure: Execution order for RowInserting event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowInserting(PXCache sender,
                                            PXRowInsertingEventArgs e)
{
```

```
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event

- *(required)* `PXRowInsertingEventArgs e`

  The instance of the *PXRowInsertingEventArgs* type that holds data for the `RowInserting` event

Examples of Use

The code below evaluates the data record that is being inserted and cancels the insert operation.

```
public class CashAccountMaint : PXGraph<CashAccountMaint>
{
    ...

    protected virtual void PaymentMethodAccount_RowInserting(
        PXCache sender,
        PXRowInsertingEventArgs e)
    {
        PaymentMethodAccount row = (PaymentMethodAccount)e.Row;
        if (row.PaymentMethodID != null)
            foreach (PaymentMethodAccount it in Details.Select())
                if (!object.ReferenceEquals(row, it) &&
                    it.PaymentMethodID == row.PaymentMethodID)
                    throw new PXException(
                        Messages.DuplicatedPaymentMethodForCashAccount,
                        row.PaymentMethodID);
        if (row.APIsDefault == true &&
            String.IsNullOrEmpty(row.PaymentMethodID))
            throw new PXException(ErrorMessages.FieldIsEmpty,
                                  typeof(PaymentMethodAccount.
                                      paymentMethodID).Name);
    }

    ...
}
```

The code below assigns the default field values to the data record that is being inserted.

```
public class MyCaseDetailsMaint : PXGraph<MyCaseDetailsMaint>
{
    ...

    protected virtual void EPActivity_RowInserting(PXCache sender,
                                                   PXRowInsertingEventArgs e)
    {
        EPActivity row = e.Row as EPActivity;
        if (Case.Current != null)
        {
            row.StartDate = PXTimeZoneInfo.Now;
            row.RefNoteID = Case.Current.NoteID;
            row.ClassID = CRActivityClass.Activity;
            row.IsExternal = true;
        }
    }

    ...
}
```

Related Types

- *PXRowInsertingEventArgs Class*
- *PXEntryStatus Enumeration*

PXRowInsertingEventArgs Class

Provides data for the *RowInserting* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXRowInsertingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the DAC object that is being inserted.
- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `RowInserting` event handlers specified within DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.
- `public bool ExternalCall`

  Gets the value indicating, if it equals `true`, that the DAC object is being inserted from the UI or through the Web Service API.

RowInserted Event

The `RowInserted` event is triggered after a new data record has been successfully inserted into the `PXCache` object as a result of:

- Insertion initiated in the user interface (UI) or through the Web Service application programming interface (API).
- Invocation of any of the following `PXCache` class methods:
  - `Insert()`
  - `Insert(object)`
  - `Insert(IDictionary)`

The `RowInserted` event handler is used to implement the business logic for:

- Inserting the detail data records in a one-to-many relationship.
- Updating the master data record in a many-to-one relationship.
- Inserting or updating the related data record in a one-to-one relationship.

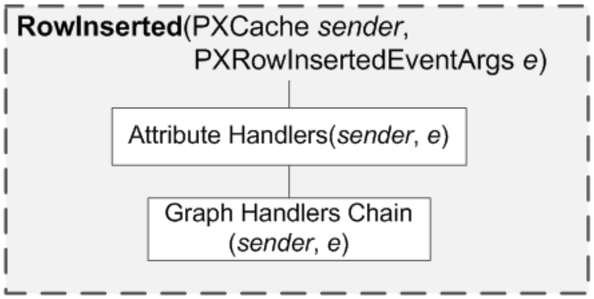


Figure: Execution order for RowInserted event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowInserted(PXCache sender,
                                           PXRowInsertedEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXRowInsertedEventArgs e`

  The instance of the *PXRowInsertedEventArgs* type that holds data for the `RowInserted` event

Examples of Use

The code below inserts the detail data records in a one-to-many relationship.

```
public class VendorClassMaint : PXGraph<VendorClassMaint>
{
    ...

    public virtual void VendorClass_RowInserted(PXCache sender,
                                                PXRowInsertedEventArgs e)
    {
        VendorClass row = (VendorClass)e.Row;
        if (row == null || row.VendorClassID == null) return;

        foreach (APNotification n in PXSelect<
            APNotification,
            Where<APNotification.sourceCD,
                  Equal<APNotificationSource.vendor>>>.
            Select(this))
        {
            NotificationSource source = new NotificationSource();
            source.SetupID = n.SetupID;
            NotificationSources.Insert(source);
        }
    }

    ...
}
```

The code below updates the master data record in a many-to-one relationship.

```
public class InventoryItemMaint : PXGraph<InventoryItemMaint>
{
    ...

    protected virtual void POVendorInventory_RowInserted(
        PXCache sender,
        PXRowInsertedEventArgs e)
    {
        POVendorInventory current = e.Row as POVendorInventory;
        if (current.IsDefault == true && current.VendorID != null &&
            current.VendorLocationID != null && current.SubItemID != null &&
            this.Item.Current.PreferredVendorLocationID !=
            current.VendorLocationID)
        {
            InventoryItem upd = Item.Current;
            upd.PreferredVendorID = current.IsDefault == true ?
                                                             current.VendorID :
                                                             null;
            upd = this.Item.Update(upd);
            upd.PreferredVendorLocationID = current.IsDefault ==
                true ? current.VendorLocationID : null;
            Item.Update(upd);
        }
    }

    ...
}
```

Related Types

- *PXRowInsertedEventArgs Class*
- *PXEntryStatus Enumeration*

PXRowInsertedEventArgs Class

Provides data for the *RowInserted* event.

Inherits

`EventArgs`

Syntax

```
public sealed class PXRowInsertedEventArgs : EventArgs
```

Properties

- `public object Row`

  Gets the DAC object that has been inserted
- `public bool ExternalCall`

  Gets the value indicating, if it equals `true`, that the DAC object has been inserted in the UI or through the Web Service API

RowUpdating Event

The `RowUpdating` event is triggered before the data record is actually updated in the `PXCache` object during an update initiated:

- In the user interface (UI) or through the Web Service application programming interface (API).

- By invocation of the following methods of the `PXCache` class:
  - `Update(object)`
  - `Update(IDictionary, IDictionary)`

Updating of a data record is executed only when there is a data record with the same values of the DAC key fields in either the `PXCache` object or the database. Otherwise, the process of inserting the data record is started.

The `RowUpdating` event handler is used to evaluate the data record that is being updated and cancel the update operation if the data record does not fit the business logic requirements.

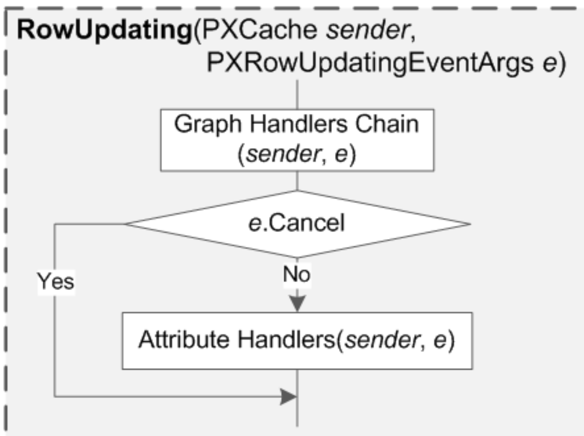


Figure: Execution order for RowUpdating event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowUpdating(PXCache sender,
                                           PXRowUpdatingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXRowUpdatingEventArgs e`

  The instance of the *PXRowUpdatingEventArgs* type that holds data for the `RowUpdating` event

Examples of Use

The code below evaluates the data record that is being updated, cancels the update operation, and shows a message box.

```
public class APPaymentEntry : APDataEntryGraph<APPaymentEntry, APPayment>
{
    ...

    protected virtual void APAdjust_RowUpdating(PXCache sender,
                                                PXRowUpdatingEventArgs e)
    {
```

```
        APAdjust adj = (APAdjust)e.Row;
        if (_IsVoidCheckInProgress == false && adj.Voided == true)
        {
            throw new PXException(ErrorMessages.CantUpdateRecord);
        }
    }

    ...
}
```

The code below evaluates the data record that is being updated, cancels the update operation, and shows the warning or error indication near the input control for one field or multiple fields.

```
protected virtual void INLotSerClass_RowUpdating(PXCache sender,
                                                 PXRowUpdatingEventArgs e)
{
    INLotSerClass row = (INLotSerClass) e.NewRow;
    if (row.LotSerTrackExpiration != true &&
        row.LotSerIssueMethod == INLotSerIssueMethod.Expiration)
    {
        sender.RaiseExceptionHandling<INLotSerClass.lotSerIssueMethod>(
            row, null,
            new PXSetPropertyException(
                Messages.LotSerTrackExpirationInvalid,
                typeof(INLotSerClass.lotSerIssueMethod).Name));
        e.Cancel = true;
    }
}
```

Related Types

- *PXRowUpdatingEventArgs Class*
- *PXEntryStatus Enumeration*

PXRowUpdatingEventArgs Class

Provides data for the *RowUpdating* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXRowUpdatingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the original DAC object that is being updated.
- `public object NewRow`

  Gets the updated copy of the DAC object that is going to be merged with the original one.
- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `RowUpdating` event handlers specified within the DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

Fields

- `public bool ExternalCall`

  Gets the value indicating, if it equals `true`, that the update of the DAC object has been initiated from the UI or through the Web Service API

RowUpdated Event

The `RowUpdated` event is triggered after the data record has been successfully updated in the `PXCache` object as a resulf of:

- An update initiated in the user interface (UI) or through the Web Service application programming interface (API).
- Invocation of the following methods of the `PXCache` class:
  - `Update(object)`
  - `Update(IDictionary, IDictionary)`

Updating of a data record is executed only when there is a data record with the same values of the data access class (DAC) key fields, either in the `PXCache` object or in the database. Otherwise, the process of inserting the data record is started.

The `RowUpdated` event handler is used to implement the business logic of:

- Updating the master data record in a many-to-one relationship.
- Inserting or updating the detail data records in a one-to-many relationship.
- Updating the related data record in a one-to-one relationship.

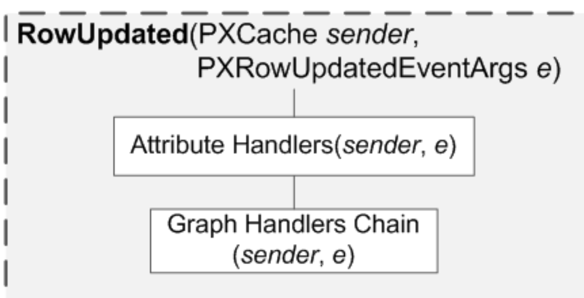


Figure: Execution order for RowUpdated event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowUpdated(PXCache sender,
                                          PXRowUpdatedEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXRowUpdatedEventArgs e`

  The instance of the *PXRowUpdatedEventArgs* type that holds data for the `RowUpdated` event

Examples of Use

The code below updates the detail data records in a one-to-many relationship.

```
public class DraftScheduleMaint : PXGraph<DraftScheduleMaint, DRSchedule>
{
    ...

    protected virtual void DRSchedule_RowUpdated(PXCache sender,
                                                  PXRowUpdatedEventArgs e)
    {
        DRSchedule row = e.Row as DRSchedule;
        if (!sender.ObjectsEqual<DRSchedule.documentType, DRSchedule.refNbr,
                                 DRSchedule.lineNbr, DRSchedule.bAccountID,
                                 DRSchedule.finPeriodID,
                                 DRSchedule.docDate>(e.Row, e.OldRow))
        {
            foreach (DRScheduleDetail detail in Components.Select())
            {
                detail.Module = row.Module;
                detail.DocumentType = row.DocumentType;
                detail.DocType = row.DocType;
                detail.RefNbr = row.RefNbr;
                detail.LineNbr = row.LineNbr;
                detail.BAccountID = row.BAccountID;
                detail.FinPeriodID = row.FinPeriodID;
                detail.DocDate = row.DocDate;
                Components.Update(detail);
            }
        }
    }

    ...
}
```

The code below updates the master data record in a many-to-one relationship.

```
public class ARInvoiceEntry : ARDataEntryGraph<ARInvoiceEntry, ARInvoice>,
                              PXImportAttribute.IPXPrepareItems
{
    ...

    protected virtual void ARTran_RowUpdated(PXCache sender,
                                             PXRowUpdatedEventArgs e)
    {
        ARTran row = (ARTran)e.Row;
        ARTran oldRow = (ARTran)e.OldRow;
        if (Document.Current != null &&
            IsExternalTax == true &&
            !sender.ObjectsEqual<ARTran.accountID, ARTran.inventoryID,
                                 ARTran.tranDesc,
                                 ARTran.tranAmt, ARTran.tranDate,
                                 ARTran.taxCategoryID>(e.Row, e.OldRow))
        {
            ARInvoice copy = Document.Current;
            copy.IsTaxValid = false;
            Document.Update(copy);
        }
    }

    ...
}
```

Related Types

- *PXRowUpdatedEventArgs Class*

- *PXEntryStatus Enumeration*

PXRowUpdatedEventArgs Class

Provides data for the *RowUpdated* event.

Inherits

`EventArgs`

Syntax

```
public sealed class PXRowUpdatedEventArgs : EventArgs
```

Properties

- `public object Row`

  Gets the DAC object that has been updated

- `public object OldRow`

  Gets the copy of the original DAC object before the Update operation

Fields

- `public bool ExternalCall`

  Gets the value indicating, if it equals `true`, that the DAC object has been updated from the UI or through the Web Service API

RowDeleting Event

The `RowDeleting` event is triggered for a data record that is being deleted from the `PXCache` object after its status has been set to `Deleted` or `InsertedDeleted`, but the data record can still be reverted to the previous state by canceling the delete operation (see *Examples of Use*). The status of the data record is set to `Deleted` or `InsertedDeleted` as a result of:

- Deletion initiated in the user interface (UI) or through the Web Service application programming interface (API).
- Invocation of the following methods of the `PXCache` class:
  - `Delete(object)`
  - `Delete(IDictionary, IDictionary)`

When a data record is deleted that has already been stored in the database (and, hence, exists in both the database and the `PXCache` object), the status of the data record is set to `Deleted`. For a data record that has not yet been stored in the database but was only inserted in the `PXCache` object, the status of the data record is set to `InsertedDeleted`.

The `RowDeleting` event handler is used to evaluate the data record that is marked as `Deleted` or `InsertedDeleted` and cancel the delete operation if it is required by the business logic.

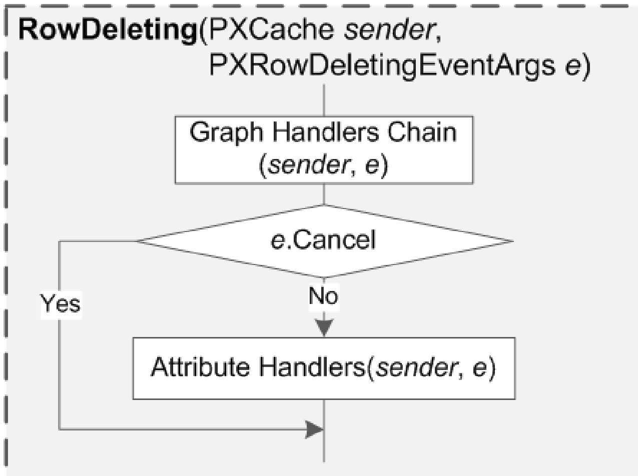


Figure: Execution order for RowDeleting event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowDeleting(PXCache sender,
                                           PXRowDeletingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event

- *(required)* `PXRowDeletingEventArgs e`

  The instance of the *PXRowDeletingEventArgs* type that holds data for the `RowDeleting` event

Examples of Use

The code below evaluates the data record that is being deleted and cancels the delete operation by throwing an exception.

```
public class VendorMaint : BusinessAccountGraphBase<
    VendorR, VendorR,
     Where<BAccount.type,
           Equal<BAccountType.vendorType>,
           Or<BAccount.type,
              Equal<BAccountType.combinedType>>>>
{
    ...

    protected virtual void Vendor_RowDeleting(PXCache sender,
                                              PXRowDeletingEventArgs e)
    {
        Vendor row = e.Row as Vendor;

        TX.Tax tax = PXSelect<
            TX.Tax,
            Where<TX.Tax.taxVendorID,
                  Equal<Current<Vendor.bAccountID>>>>.
```

```
            Select(this);
        if (tax != null)
            throw new PXException(Messages.TaxVendorDeleteErr);
    }

    ...
}
```

Related Types

- *PXRowDeletingEventArgs Class*
- *PXEntryStatus Enumeration*

PXRowDeletingEventArgs Class

Provides data for the *RowDeleting* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXRowDeletingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the DAC object that has been marked as `Deleted`.

- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `RowDeleting` event handlers specified within DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

- `public bool ExternalCall`

  Gets the value indicating, if it equals `true`, that the DAC object has been marked as `Deleted` in the UI or through the Web Service API.

RowDeleted Event

The `RowDeleted` event is triggered for a data record that is being deleted from the `PXCache` object—that is, a data record whose status has been successfully set to `Deleted` or `InsertedDeleted` as result of:

- Deletion initiated in the user interface (UI) or through the Web Service application programming interface (API).
- Invocation of the following methods of the `PXCache` class:
  - `Delete(object)`
  - `Delete(IDictionary, IDictionary)`

When a data record is deleted that has already been stored in the database (and, hence, exists in both the database and the `PXCache` object), the status of the data record is set to `Deleted`. For a data record that has not yet been stored in the database but was only inserted in the `PXCache` object, the status of the data record is set to `InsertedDeleted`.

The `RowDeleted` event handler is used to implement the business logic of:

- Deleting the detail data records in a one-to-many relationship.

- Updating the master data record in a many-to-one relationship.
- Deleting or updating the related data record in a one-to-one relationship.

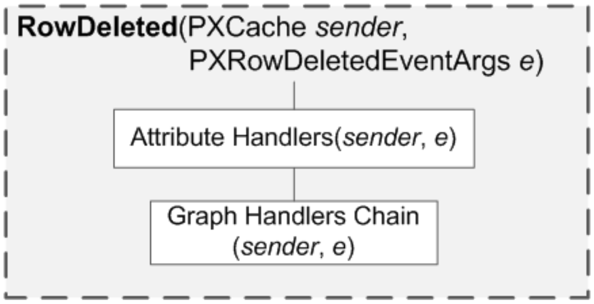


Figure: Execution order for RowDeleted event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowDeleted(PXCache sender,
                                          PXRowDeletedEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXRowDeletedEventArgs e`

  The instance of the *PXRowDeletedEventArgs* type that holds data for the `RowDeleted` event

Examples of Use

The code below deletes detail data records in a one-to-many relationship.

```
public class CashTransferEntry : PXGraph<CashTransferEntry, CATransfer>
{
    ...

    public virtual void CATransfer_RowDeleted(PXCache sender,
                                              PXRowDeletedEventArgs e)
    {
        foreach (CATran item in TransferTran.Select())
            TransferTran.Delete(item);
    }

    ...
}
```

The code below updates the master data record in a many-to-one relationship.

```
public class INSiteMaint : PXGraph<INSiteMaint, INSite>
{
    ...

    protected virtual void INLocation_RowDeleted(PXCache sender,
```

```
                                                        PXRowDeletedEventArgs e)
    {
        INLocation l = (INLocation)e.Row;
        if (site.Current == null || l == null ||
            site.Cache.GetStatus(site.Current) == PXEntryStatus.Deleted)
            return;

        INSite s = site.Current;
        if (s.DropShipLocationID == l.LocationID)
            s.DropShipLocationID = null;
        if (s.ReceiptLocationID == l.LocationID)
            s.ReceiptLocationID = null;
        if (s.ShipLocationID == l.LocationID)
            s.ShipLocationID = null;
        if (s.ReturnLocationID == l.LocationID)
            s.ReturnLocationID = null;
        site.Update(s);
    }

    ...
}
```

Related Types

- *PXRowDeletedEventArgs Class*
- *PXEntryStatus Enumeration*

PXRowDeletedEventArgs Class

Provides data for the *RowDeleted* event.

Inherits

`EventArgs`

Syntax

```
public sealed class PXRowDeletedEventArgs : EventArgs
```

Properties

- `public object Row`

  Gets the DAC object that has been marked as **`Deleted`**
- `public bool ExternalCall`

  Gets the value indicating, if it equals `true`, that the DAC object has been marked as **`Deleted`** in the UI or through the Web Services API

CommandPreparing Event

The `CommandPreparing` event is triggered each time the Acumatica Data Access Layer prepares a database-specific SQL statement for SELECT, INSERT, UPDATE, or DELETE operation. This event is raised for every data access class (DAC) field placed in the `PXCache` object. By using the `CommandPreparing` event subscriber, the application developer can alter the property values of the `PXCommandPreparingEventArgs.FieldDescription` object that is used in the generation of an SQL statement.

The `CommandPreparing` event handler is used to:

- Exclude a DAC field from a SELECT, INSERT, or UPDATE operation
- Replace a DAC field from a SELECT operation with a custom SQL statement

- Transform a DAC field value submitted to the server for INSERT, UPDATE, or DELETE operation

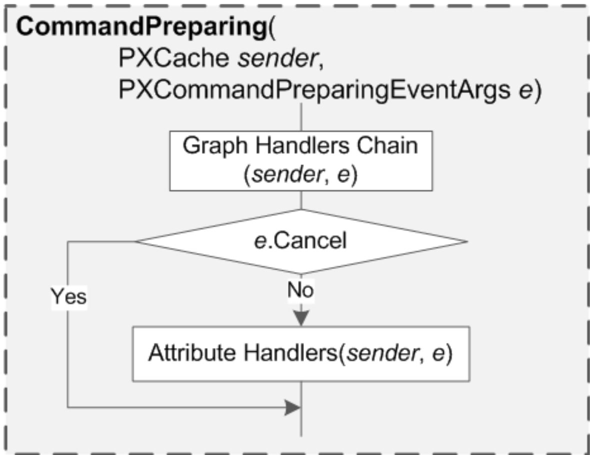


Figure: Execution order for CommandPreparing event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_FieldName_CommandPreparing(
    PXCache sender,
    PXCommandPreparingEventArgs e)
{
  ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event

- *(required)* `PXCommandPreparingEventArgs e`

  The instance of the *PXCommandPreparingEventArgs* type that hold data for the `CommandPreparing` event

Examples of Use

The code below excludes a DAC field from the UPDATE operation.

```
public class APReleaseProcess : PXGraph<APReleaseProcess>
{
    ...

    protected virtual void APRegister_FinPeriodID_CommandPreparing(
        PXCache sender,
        PXCommandPreparingEventArgs e)
    {
        if ((e.Operation & PXDBOperation.Command) == PXDBOperation.Update)
        {
            e.FieldName = string.Empty;
            e.Cancel = true;
        }
    }
}
```

The code below replaces a DAC field with a custom T-SQL statement.

```
[PXAttributeFamily(typeof(PXDBFieldAttribute))]
public class BillContactFullNameAttribute : PXDBFieldAttribute
{
    public override void CommandPreparing(PXCache sender,
                                          PXCommandPreparingEventArgs e)
    {
        if ((e.Operation & PXDBOperation.Command) == PXDBOperation.Select)
        {
            BqlCommand search = new Search<SOContact.fullName,
                                    Where<SOContact.contactID,
                                          Equal<SOOrder.billContactID>>>();
            StringBuilder text = new StringBuilder();
            BqlCommand.Selection selection = new BqlCommand.Selection();
            search.Parse(sender.Graph, new List<IBqlParameter>(),
                         new List<Type>(),
                         null, null, text, selection);

            e.BqlTable = _BqlTable;
            Type field = ((IBqlSearch)search).GetField();
            Type table = BqlCommand.GetItemType(field);
            e.FieldName = BqlCommand.SubSelect +
                          selection.Get(table.Name + "." +
                          field.Name) + text.ToString() + ")";
        }
    }
}

public partial class SOOrder : PX.Data.IBqlTable, PX.Data.EP.IAssign,
                               IFreightBase, ICCAuthorizePayment,
                               ICCCapturePayment, IInvoice
{
    ...

    #region BillContactFullName
    public abstract class billContactFullName : PX.Data.IBqlField
    {
    }
    [PXString(255, IsUnicode = true)]
    [BillContactFullNameAttribute]
    [PXUIField(DisplayName = "Business Name", IsReadOnly = true)]
    public virtual String BillContactFullName { get; set; }
    #endregion
}
```

The code below transforms the DAC field value during INSERT and UPDATE operations.

```
public class PXDBCryptStringAttribute : PXDBStringAttribute,
                                        IPXFieldVerifyingSubscriber,
                                        IPXRowUpdatingSubscriber,
                                        IPXRowSelectingSubscriber
{
    ...

    public override void CommandPreparing(PXCache sender,
                                          PXCommandPreparingEventArgs e)
    {
        if ((e.Operation & PXDBOperation.Command) == PXDBOperation.Insert ||
            (e.Operation & PXDBOperation.Command) == PXDBOperation.Update)
        {
            string value = (string)sender.GetValue(e.Row, _FieldOrdinal);

            e.Value = !string.IsNullOrEmpty(value) ?
                      Convert.ToBase64String(
                          Encrypt(Encoding.Unicode.GetBytes(value))) :
                      null;
```

```
        }
        base.CommandPreparing(sender, e);
    }

    ...
}
```

Related Types

- *PXCommandPreparingEventArgs Class*
- *PXDbType Enumeration*
- *PXDBOperation Enumeration*

PXCommandPreparingEventArgs Class

Provides data for the *CommandPreparing* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXCommandPreparingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the current DAC object.
- `public object Value`

  Gets or sets the current DAC field value.
- `public PXDBOperation Operation`

  Gets the `PXDBOperation` value of the current operation.
- `public Type Table`

  Gets the type of DAC objects placed in the cache.
- `public Type BqlTable`

  Gets or sets the type of the DAC being used during the current operation.
- `public string FieldName`

  Gets or sets the name of the DAC field being used during the current operation.
- `public PXDbType DataType`

  Gets or sets the `PXDbType` of the DAC field being used during the current operation.
- `public int? DataLength`

  Gets or sets the number of characters in the DAC field being used during the current operation.
- `public object DataValue`

  Gets or sets the DAC field value being used during the current operation.
- `public bool IsRestriction`

  Gets or sets the value indicating that the DAC field being used during the UPDATE or DELETE operation is placed in the WHERE clause.

- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `CommandPreparing` event handlers specified within the DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

FieldDescription Class

The nested class that provides information about the field required for the T-SQL statement generation.

Syntax:

```
public sealed class FieldDescription
```

Properties:

- `public readonly Type BqlTable`

  Gets the type of DAC objects placed in the cache
- `public readonly string FieldName`

  Gets the name of the DAC field
- `public readonly PXDbType DataType`

  Gets the `PXDbType` of the DAC field
- `public readonly int? DataLength`

  Gets the storage size of the DAC field
- `public readonly object DataValue`

  Gets the value stored in the DAC field
- `public readonly bool IsRestriction`

  Gets the value indicating that the DAC field being used during the UPDATE or DELETE operation is placed in the WHERE clause

RowSelecting Event

The `RowSelecting` event is triggered for each retrieved data record when the result of a BQL statement is processed. For a BQL statement that contains a JOIN clause, the `RowSelecting` event is raised for every joined data access class (DAC).

The `RowSelecting` event handler is used to:

- Calculate DAC field values that are not bound to specific database columns.
- Convert the database table value of a DAC field to its presentation form.

The application developer can execute additional BQL statements within a `RowSelecting` event handler. However, the connection scope used to retrieve data, which triggered the `RowSelecting` event, is still busy at the moment, so no other operations on this connection scope are allowed. Therefore, to execute additional BQL statements in a `RowSelecting` handler, it is necessary to use a separate connection scope (see *Examples of Use*).

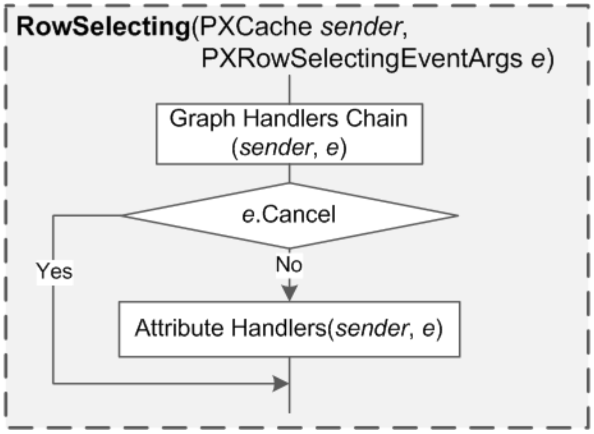


Figure: Execution order for RowSelecting event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowSelecting(PXCache sender,
                                            PXRowSelectingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event

- *(required)* `PXRowSelectingEventArgs e`

  The instance of the *PXRowSelectingEventArgs* type that holds data for the `RowSelecting` event

Examples of Use

The code below calculates a DAC field value that is not bound to a specific column in a database table.

```
public class LocationMaint :
    LocationMaintBase<Location, Location,
                      Where<Location.bAccountID,
                            Equal<Optional<Location.bAccountID>>>>
{

    ...

    protected virtual void Location_RowSelecting(PXCache sender,
                                                 PXRowSelectingEventArgs e)
    {
        Location record = (Location)e.Row;
        if (record != null)
            record.IsARAccountSameAsMain =
                !object.Equals(record.LocationID, record.CARAccountLocationID);
    }

    ...
}
```

The code below executes an additional BQL statement to calculate a DAC field value that is not bound to a specific column in a database table.

```
public class SOInvoiceEntry : ARInvoiceEntry
{
    ...

    protected virtual void ARInvoice_RowSelecting(PXCache sender,
                                                  PXRowSelectingEventArgs e)
    {
        ARInvoice row = (ARInvoice)e.Row;
        if (row != null && !String.IsNullOrEmpty(row.DocType)
                        && !String.IsNullOrEmpty(row.RefNbr))
        {
            row.IsCCPayment = false;
            using (new PXConnectionScope())
            {
                if (PXSelectJoin<
                    CustomerPaymentMethodC,
                    InnerJoin<
                        CA.PaymentMethod,
                        On<CA.PaymentMethod.paymentMethodID,
                           Equal<CustomerPaymentMethodC.paymentMethodID>>,
                        InnerJoin<
                            SOInvoice,
                            On<SOInvoice.pMInstanceID,
                               Equal<CustomerPaymentMethodC.pMInstanceID>>>>,
                    Where<SOInvoice.docType,
                          Equal<Required<SOInvoice.docType>>,
                          And<SOInvoice.refNbr,
                              Equal<Required<SOInvoice.refNbr>>,
                              And<CA.PaymentMethod.paymentType,
                                  Equal<CA.PaymentMethodType.creditCard>,
                              And<CA.PaymentMethod.aRIsProcessingRequired,
                                  Equal<True>>>>>>.
                    Select(this, row.DocType, row.RefNbr).Count > 0)
                {
                    row.IsCCPayment = true;
                }
            }
        }
    }

    ...
}
```

The code below converts the database table value of a DAC field to the internal presentation.

```
public class PXDBCryptStringAttribute : PXDBStringAttribute,
                                        IPXFieldVerifyingSubscriber,
                                        IPXRowUpdatingSubscriber,
                                        IPXRowSelectingSubscriber
{
    ...

    public override void RowSelecting(PXCache sender,
                                      PXRowSelectingEventArgs e)
    {
        base.RowSelecting(sender, e);
        if (e.Row == null || sender.GetStatus(e.Row)
                          != PXEntryStatus.Notchanged) return;
        string value = (string)sender.GetValue(e.Row, _FieldOrdinal);
        string result = string.Empty;
        if (!string.IsNullOrEmpty(value))
        {
            try
            {
```

```
                result = Encoding.
                    Unicode.
                    GetString(Decrypt(Convert.FromBase64String(value)));
            }
            catch (Exception)
            {
                try
                {
                    result = Encoding.Unicode.
                        GetString(Convert.FromBase64String(value));
                }
                catch (Exception)
                {
                    result = value;
                }
            }
        }
        sender.SetValue(e.Row, _FieldOrdinal,
                        result.Replace("\0", string.Empty));
    }

    ...
}
```

Related Types

- *PXRowSelectingEventArgs Class*
- *PXDataRecord Class*

PXRowSelectingEventArgs Class

Provides data for the *RowSelecting* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXRowSelectingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the DAC object that is being processed.
- `public PXDataRecord Record`

  Gets the proceeded data record in the result set.
- `public object Position`

  Gets or sets the index of the proceeded column in the result set.
- `public object IsReadOnly`

  Gets the value indicating that the DAC object is read-only.
- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `RowSelecting` event handlers specified within the DAC field attributes should be invoked. The handlers will not be invoked if the property is set to `true`.

PXDataRecord Class

Used for wrapping a single record of a result set obtained by executing a BQL statement. A record includes data fields of all joined tables.

Inherits

`IDisposable`

Syntax

```
public class PXDataRecord : IDisposable
```

Properties

- `public virtual int FieldCount`

  Gets the number of columns in the current data record. If the `PXDataRecord` instance is not positioned in a valid data record, the value is `0`. The default value is `-1`.

Methods

- `public PXDataRecord(IDataReader reader, IDbCommand command, IDataReader reader)`
- `public virtual bool? GetBoolean(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The Boolean value of the column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.
- `public virtual byte? GetByte(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The 8-bit unsigned integer value of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.
- `public virtual long GetBytes(int i, long fieldOffset, byte[] buffer, int bufferoffset, int i)`

  Reads a stream of bytes from the specified column offset into the buffer as an array, starting at the given buffer offset.

Parameters:

- `buffer`

  The buffer into which to read the stream of bytes.

- `bufferoffset`

  The index for the buffer to start reading.

- `fieldOffset`

  The index within the field from which reading should start.

- `i`

  The index of the zero-based column.

- `length`

  The number of bytes to read.

Returns:

The actual number of bytes read.

Exceptions:

- `System.IndexOutOfRangeException`

  The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual byte[] GetTimeStamp(int i)`
- `public virtual byte[] GetBytes(int i)`
- `public virtual char? GetChar(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The character value of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual long GetChars(int i, long fieldoffset, char[] buffer, int bufferoffset, int length)`

  Reads a stream of characters from the specified column and the offset within it into the buffer as an array, starting from the provided offset.

  Parameters:

  - `i`

    The index of the zero-based column.

  - `fieldoffset`

    The index within the row from which to start reading.

  - `buffer`

The buffer into which the stream of bytes should be read.

- `bufferoffset`

  The index in the buffer to start reading from.

- `length`

  The number of bytes to read.

Returns:

The actual number of characters read.

Exceptions:

- `System.IndexOutOfRangeException`

  The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual string GetDataTypeName(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The data type information for the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual DateTime? GetDateTime(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The date and time value of the specified field.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual decimal? GetDecimal(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The fixed-position numeric value of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual double? GetDouble(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The double-precision floating point value of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual Type GetFieldType(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The `System.Type` information corresponding to the type of `System.Object` that would be returned by `System.Data.IDataRecord.GetValue(System.Int32)`.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual float? GetFloat(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The single-precision floating point number of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual Guid? GetGuid(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

The GUID value of the specified column.

Exceptions:

- `System.IndexOutOfRangeException`

  The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual short? GetInt16(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The 16-bit signed integer value of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual int? GetInt32(int i)`

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The 32-bit signed integer value of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual long? GetInt64(int i)`

  Parameters:

  - `i`

    The zero-based column's index.

  Returns:

  the 64-bit signed integer value of the specified field.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual string GetName(int i)`

  Parameters:

  - `i`

    The zero-based column's index.

Returns:

The name of the specified column or the empty string (`""`), if there is no value to return.

Exceptions:

- `System.IndexOutOfRangeException`

  The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual string GetString(int i)`

  Parameters:

  - `i`

    The zero-based column's index.

  Returns:

  The string value of the specified column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual object GetValue(int i)`

  Returns the value of the specified column.

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  The `System.Object` containing the value of the column.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

- `public virtual bool IsDBNull(int i)`

  Specifies whether the value of the specified column is `null`.

  Parameters:

  - `i`

    The index of the zero-based column.

  Returns:

  `true` if the specified column is set to `null` and `false` otherwise.

  Exceptions:

  - `System.IndexOutOfRangeException`

    The index passed was outside the range from `0` to `System.Data.IDataRecord.FieldCount - 1`.

RowPersisting Event

The `RowPersisting` event is triggered in the process of committing changes to the database for every data record whose status is `Inserted`, `Updated`, or `Deleted` before the corresponding changes for the data record are committed to the database.

Committing changes to a database is initiated by invoking the `Actions.PressSave()` method of the business logic controller (BLC). While processing this method, the Acumatica Data Access Layer first commits every `Inserted` data record, then every `Updated` data record, and finally each `Deleted` data record.

Avoid executing additional BQL statements in a `RowPersisting` event handler. When the `RowPersisting` event is raised, the associated transaction scope is busy saving the changes, and any other operation performed within this transaction scope may cause performance degradation and deadlocks.

The `RowPersisting` event handler is used to:

- Validate the data record before it has been committed to the database.
- Cancel the commit operation of the data record by throwing an exception (see *Examples of Use*).

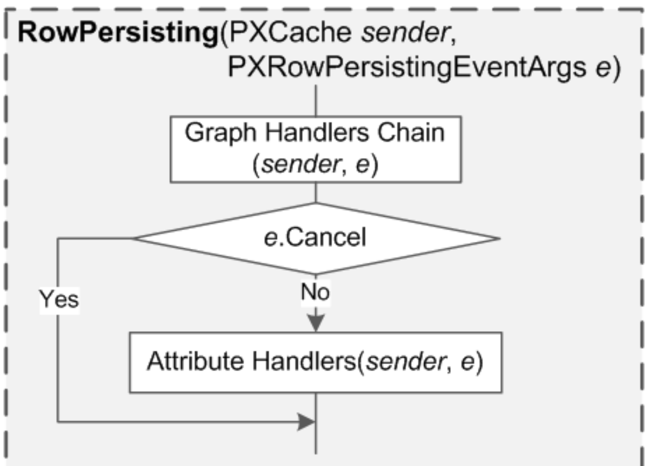


Figure: Execution order for RowPersisting event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowPersisting(PXCache sender,
                                             PXRowPersistingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXRowPersistingEventArgs e`

  The instance of the *PXRowPersistingEventArgs* type that holds data for the `RowPersisting` event

Examples of Use

The code below validates the data record before it is committed to the database.

```
public class CCProcessingCenterMaint : PXGraph<CCProcessingCenterMaint,
                                              CCProcessingCenter>,
                                       IProcessingCenterSettingsStorage
{
    ...

    protected virtual void CCProcessingCenter_RowPersisting(
        PXCache sender,
        PXRowPersistingEventArgs e)
    {
        if ((e.Operation & PXDBOperation.Command) != PXDBOperation.Delete &&
             e.Row != null &&
             (bool)((CCProcessingCenter)e.Row).IsActive &&
             string.IsNullOrEmpty(((CCProcessingCenter)e.Row).
                 ProcessingTypeName))
        {
            throw new PXRowPersistingException(
                typeof(CCProcessingCenter.processingTypeName).Name,
                null,
                ErrorMessages.FieldIsEmpty,
                typeof(CCProcessingCenter.processingTypeName).Name);
        }
    }

    ...
}
```

The code below shows a message box as well as the warning and error indications near the input control for one or multiple fields.

```
protected virtual void APInvoice_RowPersisting(PXCache sender,
                                               PXRowPersistingEventArgs e)
{
    APInvoice doc = (APInvoice)e.Row;
    if (doc.PaySel == true && doc.PayDate == null)
    {
        sender.RaiseExceptionHandling<APInvoice.payDate>(
            doc, null,
            new PXSetPropertyException(ErrorMessages.FieldIsEmpty,
                                       typeof(APInvoice.payDate).Name));
    }
    if (doc.PaySel == true && doc.PayDate != null &&
        ((DateTime)doc.DocDate).CompareTo((DateTime)doc.PayDate) > 0)
    {
        sender.RaiseExceptionHandling<APInvoice.payDate>(
            e.Row, doc.PayDate,
            new PXSetPropertyException(Messages.ApplDate_Less_DocDate,
                                       PXErrorLevel.RowError,
                                       typeof(APInvoice.payDate).Name));
    }
}
```

The code below cancels the operation of committing a data record.

```
public class CampaignMemberMassProcess : PXGraph<CampaignMemberMassProcess>
{
    ...

    protected virtual void Contact_RowPersisting(PXCache sender,
                                                 PXRowPersistingEventArgs e)
    {
        e.Cancel = true;
    }
```

```
    ...
}
```

Related Types

- *PXRowPersistingEventArgs Class*
- *PXEntryStatus Enumeration*
- *PXDBOperation Enumeration*

PXRowPersistingEventArgs Class

Provides data for the *RowPersisting* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXRowPersistingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the DAC object that is being committed to the database.
- `public PXDBOperation Operation`

  Gets the `PXDBOperation` of the current commit operation
- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `RowPersisting` event handlers specified within the DAC field attributes should be invoked. If the property is set to `true`, the handlers will not be invoked and the commit operation of the data record will be canceled. Otherwise, the handlers will be invoked and the commit operation will not be cancelled.

RowPersisted Event

The `RowPersisted` event is triggered in the process of committing changes to the database for every data record whose status is `Inserted`, `Updated`, or `Deleted`. The `RowPersisted` event is triggered twice:

- When the data record has been committed to the database and the status of the transaction scope (indicated in the `e.TranStatus` field) is `Open`
- When the status of the transaction scope has changed to `Completed`, indicating successful committing, or `Aborted`, indicating that a database error has occurred and changes to the database have been dropped

The `Actions.PressSave()` method of the business logic controller (graph) initiates committing changes to a database. While processing this method, the Acumatica Data Access Layer first commits every `Inserted` data record, then each `Updated` data record, and finally each `Deleted` data record.

Avoid executing additional BQL statements in a `RowPersisted` event handler when the status of the transaction scope is `Open`. When the `RowPersisted` event is raised with this status, the associated transaction scope is busy saving the changes, and any other operation performed within this transcation scope may cause performance degradation and deadlocks.

The `RowPersisted` event handler is used to:

- Retrieve data generated by the database.
- Restore data access class (DAC) field values if the status of the transaction scope is `Aborted` (changes have not been saved). Note that in this case the DAC fields do not revert to any previous state automatically but are left by the Acumatica Data Access Layer in exactly the state they were in before the committing was initiated.
- Validate the data record while committing it to the database.

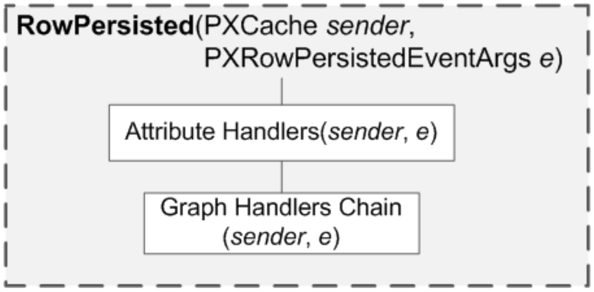


Figure: Execution order for RowPersisted event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_RowPersisted(PXCache sender,
                                            PXRowPersistedEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXRowPersistedEventArgs e`

  The instance of the *PXRowPersistedEventArgs* type that holds data for the `RowPersisted` event

Examples of Use

The code below retrieves data generated by the database.

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Parameter |
                AttributeTargets.Class | AttributeTargets.Method)]
public class PXDBIdentityAttribute : PXDBFieldAttribute,
                                     IPXFieldDefaultingSubscriber,
                                     IPXRowSelectingSubscriber,
                                     IPXCommandPreparingSubscriber,
                                     IPXFieldUpdatingSubscriber,
                                     IPXFieldSelectingSubscriber,
                                     IPXRowPersistedSubscriber,
                                     IPXFieldVerifyingSubscriber
{
    ...

    public virtual void RowPersisted(PXCache sender,
                                     PXRowPersistedEventArgs e)
    {
        if ((e.Operation & PXDBOperation.Command) == PXDBOperation.Insert)
```

```
        {
            if (e.TranStatus == PXTranStatus.Open)
            {
                if (_KeyToAbort == null)
                    _KeyToAbort = (int?)sender.GetValue(e.Row, _FieldOrdinal);
                if (_KeyToAbort < 0)
                {
                    int? id =
                        Convert.ToInt32(PXDatabase.SelectIdentity(_BqlTable));
                    if ((id ?? 0m) == 0m)
                    {
                        PXDataField[] pars =
                            new PXDataField[sender.Keys.Count + 1];
                        pars[0] = new PXDataField(_DatabaseFieldName);
                        for (int i = 0; i < sender.Keys.Count; i++)
                        {
                            string name = sender.Keys[i];
                            PXCommandPreparingEventArgs.
                                FieldDescription description = null;
                            sender.RaiseCommandPreparing(
                                name, e.Row,
                                sender.GetValue(e.Row, name),
                                PXDBOperation.Select,
                                _BqlTable, out description);
                            if (description != null &&
                                !String.IsNullOrEmpty(
                                    description.FieldName) &&
                                description.IsRestriction)
                            {
                                pars[i + 1] = new PXDataFieldValue(
                                    description.FieldName,
                                    description.DataType,
                                    description.DataLength,
                                    description.DataValue);
                            }
                        }
                        using (PXDataRecord record =
                            PXDatabase.SelectSingle(_BqlTable, pars))
                        {
                            if (record != null)
                                id = record.GetInt32(0);
                        }
                    }
                    sender.SetValue(e.Row, _FieldOrdinal, id);
                }
                else
                    _KeyToAbort = null;
            }
            else if (e.TranStatus == PXTranStatus.Aborted &&
                     _KeyToAbort != null)
            {
                sender.SetValue(e.Row, _FieldOrdinal, _KeyToAbort);
                _KeyToAbort = null;
            }
        }
    }

    ...
}
```

The code below restores the values of a DAC field if the commit operation failed—resulting in the `Aborted` status of the transaction scope.

```
public class AddressRevisionIDAttribute : PXEventSubscriberAttribute,
                                          IPXRowPersistingSubscriber,
                                          IPXRowPersistedSubscriber
{
    ...
```

```
    public virtual void RowPersisted(PXCache sender,
                                     PXRowPersistedEventArgs e)
    {
        if (e.TranStatus == PXTranStatus.Aborted &&
            (e.Operation == PXDBOperation.Insert || e.Operation ==
             PXDBOperation.Update))
        {
            int? revision = (int?)sender.GetValue(e.Row, _FieldOrdinal);
            revision--;
            sender.SetValue(e.Row, _FieldOrdinal, revision);
        }
    }

    ...
}
```

The code below validates a data record while it is being committed to the database.

```
protected virtual void Batch_RowPersisted(PXCache sender, PXRowPersistedEventArgs e)
{
    if (e.TranStatus == PXTranStatus.Open &&
        Convert.ToInt32(((Batch)e.Row).BatchNbr) > 10)
        throw new PXRowPersistedException(
            typeof(Batch.batchNbr).Name,
            ((Batch)e.Row).BatchNbr,
           "Number of batches created should not exceed 10 in trial mode.");
}
```

Related Types

- *PXRowPersistedEventArgs Class*
- *PXTranStatus Enumeration*
- *PXEntryStatus Enumeration*
- *PXDBOperation Enumeration*

PXRowPersistedEventArgs Class

Provides data for the *RowPersisted* event.

Inherits

`EventArgs`

Syntax

```
public sealed class PXRowPersistedEventArgs : EventArgs
```

Properties

- `public object Row`

  Gets the DAC object that has been committed to the database
- `public PXDBOperation Operation`

  Gets the `PXDBOperation` value indicating the type of the current commit operation
- `public Exception Exception`

  Gets the `Exception` object thrown while changes are committed to the database
- `public PXTranStatus TranStatus`

Gets the status of the transation scope associated with the current committing operation

PXTranStatus Enumeration

Describes the current status of a transaction scope.

Syntax

```
public enum PXTranStatus
```

Members

- Open

  The status of the transaction is unknown, because some participants still have to be polled.
- Completed

  The changes associated with the transaction scope have been successfully committed to the database.
- Aborted

  The changes within the transaction scope have been dropped because of an error.

ExceptionHandling Event

The ExceptionHandling event is triggered under the following circumstances:

- When the PXSetPropertyException exception is thrown while the system is:
  - Processing a data access class (DAC) field value received from the user interface (UI) or through the Web Service application programming interface (API) when a data record is being inserted or updated in the PXCache object.
  - Processing DAC key field values when deletion of a data record from the PXCache object is initiated in the UI or through the Web Service API.
  - Assigning any field its default value or updating the value when the asignment or update is initiated by any of the following methods of the PXCache class:
    - Insert(IDictionary)
    - SetDefaultExt(object, string)
    - SetDefaultExt<Field>(object)
    - Update(IDictionary, IDictionary)
    - SetValueExt(object, string, object)
    - SetValueExt<Field>(object, object)
  - Converting the external DAC key field presentation to the internal field value initiated by any of the following methods of the PXCache class:
    - Locate(IDictionary)
    - Update(IDictionary, IDictionary)
    - Delete(IDictionary, IDictionary)
- When the PXCommandPreparingException, PXRowPersistingException, or PXRowPersistedException exception is thrown in the process of saving an inserted, updated, or deleted data record in the database.

The ExceptionHandling event handler is used to:

- Catch and handle the exceptions mentioned above (the platform rethrows all unhandled exceptions).
- Implement non-standard handling of the exceptions mentioned above.

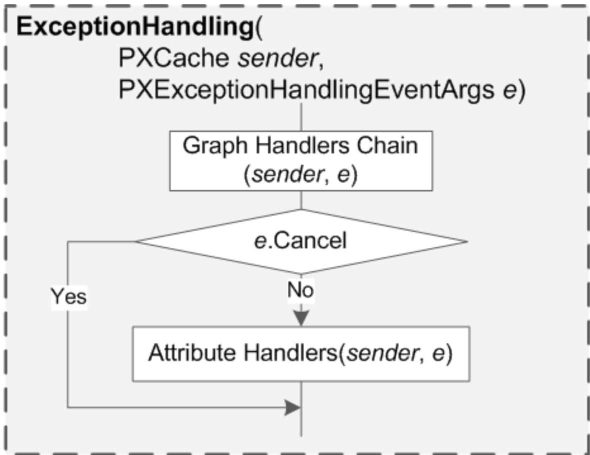


Figure: Execution order for ExceptionHandling event handlers

Syntax

You should define a graph event handler as follows.

```
protected virtual void DACName_FieldName_ExceptionHandling(
    PXCache sender,
    PXExceptionHandlingEventArgs e)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event
- *(required)* `PXExceptionHandlingEventArgs e`

  The instance of the *PXExceptionHandlingEventArgs* type that holds data for the `ExceptionHandling` event

Examples of Use

The code below handles an exception on a DAC field and sets the field value.

```
public class APVendorBalanceEnq : PXGraph<APVendorBalanceEnq>
{
    ...

    protected virtual void APHistoryFilter_AccountID_ExceptionHandling(
        PXCache sender,
        PXExceptionHandlingEventArgs e)
    {
        APHistoryFilter header = e.Row as APHistoryFilter;
        if (header != null)
        {
            e.Cancel = true;
            header.AccountID = null;
```

```
        }
    }

    ...
}
```

The code below alters an exception on a DAC field by setting its description.

```
public class CustomerMaint :
    BusinessAccountGraphBase<Customer, Customer,
                             Where<BAccount.type,
                                   Equal<BAccountType.customerType>,
                                   Or<BAccount.type,
                                      Equal<BAccountType.combinedType>>>>
{
    ...

    protected virtual void Customer_CustomerClassID_ExceptionHandling(
        PXCache sender,
        PXExceptionHandlingEventArgs e)
    {
        PXSetPropertyException ex = e.Exception as PXSetPropertyException;
        if (ex != null)
        {
            ex.SetMessage(ex.Message + System.Environment.NewLine +
                          System.Environment.NewLine +
                          "Stack Trace:" + System.Environment.NewLine +
                          ex.StackTrace);
        }
    }

    ...
}
```

Related Types

- *PXExceptionHandlingEventArgs Class*

PXExceptionHandlingEventArgs Class

Provides data for the *ExceptionHandling* event.

Inherits

`CancelEventArgs`

Syntax

```
public sealed class PXExceptionHandlingEventArgs : CancelEventArgs
```

Properties

- `public object Row`

  Gets the current DAC object.
- `public object NewValue`

  Gets or sets the values of the DAC field. By default, containsvalues that are:

  - Generated in the process of assigning a DAC field its default value.
  - Passed as new values when a field is updated.
  - Entered in the UI or through the Web Service API.

    - Received with the `PXCommandPreparingException`, `PXRowPersistingException`, or `PXRowPersistedException` exception.

- `public Exception Exception`

  Gets the initial exception that caused the event to be raised.

- `public bool Cancel`

  Inherited from the `CancelEventArgs` ancestor class; gets or sets the value indicating whether `ExceptionHandling` event handlers specified within the DAC field attributes should be invoked. If the property is set to `true`, the handlers will not be invoked and the exception will be handled. Otherwise, the exception is rethrown.

CacheAttached Event

The `CacheAttached` handler is used to override data access class (DAC) field attributes declared directly within the DAC. By declaring a `CacheAttached` handler and attaching appropriate attributes to the handler within a graph, the developer forces the framework to completely override DAC field attributes within this graph.

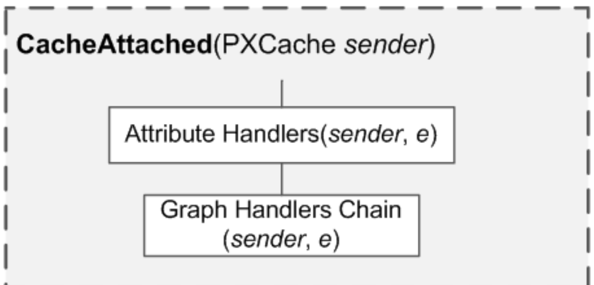


Figure: Execution order for CacheAttached event handlers

Syntax

You should define a graph event handler as follows.

```
[DAC_Field_Attribute_1]
...
[DAC_Field_Attribute_N]
protected virtual void DACName_FieldName_CacheAttached(PXCache sender)
{
    ...
}
```

Parameters

- *(required)* `PXCache sender`

  The cache object that raised the event

Examples of Use

The code below overrides DAC field attributes within a graph.

```
public class DimensionMaint : PXGraph<DimensionMaint, Dimension>
{
    ...

    [PXDBString(15, IsUnicode = true, IsKey = true)]
    [PXDefault(typeof(Dimension.dimensionID))]
```

```
    [PXUIField(DisplayName = "Dimension ID", Visibility =
    PXUIVisibility.Invisible, Visible = false)]
    [PXSelector(typeof(Dimension.dimensionID), DirtyRead = true)]
    protected virtual void Segment_DimensionID_CacheAttached(PXCache sender)
    {

    }

    ...
}
```

Related Types

- *PXUIVisibility Enumeration*

BQL

This document describes BQL (business query language). BQL is a part of the data access layer of the Acumatica Framework. BQL statements represent specific SQL queries and are translated into SQL by the framework. This helps the developer to avoid specifics of the database provider and validate the queries at compile time.

Most BQL components are directly mapped to SQL keywords (such as different types of joins, `OrderBy`, `GroupBy`, etc.). In addition, BQL introduces custom syntax of `Current`, `Required`, and `Optional` parameters. The parameters are substituted with specific values taken from the current objects or specified in code.

The following chapters cover specific topics related to BQL statements construction and execution:

- *Constructing Statements*
- *Filtering*
- *Querying Multiple Tables*
- *Grouping and Aggregating*
- *Using Parameters*
- *Using Functions*
- *Executing Statements*
- *Appendix*

Constructing Statements

To construct a specific BQL statement, you take the generic `PXSelect<>` class or one of its variants and set its type parameters to the data access class (DAC), which represents a database table, and BQL classes that represent SQL clauses and keywords.

Defining a DAC

To select data from a database table, you need to define the DAC. For example, to select data from the `Product` table, you define the `Product` DAC.

```
[System.SerializableAttribute()]
public class Product : PX.Data.IBqlTable
{
    // The type will be used to reference the ProductID field in BQL statements
    public abstract class productID : PX.Data.IBqlField
    {
    }
    // The property will hold the ProductID value
```

```
    [PXDBIdentity]
    public virtual int? ProductID { get; set; }

    // The type will be used to reference the ProductID UnitPrice in BQL statements
    public abstract class unitPrice : PX.Data.IBqlField
    {
    }
    // The property will hold the UnitPrice value
    [PXDBDecimal(2)]
    public virtual int? UnitPrice { get; set; }
}
```

The definition of a field consists of the type and the property. The type is used to reference the field in the BQL statements. The property holds the value of a field.

To indicate that the field is bound to the database and represents a table column, you place the `PXDB`*`Type`* attribute on the definition of the property.

Basic Select Statement

The following BQL statement selects all data records from the `Product` table.

```
PXSelect<Product>
```

To execute such BQL statement, the application must define the DAC representing the `Product` database table. This BQL statement will be translated into the following SQL query.

```
SELECT Product.ProductID, Product.UnitPrice FROM Product
```

The actual SQL query will also include ordering by DAC key fields in ascending order. The framework adds such ordering to the end of the SQL query if the BQL statement does not specify ordering.

The SQL query generated by the framework selects all bound fields of the requested DACs. We will use the `*` sign in further examples to represent selected columns.

Adding the Where Clause

`PXSelect` has several variants allowing additional clauses. The `Where` clause is used to specify conditions.

```
PXSelect<Product,
    Where<Product.productID, Equal<Required<Product.productID>>>>
```

This statement will be translated into the following SQL query, which selects the `Product` data record that satisfy the condition in the `Where` clause.

```
SELECT * FROM Product
    WHERE Product.ProductID = [parameter]
```

Here, `[parameter]` will be replaced with the value passed to the `Select()` method.

To reference a field in a BQL statement, you use the type that is defined in the DAC and represents the field (`Product.productID`). The field name must be preceded with the DAC name and start with a lowercase letter (to distiguish it from the property that holds the value of a field).

The `Where` clause can be used to specify complex filtering conditions chained by logical operators `Or`, `And`, and `Not` and nested `Where` clauses. See examples in *Filtering*.

Adding the OrderBy Clause

The result set of a BQL statement is ordered using the `OrderBy` clause. It can be specified as the second type parameter in the `PXSelectOrderBy` statement, as the third type parameter in a `PXSelect` statement next to the `Where` clause, or in more complex constructions with aggregations and joins.

It is possible to order the result set by one or several columns. For each column, the `Asc` or `Desc` clause must be used to specify whether to sort records in ascending or descending order, respectively.

The following statement selects all `Product` data records and sorts them by the `UnitPrice` field in ascending order.

```
PXSelectOrderBy<Product, OrderBy<Asc<Product.unitPrice>>>
```

This statement is translated into the following SQL query.

```
SELECT * FROM Product
    ORDER BY Product.UnitPrice
```

Using variants of `Asc` and `Desc` with two type parameters, you can request ordering by several columns, as in the following example.

```
PXSelect<Product,
    OrderBy<Asc<Product.unitPrice, Desc<Product.availQty>>>>
```

The corresponding SQL query will look like this.

```
SELECT * FROM Product
    ORDER BY Product.UnitPrice, Product.AvailQty DESC
```

In the following example the `OrderBy` clause is added to a statement with a `Where` clause.

```
PXSelect<DocTransation,
    Where<DocTransation.lastTransactionDate, Less<Today>>,
    OrderBy<Desc<DocTransation.lastTransactionPrice>>>
```

This statement selects all `DocTransation` records of transactions carried out before today and sorts them by the `LastTransactionPrice` field in the descending order (records with greater values of this field go first). The statement is translated into the following SQL query.

```
SELECT * FROM DocTransaction
    WHERE DocTransaction.LastTransactionDate < [today date]
    ORDER BY DocTransation.LastTransactionPrice DESC
```

You can construct any combination of *Where*, `OrderBy`, *Join*, and *GroupBy*.

BQL Statement Execution

To execute a BQL statement, you invoke its `Select()` method (either statically or dynamically). For example, the following code may be found in some graph method.

```
foreach(Product p in PXSelect<Product,
                         Where<Product.UnitPrice, IsNotNull>>.Select(this))
{
    ...
}
```

See *Executing Statements* for details on execution of BQL statements and processing of the result set.

PXSelect, Select, and Search Classes

The `Select` class and *its variants* represent BQL commands. These classes can parse themselves into SQL and provide methods for modifying the BQL command. However, you cannot use the `Select` class to execute the BQL command.

The `PXSelect` class and *its variants* wrap instances of `Select` and give you convenient interfaces to execute the BQL command and interact with the cache. The instances of `PXSelect` classes are complex objects containing:

- Reference to the *PXView* object constructed to process the BQL command
- Reference to the `Select` object—through the `PXView` object
- Reference to the graph
- Reference to the cache of the DAC type that is specified in the first type parameter of `PXSelect`

You use `PXSelect` classes to define data views in a graph and select data from the database in code.

The `Search` class and *its variants* also represent BQL commands but select only one particular field, while the `Select` classes select all fields. In a BQL expression based on `Select` or `PXSelect` the first type parameter is a DAC.

```
Select<Product>
```

In a `Search`-based statement, the first type parameter is a DAC field.

```
Search<Product.unitPrice>
```

The `Select` and `Search` classes are used to specify BQL commands when the interfaces to the `PXView` and cache are not needed. Typically, you use `Select` and `Search` in attributes in DACs. For example, `Select` is used in the *PXProjection* attribute and `Search` is used in the *PXDBScalar* attribute.

The syntax of `PXSelect`, `Select`, and `Search` statements is equivalent (except that `Search` references a field in the first parameter). Further examples show the syntax only for `PXSelect`.

PXSelect Classes

The `PXSelect` class and other classes derived from `PXSelectBase` (referred to below as `PXSelect` classes) are used as a basis for building BQL statements. Such classes are translated into the SQL `SELECT` statements.

```
PXSelect<Table>
```

The BQL statement above is translated into the following SQL query.

```
SELECT * FROM Table
```

The first type parameter of all `PXSelect` classes is a data access class (DAC) bound to a database table. The resulting SQL query will select records from this table. Other type parameters are optional and represent clauses that can be added to the basic select statement:

- *Where*
- *OrderBy*
- *Join*
- *Aggregate*

Depending on the clauses that will be used in a query, an appropriate variant of the `PXSelect` class is picked.

For example, `Where`, `OrderBy`, and `Join` clauses may be combined in the `PXSelectJoin<Table, Join, Where, OrderBy>` class as follows.

```
PXSelectJoinGroupBy<Table1,
    LeftJoin<Table2, On<Table2.field2, Equal<Table1.field1>>>,
    Where<Table1.field3, IsNotNull>,
    Aggregate<GroupBy<Table1.field1,
              Min<Table2.field2>>>,
    OrderBy<Asc<Table1.field1>>>
```

This is translated into the following SQL query.

```
SELECT * FROM Table1
    LEFT JOIN Table2 ON Table2.Field2 = Table1.Field1
    WHERE Table1.Field3 IS NOT NULL
    ORDER BY Table1.Field1
```

PXSelect<Table> : PXSelectBase<Table>

Selects records from one table. The result set is merged with the modified data records kept in the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`

PXSelect<Table, Where> : PXSelectBase<Table>

Selects records from one table filtered by an expression set in `Where`. The result set is merged with the modified data records kept in the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Where : IBqlWhere, new()`

PXSelect<Table, Where, OrderBy> : PXSelectBase<Table>

Selects records from one table, filters them by an expression set in `Where`, and orders by fields specified in `OrderBy`. The result set is merged with the modified data records kept in the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectJoin<Table, Join> : PXSelectBase<Table>

Selects records from multiple tables linked via the `Join` clause. The resulting data records from the main table are merged with the modified data records from the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`

PXSelectJoin<Table, Join, Where> : PXSelectBase<Table>

Selects records from multiple tables linked via the `Join` clause and filters the result set according to expression set in `Where`. The resulting data records from the main table are merged with the modified data records from the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`

PXSelectJoinOrderBy<Table, Join, OrderBy> : PXSelectBase<Table>

Selects records from multiple tables linked via the `Join` clause, filters the result set by the expression set in `Where`, and sorts it by the fields specified in `OrderBy`. The resulting data records from the main table are merged with the modified data records from the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectJoin<Table, Join, Where, OrderBy> : PXSelectBase<Table>

Selects records from multiple tables. The resulting data records from the main table are merged with the modified data records from the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectOrderBy<Table, OrderBy> : PXSelectBase<Table>

Selects records from one table and sorts them by fields specified in `OrderBy`. The result set is merged with the modified data records kept in the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectOrderBy<Table, Join, OrderBy> : PXSelectBase<Table>

Selects records from multiple tables. The resulting data records from the main table are merged with the modified data records from the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectReadonly<Table> : PXSelectBase<Table>

Selects records from one table without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`

PXSelectReadonly<Table, Where> : PXSelectBase<Table>

Selects records from one table without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Where : IBqlWhere, new()`

PXSelectReadonly<Table, Where, OrderBy> : PXSelectBase<Table>

Selects records from one table without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectReadonly2<Table, Join> : PXSelectBase<Table>

Selects records from one table without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`

PXSelectReadonly2<Table, Join, Where> : PXSelectBase<Table>

Selects records from multiple tables without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`

PXSelectReadonly2<Table, Join, Where, OrderBy> : PXSelectBase<Table>

Selects records from multiple tables without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectReadonly3<Table, OrderBy> : PXSelectBase<Table>

Selects records from one table without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectReadonly3<Table, Join, OrderBy> : PXSelectBase<Table>

Selects records from multiple tables without merging the result set with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectGroupBy<Table, Aggregate> : PXSelectBase<Table>

Selects records from the one table, grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Aggregate : IBqlAggregate, new()`

PXSelectGroupBy<Table, Where, Aggregate> : PXSelectBase<Table>

Selects records from one table, grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`

PXSelectGroupBy<Table, Where, Aggregate, OrderBy> : PXSelectBase<Table>

Selects records from one table grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectJoinGroupBy<Table, Join, Aggregate> : PXSelectBase<Table>

Selects records from multiple tables, grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Aggregate : IBqlAggregate, new()`

PXSelectJoinGroupBy<Table, Join, Where, Aggregate> : PXSelectBase<Table>

Selects records from multiple tables, grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`

PXSelectJoinGroupBy<Table, Join, Where, Aggregate, OrderBy> : PXSelectBase<Table>

Selects records from multiple tables, grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectGroupByOrderBy<Table, Aggregate, OrderBy> : PXSelectBase<Table>

Selects records from one table, grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

PXSelectGroupByOrderBy<Table, Join, Aggregate, OrderBy> : PXSelectBase<Table>

Selects records from multiple tables, grouping and applying aggregations. The result set is not merged with the `PXCache<Table>` object.

Type Parameters:

- `Table : class, IBqlTable, new()`
- `Join : IBqlJoin, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

OrderBy Clause

The `OrderBy` clause sorts the result set of a BQL statement. Sorting may be performed by one or several fields in ascending (`Asc`) or descending (`Desc`) order. The type parameter of `OrderBy` clause is set to the `Asc` or `Desc` operator specifying the field to sort by. For example:

```
PXSelect<Table, OrderBy<Asc<Table.field1>>>
```

This is translated into:

```
SELECT * FROM Table
ORDER BY Table.field1
```

An example of sorting by two fields:

```
PXSelect<Table,
    OrderBy<Asc<Table.field1,
            Desc<Table.field2>>>>
```

Note that to attach the second ordering field, a variant of `Asc` with two type parameters is used. To add sorting by even more fields, you would insert another `Asc` or `Desc` operator in the last such operator. The BQL statement above is translated into:

```
SELECT * FROM Table
ORDER BY Table.field1, Table.field2 DESC
```

The result set is sorted by the first field. Then the records that have the same value in the first field are sorted by the second field, and so on.

If a BQL statement does not include `OrderBy`,Acumatica Framework automatically appends ordering by DAC key fields to the SQL query.

OrderBy<List> : IBqlOrderBy

The clause for specifying how to order the result set of a BQL statetement, equivalent to the SQL clause `ORDER BY`.

Type Parameters:

- `List : IBqlSortColumn`

Asc<Field> : IBqlSortColumn

Indication of sorting in ascending order: from the least value to the largest value. The field to order by is specified in the `Field` type parameter. The clause itself is used as a type parameter in `OrderBy`.

Type Parameters:

- `Field : IBqlOperand`

Desc<Field> : IBqlSortColumn

Indication of sorting in descending order: from the largest value down to the least value. The field to order by is specified in the `Field` type parameter. The clause itself is used as a type parameter in `OrderBy`.

Type Parameters:

- `Field : IBqlOperand`

Asc<Field, NextField> : IBqlSortColumn

A variant of the `Asc` clause used to add additional sort expression.

Type Parameters:

- Field : IBqlOperand
- NextField : IBqlSortColumn

Desc<Field, NextField> : IBqlSortColumn

A variant of the Desc clause used to add additional sort expression.

Type Parameters:

- Field : IBqlOperand
- NextField : IBqlSortColumn

Filtering

Filtering conditions are constructed using the *Where* clause. One Where clause can contain several conditions chained by logical operators. Also, conditions can be organized in nested Where clauses, which is equivalent to placing conditions in brackets.

Comparisons and Constants

Typically, a condition is a comparison of a particular field with another field, a constant, or null. The compared field is specified in the first type parameter, while the comparison goes in the second.

```
PXSelect<Product, Where<Product.bookedQty, Greater<Product.availQty>>>
```

This statement is translated into the following SQL query, which selects all Product records with the BookedQty field greater than the AvailQty field.

```
SELECT * FROM Product WHERE Product.BookedQty > Product.AvailQty
```

There are a number of other *comparisons* such as NotEqual, Greater, and Less. They all can be used to compare one field to another field or a constant.

Constants are BQL classes derived from the Constant<Type> class. The *predefined constants* include boolean values True and False, integer Zero, datetime Now, Today, and MaxDate, and string StringEmpty. The following BQL statement selects all Product records with the Active field equal to True.

```
PXSelect<Product, Where<Product.active, Equal<True>>>
```

A field can also be compared to null (to check if the field value has not been specified) using the IsNull comparison, as follows.

```
PXSelect<Product, Where<Product.bookedQty, IsNull>>
```

This statement is translated into the following SQL query.

```
SELECT * FROM Product WHERE Product.BookedQty IS NULL
```

Or you could reverse this condition by using a variant of Where with one type parameter and the logical operator Not.

```
PXSelect<Product, Where<Not<Product.bookedQty, IsNull>>>
```

Below is the corresponding SQL query.

```
SELECT * FROM Product WHERE NOT (Product.BookedQty IS NULL)
```

The predefined constant Null cannot be used in the Where clause with Equal to select records with null fields. The Null constant is used in Switch conditions in *Arithmetic Operations*.

Several Conditions in One Where Clause

It is possible to specify several comparisons in one Where clause. For this purpose, you should use a variant of the Where clause with three type parameters: Where<Operand, Comparison, NextOperator>. The third type parameter is set to a logical operator (And/And2 or Or/Or2), as the following example shows.

```
PXSelect<Product,
    Where<Product.bookedQty, Greater<Product.availQty>,
        Or<Product.availQty, Less<Product.minAvailQty>>>>
```

This statement will be translated into the following SQL query.

```
SELECT * FROM Product
WHERE Product.BookedQty > Product.AvailQty
    OR Product.AvailQty < Product.MinAvailQty
```

This query selects products with BookedQty greater than AvailQty or AvailQty less than MinAvailQty.

You can chain any number of comparisons using binary operators with three type parameters. The third type parameter is again a set to binary operator, as shown in the following example.

```
PXSelect<Product,
    Where<Product.bookedQty, Greater<Product.availQty>,
        Or<Product.availQty, Less<Product.minAvailQty>,
        Or<Product.availQty, IsNull>>>>
```

The corresponding SQL query is given below.

```
SELECT * FROM Product
WHERE Product.BookedQty > Product.AvailQty
    OR Product.AvailQty < Product.MinAvailQty
    OR Product.AvailQty IsNull
```

To write more complex conditional expressions with logical operator of different type, you may need to separate some conditions with brackets. For this purpose, you should use nested Where/Where2 clauses.

In the last example above, brackets would be superfluous, since the conditions are joined by the same logical operator.

Building Complex Where Clauses

To surround a conditional expression part by brackets in the resulting SQL query, you should use a nested Where/Where2 clause. Brackets may be required in expressions that use different types of logical operators.

The steps below illustrate the construction of a complex conditional expression on two samples. One sample expression starts with a simple condition (an operand and a comparison) and has the following form: (C1 and not C2 and (C3 or C4 or (C5 and C6)) and not (C7 or C8)). Here, C with a number denotes a specific condition. This expression will be wrapped into the Where clause. The other sample expressions starts with a group of simple conditions: ((C1 or C2) and (C3 or C4) and (C5 or C6)). This expression will be wrapped into the Where2 clause.

A conditional expression is build by the following rules:

1. Each group (a pair of brackets) is replaced by a Where, Where2, Not, or Not2 clause:

    - Where is used for groups that start with a simple condition. Not is used for the same groups but preceded with logical "not".

- `Where2` is used for groups that start with a group. `Not2` is used for the same groups but preceded with logical "not".

2. Components of each group are chained using `And` (`Or`) or `And2` (`Or2`):

   - Simple conditions at the beginning of a group are chained using `And` (`Or`). If a condition is preceded by `not`, it is wrapped in `Not`.

     ```
     Where<C1, And<Not<C2>, ... >
     ```

   - All groups except for the last one are chained using `And2` (`Or2`), the last one is chained using `And`. The first parameter inside a logical operator is `Where` (or `Where2`). `not` preceding a group is placed inside a `Where` clause.

     ```
     Where<C1, And<Not<C2>, And2<Where<...>, And<Not<...>>>>>
     ```

     "Chained" means that each next logical operator is inserted as a type parameter into the previous one. Below is another example.

     ```
     Where2<Where<C1 or C2>, And2<Where<C3 or C4>, And<Where<C5 or C6>>>>
     ```

3. Align logical operators of the same level so that they have the same indent (typically, the indent of the enclosing `Where` clause plus four more spaces). Do not add line breaks before nested `Where` clauses.

   ```
   Where<C1,
       And<Not<C2>,
       And2<Where<C3 or C4 or (C5 and C6)>,
       And<Not<C7 or C8>>>>>
   ```

   Expanding nested `Where` clauses and breaking them into lines, we get the following.

   ```
   Where<C1,
       And<Not<C2>,
       And2<Where<C3,
                Or<C4,
                Or<Where<C5,
                       And<C6>>>>>,
       And<Not<C7,
                Or<C8>>>>>>
   ```

   As a result, each simple condition is placed on a separate line. For the second example, you first get the following code.

   ```
   Where2<Where<C1 or C2>,
       And2<Where<C3 or C4>,
       And<Where<C5 or C6>>>>
   ```

   And, expanding nested `Where` clauses and breaking them into lines, you get this:

   ```
   Where2<Where<C1,
              Or<C2>>,
       And2<Where<C3,
              Or C4>,
       And<Where<C5,
              Or<C6>>>>>
   ```

4. Conditions are substituted by the corresponding field-comparison pairs.

   ```
   Where<Field1, Comparison1,
       And<Not<Field2, Comparison2>,
       And2<Where<Field3, Comparison3,
   ```

```
              Or<Field4, Comparison4,
              Or<Where<Field5, Comparison5,
                     And<Field6, Comparison6>>>>>,
     And<Not<Field7, Comparison7,
              Or<Field8, Comparison8>>>>>>
```

Unlike the previous examples, this is at last valid BQL code (provided fields and comparisons are represented by valid BQL code). It can be used in `PXSelect` statements as the `Where` clause.

5. All lines except the last line of the BQL statement are ended with a comma. You should ensure that the right number of closing angle brackets are inserted.

In the BQL statements above, the type parameters set to fields are actually *operand*s. An operand can be a field as well as an *arithmetic expressions* involving several fields.

Example with Products

Suppose you need to select all `Product` data records with the `Active` field equal to `True`, and either `BookedQty` greater than `AvailQty` or `AvailQty` less than `MinAvailQty`.

This is a group of a simple condition (`Product.active` equals `True`) and another group joined by the "and" operator. This is implemented by the following BQL statement.

```
PXSelect<Product,
    Where<Product.active, Equal<True>,
        And<Where<Product.bookedQty, Greater<Product.availQty>,
                Or<Product.availQty, Less<Product.minAvailQty>>>>>>
```

The corresponding SQL query look as follows.

```
SELECT * FROM Product
WHERE Product.active = 1
    AND(Product.bookedQty > Product.availQty
        OR Product.availQty < Product.minAvailQty)
```

Suppose the conditional expression in this example is extended to take into account `Product` data records with the null `Active` field values. Then the new condition is added to "`Product.active` equals `True`" using `Or`. The resulting conditional expression will consists of two `Where` groups enclosed in `Where2`.

```
Where2<Where<...>, And<Where<...>>>
```

Nested `Where` clauses have the following structure.

```
Where<C1, Or<C2>>
```

The entire BQL statement will look a follows.

```
PXSelect<Product,
    Where2<Where<Product.active, Equal<True>,
               Or<Product.active, IsNull>>,
       And<Where<Product.bookedQty, Greater<Product.availQty>,
               Or<Product.availQty, Less<Product.minAvailQty>>>>>>
```

This statement is translated into the following SQL query.

```
SELECT * FROM Product
WHERE (Product.Active = 1
       OR Product.Active IS NULL)
   AND(Product.BookedQty > Product.AvailQty
       OR Product.AvailQty < Product.MinAvailQty)
```

To additionally ensure that none of the `BookedQty`, `AvailQty`, and `MinAvailQty` is null, you can join three simple conditions to the existing conditional expression using `And`.

If the new conditions are added to the end of the overall expression, `Where2` remains the outer clause (since the first its component is still a `Where` group). The `And` operator chaining nested `Where` groups becomes `And2`.

```
PXSelect<Product,
    Where2<Where<Product.active, Equal<True>,
               Or<Product.active, IsNull>>,
       And2<Where<Product.bookedQty, Greater<Product.availQty>,
               Or<Product.availQty, Less<Product.minAvailQty>>>,
       And<Product.bookedQty, IsNotNull,
       And<Product.availQty, IsNotNull,
       And<Product.minAvailQty, IsNotNull>>>>>>
```

If the new conditions are added to the beginning of the expression, the outer clause changes to `Where`, and the first nested `Where` group becomes chained using `And2`.

```
PXSelect<Product,
    Where<Product.bookedQty, IsNotNull,
        And<Product.availQty, IsNotNull,
        And<Product.minAvailQty, IsNotNull,
        And2<Where<Product.active, Equal<True>,
                 Or<Product.active, IsNull>>,
        And<Where<Product.bookedQty, Greater<Product.availQty>,
                 Or<Product.availQty, Less<Product.minAvailQty>>>>>>>>>
```

These two BQL statements are equivalent and correspond to the following SQL query.

```
SELECT * FROM Product
WHERE (Product.Active = 1
       OR Product.Active IS NULL)
   AND(Product.BookedQty > Product.AvailQty
       OR Product.AvailQty < Product.MinAvailQty)
   AND Product.BookedQty IS NOT NULL
   AND Product.AavailQty IS NOT NULL
   AND Product.MinAvailQty IS NOT NULL
```

Finally, suppose the resulting set should be extended with the `Product` data records that have the null `AvailQty` field. Then the "`Product.availQty` is not null" condition should be appended to the entire conditional expression from the previous example via `Or`. They should be wrapped by a new `Where2` clause in the following way.

```
Where2<ExistingExpression, Or<NewCondition>>
```

The BQL statement will become something like this.

```
PXSelect<Product,
    Where2<Where<Product.bookedQty, IsNotNull,
               And<Product.availQty, IsNotNull,
               And<Product.minAvailQty, IsNotNull,
               And2<Where<Product.active, Equal<True>,
                       Or<Product.active, IsNull>>,
               And<Where<Product.bookedQty, Greater<Product.availQty>,
                       Or<Product.availQty, Less<Product.minAvailQty>>>>>>>>,
        Or<Product.availQty, IsNotNull>>>
```

It is translated into the following SQL query.

```
SELECT * FROM Product
WHERE (Product.BookedQty IS NOT NULL
       AND Product.AavailQty IS NOT NULL
       AND Product.MinAvailQty IS NOT NULL
       AND(Product.Active = 1
```

```
            OR Product.Active IS NULL)
        AND(Product.BookedQty > Product.AvailQty
            OR Product.AvailQty < Product.MinAvailQty))
     OR Product.AvailQty IS NOT NULL
```

Where Clauses

The `Where` clause specifies filtering expressions for BQL statements. A `PXSelect` statement with the `Where` clause selects only the data records that satisfy the filtering expression.

The `Where` clause can be specified in `PXSelect`, `Select`, and `Search` statements as well as in the *On* and *Case* clause. Also, a group of conditions in brackets is implemented in a BQL statement by a *nested* `Where` clause.

Where<Operand, Comparison> : IBqlWhere

Specifies a single filtering condition.

Examples:

```
PXSelect<Table,
    Where<Table.field1, Equal<Table.field2>>>
```

This is translated into:

```
SELECT * FROM Table
WHERE Table.Field1 = Table.Field2
```

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison`

Where<Operand, Comparison, NextOperator> : IBqlWhere

Specifies a particular condition in the two first type parameters and attaches one more logical operator (`And` or `Or`).

Examples:

```
PXSelect<Table,
    Where<Table.field1, Greater<Table.field2>,
        And<Table.field3, IsNull>>>
```

The `NextOperator` type parameter can specify a single condition or a group of conditions, or again continue the `Where` expression:

```
PXSelect<Table,
    Where<Table.field1, Greater<Table.field2>,
        And<Table.field3, IsNull,
        And<Table.field4, Equal<Today>>>>>
```

This is translated into:

```
SELECT * FROM Table
WHERE Table.Field1 > Table.Field2
    AND Table.Field3 IS NULL
    AND Table.Field4 = [today date]
```

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison`

- `NextOperator : IBqlBinary`

Where<Operator> : IBqlWhere

Specifies an unary operator as the filtering expression. The unary operator is either the *Not* or *Match* operator.

Examples:

```
PXSelect<Table,
    Where<Not<Table.field1, IsNotNull,
              And<Table.field2, LessEqual<Table.field1>>>>>
```

This is translated into:

```
SELECT * FROM Table
WHERE NOT ( Table.Field1 IS NOT NULL
            AND Table.Field2 <= Table.Field1 )
```

Type Parameters:

- `Operator : IBqlUnary`

Where2<Operator, NextOperator> : IBqlWhere

Specifies a complex condition group where the first component is again a group.

Examples:

A filtering expression of the form `((C1 and C2) or (C3 and C4))`, where `C` with a number denotes a single condition, is implemented by the BQL code of the following form:

```
Where2<Where<C1,
           And<C2>>,
    Or<Where<C3,
           And<C4>>>>
```

A full expression of this type may look as something like this:

```
Where2<Where<Table.field2, Greater<Table.field1>
           And<Table.field3, Between<Table.field1, Table.field2>>>,
    Or<Where<Table.field3, IsNull,
           And<Table.field1, Equal<Table.field2>>>>>
```

This is translated into:

```
WHERE ( Table.Field2 > Table.Field1
        AND Table.Field3 BETWEEN Table.Field1 AND Table.Field2 )
   OR ( Table.Field3 IS NULL
        AND Table.Field1 = Table.Field2 )
```

Type Parameters:

- `Operator : IBqlUnary`
- `NextOperator : IBqlBinary`

Comparisons

Comparison operators compare an operand with another operand. An operand is a constant, a particular field, or an expression built from fields and constants using *functions*.

The following BQL statement demonstrates the usage of the Greater and Between comparison operators.

```
PXSelect<Table,
    Where<Table.field1, Greater<Table.field2>,
        And<Table.field3, Between<Table.field1, Table.field2>>>>
```

The first compared operand goes in the BQL statement right before the comparison. The second compared operand is specified as the type parameter of a comparison. Here, the Greater operator compares Table.field1 with Table.field2. The condition is true if the latter is greater than the former. The Between operator sets the condition that is true when Table.field3 value is between the Table.field1 and Table.field2 values.

The BQL statement above is translated into the following SQL query.

```
SELECT * FROM Table
WHERE Table.Field1 > Table.Field2
    AND Table.Field3 BETWEEN Table.Field1 AND Table.Field2
```

The preceding operand and the comparison together constitute a condition. Conditions are concatenated using *logical operators*.

Equal<Operand> : IBqlComparison

Compares the preceding operand with Operand for equality.

Type Parameters:

- Operand : IBqlOperand

NotEqual<Operand> : IBqlComparison

Checks if the preceding operand is not equal to Operand.

Type Parameters:

- Operand : IBqlOperand

Greater<Operand> : IBqlComparison

Checks if the preceding operand is greater than Operand.

Type Parameters:

- Operand : IBqlOperand

Less<Operand> : IBqlComparison

Checks if the preceding operand is less than Operand.

Type Parameters:

- Operand : IBqlOperand

LessEqual<Operand> : IBqlComparison

Checks if the preceding operand is less or equal to Operand.

Type Parameters:

- Operand : IBqlOperand

GreaterEqual<Operand> : IBqlComparison

Checks if the preceding operand is greater or equal to Operand.

Type Parameters:

- `Operand : IBqlOperand`

Like<Operand> : IBqlComparison

Compares the preceding operand with the pattern specified in `Operand`. Equivalent to the SQL operator `LIKE`.

`Operand` should have a wildcard string value in which the sign "`%`" is used to substitute missing letters. For example, "`%land%`" will be matched by "`Iceland`" and "`Laplandia`".

Type Parameters:

- `Operand : IBqlOperand`

NotLike<Operand> : IBqlComparison

Checks if the preceding operand does not match the pattern specified in `Operand`. Equivalent to SQL operator `NOT LIKE`.

Type Parameters:

- `Operand : IBqlOperand`

Between<Operand1, Operand2> : IBqlComparison

Checks if the value of the preceding operand falls between the values of `Operand1` and `Operand2`. Equivalent to SQL operator `BETWEEN`.

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

NotBetween<Operand1, Operand2> : IBqlComparison

Checks if the value of the preceding operand does not fall between the values of `Operand1` and `Operand2`. Equivalent to SQL operator `NOT BETWEEN`.

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

IsNull : IBqlComparison

Checks if the preceding field is null. Equivalent to SQL operator `IS NULL`.

IsNotNull : IBqlComparison

Checks if the preceding field is not null. Results in true for data records with this field containing a value. Equivalent to SQL operator `IS NOT NULL`.

Logical Operators

Logical operators concatenate conditions and condition groups into conditional expressions. They can be used in *Where* and *On* clauses.

To append one more logical operator to the current one, you should use a form with the `NextOperator` type parameter. `NextOperator` is set to the next logical operator. For example, an expression `(C1 and`

`C2 and C3 and C4)` corresponds to a BQL code of the following form (`C` with a number denotes a single condition).

```
Where<C1, And<C2, And<C3, And<C4>>>>
```

The BQL statement below gives an example of such expression.

```
PXSelect<Table
    Where<Table.field1, Equal<Table.field2>,
        And<Table.field3, Greater<Zero>,
        And<Table.field3, IsNotNull>,
        And<Table.field4, Less<Table.field5>>>>>>
```

This is translated into the following SQL query.

```
SELECT * FROM Table
WHERE Table.Field1 = Table.Field2
    AND Table.Field3 > 0
    AND Table.Field3 IS NOT NULL
    AND Table.Field4 < Table.Field5
```

And<Operand, Comparison> : IBqlBinary

Appends a single condition to a conditional expression via logical "and".

Examples:

```
And<Table.field1, Greater<Table.field2>>
```

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison`

And<Operand, Comparison, NextOperator> : IBqlBinary

Appends a single condition to a conditional expression via logical "and" and continues the chain of conditions. The condition is set by `Operand` and `Comparison`. `NextOperator` is set to `And` (`And2`) or `Or` (`Or2`) operator which continues the filtering expression.

Examples:

```
And<Table.field1, IsNull,
And<Table.field2, IsNotNull,
And<...>>>
```

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison, new()`
- `NextOperator : IBqlBinary`

And<Operator> : IBqlBinary

Appends a unary operator to a conditional expression via logical "and". The unary operator is the `Not`, `Where`, or `Match` operator.

Examples:

```
And<Not<Table.field1, Equal<Zero>>>
```

Type Parameters:

- `Operator : IBqlUnary`

And2<Operator, NextOperator> : IBqlBinary

Appends a unary operator to a conditional expression via logical "and" and continues the chain of conditions. The unary operator is the `Not`, `Where`, or `Match` operator.

Type Parameters:

- `Operator : IBqlUnary, new()`
- `NextOperator : IBqlBinary`

Or<Operand, Comparison> : IBqlBinary

Appends a single condition or a group of conditions wrapped in `Where` to a conditional expression via logical "or".

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison`

Or<Operand, Comparison, NextOperator> : IBqlBinary

Appends a single condition to a conditional expression via logical "or" and continues the chain of conditions. The condition is set by `Operand` and `Comparison`. `NextOperator` is set to `And` (`And2`) or `Or` (`Or2`) operator which continues the filtering expression.

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison, new()`
- `NextOperator : IBqlBinary`

Or<Operator> : IBqlBinary

Appends a unary operator to a conditional expression via logical "or". The unary operator is the `Not`, `Where`, or `Match` operator.

Type Parameters:

- `Operator : IBqlUnary`

Or2<Operator, NextOperator> : IBqlBinary

Appends a unary operator to a conditional expression via logical "or" and continues the chain of conditions. The unary operator is the `Not`, `Where`, or `Match` operator.

Type Parameters:

- `Operator : IBqlUnary, new()`
- `NextOperator : IBqlBinary`

Not<Operand, Comparison> : IBqlUnary

Adds logical "not" to a single condition.

Type Parameters:

- `Operand : IBqlOperand`

- `Comparison : IBqlComparison`

Not<Operand, Comparison, NextOperator> : IBqlUnary

Adds logical "not" to a conditional expression. In the resulting SQL, the group is preceded with `not` and surrounded by brackets.

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison`
- `NextOperator : IBqlBinary`

Not<Operator> : IBqlUnary

Add logical "not" to a unary operator. A unary operator is the `Where` or `Match` operator. In the resulting SQL the group is preceded with `not` and surrounded by brackets.

Type Parameters:

- `Operator : IBqlUnary`

Not2<Operator, NextOperator> : IBqlUnary

Add logical "not" to a unary operator. A unary operator is the `Where` or `Match` operator. In the resulting SQL the group is preceded with `not` and surrounded by brackets.

Type Parameters:

- `Operator : IBqlUnary`
- `NextOperator : IBqlBinary`

Match<Parameter> : IBqlUnary

Matches only the data records the specified user has access rights for. The condition is applied to the data records of the first table mentioned in a BQL statement. The user is specified in `Parameter`, typically through the `Current` parameter.

Examples:

```
PXSelect<Table,
    Where<Match<Current<AccessInfo.userName>>>>
```

Type Parameters:

- `Parameter : IBqlParameter`

Match<Table, Parameter> : IBqlUnary

Matches only the data records the specified user has access rights for. The condition is applied to the data records of the table set with `Table`. The user is specified in `Parameter`, typically through the `Current` parameter.

This form of `Match` is used when the filtered table is added though a join clause.

Examples:

```
PXSelectJoin<Table1,
    InnerJoin<Table2, On<Table1.field1, Equal<Table2.field2>>>,
    Where<Match<Table2, Current<AccessInfo.userName>>>>
```

Type Parameters:

- `Table : IBqlTable`
- `Parameter : IBqlParameter`

CurrentMatch<Field> : IBqlUnary

Equivalent to `Match<Field>`, but is used in the `PXProjection` attribute.

Type Parameters:

- `Field : IBqlField`

CurrentMatch<Table, Field> : IBqlUnary

Equivalent to `Match<Table, Field>`, but is used in the `PXProjection` attribute.

Type Parameters:

- `Table : IBqlTable`
- `Field : IBqlField`

MatchWithBranch<Field> : IBqlUnary

Matches the data records whose field is null or holds the ID of a branch that can be accessed from within the current branch. The current branch is the branch to which the user is signed in.

Type Parameters:

- `Field : IBqlOperand`

  A field where to look for the branch ID whose rights should be checked.

MatchWithBranch<Field, Parameter> : IBqlUnary

Matches the data records whose field is null or holds the ID of a branch that can be accessed from within the specified branch or its subsidiaries.

Type Parameters:

- `Field : IBqlOperand`

  A field where to look for the branch ID whose rights should be checked.
- `Parameter : IBqlParameter`

  The branch to check against the branch found in `Field`.

Constants

Constants represent predefined values. They can be used in conditional expressions, for comparison with fields, and in arithmetic expressions.

Constants are implemented as classes derived from the generic `Constant<ConstType>` class. You can define custom constants.

Constant<ConstType> : Constant, IBqlOperand, IBqlCreator

The base class for BQL constants.

To define a custom constant in the application, derive a class from `Constant`. Specify constant's type in the `ConstType` type parameter and implement the constructor. The constructor should inherit base class constructor and provide the constant's actual value in its argument.

Examples:

The predefined constant Zero represents integer 0 and is not suitable for comparison with decimal values. The application should define a custom constant for decimal zero, deriving it from Constant<Decimal> in the following way:

```
public class decimal_0 : Constant<Decimal>
{
    public decimal_0()
        : base(0m)
    {
    }
}
```

This constant can be used in BQL statements in the following way:

```
PXSelect<Table,
    Where<Table.decimalField, Greater<decimal_0>>>
```

This BQL statement is tranlsated into the following SQL query:

```
SELECT * FROM Table
WHERE Table.DecimalField > .0
```

Null : IBqlOperand, IBqlCreator

The null value used in Switch clauses as a default value. Don't use this constant for checking fields for null value — use the IsNull and IsNotNull comparisons instead.

Now : Constant<DateTime>

Current UTC time.

Today : Constant<DateTime>

Represents today date.

Tomorrow : Constant<DateTime>

Represents tomorrow date.

True : Constant<short>

The true value for comparing with boolean fields. In translation to SQL corresponds to CONVERT(BIT, 1).

False : Constant<short>

The false value for comparing with boolean fields. In translation to SQL corresponds to CONVERT(BIT, 0).

Zero : Constant<int>

The integer zero, not comparable with floating point numeric types (such as decimal).

StringEmpty : Constant<string>

An empty string.

MaxDate : Constant<DateTime>

The maximum date: 06/06/2079.

Querying Multiple Tables

BQL statements can join several database tables using the following clauses directly mapped to SQL `JOIN` clauses:

- `InnerJoin` returns all records where there is at least one match in *both* tables.
- `LeftJoin` returns all records from the left table, and the matched records from the right table. Where there are no matched records from the right table, null values are inserted.
- `RightJoin` returns all records from the right table, and the matched records from the left table. Where there are no matched records from the left table, null values are inserted.
- `FullJoin` returns all records when there is a match in *one* of the tables.
- `CrossJoin` returns the entire Cartesian product of two tables.

A result set record of a BQL statement with joins consists of all fields of each of the joined tables. Such record (as an instance of the `PXResult<>` class) *can be cast* to any of the DACs corresponding to the joined tables.

A join clause is specified as the second type parameter of `PXSelectJoin` and other forms of `PXSelectJoin` that have a type parameter derived from `IBqlJoin`, as follows.

```
PXSelectJoin<SalesOrder,
    InnerJoin<OrderDetail,
        On<OrderDetail.orderNbr, Equal<SalesOrder.orderNbr>>>>
```

This BQL statement will select all `SalesOrder` records along with related `OrderDetail` records. It is translated into the following SQL code.

```
SELECT * FROM SalesOrder
INNER JOIN OrderDetail
    ON OrderDetail.OrderNbr = SalesOrder.OrderNbr
```

Each join clause has two variants, with two type parameters and with three type parameters. You use the version with two type parameters to provide one join clause. To specify several join clauses, you the version with three type parameters. Each next join clause is specified as the last type parameter of the previous join clause, as shown in the following code.

```
PXSelectJoin<SalesOrder,
    InnerJoin<OrderDetail,
        On<OrderDetail.orderNbr, Equal<SalesOrder.orderNbr>>,
    LeftJoin<Employee,
        On<Employee.employeeID, Equal<SalesOrder.employeeID>>>>>
```

Which is translated to the following SQL query.

```
SELECT * FROM SalesOrder
INNER JOIN OrderDetail
    ON OrderDetail.OrderNbr = SalesOrder.OrderNbr
LEFT JOIN Employee
    ON Employee.EmployeeID = SalesOrder.EmployeeID
```

The `On` conditions in subsequent join clauses can refer to fields from any linked table. Also, the `On` clause can contain any number of conditions. These conditions should be chained by logical operators as in filtering conditions.

```
PXSelectJoin<SalesOrder,
    InnerJoin<OrderDetail,
        On<OrderDetail.orderNbr, Equal<SalesOrder.orderNbr>>,
    LeftJoin<Employee,
        On<Employee.employeeID, Equal<SalesOrder.employeeID>>,
    RightJoin<Product,
        On<Product.productID, Equal<OrderDetail.productID>,
```

```
        And<Product.unitPrice, Equal<OrderDetail.unitPrice>>>>>>>
```

This is traslated into the following SQL query.

```
SELECT * FROM SalesOrder
INNER JOIN OrderDetail
    ON OrderDetail.OrderNbr = SalesOrder.OrderNbr
LEFT JOIN Employee
    ON Employee.EmployeeID = SalesOrder.EmployeeID
RIGHT JOIN Product
    ON (Product.ProductID = OrderDetail.ProductID AND
        Product.UnitPrice = OrderDetail.UnitPrice)
```

For `CrossJoin`, the `On` condition is not spesified, since it creates an unrestricted set of all possible pairs of records from two tables. An example is given below.

```
PXSelectJoin<Product, CrossJoin<Supplier>>
```

This is translated into the following SQL query.

```
SELECT * FROM Product CROSS JOIN Supplier
```

Attaching the Where Clause

To add the `Where` clause, you should take an appropriate `PXSelect` variant. `Where` is specified after the joining operator.

The following BQL statement joins the `SupplierProduct` (which implements a many-to-many relationship) and `Supplier` tables to the `Product` table and filters them by `SupplierProduct` fields.

```
PXSelectJoin<Product,
    InnerJoin<SupplierProduct,
        On<SupplierProduct.productID, Equal<Product.productID>>,
    InnerJoin<Supplier,
        On<Supplier.accountID, Equal<SupplierProduct.accountID>>>>,
    Where<SupplierProduct.lastPurchaseDate, IsNotNull,
        And<SupplierProduct.lastSupplierPrice, LessEqual<Product.unitPrice>>>>
```

This BQL statement is translated into the following SQL code.

```
SELECT * FROM Product
INNER JOIN SupplierProduct
    ON SupplierProduct.ProductID = Product.ProductID
INNER JOIN Supplier
    ON Supplier.AccountID = SupplierProduct.AccountID
WHERE SupplierProduct.LastPurchaseDate IS NOT NULL
    AND SupplierProduct.LastSupplierPrice <= Product.UnitPrice
```

Note that the `Where` conditional expression applies to the set formed by all joined tables. In particular, the filtering conditions can refer to any field of any of the joined tables.

Attaching the OrderBy Clause

The `OrderBy` clause is specified after the `Where` clause if there is one in the statement, or after the join clause.

If a BQL statement should include a join clause and applying filtering and ordering, it is based on the `PXSelectJoin` version of with four type parameters.

```
PXSelectJoin<SalesOrder,
    InnerJoin<OrderDetail,
        On<OrderDetail.orderNbr, Equal<SalesOrder.orderNbr>>>,
    Where<SalesOrder.requiredDate, Less<Today>>,
```

```
    OrderBy<Desc<OrderDetail.orderDetailQty>>>
```

This BQL statement is translated into the following SQL query.

```
SELECT * FROM SalesOrder
INNER JOIN OrderDetail
    ON OrderDetail.OrderNbr = SalesOrder.OrderNbr
WHERE SalesOrder.RequiredDate < [today date]
ORDER BY OrderDetail.OrderDetailQty DESC
```

If a BQL statement should include only a join clause and apply ordering, it is based on `PXSelectOrderBy` with three type parameters, as follows.

```
PXSelectOrderBy<Product,
    LeftJoin<OrderDetail,
        On<OrderDetail.productID, Equal<Product.productID>,
        AND<OrderDetail.unitPrice, Equal<Product.unitPrice>>>>,
    OrderBy<Asc<Product.productName>>>>
```

This is translated into the following SQL query.

```
SELECT * FROM Product
LEFT JOIN OrderDetail
    ON (OrderDetail.ProductID = Product.ProductID AND
        OrderDetail.UnitPrice = Product.UnitPrice)
ORDER BY Product.ProductName
```

Join Clauses

"Join" clauses link other tables to the main one specified as the first type parameter in the BQL statement. An example is given below.

```
PXSelectJoin<Table1,
    InnerJoin<Table2, On<Table2.field2, Equal<Table1.field1>>,
    LeftJoin<Table3, On<Table3.field3, Equal<Table1.field4>>>>>
```

This is tranlsated into the following SQL query.

```
SELECT * FROM Table1
INNER JOIN Table2
    ON Table2.Field2 = Table1.Field1
LEFT JOIN Table3
    ON Table3.Field3 = Table1.Field4
```

Conditional expression for joining is specified using the *On* classes. The syntax for conditional expressions set in `On` is the same as used in `Where`.

InnerJoin<Table, On> : IBqlJoin

Joins a table via `INNER JOIN`.

Type Parameters:

- `Table : IBqlTable`
- `On : IBqlOn`

InnerJoin<Table, On, NextJoin> : IBqlJoin

Joins a table via `INNER JOIN` and allows joining one or several more tables..

Type Parameters:

- `Table : IBqlTable`

- `On : IBqlOn, new()`
- `NextJoin : IBqlJoin`

LeftJoin<Table, On> : IBqlJoin

Joins a table via `LEFT JOIN`.

Type Parameters:

- `Table : IBqlTable`
- `On : IBqlOn`

LeftJoin<Table, On, NextJoin> : IBqlJoin

Joins a table via `LEFT JOIN` and allows joining one or several more tables..

Type Parameters:

- `Table : IBqlTable`
- `On : IBqlOn, new()`
- `NextJoin : IBqlJoin`

RightJoin<Table, On> : IBqlJoin

Joins a table via `RIGHT JOIN`.

Type Parameters:

- `Table : IBqlTable`
- `On : IBqlOn`

RightJoin<Table, On, NextJoin> : IBqlJoin

Joins a table via `RIGHT JOIN` and allows joining one or several more tables..

Type Parameters:

- `Table : IBqlTable`
- `On : IBqlOn, new()`
- `NextJoin : IBqlJoin`

FullJoin<Table, On> : IBqlJoin

Joins a table via `FULL JOIN`.

Type Parameters:

- `Table : IBqlTable`
- `On : IBqlOn`

FullJoin<Table, On, NextJoin> : IBqlJoin

Joins a table via `FULL JOIN` and allows joining one or several more tables..

Type Parameters:

- `Table : IBqlTable`
- `On : IBqlOn, new()`

- `NextJoin : IBqlJoin`

CrossJoin<Table> : IBqlJoin

Joins a table via `CROSS JOIN`. Not joining condition is specified.

Examples:

```
PXSelectJoin<Table1, CrossJoin<Table2>>
```

This is translated into:

```
SELECT * FROM Table1 CROSS JOIN Table2
```

Type Parameters:

- `Table : IBqlTable`

CrossJoin<Table, NextJoin> : IBqlJoin

Joins a table via `CROSS JOIN` and allows joining one or several more tables.

Type Parameters:

- `Table : IBqlTable`
- `NextJoin : IBqlJoin`

On Clause

The `On` clause defines the conditional expression for table *joining*.

On<Operand, Comparison> : IBqlOn

Specifies a single joining condition. Corresponds to SQL keyword `ON`.

Examples:

```
PXSelectJoin<Table1,
    InnerJoin<Table2, On<Table2.field2, Equal<Table1.field1>>>>
```

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison`

On<Operator> : IBqlOn

Specifies the joining condition through the `Not`, `Where`, or `Where2` clause. Corresponds to SQL keyword `ON`.

Examples:

```
PXSelectJoin<Table1,
    InnerJoin<Table2, On<Not<Table2.field2, Equal<Table1.field1>>>>>
```

Type Parameters:

- `Operator : IBqlUnary`

On<Operand, Comparison, NextOperator> : IBqlOn

Specifies a single joining condition and allows continuing the chain of conditions using a logical operator. Corresponds to SQL keyword `ON`.

Examples:

```
PXSelectJoin<Table1,
    InnerJoin<Table2,
        On<Table2.field1, Equal<Table1.field2>,
        And<Table2.field3, Equal<Table1.field4>>>>>
```

This is translated into:

```
SELECT * FROM Table1
INNER JOIN Table2 ON
    Table2.Field1 = Table1.Field2 AND Table2.Field3 = Table1.Field4
```

Type Parameters:

- `Operand : IBqlOperand`
- `Comparison : IBqlComparison`
- `NextOperator : IBqlBinary`

On2<Operator, NextOperator> : IBqlOn

Specifies the joining condition using `Not`, `Where`, or `Where2` and allows continuing the chain of conditions using a logical operator. Corresponds to SQL keyword `ON`.

Type Parameters:

- `Operator : IBqlUnary`
- `NextOperator : IBqlBinary`

Grouping and Aggregating

The BQL grouping and aggregating syntax is similar to the SQL syntax. BQL implements:

- The `GroupBy` clause for grouping
- The equivalents of SQL aggregation functions: `Min`, `Max`, `Sum`, `Avg`, and `Count`.

All grouping conditions and aggregation functions are specified in the `Aggregate` clause. For example, to group the result set by a field, place the `GroupBy` clause into `Aggregate` as follows.

```
PXSelectGroupBy<Product,
    Aggregate<GroupBy<Product.categoryCD>>>
```

Note that you should take an appropriate `PXSelect` version with the `Aggregate` type parameter, such as `PXSelectGroupBy<Table, Aggregate>`. The statement above is translated into the following SQL code.

```
SELECT Product.CategoryCD,
       [MAX(Field) or NULL for other fields]
FROM Product
GROUP BY Product.CategoryCD
```

Fields specified in `GroupBy` clauses are selected as is. To all other fields, an aggregation function is applied. The default `Max` function is used if no function is specified for a field. If `Max` cannot be applied to the type of a field, `NULL` is selected for it.

Another `GroupBy` clause or aggregation function is inserted as the second type parameter of the previous `GroupBy` clause or aggregation function.

```
PXSelectGroupBy<Product,
    Aggregate<GroupBy<Product.categoryCD, Sum<Product.availQty>>>>
```

This BQL statement will count the sum of of the `AvailQty` field for each group of records with equal `CategoryCD` field values. `NULL` is also considered a value here. The following SQL query corresponds to the statement above.

```
SELECT Product.CategoryCD, SUM(Product.AvailQty),
       [MAX(Field) or NULL for other fields]
FROM Product
GROUP BY Product.CategoryCD
```

Grouping can be applied to several fields. In this case, a combination of such fields is considered equal to another one only if all fields in them concide.

The previous example can be extended by adding the `GroupBy` clause for the `StockUnit` field. As a result, `Product` records will be grouped by both categories and stock units. Some aggregation functions might be added as well, as in the following example.

```
PXSelectGroupBy<Product,
    Aggregate<GroupBy<Product.categoryCD,
              Sum<Product.availQty,
              Sum<Product.bookedQty,
              GroupBy<Product.stockUnit,
              Min<Product.unitPrice>>>>>>>
```

This is translated into the following SQL query.

```
SELECT Product.CategoryCD, Product.StockUnit,
       SUM(Product.AvailQty), SUM(Product.AvailQty), MIN(Product.UnitPrice),
       [MAX(Field) or NULL for other fields]
FROM Product
GROUP BY Product.CategoryCD, Product.StockUnit
```

Aggregate and GroupBy Clauses

This set of classes implement SQL `GROUP BY` and the aggregate functions.

Unlike SQL, all grouping clauses and aggregations are gathered in a BQL statement in one `Aggregate` clause. The `Aggregate` clause is specified as the `PXSelectGroupBy` variant's type parameter .

In the SQL translation, alll fields not specified in `GroupBy` clauses are aggregated using:

- The aggregation function specified in the `Aggregate` clause
- The `MAX` function if no aggregation function is specified explicitly for a field
- `NULL` if `MAX` is not applicable to the field

For example, consider the folloing BQL statement.

```
PXSelectGroupBy<Table,
    Aggregate<GroupBy<Table.field1>>>
```

It is translated into:

```
SELECT Table.Field1,
       [MAX(Table.Field) or NULL for all fields]
FROM Table
GROUP BY Table.Field1
```

While the following BQL statement:

```
PXSelectGroupBy<Table,
    Aggregate<GroupBy<Table.field1,
              Avg<Table.field2,
              Min<Table.field3>>>>>
```

is translated into:

```
SELECT Table.Field1,
       AVG(Table.Field2), MIN(Table.Field3),
       [MAX(Table.Field) or NULL for all other fields]
FROM Table
GROUP BY Table.Field1
```

An aggregation BQL statement has a read-only result set.

Aggregate<Function> : IBqlAggregate

A wrapper clause for the `GroupBy` clauses and aggregation functions.

Examples:

The following BQL statement groups `Table` records by the `Table.field1` field and calculates sums of the `Table.field2` field in each group.

```
PXSelectGroupBy<Table,
    Aggregate<GroupBy<Table.field1, Sum<Table.field2>>>>
```

This is translated into the following SQL code.

```
SELECT Table.Field1, SUM(Table.Field2),
       [MAX(Table.Field) or NULL for other fields]
FROM Table
GROUP BY Table.Field1
```

Type Parameters:

- `Function : IBqlFunction`

GroupBy<Field> : IBqlFunction

Adds grouping by the field specified in `Field`. Equivalent to SQL operator `GROUP BY`.

Type Parameters:

- `Field : IBqlField`

GroupBy<Field, NextAggregate> : IBqlFunction

Adds grouping by the field specified in `Field` and continues the aggregation clause with `NextAggregate`. Equivalent to SQL operator `GROUP BY`.

Type Parameters:

- `Field : IBqlField`
- `NextAggregate : IBqlFunction`

Aggregation Functions

The aggregation functions are calculated for all field values in a group. To apply an aggregation to a field, specify the field in the type parameter and append the clause to the `Aggregate` operator.

Sum<Field> : IBqlFunction

Returns the sum of all `Field` values in a group. Equivalent to SQL function `SUM`.

Type Parameters:

- `Field : IBqlField`

Sum<Field, NextAggregate> : IBqlFunction

Returns the sum of all `Field` values in a group and continues the aggregation clause with `NextAggregate`. Equivalent to SQL function `SUM`.

Examples:

```
PXSelectGroupBy<Table,
    Aggregate<Sum<Table.field1,
              Sum<Table.field2,
              GroupBy<Table.field3>>>>>
```

Type Parameters:

- `Field : IBqlField`
- `NextAggregate : IBqlFunction`

Avg<Field> : IBqlFunction

Returns the average of the values of `Field` in a group. Equivalent to SQL function `AVG`.

Type Parameters:

- `Field : IBqlField`

Avg<Field, NextAggregate> : IBqlFunction

Returns the average of the values of `Field` in a group and continues the aggregation clause with `NextAggregate`. Equivalent to SQL function `AVG`.

Type Parameters:

- `Field : IBqlField`
- `NextAggregate : IBqlFunction`

Min<Field> : IBqlFunction

Returns the minimum value of `Field` in a group. Equivalent to SQL function `MIN`.

Type Parameters:

- `Field : IBqlField`

Min<Field, NextAggregate> : IBqlFunction

Returns the minimum value of `Field` in a group and continues the aggregation clause with `NextAggregate`. Equivalent to SQL function `MIN`.

Type Parameters:

- `Field : IBqlField`
- `NextAggregate : IBqlFunction`

Max<Field> : IBqlFunction

Returns the maximum value of `Field` in a group. Equivalent to SQL function `MAX`.

Type Parameters:

- `Field : IBqlField`

Max<Field, NextAggregate> : IBqlFunction

Returns the maximum value of `Field` in a group and continues the aggregation clause with `NextAggregate`. Equivalent to SQL function `MAX`.

Type Parameters:

- `Field : IBqlField`
- `NextAggregate : IBqlFunction`

Count : IBqlFunction

Counts the number of items in a group if a `GroupBy` clause is specified or, otherwise, the total number of records in the result set. In the translation to SQL, it is represented by `COUNT(*)` added to the list of selected columns.

You access the calculated value through the `RowCount` property of the `PXResult<>` type.

Examples:

```
PXResult<Table> res =
    PXSelectGroupBy<Table, Aggregate<Count>>.Select(this);

// The calculated number of records is stored in the
// PXResult.RowCount property.
int tableRecordsNumber = res.RowCount;
```

The BQL code in this example is translated into the following SQL query.

```
SELECT [MAX(Table.Field) or NULL for all fields defined in the Table DAC],
       COUNT(*)
FROM Table
```

Count<Field> : IBqlFunction

Counts distinct values of the specified field in a group. Equivalent to SQL function `COUNT DISTINCT`.

You access the calculated value through the `RowCount` property of the `PXResult<>` type. Note that you should use only one `Count<>` function in a BQL query, because you won't be able to access other such counted values.

Examples:

```
foreach(PXResult<Table> row in PXSelectGroupBy<Table,
    Aggregate<GroupBy<Table.field1, Count<Table.field2>>>>.Select(this))
{
    // The calculated number of distinct values of field2 in a group
    int field2CountInGroup = row.RowCount;
    ...
}
```

The BQL code in this example is translated into the following SQL query.

```
SELECT COUNT(DISTINCT Table.Field2),
      [MAX(Table.Field) or NULL for all other fields defined in the Table DAC]
FROM Table
GROUP BY Table.Field1
```

Type Parameters:

- `Field : IBqlField`

Using Parameters

BQL parameters are replaced in the translation to SQL with specific values. There are four type of parameters: `Current` (`Current2`), `Optional` (`Optional2`), `Required`, and `Argument`.

Current Parameter

By using the `Current` parameter in the declaration of a data view, you can reference another view to relate them to each other. A typical example is referencing the current master record on master-detail webpages.

The `Current` parameter actually inserts the `Current` object's field value from the `PXCache` object. For example, suppose the following BQL statement defines the master view.

```
// The view declaration in a graph
PXSelect<SalesOrder> MasterRecords;
```

The details view might be defined as follows.

```
// The view declaration in the same graph
PXSelect<OrderDetail,
    Where<OrderDetail.orderNbr, Equal<Current<SalesOrder.orderNbr>>>>
        DetailsRecords;
```

Exectuion of the second data view will produce the following SQL query.

```
SELECT * FROM OrderDetail
WHERE OrderDetail.OrderNbr = [parameter]
```

Here `[parameter]` is the `OrderNbr` value taken from the `Current` property of the `OrderDetail` cache.

This value can be obtained through the following code executed in a graph: `((OrderDetail)Caches[typeof(OrderDetail)].Current).OrderNbr`.

Suppose there is a many-to-one relationship between the `DocTransaction` and `Document` DACs. Let it be implemented through the `DocNbr` and `DocType` key fields. The views connecting `Document` and `DocTransaction` records might be defined as follows.

```
// The views declarations in a graph
PXSelect<Document> Documents;
PXSelect<DocTransaction,
    Where<DocTransaction.docNbr, Equal<Current<Document.docNbr>>,
        And<DocTransaction.docType, Equal<Current<Document.docType>>>>>
            DocTransactions;
```

Second view's execution will produce the following SQL query:

```
SELECT * FROM DocTransaction
WHERE DocTransaction.DocNbr = [parameter1]
    AND DocTransaction.DocType = [parameter2]
```

Where `[parameter1]` is the `DocNbr` value and `[parameter2]` is the `DocType` value taken from the `Current` property of the `DocTransaction` cache.

If the field specified in the `Current` parameter is null, the default value will be inserted. The default value assignment procedure takes into account the `PXDefault` attribute value and triggers the `FieldDefaulting` event handlers. The value eventually returned by the procedure is inserted into the SQL query in place of the `Current` parameter.

This procedure doesn't start if the `Current2` version of the parameter is used.

Required Parameter

To pass a specific value to the SQL query, you should use the `Required` parameter. To execute a BQL statement with the `Required` parameter, specify the value as the `Select()` method argument.

The `Required` parameter should be used only in the BQL statements that are executed in the application code. The value passed to `Select()` must be of the same type as the specified field.

The code below shows execution of BQL statement with the `Required` parameter.

```
// Suppose an event handendler related to the Product DAC
// is being executed
Product product = (Product)e.Row;

// Select the Category record with the specified CategoryCD
Category category =
    PXSelect<Category,
        Where<Category.categoryCD, Equal<Required<Category.categoryCD>>>>
        .Select(this, product.CategoryCD);
```

The BQL statement used in this example is translated into the following SQL query.

```
SELECT * FROM Category
WHERE Category.CategoryCD = [parameter]
```

Where `[parameter]` is the `product.CategoryCD` variable's value at the moment the `Select()` method is invoked.

A BQL statement can include several `Required` parameters. The number of `Required` parameters must match the number of parameters passed to the `Select()` function. See the example below.

```
// Suppose an event handler related to the DocTransaction DAC
// is being executed
DocTransaction line = (DocTransaction)e.Row;
...
Document doc =
    PXSelect<Document,
        Where<Document.docNbr, Equal<Required<DocTransaction.docNbr>,
            And<Document.docType, Equal<Required<DocTransaction.docType>>>>>
    .Select(this, line.DocNbr, line.DocType);
```

In this example, the BQL statement corresponds to the following SQL query.

```
SELECT * FROM Document
WHERE Document.DocNbr = [line.DocNbr value]
    AND Document.DocType = [line.DocType value]
```

The `Required` parameter can be used together with other parameter as follows.

```
// Suppose an event handler related to the DocTransaction DAC
// is being executed
DocTransaction line = (DocTransaction)e.Row;
...
SupplierProduct suppdata =
   PXSelect<SupplierProduct,
      Where<SupplierProduct.accountID, Equal<Current<Document.accountID>>,
         And<SupplierProduct.productID, Equal<Required<Product.productID>>>>>
   .Select(this, line.ProductID);
```

Here only one parameter is passed to the `Select()` method (excluding graph reference), because `Current` doesn't need an explicitly passed value.

Optional Parameter

The `Optional` parameter is used to pass field's "external value" to the SQL query. Parameter execution triggers the `FieldUpdating` event handlers, which can transform it to "internal value". The value is passed to the `Select()` method. If the value is not specified or is null, the default field value is used.

For example, suppose the `OrderDetail` DAC adds the `PXSelector` attribute to the `ProductID` field. `PXSelector` replaces it in the user interface (UI) with the human-readable `ProductCD` field.

```
[PXSelector(typeof(Search<Product.productID>),
            new Type [] {
                    typeof(Product.productCD),
                    typeof(Product.productName)
            },
            SubstituteKey = typeof(Product.productCD))]
public virtual int? ProductID { get; set; }
```

In the UI control for this field, the user inputs a `ProductCD` value. The `PXSelector` attribute implements the `FieldUpdating` event handler which replaces it with the corresponding `ProductID` value. The following code could be used to select `OrderDetail` records related to a `Product` record.

```
// Product data record obtained somehow
Product p = ...
// Selecting OrderDetail records - ProductCD value is passed
// to the Select() method.
PXSelect<OrderDetail,
    Where<OrderDetail.ProductID, Equal<Optional<OrderDetail.ProductID>>>>
        .Select(this, p.ProductCD);
```

If the `Required` parameter goes after an `Optional` parameter in a BQL command, the `Optional` parameter *has to* be provided with a value. The general rule is to provide values for all `Optional`, `Required`, and `Argument` parameters up to the last `Required` or `Argument` parameter in the BQL statement.

```
// Related OrderDetail and Product records obtained somehow
OrderDetail od = ...
Product p = ...

// At least three values (in addition to graph reference) must
// be passed to the Select() method below.
// The second Optional parameter here will be substituted with the
// default UnitPrice value.
PXResultSet<OrderDetail> details =
   PXSelect<OrderDetail,
      Where<OrderDetail.productID, Equal<Optional<OrderDetail.productID>>,
          And<OrderDetail.extPrice, Less<Required<OrderDetail.extPrice>>,
          And<OrderDetail.unitPrice, Greater<Required<OrderDetail.unitPrice>>,
          And<OrderDetail.taxRate, Equal<Optional<OrderDetail.taxRate>>>>>>>
      .Select(this, p.ProductCD, od.ExtPrice, od.UnitPrice);
```

The BQL statement in this example is translated into the following SQL query.

```
SELECT * FROM OrderDetail
WHERE OrderDetail.ProductID = [line.ProductID value or default]
    AND OrderDetail.ExtPrice < [line.ExtPrice value]
    AND OrderDetail.UnitPrice > [line.UnitPrice value]
    AND OrderDetail.TaxRate = [Default TaxRate value]
```

Argument Parameter

The `Argument` parameter is used to pass values from UI controls to the *optional method* of a data view. In this case, the optional method should have the parameters through which you can access the values passed from the UI. When a BQL statement with the `Argument` parameter is executed in code, the value must be specifed in the parameters of the `Select()` method.

In the `Argument` type parameter, you specify the data type of the expected value, as follows.

```
PXSelect<TreeViewItem,
    Where<TreeViewItem.parentID, Equal<Argument<int?>>>,
    OrderBy<Asc<TreeViewItem.parentID>>> GridDataSource;
```

The BQL statement from this example in translated into the following SQL query.

```
SELECT * FROM TreeViewItem
WHERE TreeViewItem.ParentID = [parameter]
ORDER BY TreeViewItem.ParentID
```

Where `[parameter]` will contain the value received from the UI control and passed to the `Select()` method.

Parameters

Parameters are used as operands in conditional expressions to pass values determined at run time into the resulting SQL.

Current<Field> : IBqlParameter

Inserts the field value from the `Current` property of the cache. If the `Current` property is null or the field value is null, the parameter is replaced by the default value.

Examples:

```
// Declaration of views in a BLC
PXSelect<Table1> MasterRecords;
PXSelect<Table2,
    Where<Table2.tableID, Equal<Current<Table1.tableID>>>> DetailRecords;
```

The second view corresponds to the following SQL query.

```
SELECT * FROM Table2
WHERE Table2.TableID = [value]
```

Where `[value]` is the `TableID` value from the `Current` property of the `PXCache<Table1>` object.

Type Parameters:

- `Field : IBqlField`

Current2<Field> : IBqlParameter

The same as `Current`, but in case the null value is passed to the parameter, doesn't insert the default value.

Type Parameters:

- `Field : IBqlField`

CurrentValue<Field> : IBqlOperand, IBqlCreator

Equivalent to the `Current` parameter, but is used in the `PXProjection` attribute.

Type Parameters:

- `Field : IBqlField`

Required<Field> : IBqlParameter

Is replaced by a value passed to the `Select()` method. The value type should match the type of the field specified as `Field`.

Examples:

```
PXResutset<Table> res =
    PXSelect<Table, Where<Table.field1, Equal<Required<Table.field1>>>>
        .Select(this, val);
```

The BQL statement in this example is translated into the following SQL query.

```
SELECT * FROM Table
WHERE Table.Field1 = [the val variable value]
```

Type Parameters:

- `Field : IBqlField`

Optional<Field> : IBqlParameter

Inserts the value from the `Current` property of the cache or the value explicitly passed to the `Select()` method. In the latter case, the parameter causes raising of the `FieldUpdating` event for the specified field (which can modify or substitute the value). If the `null` value is passed or the `Current` property is `null`, the default value of the field is inserted.

Examples:

```
PXResutset<Table1> res =
    PXSelect<Table1, Where<Table1.field1, Equal<Optional<Table2.field1>>>>
        .Select(this, val);
```

The view corresponds to the following SQL query:

```
SELECT * FROM Table1
WHERE Table1.Field1 = [value]
```

Where `[value]` is the value of the `val` variable, possibly, modified by `FieldUpdating` event handlers.

Type Parameters:

- `Field : IBqlField`

Optional2<Field> : IBqlParameter

The same as `Optional`, but in case the null value is passed to the parameter, doesn't insert the default value.

Type Parameters:

- `Field : IBqlField`

Argument<ArgumentType> : IBqlParameter

Is used to pass a value of a particular data type from a UI control to the associated view. When a BQL statement with `Argument` is executed in code, a value is passed in the `Select()` method's arguments.

Examples:

```
// Declaration of a view in a BLC
PXSelect<Table, Where<Table.field1, Greater<Argument<int?>>>> Records;
...
// Execution of the view in code
foreach(Table rec in Records.Select(5))
    ...
```

The BQL here is translated into the following SQL query.

```
SELECT * FROM Table
WHERE Table.Field1 > 5
```

Type Parameters:

- `ArgumentType : Type`

Using Functions

Functions are primaritly used in attributes to calculate a field from other fields. They can also be used as operands in `Where` and `OrderBy` clauses.

Arithmetic Operations

The following BQL classes implement arithmetic operations:

- `Add<Operand1, Operand2>` corresponds to `(Operand1 + Operand2)`.
- `Sub<Operand1, Operand2>` corresponds to `(Operand1 - Operand2)`.
- `Mult<Operand1, Operand2>` corresponds to `(Operand1 * Operand2)`.
- `Div<Operand1, Operand2>` corresponds to `(Operand1 / Operand2)`.
- `Minus<Operand>` corresponds to `-Operand`.

For example, product reorder discrepancy can be calculated using the following expression:

```
Minus<
    Sub<Sub<IsNull<Product.availQty, decimal_0>,
            IsNull<Product.bookedQty, decimal_0>>,
        Product.minAvailQty>>
```

Where the `decimal_0` *constant* represents the 0 decimal value. The expression is translated to the following SQL code:

```
 -((ISNULL(Product.AvailQty, .0) - ISNULL(Product.BookedQty, .0))
    - Product.MinAvailQty)
```

`IsNull` returns the first argument if it is not null or the second argument otherwise.

Such expression could be used in an attribute (for instance, `PXDBCacled`) to define a calculated field not bound to a database column:

```
// Data field definition in a DAC
[PXDecimal(2)]
[PXDBCalced(typeof(Minus<
                       Sub<Sub<IsNull<Product.availQty, decimal_0>,
                               IsNull<Product.bookedQty, decimal_0>>,
                           Product.minAvailQty>>),
            typeof(Decimal))]
public virtual decimal? Discrepancy { get; set; }
```

Also, it may be used in a conditional expression in a BQL statement like the following one.

```
PXSelect<Product,
    Where<Minus<
              Sub<Sub<IsNull<Product.availQty, decimal_0>,
                      IsNull<Product.bookedQty, decimal_0>>,
                  Product.minAvailQty>>,
          NotEqual<decimal_0>>>
```

This corresponding SQL query retrieves `Product` records that don't make the expression equal 0.

```
SELECT * FROM Product
WHERE -((ISNULL(Product.AvailQty, .0) - ISNULL(Product.BookedQty, .0))
        - Product.MinAvailQty) <> .0
```

Let us consider another example. Suppose an `OrderDetail` record represents sales order information for a single product. Then its total discount price may be calculated by the following formula.

```
Quantity * UnitPrice * (1 - DiscountRate/100)
```

This formula may be implemented in BQL as follows.

```
Mult<Mult<OrderDetail.unitPrice,
          OrderDetail.orderDetailQty>,
     Sub<decimal_1,
         Div<OrderDetail.discountRate,
             decimal_100>>>
```

Here, `decimal_1` and `decimal_100` are classes derived from `Constant<decimal>` and represent the 1 and 100 decimal values.

This expression could be written differently in BQL. For example, rounding the discount as shown below.

```
Sub<Mult<OrderDetail.unitPrice,
         OrderDetail.orderDetailQty>,
    Round<Div<Mult<Mult<OrderDetail.unitPrice,
                        OrderDetail.orderDetailQty>,
                   OrderDetail.discountRate>,
              decimal_100>,
          Minus<int_1>>>
```

The latter expression will be translated into the following SQL code.

```
((OrderDetail.UnitPrice * OrderDetail.OrderDetailQty)
  - ROUND(OrderDetail.UnitPrice *
          OrderDetail.OrderDetailQty *
          OrderDetail.DiscountRate / 100., -1))
```

Equivalents of SQL Functions

The BQL library defines the following SQL function equivalents:

- `IsNull<Operand1, Operand2>` corresponds to `ISNULL(Operand1, Operand2)`.
- `NullIf<Operand1, Operand2>` corresponds to `NULLIF(Operand1, Operand2)`.
- `Round<Operand1, Operand2>` corresponds to `ROUND(Operand1, Operand2)`.
- `Substring<Operand, Start, Length>` corresponds to `SUBSTRING(Operand, Start, Length)`.
- `Replace<Operand, toReplace, replaceWith>` corresponds to `REPLACE(Operand, toReplace, replaceWith)`.
- `DateDiff<Operand1, Operand2, OUM>` corresponds to `DATEDIFF(OUM, Operand1, Operand2)`.

Also, the BQL library defines the `Switch` class translated to SQL operator `CASE`.

Example – Conditional Ordering

The `Switch` clause can be used in `OrderBy` to sort data records according to a condition.

For example, data records with a specific field greater than another one can be placed above other data records. In this case, you should put the `Switch` clause inside `Asc` or `Desc` as in `OrderBy<Asc<Switch<...>>>`, as follows.

```
PXSelectOrderBy<Product,
   OrderBy<Asc<
       Switch<Case<Where<Product.availQty, Greater<Product.bookedQty>>, True>,
              False>>>>
```

This BQL statement is translated into the following SQL query.

```
SELECT * FROM Product
ORDER BY
    ( CASE
        WHEN Product.AvailQty > Product.BookedQty THEN 1
        ELSE 0
      END )
```

In the result set, the records with `AvailQty` values less or equal to `BookedQty` values will go first.

Arithmetic Operations

Arithmetic functions are used to construct arithmetic expressions out of fields, constants, and other functions.

Add<Operand1, Operand2> : IBqlOperand, IBqlCreator

Returns the sum of `Operand1` and `Operand2`.

Examples:

```
Add<Table.field1, Table.field2>
```

This is tranlsated into:

```
(Table.Field1 + Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

Sub<Operand1, Operand2> : IBqlOperand, IBqlCreator

Returns the substraction of `Operand2` from `Operand1`

Examples:

```
Sub<Table.field1, Table.field2>
```

This is tranlsated into:

```
(Table.Field1 - Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

Mult<Operand1, Operand2> : IBqlOperand, IBqlCreator

Returns the multiplication of `Operand1` by `Operand2`.

Examples:

```
Mult<Table.field1, Table.field2>
```

This is tranlsated into:

```
(Table.Field1 * Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

Div<Operand1, Operand2> : IBqlOperand, IBqlCreator

Return the division of `Operand1` on `Operand2`.

Examples:

```
Div<Table.field1, Table.field2>
```

This is tranlsated into:

```
(Table.Field1 / Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

Minus<Operand> : IBqlOperand, IBqlCreator

Returns -`Operand` (multiplies by -1).

Examples:

```
Minus<Table.field>
```

This is tranlsated into:

```
-Table.Field
```

Type Parameters:

- `Operand : IBqlOperand`

Common Functions

Common functions are translated to the equivalent SQL functions.

IsNull<Operand1, Operand2> : IBqlOperand, IBqlCreator

Returns `Operand1` if it is not null, or `Operand2` otherwise. Equivalent to SQL function `ISNULL`.

Examples:

```
IsNull<Table.field1, Table.field2>
```

This is tranlsated into:

```
ISNULL(Table.Field1, Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

Substring<Operand, Start, Length> : IBqlOperand, IBqlCreator

Returns the `Length` characters from the `Operand` string starting from the `Start` index (the first character has index 1). Equivalent to SQL function `SUBSTRING`.

To use constant numeric values in `Start` and `Length`, define the corresponding integer constants as classes derived from *Constant<int>*.

Examples:

```
Substring<Table.field, int_1, int_5>
```

Provided `int_1` and `int_5` are classes representing integer constants 1 and 5, this is tranlsated into:

```
SUBSTRING(Table.Field, 1, 5)
```

Type Parameters:

- `Operand : IBqlOperand`
- `Start : IBqlOperand`
- `Length : IBqlOperand`

Round<Operand1, Operand2> : IBqlOperand, IBqlCreator

Returns a numeric value rounded to the specified precision. Equivalent to SQL function `ROUND`.

Examples:

```
Round<Table.field1, Table.field2>
```

This is tranlsated into:

```
Round(Table.Field1, Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`

NullIf<Operand1, Operand2> : IBqlOperand, IBqlCreator

Returns null if `Operand1` equals `Operand2` and returns `Operand1` if the two expression are not equal. Equivalent to SQL function `NULLIF`.

Examples:

```
NullIf<Table.field1, Table.field2>
```

This is tranlsated into:

```
NULLIF(Table.Field1, Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`

- `Operand2 : IBqlOperand`

Replace<Operand, toReplace, replaceWith> : IBqlOperand, IBqlCreator

Replaces all occurrences of a string with another string in the source expression. Equivalent to SQL function `REPLACE`.

Examples:

```
Replace<Table.field, str_AAA, str_BBB>
```

Provided `str_AAA` and `str_BBB` are classes representing string constants "AAA" and "BBB", this is tranlsated into:

```
REPLACE(Table.Field, "AAA", "BBB")
```

Type Parameters:

- `Operand : IBqlOperand`
- `toReplace : IBqlOperand`
- `replaceWith : IBqlOperand`

DateDiff<Operand1, Operand2, UOM> : IBqlOperand, IBqlCreator

Returns the count of the datepart boundaries specified in `UOM` crossed between `Operand1` and `Operand2`. Equivalent to SQL function `DATEDIFF`.

Examples:

```
DateDiff<Table.field1, Table.field2, DateDiff.hour>
```

This is translated into:

```
DATEDIFF(hh, Table.Field1, Table.Field2)
```

Type Parameters:

- `Operand1 : IBqlOperand`
- `Operand2 : IBqlOperand`
- `UOM : Constant<string>, new()`

DateDiff

Wraps string constants that can be used as the third argument in the `DateDiff` function.

- `public class day : Constant<string>`

  Constant `dd`.
- `public class hour : Constant<string>`

  Constant `hh`.
- `public class minute : Constant<string>`

  Constant `mi`.
- `public class second : Constant<string>`

  Constant `ss`.
- `public class millisecond : Constant<string>`

Constant `ms`.

Switch Clause

The `Switch` clause returns one of the possible values depending on a condition.

Switch<Case> : IBqlOperand, IBqlCreator

Evaluates conditions and returns one of multiple possible values. Equivalent to SQL `CASE` expression without the `ELSE` expression. Pairs condition-value are specified via the `Case` clause.

The `Switch` clause can be used as an `Operand` type parameter in the `Where` or `OrderBy` clause.

Examples:

```
Switch<
    Case<Where<Table.field1, Less<Table.field2>>, Table.field3,
    Case<Where<Table.field1, Equal<Table.field2>>, Table.field4,
    Case<Where<Table.field1, Greater<Table.field2>>, Table.field5>>>>
```

This is translated into:

```
CASE
    WHEN Table.Field1 < Table.Field2 THEN Table.Field3
    WHEN Table.Field1 = Table.Field2 THEN Table.Field4
    WHEN Table.Field1 > Table.Field2 THEN Table.Field5
END
```

Type Parameters:

- `Case : IBqlCase, new()`

Switch<Case, Default> : IBqlOperand, IBqlCreator, ISwitch

Evaluates conditions and returns one of multiple possible values or the default value if none of the conditions is satisfied. Equivalent to SQL `CASE-ELSE` expression. Pairs condition-value are specified via the `Case` clause.

Examples:

```
Switch<
    Case<Where<Table.field1, Greater<Table.field2>,
             Or<Table.field2, IsNull>>, True>,
    False>
```

This is translated into:

```
CASE
    WHEN Table.Field1 > Table.Field2 OR Table.Field2 IS NULL THEN 1
    ELSE 0
END
```

Type Parameters:

- `Case : IBqlCase, new()`
- `Default : IBqlOperand`

Case<Where, Operand> : IBqlCase

Specifies a condition to evaluate in the `Switch` clause and the expression to return if the condition is satisfied.

The condition is set by the `Where` clause. In the tranlation to SQL, `Case` is replaced with `WHEN [conditions] THEN [expression]`.

Type Parameters:

- `Where : IBqlWhere, new()`
- `Operand : IBqlOperand`

Case<Where, Operand, NextCase> : IBqlCase

Specifies a single condition to evaluate and the expression to return if the condition is satisfied, and allows attaching more `Case` clauses.

Examples:

```
Switch<
    Case<Where<Table.field1, Equal<Table.field2>>, int0,
    Case<Where<Table.field1, Equal<Table.field3>>, int1>,
    int2>
```

Where `int0`, `int1`, and `int2` are derived from `Constant<int>` and represent the 0, 1, and 2 integers. The corresponding SQL code:

```
CASE
    WHEN Table.Field1 = Table.Field2 THEN 0
    WHEN Table.Field1 = Table.Field3 THEN 1
    ELSE 2
END
```

Type Parameters:

- `Where : IBqlWhere, new()`
- `Operand : IBqlOperand`
- `NextCase : IBqlCase, new()`

Executing Statements

To send a request to the database, you should call the `Select()` method of the `PXSelect` class. Additional parameters are provided if a BQL statement includes *parameters*. The `Select()` method returns the `PXResultset<>` object, which represents the result set.

The `Select()` method invokes the method of the underlying `PXView` object, which is responsible for further processing of the request. The `PXView` object translates the BQL statement into the SQL query, sends it to the database, and maintains the result set.

Different Ways of Executing a Statement

You use the `PXSelect` class or its variant to define a data view in one of the following ways:

- Declared as a member in a graph. Such data view can be specified as the data member of the webpage control and used for basic data manipulation (inserting a data record, updating a data records, and deleting a data record). You can execute the data view by calling the `Select()` method.
- Executed using the *static* `Select()` method. As the first parameter, you provide a graph object (typically, as the `this` variable).
- *Dynamically* instantiatated in code and executed using the `Select()` method. You provide the graph object as a parameter to the `PXSelect` constructor.

The following code example demonstrates different ways of BQL statement execution.

```
// Business logic controller (graph) declaration.
public class OrderDataEntry : PXGraph<OrderDataEntry, SalesOrder>
{
```

```
    // A data view declared as a graph member
    public PXSelectOrderBy<SalesOrder,
               OrderBy<Asc<SalesOrder.orderNbr>>> Orders;
    ...
    public void SomeMethod()
    {
        // An execution of the data view in code
        foreach(SalesOrder so in Orders.Select())
        {
            // The SalesOrder record selected by a data view can
            // be modified and updated through the Update() method
            so.OrderTotal = so.LinesTotal + so.FreightAmt;
            // Update the SalesOrder data record in the cache
            Orders.Update(so);
        }

        // Execution through the static Select() method
        foreach(SalesOrder so in
                PXSelectReadOnly3<SalesOrder,
                    OrderBy<Asc<SalesOrder.orderNbr>>>.Select(this))
            ...

        // Dynamic instantiation of a data view
        PXSelectBase<SalesOrder> orders =
            new PXSelectOrderBy<SalesOrder,
                    OrderBy<Asc<SalesOrder.orderNbr>>>(this);

        // An execution of a dynamically created BQL statement
        foreach(SalesOrder so in orders.Select())
            ...
    }
}
```

Note that the statically executed statement here is based on the `PXSelectReadOnly` class. Its result set will not reflect the changes to the `SalesOrder` records made three lines above. At the same time, `orders.Select()` will reflect the changes, because the result set will be merged with the cache.

Result Set Merging with Cache

If a BQL statement is not *read-only* and does not contain joins, the result set is merged with the appropriate `PXCache` object and the `Select()` method returns the merged result set.

If the BQL statement is not *read-only* and joins data from multiple tables, the result set is merged only with the `PXCache` object that corresponds to the first table of the BQL statement. The `PXResultset<>` object, which represents the result set, contains objects of generic `PXResult<>` type. This type can be cast to the DACs that represent the joined tables. The instance of the primary DAC to which the `PXResult<>` is cast will contain the modifications stored in the cache. Moreover, the casting will return the instance *from* the cache. On the other hand, casting `PXResult<>` to joined DACs will return the instances that contain values from the database and have no relation with the caches of the correponding DAC types.

A BQL statement is read-only if it uses aggregation or is based on the `PXSelectReadonly` class or its variant. For such statements, the result set is not merged with a `PXCache` object. The `Select()` method returns the data records as they are currently stored in the database.

Processing the Result Set

`Select()` returns the `PXResultset<>` object. The type parameter is set to the first table selected by the BQL statement.

You can iterate through the result set in a `foreach` loop, obtaining:

- DAC instances
- `PXResult<>` instances

A `PXResult<>` instance represents a whole result set record. It can be cast to any of the DAC types joined in the BQL statement.

In the following example, records are selected from one table.

```
// Result set records are implicitly casted to the Document DAC.
foreach(Document doc in PXSelect<Document>.Select(this))
{
    ...
}
```

The following example shows how to process a result set of a BQL statement joining two tables.

```
// The static Select() method is called to execute a BQL command.
PXResultset<OrderDetail> result =
    PXSelectJoin<OrderDetail, InnerJoin<SalesOrder,
        On<SalesOrder.orderNbr, Equal<OrderDetail.orderNbr>>>>.Select(this);

// Iterating over the result set.
// PXResult should be specialized with DACs of all joined tables
// to be able to cast to these DACs.
foreach(PXResult<OrderDetail, SalesOrder> record in result)
{
    // Casting a result set record to the OrderDetail DAC.
    OrderDetail detail = (OrderDetail)record;
    // Casting a result set record to the SalesOrder DAC.
    SalesOrder order = (SalesOrder)record;
    ...
}
```

Note that the `PXResult<>` type should be specialized with DACs of all joined tables. In the example above, the DACs are `OrderDetail` and `SalesOrder`.

The `detail` variable above references the `OrderDetail` instance located in the `OrderDetail` cache.

The `order` variable above references a `SalesOrder` instance that is initialized with the values from the database and is placed in the part of the memory that have no relation to the `SalesOrder` cache.

Executing Statements with Parameters

`Current`, `Optional`, and `Required` parameters are used to pass specific values to a BQL statement. The following example demonstrates their usage.

```
// Declaration of a BLC
public class ReceiptDataEntry : PXGraph<ReceiptDataEntry, Document>
{
    // When a screen associated with this BLC is first opened,
    // the Optional parameter will be replaced with the default DocType value.
    public PXSelect<Document,
        Where<Document.docType, Equal<Optional<Document.docType>>>> Receipts;

    // The Current parameters will be replaced with the values from
    // the PXCache<Document> object's Current property.
    public PXSelect<DocTransaction,
        Where<DocTransaction.docNbr, Equal<Current<Document.docNbr>>,
            And<DocTransaction.docType, Equal<Current<Document.docType>>>>,
        OrderBy<Asc<DocTransaction.lineNbr>>> ReceiptTransactions;

    public void SomeMethod()
    {
        // Select documents of the same DocType as the Current document
        // has or of the default DocType if Current is null.
        PXResult<Document> res1 = Receipts.Select();

        // Select documents of the "N" DocType
        PXResult<Document> res2 = Receipts.Select("N");
```

```
        // Parameter values are taken from the Current document
        PXResult<DocTransaction> res3 = ReceiptTransactions.Select();

        // Use the Required parameter to provide values in code.
        // The result set here is the same as res2.
        PXResult<Document> res4 =
            PXSelect<Document,
                Where<Document.docType, Equal<Required<Document.docType>>>>
            .Select(this, "N");
    }
    ...
}
```

For more details on parameters usage in BQL statement, see *Using Parameters*.

More Methods

Using other methods of the *PXSelectBase* class you can select a specific number of records, append additional conditions to the `Where` clause, join more tables, and redefine ordering.

Implementing Optional Select Method

In some cases the data requested from the database cannot be described by a declarative BQL statement. In this case you can implement the *optional method* that will be used instead of the standard `Select()` logic to retrieve data from the database. The data request will still be executed via the `Select()` method, but his will result in the optional method invocation.

If the optional method is not defined or returns null, the standard `Select()` logic will be executed.

The optional method of a data view should have the same name as the data view except for the first letter, which must have a different case. The optional method should return `IEnumerable`, as shown in the following example.

```
// A view declaration in a graph.
public PXSelectJoin<BalancedAPDocument,
           LeftJoin<APInvoice,
               On<APInvoice.docType, Equal<BalancedAPDocument.docType>,
               And<APInvoice.refNbr, Equal<BalancedAPDocument.refNbr>>>,
           LeftJoin<APPayment,
               On<APPayment.docType, Equal<BalancedAPDocument.docType>,
               And<APPayment.refNbr, Equal<BalancedAPDocument.refNbr>>>>>>
           DocumentList;

// The optional method executed on DocumentList.Select().
protected virtual IEnumerable apdocumentlist()
{
    // An empty result set is created.
    // All DAC types that will be included in the set must be specified.
    PXResultset<BalancedAPDocument, APInvoice, APPayment>
        ret = new PXResultset<BalancedAPDocument, APInvoice, APPayment>();

    // Iterating over the result set of a complex BQL statement.
    foreach (PXResult<BalancedAPDocument, APInvoice, APPayment, APAdjust> res in
        PXSelectJoinGroupBy<BalancedAPDocument,
            LeftJoin<APInvoice,
                On<APInvoice.docType, Equal<BalancedAPDocument.docType>,
                And<APInvoice.refNbr, Equal<BalancedAPDocument.refNbr>>>,
            LeftJoin<APPayment,
                On<APPayment.docType, Equal<BalancedAPDocument.docType>,
                And<APPayment.refNbr, Equal<BalancedAPDocument.refNbr>>>,
            LeftJoin<APAdjust,
                On<APAdjust.adjgDocType,Equal<BalancedAPDocument.docType>>>>>,
            Aggregate<GroupBy<BalancedAPDocument.docType,
                GroupBy<BalancedAPDocument.refNbr,
                GroupBy<BalancedAPDocument.released,
```

```
                 GroupBy<BalancedAPDocument.prebooked,
                 GroupBy<BalancedAPDocument.openDoc>>>>>>>.Select(this))
    {
        // Casting a result set record to DAC types.
        BalancedAPDocument apdoc = (BalancedAPDocument)res;
        APAdjust adj = (APAdjust)res;
        // Checking some conditions and modifying records
        ...
        // Adding a record to the result set.
        // PXResult record passed as a parameter to the constructor is
        // automatically casted to the appropriate type.
        ret.Add(new PXResult<BalancedAPDocument, APInvoice, APPayment>(
            apdoc, res, res));
    }
    return ret;
}
```

In this example, the `apdocumentlist()` method creates an empty result set. The `PXResultSet` type in this case should be parametrized with all DAC types that will be wrapped in a result set record. The `apdocumentlist()` method then executes a complex SQL query with aggregation, processes the result set and constructs records for the output result set.

A record is added to the `PXResultset` object via the `Add()` method. Note that you can pass a `PXResult` object as a parameter to the `PXResult` contructor. The `PXResult` object will be implicitly casted to the appropriate DAC type (here, `APInvoice` and `APPayment`).

Appendix

This chapter provides reference information for the following BQL API components:

- *Search Classes*
- *Select Classes*

Search Classes

The `Search` classes are used for specifying BQL statements in such attributes as `PXSelector`, `PXDBScalar`, and `PXDefault`. A `Search` statement selects a value of a particular field rather than a whole record. The field is specified as the first type parameter instead of the table. Apart from this, the syntax of BQL statements based on `Search` and `PXSelect` classes is identical.

In the example below, the `PXDBScalar` attribute will add a subrequest into SQL queries that request `SomeField`.

```
// Declaration of a field in the DAC representing Table1.
// SomeField will be assigned a value retrieved from Table2.
[PXDecimal(2)]
[PXDBScalar(typeof(
    Search<Table2.someField,
        Where<Table2.field2, Equal<Table1.field1>>>))]
public virtual decimal? SomeField { get; set; }
```

For more details on attributes and examples, see *Attributes Reference*.

Search<Field> : BqlCommand, IBqlSearch

Retrieves a field value.

Type Parameters:

- `Field : IBqlField`

Search<Field, Where> : BqlCommand, IBqlSearch

Retrieves a field value, applying filtering.

Type Parameters:

- `Field : IBqlField`
- `Where : IBqlWhere, new()`

Search<Field, Where, OrderBy> : BqlCommand, IBqlSearch

Retrieves a field from a table, applying filtering and ordering.

Type Parameters:

- `Field : IBqlField`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

Search2<Field, Join> : BqlCommand, IBqlSearch, IBqlJoinedSelect

Retrieves a field from a table joined with other tables.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`

Search2<Field, Join, Where> : BqlCommand, IBqlSearch, IBqlJoinedSelect

Retrieves a field from a table joined with other tables, applying filtering.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`

Search2<Field, Join, Where, OrderBy> : BqlCommand, IBqlSearch, IBqlJoinedSelect

Retrieves a field from a table joined with other tables, applying filtering and ordering.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

Search3<Field, OrderBy> : BqlCommand, IBqlSearch

Retrieves a field value, applying ordering.

Type Parameters:

- `Field : IBqlField`
- `OrderBy : IBqlOrderBy, new()`

Search3<Field, Join, OrderBy> : BqlCommand, IBqlSearch, IBqlJoinedSelect

Retrieves a field value from a table joined with other tables, applying ordering.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`
- `OrderBy : IBqlOrderBy, new()`

Search4<Field, Aggregate> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieves an aggregated field value.

Type Parameters:

- `Field : IBqlField`
- `Aggregate : IBqlAggregate, new()`

Search4<Field, Where, Aggregate> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieve an aggregated field value, applying filtering.

Type Parameters:

- `Field : IBqlField`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`

Search4<Field, Where, Aggregate, OrderBy> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieves an aggregated field value, applying filtering and ordering.

Type Parameters:

- `Field : IBqlField`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Search5<Field, Join, Aggregate> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieves an aggregated field value from one table joined with other tables.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`
- `Aggregate : IBqlAggregate, new()`

Search5<Field, Join, Where, Aggregate> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieves an aggregated field value from one table joined with other tables, applying filtering.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`

Search5<Field, Join, Where, Aggregate, OrderBy> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieves an aggregated field value from one table joined with other tables, applying filtering and ordering.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Search6<Field, Aggregate, OrderBy> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieves an aggregated field value, applying ordering.

Type Parameters:

- `Field : IBqlField`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Search6<Field, Join, Aggregate, OrderBy> : BqlCommand, IBqlSearch, IBqlAggregate

Retrieves an aggregated field value from one table joined with other tables, applying ordering.

Type Parameters:

- `Field : IBqlField`
- `Join : IBqlJoin, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Coalesce<Search1, Search2> : BqlCommand, IBqlSearch, IBqlCoalesce

Retrieves a value using `Search1` or, if it returns null, `Search2`.

Type Parameters:

- `Search1 : IBqlSearch, new()`
- `Search2 : IBqlSearch, new()`

Select Classes

The `Select` classes represent BQL commands and are primarily passed to `PXView` objects, which execute the BQL command. However, to select data from the database, you use one of the `PXSelect` classes, which initializes the `Select` object and passes it to the `PXView` object for you.

The `Select` and `PXSelect` BQL statements syntax is identical, only the names of the classes themselves are different. For example, the `PXSelectJoinOrderBy<Table, Join, OrderBy>` type initializes the object of `Select3<Table, Join, OrderBy>` type.

The `Select` classes are also used for specifying BQL statements in such attributes as `PXParent` and `PXProjection`.

For more details on attributes and examples, see *Attributes Reference*.

Select<Table> : BqlCommand, IBqlSelect

Selects data records from a single table.

Type Parameters:

- `Table : IBqlTable`

Select<Table, Where> : BqlCommand, IBqlSelect

Selects data records from a single table with filtering.

Type Parameters:

- `Table : IBqlTable`
- `Where : IBqlWhere, new()`

Select<Table, Where, OrderBy> : BqlCommand, IBqlSelect

Selects data records from a single table with filtering and ordering.

Type Parameters:

- `Table : IBqlTable`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

Select2<Table, Join> : BqlCommand, IBqlSelect, IBqlJoinedSelect

Selects data records from multiple tables.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`

Select2<Table, Join, Where> : BqlCommand, IBqlSelect, IBqlJoinedSelect

Selects data records from multiple tables with filtering.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`

Select2<Table, Join, Where, OrderBy> : BqlCommand, IBqlSelect, IBqlJoinedSelect

Selects data records from multiple tables with filtering and ordering.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `OrderBy : IBqlOrderBy, new()`

Select3<Table, OrderBy> : BqlCommand, IBqlSelect

Selects data records from a single table with ordering.

Type Parameters:

- `Table : IBqlTable`
- `OrderBy : IBqlOrderBy, new()`

Select3<Table, Join, OrderBy> : BqlCommand, IBqlSelect, IBqlJoinedSelect

Selects data records from multiple tables with ordering.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`
- `OrderBy : IBqlOrderBy, new()`

Select4<Table, Aggregate> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from a single table.

Type Parameters:

- `Table : IBqlTable`
- `Aggregate : IBqlAggregate, new()`

Select4<Table, Where, Aggregate> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from a single table with filtering.

Type Parameters:

- `Table : IBqlTable`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`

Select4<Table, Where, Aggregate, OrderBy> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from a single table with filtering and ordering.

Type Parameters:

- `Table : IBqlTable`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Select5<Table, Join, Aggregate> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from multiple tables.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`
- `Aggregate : IBqlAggregate, new()`

Select5<Table, Join, Where, Aggregate> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from multiple tables with filtering.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`

Select5<Table, Join, Where, Aggregate, OrderBy> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from multiple tables with filtering and ordering.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`
- `Where : IBqlWhere, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Select6<Table, Aggregate, OrderBy> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from a single table with ordering.

Type Parameters:

- `Table : IBqlTable`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Select6<Table, Join, Aggregate, OrderBy> : BqlCommand, IBqlSelect, IBqlAggregate

Selects aggregated values from multiple tables with ordering.

Type Parameters:

- `Table : IBqlTable`
- `Join : IBqlJoin, new()`
- `Aggregate : IBqlAggregate, new()`
- `OrderBy : IBqlOrderBy, new()`

Core Classes

The developer of Acumatica Framework applications deals most of the time with the following classes that form the core of the framework:

- The *PXCache<>* class represents the cache and the controller of modified data records from a particular database table.
- The *PXSelect<>* and related classes define a data view for retrieving a particular data set from the database.

- The successors of the *PXGraph* class are the base types for business logic controllers (graphs). In a graph, the application defines data views, actions, and event handlers.
- The *PXView* class is instantiated to execute a data view. The objects of this type are handled mostly internally.

PXCache<Table> Class

Represents the cache of modified data records from a paricular table and the controller for basic operations over these data records. The type parameter is set to the data access class (DAC) that represents this table.

The cache objects consists conceptually of two parts:

- The collections of the data records that were modified and not yet saved to the database, such as Updated, Inserted, Deleted, and Dirty. See *Properties* for description of these items.
- The controller that executes basic data-related operations through the use of the methods, such as *Update()*, *Insert()*, *Delete()*, *Persist()*, and other *methods*.

During execution of these methods, the cache raises events. The graph and attributes can subscribe to these events to implement business logic. The methods applied to a previously unchanged data record result in placing of the data record into the cache.

See *Remarks* for more details.

Inheritance Hierarchy

```
PXCache
```

Syntax

```
[System.Security.Permissions.ReflectionPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
[System.Security.Permissions.SecurityPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
[DebuggerTypeProxy(typeof(PXCache<>.PXCacheDebugView))]
public class PXCache<TNode> : PXCache
    where TNode : class, IBqlTable, new()
```

The PXCache<Table> type exposes the following members.

Constructors

The application does not need to instantiate PXCache directly, as the system creates caches automatically whenever they are needed. A cache instance is always bound to an instance of the business logic controller (graph). The application typically accesses a cache instance through the Cache property of a data view. The property always returns the valid cache instance, even if it didn't exist before the property was accessed. A cache instance is also available through the Caches property of the graph to which the cache instance is bound.

Properties

- public virtual bool **AllowDelete**

  Gets or sets the value that indicates whether the cache allows deletion of data records from the user interface. This value does not affect the ability to delete a data record via the methods. By default, the property equals true.

- public virtual bool **AllowInsert**

Gets or sets the value that indicates whether the cache allows insertion of data records from the user interface. This value does not affect the ability to insert a data record via the methods. By default, the property equals `true`.

- `public virtual bool` **`AllowSelect`**

  Get, set. By default, the property equals `true`.

- `public virtual bool` **`AllowUpdate`**

  Gets or sets the value that indicates whether the cache allows update of data records from the user interface. This value does not affect the ability to update a data record via the methods. By default, the property equals `true`.

- `public override object` **`Current`**

  Gets or sets the current data record. This property points to the last data record displayed in the user interface. If the user selects a data record in a grid, this property points to this data record. If the user or the application inserts, updates, or deletes a data record, the property points to this data record. Assigning this property raises the `RowSelected` event.

  You can reference the `Current` data record and its fields in the `PXSelect` BQL statements by using the `Current` parameter.

- `public virtual PXGraph` **`Graph`**

  Gets or sets the business logic controller the cache is related to.

- `public override IEnumerable` **`Dirty`**

  Gets the collection of updated, inserted, and deleted data records. The collection contains data records with the `Updated`, `Inserted`, or `Deleted` status.

- `public override IEnumerable` **`Updated`**

  Gets the collection of updated data records that exist in the database. The collection contains data records with the `Updated` status.

- `public override IEnumerable` **`Inserted`**

  Gets the collection of inserted data records that does not exist in the database. The collection contains data records with the `Inserted` status.

- `public override IEnumerable` **`Deleted`**

  Gets the collection of deleted data records that exist in the database. The collection contains data records with the `Deleted` status.

- `public override IEnumerable` **`Cached`**

  Get the collection of all cached data records. The collection contains data records with any status. The developer should not rely on the presense of data records with statuses other than `Updated`, `Inserted`, and `Deleted` in this collection.

- `public override bool` **`IsInsertedUpdatedDeleted`**

  Gets the value that indicates if the cache contains modified data records to be saved to database.

- `public virtual bool` **`IsDirty`**

  Gets or sets the value that indicates whether the cache contains the modified data records.

- `public override PXFieldCollection` **`Fields`**

  Gets the collection of names of fields and virtual fields. By default, the collection includes all public properties of the DAC that is associated with the cache. The collection may also include the virtual fields that are injected by attributes (such as the description field of the *PXSelector* attribute). The developer can add any field to the collection.

- `public virtual List<string>` **`AlteredFields`**

  Gets the collection of field names. Placing the field name in this collection forces calculation of the `PXFieldState` object in the *GetValueExt<>()* method.

- `public virtual List<string>` **`Keys`**

  Gets the list of the key fied names (that form the identity of a data record). The collection contains the fields that have the `IsKey` property set to `true` in the attribute that specifies the field data type.

- `public virtual string` **`Identity`**

  Gets the name of the identity field if the DAC defines it.

- `public override List<Type>` **`BqlFields`**

  Gets the list of classes that implement `IBqlField` and are nested in the DAC and its base type. These types represent DAC fields in BQL queries. This list differs from the list that the `Fields` property returns.

- `public override List<Type>` **`BqlKeys`**

  Gets the collection of BQL types that correspond to the key fields which the DAC defines.

- `public override Type` **`BqlTable`**

  Gets the DAC the cache is associated with. The DAC is specified through the type parameter when the cache is instantiated.

- `public string` **`DisplayName`**

  Gets or sets the user-friendly name set via the *PXCacheName* attribute.

Methods

Method	Description
Clear()	Clears the cache from all data
ClearQueryCache()	Clears the internal cache of database query results
CreateCopy(Table)	Initializes a new data record with the field values got from the provided data record
CreateCopy(object)	Creates a clone of the provided data record by initializing a new data record with the field values get from the provided data record
CreateInstance()	Returns a new data record of the DAC type of the cache
Delete(object)	Places the data record into the cache with the `Deleted` or `InsertedDeleted` status
Delete(IDictionary, IDictionary)	Initializes the data record with the provided key values and places it into the cache with the `Deleted` or `InsertedDeleted` status
Extend<Parent>(Parent)	Initializes a data record of the DAC type of the cache from the provided data record of the base DAC type and inserts the new data record into the cache
FromXml(string)	Initializes the data record from the provided XML string

Method	Description
GetAttributes(string)	Returns the cach-level instances of attributes placed on the specified field and all item-level instances currently stored in the cache
GetAttributes(object, string)	Returns the item-level instances of attributes placed on the specified field
GetAttributes<Field>()	Returns the cach-level instances of attributes placed on the specified field and all item-level instances currently stored in the cache
GetAttributes<Field>(object)	Returns the item-level instances of attributes placed on the specified field
GetAttributesReadonly(string)	Returns the cache-level instances of attributes placed on the specified field in the DAC
GetAttributesReadonly(string, bool)	Returns the cache-level instances of attributes placed on the specified field in the DAC
GetAttributesReadonly(object, string)	Returns the item-level attribute instances placed on the specified field if such instances exist for the provided data record or the cache-level instances otherwise
GetAttributesReadonly<Field>()	Returns the cache-level instances of attributes placed on the specified field in the DAC
GetAttributesReadonly<Field>(object)	Returns the item-level instances of attributes placed on the specified field if such instances exist for the provided data record or the cache-level instances otherwise
GetBqlField(string)	Gets the type that represents the field with the provided name in BQL expressions
GetBqlTable(Type)	Gets the base DAC type by which the provided DAC type is bound to the database
GetExtension<Extension>(object)	Gets the instance of the DAC extension of the specified type
GetField(Type)	Searches the `Fields` collection for the name of the specified type
GetFieldCount()	Returns the number of fields and virtual fields which comprise the `Fields` collection
GetFieldOrdinal(string)	Returns the index of the specified field in the internally kept fields map
GetFieldOrdinal<Field>()	Returns the index of the specified field in the internally kept fields map
GetItemType()	Returns the DAC type of the data records in the cache
GetObjectHashCode(object)	Returns the hash code generated from key field values
GetStateExt(object, string)	Gets the `PXFieldState` object of the specified field in the given data record

Method	Description
GetStateExt<Field>(object)	Gets the `PXFieldState` object of the specified field in the given data record
GetStatus(object)	Returns the status of the provided data record
GetValue(object, int)	Returns the value of the specified field in the given data record without raising any events
GetValue(object, string)	Returns the value of the specified field in the given data record without raising any events
GetValue<Field>(object)	Returns the value of the specified field in the given data record without raising any events
GetValueExt(object, string)	Returns the value or the `PXFieldState` object of the specified field in the given data record
GetValueExt<Field>(object)	Gets either the value or `PXFieldState` object of the specified field in the given data record
GetValueOriginal(object, string)	Returns the value of the specified field for the data record as it is stored in the database
GetValueOriginal<Field>(object)	Returns the value of the specified field for the data record as it is stored in the database
GetValuePending(object, string)	Returns the value of the field from the provided data record when the data record's update or insertion is in process
GetValuePending<Field>(object)	Returns the value of the field from the provided data record when the data record's update or insertion is in process
HasAttributes(object)	Checks if the provided data record has any attributes attached to its fields
Insert()	Initializes a new data record with default values and inserts it into the cache by invoking the *Insert(object)* method
Insert(object)	Inserts the provided data record into the cache
Insert(IDictionary)	Initializes a new data record using the provided field values and inserts the data record into the cache
Load()	Loads dirty items and other cache state objects from the session
Locate(object)	Searches the cache for a data record that has the same key fields as the provided data record
Locate(IDictionary)	Searches the cache for a data record that has the same key fields as in the provided dictionary
Normalize()	Recalculates internally stored hash codes
ObjectToString(object)	Returns a string of key fields and their values in the *{key1=value1, key2=value2}* format
ObjectsEqual(object, object)	Compares two data records by the key fields
ObjectsEqual<Field1>(object, object)	Compares two data records by the field value

Method	Description
ObjectsEqual<Field1, Field2>(object, object)	Compares two data records by the values of the specified fields
ObjectsEqual<Field1, Field2, Field3>(object, object)	Compares two data records by the values of the specified fields
ObjectsEqual<Field1, Field2, Field3, Field4>(object, object)	Compares two data records by the values of the specified fields
ObjectsEqual<Field1, Field2, Field3, Field4, Field5>(object, object)	Compares two data records by the values of the specified fields
ObjectsEqual<Field1, Field2, Field3, Field4, Field5, Field6>(object, object)	Compares two data records by the values of the specified fields
ObjectsEqual<Field1, Field2, Field3, Field4, Field5, Field6, Field7>(object, object)	Compares two data records by the values of the specified fields
ObjectsEqual<Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8>(object, object)	Compares two data records by the values of the specified fields
Persist(PXDBOperation)	Saves the modifications of a particular type from the cache to the database
Persist(object, PXDBOperation)	Saves the modification of the specified type from the cache to the database for a particular data record
PersistDeleted(object)	Deletes the provided data record from the database by the key fields
PersistInserted(object)	Inserts the provided data record into the database
PersistUpdated(object)	Updates the provided data record in the database
Persisted(bool)	Completes saving changes to the database by raising the `RowPersisted` event for all persisted data records
RaiseCommandPreparing(string, object, object, PXDBOperation, Type, out)	Raises the `CommandPreparing` event for the specified field and data record
RaiseCommandPreparing<Field>(object, object, PXDBOperation, Type, out)	Raises the `CommandPreparing` event for the specified field and data record
RaiseExceptionHandling(string, object, object, Exception)	Raises the `ExceptionHandling` event for the specified field and data record
RaiseExceptionHandling<Field>(object, object, Exception)	Raises the `ExceptionHandling` event for the specified field and data record
RaiseFieldDefaulting(string, object, out)	Raises the `FieldDefaulting` event for the specified field and data record
RaiseFieldDefaulting<Field>(object, out)	Raises the `FieldDefaulting` event for the specified field and data record
RaiseFieldSelecting(string, object, ref, bool)	Raises the `FieldSelecting` event for the specified field and data record
RaiseFieldSelecting<Field>(object, ref, bool)	Raises the `FieldSelecting` event for the specified field and data record
RaiseFieldUpdated(string, object, object)	Raises the `FieldUpdated` event for the specified field and data record

Method	Description
RaiseFieldUpdated<Field>(object, object)	Raises the `FieldUpdated` event for the specified field and data record
RaiseFieldUpdating(string, object, ref)	Raises the `FieldUpdating` event for the specified field and data record
RaiseFieldUpdating<Field>(object, ref)	Raises the `FieldUpdating` event for the specified field and data record
RaiseFieldVerifying(string, object, ref)	Raises the `FieldVerifying` event for the specified field and data record
RaiseFieldVerifying<Field>(object, ref)	Raises the `FieldVerifying` event for the specified field and data record
RaiseRowDeleted(object)	Raises the `RowDeleted` event for the specified data record
RaiseRowDeleting(object)	Raises the `RowDeleting` event for the specified data record
RaiseRowInserted(object)	Raises the `RowInserted` event for the specified data record
RaiseRowInserting(object)	Raises the `RowInserting` event for the specified data record
RaiseRowPersisted(object, PXDBOperation, PXTranStatus, Exception)	Raises the `RowPersisted` event for the specified data record
RaiseRowPersisting(object, PXDBOperation)	Raises the `RowPersisting` event for the specified data record
RaiseRowSelected(object)	Raises the `RowSelected` event for the specified data record
RaiseRowSelecting(object, PXDataRecord, ref int, bool)	Raises the `RowSelecting` event for the specified data record
RaiseRowUpdated(object, object)	Raises the `RowUpdated` event for the specified data record
RaiseRowUpdating(object, object)	Raises the `RowUpdating` event for the specified data record
Remove(object)	Completely removes the provided data record from the cache without raising any events
RestoreCopy(object, object)	Copies values of all fields from the second data record to the first data record
RestoreCopy(Table, Table)	Copies values of all fields from the second data record to the first data record
Select(PXDataRecord, ref int, bool, out bool)	Creates a data record from the `PXDataRecord` object and places it into the cache with the `NotChanged` status if the data record isn't found among the modified data records in the cache
SetAltered(string, bool)	Adds the field to the `AlteredFields` list or removes it from this list

Method	Description
SetAltered<Field>(bool)	Adds the field to the `AlteredFields` list or removes it from this list
SetDefaultExt(object, string)	Sets the default value to the field in the provided data record
SetDefaultExt<Field>(object)	Sets the default value to the field in the provided data record
SetStatus(object, PXEntryStatus)	Sets the status to the provided data record
SetValue(object, int, object)	Sets the value of the field in the provided data record without raising events
SetValue(object, string, object)	Sets the value of the field in the provided data record without raising events
SetValue<Field>(object, object)	Sets the value of the field in the provided data record without raising events
SetValueExt(object, string, object)	Sets the value of the field in the provided data record
SetValueExt<Field>(object, object)	Sets the value of the field in the provided data record
SetValuePending(object, string, object)	Sets the value of the field in the provided data record when the data record's update or insertion is in process and the field possibly hasn't been updated in the cache yet
SetValuePending<Field>(object, object)	Sets the value of the field in the provided data record when the data record's update or insertion is in process and the field possibly hasn't been updated in the cache yet
ToDictionary(object)	Converts the provided data record to the dictionary of field names and field values
ToString()	Returns the string representing the current cache object
ToXml(object)	Returns the XML string representing the provided data record
Unload()	Serializes the cache to the session
Update(object)	Updates the provided data record in the cache
Update(IDictionary, IDictionary)	Updates the data record in the cache with the provided values
ValueFromString(string, string)	Converts the provided value of the field from a string to the appropriate type and returns the resulting value
ValueToString(string, object)	Converts the provided value of the field to string and returns the resulting value

Remarks

The system creates and destroys `PXCache` instances (caches) on each request. If the user or the code modifies a data record, it is placed into the cache. When request execution is completed, the system serializes the modified records from the caches to the session. At run time, the cache may also include the unchanged data records retrieved during request execution. These data records are discarded once the request is served.

On the next round trip, the modified data records are loaded from the session to the caches. The cache merges the data retrieved from the database with the modified data, and the application accesses the data as if the entire data set has been preserved from the time of previous request.

The cache maintains the modified data until the changes are discarded or saved to the database.

The cache is the issuer of all data-related events, which can be handled by the graph and attributes.

PXCache<Table> Methods

The *PXCache<Table>* type exposes the following methods.

Clear()

Clears the cache from all data.

Syntax:

```
public override void Clear()
```

Examples:

The code below clears the cache of the `POReceipt` data records.

```
// Declaration of a data view in a graph
public PXSelect<POReceipt> poreceiptslist;
...
// Clearing the cache of POReceipt data records
poreceiptslist.Cache.Clear();
```

ClearQueryCache()

Clears the internal cache of database query results.

Syntax:

```
public override void ClearQueryCache()
```

CreateCopy(Table)

Initializes a new data record with the field values from the provided data record.

Syntax:

```
public static Table CreateCopy(Table item)
```

Parameters:

- `item`

  The data record to copy.

Examples:

The code below creates a copy of the `Current` data record of a data view.

```
public PXSelect<APInvoice, ... > Document;
...
APInvoice newdoc = PXCache<APInvoice>.CreateCopy(Document.Current);
```

CreateCopy(object)

Creates a clone of the provided data record by initializing a new data record with the field values get from the provided data record.

Syntax:

```
public override object CreateCopy(object item)
```

Parameters:

- `item`

  The data record to copy.

CreateInstance()

Returns a new data record of the DAC type of the cache. The method may be used to initialize a data record of the type appropriate for the `PXCache` instance when its DAC type is unknown.

Syntax:

```
public override object CreateInstance()
```

Delete(object)

Places the data record into the cache with the `Deleted` or `InsertedDeleted` status. The method assigns the `InsertedDeleted` status to the data record if it has the `Inserted` status when the method is invoked.

The method raises the `RowDeleting` and `RowDeleted` events. See *Deleting a Data Record* for the events flowchart.

The `AllowDelete` property does not affect this method.

Syntax:

```
public override object Delete(object data)
```

Parameters:

- `data`

  The data record to delete.

Examples:

The code below deletes an `APInvoice` data record.

```
APInvoice item = ...
Documents.Cache.Delete(item);
```

The second line above is equivalent to the following line.

```
Documents.Delete(item);
```

Delete(IDictionary, IDictionary)

Initializes the data record with the provided key values and places it into the cache with the `Deleted` or `InsertedDeleted` status. The method assigns the `InsertedDeleted` status to the data record if it has the `Inserted` status when the method is invoked.

The method raises the following events: `FieldUpdating`, `FieldUpdated`, `RowDeleting`, and `RowDeleted` events. See *Deleting a Data Record* for the events flowchart.

This method is typically used to process deletion initiated from the user interface. If the `AllowDelete` property is `false`, the data record is not marked deleted and the method returns 0. The method returns 1 if the data record is successfully marked deleted.

Syntax:

```
public override int Delete(IDictionary keys, IDictionary values)
```

Parameters:

- `keys`

  The values of key fields.

- `values`

  The values of all fields. The parameter is not used in the method.

Extend<Parent>(Parent)

Initializes a data record of the DAC type of the cache from the provided data record of the base DAC type and inserts the new data record into the cache. Returns the inserted data record.

Syntax:

```
public override object Extend<Parent>(Parent item)
```

The DAC type of the cache should derive from the `Parent` DAC.

Parameters:

- `item`

  The data record of the base DAC type which field values are used to initialize the data record.

Examples:

See the *Extend<Parent>(Parent)* method of the `PXSelectBase<>` class.

FromXml(string)

Initializes the data record from the provided XML string.

The data record is represented in the XML by the *<Row>* element with the *type* attribute set to the DAC name. Each field is represented by the *<Field>* element with the *name* attribute holding the field name and the *value* attribute holding the field value.

Syntax:

```
public override object FromXml(string xml)
```

Parameters:

- `xml`

  The XML string to parse.

GetAttributes(string)

Returns the cach-level instances of attributes placed on the specified field and all item-level instances currently stored in the cache.

Syntax:

```
public override List<PXEventSubscriberAttribute> GetAttributes(string name)
```

Parameters:

- `name`

  The name of the field whose attributes are returned. If `null`, the method returns attributes from all fields.

GetAttributes(object, string)

Returns the item-level instances of attributes placed on the specified field. If such instances are not exist for the provided data record, the method creates them by copying all cache-level attributes and storing them in the internal collection that contains the data record specific attributes. To avoid cloning cache-level attributes, use the *GetAttributesReadonly(object, string)* method.

Syntax:

```
public override List<PXEventSubscriberAttribute>
    GetAttributes(object data, string name)
```

Parameters:

- `data`

  The data record.

- `name`

  The name of the field whose attributes are returned. If `null`, the method returns attributes from all fields.

GetAttributes<Field>()

Returns the cach-level instances of attributes placed on the specified field and all item-level instances currently stored in the cache. The field is specified as the type parameter.

Syntax:

```
public List<PXEventSubscriberAttribute> GetAttributes<Field>()
    where Field : IBqlField
```

GetAttributes<Field>(object)

Returns the item-level instances of attributes placed on the specified field. If such instances are not exist for the provided data record, the method creates them by copying all cache-level attributes and storing them in the internal collection that contains the data record specific attributes. To avoid cloning cache-level attributes, use the *GetAttributesReadonly(object, string)* method. The field is specified as the type parameter.

Syntax:

```
public List<PXEventSubscriberAttribute> GetAttributes<Field>(object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.

Examples:

```
foreach (PXEventSubscriberAttribute attr in sender.GetAttributes<Field>(data))
{
    if (attr is PXUIFieldAttribute)
    {
```

```
        // Doing something
    }
}
```

GetAttributesReadonly(string)

Returns the cache-level instances of attributes placed on the specified field in the DAC.

Syntax:

```
public override List<PXEventSubscriberAttribute> GetAttributesReadonly(
    string name)
```

Parameters:

- `name`

  The name of the field whose attributes are returned. If `null`, the method returns attributes from all fields.

Remarks:

The system maintains instances of attributes on three different levels. On its instantiation, a cache object copies appropriate attributes from the global level to the cache level and stores them in an internal collection. When an attribute needs to be modified for a particular data record, the cache creates item-level copies of all attributes and stores them associated with the data record.

GetAttributesReadonly(string, bool)

Returns the cache-level instances of attributes placed on the specified field in the DAC.

Using this method, you can prevent expanding the aggregate attributes by setting the second parameter to `false`. Other overloads of this method always include both the aggregate attributes and the attributes that comprise such attributes.

Syntax:

```
public override List<PXEventSubscriberAttribute> GetAttributesReadonly(
    string name, bool extractEmmbeddedAttr)
```

Parameters:

- `name`

  The data record.

- `extractEmmbeddedAttr`

  The value that indicates whether the attributes embedded into an aggregate attribute are included into the list. If `true`, both the aggregate attribute and the attributes embedded into it are included in the list. Otherwise, only the aggregate attribute is included.

  An aggregate attribute is an attribute that derives from the `PXAggregateAttribute` class. This class allows combining multiple different attributes in a single one.

GetAttributesReadonly(object, string)

Returns the item-level attribute instances placed on the specified field, if such instances exist for the provided data record, or the cache-level instances, otherwise.

Syntax:

```
public override List<PXEventSubscriberAttribute> GetAttributesReadonly(
    object data, string name)
```

Parameters:

- `data`

  The data record.

- `name`

  The name of the field whose attributes are returned. If `null`, the method returns attributes from all fields.

Examples:

The code below gets the attributes and places them into a list.

```
protected virtual void InventoryItem_ValMethod_FieldVerifying(
    PXCache sender, PXFieldVerifyingEventArgs e)
{
    List<PXEventSubscriberAttribute> attrlist =
        sender.GetAttributesReadonly(e.Row, "ValMethod");
    ...
}
```

GetAttributesReadonly<Field>()

Returns the cache-level instances of attributes placed on the specified field in the DAC. The field is specified as the type parameter.

Syntax:

```
public List<PXEventSubscriberAttribute> GetAttributesReadonly<Field>()
    where Field : IBqlField
```

GetAttributesReadonly<Field>(object)

Returns the item-level instances of attributes placed on the specified field if such instances exist for the provided data record or the cache-level instances otherwise. The field is specified as the type parameter.

Syntax:

```
public List<PXEventSubscriberAttribute> GetAttributesReadonly<Field>(
    object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.

GetBqlField(string)

Gets the type that represents the field with the provided name in BQL expressions.

The method searches the field by its name in the `BqlFields` collection.

Syntax:

```
public Type GetBqlField(string field)
```

Parameters:

- `field`

  The name of the field.

GetBqlTable(Type)

Gets the base DAC type by which the provided DAC type is bound to the database.

Syntax:

```
public static Type GetBqlTable(Type dac)
```

Parameters:

- `dac`

  The DAC type for which the base DAC type is searched.

GetExtension<Extension>(object)

Gets the instance of the DAC extension of the specified type. The extension type is specified as the type parameter.

Syntax:

```
public override Extension GetExtension<Extension>(object item)
```

Parameters:

- `item`

  The standard data record whose extension is returned.

Examples:

The code below gets an extension data record corresponding to the given instance of the base data record.

```
InventoryItem item = cache.Current as InventoryItem;
InventoryItemExtension itemExt =
    cache.GetExtension<InventoryItemExtension>(item);
```

GetExtension<Extension>(Table)

Gets the instance of the DAC extension of the specified type. The extension type is specified as the type parameter.

Syntax:

```
public static Extension GetExtension<Extension>(Table item)
    where Extension : PXCacheExtension<Table>
```

Parameters:

- `item`

  The standard data record whose extension is returned.

Examples:

The code below gets an extension data record corresponding to the given instance of the base data record.

```
InventoryItem item = cache.Current as InventoryItem;
InventoryItemExtension itemExt =
    PXCache<InventoryItem>.GetExtension<InventoryItemExtension>(item);
```

GetField(Type)

Searches the `Fields` collection for the name of the specified type. Returns the field name if the field is found in the collection or `null` otherwise.

Syntax:

```
public string GetField(Type bqlField)
```

Parameters:

- `bqlField`

  The type declaration of the field in the DAC.

GetFieldCount()

Returns the number of fields and virtual fields which comprise the `Fields` collection.

Syntax:

```
public override int GetFieldCount()
```

GetFieldOrdinal(string)

Returns the index of the specified field in the internally kept fields map.

Syntax:

```
public override int GetFieldOrdinal(string field)
```

Parameters:

- `field`

  The name of the field whose index is returned.

GetFieldOrdinal<Field>()

Returns the index of the specified field in the internally kept fields map. The pare

Syntax:

```
public override int GetFieldOrdinal<Field>()
```

GetItemType()

Returns the DAC type of the data records in the cache.

Syntax:

```
public override Type GetItemType()
```

GetObjectHashCode(object)

Returns the hash code generated from key field values.

Syntax:

```
public override int GetObjectHashCode(object data)
```

Parameters:

- `data`

The data record.

GetStateExt(object, string)

Gets the `PXFieldState` object of the specified field in the given data record.

The method raises the `FieldSelecting` event.

Syntax:

```
public override object GetStateExt(object data, string fieldName)
```

Parameters:

- `data`

  The data record.

- `fieldName`

  The name of the field whose `PXFieldState` object is created.

GetStateExt<Field>(object)

Gets the `PXFieldState` object of the specified field in the given data record. The field is specified as the type parameter.

The method raises the `FieldSelecting` event.

Syntax:

```
public object GetStateExt<Field>(object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.

GetStatus(object)

Returns the status of the provided data record. The *PXEntryStatus* enumeration defines the possible status values. For example, the status can indicate whether the data record has been inserted, updated, or deleted.

Syntax:

```
public override PXEntryStatus GetStatus(object item)
```

Parameters:

- `item`

  The data record whose status is requested.

Examples:

The code below shows how a status of a data record can be checked in an event handler.

```
protected virtual void Vendor_RowSelected(PXCache sender,
                                          PXRowSelectedEventArgs e)
{
    Vendor vend = e.Row as Vendor;
    if (vend != null && sender.GetStatus(vend) == PXEntryStatus.Notchanged)
    {
        ...
```

```
    }
}
```

GetValue(object, int)

Returns the value of the specified field in the given data record without raising any events. The field is specified by its index—see the *GetFieldOrdinal(string)* method.

Syntax:

```
public override object GetValue(object data, int ordinal)
```

Parameters:

- `data`

  The data record.
- `ordinal`

  The index of the field whose value is returned.

GetValue(object, string)

Returns the value of the specified field in the given data record without raising any events.

Syntax:

```
public override object GetValue(object data, string fieldName)
```

Parameters:

- `data`

  The data record.
- `fieldName`

  The name of the field whose value is returned.

Remarks:

To get the field of a data record of a known DAC type, you can use DAC properties. If a type of a data record is unknown (for example, when it is available as `object`), you can use the `GetValue()` methods to get a value of a field. These methods can also be used to get values of fields defined in extensions (another way is to get the extension data record through the *GetExtension<>()* method).

The *GetValueExt()* methods are used to get the value or the field state object and raise events.

Examples:

The code below iterates over all fields of a specific DAC (including fields defined in extensions) and checks whether a value is null.

```
foreach (string field in sender.Fields)
{
    if (sender.GetValue(row, field) == null)
        ...
}
```

Here, `sender` is an instance of the `PXCache<Table>` type and `row` references an instance of `Table` (although the `row` variable may be of `object` type).

GetValue<Field>(object)

Returns the value of the specified field in the given data record without raising any events. The field is specified as the type parameter.

Syntax:

```
public object GetValue<Field>(object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record whose field value is returned.

Examples:

The code below gets the value of one field and assigns it to another field.

```
protected virtual void APInvoice_VendorLocationID_FieldUpdated(
    PXCache sender, PXFieldUpdatedEventArgs e)
{
    sender.SetValue<APInvoice.payLocationID>(
        e.Row, sender.GetValue<APInvoice.vendorLocationID>(e.Row));
}
```

GetValueExt(object, string)

Returns the value or the `PXFieldState` object of the specified field in the given data record. The `PXFieldState` object is returned if the field is in the `AlteredFields` collection.

The method raises the `FieldSelecting` event.

Syntax:

```
public override object GetValueExt(object data, string fieldName)
```

Parameters:

- `data`

  The data record.

- `fieldName`

  The name of the field whose value or `PXFieldState` object is returned.

GetValueExt<Field>(object)

Gets either the value or `PXFieldState` object of the specified field in the given data record. The `PXFieldState` object is returned if the field name is in the `AlteredFields` collection. The field is specified as the type parameter.

The method raises the `FieldSelecting` event.

Syntax:

```
public object GetValueExt<Field>(object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record whose field value or `PXFieldState` object is returned.

Examples:

The code below shows how you can get the value of a field if the `GetValueExt<>()` method returns the field state object.

```
object finPeriodID = cache.GetValueExt<APRegister.finPeriodID>(doc);
if (finPeriodID is PXFieldState)
{
    finPeriodID = ((PXFieldState)finPeriodID).Value;
}
```

GetValueOriginal(object, string)

Returns the value of the specified field for the data record as it is stored in the database.

Syntax:

```
public override object GetValueOriginal(object data, string fieldName)
```

Parameters:

- `data`

  The data record.

- `fieldName`

  The name of the field whose original value is returned.

GetValueOriginal<Field>(object)

Returns the value of the specified field for the data record as it is stored in the database. The field is specified as the type parameter.

Syntax:

```
public object GetValueOriginal<Field>(object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.

GetValuePending(object, string)

Returns the value of the field from the provided data record when the data record's update or insertion is in progress.

The method raises the `FieldSelecting` event.

Syntax:

```
public override object GetValuePending(object data, string fieldName)
```

Parameters:

- `data`

  The data record.

- `fieldName`

  The field name.

GetValuePending<Field>(object)

Returns the value of the field from the provided data record when the data record's update or insertion is in progress. The field is specified as the type parameter.

The method raises the `FieldSelecting` event.

Syntax:

```
public object GetValuePending<Field>(object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.

HasAttributes(object)

Checks if the provided data record has any attributes attached to its fields.

Syntax:

```
public override bool HasAttributes(object data)
```

Parameters:

- `data`

  The data record.

Insert()

Initializes a new data record with default values and inserts it into the cache by invoking the *Insert(object)* method. Returns the new data record inserted into the cache.

Syntax:

```
public override object Insert()
```

Examples:

```
APInvoice newItem = cache.Insert();
```

Insert(object)

Inserts the provided data record into the cache. Returns the inserted data record or `null` if the data record wasn't inserted.

The method raises the following events: `FieldDefaulting`, `FieldUpdating`, `FieldVerifying`, `FieldUpdated`, `RowInserting`, and `RowInserted`. See *Inserting a Data Record* for the events chart.

The method does not check if the data record exists in the database. The AllowInsert property does not affect this method unlike the *Insert(IDictionary)* method.

In case of successful insertion, the method marks the data record as `Inserted`, and it becomes accessible through the `Inserted` collection.

Syntax:

```
public override object Insert(object data)
```

Parameters:

- `data`

  The data record to insert into the cache.

Examples:

The code below initializes a new instance of the `APInvoice` data record and inserts it into the cache.

```
APInvoice newDoc = new APInvoice();
newDoc.VendorID = Document.Current.VendorID;
Document.Insert(newDoc);
```

Here `Document` is a data view that selects `APInvoice` data records. Invoking the `Insert()` method on it is a shortcut for the following code.

```
Document.Cache.Insert(newDoc);
```

Insert(IDictionary)

Initializes a new data record using the provided field values and inserts the data record into the cache. Returns 1 in case of successful insertion, and 0 otherwise.

The method raises the following events: `FieldDefaulting`, `FieldUpdating`, `FieldVerifying`, `FieldUpdated`, `RowInserting`, and `RowInserted`. See *Inserting a Data Record* for the events chart.

The method does not check if the data record exists in the database. The values provided in the dictionary are not readonly and can be updated during execution of the method. The method is typically used by the system when the values are received from the user interface. If the `AllowInsert` property is `false`, the data record is not inserted and the method returns 0.

In case of successful insertion, the method marks the data record as `Inserted`, and it becomes accessible through the `Inserted` collection.

Syntax:

```
public override int Insert(IDictionary values)
```

Parameters:

- `values`

  The dictionary with values to initialize the data record fields. The dictionary keys are field names.

Load()

Loads dirty items and other cache state objects from the session. The application does not typically use this method.

Syntax:

```
public override void Load()
```

Locate(object)

Searches the cache for a data record that has the same key fields as the provided data record. If the data record is not found in the cache, the method retrieves the data record from the database and places it into the cache with the `NotChanged` status. The method returns the located or retrieved data record.

The `AllowSelect` property does not affect this method unlike the *Locate(IDictionary)* method.

Syntax:

```
public override object Locate(object item)
```

Parameters:

- `item`

  The data record to locate in the cache.

Locate(IDictionary)

Searches the cache for a data record that has the same key fields as in the provided dictionary. If the data record is not found in the cache, the method initializes a new data record with the provided values and places it into the cache with the `NotChanged` status.

Returns 1 if a data record is successfully located or placed into the cache, and returns 0 if placing into the cache fails or the `AllowSelect` property is `false`.

Syntax:

```
public override int Locate(IDictionary keys)
```

Parameters:

- `keys`

  The dictionary with values to initialize the data record fields. The dictionary keys are field names.

Normalize()

Recalculates internally stored hash codes. The method should be called after a key field is modified in a data record from the cache.

Syntax:

```
public override void Normalize()
```

ObjectToString(object)

Returns a string of key fields and their values in the *{key1=value1, key2=value2}* format.

Syntax:

```
public override string ObjectToString(object data)
```

Parameters:

- `data`

  The data record which key fields are written to a string.

ObjectsEqual(object, object)

Compares two data records by the key fields. Returns `true` if the values of all key fields in the data records are equal. Otherwise, returns `false`.

Syntax:

```
public override bool ObjectsEqual(object a, object b)
```

Parameters:

- `a`

  The first data record to compare.

- `b`

  The second data record to compare.

ObjectsEqual<Field1>(object, object)

Compares two data records by the field value.

Syntax:

```
public bool ObjectsEqual<Field1>(object a, object b)
    where Field1 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.
- `b`

  The second data record to compare.

ObjectsEqual<Field1, Field2>(object, object)

Compares two data records by the values of the specified fields.

Syntax:

```
public bool ObjectsEqual<Field1, Field2>(object a, object b)
    where Field1 : IBqlField
    where Field2 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.
- `b`

  The second data record to compare.

ObjectsEqual<Field1, Field2, Field3>(object, object)

Compares two data records by the values of the specified fields.

Syntax:

```
public bool ObjectsEqual<Field1, Field2, Field3>(object a, object b)
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.
- `b`

  The second data record to compare.

Examples:

This method and its overloads are often used in the `FieldUpdated` or `RowUpdated` event handlers. The following code can be used in such event handlers for the `APInvoice` data records.

```
if (!sender.ObjectsEqual<APInvoice.docDate,
                         APInvoice.finPeriodID,
                         APInvoice.curyID>(e.Row, e.OldRow))
```

```
    ...
```

ObjectsEqual<Field1, Field2, Field3, Field4>(object, object)

Compares two data records by the values of the specified fields.

Syntax:

```
public bool ObjectsEqual<Field1, Field2, Field3, Field4>(object a, object b)
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.
- `b`

  The second data record to compare.

ObjectsEqual<Field1, Field2, Field3, Field4, Field5>(object, object)

Compares two data records by the values of the specified fields.

Syntax:

```
public bool ObjectsEqual<Field1, Field2, Field3,
                         Field4, Field5>(object a, object b)
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.
- `b`

  The second data record to compare.

ObjectsEqual<Field1, Field2, Field3, Field4, Field5, Field6>(object, object)

Compares two data records by the values of the specified fields.

Syntax:

```
public bool ObjectsEqual<Field1, Field2, Field3,
                         Field4, Field5, Field6>(object a, object b)
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.

- `b`

  The second data record to compare.

ObjectsEqual<Field1, Field2, Field3, Field4, Field5, Field6, Field7>(object, object)

Compares two data records by the values of the specified fields.

Syntax:

```
public bool ObjectsEqual<Field1, Field2, Field3, Field4,
                         Field5, Field6, Field7>(object a, object b)
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
    where Field7 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.

- `b`

  The second data record to compare.

ObjectsEqual<Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8>(object, object)

Compares two data records by the values of the specified fields.

Syntax:

```
public bool ObjectsEqual<Field1, Field2, Field3, Field4,
                         Field5, Field6, Field7, Field8>(object a, object b)
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
    where Field7 : IBqlField
    where Field8 : IBqlField
```

Parameters:

- `a`

  The first data record to compare.

- `b`

  The second data record to compare.

Persist(PXDBOperation)

Saves the modifications of a particular type from the cache to the database. Returns the number of saved data records.

Using this method, you can update, delete, or insert all data records kept by the cache. You can also perform different operations at once by passing a combination of `PXDBOperation` values, such as `PXDBOperation.Insert | PXDBOperation.Update`.

The method raises the following events: `RowPersisting`, `CommandPreparing`, `RowPersisted`, `ExceptionHandling`.

Syntax:

```
public override int Persist(PXDBOperation operation)
```

Parameters:

- `operation`

  The value that indicates the types of database operations to execute, either one of `PXDBOperation.Insert`, `PXDBOperation.Update`, and `PXDBOperation.Delete` values or their bitwise "or" (`|`) combination.

Examples:

The code below modifies a `Vendor` data record, updates it in the cache, saves changes to update `Vendor` data records to the database, and causes raising of the `RowPersisted` event with indication that the operation has completed successfully.

```
vendor.Status = BAccount.status.Inactive;
Caches[typeof(Vendor)].Update(vendor);
Caches[typeof(Vendor)].Persist(PXDBOperation.Update);
Caches[typeof(Vendor)].Persisted(false);
```

Persist(object, PXDBOperation)

Saves the modification of the specified type from the cache to the database for a particular data record.

Syntax:

```
public override void Persist(object row, PXDBOperation operation)
```

Parameters:

- `row`

  The data record to save to the database.

- `operation`

  The database operation to perform for the data record, either one of `PXDBOperation.Insert`, `PXDBOperation.Update`, and `PXDBOperation.Delete` values or their bitwise "or" (`|`) combination.

PersistDeleted(object)

Deletes the provided data record from the database by the key fields. Returns `true` if the data record has been deleted sucessfully, or `false` otherwise.

The method raises the following events: `RowPersisting`, `CommandPreparing`, `RowPersisted`, `ExceptionHandling`.

The default behavior can be modified by the `PXDBInterceptor` attribute.

Syntax:

```
public override bool PersistDeleted(object row)
```

Parameters:

- `row`

  The data record to deleted from the database.

PersistInserted(object)

Inserts the provided data record into the database. Returns `true` if the data record has been inserted sucessfully, or `false` otherwise.

The method throws an exception if the data record with such keys exists in the database.

The method raises the following events: `RowPersisting`, `CommandPreparing`, `RowPersisted`, `ExceptionHandling`.

The default behavior can be modified by the `PXDBInterceptor` attribute.

Syntax:

```
public override bool PersistInserted(object row)
```

Parameters:

- `row`

  The data record to insert into the database.

PersistUpdated(object)

Updates the provided data record in the database. Returns `true` if the data record has been updated sucessfully, or `false` otherwise.

The method raises the following events: `RowPersisting`, `CommandPreparing`, `RowPersisted`, `ExceptionHandling`.

The default behavior can be modified by the `PXDBInterceptor` attribute.

Syntax:

```
public override bool PersistUpdated(object row)
```

Parameters:

- `row`

  The data record to update in the database.

Persisted(bool)

Completes saving changes to the database by raising the `RowPersisted` event for all persisted data records.

Syntax:

```
public override void Persisted(bool isAborted)
```

Parameters:

- `isAborted`

  The value indicating whether the database operation has been aborted or completed.

Examples:

You need to call this method in the application only when you call the `Persist()`, `PersistInserted()`, `PersistUpdated()`, or `PersistDeleted()` method, as the following example shows.

```
// Opening a transaction and saving changes to the provided
// new data record
using (PXTransactionScope ts = new PXTransactionScope())
{
    cache.PersistInserted(item);
    ts.Complete(this);
```

```
}

// Indicating successful completion of saving changes to the database
cache.Persisted(false);
```

RaiseCommandPreparing(string, object, object, PXDBOperation, Type, out)

Raises the `CommandPreparing` event for the specified field and data record.

Syntax:

```
public bool RaiseCommandPreparing(
    string name, object row, object value, PXDBOperation operation,
    Type table, out PXCommandPreparingEventArgs.FieldDescription description)
```

Parameters:

- `name`

  The name of the field for which the event is raised.
- `row`

  The data record for which the event is raised.
- `value`

  The current field value.
- `operation`

  The current database operation.
- `table`

  The type of DAC objects placed in the cache.
- `(out) description`

  The *FieldDescription* object containing the description of the current field.

RaiseCommandPreparing<Field>(object, object, PXDBOperation, Type, out)

Raises the `CommandPreparing` event for the specified field and data record.

Syntax:

```
public bool RaiseCommandPreparing<Field>(
    object row, object value, PXDBOperation operation,
    Type table, out PXCommandPreparingEventArgs.FieldDescription description)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record for which the event is raised.
- `value`

  The current field value.
- `operation`

  The current database operation.
- `table`

  The type of DAC objects placed in the cache.

- `(out) description`

  The *FieldDescription* object containing the description of the current field.

RaiseExceptionHandling(string, object, object, Exception)

Raises the `ExceptionHandling` event for the specified field and data record.

Syntax:

```
public bool RaiseExceptionHandling(string name, object row,
                                   object newValue, Exception exception)
```

Parameters:

- `name`

  The name of the field for which the event is raised.

- `row`

  The data record for which the event is raised.

- `newValue`

  The new value of the current field generated by the operation that causes the exception.

- `exception`

  The exception that causes the event.

RaiseExceptionHandling<Field>(object, object, Exception)

Raises the `ExceptionHandling` event for the specified field and data record.

Syntax:

```
public bool RaiseExceptionHandling<Field>(object row, object newValue,
                                          Exception exception)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record for which the event is raised.

- `newValue`

  The new value of the current field generated by the operation that causes the exception.

- `exception`

  The exception that causes the event.

Examples:

A typical use of the method is found in event handlers when the value of a field doesn't pass validation. If the value is validated in a `RowUpdating` event handler, you should pass an instance of `PXSetPropertyException` with the error message to the method. The code below gives an example for this case.

```
INComponent row = e.NewRow as INComponent;

if (row != null && row.Qty != null &&
    row.MinQty != null && row.Qty <= row.MinQty)
{
    sender.RaiseExceptionHandling<INComponent.qty>(
        row, row.Qty, new PXSetPropertyException(
```

```
            "Quantity must be greater or equal to Min. Quantity."));
}
```

RaiseFieldDefaulting(string, object, out)

Raises the `FieldDefaulting` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldDefaulting(string name, object row, out object newValue)
```

Parameters:

- `name`

  The name of the field for which the event is raised.
- `row`

  The data record for which the event is raised.
- `newValue`

  The default value for the current field.

RaiseFieldDefaulting<Field>(object, out)

Raises the `FieldDefaulting` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldDefaulting<Field>(object row, out object newValue)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record for which the event is raised.
- `newValue`

  The default value for the current field.

Examples:

The code below shows how to raise an event.

```
CashAccount acct = null;

// Get the cache (the other way is to use Cache property of a data view)
PXCache cache = this.Caches[typeof(ARPayment)].Cache;

// Initialize a new ARPayment data record
ARPayment payment = new ARPayment();
payment.CustomerID = aDoc.CustomerID;
payment.CustomerLocationID = aDoc.CustomerLocationID;

// You could execute cache.Insert(payment) to insert the data record
// in the cache and raise the events including FieldDefaulting.
// However, we need to raise FieldDefaulting only on one field.

// Declare a variable for the value
object newValue;

// Raise the FieldDefaulting event
cache.RaiseFieldDefaulting<ARPayment.cashAccountID>(payment, out newValue);

// Convert the object to the data type of the field
```

```
Int32? acctID = newValue as Int32?;

// Use the value to retrieve the CashAccount data record
if (acctID.HasValue)
{
 acct = PXSelect<CashAccount,
        Where<CashAccount.cashAccountID,
            Equal<Required<CashAccount.cashAccountID>>>>.
        Select(this, acctID);
}
```

RaiseFieldSelecting(string, object, ref, bool)

Raises the `FieldSelecting` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldSelecting(string name, object row,
                                ref object returnValue,
                                bool forceState)
```

Parameters:

- `name`

  The name of the field for which the event is raised.
- `row`

  The data record for which the event is raised.
- `returnValue`

  The external presentation of the value of the current field.
- `forceState`

  The value indicating whether the *PXFieldState* object should be generated.

RaiseFieldSelecting<Field>(object, ref, bool)

Raises the `FieldSelecting` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldSelecting<Field>(object row, ref object returnValue,
                                         bool forceState)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record for which the event is raised.
- `returnValue`

  The external presentation of the value of the current field.
- `forceState`

  The value indicating whether the *PXFieldState* object should be generated.

RaiseFieldUpdated(string, object, object)

Raises the `FieldUpdated` event for the specified field and data record.

Syntax:

```
public void RaiseFieldUpdated(string name, object row, object oldValue)
```

Parameters:

- `name`

  The name of the field for which the event is raised.

- `row`

  The data record for which the event is raised.

- `oldValue`

  The value of the current field befor update.

RaiseFieldUpdated<Field>(object, object)

Raises the `FieldUpdated` event for the specified field and data record.

Syntax:

```
public void RaiseFieldUpdated<Field>(object row, object oldValue)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record for which the event is raised.

- `oldValue`

  The value of the current field befor update.

RaiseFieldUpdating(string, object, ref)

Raises the `FieldUpdating` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldUpdating(string name, object row, ref object newValue)
```

Parameters:

- `name`

  The name of the field for which the event is raised.

- `row`

  The data record for which the event is raised.

- `newValue`

  The updated value of the current field.

RaiseFieldUpdating<Field>(object, ref)

Raises the `FieldUpdating` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldUpdating<Field>(object row, ref object newValue)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record for which the event is raised.

- `newValue`

  The updated value of the current field.

RaiseFieldVerifying(string, object, ref)

Raises the `FieldVerifying` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldVerifying(string name, object row, ref object newValue)
```

Parameters:

- `name`

  The name of the field for which the event is raised.

- `row`

  The data record for which the event is raised.

- `newValue`

  The updated value of the current field.

RaiseFieldVerifying<Field>(object, ref)

Raises the `FieldVerifying` event for the specified field and data record.

Syntax:

```
public bool RaiseFieldVerifying<Field>(object row, ref object newValue)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record for which the event is raised.

- `newValue`

  The updated value of the current field.

RaiseRowDeleted(object)

Raises the `RowDeleted` event for the specified data record.

Syntax:

```
public void RaiseRowDeleted(object item)
```

Parameters:

- `item`

  The data record for which the event is raised.

RaiseRowDeleting(object)

Raises the `RowDeleting` event for the specified data record.

Syntax:

```
public bool RaiseRowDeleting(object item)
```

Parameters:

- `item`

  The data record for which the event is raised.

RaiseRowInserted(object)

Raises the `RowInserted` event for the specified data record.

Syntax:

```
public void RaiseRowInserted(object item)
```

Parameters:

- `item`

  The data record for which the event is raised.

RaiseRowInserting(object)

Raises the `RowInserting` event for the specified data record.

Syntax:

```
public bool RaiseRowInserting(object item)
```

Parameters:

- `item`

  The data record for which the event is raised.

RaiseRowPersisted(object, PXDBOperation, PXTranStatus, Exception)

Raises the `RowPersisted` event for the specified data record.

Syntax:

```
public void RaiseRowPersisted(object item, PXDBOperation operation,
                              PXTranStatus tranStatus, Exception exception)
```

Parameters:

- `item`

  The data record for which the event is raised.

- `operation`

  The `PXDBOperation` value indicating the type of the current database operation.

- `tranStatus`

  The *PXTranStatus* value indicating the status of the transaction.

- `exception`

  The exception thrown while the database operation was executed.

RaiseRowPersisting(object, PXDBOperation)

Raises the `RowPersisting` event for the specified data record.

Syntax:

```
public bool RaiseRowPersisting(object item, PXDBOperation operation)
```

Parameters:

- `item`

  The data record for which the event is raised.
- `operation`

  he `PXDBOperation` value indicating the type of the current database operation.

RaiseRowSelected(object)

Raises the `RowSelected` event for the specified data record.

Syntax:

```
public void RaiseRowSelected(object item)
```

Parameters:

- `item`

  The data record for which the event is raised.

RaiseRowSelecting(object, PXDataRecord, ref int, bool)

Raises the `RowSelecting` event for the specified data record.

Syntax:

```
public bool RaiseRowSelecting(object item, PXDataRecord record,
                              ref int position, bool isReadOnly)
```

Parameters:

- `item`

  The data record for which the event is raised.
- `record`

  The `PXDataRecord` object wrapping the result set row.
- `(ref) position`

  The current index in the list of `PXDataRecord` columns.
- `isReadOnly`

  The value indicating if the data record is read-only.

RaiseRowUpdated(object, object)

Raises the `RowUpdated` event for the specified data record.

Syntax:

```
public void RaiseRowUpdated(object newItem, object oldItem)
```

Parameters:

- `newItem`

  The updated version of the data record.
- `oldItem`

  The version of the data record before update.

RaiseRowUpdating(object, object)

Raises the `RowUpdating` event for the specified data record.

Syntax:

```
public bool RaiseRowUpdating(object item, object newItem)
```

Parameters:

- `item`

  The version of the data record before update.
- `newItem`

  The updated version of the data record.

Remove(object)

Completely removes the provided data record from the cache without raising any events.

Syntax:

```
public override void Remove(object item)
```

Parameters:

- `item`

  The data record to remove from the cache.

Examples:

The code below locates a data record in the cache and, if the data record has not been changed, silently removes it from the cache.

```
// Searching the data record by its key fields in the cache
object cached = sender.Locate(item);

// Checking the status
if (cached != null && (sender.GetStatus(cached) == PXEntryStatus.Held ||
                       sender.GetStatus(cached) == PXEntryStatus.Notchanged))
{
    // Removing without events
    sender.Remove(cached);
}
```

The `Held` status indicates that a data record has not been changed but needs to the preserved in the session.

RestoreCopy(object, object)

Copies values of all fields from the second data record to the first data record.

The data records should have the DAC type of the cache, or the method does nothing.

Syntax:

```
public override void RestoreCopy(object item, object copy)
```

Parameters:

- `item`

  The data record whose field values are updated.
- `copy`

  The data record whose field values are copied.

RestoreCopy(Table, Table)

Copies values of all fields from the second data record to the first data record.

Syntax:

```
public static void RestoreCopy(Table item, Table copy)
```

Parameters:

- `item`

  The data record whose field values are updated.
- `copy`

  The data record whose field values are copied.

Examples:

The code below modifies an `APRegister` data record and copies the values of all its fields to an `APInvoice` data record.

```
APRegister doc = ...
APInvoice apdoc = ...
...
// Modifying the doc data record
doc.OpenDoc = true;
doc.ClosedFinPeriodID = null;
...
// Copying all fields of doc to apdoc (APInvoice derives from APRegister)
PXCache<APRegister>.RestoreCopy(apdoc, doc);
```

Select(PXDataRecord, ref int, bool, out bool)

Creates a data record from the `PXDataRecord` object and places it into the cache with the `NotChanged` status if the data record isn't found among the modified data records in the cache.

If `isReadOnly` is `false` then:

- If the cache already contains the data record with the same keys and the `NotChanged` status, the method returns this data record updated to the state of `PXDataRecord`.
- If the cache contains the same data record with the `Updated` or `Inserted` status, the method returns this data record.

In other cases and when `isReadonly` is `true`, the method returns the data record created from the `PXDataRecord` object.

If the `AllowSelect` property is `false`, the methods returns a new empty data record and the logic described above is not executed.

The method raises the `RowSelecting` event.

Syntax:

```
public override object Select(PXDataRecord record,
                              ref int position,
                              bool isReadOnly,
                              out bool wasUpdated)
```

Parameters:

- `record`

  The `PXDataRecord` object to convert to the DAC type of the cache.
- `(ref) position`

  The index of the first field to read in the list of columns comprising the `PXDataRecord` object.
- `isReadOnly`

  The value indicating if the data record with the same key fields should be located in the cache and updated.
- `(out) bool`

  The value indicating whether the data record with the same keys existed in the cache among the modified data records.

SetAltered(string, bool)

Adds the field to the `AlteredFields` list or removes it from this list.

Syntax:

```
public virtual void SetAltered(string field, bool isAltered)
```

Parameters:

- `field`

  The field name.
- `isAltered`

  The value indicating whether the field is added or removed.

SetAltered<Field>(bool)

Adds the field to the `AlteredFields` list or removes it from this list. The field is specified in the type parameter.

Syntax:

```
public virtual void SetAltered<Field>(bool isAltered)
    where Field : IBqlField
```

Parameters:

- `isAltered`

  The value indicating whether the field is added or removed.

Examples:

```
Items.Cache.SetAltered<FlatPriceItem.inventoryID>(true);
```

SetDefaultExt(object, string)

Sets the default value to the field in the provided data record.

The method raises `FieldDefaulting`, `FieldUpdating`, `FieldVerifying`, and `FieldUpdated`.

Syntax:

```
public override void SetDefaultExt(object data, string fieldName)
```

Parameters:

- `data`

  The data record.

- `fieldName`

  The name of the field to set.

SetDefaultExt<Field>(object)

Sets the default value to the field in the provided data record. The field is specified as the type parameter.

The method raises `FieldDefaulting`, `FieldUpdating`, `FieldVerifying`, and `FieldUpdated`.

Syntax:

```
public void SetDefaultExt<Field>(object data)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.

SetStatus(object, PXEntryStatus)

Sets the status to the provided data record. The *PXEntryStatus* enumeration defines the possible status values.

Syntax:

```
public override void SetStatus(object item, PXEntryStatus status)
```

Parameters:

- `item`

  The data record to set status to.

- `status`

  The new status.

Examples:

The code below checks the status of a data record and sets the status to `Updated` if the status is `Notchanged`.

```
if (Transactions.Cache.GetStatus(tran) == PXEntryStatus.Notchanged)
{
    Transactions.Cache.SetStatus(tran, PXEntryStatus.Updated);
}
```

SetValue(object, int, object)

Sets the value of the field in the provided data record without raising events. The field is specified by its index in the field map.

To set the value, raising the field-related events, use the *SetValueExt(object, string, object)* method.

Syntax:

```
public override void SetValue(object data, int ordinal, object value)
```

Parameters:

- `data`

  The data record.
- `ordinal`

  The index of the field in the internally stored field map. To get the index of a specific field, use the *GetFieldOrdinal(string)* method.
- `value`

  The value to set to the field.

SetValue(object, string, object)

Sets the value of the field in the provided data record without raising events.

To set the value, raising the field-related events, use the *SetValueExt(object, string, object)* method.

Syntax:

```
public override void SetValue(object data, string fieldName, object value)
```

Parameters:

- `data`

  The data record.
- `fieldName`

  The name of the field that is set to the value.
- `value`

  The value to set to the field.

SetValue<Field>(object, object)

Sets the value of the field in the provided data record without raising events. The field is specified in the type parameter.

To set the value, raising the field-related events, use the *SetValueExt<Field>(object, object)* method.

Syntax:

```
public void SetValue<Field>(object data, object value)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record
- `value`

The value to set to the field.

SetValueExt(object, string, object)

Sets the value of the field in the provided data record.

The method raises the `FieldUpdating`, `FieldVerifying`, and `FieldUpdated` events. To set the value to the field without raising events, use the *SetValue(object, string, object)* method.

Syntax:

```
public override void SetValueExt(object data, string fieldName, object value)
```

Parameters:

- `data`

  The data record.
- `fieldName`

  The name of the field that is set to the value.
- `value`

  The value to set to the field.

SetValueExt<Field>(object, object)

Sets the value of the field in the provided data record. The field is specified in the type parameter.

The method raises the `FieldUpdating`, `FieldVerifying`, and `FieldUpdated` events. To set the value to the field without raising events, use the *SetValue<Field>(object, object)* method.

Syntax:

```
public void SetValueExt<Field>(object data, object value)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.
- `value`

  The value to set to the field.

Examples:

The code below checks the value of one field of the `APInvoice` data record and sets another field to this value with raising of events.

```
APInvoice doc = e.Row as APInvoice;
if (doc != null && doc.CuryDocBal != null && doc.CuryDocBal != 0)
    sender.SetValueExt<APInvoice.curyOrigDocAmt>(doc, doc.CuryDocBal);
```

SetValuePending(object, string, object)

Sets the value of the field in the provided data record when the data record's update or insertion is in process and the field possibly hasn't been updated in the cache yet. The field is specified in the type parameter.

The method raises the `FieldUpdating` event.

Syntax:

```
public override void SetValuePending(object data, string fieldName, object value)
```

Parameters:

- `data`

  The data record.
- `fieldName`

  The name of the field that is set to the value.
- `value`

  The value to set to the field.

SetValuePending<Field>(object, object)

Sets the value of the field in the provided data record when the data record's update or insertion is in process and the field possibly hasn't been updated in the cache yet.

The method raises the `FieldUpdating` event.

Syntax:

```
public void SetValuePending<Field>(object data, object value)
    where Field : IBqlField
```

Parameters:

- `data`

  The data record.
- `value`

  The value to set to the field.

ToDictionary(object)

Converts the provided data record to the dictionary of field names and field values. Returns the resulting dictionary object.

The method raises the `FieldSelecting` event for each field.

Syntax:

```
public override Dictionary<string, object> ToDictionary(object data)
```

Parameters:

- `data`

  The data record to convert to a dictionary.

ToString()

Returns the string representing the current cache object.

Syntax:

```
public override string ToString()
```

ToXml(object)

Returns the XML string representing the provided data record.

The data record is represented in the XML by the *<Row>* element with the *type* attribute set to the DAC name. Each field is represented by the *<Field>* element with the *name* attribute holding the field name and the *value* attribute holding the field value.

To initialize a data record from the XML string returned by this method, use the *FromXml(string)* method.

Syntax:

```
public override string ToXml(object data)
```

Parameters:

- `data`

  The data record to convert to XML.

Unload()

Serializes the cache to the session.

Syntax:

```
public override void Unload()
```

Update(object)

Updates the provided data record in the cache.

If the data record does not exist in the cache, the method tries to retrieve it from the database. If the data record exists in the cache or database, it gets the `Updated` status. If the data record does not exist in the database, the method inserts a new data record into the cache with the `Inserted` status.

The method raises the following events: `FieldUpdating`, `FieldVerifying`, `FieldUpdated`, `RowUpdating`, and `RowUpdated`. See *Updating a Data Record* for the events flowchart. If the data record does not exist in the database, the method also causes the events of the *Insert(object)* method.

The `AllowUpdate` property does not affect the method unlike the *Update(IDictionary, IDictionary)* method.

Syntax:

```
public override object Update(object data)
```

Parameters:

- `data`

  The data record to update in the cache.

Examples:

The code below modifies an `APRegister` data record and places it in the cache with the `Updated` status or updates it in the cache if the data record is already there.

```
// Declaring a data view in a graph
public PXSelect<APRegister> APDocument;
...

APRegister apdoc = ...
// Modifying the data record
apdoc.Voided = true;
apdoc.OpenDoc = false;
```

```
apdoc.CuryDocBal = 0m;
apdoc.DocBal = 0m;

// Updating the data record in the cache
APDocument.Cache.Update(apdoc);
```

Update(IDictionary, IDictionary)

Updates the data record in the cache with the provided values.

The method initalizes a data record with the provided key fields. If the data record with such keys does not exist in the cache, the method tries to retrieve it from the database. If the data record exists in the cache or database, it gets the `Updated` status. If the data record does not exist in the database, the method inserts a new data record into the cache with the `Inserted` status.

The method raises the following events: `FieldUpdating`, `FieldVerifying`, `FieldUpdated`, `RowUpdating`, and `RowUpdated`. See *Updating a Data Record* for the events flowchart. If the data record does not exist in the database, the method also causes the events of the *Insert(object)* method.

If the `AllowUpdate` property is `false`, the data record is not updated and the methods returns 0. The method returns 1 if the data record is successfully updated or inserted.

Syntax:

```
public override int Update(IDictionary keys, IDictionary values)
```

Parameters:

- `keys`

  The values of the key fields of the data record to update.
- `values`

  The new values with which the data record fields are updated.

ValueFromString(string, string)

Converts the provided value of the field from a string to the appropriate type and returns the resulting value. No events are raised.

Syntax:

```
public override object ValueFromString(string fieldName, string val)
```

Parameters:

- `fieldName`

  The name of the field.
- `val`

  The string representation of the field value.

ValueToString(string, object)

Converts the provided value of the field to string and returns the resulting value. No events are raised.

Syntax:

```
public override string ValueToString(string fieldName, object val)
```

Parameters:

- `fieldName`

The name of the field.

- `val`

  The field value.

PXSelectBase<Table> Class

The base type for classes that define BQL statements, such as *PXSelect<>* class and its variants and the *PXProcessing<>* class and its successors.

Inheritance Hierarchy

```
PXSelectBase
```

Syntax

```
public abstract class PXSelectBase<Table> : PXSelectBase
    where Table : class, IBqlTable, new()
```

The `PXSelectBase<Table>` type exposes the following members.

Properties

- `public virtual Table` **`Current`**

  Gets or sets the `Current` property of the cache that corresponds to the DAC specified in the type parameter.

Fields

- `public PXView` **`View`**

  The *PXView* object that is created to execute the BQL statement.

Methods

Method	Description
Ask(string, string, MessageButtons)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, string, MessageButtons)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, MessageButtons, bool)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, MessageButtons, MessageIcon)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, string, MessageButtons, bool)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, string, MessageButtons, MessageIcon)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, MessageButtons, MessageIcon, bool)	Displays the dialog window with single or multiple choices for the user

Method	Description
Ask(string, string, string, MessageButtons, MessageIcon, bool)	Displays the dialog window with single or multiple choices for the user
AskExt()	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(bool)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(PXView.InitializePanel)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string, bool)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string, PXView.InitializePanel)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(PXView.InitializePanel, bool)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string, PXView.InitializePanel, bool)	Displays the dialog window configured by the `PXSmartPanel` control
ClearDialog()	Clears the dialog information saved by the graph on last invocation of the `Ask()` method
Delete(Table)	Deletes the data record by invoking the *Delete(object)* method on the cache
Extend<Parent>(Parent)	Initializes a data record of the derived DAC from the provided data record of the base DAC and inserts the new data record into the cache
GetItemType()	Returns the type of the DAC provided as the type parameter of `PXSelectBase<>` class
GetValueExt<Field>(Table)	Gets the value of the specified field for the given data record
Insert()	Inserts a new data record into the cache by invoking the *Insert()* method on the cache
Insert(Table)	Inserts the provided data record into the cache by invoking the *Insert(object)* method on the cache
Join<join>()	Appends a joining clause to the BQL statement
Locate(Table)	Searches the cache for the data record that has the same key fields as the provided data record, by invoking the *Locate(object)* method on the cache
OrderByNew<newOrderBy>()	Replaces the `OrderBy` clause if the BQL statement has one, otherwise the new `OrderBy` clause is simply attached to the BQL statement
Search<Field0>(object, params object[])	Searches for a data record by the value of specified field in the data set that corresponds to the BQL statement

Method	Description
Search<Field0, Field1>(object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2>(object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3>(object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4>(object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5>(object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6>(object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7>(object, object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8>(object, object, object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8, Field9>(object, object, object, object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
SearchAll<Sort>(object[], params object[])	Searches the data set that corresponds to the BQL statement for all data records whose fields have the specified values
SearchWindowed<Sort>(object[], int, int, params object[])	Retrieves the specified number of contiguous data records starting from the given position in the filtered data set
Select(params object[])	Executes the BQL statement and retrieves all matching data records
SelectSingle(params object[])	Retrieves the top data record of the data set that corresponds to the BQL statement
SelectWindowed(int, int, params object[])	Retrieves the specified number of data records starting from the given position
SetValueExt<Field>(Table, object)	Sets the value of the specified field in the given data record

Method	Description
Update(Table)	Updates the data record in the cache by invoking the *Update(object)* method on the cache
WhereAnd<TWhere>()	Appends a filtering expression to the BQL statement via the logical "and"
WhereNew<newWhere>()	Replaces the filtering expression in the BQL statement
WhereNot()	Adds logical "not" to the whole `Where` clause of the BQL statement, reversing the condition to the opposite
WhereOr<TWhere>()	Appends a filtering expression to the BQL statement via the logical "or"

Examples

The code below defines a data view, extends its `Where` conditional expression, and executes the data view.

```
// Definition of a data view
PXSelectBase<ARDocumentResult> sel = new PXSelectReadOnly2<ARDocumentResult,
    LeftJoin<ARInvoice, On<ARInvoice.docType, Equal<ARDocumentResult.docType>,
        And<ARInvoice.refNbr, Equal<ARDocumentResult.refNbr>>>,
    Where<ARRegister.customerID, Equal<Current<ARDocumentFilter.customerID>>>>
        (this);

ARDocumentFilter header = Filter.Current;

// Appending a condition if BranchID is specified in the filter
if (header.BranchID != null)
{
    sel.WhereAnd<Where<ARRegister.branchID,
                     Equal<Current<ARDocumentFilter.branchID>>>>();
}

// Appending a condition if DocType is specified in the filter
if (header.DocType != null)
{
    sel.WhereAnd<Where<ARRegister.docType,
                     Equal<Current<ARDocumentFilter.docType>>>>();
}

// Execution of the data view and iteration through the result set
foreach (PXResult<ARDocumentResult, ARInvoice> reg in sel.Select())
{
    ARDocumentResult res = reg;
    ARInvoice invoice = reg;
    ...
}
```

PXSelectBase<Table> Methods

The *PXSelectBase<Table>* type exposes the following methods.

Ask(string, string, MessageButtons)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string header, string message,
                           MessageButtons buttons)
```

Parameters:

- `header`

  The string displayed as the title of the dialog window.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

Ask(string, string, string, MessageButtons)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string key, string header,
                           string message, MessageButtons buttons)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `header`

  The string displayed as the title of the dialog window.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

Ask(string, string, MessageButtons, bool)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string header, string message,
                           MessageButtons buttons, bool refreshRequired)
```

Parameters:

- `header`

  The string displayed as the title of the dialog window.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

- `refreshRequired`

The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

Ask(string, string, MessageButtons, MessageIcon)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string header, string message,
                           MessageButtons buttons, MessageIcon icon)
```

Parameters:

- `header`

  The string displayed as the title of the dialog window.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

- `icon`

  The value from the *MessageIcon* enumeration that indicate which icon to display beside the message in the dialog window.

Ask(string, string, string, MessageButtons, bool)

Displays the dialog window with single or multiple choices for the user. Returns the *WebDialogResult* value that indicates which button was clicked.

This method and its overloads provide the interface for the *corresponding methods* of the `PXView` class.

Syntax:

```
public WebDialogResult Ask(string key, string header,
                           string message, MessageButtons buttons,
                           bool refreshRequired)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `header`

  The string displayed as the title of the dialog window.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

Remarks:

The method can be used to display the panel configured by the `PXSmartPanel` control. In this case, the `key` parameter is set to the `Key` property of the control, `refreshRequired` is typically set to `true`, and other parameters are set to `null`. The more common way to display a panel is to call the *AskExt(key)* method.

Note that the method is executed asynchronously. When the method invocation is reached for the first time, execution of the enclosing method stops, and a request is send to the client to display the dialog. When the user clicks one of the buttons, the webpage sends a request to the server, and the system starts execution of the method that invoked `Ask()` one more time. This time the `Ask()` method returns the value that indicates the user's choice, and code execution continues.

Examples:

The code below defines an event handler that asks for confirmation to continue deletion of a data record.

```
public PXSelect<INComponent> Components;

protected void INComponent_RowDeleting(
    PXCache sender, PXRowDeletingEventArgs e)
{
    if (Components.Ask("Deleting Revenue Component",
                       "Are you sure?",
                       MessageButtons.YesNo) != WebDialogResult.Yes)
        e.Cancel = true;
}
```

Ask(string, string, string, MessageButtons, MessageIcon)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string key, string header,
                           string message, MessageButtons buttons,
                           MessageIcon icon)
```

Parameters:

- `key`

  The identifier of the panel to display.
- `header`

  The string displayed as the title of the dialog window.
- `message`

  The string displayed as the message inside the dialog window.
- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.
- `icon`

  The value from the *MessageIcon* enumeration that indicate which icon to display beside the message in the dialog window.

Ask(string, string, MessageButtons, MessageIcon, bool)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string header, string message,
                           MessageButtons buttons, MessageIcon icon,
                           bool refreshRequired)
```

Parameters:

- `header`

  The string displayed as the title of the dialog window.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

- `icon`

  The value from the *MessageIcon* enumeration that indicate which icon to display beside the message in the dialog window.

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

Ask(string, string, string, MessageButtons, MessageIcon, bool)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string key, string header,
                           string message, MessageButtons buttons,
                           MessageIcon icon, bool refreshRequired)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `header`

  The string displayed as the title of the dialog window.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

- `icon`

  The value from the *MessageIcon* enumeration that indicate which icon to display beside the message in the dialog window.

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt()

Displays the dialog window configured by the `PXSmartPanel` control. As a key, the method uses the name of the variable that holds the BQL statement. The method requests repainting of the panel.

Syntax:

```
public WebDialogResult AskExt()
```

AskExt(string)

Displays the dialog window configured by the `PXSmartPanel` control. The method requests repainting of the panel.

Syntax:

```
public WebDialogResult AskExt(string key)
```

Parameters:

- `key`

  The identifier of the panel to display.

AskExt(bool)

Displays the dialog window configured by the `PXSmartPanel` control. As a key, the method uses the name of the variable that holds the BQL statement.

Syntax:

```
public WebDialogResult AskExt(bool refreshRequired)
```

Parameters:

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt(PXView.InitializePanel)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(PXView.InitializePanel initializeHandler)
```

Parameters:

- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.

AskExt(string, bool)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(string key, bool refreshRequired)
```

Parameters:

- `key`

The identifier of the panel to display.

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt(string, PXView.InitializePanel)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(string key,
                              PXView.InitializePanel initializeHandler)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.

AskExt(PXView.InitializePanel, bool)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(PXView.InitializePanel initializeHandler,
                              bool refreshRequired)
```

Parameters:

- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt(string, PXView.InitializePanel, bool)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(string key,
                              PXView.InitializePanel initializeHandler,
                              bool refreshRequired)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.

- `refreshRequired`

The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

ClearDialog()

Clears the dialog information saved by the graph on last invocation of the `Ask()` method.

Syntax:

```
public void ClearDialog()
```

Delete(Table)

Deletes the data record by invoking the *Delete(object)* method on the cache. Returns the data record marked as deleted.

Syntax:

```
public virtual Table Delete(Table item)
```

Parameters:

- `item`

  The data record to delete.

Extend<Parent>(Parent)

Initializes a data record of the derived DAC from the provided data record of the base DAC and inserts the new data record into the cache. Returns the inserted data record.

The method relies on the *Extend<Parent>(Parent)* method called on the cache.

Syntax:

```
public virtual Table Extend<Parent>(Parent item)
    where Parent : class, IBqlTable, new()
```

`Table` must derive from `Parent`. The current cache object should be of `PXCache<Table>` type.

Parameters:

- `item`

  The instance of the base DAC.

Examples:

Suppose that the `B` DAC derives from the `A` DAC, as follows.

```
[Serializable]
public class A : IBqlTable { ... }

[Serializable]
public class B : A { ... }
```

The following data views can be declared in a graph.

```
PXSelect<A> BaseRecords;
PXSelect<B> Records;
```

The code above will result in initialization of two caches, of `PXCache<A>` and `PXCache<B>` types. The following code initializes a data record of derived type and inserts it into the cache.

```
A baseRec = BaseRecords.Insert();
```

```
B rec = Records.Extend<B>(baseRec);
```

GetItemType()

Returns the type of the DAC provided as the type parameter of `PXSelectBase<>` class. For BQL statements that are derived from `PXSelectBase<>`, it is the first mentioned DAC.

Syntax:

```
public Type GetItemType()
```

GetValueExt<Field>(Table)

Gets the value of the specified field for the given data record. The method relies on the *GetValueExt<Field>(Table, object)* method of the cache, but unlike the cache's method always returns a value, not a `PXFieldState` object.

Syntax:

```
public virtual object GetValueExt<Field>(Table row)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record whose field value is returned.

Insert()

Inserts a new data record into the cache by invoking the *Insert()* method on the cache. Returns the inserted data record or `null`-if the insertion fails.

Syntax:

```
public virtual Table Insert()
```

Insert(Table)

Inserts the provided data record into the cache by invoking the *Insert(object)* method on the cache. Returns the inserted data record or `null`-if the insertion fails.

Syntax:

```
public virtual Table Insert(Table item)
```

Parameters:

- `item`

  The data record to insert.

Join<join>()

Appends a joining clause to the BQL statement.

Syntax:

```
public virtual void Join<join>()
    where join : IBqlJoin, new()
```

Examples:

The code below appends the `LeftJoin` clause to the BQL statement.

```
PXSelectBase<GLTran> select = new PXSelect<GLTran>(this);

select.Join<LeftJoin<AP.APTran,
    On<AP.APTran.refNbr, Equal<GLTran.refNbr>,
    And<AP.APTran.lineNbr, Equal<GLTran.tranLineNbr>>>>>();
```

Locate(Table)

Searches the cache for the data record that has the same key fields as the provided data record, by invoking the *Locate(object)* method on the cache. Returns the data record if it is found in the cache or null otherwise.

Syntax:

```
public virtual Table Locate(Table item)
```

Parameters:

- `item`

  The data record that is searched in the cache by the values of its key fields.

OrderByNew<newOrderBy>()

Replaces the `OrderBy` clause if the BQL statement has one, otherwise the new `OrderBy` clause is simply attached to the BQL statement.

Syntax:

```
public virtual void OrderByNew<newOrderBy>()
    where newOrderBy : IBqlOrderBy, new()
```

Examples:

The code below initializes a data view as a local variable and adds different ordering expression depending on the value of a variable.

```
// Initialization of a data view
PXSelectBase<INLotSerialStatus> cmd =
    new PXSelect<INLotSerialStatus, ...>(this);

// Adding a different ordering expression depending on
// a variable's value
switch (lotSerIssueMethod)
{
    case INLotSerIssueMethod.FIFO:
        cmd.OrderByNew<
            OrderBy<Asc<INLocation.pickPriority,
                    Asc<INLotSerialStatus.receiptDate,
                    Asc<INLotSerialStatus.lotSerialNbr>>>>>();
        break;
    case INLotSerIssueMethod.LIFO:
        cmd.OrderByNew<
            OrderBy<Asc<INLocation.pickPriority,
                    Desc<INLotSerialStatus.receiptDate,
                    Asc<INLotSerialStatus.lotSerialNbr>>>>>();
        break;
    ...
}
```

Search<Field0>(object, params object[])

Searches for a data record by the value of specified field in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified field and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0>(
    object field0, params object[] arguments)
    where Field0 : IBqlField
```

Parameters:

- `field0`

  The value of `Field0` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Examples:

The code below finds the data record with the given reference numbe among the possible results of the data view.

```
// Defining the data view in a graph
public PXSelect<ARInvoice,
    Where<ARInvoice.docType, Equal<Optional<ARInvoice.docType>>>> Document;
...
// Search a data record with the given value of the RefNbr field
Document.Search<ARInvoice.refNbr>(ardoc.RefNbr, ardoc.DocType);

// The Current property is now pointing to the data record found
// by Search<>(...)
Document.Current.InstallmentCntr = Convert.ToInt16(installments.Count);
...
```

Note that the `Search<>(...)` method has two parameters here. The first one is the value of the `RefNbr` field to search by, while the second one is the value to replace the `Optional` parameter in the BQL command.

Search<Field0, Field1>(object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1>(
    object field0, object field1, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
```

Parameters:

- `field0, field1`

  The values of `Field0` and `Field1` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2>(object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2>(
    object field0, object field1, object field2, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
```

Parameters:

- `field0 - field2`

  The values of `Field0`-`Field2` by which the data set is filtered and sorted.
- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3>(object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2, Field3>(
    object field0, object field1, object field2,
    object field3, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
```

Parameters:

- `field0 - field3`

  The values of `Field0`-`Field3` by which the data set is filtered and sorted.
- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4>(object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2,
                                          Field3, Field4>(
    object field0, object field1, object field2, object field3,
    object field4, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
```

```
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
```

Parameters:

- `field0 - field4`

  The values of `Field0`-`Field4` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5>(object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2,
                                          Field3, Field4, Field5>(
    object field0, object field1, object field2, object field3,
    object field4, object field5, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
```

Parameters:

- `field0 - field5`

  The values of `Field0`-`Field5` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6>(object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2, Field3,
                                          Field4, Field5, Field6>(
    object field0, object field1, object field2, object field3,
    object field4, object field5, object field6, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
```

Parameters:

- `field0 - field6`

  The values of `Field0`-`Field6` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7>(object, object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2,
                                          Field3, Field4, Field5,
                                          Field6, Field7>(
    object field0, object field1, object field2, object field3,
    object field4, object field5, object field6, object field7,
    params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
    where Field7 : IBqlField
```

Parameters:

- `field0 - field7`

  The values of `Field0`-`Field7` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8>(object, object, object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2,
                                          Field3, Field4, Field5,
                                          Field6, Field7, Field8>(
    object field0, object field1, object field2, object field3,
    object field4, object field5, object field6, object field7,
    object field8, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
```

```
    where Field6 : IBqlField
    where Field7 : IBqlField
    where Field8 : IBqlField
```

Parameters:

- `field0 - field8`

  The values of `Field0`-`Field8` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8, Field9>(object, object, object, object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public virtual PXResultset<Table> Search<Field0, Field1, Field2, Field3,
                                          Field4, Field5, Field6, Field7,
                                          Field8, Field9>(
    object field0, object field1, object field2, object field3,
    object field4, object field5, object field6, object field7,
    object field8, object field9, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
    where Field7 : IBqlField
    where Field8 : IBqlField
    where Field9 : IBqlField
```

Parameters:

- `field0 - field9`

  The values of `Field0`-`Field9` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

SearchAll<Sort>(object[], params object[])

Searches the data set that corresponds to the BQL statement for all data records whose fields have the specified values. The fields are specified in the type parameter. The method extends the BQL statement with filtering and ordering by the fields and retrieves all data records from the resulting data set.

Though ordering may seem superfluous here, it is needed for better performance of the selection from the database.

Syntax:

```
public virtual PXResultset<Table> SearchAll<Sort>(
    object[] searchValues, params object[] arguments)
    where Sort : IBqlSortColumn
```

Parameters:

- `searchValues`

  The values of fields referenced in `Sort` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Examples:

The code below searches the data view for all data records whose `TranClass` field has the *G* value.

```
// Data view definition in a graph
public PXSelect<GLTran,
    Where<GLTran.module, Equal<Current<Batch.module>>,
        And<GLTran.batchNbr, Equal<Current<Batch.batchNbr>>>>> Trans;
...
// Code in some method
foreach(GLTran tran in
            Trans.SearchAll<Asc<GLTran.tranClass>>(new object [] {"G"}))
    ...
```

SearchWindowed<Sort>(object[], int, int, params object[])

Retrieves the specified number of contiguous data records starting from the given position in the filtered data set. The fields are specified in the type parameter. The method extends the BQL statement with filtering and ordering by the fields and requests the limited numer of data records.

Syntax:

```
public virtual PXResultset<Table> SearchWindowed<Sort>(
    object[] searchValues, int startRow, int totalRows,
    params object[] arguments)
    where Sort : IBqlSortColumn
```

Parameters:

- `searchValues`

  The values of fields referenced in `Sort` by which the data set is filtered and sorted.

- `startRow`

  The 0-based index of the first data record to retrieve.

- `totalRows`

  The number of data records to retrieve.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Examples:

The code below retrieves the first five data records whose `TranClass` field has the *G* value from the data view.

```
// Data view definition in a graph
public PXSelect<GLTran,
    Where<GLTran.module, Equal<Current<Batch.module>>,
        And<GLTran.batchNbr, Equal<Current<Batch.batchNbr>>>>> Trans;
...
// Code in some method
PXResultset<GLTran> res =
```

```
    Trans.SearchWindowed<Asc<GLTran.tranClass>>(new object [] {"G"}, 0, 5);
```

Select(params object[])

Executes the BQL statement and retrieves all matching data records.

Syntax:

```
public virtual PXResultset<Table> Select(params object[] arguments)
```

Parameters:

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

SelectSingle(params object[])

Retrieves the top data record of the data set that corresponds to the BQL statement.

Syntax:

```
public virtual Table SelectSingle(params object[] arguments)
```

Parameters:

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

SelectWindowed(int, int, params object[])

Retrieves the specified number of data records starting from the given position.

Syntax:

```
public virtual PXResultset<Table> SelectWindowed(int startRow, int totalRows,
                                                 params object[] arguments)
```

Parameters:

- `startRow`

  The 0-based index of the first data record to retrieve.

- `totalRows`

  The number of data records to retrieve.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Examples:

The code below retrieves the first data record from the data set that corresponds to the BQL statement.

```
// Initializing the data view
PXSelectBase<FinPeriod> select = new PXSelect<FinPeriod,
    Where<FinPeriod.finYear, Equal<Required<FinPeriod.finYear>>>,
    OrderBy<Asc<FinPeriod.periodNbr>>>(sender.Graph);

// Executing the data view
```

```
FinPeriod fp = select.SelectWindowed(0, 1, DateTime.Now.Year);
```

In the third parameter, the method provides the value for the `Requried` parameter.

SetValueExt<Field>(Table, object)

Sets the value of the specified field in the given data record. The method relies on the *SetValueExt<Field>(Table, object)* method of the cache.

Syntax:

```
public virtual void SetValueExt<Field>(Table row, object value)
    where Field : IBqlField
```

Parameters:

- `row`

  The data record whose field value is set.

- `value`

  The value to set to the field.

Update(Table)

Updates the data record in the cache by invoking the *Update(object)* method on the cache. Returns the updated data record.

Syntax:

```
public virtual Table Update(Table item)
```

Parameters:

- `item`

  The updated version of the data record.

WhereAnd<TWhere>()

Appends a filtering expression to the BQL statement via the logical "and". The additional filtering expression is provided in the type parameter.

Syntax:

```
public void WhereAnd<TWhere>()
    where TWhere : IBqlWhere, new()
```

Examples:

The code below appends additional comparison to the BQL statement when the corresponding field in the filter is set to a value.

```
// Initializing the data view
PXSelectBase<APDocumentResult> sel = new PXSelect<APDocumentResult,
    Where<APRegister.vendorID, Equal<Current<APDocumentFilter.vendorID>>>,
    OrderBy<Desc<APDocumentResult.docDate>>>>(this);

// Checking whether a filter object has a value in the BranchID field
if (Filter.Current.BranchID != null)
{
    // Extending the Where clause with additional condition
    sel.WhereAnd<Where<APRegister.branchID,
                     Equal<Current<APDocumentFilter.branchID>>>>();
}
```

WhereNew<newWhere>()

Replaces the filtering expression in the BQL statement. The new filtering expression is provided in the type parameter.

Syntax:

```
public void WhereNew<newWhere>()
    where newWhere : IBqlWhere, new()
```

Examples:

The code below replaces the `Where` clause in a data view

```
// Defining the data view in a graph
public PXSelect<ARInvoice,
    Where<ARInvoice.docType, Equal<Current<ARInvoice.docType>>,
        And2<Where<ARInvoice.origModule, Equal<BatchModule.moduleAR>,
                 Or<ARInvoice.released, Equal<True>>>>>> Document;
...
// Replacing the Where clause
Document.WhereNew<
    Where<ARInvoice.docType, Equal<Required<ARInvoice.docType>>>>();

// Getting an ARInvoice data record
ARInvoice ardoc = (ARInvoice)resultsetRecord;

// Executing the modified data view
Document.Select(ardoc.DocType);
```

WhereNot()

Adds logical "not" to the whole `Where` clause of the BQL statement, reversing the condition to the opposite.

Syntax:

```
public void WhereNot()
```

WhereOr<TWhere>()

Appends a filtering expression to the BQL statement via the logical "or". The additional filtering expression is provided in the type parameter.

Syntax:

```
public void WhereOr<TWhere>()
    where TWhere : IBqlWhere, new()
```

WebDialogResult Enumeration

Defines values that indicate which button the user cliked in the dialog opened by the `Ask()` method.

Members

- `None`

  None of the buttons was clicked

- `OK`

  The user clicked **OK**

- `Cancel`

  The user clicked **Cancel**

- `Abort`

  The user clicked **Abort**
- `Retry`

  The user clicked **Retry**
- `Ignore`

  The user clicked **Ignore**
- `Yes`

  The user clicked **Yes**
- `No`

  The user clicked **No**

MessageButtons Enumeration

Defines possible sets of standard buttons that can be displayed in a dialog window created by the `Ask()` method.

Members

- `OK`

  Only the **OK** button is displayed.
- `OKCancel`

  The **OK** and **Cancel** buttons are displayed.
- `AbortRetryIgnore`

  The **Abort**, **Retry**, and **Ignore** buttons are displayed.
- `YesNoCancel`

  The **Yes**, **No**, and **Cancel** buttons are displayed.
- `YesNo`

  The **Yes** and **No** buttons are displayed.
- `RetryCancel`

  The **Retry** and **Cancel** buttons are displayed.
- `None`

  No buttons are displayed.

MessageIcon Enumeration

Defines possible icons that can be displayed beside the message in the dialog window opened by the `Ask()` method.

Members

- `None`

  No icon is displayed.
- `Error`

  The error sign is displayed.
- `Question`

The question mark sign is displayed.

- `Warning`

  The warning sign is displayed.

- `Information`

  The information sign is displayed.

PXSelect<Table> Class

Defines a data view for retrieving a particular data set from the database and provides the interface to the cache for inserting, updating, and deleting the data records.

See *Remarks* for more details and *Examples* for examples of usage.

Inheritance Hierarchy

```
PXSelectBase<Table>
```

Syntax

```
public class PXSelect<Table> : PXSelectBase<Table>
    where Table : class, IBqlTable, new()
```

There are *a number of other types* derived from `PXSelectBase<Table>` that are used in the same way and have exactly the same set of methods as `PXSelect<Table>` has, and only allow building more complex BQL expressions.

The `PXSelect` type exposes the following members.

Constructors

Constructor	Description
PXSelect(PXGraph)	Initializes a new instance of a data view bound to the specified graph.
PXSelect(PXGraph, Delegate)	Initializes a new instance of a data view that is bound to the specified graph and uses the provided method to retrieve data.

Methods

Method	Description
Clear(PXGraph)	Clears the results of BQL statement execution stored in the provided graph
GetCommand()	Returns the `BqlCommand` object representing the BLQ statement
Search<Field0>(PXGraph, object, params object[])	Searches for a data record by the value of specified field in the data set that corresponds to the BQL statement
Search<Field0, Field1>(PXGraph, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement

Method	Description
Search<Field0, Field1, Field2>(PXGraph, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3>(PXGraph, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4>(PXGraph, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5>(PXGraph, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6>(PXGraph, object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7>(PXGraph, object, object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8>(PXGraph, object, object, object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8, Field9>(PXGraph, object, object, object, object, object, object, object, object, object, object, params object[])	Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement
SearchAll<Sort>(PXGraph, object[], params object[])	Searches the data set that corresponds to the BQL statement for all data records whose fields have the specified values
SearchAll<Resultset, Sort>(PXGraph, object[], params object[])	Searches the data set that corresponds to the BQL statement for all data records whose fields have the specified values
SearchWindowed<Resultset, Sort>(PXGraph, object[], int, int, params object[])	Searches the data set that corresponds to the BQL statement for the data records whose fields have the specified values
Select(PXGraph, params object[])	Executes the BQL statement and retrieves all matching data records
Select<Resultset>(PXGraph, params object[])	Executes the BQL statement and retrieves all matching data records
SelectMultiBound(PXGraph, object[], params object[])	Executes the BQL statement with the specified values to substitute current object and retrieves all matching data records

Method	Description
SelectWindowed(PXGraph, int, int, params object[])	Retrieves the specified number of data records starting from the given position
SelectWindowed<Resultset>(PXGraph, int, int, params object[])	Retrieves the specified number of data records starting from the given position
StoreCached(PXGraph, PXCommandKey, List<object>)	Stores in the caches the results of BQL statement execution

Remarks

A `PXSelect<Table>` object wraps the `Select<Table>` object, which represents the BQL command, and the *PXView* object, which executes this BQL command. The `PXSelect<Table>` object also holds the reference of the *cache* of the `Table` data records and the graph.

The `PXSelect<Table>` type provides interfaces to both the `PXView` object and the cache. So you can execute the underlying BQL command and invoke cache methods through the methods of the `PXSelect<Table>`.

Examples

The code below shows the declaration of a data view in a graph and execution of this data view.

```
public class VendorClassMaint : PXGraph<VendorClassMaint>
{
    public PXSelect<Vendor,
       Where<Vendor.vendorClassID, Equal<Current<VendorClass.vendorClassID>>>>
           Vendors;
    ...
    public void SomeMethod()
    {
        // Data view execution
        foreach (Vendor vend in Vendors.Select())
            ...
    }
}
```

Note that the data view is not initialized. The graph initializes it automatically.

Suppose the following data view is defined in a graph. This data view cannot be used as the data member of a webpage control, because the BQL expression includes the `Required` parameter.

```
public PXSelect<ARPayment,
    Where<ARPayment.refNbr, Equal<Required<ARPayment.refNbr>>>> arPayment;
```

The code below executes this data view, selects the top data record, and initializes a new data record with values from the retrieved data record.

```
// Execute the data view
ARPayment rec = arPayment.SelectSingle(refNbrValue);

// Create a new data record
ARPayment payment = new ARPayment();
payment.CustomerID = rec.CustomerID;

// Insert the new data record into the cache of ARPayment data records
arPayment.Insert(payment);
```

See *Executing Statements* for more examples of BQL statements execution.

PXSelect<Table> Constructors

The *PXSelect<Table>* type exposes the following constructors.

PXSelect(PXGraph)

Initializes a new instance of a data view bound to the specified graph.

Syntax:

```
public PXSelect(PXGraph graph)
```

Parameters:

- `graph`

  The graph with which the data view is associated.

PXSelect(PXGraph, Delegate)

Initializes a new instance of a data view that is bound to the specified graph and uses the provided method to retrieve data.

Syntax:

```
public PXSelect(PXGraph graph, Delegate handler)
```

Parameters:

- `graph`

  The graph with which the data view is associated.

- `handler`

  The delegate of the method that is used to retrieve the data from the database (or other source). This method is invoked when one of the `Select()` methods is called.

Examples

The code below shows declaration of a data view in a graph. The data view is not initialized explicitly. The graph automatically initializes the data view.

```
public class MyGraph : PXGraph<MyGraph>
{
    public PXSelect<MyDAC> Records;
    ...
}
```

The code below shows declaration of a data view that have the optional method.

```
public class MyGraph : PXGraph<MyGraph>
{
    public PXSelect<MyDAC> Records;
    protected IEnumerable records()
    {
        ...
    }
    ...
}
```

The code below shows explicit initialization of a data view in code in a graph.

```
PXSelectBase<MyDAC> records = new PXSelect<MyDAC,
    Where<MyDAC.field1, IsNotNull>>(this);
```

PXSelect<Table> Methods

The *PXSelect<Table>* type exposes the following methods.

Clear(PXGraph)

Clears the results of BQL statement execution stored in the provided graph.

Syntax:

```
public static void Clear(PXGraph graph)
```

Parameters:

- `graph`

  The graph where the data is cleared.

Examples:

The code below clears the query cache to load the records directly from the database (the data records are still merged with the modifications stored in the `PXCache` object).

```
// Clearing the query cache
PXSelect<CRMergeCriteria,
    Where<CRMergeCriteria.mergeID, Equal<Required<CRMerge.mergeID>>>>.
    Clear(this);

// Selecting data records directly from the database (not from the query
// cache) and merging with the PXCache<> object
foreach (CRMergeCriteria item in
     PXSelect<CRMergeCriteria,
         Where<CRMergeCriteria.mergeID, Equal<Required<CRMerge.mergeID>>>>.
         Select(this, document.MergeID))
{
    Criteria.Cache.Delete(item);
}
```

GetCommand()

Returns the `BqlCommand` object representing the BLQ statement.

Syntax:

```
public static BqlCommand GetCommand()
```

Search<Field0>(PXGraph, object, params object[])

Searches for a data record by the value of specified field in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified field and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0>(
    PXGraph graph, object field0, params object[] arguments)
    where Field0 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0`

  The value of `Field0` by which the data set is filtered and sorted.

- `arguments`

The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1>(PXGraph, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1>(
    PXGraph graph, object field0, object field1, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0 - field1`

  The values of `Field0` and `Field1` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Examples:

The code below checks whether a duplicate of the `APInvoice` data record exists by searching by the key fields.

```
APInvoice duplicate = PXSelect<APInvoice>.
    Search<APInvoice.docType, APInvoice.refNbr>(
        this, invoice.DocType, invoice.OrigRefNbr);

// If the data record exists, throw an exception
if (duplicate != null)
    throw new PXException(ErrorMessages.RecordExists);
```

Search<Field0, Field1, Field2>(PXGraph, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2>(
    PXGraph graph, object field0, object field1,
    object field2,params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
```

Parameters:

- `graph`

The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0 - field2`

  The values of `Field0`-`Field2` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3>(PXGraph, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2, Field3>(
    PXGraph graph, object field0, object field1, object field2,
    object field3, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0 - field3`

  The values of `Field0`-`Field3` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4>(PXGraph, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2,
                                          Field3, Field4>(
    PXGraph graph, object field0, object field1, object field2,
    object field3, object field4, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0 - field4`

  The values of `Field0`-`Field4` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5>(PXGraph, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2,
                                        Field3, Field4, Field5>(
    PXGraph graph, object field0, object field1, object field2,
    object field3, object field4, object field5, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0 - field5`

  The values of `Field0`-`Field5` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6>(PXGraph, object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2, Field3,
                                        Field4, Field5, Field6>(
    PXGraph graph, object field0, object field1, object field2, object field3,
    object field4, object field5, object field6, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
```

```
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0 - field6`

  The values of `Field0`-`Field6` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7>(PXGraph, object, object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2, Field3,
                                    Field4, Field5, Field6, Field7>(
    PXGraph graph, object field0, object field1, object field2,
    object field3, object field4, object field5, object field6,
    object field7, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
    where Field7 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `field0 - field7`

  The values of `Field0`-`Field7` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8>(PXGraph, object, object, object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2,
                                        Field3, Field4, Field5,
                                        Field6, Field7, Field8>(
    PXGraph graph, object field0, object field1, object field2,
    object field3, object field4, object field5, object field6,
    object field7, object field8, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
    where Field7 : IBqlField
    where Field8 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.
- `field0 - field8`

  The values of `Field0`-`Field8` by which the data set is filtered and sorted.
- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Search<Field0, Field1, Field2, Field3, Field4, Field5, Field6, Field7, Field8, Field9>(PXGraph, object, object, object, object, object, object, object, object, object, object, params object[])

Searches for a data record by the values of specified fields in the data set that corresponds to the BQL statement. The method extends the BQL statement with filtering and ordering by the specified fields and retrieves the top data record.

Syntax:

```
public static PXResultset<Table> Search<Field0, Field1, Field2, Field3,
                                        Field4, Field5, Field6, Field7,
                                        Field8, Field9>(
    PXGraph graph, object field0, object field1, object field2,
    object field3, object field4, object field5, object field6,
    object field7, object field8, object field9, params object[] arguments)
    where Field0 : IBqlField
    where Field1 : IBqlField
    where Field2 : IBqlField
    where Field3 : IBqlField
    where Field4 : IBqlField
    where Field5 : IBqlField
    where Field6 : IBqlField
    where Field7 : IBqlField
    where Field8 : IBqlField
    where Field9 : IBqlField
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.
- `field0 - field9`

The values of `Field0`-`Field9` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

SearchAll<Sort>(PXGraph, object[], params object[])

Searches the data set that corresponds to the BQL statement for all data records whose fields have the specified values. The fields are specified in the type parameter. The method extends the BQL statement with filtering and ordering by the fields and retrieves all data records from the resulting data set.

Syntax:

```
public static PXResultset<Table> SearchAll<Sort>(PXGraph graph,
                                                     object[] searchValues,
                                                     params object[] pars)
    where Sort : IBqlSortColumn
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `searchValues`

  The values of fields referenced in `Sort` by which the data set is filtered and sorted.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

SearchAll<Resultset, Sort>(PXGraph, object[], params object[])

Searches the data set that corresponds to the BQL statement for all data records whose fields have the specified values.

The fields are specified in the `Sort` type parameter. The method extends the BQL statement with filtering and ordering by the fields and retrieves all data records from the resulting data set. A specific `PXResultset<>` type can be specified in the `Resultset` type parameter.

Syntax:

```
public static Resultset SearchAll<Resultset, Sort>(PXGraph graph,
                                                     object[] searchValues,
                                                     params object[] pars)
    where Resultset : PXResultset<Table>, new()
    where Sort : IBqlSortColumn
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `searchValues`

  The values of fields referenced in `Sort` by which the data set is filtered and sorted.

- `arguments`

The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

SearchWindowed<Resultset, Sort>(PXGraph, object[], int, int, params object[])

Searches the data set that corresponds to the BQL statement for the data records whose fields have the specified values. Retrieves the specified number of such data records starting from the given position.

The fields are specified in the `Sort` type parameter. The method extends the BQL statement with filtering and ordering by the fields and retrieves all data records from the resulting data set. A specific `PXResultset<>` type can be specified in the `Resultset` type parameter.

Syntax:

```
public static Resultset SearchWindowed<Resultset, Sort>(
    PXGraph graph, object[] searchValues,
    int startRow, int totalRows, params object[] pars)
    where Resultset : PXResultset<Table>, new()
    where Sort : IBqlSortColumn
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `searchValues`

  The values of fields referenced in `Sort` by which the data set is filtered and sorted.

- `startRow`

  The 0-based index of the first data record to retrieve.

- `totalRows`

  The number of data records to retrieve.

- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Select(PXGraph, params object[])

Executes the BQL statement and retrieves all matching data records.

Syntax:

```
public static PXResultset<Table> Select(PXGraph graph,
                                          params object[] pars)
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `pars`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

Select<Resultset>(PXGraph, params object[])

Executes the BQL statement and retrieves all matching data records. A specific PXResultset<> type can be specified in the type parameter. To wrap the retrieved data records, the non-generic Select() method uses the PXResultset<Table> type, where Table is the first DAC specified in the BQL statement.

Syntax:

```
public static Resultset Select<Resultset>(PXGraph graph, params object[] pars)
    where Resultset : PXResultset<Table>, new()
```

Parameters:

- graph

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- pars

  The values to substitute BQL parameters, such as Optional, Required, and Argument, in the BQL statement.

SelectMultiBound(PXGraph, object[], params object[])

Executes the BQL statement with the specified values to substitute current object and retrieves all matching data records.

Syntax:

```
public static PXResultset<Table> SelectMultiBound(
    PXGraph graph, object[] currents, params object[] pars)
```

Parameters:

- graph

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- currents

  The objects to be used instead of the data records referenced by the Current property of the caches.

- pars

  The values to substitute BQL parameters, such as Optional, Required, and Argument, in the BQL statement.

SelectWindowed(PXGraph, int, int, params object[])

Retrieves the specified number of data records starting from the given position.

Syntax:

```
public static PXResultset<Table> SelectWindowed(
    PXGraph graph, int startRow, int totalRows, params object[] pars)
```

Parameters:

- graph

  The graph that is used to cache the retrieved data record and merge them with the modified data records.

- `startRow`

  The 0-based index of the first data record to retrieve.
- `totalRows`

  The number of data records to retrieve.
- `arguments`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

SelectWindowed<Resultset>(PXGraph, int, int, params object[])

Retrieves the specified number of data records starting from the given position. A specific `PXResultset<>` type can be specified in the type parameter.

Syntax:

```
public static Resultset SelectWindowed<Resultset>(
    PXGraph graph, int startRow, int totalRows, params object[] pars)
    where Resultset : PXResultset<Table>, new()
```

Parameters:

- `graph`

  The graph that is used to cache the retrieved data record and merge them with the modified data records.
- `startRow`

  The 0-based index of the first data record to retrieve.
- `totalRows`

  The number of data records to retrieve.
- `pars`

  The values to substitute BQL parameters, such as `Optional`, `Required`, and `Argument`, in the BQL statement.

StoreCached(PXGraph, PXCommandKey, List<object>)

Stores in the caches the results of BQL statement execution.

Syntax:

```
public static void StoreCached(PXGraph graph, PXCommandKey queryKey,
                               List<object> records)
```

Parameters:

- `graph`

  The graph object whose caches are used to store the data records.
- `queryKey`
- `records`

PXProcessing<Table> Class

Defines a special data view used on processing webpages, which are intended for mass processing of data records.

The `PXProcessing<Table>` type is used to define the data view in a graph bound to a processing webpage. A data view of this type includes definitions of two actions, `Process` and `ProcessAll`, which are added to the graph and are used to invoke the processing. You should set the processing method by invoking one of the *SetProcessDelegate(...)* methods in the constructor of the graph.

Inheritance Hierarchy

```
PXSelectBase<Table>
```

Syntax

```
public class PXProcessing<Table> : PXSelectBase<Table>, IPXProcessing,
 IPXProcessingWithCustomDelegate
    where Table : class, IBqlTable, new(),
```

The `PXProcessing<Table>` type exposes the following members.

Constructors

Constructor	Description
PXProcessing(PXGraph)	Initializes a new instance of a data view bound to the specified graph.
PXProcessing(PXGraph, Delegate)	Initializes a new instance of a data view that is bound to the specified graph and uses the provided method to retrieve data.

Properties

- `public virtual Delegate` **`CustomViewDelegate`**

  Gets or sets the delegate of the method that retrieves the data (the optional method of the data view).

Delegates

The `PXProcessing<Table>` type defines the following delegates, which may be passed to `SetProcessDelegate(...)` methods.

- `public delegate void ProcessListDelegate(List<Table> list);`

  The delegate of the method for processing a list of data records.

- `public delegate void ProcessItemDelegate(Table item);`

  The delegate of the method for processing a single data record.

- `public delegate void ProcessItemDelegate<Graph>(Graph graph, Table item) where Graph : PXGraph, new();`

  The delegate of the method for processing a single data record. The delegate allows you to receive the same instance of the provided graph type to each invocation of the processing method during the processing operation.

- `public delegate void FinallyProcesselegate<Graph>(Graph graph) where Graph : PXGraph, new();`

  The delegate of the method that is executed after all data records are processed. In the parameter, the method receives the graph that was passed to each invocation of the data record processing method during the processing operation.

Methods

Method	Description
GetProcessDelegate()	Returns the delegate of the processing method, which is set by one of the *SetProcessDelegate()* methods
Join<join>()	Appends the join clause to the underlying BQL command
OrderByNew<newOrderBy>()	Replaces the sorting expression in the underlying BQL command
SetAutoPersist(bool)	Sets the value that indicates whether the changes in the graph should be automatically saved in the database before the data records are processed
SetCurrentItem(Table)	Sets the current data record to process
SetError(string)	Sets the provided string as the error message of the processing operation
SetError(Exception)	Sets the provided exception as the error of the processing operation
SetError(int, string)	Sets the error message on the data record with the specified index
SetError(int, Exception)	Sets the provided exception as the error on the data record with the specified index
SetInfo(string)	Sets the information message for the processing operation
SetInfo(Exception)	Sets the provided exception as the information-level error for the processing operation
SetInfo(int, string)	Attaches the provided information message to the data record with the specified index
SetInfo(int, Exception)	Attaches the provided exception as the information-level error to the data record with the specified index
SetProcessAllCaption(string)	Sets the display name of the button that processes all data records selected by the data view
SetProcessAllEnabled(bool)	Enables or disables the button that processes all data records selected by the data view
SetProcessAllTooltip(string)	Sets the tooltip for the button that processes all data records selected by the data view
SetProcessAllVisible(bool)	Displays or hides the button that processes all data records selected by the data view
SetProcessCaption(string)	Sets the display name of the button that processes the selected data records
SetProcessDelegate(ProcessListDelegate)	Sets the method that is invoked to process multiple data records
SetProcessDelegate(ProcessItemDelegate)	Sets the method that is invoked to process each data record

Method	Description
SetProcessDelegate<Graph>(ProcessItemDelegate<Graph>)	Sets the method that is invoked to process each data record
SetProcessDelegate<Graph>(ProcessItemDelegate<Graph>, FinallyProcesselegate<Graph>)	Sets the method that is invoked to process each data record and the method that is invoked after all data records are processed
SetProcessEnabled(bool)	Enables or disables the button that processes the selected data records
SetProcessTooltip(string)	Sets the tooltip for the button that processes the selected data records
SetProcessVisible(bool)	Displays or hides the button that processes the selected data records
SetProcessed()	Sets the information message confirming that a data record has been processed successfully
SetSelected<Field>()	Sets the DAC field by which the user can mark data records that should be processed
SetWarning(string)	Sets the warning message for the processing operation
SetWarning(Exception)	Sets the provided exceptiona as the warning-level error of the processing operation
SetWarning(int, string)	Sets the warning message on the data record with the specified index
SetWarning(int, Exception)	Attaches the provided exception as the warning-level error to the data record with the specified index

The following classes derive from PXProcessing<Table>. These classes expose exactly the same members as PXProcessing<Table> and serve only for specifying more complex BQL expressions.

PXProcessing<Table, Where> Class

Selects data records from one table filtered by the expression set in Where.

Syntax:

```
public class PXProcessing<Table, Where> : PXProcessing<Table>
    where Table : class, IBqlTable, new()
    where Where : IBqlWhere, new()
```

PXProcessing<Table, Where, OrderBy> Class

Selects data records from one table filtered by the expression set in Where and ordered by the fields specified in OrderBy.

Syntax:

```
public class PXProcessing<Table, Where, OrderBy> : PXProcessing<Table, Where>
    where Table : class, IBqlTable, new()
    where Where : IBqlWhere, new()
    where OrderBy : IBqlOrderBy, new()
```

PXProcessingJoin<Table, Join> Class

Selects data records from multiple tables linked by the Join clause.

Syntax:

```
public class PXProcessingJoin<Table, Join> : PXProcessing<Table>
    where Table : class, IBqlTable, new()
    where Join : IBqlJoin, new()
```

PXProcessingJoin<Table, Join, Where> Class

Selects data records from multiple tables linked by the `Join` clause and filtered according to the expression set in `Where`.

Syntax:

```
public class PXProcessingJoin<Table, Join, Where> : PXProcessingJoin<Table, Join>
    where Table : class, IBqlTable, new()
    where Join : IBqlJoin, new()
    where Where : IBqlWhere, new()
```

PXProcessingJoin<Table, Join, Where, OrderBy> Class

Selects data records from multiple tables linked by the `Join` clause, filtered according to the expression set in `Where`, and ordered by the fields specified in `OrderBy`.

Syntax:

```
public class PXProcessingJoin<Table, Join, Where, OrderBy> : PXProcessingJoin<Table,
 Join, Where>
    where Table : class, IBqlTable, new()
    where Join : IBqlJoin, new()
    where Where : IBqlWhere, new()
    where OrderBy : IBqlOrderBy, new()
```

PXFilteredProcessing<Table, FilterTable> Class

Selects data records from one table and applies the user filter.

Syntax:

```
public class PXFilteredProcessing<Table, FilterTable> : PXProcessing<Table>
    where FilterTable : class, IBqlTable, new()
    where Table : class, IBqlTable, new()
```

PXFilteredProcessing<Table, FilterTable, Where> Class

Selects data records from one table filtered by the expression set in `Where` and applies the user filter.

Syntax:

```
public class PXFilteredProcessing<Table, FilterTable, Where> :
 PXFilteredProcessing<Table, FilterTable>
    where FilterTable : class, IBqlTable, new()
    where Table : class, IBqlTable, new()
    where Where : IBqlWhere, new()
```

PXFilteredProcessing<Table, FilterTable, Where, OrderBy> Class

Selects data records from one table filtered by the expression set in `Where` and ordered by the fields specified in `OrderBy` and applies the user filter.

Syntax:

```
public class PXFilteredProcessing<Table, FilterTable, Where, OrderBy> :
 PXFilteredProcessing<Table, FilterTable, Where>
```

```
    where FilterTable : class, IBqlTable, new()
    where Table : class, IBqlTable, new()
    where Where : IBqlWhere, new()
    where OrderBy : IBqlOrderBy, new()
```

PXFilteredProcessingJoin<Table, FilterTable, Join> Class

Selects data records from multiple tables linked by the `Join` clause and applies the user filter.

Syntax:

```
public class PXFilteredProcessingJoin<Table, FilterTable, Join> :
 PXFilteredProcessing<Table, FilterTable>
    where FilterTable : class, IBqlTable, new()
    where Table : class, IBqlTable, new()
    where Join : IBqlJoin, new()
```

PXFilteredProcessingJoin<Table, FilterTable, Join, Where> Class

Selects data records from multiple tables linked by the `Join` clause and filtered according to the expression set in `Where` and applies the user filter.

Syntax:

```
public class PXFilteredProcessingJoin<Table, FilterTable, Join, Where> :
 PXFilteredProcessingJoin<Table, FilterTable, Join>
    where FilterTable : class, IBqlTable, new()
    where Table : class, IBqlTable, new()
    where Join : IBqlJoin, new()
    where Where : IBqlWhere, new()
```

PXFilteredProcessingJoin<Table, FilterTable, Join, Where, OrderBy> Class

Selects data records from multiple tables linked by the `Join` clause, filtered according to the expression set in `Where`, and ordered by the fields specified in `OrderBy` and applies the user filter.

Syntax:

```
public class PXFilteredProcessingJoin<Table, FilterTable, Join, Where, OrderBy> :
 PXFilteredProcessingJoin<Table, FilterTable, Join>
    where FilterTable : class, IBqlTable, new()
    where Table : class, IBqlTable, new()
    where Join : IBqlJoin, new()
    where Where : IBqlWhere, new()
    where OrderBy : IBqlOrderBy, new()
```

PXFilteredProcessingJoinGroupBy<Table, FilterTable, Join, Where, Aggregate> Class

Selects aggregated data records from multiple tables linked by the `Join` clause, filtered according to the expression set in `Where`, and ordered by the fields specified in `OrderBy` and applies the user filter.

Syntax:

```
public class PXFilteredProcessingJoinGroupBy<Table, FilterTable, Join, Where,
 Aggregate> : PXFilteredProcessingJoin<Table, FilterTable, Join>
    where FilterTable : class, IBqlTable, new()
    where Table : class, IBqlTable, new()
    where Join : IBqlJoin, new()
    where Where : IBqlWhere, new()
    where Aggregate : IBqlAggregate, new()
```

Examples

The code below shows definition of the graph that contains the processing data view.

```
public class ARPaymentsProcessing : PXGraph<ARPaymentsProcessing>
{
    // Definition of the data view to process
    public PXProcessing<ARPaymentInfo> ARDocumentList;

    // The constructor of the graph
    public ARPaymentsAutoProcessing()
    {
        // Specifying the field to mark data records for processing
        ARDocumentList.SetSelected<ARPaymentInfo.selected>();
        // Setting the processing method
        ARDocumentList.SetProcessDelegate(Process);
    }

    // The processing method (must be static)
    public static void Process(List<ARPaymentInfo> products)
    {
        ...
    }
    ...
}
```

PXProcessing<Table> Constructors

The *PXProcessing<Table>* type exposes the following constructors.

PXProcessing(PXGraph)

Initializes a new instance of a data view bound to the specified graph.

Syntax:

```
public PXProcessing(PXGraph graph) : this(graph, null)
```

Parameters:

- `graph`

  The graph with which the data view is associated.

PXProcessing(PXGraph, Delegate)

Initializes a new instance of a data view that is bound to the specified graph and uses the provided method to retrieve data.

Syntax:

```
public PXProcessing(PXGraph graph, Delegate handler)
```

Parameters:

- `graph`

  The graph with which the data view is associated.

- `handler`

  The delegate of the method that is used to retrieve the data from the database (or other source).

PXProcessing<Table> Methods

The *PXProcessing<Table>* type exposes the following methods.

GetProcessDelegate()

Returns the delegate of the processing method, which is set by one of the *SetProcessDelegate()* methods.

Syntax:

```
public Delegate GetProcessDelegate()
```

Join<join>()

Appends the join clause to the underlying BQL command.

Syntax:

```
public override void Join<join>()
```

OrderByNew<newOrderBy>()

Replaces the sorting expression in the underlying BQL command.

Syntax:

```
public override void OrderByNew<newOrderBy>()
```

SetAutoPersist(bool)

Sets the value that indicates whether the changes in the graph should be automatically saved in the database before the data records are processed. By default, the changes are not saved automatically.

Syntax:

```
public virtual void SetAutoPersist(bool autoPersist)
```

Parameters:

- `autoPersist`

  The value indicating whether to save the changes.

SetCurrentItem(Table)

Sets the current data record to process.

Syntax:

```
public static void SetCurrentItem(Table currentItem)
```

Parameters:

- `currentItem`

  The data record to be set as the current.

SetError(string)

Sets the provided string as the error message of the processing operation.

Syntax:

```
public static bool SetError(string message)
```

Parameters:

- `message`

  The error message.

SetError(Exception)

Sets the provided exception as the error of the processing operation.

Syntax:

```
public static bool SetError(Exception e)
```

Parameters:

- `e`

  The exception containing information about the error.

SetError(int, string)

Sets the error message on the data record with the specified index.

Syntax:

```
public static bool SetError(int index, string message)
```

Parameters:

- `index`

  The index of the data record marked with error.

- `message`

  The error message.

SetError(int, Exception)

Sets the provided exception as the error on the data record with the specified index.

Syntax:

```
public static bool SetError(int index, Exception e)
```

Parameters:

- `index`

  The index of the data record marked with error.

- `e`

  The exception containing information about the error.

SetInfo(string)

Sets the information message for the processing operation.

Syntax:

```
public static bool SetInfo(string message)
```

Parameters:

- `message`

  The information message.

SetInfo(Exception)

Sets the provided exception as the information-level error for the processing operation.

Syntax:

```
public static bool SetInfo(Exception e)
```

Parameters:

- `e`

  The exception containing information.

SetInfo(int, string)

Attaches the provided information message to the data record with the specified index.

Syntax:

```
public static bool SetInfo(int index, string message)
```

Parameters:

- `index`

  The index of the data record to which the message is attached.

- `message`

  The information message.

SetInfo(int, Exception)

Attaches the provided exception as the information-level error to the data record with the specified index.

Syntax:

```
public static bool SetInfo(int index, Exception e)
```

Parameters:

- `index`

  The index of the data record that is marked with the exception.

- `e`

  The exception containing information.

SetProcessAllCaption(string)

Sets the display name of the button that processes all data records selected by the data view.

Syntax:

```
public virtual void SetProcessAllCaption(string caption)
```

Parameters:

- `caption`

  The string used as the display name.

SetProcessAllEnabled(bool)

Enables or disables the button that processes all data records selected by the data view.

Syntax:

```
public virtual void SetProcessAllEnabled(bool enabled)
```

Parameters:

- `enabled`

  The value indicating whether the button is enalbed.

SetProcessAllTooltip(string)

Sets the tooltip for the button that processes all data records selected by the data view.

Syntax:

```
public virtual void SetProcessAllTooltip(string tooltip)
```

Parameters:

- `tooltip`

  The string used as the tooltip.

SetProcessAllVisible(bool)

Displays or hides the button that processes all data records selected by the data view.

Syntax:

```
public virtual void SetProcessAllVisible(bool visible)
```

Parameters:

- `visible`

  The value indicating whether the button is visible.

SetProcessCaption(string)

Sets the display name of the button that processes the selected data records.

Syntax:

```
public virtual void SetProcessCaption(string caption)
```

Parameters:

- `caption`

  The string used as the display name.

SetProcessDelegate(ProcessListDelegate)

Sets the method that is invoked to process multiple data records.

The method receives the list of the data records to process in the parameter. Depending on the buttion the user clicked to start processing, the data records are either the data records selected by the user in the grid or all data records selected by the data view.

Syntax:

```
public virtual void SetProcessDelegate(ProcessListDelegate handler)
```

Parameters:

- `handler`

  The delegate of the processing method.

Examples:

The code below sets the processing method for a processing data view in a graph.

```
// Definition of the processing data view
public PXProcessingJoin<BalancedAPDocument, ... > APDocumentList;
...

// The constructor of the graph
public APDocumentRelease()
{
    ...
    // Setting the delegate of a processing method and defining the
    // processing method in place
    APDocumentList.SetProcessDelegate(
        delegate(List<BalancedAPDocument> list)
        {
            List<APRegister> newlist = new List<APRegister>(list.Count);
            foreach (BalancedAPDocument doc in list)
            {
                newlist.Add(doc);
            }
            ReleaseDoc(newlist, true);
        }
    );
}

// Definition of the method that does actual processing
public static void ReleaseDoc(List<APRegister> list, bool isMassProcess)
{
    ...
}
```

SetProcessDelegate(ProcessItemDelegate)

Sets the method that is invoked to process each data record.

The method receives the data records to process in the parameter. Depending on the buttion the user clicked to start processing, the method is invoked for each data record selected by the user in the grid, or for each data record selected by the data view.

Syntax:

```
public virtual void SetProcessDelegate(ProcessItemDelegate handler)
```

Parameters:

- `handler`

  The delegate of the processing method.

SetProcessDelegate<Graph>(ProcessItemDelegate<Graph>)

Sets the method that is invoked to process each data record.

The method should have two parameters, the graph and the data record. When the user initiates processing, the data view initializes the instance of the specified graph type and passes it to the processing method while it is invoked for each data record.

Syntax:

```
public void SetProcessDelegate<Graph>(ProcessItemDelegate<Graph> handler)
    where Graph : PXGraph, new()
```

Parameters:

- `handler`

  The delegate of the processing method.

Examples:

The code below sets the processing method, which will process each data record, for a processing data view in a graph.

```
// Definition of the processing data view
public PXFilteredProcessing<ARPaymentInfo> ARDocumentList;
...
ARDocumentList.SetProcessDelegate<ARPaymentCCProcessing>(
    delegate(ARPaymentCCProcessing aGraph,ARPaymentInfo doc)
    {
        ProcessPayment(aGraph, doc);
    }
);
```

The `ProcessPayment(...)` should be the static method of the current graph.

SetProcessDelegate<Graph>(ProcessItemDelegate<Graph>, FinallyProcesselegate<Graph>)

Sets the method that is invoked to process each data record and the method that is invoked after all data records are processed.

The processing method should have two parameters, the graph and the data record. When the user initiates processing, the data view initializes the instance of the specified graph type and passes it to the processing method while it is invoked for each data record.

The second method has the only parameter, the graph. This method is invoked once when all data record are processed. The parameter of the method is set to the graph that was passed to the processing method for each data record.

Syntax:

```
public virtual void SetProcessDelegate<Graph>(
    ProcessItemDelegate<Graph> handler,
    FinallyProcesselegate<Graph> handlerFinally)
    where Graph : PXGraph, new()
```

Parameters:

- `handler`

  The delegate of the processing method.

- `handlerFinally`

  The delegate of the method invoked when all data records are processed.

SetProcessEnabled(bool)

Enables or disables the button that processes the selected data records.

Syntax:

```
public virtual void SetProcessEnabled(bool enabled)
```

Parameters:

- `enabled`

  The value indicating whether the button is enabled.

SetProcessTooltip(string)

Sets the tooltip for the button that processes the selected data records.

Syntax:

```
public virtual void SetProcessTooltip(string tooltip)
```

Parameters:

- `tooltip`

  The string used as the tooltip.

SetProcessVisible(bool)

Displays or hides the button that processes the selected data records.

Syntax:

```
public virtual void SetProcessVisible(bool visible)
```

Parameters:

- `visible`

  The value indicating whether the button is visible.

SetProcessed()

Sets the information message confirming that a data record has been processed successfully

Syntax:

```
public static bool SetProcessed()
```

SetSelected<Field>()

Sets the DAC field by which the user can mark data records that should be processed. The method enables this field and disabled all other fields.

Syntax:

```
public virtual void SetSelected<Field>()
    where Field : IBqlField
```

SetWarning(string)

Sets the warning message for the processing operation.

Syntax:

```
public static bool SetWarning(string message)
```

Parameters:

- `message`

  The warning message.

SetWarning(Exception)

Sets the provided exceptiona as the warning-level error of the processing operation.

Syntax:

```
public static bool SetWarning(Exception e)
```

Parameters:

- `e`

  The exception containing warning information.

SetWarning(int, string)

Sets the warning message on the data record with the specified index.

Syntax:

```
public static bool SetWarning(int index, string message)
```

Parameters:

- `index`

  The index of the data record to which the message is attached.

- `message`

  The warning message.

SetWarning(int, Exception)

Attaches the provided exception as the warning-level error to the data record with the specified index.

Syntax:

```
public static bool SetWarning(int index, Exception e)
```

Parameters:

- `index`

  The index of the data record to which the exception is attached.

- `e`

  The exception containing warning information.

PXGraph Class

The base type that defines the common interface of business logic controllers (graphs), which you should derive from either *PXGraph<TGraph>* or *PXGraph<TGraph, TPrimary>*.

Each webpage references a graph (through the `PXDatasource` control). An instance of this graph is created and destroyed on each user's request, while the modified data records are preserved between requests in the session.

Syntax

```
[System.Security.Permissions.ReflectionPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
[System.Security.Permissions.SecurityPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
[DebuggerTypeProxy(typeof(PXGraph.PXDebugView))]
public class PXGraph: IXmlSerializable
```

The `PXGraph` type exposes the following members.

Constructors

The `PXGraph` constructor is not called directly. To initialize a new instance of the `PXGraph` or `PXGraph<>` class, use the *CreateInstance<>()* method.

Classes that derive from `PXGraph<>` (graphs) can define their own constructors without parameters to perform layout configuration or configure background processing operations.

Properties

- `public AccessInfo` **`Accessinfo`**

  Get an instance of the `AccessInfo` DAC, which contains some application settings of the current user, such as the branch ID, user ID and name, webpage ID, and other settings. The fields of this DAC can be referenced in BQL statements through the `Current` parameter. For example, `Current<AccessInfo.branchID>`.

- `public object` **`UID`**

  Gets or sets the uniquer identifier that is used for setting up the processing operations.

- `public CultureInfo` **`Culture`**

  Gets or sets the culture information.

- `public byte[]` **`TimeStamp`**

  Gets or sets the value of the global timestamp.

- `public virtual bool` **`IsDirty`**

  Gets the value that indicates whether there are modified data records not saved to the database in the caches related to the graph data views. If the `IsDirty` property of at least one cache object is `true`, the `IsDirty` property of the graph is also `true`.

The following properties provide access to the collections of event handlers defined in the graph or added at run time:

- `public RowSelectingEvents` **`RowSelecting`**

  Gets the instance of *RowSelectingEvents* type that represents the collection of *RowSelecting* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowSelectedEvents` **`RowSelected`**

  Gets the instance of *RowSelectedEvents* type that represents the collection of *RowSelected* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowInsertingEvents` **`RowInserting`**

Gets the instance of *RowInsertingEvents* type that represents the collection of *RowInserting* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowInsertedEvents` **`RowInserted`**

  Gets the instance of *RowInsertedEvents* type that represents the collection of *RowInserted* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowUpdatingEvents` **`RowUpdating`**

  Gets the instance of *RowUpdatingEvents* type that represents the collection of *RowUpdating* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowUpdatedEvents` **`RowUpdated`**

  Gets the instance of *RowUpdatedEvents* type that represents the collection of *RowUpdated* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowDeletingEvents` **`RowDeleting`**

  Gets the instance of *RowDeletingEvents* type that represents the collection of *RowDeleting* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowDeletedEvents` **`RowDeleted`**

  Gets the instance of *RowDeletedEvents* type that represents the collection of *RowDeleted* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowPersistingEvents` **`RowPersisting`**

  Gets the instance of *RowPersistingEvents* type that represents the collection of *RowPersisting* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public RowPersistedEvents` **`RowPersisted`**

  Gets the instance of *RowPersistedEvents* type that represents the collection of *RowPersisted* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public CommandPreparingEvents` **`CommandPreparing`**

  Gets the instance of *CommandPreparingEvents* type that represents the collection of *CommandPreparing* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public FieldDefaultingEvents` **`FieldDefaulting`**

  Gets the instance of *FieldDefaultingEvents* type that represents the collection of *FieldDefaulting* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public FieldUpdatingEvents` **`FieldUpdating`**

  Gets the instance of *FieldUpdatingEvents* type that represents the collection of *FieldUpdating* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public FieldVerifyingEvents` **`FieldVerifying`**

Gets the instance of *FieldVerifyingEvents* type that represents the collection of *FieldVerifying* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public FieldUpdatedEvents` **`FieldUpdated`**

  Gets the instance of *FieldUpdatedEvents* type that represents the collection of *FieldUpdated* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public FieldSelectingEvents` **`FieldSelecting`**

  Gets the instance of *FieldSelectingEvents* type that represents the collection of *FieldSelecting* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

- `public ExceptionHandlingEvents` **`ExceptionHandling`**

  Gets the instance of *ExceptionHandlingEvents* type that represents the collection of *ExceptionHandling* event handlers related to the graph. The collection initially contains the event handlers defined in the graph, but it can be modified at run time.

Methods

Method	Description
AllowDelete(string)	Returns the value indicating if the cache related to the data view allows deleting data records through the user interface
AllowInsert(string)	Returns the value indicating if the cache related to the data view allows inserting data records through the user interface
AllowSelect(string)	Returns the value indicating if the cache related to the data view allows selecting data records through the user interface
AllowUpdate(string)	Returns the value indicating if the cache related to the data view allows updating data records through the user interface
Clear()	Clears the graph state stored in the session by clearing the data from each cache
Clear(PXClearOption)	Clears a part of the graph state according to the provided option
CreateInstance(Type)	Initializes a new graph instance of the specified type and extension types if the customization exists
CreateInstance<Graph>()	Initializes a new graph instance of the specified type and extension types if the customization exists
ExecuteDelete(string, IDictionary, IDictionary, params object[])	Deletes the data record from the cache related to the data view by invoking the *Delete(IDictionary)* method on the cache
ExecuteInsert(string, IDictionary, params object[])	Inserts a new data record into the cache related to the data view by invoking the *Insert(IDictionary)* method on the cache

Method	Description
ExecuteSelect(string, object[], object[], string[], bool[], PXFilterRow[], ref int, int, ref int)	Executes the specified data view and returns the data records the data view selects
ExecuteUpdate(string, IDictionary, IDictionary, params object[])	Updates a data record in the cache related to the data view by invoking the *Update(IDictionary)* method on the cache
GetAttributes(string, string)	Gets all instances of attributes placed on the specified field from the cache related to the data view
GetExtension<Extension>()	Returns the instance of the graph extension of the specified type
GetFieldNames(string)	Returns the names of all fields from all DACs referenced by the BQL command of the data view
GetItemType(string)	Returns the type of the first DAC referenced by the data view
GetKeyNames(string)	Returns the names of the keys fields of the cache related to the data view
GetParameterNames(string)	Returns the names of parameters of the data view by invoking the *GetParameterNames(string)* method on the data view
GetSortColumns(string)	Returns pairs of the names of the fields by which the data view result will be sorted and values indicating if the sort by the field is descending
GetStateExt(string, object, string)	Gets the value as the `PXFieldState` object of the specified field in the data record
GetStatus(string)	Returns the status of the `Current` data record of the cache related to the data view
GetUpdatable(string)	Returns the value indicating if the data view is *read-only*
GetValue(string, object, string)	Gets the value of the specified field in the data record without raising any events
GetValueExt(string, object, string)	Gets the value or the `PXFieldState` object of the specified field in the data record
GetViewNames()	Retrieves the names of all data views defined in the graph
HasException()	Returns the value indicating if any updatable cache has an exception
Load()	Loads the state of the graph and caches from the session
Persist()	Saves the modified data records kept in the caches to the database
Persist(Type, PXDBOperation)	Saves the modifications of a particular type from the specified cache to the database

Method	Description
ProviderDelete(Type, params PXDataFieldRestrict[])	Performs a database delete operation
ProviderDelete<Table>(params PXDataFieldRestrict[])	Performs a database delete operation
ProviderEnsure(Type, PXDataFieldAssign[], PXDataField[])	
ProviderExecute(string, params PXSPParameter[])	Executes a database stored procedure
ProviderInsert(Type, params PXDataFieldAssign[])	Performs a database insert operation
ProviderInsert<Table>(params PXDataFieldAssign[])	Performs a database delete operation
ProviderSelect(BqlCommand, int, params PXDataValue[])	Selects the specified amount of top records from the database table
ProviderSelectMulti(Type, params PXDataField[])	Selects multiple records from the database table
ProviderSelectMulti<Table>(params PXDataField[])	Selects multiple records from the database table
ProviderSelectSingle(Type, params PXDataField[])	Selects a single record from the database table
ProviderSelectSingle<Table>(params PXDataField[])	Selects a single record from the database table
ProviderUpdate(Type, params PXDataFieldParam[])	Performs a database update operation
ProviderUpdate<Table>(params PXDataFieldParam[])	Performs a database update operation
SelectTimeStamp()	Retrieves the timestamp value from the database and stores this value in the `TimeStamp` property of the graph
SetValue(string, object, string, object)	Sets the value of the field by field name in the data record without raising any events
SetValueExt(string, object, string, object)	Sets the value of the specified field in the data record
Unload()	Stores the graph state and the modified data records from all caches to the user session
UpdateRights(string)	Returns a value that indicates if updating of the cache related to the data view is allowed

Fields

- `public PXCacheCollection` **`Caches`**

  The dictionary that maps DACs to the related cache objects. An access to the indexer `[]` of this collection implicitly adds an element to the dictionary if the appropriate element does not exist.

- `public readonly PXActionCollection` **`Actions`**

The collection of actions defined in the graph.

- `public PXViewCollection` **`Views`**

  The collection of data views defined in the graph.

- `public readonly Dictionary<PXView, string>` **`ViewNames`**

  The dictionary that allows getting the name of the data view by the corresponding `PXView` object.

- `public PXTypedViewCollection` **`TypedViews`**

  The collection of `PXView` objects indexed by the first DACs referenced by the corresponding BQL commands.

- `public static InstanceCreatedEvents` **`InstanceCreated`**

  The instance of *InstanceCreatedEvents* type representing the collection of `InstanceCreated` event handlers.

Nested Classes

The `PXGraph` type includes definitions of a number of *nested classes*, which all represent collections of graph event handlers of specific types. The methods of these classes can be used to modify the collections at run time, adding and removing event handlers. Note that, depending on the type of event, new event handlers are added to either the start or the end of the collection. Also, the collections do not include event handlers that are defined in attributes, because attribute event handlers are maintained by caches.

PXGraph Methods

The *PXGraph* type exposes the following methods.

AllowDelete(string)

Returns the value indicating if the cache related to the data view allows deleting data records through the user interface. This flag does not affect the ability to delete a data record through code.

Syntax:

```
public virtual bool AllowDelete(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

AllowInsert(string)

Returns the value indicating if the cache related to the data view allows inserting data records through the user interface. This flag does not affect the ability to insert a data record through code.

Syntax:

```
public virtual bool AllowInsert(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

AllowSelect(string)

Returns the value indicating if the cache related to the data view allows selecting data records through the user interface. This flag does not affect the ability to select data records through code.

Syntax:

```
public virtual bool AllowSelect(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

AllowUpdate(string)

Returns the value indicating if the cache related to the data view allows updating data records through the user interface. This flag does not affect the ability to update a data record through code.

Syntax:

```
public virtual bool AllowUpdate(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

Clear()

Clears the graph state stored in the session by clearing the data from each cache.

Syntax:

```
public virtual void Clear()
```

Clear(PXClearOption)

Clears a part of the graph state according to the provided option.

Syntax:

```
public virtual void Clear(PXClearOption option)
```

Parameters:

- `option`

  The value of *PXClearOption* type that specifies which data to clear.

CreateInstance(Type)

Initializes a new graph instance of the specified type and extension types if the customization exists. This method provides a preferred way of initializing a graph.

Syntax:

```
public static PXGraph CreateInstance(Type graphType)
```

Parameters:

- `graphType`

A type derived from `PXGraph`.

CreateInstance<Graph>()

Initializes a new graph instance of the specified type and extension types if the customization exists. This method provides a preferred way of initializing a graph. The graph type is specified in the type parameter.

Syntax:

```
public static Graph CreateInstance<Graph>()
    where Graph : PXGraph, new()
```

Examples:

The code below initializes an instance of the `JournalEntry` graph.

```
JournalEntry graph = PXGraph.CreateInstance<JournalEntry>();
```

ExecuteDelete(string, IDictionary, IDictionary, params object[])

Deletes the data record from the cache related to the data view by invoking the *Delete(IDictionary)* method on the cache. Returns 1 in case of successful deletion and 0 otherwise.

The method is used by the user interface.

Syntax:

```
public virtual int ExecuteDelete(string viewName, IDictionary keys, IDictionary
 values, params object[] parameters)
```

Parameters:

- `viewName`

  The name of the data view.
- `keys`

  The keys that identify the data record.
- `values`

  The values of the data record fields.

ExecuteInsert(string, IDictionary, params object[])

Inserts a new data record into the cache related to the data view by invoking the *Insert(IDictionary)* method on the cache. Returns 1 in case of successful insertion and 0 otherwise.

The method is used by the user interface.

Syntax:

```
public virtual int ExecuteInsert(string viewName, IDictionary values, params
 object[] parameters)
```

Parameters:

- `viewName`

  The name of the data view.
- `values`

  The values to populates the data record fields .

ExecuteSelect(string, object[], object[], string[], bool[], PXFilterRow[], ref int, int, ref int)

Executes the specified data view and returns the data records the data view selects.

The method raises the `RowSelected` event for each retrieved data record and sets the `Current` property of the cache to the last data record retrieved.

The method is used by the user interface. The application code does not typically need to use this method and selects the data directly through the data views.

Syntax:

```
public virtual IEnumerable ExecuteSelect(
    string viewName, object[] parameters,
    object[] searches, string[] sortcolumns,
    bool[] descendings, PXFilterRow[] filters,
    ref int startRow, int maximumRows, ref int totalRows)
```

Parameters:

- `viewName`

  The name of the data view.

- `parameters`

  Parameters for the BQL command.

- `searches`

  The values by which the data is filtered.

- `sortcolumns`

  The fields by which the if sorted and filtered (the filtering values are provided in the `searches` parameter)

- `(ref) startRow`

  The index of the data record to start retreiving with (after filtering by the `searches` parameter).

- `maximumRows`

  The maximum number of data records to retrieve.

- `(ref) totalRows`

  The total amount of data records in the resultset.

ExecuteUpdate(string, IDictionary, IDictionary, params object[])

Updates a data record in the cache related to the data view by invoking the *Update(IDictionary)* method on the cache. Returns 1 in case of successful update and 0 otherwise.

The method is used by the user interface.

Syntax:

```
public virtual int ExecuteUpdate(string viewName, IDictionary keys, IDictionary
 values, params object[] parameters)
```

Parameters:

- `viewName`

  The name of the data view.

- `keys`

  The keys that identify the data record.

- `values`

  The new values of the data record fields.

GetAttributes(string, string)

Gets all instances of attributes placed on the specified field from the cache related to the data view. The method relies on the *GetAttributes(string)* method of the cache.

Syntax:

```
public PXEventSubscriberAttribute[] GetAttributes(string viewName, string name)
```

Parameters:

- `viewName`

  The name of the data view.

- `name`

  The name of the field whose attributes are returned. If `null`, the attributes from all fields are returned.

GetExtension<Extension>()

Returns the instance of the graph extension of the specified type. The type of the extension is specified in the type parameter.

Syntax:

```
public virtual Extension GetExtension<Extension>()
    where Extension : PXGraphExtension
```

Examples:

An extension of a graph is a class that derives from the `PXGraphExtension<>` type. The example below shows the definition of an extension on the `InventoryItemMaint` graph.

```
public class InventoryItemMaintExtension :
    PXGraphExtension<InventoryItemMaint>
{
    public void SomeMethod()
    {
        // The Base variable references the instance of InventoryItemMaint
        InventoryItemMaintExtension ext =
            Base.GetExtension<InventoryItemMaintExtension>();
        ...
    }
}
```

GetFieldNames(string)

Returns the names of all fields from all DACs referenced by the BQL command of the data view.

Syntax:

```
public string[] GetFieldNames(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

GetItemType(string)

Returns the type of the first DAC referenced by the data view.

Syntax:

```
public Type GetItemType(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

GetKeyNames(string)

Returns the names of the keys fields of the cache related to the data view.

Syntax:

```
public string[] GetKeyNames(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

GetParameterNames(string)

Returns the names of parameters of the data view by invoking the *GetParameterNames(string)* method on the data view.

Syntax:

```
public string[] GetParameterNames(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

GetSortColumns(string)

Returns pairs of the names of the fields by which the data view result will be sorted and values indicating if the sort by the field is descending.

Syntax:

```
public virtual KeyValuePair<string, bool>[] GetSortColumns(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

GetStateExt(string, object, string)

Gets the value as the `PXFieldState` object of the specified field in the data record. The method relies on the *GetStateExt(object, string)* method of the cache.

Syntax:

```
public virtual object GetStateExt(string viewName, object data, string fieldName)
```

Parameters:

- `viewName`

  The name of the data view.
- `data`

  The data record from the cache related to the data view.
- `fieldName`

  The name of the field whose state is returned.

GetStatus(string)

Returns the status of the `Current` data record of the cache related to the data view. If the `Current` property of the cache is `null`, the method returns the `Notchanged` status.

Syntax:

```
public PXEntryStatus GetStatus(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

GetUpdatable(string)

Returns the value indicating if the data view is *read-only*.

Syntax:

```
public virtual bool GetUpdatable(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

GetValue(string, object, string)

Gets the value of the specified field in the data record without raising any events. The method relies on the *GetValue(object, string)* method of the cache related to the data view.

Syntax:

```
public virtual object GetValue(string viewName, object data, string fieldName)
```

Parameters:

- `viewName`

  The name of the data view.
- `data`

  The data record from the cache related to the data view.
- `fieldName`

The name of the field whose value is returned.

GetValueExt(string, object, string)

Gets the value or the `PXFieldState` object of the specified field in the data record. The method relies on the *GetValueExt(object, string)* method of the cache related to the data view.

Syntax:

```
public virtual object GetValueExt(string viewName, object data, string fieldName)
```

Parameters:

- `viewName`

  The name of the data view.
- `data`

  The data record from the cache related to the data view.
- `fieldName`

  The name of the field whose value or state is returned.

GetViewNames()

Retrieves the names of all data views defined in the graph.

Syntax:

```
public virtual IEnumerable<string> GetViewNames()
```

HasException()

Returns the value indicating if any updatable cache has an exception.

Syntax:

```
public bool HasException()
```

Load()

Loads the state of the graph and caches from the session.

The state is stored in the session through the *Unload()* method.

Syntax:

```
public virtual void Load()
```

Persist()

Saves the modified data records kept in the caches to the database.

All data records are saved within a single transaction context. The method takes into account only the caches from `Views.Caches` collection.

The method saves the data records in the following order:

1. Data records with the `Inserted` status from all caches.
2. Data records with the `Updated` status from all caches.
3. Data records with the `Deleted` status from all caches.

Syntax:

```
public virtual void Persist()
```

Remarks:

The application does not typically saves the changes through this method directly. The preferred way of saving the changes to the database is to executed `Actions.PressSave()` on the graph. The `PressSave()` method of the `Actions` collection is invokes the `Persist()` method on the graph and performs additional procedures.

Persist(Type, PXDBOperation)

Saves the modifications of a particular type from the specified cache to the database. The method relise on the *Persist(PXDBOperation)* method of the cache.

Syntax:

```
public virtual int Persist(Type cacheType, PXDBOperation operation)
```

Parameters:

- `cacheType`

  The DAC type of the cache whose changes are saved.

ProviderDelete(Type, params PXDataFieldRestrict[])

Performs a database delete operation.

Syntax:

```
public virtual bool ProviderDelete(Type table, params PXDataFieldRestrict[] pars)
```

Parameters:

- `table`

  The DAC representing the table whose records are deleted.

- `pars`

  The parameters.

ProviderDelete<Table>(params PXDataFieldRestrict[])

Performs a database delete operation. The table is specified as the DAC through the type parameter.

Syntax:

```
public virtual bool ProviderDelete<Table>(params PXDataFieldRestrict[] pars)
    where Table : IBqlTable
```

Parameters:

- `pars`

  The parameters.

ProviderEnsure(Type, PXDataFieldAssign[], PXDataField[])

Syntax:

```
public virtual bool ProviderEnsure(Type table, PXDataFieldAssign[] values,
```

```
                                    PXDataField[] pars)
```

Parameters:

- `table`

  The DAC representing the table.
- `values`

  The values.
- `pars`

  The parameters.

ProviderExecute(string, params PXSPParameter[])

Executes a database stored procedure.

Syntax:

```
public virtual object[] ProviderExecute(string procedureName,
                                        params PXSPParameter[] pars)
```

Parameters:

- `procedureName`

  The name of the stored procedure to execute.
- `pars`

  The parameters.

ProviderInsert(Type, params PXDataFieldAssign[])

Performs a database insert operation.

Syntax:

```
public virtual bool ProviderInsert(Type table, params PXDataFieldAssign[] pars)
```

Parameters:

- `table`

  The DAC representing the table to which the data records are inserted.
- `pars`

  The parameters.

ProviderInsert<Table>(params PXDataFieldAssign[])

Performs a database delete operation. The table is specified as the DAC through the type parameter.

Syntax:

```
public virtual bool ProviderInsert<Table>(params PXDataFieldAssign[] pars)
    where Table : IBqlTable
```

Parameters:

- `pars`

  The parameters.

ProviderSelect(BqlCommand, int, params PXDataValue[])

Selects the specified amount of top records from the database table.

Syntax:

```
public virtual IEnumerable<PXDataRecord> ProviderSelect(
    BqlCommand command, int topCount, params PXDataValue[] pars)
```

Parameters:

- `command`

  The BQL command defining the select query to execute.

- `topCount`

  The number of the data record to retreive from the top of the data set.

- `pars`

  The parameters.

ProviderSelectMulti(Type, params PXDataField[])

Selects multiple records from the database table.

Syntax:

```
public virtual IEnumerable<PXDataRecord> ProviderSelectMulti(
    Type table, params PXDataField[] pars)
```

Parameters:

- `table`

  The DAC representing the table from which the data records are selected.

- `pars`

  The parameters.

ProviderSelectMulti<Table>(params PXDataField[])

Selects multiple records from the database table. The table is specified as the DAC through the type parameter.

Syntax:

```
public virtual IEnumerable<PXDataRecord> ProviderSelectMulti<Table>(
    params PXDataField[] pars)
    where Table : IBqlTable
```

Parameters:

- `pars`

  The parameters.

ProviderSelectSingle(Type, params PXDataField[])

Selects a single record from the database table.

Syntax:

```
public virtual PXDataRecord ProviderSelectSingle(Type table,
                                                 params PXDataField[] pars)
```

Parameters:

- `table`

  The DAC representing the table from which the data record is selected.

- `pars`

  The parameters.

ProviderSelectSingle<Table>(params PXDataField[])

Selects a single record from the database table. The table is specified as the DAC through the type parameter.

Syntax:

```
public virtual PXDataRecord ProviderSelectSingle<Table>(params PXDataField[] pars)
    where Table : IBqlTable
```

Parameters:

- `pars`

  The parameters.

ProviderUpdate(Type, params PXDataFieldParam[])

Performs a database update operation.

Syntax:

```
public virtual bool ProviderUpdate(Type table, params PXDataFieldParam[] pars)
```

Parameters:

- `table`

  The DAC representing the table from where the data records are updated.

- `pars`

  The parameters.

ProviderUpdate<Table>(params PXDataFieldParam[])

Performs a database update operation. The table is specified as the DAC through the type parameter.

Syntax:

```
public virtual bool ProviderUpdate<Table>(params PXDataFieldParam[] pars)
    where Table : IBqlTable
```

Parameters:

- `pars`

  The parameters.

SelectTimeStamp()

Retrieves the timestamp value from the database and stores this value in the `TimeStamp` property of the graph.

Syntax:

```
public virtual void SelectTimeStamp()
```

SetValue(string, object, string, object)

Sets the value of the field by field name in the data record without raising any events. The method relies on the *SetValue(object, string, object)* method of the cache related to the data view.

Syntax:

```
public virtual void SetValue(string viewName, object data,
                             string fieldName, object value)
```

Parameters:

- `viewName`

  The name of the data view.
- `data`

  The data record to update.
- `fieldName`

  The name of the field to update.
- `value`

  The new value for the field.

SetValueExt(string, object, string, object)

Sets the value of the specified field in the data record. The method relies on the *SetValueExt(object, string, object)* method of the cache related to the data view.

Syntax:

```
public virtual void SetValueExt(string viewName, object data,
                                string fieldName, object value)
```

Parameters:

- `viewName`

  The name of the data view.
- `data`

  The data record to update as an instance of the DAC or `IDictionary` of field names and field values.
- `fieldName`

  The name of the field to update.
- `value`

  The new value for the field.

Unload()

Stores the graph state and the modified data records from all caches to the user session.

Syntax:

```
public virtual void Unload()
```

Remarks:

The instance of the graph is destroyed at the end of the each callback. To preserve user data not saved in the database between callbacks, the caches of modified data record are serialized to the session using this method.

UpdateRights(string)

Returns a value that indicates if updating of the cache related to the data view is allowed.

Syntax:

```
public virtual bool UpdateRights(string viewName)
```

Parameters:

- `viewName`

  The name of the data view.

PXClearOption Enumeration

Defines possible options of clearing the graph data through the *Clear(PXClearOption)* method.

Members

- `PreserveData`

  Data records are preserved.

- `PreserveTimeStamp`

  The timestamp is preserved.

- `PreserveQueries`

  The query cache is preserved.

- `ClearAll`

  Everything is removed.

- `ClearQueriesOnly`

  Only the query cache is cleared.

PXGraph Nested Classes

The *PXGraph* type exposes the following nested classes.

InstanceCreatedEvents Class

Represents the colection of `InstanceCreated` event handlers, which are invoked when a new instance of the graph is initialized.

Syntax:

```
public sealed class InstanceCreatedEvents
    where TGraph : PXGraph
```

Methods:

- `public void AddHandler<TGraph>(InstanceCreatedDelegate<TGraph> del)`

Adds the provided handler to the collection for the specified graph type.

- public void RemoveHandler<TGraph>(InstanceCreatedDelegate<TGraph> del)

  Removes the provided handler from the collection for the specified graph type.

RowSelectingEvents Class

Represents the collections of RowSelecting event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowSelectingEvents
```

Constructors:

- public RowSelectingEvents(PXGraph graph)

  Initializes an instance and binds it to the provided graph.

Methods:

- public void AddHandler(string view, PXRowSelecting handler)

  Adds the event handler to the end of the collection for the primary DAC of the specified data view.
- public void RemoveHandler(string view, PXRowSelecting handler)

  Removes the event handler from the collection related to the primary DAC of the data view.
- public void AddHandler<Type>(PXRowSelecting handler)

  Adds the event handler to the end of the collection for the specified DAC.
- public void RemoveHandler<Type>(PXRowSelecting handler)

  Removes the event handler from the collection related to the specified DAC.
- public void AddHandler(Type type, PXRowSelecting handler)

  Adds the event handler to the end of the collection for the specified DAC.
- public void RemoveHandler(Type type, PXRowSelecting handler)

  Removes the event handler from the collection related to the specified DAC.

RowSelectedEvents Class

Represents the collection of RowSelected event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowSelectedEvents
```

Constructors:

- public RowSelectedEvents(PXGraph graph)

  Initializes an instance and binds it to the provided graph.

Methods:

- public void AddHandler(string view, PXRowSelected handler)

  Adds the event handler to the end of the collection for the primary DAC of the specified data view.
- public void RemoveHandler(string view, PXRowSelected handler)

Removes the event handler from the collection related to the primary DAC of the data view.

- `public void AddHandler<Type>(PXRowSelected handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowSelected handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowSelected handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowSelected handler)`

  Removes the event handler from the collection related to the specified DAC.

RowInsertingEvents Class

Represents the collection of `RowInserting` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowInsertingEvents
```

Constructors:

- `public RowInsertingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowInserting handler)`

  Adds the event handler to the beginning of the collection for the primary DAC of the data view.
- `public void RemoveHandler(string view, PXRowInserting handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowInserting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowInserting handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowInserting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowInserting handler)`

  Removes the event handler from the collection related to the specified DAC.

RowInsertedEvents Class

Represents the collection of `RowInserted` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowInsertedEvents
```

Constructors:

- `public RowInsertedEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowInserted handler)`

  Adds the event handler to the end of the collection for the primary DAC of the specified data view.
- `public void RemoveHandler(string view, PXRowInserted handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowInserted handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowInserted handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowInserted handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowInserted handler)`

  Removes the event handler from the collection related to the specified DAC.

RowUpdatingEvents Class

Represents the collection of `RowUpdating` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowUpdatingEvents
```

Constructors:

- `public RowUpdatingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowUpdating handler)`

  Adds the event handler to the beginning of the collection for the primary DAC of the data view.
- `public void RemoveHandler(string view, PXRowUpdating handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowUpdating handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowUpdating handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowUpdating handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowUpdating handler)`

  Removes the event handler from the collection related to the specified DAC.

RowUpdatedEvents Class

Represents the collection of `RowUpdated` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowUpdatedEvents
```

Constructors:

- `public RowUpdatedEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowUpdated handler)`

  Adds the event handler to the end of the collection for the primary DAC of the specified data view.
- `public void RemoveHandler(string view, PXRowUpdated handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowUpdated handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowUpdated handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowUpdated handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowUpdated handler)`

  Removes the event handler from the collection related to the specified DAC.

RowDeletingEvents Class

Represents the collection of `RowDeleting` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowDeletingEvents
```

Constructors:

- `public RowDeletingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowDeleting handler)`

  Adds the event handler to the beginning of the collection for the primary DAC of the data view.
- `public void RemoveHandler(string view, PXRowDeleting handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowDeleting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowDeleting handler)`

Removes the event handler from the collection related to the specified DAC.

- `public void AddHandler(Type type, PXRowDeleting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowDeleting handler)`

  Removes the event handler from the collection related to the specified DAC.

RowDeletedEvents Class

Represents the collection of `RowDeleted` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowDeletedEvents
```

Constructors:

- `public RowDeletedEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowDeleted handler)`

  Adds the event handler to the end of the collection for the primary DAC of the specified data view.
- `public void RemoveHandler(string view, PXRowDeleted handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowDeleted handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowDeleted handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowDeleted handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowDeleted handler)`

  Removes the event handler from the collection related to the specified DAC.

RowPersistingEvents Class

Represents the collection of `RowPersisting` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowPersistingEvents
```

Constructors:

- `public RowPersistingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowPersisting handler)`

Adds the event handler to the beginning of the collection for the primary DAC of the data view.

- `public void RemoveHandler(string view, PXRowPersisting handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowPersisting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowPersisting handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowPersisting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowPersisting handler)`

  Removes the event handler from the collection related to the specified DAC.

RowPersistedEvents Class

Represents the collection of `RowPersisted` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class RowPersistedEvents
```

Constructors:

- `public RowPersistedEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, PXRowPersisted handler)`

  Adds the event handler to the end of the collection for the primary DAC of the specified data view.
- `public void RemoveHandler(string view, PXRowPersisted handler)`

  Removes the event handler from the collection related to the primary DAC of the data view.
- `public void AddHandler<Type>(PXRowPersisted handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler<Type>(PXRowPersisted handler)`

  Removes the event handler from the collection related to the specified DAC.
- `public void AddHandler(Type type, PXRowPersisted handler)`

  Adds the event handler to the end of the collection for the specified DAC.
- `public void RemoveHandler(Type type, PXRowPersisted handler)`

  Removes the event handler from the collection related to the specified DAC.

CommandPreparingEvents Class

Represents the collection of `CommandPreparing` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class CommandPreparingEvents
```

Constructors:

- `public CommandPreparingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, string field, PXCommandPreparing handler)`

  Adds the event handler to the beginning of the collection for the specified field defined in the primary DAC of the data view.
- `public void RemoveHandler(string view, string field, PXCommandPreparing handler)`

  Removes the event handler from the collection related to the specified field defined in the primary DAC of the data view.
- `public void AddHandler<Field>(PXCommandPreparing handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.
- `public void RemoveHandler<Field>(PXCommandPreparing handler)`

  Removes the event handler from the collection related to the specified DAC field.
- `public void AddHandler(Type type, string field, PXCommandPreparing handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.
- `public void RemoveHandler(Type type, string field, PXCommandPreparing handler)`

  Removes the event handler from the collection related to the specified DAC field.

FieldDefaultingEvents Class

Represents the collection of `FieldDefaulting` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class FieldDefaultingEvents
```

Constructors:

- `public FieldDefaultingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, string field, PXFieldDefaulting handler)`

  Adds the event handler to the beginning of the collection for the specified field defined in the primary DAC of the data view.
- `public void RemoveHandler(string view, string field, PXFieldDefaulting handler)`

Removes the event handler from the collection related to the specified field defined in the primary DAC of the data view.

- public void AddHandler<Field>(PXFieldDefaulting handler)

  Adds the event handler to the beginning of the collection for the specified DAC field.

- public void RemoveHandler<Field>(PXFieldDefaulting handler)

  Removes the event handler from the collection related to the specified DAC field.

- public void AddHandler(Type type, string field, PXFieldDefaulting handler)

  Adds the event handler to the beginning of the collection for the specified DAC field.

- public void RemoveHandler(Type type, string field, PXFieldDefaulting handler)

  Removes the event handler from the collection related to the specified DAC field.

FieldUpdatingEvents Class

Represents the collection of FieldUpdating event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class FieldUpdatingEvents
```

Constructors:

- public FieldUpdatingEvents(PXGraph graph)

  Initializes an instance and binds it to the provided graph.

Methods:

- public void AddHandler(string view, string field, PXFieldUpdating handler)

  Adds the event handler to the beginning of the collection for the specified field defined in the primary DAC of the data view.

- public void RemoveHandler(string view, string field, PXFieldUpdating handler)

  Removes the event handler from the collection related to the specified field defined in the primary DAC of the data view.

- public void AddHandler<Field>(PXFieldUpdating handler)

  Adds the event handler to the beginning of the collection for the specified DAC field.

- public void RemoveHandler<Field>(PXFieldUpdating handler)

  Removes the event handler from the collection related to the specified DAC field.

- public void AddHandler(Type type, string field, PXFieldUpdating handler)

  Adds the event handler to the beginning of the collection for the specified DAC field.

- public void RemoveHandler(Type type, string field, PXFieldUpdating handler)

  Removes the event handler from the collection related to the specified DAC field.

FieldUpdatedEvents Class

Represents the collection of `FieldUpdated` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class FieldUpdatedEvents
```

Constructors:

- `public FieldUpdatedEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, string field, PXFieldUpdated handler)`

  Adds the event handler to the end of the collection for the specified field defined in the primary DAC of the data view.

- `public void RemoveHandler(string view, string field, PXFieldUpdated handler)`

  Removes the event handler from the collection related to the specified field defined in the primary DAC of the data view.

- `public void AddHandler<Field>(PXFieldUpdated handler)`

  Adds the event handler to the end of the collection for the specified DAC field.

- `public void RemoveHandler<Field>(PXFieldUpdated handler)`

  Removes the event handler from the collection related to the specified DAC field.

- `public void AddHandler(Type type, string field, PXFieldUpdated handler)`

  Adds the event handler to the end of the collection for the specified DAC field.

- `public void RemoveHandler(Type type, string field, PXFieldUpdated handler)`

  Removes the event handler from the collection related to the specified DAC field.

FieldSelectingEvents Class

Represents the collection of `FieldSelecting` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class FieldSelectingEvents
```

Constructors:

- `public FieldSelectingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, string field, PXFieldSelecting handler)`

  Adds the event handler to the beginning of the collection for the specified field defined in the primary DAC of the data view.

- `public void RemoveHandler(string view, string field, PXFieldSelecting handler)`

  Removes the event handler from the collection related to the specified field defined in the primary DAC of the data view.

- `public void AddHandler<Field>(PXFieldSelecting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.

- `public void RemoveHandler<Field>(PXFieldSelecting handler)`

  Removes the event handler from the collection related to the specified DAC field.

- `public void AddHandler(Type type, string field, PXFieldSelecting handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.

- `public void RemoveHandler(Type type, string field, PXFieldSelecting handler)`

  Removes the event handler from the collection related to the specified DAC field.

ExceptionHandlingEvents Class

Represents the collection of `ExceptionHandling` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class ExceptionHandlingEvents
```

Constructors:

- `public ExceptionHandlingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, string field, PXExceptionHandling handler)`

  Adds the event handler to the beginning of the collection for the specified field defined in the primary DAC of the data view.

- `public void RemoveHandler(string view, string field, PXExceptionHandling handler)`

  Removes the event handler from the collection related to the specified field defined in the primary DAC of the data view.

- `public void AddHandler<Field>(PXExceptionHandling handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.

- `public void RemoveHandler<Field>(PXExceptionHandling handler)`

  Removes the event handler from the collection related to the specified DAC field.

- `public void AddHandler(Type type, string field, PXExceptionHandling handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.

- `public void RemoveHandler(Type type, string field, PXExceptionHandling handler)`

Removes the event handler from the collection related to the specified DAC field.

FieldVerifyingEvents Class

Represents the collection of `FieldVerifying` event handlers declared as methods in the graph or added at run time.

Syntax:

```
public sealed class FieldVerifyingEvents
```

Constructors:

- `public FieldVerifyingEvents(PXGraph graph)`

  Initializes an instance and binds it to the provided graph.

Methods:

- `public void AddHandler(string view, string field, PXFieldVerifying handler)`

  Adds the event handler to the beginning of the collection for the specified field defined in the primary DAC of the data view.

- `public void RemoveHandler(string view, string field, PXFieldVerifying handler)`

  Removes the event handler from the collection related to the specified field defined in the primary DAC of the data view.

- `public void AddHandler<Field>(PXFieldVerifying handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.

- `public void RemoveHandler<Field>(PXFieldVerifying handler)`

  Removes the event handler from the collection related to the specified DAC field.

- `public void AddHandler(Type type, string field, PXFieldVerifying handler)`

  Adds the event handler to the beginning of the collection for the specified DAC field.

- `public void RemoveHandler(Type type, string field, PXFieldVerifying handler)`

  Removes the event handler from the collection related to the specified DAC field.

PXGraph<TGraph> Class

The type that is used to derive business logic controllers (graphs) in the application.

This type extends the *PXGraph* type with the ability to automatically initialize data views, actions, and event handlers that are defined as members in the current graph or in its base graphs.

Inheritance Hierarchy

```
PXGraph
```

Syntax

```
[System.Security.Permissions.ReflectionPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
[System.Security.Permissions.SecurityPermission(
```

```
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
public class PXGraph<TGraph> : PXGraph
    where TGraph : PXGraph
```

Remarks

In a graph, you can define the following members:

- Data views as objects of the *PXSelect<>* type or its variant. The type of a data view is the BQL expression which can be executed by invoking the `Select()` or `Search()` methods.
- Actions as objects of the `PXAction` type and paired by the implementation method.
- Event handlers.

For a data view you can also define the optional method that will be executed by the Select() method to retrieve the data instead of the standard logic of retreiving the data.

Data views and actions must be declared as `public`. When you declare data views and actions, you do not initialize them. The graph initializes them automatically. The `PXView` objects initialized by the data views are available through the *Views* collection of the graph. The actions are available through the *Actions* collection of the graph.

Event handlers and methods can be declared as `public`, `protected`, or `internal`. The `protected virtual` is the recommended modifier. Event handlers of particular type are available through the corresponding *collections*.

You can derive a graph from the *PXGraph<TGraph, TPrimary>* type to add pre-defined actions to the graph.

Examples

The code below declares a graph.

```
public class ARDocumentEnq : PXGraph<ARDocumentEnq>
{
}
```

The type parameter is set to the graph itself.

The code below declares a graph with a data view, an action, and an event handler.

```
public class ARDocumentEnq : PXGraph<ARDocumentEnq>
{
    // The data view declaration
    public PXSelectOrderBy<ARDocumentResult,
                OrderBy<Desc<ARDocumentResult.docDate>>> Documents;

    // The action declaration
    public PXAction<ARDocumentFilter> previousPeriod;
    [PXUIField(DisplayName = "Prev")]
    [PXPreviousButton]
    public virtual IEnumerable PreviousPeriod(PXAdapter adapter)
    {
        ...
    }

    // The event handler declaration
    public virtual void ARDocumentFilter_RowSelected(
        PXCache cache, PXRowSelectedEventArgs e)
    {
        ...
    }
}
```

PXGraph<TGraph, TPrimary> Class

The same as *PXGraph<TGraph>* but appends the following standard actions for the provided DAC: `Save`, `Insert`, `Edit`, `Delete`, `Cancel`, `Prev`, `Next`, `First`, `Last`. The DAC is specified in the second type parameter.

See *Remarks* for more details.

Inheritance Hierarchy

```
PXGraph
```

Syntax

```
[System.Security.Permissions.ReflectionPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
[System.Security.Permissions.SecurityPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
public class PXGraph<TGraph, TPrimary> : PXGraph
    where TGraph : PXGraph
    where TPrimary : class, IBqlTable, new()
```

The `PXGraph<TGraph, TPrimary>` type exposes the following members.

Fields

- `public PXSave<TPrimary>` **Save**

  The action that saves changes stored in the caches to the database. The code of an application graph typically saves changes through this action as well. To invoke it from code, use the `PressSave()` method of the `Actions` property.

- `public PXCancel<TPrimary>` **Cancel**

  The action that discard changes to the data from the caches.

- `public PXInsert<TPrimary>` **Insert**

  The action that inserts a new data record into the primary cache.

- `public PXCopyPasteAction<TPrimary>` **CopyPaste**

  The action that is represented on the user interface by an expandable menu that includes **Copy** and **Paste** items.

- `public PXDelete<TPrimary>` **Delete**

  The action that deletes the `Current` data record of the primary cache.

- `public PXFirst<TPrimary>` **First**

  The action that navigates to the first data record in the primary data view. The data record is set to the `Current` property of the primary cache.

- `public PXPrevious<TPrimary>` **Previous**

  The action that navigates to the previous data record in the primary data view. The data record is set to the `Current` property of the primary cache.

- `public PXNext<TPrimary>` **Next**

  The action that navigates to the next data record in the primary data view. The data record is set to the `Current` property of the primary cache.

- `public PXLast<TPrimary>` **`Last`**

  The action that navigates to the last data record in the primary data view. The data record is set to the `Current` property of the primary cache.

Examples

The code below declares a graph that includes a pre-defined set of actions for the `Contact` DAC.

```
public class ContactMaint : PXGraph<ContactMaint, Contact>
{
    ...
}
```

If a webpage is bound to this graph, the webpage toolbar will include the action buttons, which may be used to save, insert, delete, and navigate to `Contact` data records selected by the primary data view (the data view defined first).

PXView Class

A controller that executes the BQL command and implements interfaces for sorting, searching, merging data with the cached changes, and caching the result set.

Syntax

```
[System.Security.Permissions.ReflectionPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
[System.Security.Permissions.SecurityPermission(
    System.Security.Permissions.SecurityAction.Assert,
    Unrestricted = true)]
public class PXView
```

The `PXView` type exposes the following members.

Constructors

Constructor	Description
PXView(PXGraph, bool, BqlCommand)	Initializes an instance for executing the BQL command.
PXView(PXGraph, bool, BqlCommand, Delegate)	Initializes an instance for executing the BQL command using the provided method to retrieve data.

Properties

- `public virtual PXGraph` **`Graph`**

  Gets or sets the parent business object.

- `public virtual bool` **`IsReadOnly`**

  Gets or sets the value that indicates whether placing retrieved data records into the cache and merging them with the cache are allowed.

- `public Delegate` **`BqlDelegate`**

  Gets the delegate representing the method (called *optional method* in this reference) which is invoked by the `Select(...)` method to retrieve the data. If this method is provided to the `PXView` object, the Select(...) method doesn't retrieve data from the database and returns the result returned by the optional method.

- `public virtual PXCache` **`Cache`**

Gets the cache corresponding to the first DAC mentioned in the BQL command.

- `public virtual BqlCommand` **`BqlSelect`**

  Gets the underlying BQL command. If the current `PXView` object is associated with a variant of `PXSelect<>` object, the BQL command type has the the same type parameters as the type of this object, so it represents the same SQL query.

- `public virtual Type` **`BqlTarget`**

  Gets the class that defines the optional method of a data view. Typically, this class is the graph that defines both the data view and its optional method. The optional method is the method represented by `BqlDelegate`. When a data view is defined as a member of a graph.

- `public WebDialogResult` **`Answer`**

  Gets or sets the value indicating user's choice in the dialog window displayed through one of the `Ask()` methods.

The following static properties can be used in the optional method of the data view. The properties return the parameters passed to the currently executed *Select(...)* method.

- `public static string[]` **`SortColumns`**

  Gets the names of the fields passed to the `Select(...)` method to filter and sort the data set.

- `public static bool[]` **`Descendings`**

  Gets the values passed to the `Select(...)` method to indicate whether ordering by the sort columns should be descending or ascending.

- `public static object[]` **`Searches`**

  Gets the values passed to the `Select(...)` method to filter the data set by them.

- `public static PXGraph` **`CurrentGraph`**

  Gets the graph within which the `Select(...)` method was invoked.

- `public static PXFilterRowCollection` **`Filters`**

  Gets the filtering conditions originated on the user interface and passed to the `Select(...)` method.

- `public static object[]` **`Currents`**

  Gets the current data records passed to the `Select(...)` method to process the `Current` and `Optional` parameters.

- `public static object[]` **`Parameters`**

  Gets the values passed to the `Select(...)` method to process such parameters as `Required`, `Optional`, and `Argument`, and pre-processed by the `Select(...)` method.

- `public static int` **`StartRow`**

  Gets or sets the value passed to the `Select(...)` method as the index of the first data record to retrieve.

- `public static int` **`MaximumRows`**

  Gets the value passed to the `Select(...)` method as the number of data records to retrieve.

- `public static bool` **`ReverseOrder`**

  Gets the value indicating whether a negative value was passed as the index of the first data record to retrieve.

Methods

Method	Description
AppendTail(object, List<object>, params object[])	Selects the data records joined with the provided data record by the underlying BQL command
Ask(string, MessageButtons)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, MessageButtons)	Displays the dialog window with single or multiple choices for the user
Ask(string, MessageButtons, bool)	Displays the dialog window with single or multiple choices for the user
Ask(string, string, MessageButtons, bool)	Displays the dialog window with single or multiple choices for the user
AskExt()	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(bool)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(InitializePanel)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string, bool)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string, InitializePanel)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(InitializePanel, bool)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(string, InitializePanel, bool)	Displays the dialog window configured by the `PXSmartPanel` control
AskExt(PXGraph, string, string, InitializePanel)	Displays the dialog window configured by the `PXSmartPanel` control
Clear()	Clears the results of BQL statement execution
ClearDialog()	Clears the dialog information saved by the graph on last invocation of the `Ask()` method
DetachCache()	Initialize a new cache for storing the results of BQL statement execution
EnumParameters()	Returns the information on the fields referenced by BQL parameters and parameters of the optional method, if it is defined for the data view
FieldGetValue(PXCache, object, Type, string)	Gets the value of the specified field in the data record from the cache
Filter(IEnumerable)	

Method	Description
GetAnswer(string)	Returns the result of the dialog window that was opened through one of the `Ask()` methods and saved in the `PXView` object
GetItemType()	Returns the DAC type of the primary cache; that is, the first DAC referenced in the BQL command
GetItemTypes()	Returns all DAC types referenced in the BQL command
GetParameterNames()	Returns the names of the fields referenced by BQL parameters and the names of parameters of the optional method, if it is defined
GetSortColumns()	Returns pairs of the names of the fields by which the data view result will be sorted and values indicating if the sort by the field is descending
Join(Type)	Appends the provided join clause to the BQL command
Join<join>()	Appends the provided join clause to the BQL command
OrderByNew(Type)	Replaces the sorting expression with the new sorting expression
OrderByNew<newOrderBy>()	Replaces the sorting expression with the new sorting expression
PrepareParameters(object[], object[])	Prepares parameters, formats input values, gets default values for the hidden and not supplied parameters
RequestRefresh()	Raises the `RequestRefresh` event defined within the `PXView` object
Select(object[], object[], object[], string[], bool[], PXFilterRow[], ref int, int, ref int)	Executes the BQL command and returns the result set
SelectMulti(params object[])	Retrieves the whole data set corresponding to the BQL command
SelectMultiBound(object[], params object[])	Retrieves the whole data set corresponding to the BQL command
SelectSingle(params object[])	Retrieves the top data record from the data set corresponding to the BQL command
SelectSingleBound(object[], params object[])	Retrieves the top data record from the data set corresponding to the BQL command
SetAnswer(string, WebDialogResult)	Saves the result of the dialog window
SetAnswer(PXGraph, string, string, WebDialogResult)	Saves the result of the dialog window
Sort(IEnumerable)	Sort the provided collection of `PXResult<>` instances by the conditions currenlty stored in the `PXView` context
ToString()	Returns the string with the SQL query corresponding to the underlying BQL command

Method	Description
WhereAnd(Type)	Appends a filtering expression to the underlying BQL command via the logical "and"
WhereAnd<TWhere>()	Appends a filtering expression to the underlying BQL command via the logical "and"
WhereNew(Type)	Replaces the filtering expression in the BQL statement
WhereNew<newWhere>()	Replaces the filtering expression in the BQL statement
WhereNot()	Adds logical "not" to the whole `Where` clause of the BQL statement, reversing the condition to the opposite
WhereOr(Type)	Appends a filtering expression to the BQL statement via the logical "or"
WhereOr<TWhere>()	Appends a filtering expression to the BQL statement via the logical "or"

PXView Constructors

The *PXView* type exposes the following constructors.

PXView(PXGraph, bool, BqlCommand)

Initializes an instance for executing the BQL command.

Syntax:

```
public PXView(PXGraph graph, bool isReadOnly, BqlCommand select)
```

Parameters:

- `graph`

  The graph with which the instance is associated.
- `isReadOnly`

  The value that indicates if updating the cache and merging data with the cache are allowed.
- `select`

  The BQL command as an instance of the type derived from the `BqlCommand` class.

PXView(PXGraph, bool, BqlCommand, Delegate)

Initializes an instance for executing the BQL command using the provided method to retrieve data.

Syntax:

```
public PXView(PXGraph graph, bool isReadOnly, BqlCommand select, Delegate handler) :
 this(graph, isReadOnly, select)
```

Parameters:

- `graph`

  The graph with which the instance is associated.
- `isReadOnly`

  The value that indicates if updating the cache and merging data with the cache are allowed.
- `select`

The BQL command as an instance of the type derived from the `BqlCommand` class.

- `handler`

  Either PXPrepareDelegate or PXSelectDelegate.

PXView Methods

The *PXView* type exposes the following methods.

AppendTail(object, List<object>, params object[])

Selects the data records joined with the provided data record by the underlying BQL command.

Syntax:

```
public virtual void AppendTail(object item, List<object> list,
                               params object[] parameters)
```

Parameters:

- `item`

  First data item.

- `parameters`

  Parameters.

Returns:

The first item plus joined rows.

Ask(string, MessageButtons)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string message, MessageButtons buttons)
```

Parameters:

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

Ask(string, string, MessageButtons)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string key, string message,
                           MessageButtons buttons)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `message`

The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

Ask(string, MessageButtons, bool)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string message, MessageButtons buttons,
                           bool refreshRequired)
```

Parameters:

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

Ask(string, string, MessageButtons, bool)

Displays the dialog window with single or multiple choices for the user.

Syntax:

```
public WebDialogResult Ask(string key, string message,
                           MessageButtons buttons, bool refreshRequired)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `message`

  The string displayed as the message inside the dialog window.

- `buttons`

  The value from the *MessageButtons* enumeration that indicates which set of buttons to display in the dialog window.

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt()

Displays the dialog window configured by the `PXSmartPanel` control. As a key, the method uses the name of the variable that holds the BQL statement. The method requests repainting of the panel.

Syntax:

```
public WebDialogResult AskExt()
```

AskExt(string)

Displays the dialog window configured by the `PXSmartPanel` control. The method requests repainting of the panel.

Syntax:

```
public WebDialogResult AskExt(string key)
```

Parameters:

- `key`

  The identifier of the panel to display.

AskExt(bool)

Displays the dialog window configured by the `PXSmartPanel` control. As a key, the method uses the name of the variable that holds the BQL statement.

Syntax:

```
public WebDialogResult AskExt(bool refreshRequired)
```

Parameters:

- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt(InitializePanel)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(InitializePanel initializeHandler)
```

Parameters:

- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.

AskExt(string, bool)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(string key, bool refreshRequired)
```

Parameters:

- `key`

  The identifier of the panel to display.

- `refreshRequired`

The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt(string, InitializePanel)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(string key, InitializePanel initializeHandler)
```

Parameters:

- `key`

  The identifier of the panel to display.
- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.

AskExt(InitializePanel, bool)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(InitializePanel initializeHandler,
                              bool refreshRequired)
```

Parameters:

- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.
- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt(string, InitializePanel, bool)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public WebDialogResult AskExt(string key,
                              InitializePanel initializeHandler,
                              bool refreshRequired)
```

Parameters:

- `key`

  The identifier of the panel to display.
- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.
- `refreshRequired`

  The value that indicates whether the dialog should be repainted or displayed as it was cached. If `true`, the dialog is repainted.

AskExt(PXGraph, string, string, InitializePanel)

Displays the dialog window configured by the `PXSmartPanel` control.

Syntax:

```
public static WebDialogResult AskExt(PXGraph graph, string viewName,
                                     string key,
                                     InitializePanel initializeHandler)
```

Parameters:

- `graph`

  The graph where the data view is defined.
- `viewName`

  The name of the data view with which the dialog is associated.
- `key`

  The identifier of the panel to display.
- `initializeHandler`

  The delegate of the method that is called before the dialog is displayed.

Clear()

Clears the results of BQL statement execution.

Syntax:

```
public virtual void Clear()
```

ClearDialog()

Clears the dialog information saved by the graph on last invocation of the `Ask()` method.

Syntax:

```
public void ClearDialog()
```

DetachCache()

Initialize a new cache for storing the results of BQL statement execution.

Syntax:

```
public void DetachCache()
```

EnumParameters()

Returns the information on the fields referenced by BQL parameters and parameters of the optional method, if it is defined for the data view.

Syntax:

```
public virtual List<PXViewParameter> EnumParameters()
```

FieldGetValue(PXCache, object, Type, string)

Gets the value of the specified field in the data record from the cache.

The method may raise the `FieldDefaulting` and `FieldUpdating` events.

Syntax:

```
public static object FieldGetValue(PXCache sender, object data,
                                   Type sourceType, string sourceField)
```

Parameters:

- `sender`

  The cache object.
- `data`

  The data record.
- `sourceType`

  The DAC of the data record. The cache of this DAC type is obtained through the cache object provided in the parameter.
- `sourceField`

  The name of the field which value is returned.

Filter(IEnumerable)

Syntax:

```
public static IEnumerable Filter(IEnumerable list)
```

Parameters:

- `list`

GetAnswer(string)

Returns the result of the dialog window that was opened through one of the `Ask()` methods and saved in the `PXView` object.

Syntax:

```
public WebDialogResult GetAnswer(string key)
```

Parameters:

- `key`

  The identifier of the dialog window that was provided to the `Ask()` method or the name of the data view.

GetItemType()

Returns the DAC type of the primary cache; that is, the first DAC referenced in the BQL command.

Syntax:

```
public virtual Type GetItemType()
```

GetItemTypes()

Returns all DAC types referenced in the BQL command.

Syntax:

```
public virtual Type[] GetItemTypes()
```

GetParameterNames()

Returns the names of the fields referenced by BQL parameters and the names of parameters of the optional method, if it is defined.

Syntax:

```
public virtual string[] GetParameterNames()
```

GetSortColumns()

Returns pairs of the names of the fields by which the data view result will be sorted and values indicating if the sort by the field is descending.

Syntax:

```
public virtual KeyValuePair<string, bool>[] GetSortColumns()
```

Join(Type)

Appends the provided join clause to the BQL command.

Syntax:

```
public void Join(Type join)
```

Parameters:

- `join`

  The join clause as a type derived from `IBqlJoin`.

Join<join>()

Appends the provided join clause to the BQL command. The join clause is specified in the type parameter.

Syntax:

```
public void Join<join>()
    where join : IBqlJoin, new()
```

OrderByNew(Type)

Replaces the sorting expression with the new sorting expression.

Syntax:

```
public void OrderByNew(Type newOrderBy)
```

Parameters:

- `newOrderBy`

  The sorting expression as a type derived from `IBqlOrderBy`, such as `OrderBy<>`.

OrderByNew<newOrderBy>()

Replaces the sorting expression with the new sorting expression. The sorting expressio is specified in the type parameter.

Syntax:

```
public void OrderByNew<newOrderBy>()
    where newOrderBy : IBqlOrderBy, new()
```

PrepareParameters(object[], object[])

Prepares parameters, formats input values, gets default values for the hidden and not supplied parameters. The method returns the values that will replace the parameters including and the parameters of the custom selection method if it is defined.

Syntax:

```
public virtual object[] PrepareParameters(object[] currents, object[] parameters)
```

Parameters:

- `currents`

  The objects to use as current data records when processing `Current` and `Optional` parameters.

- `parameters`

  The explicit values for such parameters as `Required`, `Optional`, and `Argument`.

RequestRefresh()

Raises the `RequestRefresh` event defined within the `PXView` object.

Syntax:

```
public void RequestRefresh()
```

Select(object[], object[], object[], string[], bool[], PXFilterRow[], ref int, int, ref int)

Executes the BQL command and returns the result set.

This method is the main procedure for retrieving data. All other select methods eventually invoke these methods with appropriate parameters. The method can be used to retrieve all data records from the data set, the top data record, or the limited amount of data records starting from the specific position. You can also provide the list of current data records, the fields to additionally sort and filter the data set, and the parameters.

The method stores the values of parameters in the context, so that the optional method, if it is defined, of the data view can access them through the *static properties* of `PXView`.

Syntax:

```
public virtual List<object> Select(
    object[] currents, object[] parameters,
    object[] searches, string[] sortcolumns,
    bool[] descendings, PXFilterRow[] filters,
    ref int startRow, int maximumRows, ref int totalRows)
```

Parameters:

- `currents`

  The objects to use as current data records to process `Current` and `Optional` parameters.

- `parameters`

  The explicit values for such parameters as `Required`, `Optional`, and `Argument`.
- `searches`

  The values of the fields by which the data set is filtered and sorted.
- `sortcolumns`

  The fields by which the data set is filtered and sorted.
- `descendings`

  The list values indicating whether ordering by the sort columns should be descending or ascending.
- `filters`

  The filters.
- `(ref) startRow`

  The 0-based index of the first data record to retrieve.
- `maximumRows`

  The number of data records to retrieve.
- `(ref) totalRows`

  The total amount of data records in the data set defined by the BQL command.

SelectMulti(params object[])

Retrieves the whole data set corresponding to the BQL command.

Syntax:

```
public virtual List<object> SelectMulti(params object[] parameters)
```

Parameters:

- `parameters`

  The explicit values for such parameters as `Required`, `Optional`, and `Argument`.

SelectMultiBound(object[], params object[])

Retrieves the whole data set corresponding to the BQL command.

Syntax:

```
public virtual List<object> SelectMultiBound(object[] currents,
                                             params object[] parameters)
```

Parameters:

- `currents`

  The objects to use as current data records when processing `Current` and `Optional` parameters.
- `parameters`

  The explicit values for such parameters as `Required`, `Optional`, and `Argument`.

SelectSingle(params object[])

Retrieves the top data record from the data set corresponding to the BQL command.

Syntax:

```
public virtual object SelectSingle(params object[] parameters)
```

Parameters:

- `parameters`

  The explicit values for such parameters as `Required`, `Optional`, and `Argument`.

SelectSingleBound(object[], params object[])

Retrieves the top data record from the data set corresponding to the BQL command.

Syntax:

```
public virtual object SelectSingleBound(object[] currents,
                                        params object[] parameters)
```

Parameters:

- `currents`

  The objects to use as current data records when processing `Current` and `Optional` parameters.

- `parameters`

  The explicit values for such parameters as `Required`, `Optional`, and `Argument`.

Returns:

The resultset.

SetAnswer(string, WebDialogResult)

Saves the result of the dialog window.

Syntax:

```
public void SetAnswer(string key, WebDialogResult answer)
```

Parameters:

- `key`

  The identifier of the dialog window.

- `answer`

  The result value.

SetAnswer(PXGraph, string, string, WebDialogResult)

Saves the result of the dialog window.

Syntax:

```
public static void SetAnswer(PXGraph graph, string viewName,
                             string key, WebDialogResult answer)
```

Parameters:

- `graph`

  The graph with which the data view is associated.

- `viewName`

The name of the data view with which the dialog window is associated.

- `key`

  The identifier of the dialog window.

- `answer`

  The result value.

Sort(IEnumerable)

Sort the provided collection of `PXResult<>` instances by the conditions currenlty stored in the `PXView` context. This context exists only during execution of the *Select(...)* method. The `Sort(IEnumerable)` method may be called in the optional method of the data view to sort by the conditions that were provided to the `Select(...)` method, which invoked the optional method.

Syntax:

```
public static IEnumerable Sort(IEnumerable list)
```

Parameters:

- `list`

  The collection of `PXResult<>` instances to sort.

ToString()

Returns the string with the SQL query corresponding to the underlying BQL command.

Syntax:

```
public override string ToString()
```

WhereAnd(Type)

Appends a filtering expression to the underlying BQL command via the logical "and". The additional filtering expression is provided in the type parameter.

Syntax:

```
public void WhereAnd(Type where)
```

Parameters:

- `where`

  The additional filtering expression as the type derived from `IBqlWhere`.

WhereAnd<TWhere>()

Appends a filtering expression to the underlying BQL command via the logical "and". The additional filtering expression is provided in the type parameter.

Syntax:

```
public void WhereAnd<TWhere>()
    where TWhere : IBqlWhere, new()
```

WhereNew(Type)

Replaces the filtering expression in the BQL statement.

Syntax:

```
public void WhereNew(Type newWhere)
```

Parameters:

- `newWhere`

  The new filtering expression as the type derived from `IBqlWhere`.

WhereNew<newWhere>()

Replaces the filtering expression in the BQL statement. The new filtering expression is provided in the type parameter.

Syntax:

```
public void WhereNew<newWhere>()
    where newWhere : IBqlWhere, new()
```

WhereNot()

Adds logical "not" to the whole `Where` clause of the BQL statement, reversing the condition to the opposite.

Syntax:

```
public void WhereNot()
```

WhereOr(Type)

Appends a filtering expression to the BQL statement via the logical "or".

Syntax:

```
public void WhereOr(Type where)
```

Parameters:

- `where`

  The additional filtering expression as the type derived from `IBqlWhere`.

WhereOr<TWhere>()

Appends a filtering expression to the BQL statement via the logical "or". The additional filtering expression is provided in the type parameter.

Syntax:

```
public void WhereOr<TWhere>()
    where TWhere : IBqlWhere, new()
```

Attributes

Acumatica Framework attributes are used to add common business logic to the application components. This reference describes the attributes defined in the `PX.Data` namespace.

Attributes implement business logic by subscribing to events. Each attribute class directly or indirectly derives from the `PXEventSubscriberAttribute` class. Besides, an attribute class derives from the interfaces that correspond to the event handlers it implements. For example, the `PXDefault` attributes derives from the `IPXFieldDefaultingSubscriber`, `IPXRowPersistingSubscriber`, and `IPXFieldSelectingSubscriber` interfaces, which means that it implements its logic in the `FieldDefaulting`, `RowSelecting`, and `FieldSelecting` event handler methods.

Most attributes are added to data access class (DAC) field declarations. There are also attributes that are placed on a DAC declaration, view declarations in a business logic controller (BLC), and the BLC declaration itself.

Categories of Attributes

The attributes are split into a number of categories according to their usage or function.

- *Bound Field Data Types*
- *Unbound Field Data Types*
- *UI Field Configuration*
- *Default Values*
- *Complex Input Controls*
- *Referential Integrity and Calculations*
- *Audit Fields*
- *Data Projection*
- *Adhoc SQL for Fields*
- *Access Control*
- *Notes*
- *Report Optimization*
- *Attributes on DACs*
- *Attributes on Actions*
- *Attributes on Data Views*
- *Miscellaneous*

Mandatory Attributes

For each field defined in a DAC, you must specify the following attributes:

- A data type attribute – either a *bound field data type* attribute that binds the field to a database column of a particular data type, or an *unbound field data type* attribute that indicates that the field is unbound.
- The *PXUIField* attribute – mandatory for all fields that are displayed in the user interface.

The example below demonstrates a declaration of a DAC field bound to a database column and displayed in the user interface.

```
// The data access class for the POReceiptFilter database table
[Serializable]
public partial class POReceiptFilter : IBqlTable
{
    ...
    // The type declaration of a DAC field
    public abstract class receiptType : PX.Data.IBqlField
    {
    }
    // The value declaration of a DAC field - put attributes
    // before this declaration
```

```
    [PXDBString(2, IsFixed = true)]
    [PXUIField(DisplayName = "Type", Enabled = false)]
    public virtual String ReceiptType { get; set; }
    ...
}
```

A declaration of the method that implements an action in a business logic controller must be preceded with the *PXButton attribute or one of its successors*.

How to Use Attributes

To apply the attribute logic to an entity, you should place the attribute on the entity declaration. At run time, you can call the static methods of a particular attribute to adjust attribute's behavior.

An attribute may be placed on a declaration of a class or a class member, with or without parameters. Which paremeters are possible for an attribute depend on the constructor parameters and the properties defined in the attribute. The parameters of the selected constructor go first without names, named property settings follow them, as shown in the example below.

```
[PXDefault(false, PersistingCheck = PXPersistingCheck.Nothing)]
public virtual Boolean? Released { get; set; }
```

Here, the `PXDefault` attribute is created using the constructor that takes the only parameter of the boolean type (set to `false`). Additionally, the `PersistingCheck` property is specified.

You should call static methods defined in the attribute class to change the properties at run time. The static methods can affect a single attribute instance or multiple attribute instances related to a specific data record or all data records in a particular cache object. The following example shows an invocation of a static method.

```
PXUIFieldAttribute.SetVisible<APInvoice.curyID>(cache, doc, true);
```

When calling such a method, you typically specify the cache object, a data record related to this cache object, and the DAC field. The method will affect the attribute instance created for this field for the specified data record. If you pass `null` as the data record, the method will affect attribute instances related to all data records in the specified cache object.

Bound Field Data Types

The following attributes bind a data access class field to the database column of a specific type.

Attribute	C# data type	Database data type	Comment
PXDBBool	`bool?`	`bit`	Boolean value
PXDBByte	`byte?`	`tinyint`	1-byte integer value
PXDBDate	`DateTime?`	`datetime` or `smalldatetime`	Date and time
PXDBTime	`DateTime?`	`smalldatetime`	Time without date
PXDBDateAndTime	`DateTime?`	`datetime` or `smalldatetime`	Date and time values represented by separate input controls in the user interface
PXDBDecimal	`decimal?`	`decimal`	16-byte floating point numeric value with a specific precision
PXDBDecimalString	`decimal?`	`decimal`	A decimal value with a value selected by a user from the list of predefined values

Attribute	C# data type	Database data type	Comment
PXDBDouble	`double?`	`float`	8-byte floating point value
PXDBFloat	`float?`	`real`	4-byte floating point value
PXDBGuid	`Guid?`	`uniqueidentifier`	16-byte unique value
PXDBIdentity	`int?`	`int`	4-byte auto-incremented integer value
PXDBLongIdentity	`int64?`	`bigint`	8-byte auto-incremented integer value
PXDBShort	`short?`	`smallint`	2-byte integer value
PXDBInt	`int?`	`int`	4-byte integer value
PXDBLong	`int64?`	`bigint`	8-byte integer value
PXDBString	`string`	`char`, `varchar`, `nchar`, or `nvarchar`	Common string
PXDBEmail	`string`	`nvarchar`	Email address
PXDBLocalString	`string`	`char`, `varchar`, `nchar`, or `nvarchar`	Localized string
PXDBCryptString	`string`		Encrypted string
PXDB3DesCryphString	`string`		Specially encrypted string
PXDBText	`string`	`nvarchar` or `varchar`	Text
PXDBTimeSpan	`int?`	`int`	Date and time value represented by minutes passed from 01/01/1900
PXDBTimeSpanLong	`int?`	`int`	Duration in time as the number of minutes
PXDBTimestamp	`byte[]`	`timestamp`	8-byte automatically generated, unique binary numbers within a database
PXDBBinary	`byte[]`		Arbitrary array of bytes
PXDBVariant	`byte[]`	`variant`	Variant data type

Note that there are some other attributes that bind a DAC field to database columns, used in special cases. These attributes are covered in other sections of this reference.

PXDBField Attribute

The base class for attributes that map DAC fields to database columns.The attribute should not be used directly.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`

- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily(typeof(PXDBFieldAttribute))]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXDBFieldAttribute : PXEventSubscriberAttribute,
                                  IPXRowSelectingSubscriber,
                                  IPXCommandPreparingSubscriber
```

Properties

- `public virtual string` **`DatabaseFieldName`**

  Gets or sets the name of the database column that is represented by the field. By default, equals the field name.
- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field. Key fields must uniquely identify a data record. The key fields defined in the DAC should not necessarily be the same as the keys in the database.
- `public virtual bool` **`IsImmutable`**

  Gets or sets the values that indicates that the field is immutable.
- `public virtual Type` **`BqlField`**

  Returns `null` on get. Sets the BQL field representing the field in BQL queries.

PXDBBool Attribute

Maps a DAC field of `bool?` type to the database column of `bit` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBBoolAttribute : PXDBFieldAttribute,
                                 IPXRowSelectingSubscriber,
                                 IPXCommandPreparingSubscriber,
                                 IPXFieldUpdatingSubscriber,
```

```
                                IPXFieldSelectingSubscriber
```

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBBool()]
[PXDefault(false)]
public virtual Boolean? Scheduled { get; set; }
```

PXDBByte Attribute

Maps a DAC field of `byte?` type to the database column of `tinyint` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBByteAttribute : PXDBFieldAttribute,
                                 IPXRowSelectingSubscriber,
                                 IPXCommandPreparingSubscriber,
                                 IPXFieldUpdatingSubscriber,
                                 IPXFieldSelectingSubscriber
```

Properties

- `public int` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public int` **`MaxValue`**

  Gets or sets the maximum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

PXDBDate Attribute

Maps a DAC field of `DateTime?` type to the database column of `datetime` or `smalldatetime` type, depending on the `UseSmallDateTime` flag.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBDateAttribute : PXDBFieldAttribute,
                                 IPXRowSelectingSubscriber,
                                 IPXCommandPreparingSubscriber,
                                 IPXFieldUpdatingSubscriber,
                                 IPXFieldSelectingSubscriber
```

Properties

- `public string` **`InputMask`**

  Gets or sets the format string that defines how a field value inputted by a user should be formatted. The property takes the same values as `DisplayMask`.

- `public string` **`DisplayMask`**

  Gets or sets the format string that defines how a field value is displayed in the input control. If the property is set to a one-character string, the corresponding *standard date and time format string* is used. If the property value is longer, it is treated as a *custom date and time format string*. A particular pattern depends on the culture set by the application.

- `public string` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public string` **`MaxValue`**

  Gets or sets the maximum value for the field.

- `public virtual bool` **`PreserveTime`**

  Gets or sets the value that indicates whether the time part of a field value is preserved. If `false`, the time part is removed.

- `public bool` **`UseSmallDateTime`**

  Gets or sets the value that indicates the database column data type: `true` means `smalldatetime`, `false` means `datetime`. By default, `true`.

- `public virtual bool` **`UseTimeZone`**

Gets or sets the value that indicates whether the attribute should convert the time to UTC, using the local time zone. If `true`, the time is converted. By default, `true`.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

The attribute defines a field represented by a single input control in the user interface.

Examples

The attribute below binds the field to the database column and sets the minimum and maximum values for a field value.

```
[PXDBDate(MaxValue = "06/06/2079", MinValue = "01/01/1900")]
public virtual DateTime? OrderDate { get; set; }
```

The attribute below binds the field to the database column and sets the input and display masks. A field value will be displayed using the long date pattern. That is, for en-US culture the *6/15/2009 1:45:30 PM* value will be converted to *Monday, June 15, 2009*.

```
[PXDBDate(InputMask = "d", DisplayMask = "d")]
public virtual DateTime? StartDate { get; set; }
```

PXDBTime Attribute

Maps a DAC field of `DateTime?` type to the database column of `smalldatetime` type. The field value holds only time without date.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDateAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBTimeAttribute : PXDBDateAttribute
```

Properties

- `public override bool` **`PreserveTime`**

  Gets the value that indicates whether the time part of a field value is preserved. Since the constructor sets this value to `true`, this property always returns `true`.

Constructors

- `public PXDBTimeAttribute()`

  Initializes an instance of the attribute with default parameters.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

The field values keep only time without date. On the user interface, the field is represented by a control allowing a user to enter only a time value.

The attribute inherits properties of the *PXDBDate* attribute.

Examples

The code below binds the `SunStartTime` DAC field to the database column with the same name and sets the default value for the field.

```
[PXDBTime(DisplayMask = "t", UseTimeZone = false)]
[PXDefault(TypeCode.DateTime, "01/01/2008 09:00:00")]
public virtual DateTime? SunStartTime { ... }
```

Note the setting of the *DisplayMask* property inherited from the `PXDBDate` attribute.

PXDBDateAndTime Attribute

Maps a DAC field of `DateTime?` type to the database column of `datetime` or `smalldatetime` type. Defines the DAC field that is represented in the UI by two input controls: one for date, the other for time.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDateAttribute
```

Syntax

```
public class PXDBDateAndTimeAttribute : PXDBDateAttribute
```

Properties

- `public virtual bool` **`WithoutDisplayNames`**

  Gets or sets the value that indicates whether the display names of the input controls for date and time are appended with *(Date)* and *(Time)*, respectively.

- `public string` **`DisplayNameDate`**

  Gets or sets the display name for the input control that represents date.

- `public string` **`DisplayNameTime`**

  Gets or sets the display name for the input control that represents time.

Constructors

Constructor	Description
PXDBDateAndTimeAttribute()	Initializes a new instance of the attribute with default parameters.

Static Methods

Method	Description
SetDateDisplayName(PXCache, object, string, string)	Sets the display name of the input control that represents the date part of the field value
SetDateDisplayName<Field>(PXCache, object, string)	Sets the display name of the input control that represents the date part of the field value
SetDateEnabled(PXCache, object, string, bool)	Enables or disables the input control that represents the date part of the field value
SetDateEnabled<Field>(PXCache, object, bool)	Enables or disables the input control that represents the date part of the field value
SetDateVisible(PXCache, object, string, bool)	Makes visible or hides the input control that represents the data part of the field value
SetDateVisible<Field>(PXCache, object, bool)	Makes visible or hides the input control that represents the data part of the field value
SetTimeDisplayName(PXCache, object, string, string)	Sets the display name of the input control that represents the time part of the field value
SetTimeDisplayName<Field>(PXCache, object, string)	Sets the display name of the input control that represents the time part of the field value
SetTimeEnabled(PXCache, object, string, bool)	Enables or disables the input control that represents the time part of the field value
SetTimeEnabled<Field>(PXCache, object, bool)	Enables or disables the input control that represents the time part of the field value
SetTimeVisible(PXCache, object, string, bool)	Makes visible or hides the input control that represents the time part of the field value
SetTimeVisible<Field>(PXCache, object, bool)	Makes visible or hides the input control that represents the data part of the field value

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Unlike the *PXDBDate* attribute, this attribute defines the field that is represented in the UI by two input controls to specify date and time values separately.

Examples

```
[PXDBDateAndTime]
[PXUIField(DisplayName = "Start Time")]
public virtual DateTime? StartDate { get; set; }
```

PXDBDateAndTime Attribute Constructors

The *PXDBDateAndTime* attribute exposes the following constructors.

PXDBDateAndTimeAttribute()

Initializes a new instance of the attribute with default parameters.

Syntax:

```
public PXDBDateAndTimeAttribute()
```

PXDBDateAndTime Attribute Methods

The *PXDBDateAndTime* attribute exposes the following static methods.

SetDateDisplayName(PXCache, object, string, string)

Sets the display name of the input control that represents the date part of the field value.

Syntax:

```
public static void SetDateDisplayName(PXCache cache, object data,
                                      string name, string displayName)
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `name`

  The name of the field the attribute is attached to.
- `displayName`

  The string to set as the display name.

SetDateDisplayName<Field>(PXCache, object, string)

Sets the display name of the input control that represents the date part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetDateDisplayName<Field>(PXCache cache, object data,
                                             string displayName)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `displayName`

  The string to set as the display name.

SetDateEnabled(PXCache, object, string, bool)

Enables or disables the input control that represents the date part of the field value.

Syntax:

```
public static void SetDateEnabled(PXCache cache, object data,
                                  string name, bool isEnabled)
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The name of the field the attribute is attached to.

- `isEnabled`

  The value indicating whether the input control is enabled.

SetDateEnabled<Field>(PXCache, object, bool)

Enables or disables the input control that represents the date part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetDateEnabled<Field>(PXCache cache, object data,
                                         bool isEnabled)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isEnabled`

  The value indicating whether the input control is enabled.

SetDateVisible(PXCache, object, string, bool)

Makes visible or hides the input control that represents the data part of the field value.

Syntax:

```
public static void SetDateVisible(PXCache cache, object data,
                                  string name, bool isVisible)
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.

- `data`

The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The name of the field the attribute is attached to.

- `isVisible`

  The value indicating whether the input control is visible on the user interface.

SetDateVisible<Field>(PXCache, object, bool)

Makes visible or hides the input control that represents the data part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetDateVisible<Field>(PXCache cache, object data,
                                          bool isVisible)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isVisible`

  The value indicating whether the input control is visible on the user interface.

SetTimeDisplayName(PXCache, object, string, string)

Sets the display name of the input control that represents the time part of the field value.

Syntax:

```
public static void SetTimeDisplayName(PXCache cache, object data,
                                      string name, string displayName)
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The name of the field the attribute is attached to.

- `displayName`

  The string to set as the display name.

SetTimeDisplayName<Field>(PXCache, object, string)

Sets the display name of the input control that represents the time part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetTimeDisplayName<Field>(PXCache cache, object data,
                                              string displayName)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `displayName`

  The string to set as the display name.

SetTimeEnabled(PXCache, object, string, bool)

Enables or disables the input control that represents the time part of the field value.

Syntax:

```
public static void SetTimeEnabled(PXCache cache, object data,
                                  string name, bool isEnabled)
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `name`

  The name of the field the attribute is attached to.
- `isEnabled`

  The value indicating whether the input control is enabled.

SetTimeEnabled<Field>(PXCache, object, bool)

Enables or disables the input control that represents the time part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetTimeEnabled<Field>(PXCache cache, object data,
                                         bool isEnabled)
    where Field : IBqlField
```

Parameters:

- `cache`

The cache object to search for `PXDBDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isEnabled`

  The value indicating whether the input control is enabled.

SetTimeVisible(PXCache, object, string, bool)

Makes visible or hides the input control that represents the time part of the field value.

Syntax:

```
public static void SetTimeVisible(PXCache cache, object data,
                                  string name, bool isVisible)
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The name of the field the attribute is attached to.

- `isVisible`

  The value indicating whether the input control is visible on the user interface.

SetTimeVisible<Field>(PXCache, object, bool)

Makes visible or hides the input control that represents the data part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetTimeVisible<Field>(PXCache cache, object data,
                                         bool isVisible)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for `PXDBDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isVisible`

  The value indicating whether the input control is visible on the user interface.

PXDBDecimal Attribute

Maps a DAC field of `decimal?` type to the database column of `decimal` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBDecimalAttribute : PXDBFieldAttribute,
                                    IPXRowSelectingSubscriber,
                                    IPXCommandPreparingSubscriber,
                                    IPXFieldUpdatingSubscriber,
                                    IPXFieldSelectingSubscriber
```

Properties

- `public double` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public double` **`MaxValue`**

  Gets or sets the minimum value for the field.

Constructors

Constructor	Description
PXDBDecimalAttribute()	Initializes a new instance with the default precision, which equals 2
PXDBDecimalAttribute(int)	Initializes a new instance with the given precision
PXDBDecimalAttribute(Type)	Initializes a new instance with the precision calculated at runtime using a BQL query

Static Methods

Method	Description
EnsurePrecision(PXCache)	Retrieves the precision value if it is set by a BQL query specified in the constructor, and sets its to all attribute instances in the cache object
SetPrecision(PXCache, string, int?)	Sets the precision in the attribute intance that marks the field with the specified name in all data records in the cache object

Method	Description
SetPrecision(PXCache, object, string, int?)	Sets the precision in the attribute intance that marks the field with the specified name in a particular data record

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

A minimum value, maximum value, and precision can be specified. The precision can be calculated at runtime using BQL. The default precision is 2.

Examples

Declaration of a DAC field with a specific precision:

```
[PXDBDecimal(6, MinValue = 0, MaxValue = 100)]
public virtual decimal? Price { get; set; }
```

Declaration of a DAC field with a precision calculated at runtime:

```
[PXDBDecimal(typeof(
    Search<Currency.decimalPlaces,
        Where<Currency.curyID, Equal<Current<POCreateFilter.vendorID>>>>
))]
public virtual decimal? OrderTotal { get; set; }
```

The BQL query in this example will search for the `Currency` data record that satisfies the specified `Where` condition. The field precision will be set to the `DecimalPlaces` value from this data record.

PXDBDecimal Attribute Constructors

The *PXDBDecimal* attribute exposes the following constructors.

PXDBDecimalAttribute()

Initializes a new instance with the default precision, which equals 2.

Syntax:

```
public PXDBDecimalAttribute()
```

PXDBDecimalAttribute(int)

Initializes a new instance with the given precision.

Syntax:

```
public PXDBDecimalAttribute(int precision)
```

PXDBDecimalAttribute(Type)

Initializes a new instance with the precision calculated at runtime using a BQL query.

Syntax:

```
public PXDBDecimalAttribute(Type type)
```

Parameters:

- `type`

A BQL query based on a class derived from `IBqlSearch` or `IBqlField`. For example, the parameter can be set to `typeof(Search<...>)`, or `typeof(Table.field)`.

PXDBDecimal Attribute Methods

The *PXDBDecimal* attribute exposes the following static methods.

EnsurePrecision(PXCache)

Retrieves the precision value if it is set by a BQL query specified in the constructor, and sets its to all attribute instances in the cache object.

Syntax:

```
public static void EnsurePrecision(PXCache cache)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBDecimal` type.

SetPrecision(PXCache, string, int?)

Sets the precision in the attribute intance that marks the field with the specified name in all data records in the cache object.

Syntax:

```
public static void SetPrecision(PXCache cache, string name, int? precision)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBDecimal` type.
- `name`

  The name of the field that is be marked with the attribute.
- `precision`

  The new precision value.

SetPrecision(PXCache, object, string, int?)

Sets the precision in the attribute intance that marks the field with the specified name in a particular data record.

Syntax:

```
public static void SetPrecision(PXCache cache, object data, string name, int?
            precision)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBDecimal` type.
- `data`

  The data record the method is applied to.
- `name`

The name of the field that is be marked with the attribute.

- `precision`

  The new precision value.

PXDBDecimalString Attribute

Maps a DAC field of `decimal?` type to the database column of `decimal` type. The mapped DAC field can be represented in the UI by a dropdown list using the *PXDecimalList* attribute.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDecimalAttribute
```

Syntax

```
public class PXDBDecimalStringAttribute : PXDBDecimalAttribute
```

Constructors

Constructor	Description
PXDBDecimalStringAttribute()	Initializes a new instance with the default precision, which equals 2
PXDBDecimalStringAttribute(int)	Initializes a new instance with the given decimal value precision

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

In the UI, the field can be represented by a dropdown list with specific values. The UI control is configured using the `PXDecimalList` attribute.

Examples

```
// A mapping of the DAC field to the database column
[PXDBDecimalString(1)]
// UI control configuration.
// The first list configures values asigned to the field,
// the second one configures displayed labels.
[PXDecimalList(new string[] { "0.1", "0.5", "1.0", "10", "100" },
               new string[] { "0.1", "0.5", "1.0", "10", "100" })]
[PXDefault(TypeCode.Decimal, "0.1")]
[PXUIField(DisplayName = "Invoice Amount Precision")]
public virtual decimal? InvoicePrecision { get; set; }
```

PXDBDecimalString Attribute Constructors

The *PXDBDecimalString* attribute exposes the following constructors.

PXDBDecimalStringAttribute()

Initializes a new instance with the default precision, which equals 2.

Syntax:

```
public PXDBDecimalStringAttribute() : base()
```

PXDBDecimalStringAttribute(int)

Initializes a new instance with the given decimal value precision.

Syntax:

```
public PXDBDecimalStringAttribute(int precision) : base(precision)
```

PXDBDouble Attribute

Maps a DAC field of `double?` type to the 8-bytes floating point column in the database.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBDoubleAttribute : PXDBFieldAttribute,
                                   IPXRowSelectingSubscriber,
                                   IPXCommandPreparingSubscriber,
                                   IPXFieldUpdatingSubscriber,
                                   IPXFieldSelectingSubscriber
```

Properties

- `public double` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public double` **`MaxValue`**

  Gets or sets the maximum value for the field.

Constructors

Constructor	Description
PXDBDoubleAttribute()	Initializes a new instance of the attribute with default parameters
PXDBDoubleAttribute(int)	Initializes a new instance of the attribute with the given precision

Static Methods

Method	Description
SetPrecision(PXCache, string, int)	
SetPrecision(PXCache, object, string, int)	

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

PXDBDouble Attribute Constructors

The *PXDBDouble* attribute exposes the following constructors.

PXDBDoubleAttribute()

Initializes a new instance of the attribute with default parameters.

Syntax:

```
public PXDBDoubleAttribute()
```

PXDBDoubleAttribute(int)

Initializes a new instance of the attribute with the given precision. The precision is the number of digits after the comma. If a user enters a value with greater number of fractional digits, the value will be rounded.

Syntax:

```
public PXDBDoubleAttribute(int precision)
```

Parameters:

- `precision`

  The value to use as the precision.

PXDBDouble Attribute Methods

The *PXDBDouble* attribute exposes the following static methods.

SetPrecision(PXCache, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, string name, int
            precision)
```

SetPrecision(PXCache, object, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, object data, string name, int
            precision)
```

PXDBFloat Attribute

Maps a DAC field of `float?` type to the 4-bytes floating point column in the database.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBFloatAttribute : PXDBFieldAttribute,
                                  IPXRowSelectingSubscriber,
                                  IPXCommandPreparingSubscriber,
                                  IPXFieldUpdatingSubscriber,
                                  IPXFieldSelectingSubscriber
```

Properties

- `public float` **`MinValue`**

  Gets or sets the minimum value for the field.
- `public float` **`MaxValue`**

  Gets or sets the maximum value for the field.

Constructors

Constructor	Description
PXDBFloatAttribute()	Initializes a new instance with default parameters
PXDBFloatAttribute(int)	Initializes a new instance with the given precision

Static Methods

Method	Description
SetPrecision(PXCache, string, int)	
SetPrecision(PXCache, object, string, int)	

PXDBFloat Attribute Constructors

The *PXDBFloat* attribute exposes the following constructors.

PXDBFloatAttribute()

Initializes a new instance with default parameters.

Syntax:

```
public PXDBFloatAttribute()
```

PXDBFloatAttribute(int)

Initializes a new instance of the attribute with the given precision. The precision is the number of digits after the comma. If a user enters a value with greater number of fractional digits, the value will be rounded.

Syntax:

```
public PXDBFloatAttribute(int precision)
```

Parameters:

- `precision`

  The value to use as the precision.

PXDBFloat Attribute Methods

The *PXDBFloat* attribute exposes the following static methods.

SetPrecision(PXCache, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, string name, int precision)
```

SetPrecision(PXCache, object, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, object data, string name, int
 precision)
```

PXDBGuid Attribute

Map a DAC field of `Guid?` type to the database column of `uniqueidentifier` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXFieldDefaultingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
```

```
                    AttributeTargets.Parameter |
                    AttributeTargets.Class |
                    AttributeTargets.Method)]
public class PXDBGuidAttribute : PXDBFieldAttribute,
                                 IPXRowSelectingSubscriber,
                                 IPXCommandPreparingSubscriber,
                                 IPXFieldUpdatingSubscriber,
                                 IPXFieldSelectingSubscriber,
                                 IPXFieldDefaultingSubscriber
```

Constructors

Constructor	Description
PXDBGuidAttribute()	Initializes a new instance that does not assign a default value to the field
PXDBGuidAttribute(bool)	Initializes a new instance that either assigns a default value to the field or doesn't

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

The attribute below binds the field to the unique indentifier column and assigns a default value to the field.

```
[PXDBGuid(true)]
public virtual Guid? SetupID { get; set; }
```

The attribute below binds the field to the unique indentifier column. The field becomes a key field.

```
[PXDBGuid(IsKey = true)]
public virtual Guid? SetupID { get; set; }
```

PXDBGuid Attribute Constructors

The *PXDBGuid* attribute exposes the following constructors.

PXDBGuidAttribute()

Initializes a new instance that does not assign a default value to the field.

Syntax:

```
public PXDBGuidAttribute() : base() { }
```

PXDBGuidAttribute(bool)

Initializes a new instance that either assigns a default value to the field or doesn't.

Syntax:

```
public PXDBGuidAttribute(bool withDefaulting) : this()
```

Parameters:

- `withDefaulting`

If `true`, a new Guid value is assigned to the field on the `FieldDefaulting` event. Otherwise, a value is not assigned.

PXDBIdentity Attribute

Maps an auto-incremented integer DAC field of `int?` type to the `int` database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXFieldDefaultingSubscriber`
- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXRowPersistedSubscriber`
- `IPXFieldVerifyingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBIdentityAttribute : PXDBFieldAttribute,
                                     IPXFieldDefaultingSubscriber,
                                     IPXRowSelectingSubscriber,
                                     IPXCommandPreparingSubscriber,
                                     IPXFieldUpdatingSubscriber,
                                     IPXFieldSelectingSubscriber,
                                     IPXRowPersistedSubscriber,
                                     IPXFieldVerifyingSubscriber
```

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

The field value is auto-incremented by the attribute.

A field of this type is typically declared a key field. To do this, set the `IsKey` parameter to `true`.

Examples

```
[PXDBIdentity(IsKey = true)]
[PXUIField(DisplayName = "Contact ID", Visible = false)]
public virtual int? ContactID { get; set; }
```

PXDBLongIdentity Attribute

Maps an 8-byte auto-incremented integer DAC field of `int64?` type to the `bigint` database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXFieldDefaultingSubscriber`
- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXRowPersistedSubscriber`
- `IPXFieldVerifyingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBLongIdentityAttribute : PXDBFieldAttribute,
                                         IPXFieldDefaultingSubscriber,
                                         IPXRowSelectingSubscriber,
                                         IPXCommandPreparingSubscriber,
                                         IPXFieldUpdatingSubscriber,
                                         IPXFieldSelectingSubscriber,
                                         IPXRowPersistedSubscriber,
                                         IPXFieldVerifyingSubscriber
```

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name. The field value is auto-incremented by the database.

A field of this type is typically declared a key field. To do this, set the `IsKey` parameter to `true`.

Examples

```
[PXDBLongIdentity(IsKey = true)]
public virtual Int64? RecordID { ... }
```

PXDBImage Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
```

Syntax

```
public class PXDBImageAttribute : PXDBStringAttribute
```

Properties

- `public string` **`HeaderImage`**

  Get, set.

PXDBShort Attribute

Maps a DAC field of `short?` type to the database column of `smallint` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBShortAttribute : PXDBFieldAttribute,
                                  IPXRowSelectingSubscriber,
                                  IPXCommandPreparingSubscriber,
                                  IPXFieldUpdatingSubscriber,
                                  IPXFieldSelectingSubscriber
```

Properties

- `public int` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public int` **`MaxValue`**

  Gets or sets the minimum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBShort(MaxValue = 9, MinValue = 0)]
public virtual short? TaxReportPrecision { get; set; }
```

PXDBInt Attribute

Maps a DAC field of `int?` type to the database column of `int` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBIntAttribute : PXDBFieldAttribute,
                                IPXRowSelectingSubscriber,
                                IPXCommandPreparingSubscriber,
                                IPXFieldUpdatingSubscriber,
                                IPXFieldSelectingSubscriber
```

Properties

- `public int` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public int` **`MaxValue`**

  Gets or sets the maximum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBInt]
public virtual int? MajorStatus { get; set; }
```

The attribute below maps a field to the database column and explicitly sets the minimum and maximum values for the field.

```
[PXDBInt(MinValue = 0, MaxValue = 365)]
public virtual int? ReceiptTranDaysBefore { get; set; }
```

The attribute below maps a field to the database column and sets the properties inherited from the *PXDBField* attribute.

```
[PXDBInt(IsKey = true, BqlField = typeof(CuryARHistory.branchID))]
[PXSelector(typeof(Branch.branchID),
            SubstituteKey = typeof(Branch.branchCD))]
public virtual int? BranchID { get; set; }
```

PXDBLong Attribute

Maps a DAC field of `int64?` type to the database column of `bigint` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBLongAttribute : PXDBFieldAttribute,
                                 IPXRowSelectingSubscriber,
                                 IPXCommandPreparingSubscriber,
                                 IPXFieldUpdatingSubscriber,
                                 IPXFieldSelectingSubscriber
```

Properties

- `public Int64` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public Int64` **`MaxValue`**

  Gets or sets the maximum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBLong()]
public virtual long? CuryInfoID { get; set; }
```

PXDBString Attribute

Maps a DAC field of `string` type to the database field of `char`, `varchar`, `nchar`, or `nvarchar` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBStringAttribute : PXDBFieldAttribute,
                                   IPXRowSelectingSubscriber,
                                   IPXCommandPreparingSubscriber,
                                   IPXFieldUpdatingSubscriber,
                                   IPXFieldSelectingSubscriber
```

Properties

- `public int` **`Length`**

  Gets the maximum length of the string value. If a string value exceeds the maximum length, it will be trimmed. If `IsFixed` is set to `true` and the string length is less then the maximum, it will be extended with spaces.

  The default value is -1 (the string length is not limited). A different value can be set in the constructor.

- `public string` **`InputMask`**

  Gets or sets the pattern that indicates the allowed characters in a field value. The user interface will not allow the user to enter other characters in the input control associated with the field.

  The default value for the key fields is '`>aaaaaa`'.

  Control characters:

  - '>': the following chars to upper case
  - '<': the following chars to lower case
  - '`&`', '`C`': any character or a space
  - '`A`', '`a`': a letter or digit
  - '`L`', '`?`': a letter
  - '`#`', '`0`', '`9`': a digit

  Examples:

  ```
  InputMask = ">LLLLL"
  ```

  ```
  InputMask = ">aaaaaaaaaa"
  ```

  ```
  InputMask = ">CC.00.00.00"
  ```

- `public bool` **`IsUnicode`**

Gets or sets an indication that the string consists of Unicode characters. This property should be set to `true` if the database column has a Unicode string type (`nchar` or `nvarchar`). The default value is `false`.

- `public bool` **`IsFixed`**

  Gets or sets an indication that the string has a fixed length. This property should be set to `true` if the database column has a fixed length type (`char` or `nchar`). The default value is `false`.

Constructors

Constructor	Description
PXDBStringAttribute()	Initializes a new instance of the attribute
PXDBStringAttribute(int)	Initializes a new instance with the given maximum length of a field value

Static Methods

Method	Description
SetInputMask(PXCache, string, string)	Sets the input mask for the string field with the specified name for all data records in the cache object
SetInputMask(PXCache, object, string, string)	Sets the input mask for the string field with the specified name
SetInputMask<Field>(PXCache, string)	Sets the input mask for the specified string field for all data records in the cache object
SetInputMask<Field>(PXCache, object, string)	Sets the input mask for the specified string field
SetLength(PXCache, string, int)	Sets the maximum length for the string field with the specified name for all data records in the cache object
SetLength(PXCache, object, string, int)	Sets the maximum length for the string field with the specified name
SetLength<Field>(PXCache, int)	Sets the maximum length for the specified string field for all data records in the cache object
SetLength<Field>(PXCache, object, int)	Sets the maximum length for the specified string field

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

It is possible to specify the maximum length and input validation mask for the string.

You can modify the `Length` and `InputMask` properties at run time by calling the static methods.

Examples

The attribute below maps a string field to the database column (defines a bound field) and sets a limit for the value length to 50.

```
[PXDBString(50)]
public virtual string Fax { get; set; }
```

The attribute below defines a bound field taking as a value strings of any 8 characters. In the user interface, the input control will show the mask that splits the value into four groups separated by dots.

```
[PXDBString(8, InputMask = "CC.CC.CC.CC")]
public virtual string ReportID { get; set; }
```

The attribute below defines a bound field taking as a value Unicode strings of 5 uppercase characters that are strictly aphabetical letters.

```
[PXDBString(5, IsUnicode = true, InputMask = ">LLLLL")]
public virtual string CuryID { get; set; }
```

The example below shows a complex definition of a string key field represented in the user interface by a lookup control.

```
[PXDBString(15, IsUnicode = true, IsKey = true, InputMask = "")]
[PXDefault]
[PXUIField(DisplayName = "Reference Nbr.",
           Visibility = PXUIVisibility.SelectorVisible,
           TabOrder = 1)]
[PXSelector(typeof(
    Search<APRegister.refNbr,
        Where<APRegister.docType, Equal<Optional<APRegister.docType>>>>),
    Filterable = true)]
public virtual string RefNbr { get; set; }
```

In this example, the `RefNbr` field is mapped to the `nvarchar(15) RefNbr` column from the `APRegister` table.

PXDBString Attribute Constructors

The *PXDBString* attribute exposes the following constructors.

PXDBStringAttribute()

Initializes a new instance of the attribute.

Syntax:

```
public PXDBStringAttribute()
```

PXDBStringAttribute(int)

Initializes a new instance with the given maximum length of a field value.

Syntax:

```
public PXDBStringAttribute(int length)
```

Parameters:

- `length`

  The maximum length value assigned to the `Length` property.

PXDBString Attribute Methods

The *PXDBString* attribute exposes the following static methods.

SetInputMask(PXCache, string, string)

Sets the input mask for the string field with the specified name for all data records in the cache object.

Syntax:

```
public static void SetInputMask(PXCache cache, string name, string mask)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.

- `name`

  The field name.

- `mask`

  The value that is assigned to the `InputMask` property.

SetInputMask(PXCache, object, string, string)

Sets the input mask for the string field with the specified name.

Syntax:

```
public static void SetInputMask(PXCache cache, object data, string name, string
 mask)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

- `mask`

  The value that is assigned to the `InputMask` property.

SetInputMask<Field>(PXCache, string)

Sets the input mask for the specified string field for all data records in the cache object.

Syntax:

```
public static void SetInputMask<Field>(PXCache cache, string mask)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.

- `mask`

  The value that is assigned to the `InputMask` property.

SetInputMask<Field>(PXCache, object, string)

Sets the input mask for the specified string field.

Syntax:

```
public static void SetInputMask<Field>(PXCache cache, object data, string mask)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `mask`

  The value that is assigned to the `InputMask` property.

SetLength(PXCache, string, int)

Sets the maximum length for the string field with the specified name for all data records in the cache object.

Syntax:

```
public static void SetLength(PXCache cache, string name, int length)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.
- `name`

  The field name.
- `length`

  The value that is assigned to the `Length` property.

SetLength(PXCache, object, string, int)

Sets the maximum length for the string field with the specified name.

Syntax:

```
public static void SetLength(PXCache cache, object data, string name, int length)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `name`

  The field name.
- `length`

  The value that is assigned to the `Length` property.

SetLength<Field>(PXCache, int)

Sets the maximum length for the specified string field for all data records in the cache object.

Syntax:

```
public static void SetLength<Field>(PXCache cache, int length)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.
- `length`

  The value that is assigned to the `Length` property.

SetLength<Field>(PXCache, object, int)

Sets the maximum length for the specified string field.

Syntax:

```
public static void SetLength<Field>(PXCache cache, object data, int length)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBString` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `length`

  The value that is assigned to the `Length` property.

PXDBEmail Attribute

Maps a `string` DAC field representing email addresses to the database column of `nvarchar` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Method)]
public class PXDBEmailAttribute : PXDBStringAttribute
```

Constructors

- `public PXDBEmailAttribute() : base(255)`

  Initializes a new instance of the attribute. The maximum string length is set to 255. The string is marked as Unicode.

Static Methods

Method	Description
GetEMailFields(Type)	

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

The field value must be a Unicode string. The field value length is limited by 255.

Examples

```
[PXDBEmail]
[PXUIField(DisplayName = "Email",
           Visibility = PXUIVisibility.SelectorVisible)]
public virtual string Email { get; set; }
```

PXDBEmail Attribute Methods

The *PXDBEmail* attribute exposes the following static methods.

GetEMailFields(Type)

Syntax:

```
public static List<string> GetEMailFields(Type table)
```

PXDBLocalString Attribute

Maps a string DAC field to a localized string column in the database.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBLocalStringAttribute : PXDBStringAttribute
```

Constructors

Constructor	Description
PXDBLocalStringAttribute()	Initializes a new instance with the default parameters
PXDBLocalStringAttribute(int)	Initializes a new instance with the specified maximum length

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to database columns that have culture information specified in their names. For example, for the `Description` field, the English-specific column will be `DescriptionenGB`, the Russian-specific column `DescriptionruRU`, etc.

PXDBLocalString Attribute Constructors

The *PXDBLocalString* attribute exposes the following constructors.

PXDBLocalStringAttribute()

Initializes a new instance with the default parameters.

Syntax:

```
public PXDBLocalStringAttribute() : base()
```

PXDBLocalStringAttribute(int)

Initializes a new instance with the specified maximum length.

Syntax:

```
public PXDBLocalStringAttribute(int length) : base(length)
```

PXDBCryptString Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
```

Interfaces

- `IPXFieldVerifyingSubscriber`
- `IPXRowUpdatingSubscriber`
- `IPXRowSelectingSubscriber`

Syntax

```
public class PXDBCryptStringAttribute : PXDBStringAttribute,
                                        IPXFieldVerifyingSubscriber,
                                        IPXRowUpdatingSubscriber,
                                        IPXRowSelectingSubscriber
```

Properties

- `public bool` **`IsViewDecrypted`**

  Get, set.
- `public string` **`ViewAsString`**

  Get, set.
- `public Type` **`ViewAsField`**

  Get, set.

Constructors

Constructor	Description
PXDBCryptStringAttribute()	
PXDBCryptStringAttribute(int)	Initializes a new instance with the given maximum length

Static Methods

Method	Description
SetDecrypted(PXCache, string, bool)	Overrides the visible state for the particular data item
SetDecrypted(PXCache, object, string, bool)	Overrides the visible state for the particular data item
SetDecrypted<Field>(PXCache, bool)	Overrides the view as state for the particular data item
SetDecrypted<Field>(PXCache, object, bool)	Overrides the visible state for the particular data item
SetViewAs(PXCache, string, string)	Overrides the view as state for the particular data item
SetViewAs(PXCache, string, Type)	Overrides the view as state for the particular data item
SetViewAs(PXCache, object, string, string)	Overrides the view as state for the particular data item
SetViewAs(PXCache, object, string, Type)	Overrides the view as state for the particular data item
SetViewAs<Field>(PXCache, string)	Overrides the view as state for the particular data item
SetViewAs<Field>(PXCache, Type)	Overrides the view as state for the particular data item
SetViewAs<Field>(PXCache, object, string)	Overrides the view as state for the particular data item
SetViewAs<Field>(PXCache, object, Type)	Overrides the view as state for the particular data item

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

PXDBCryptString Attribute Constructors

The *PXDBCryptString* attribute exposes the following constructors.

PXDBCryptStringAttribute()

Syntax:

```
public PXDBCryptStringAttribute()
```

PXDBCryptStringAttribute(int)

Initializes a new instance with the given maximum length.

Syntax:

```
public PXDBCryptStringAttribute(int length) : base(length)
```

PXDBCryptString Attribute Methods

The *PXDBCryptString* attribute exposes the following static methods.

SetDecrypted(PXCache, string, bool)

Overrides the visible state for the particular data item.

Syntax:

```
public static void SetDecrypted(PXCache cache, string field, bool isDecrypted)
```

Parameters:

- `cache`

  Cache containing the data item.
- `def`

  Default value.

SetDecrypted(PXCache, object, string, bool)

Overrides the visible state for the particular data item.

Syntax:

```
public static void SetDecrypted(PXCache cache, object data, string field, bool
 isDecrypted)
```

Parameters:

- `cache`

  Cache containing the data item.
- `def`

  Default value.

SetDecrypted<Field>(PXCache, bool)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetDecrypted<Field>(PXCache cache, bool isDecrypted) where
 Field : IBqlField
```

Parameters:

- `cache`

  Cache containing the data item.
- `def`

  Default value.

SetDecrypted<Field>(PXCache, object, bool)

Overrides the visible state for the particular data item.

Syntax:

```
public static void SetDecrypted<Field>(PXCache cache, object data, bool isDecrypted)
 where Field : IBqlField
```

Parameters:

- `cache`

Cache containing the data item.

- `def`

  Default value.

SetViewAs(PXCache, string, string)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs(PXCache cache, string field, string source)
```

Parameters:

- `cache`

  Cache containing the data item.

- `def`

  Default value.

SetViewAs(PXCache, string, Type)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs(PXCache cache, string field, Type sourceField)
```

Parameters:

- `cache`

  Cache containing the data item.

- `def`

  Default value.

SetViewAs(PXCache, object, string, string)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs(PXCache cache, object data, string field, string
 source)
```

Parameters:

- `cache`

  Cache containing the data item.

- `def`

  Default value.

SetViewAs(PXCache, object, string, Type)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs(PXCache cache, object data, string field, Type
 sourceField)
```

Parameters:

- `cache`

  Cache containing the data item.
- `def`

  Default value.

SetViewAs<Field>(PXCache, string)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs<Field>(PXCache cache, string source) where Field :
 IBqlField
```

Parameters:

- `cache`

  Cache containing the data item.
- `def`

  Default value.

SetViewAs<Field>(PXCache, Type)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs<Field>(PXCache cache, Type sourceField) where Field :
 IBqlField
```

Parameters:

- `cache`

  Cache containing the data item.
- `def`

  Default value.

SetViewAs<Field>(PXCache, object, string)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs<Field>(PXCache cache, object data, string source) where
 Field : IBqlField
```

Parameters:

- `cache`

  Cache containing the data item.

- `def`

  Default value.

SetViewAs<Field>(PXCache, object, Type)

Overrides the view as state for the particular data item.

Syntax:

```
public static void SetViewAs<Field>(PXCache cache, object data, Type sourceField)
 where Field : IBqlField
```

Parameters:

- `cache`

  Cache containing the data item.

- `def`

  Default value.

PXRSACryptString Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
      PXDBCryptStringAttribute
```

Syntax

```
public class PXRSACryptStringAttribute : PXDBCryptStringAttribute
```

Constructors

Constructor	Description
PXRSACryptStringAttribute()	
PXRSACryptStringAttribute(int)	

Static Methods

Method	Description
Encrypt(string)	

PXRSACryptString Attribute Constructors

The *PXRSACryptString* attribute exposes the following constructors.

PXRSACryptStringAttribute()

Syntax:

```
public PXRSACryptStringAttribute()
```

PXRSACryptStringAttribute(int)

Syntax:

```
public PXRSACryptStringAttribute(int length) : base(length)
```

PXRSACryptString Attribute Methods

The *PXRSACryptString* attribute exposes the following static methods.

Encrypt(string)

Syntax:

```
public static string Encrypt(string source) : :
```

PXDB3DesCryphString Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
      PXDBCryptStringAttribute
```

Syntax

```
public class PXDB3DesCryphStringAttribute : PXDBCryptStringAttribute
```

Constructors

Constructor	Description
PXDB3DesCryphStringAttribute()	
PXDB3DesCryphStringAttribute(int)	Initializes a new instance with the given maximum length

Static Methods

Method	Description
Encrypt(string)	

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

PXDB3DesCryphString Attribute Constructors

The *PXDB3DesCryphString* attribute exposes the following constructors.

PXDB3DesCryphStringAttribute()

Syntax:

```
public PXDB3DesCryphStringAttribute()
```

PXDB3DesCryphStringAttribute(int)

Initializes a new instance with the given maximum length.

Syntax:

```
public PXDB3DesCryphStringAttribute(int length) : base(length)
```

PXDB3DesCryphString Attribute Methods

The *PXDB3DesCryphString* attribute exposes the following static methods.

Encrypt(string)

Syntax:

```
public static string Encrypt(string source)
```

PXDBText Attribute

Maps a DAC field of `string` type to the database column of `nvarchar` or `varchar` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBTextAttribute : PXDBStringAttribute
```

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBText(IsUnicode = true)]
[PXUIField(DisplayName = "Activity Details")]
public virtual string Body { ... }
```

PXDBTimeSpan Attribute

Maps a DAC field of `int?` type to the `int` database column. The field value represents a date as a number of minutes passed from 01/01/1900.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBIntAttribute
```

Syntax

```
public class PXDBTimeSpanAttribute : PXDBIntAttribute
```

Properties

- `public string` **`InputMask`**

  Gets or sets the input mask for date and time values that can be entered as value of the current field. By default, the proprty equals *HH:mm*.

- `public string` **`DisplayMask`**

  Gets or sets the display mask for date and time values that can be entered as value of the current field. By default, the proprty equals *HH:mm*.

- `public new string` **`MinValue`**

  Gets or sets the minimum value for the field. The value should be a valid string representation of a date.

- `public new string` **`MaxValue`**

  Gets or sets the maximum value for the field. The value should be a valid string representation of a date.

Constructors

- `public PXDBTimeSpanAttribute()`

  Initializes a new instance of the attribute with default parameters.

Static Methods

Method	Description
FromMinutes(int)	Returns the date obtained by adding the specified number of minutes to 01/01/1900

Nested Classes

- `public sealed class zero : Constant<string>`

  Represents the *00:00* string contant in BQL.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

The field value stores a date as a number of minutes. In the UI, the string is typically represented by a control allowing a selection from the list of time values with half-hour interval.

Examples

```
[PXDBTimeSpan]
[PXUIField(DisplayName = "Run Time")]
public virtual int? RunTime { get; set; ]
```

PXDBTimeSpan Attribute Methods

The *PXDBTimeSpan* attribute exposes the following static methods.

FromMinutes(int)

Returns the date obtained by adding the specified number of minutes to 01/01/1900.

Syntax:

```
public static DateTime FromMinutes(int minutes)
```

Examples:

```
DateTime date = PXDBTimeSpanAttribute.FromMinutes(40);
```

PXDBTimeSpanLong Attribute

Maps a DAC field of `int?` type that represents a duration in time as the number of minutes to the `int` database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBIntAttribute
```

Syntax

```
public class PXDBTimeSpanLongAttribute : PXDBIntAttribute
```

Properties

- `public TimeSpanFormatType` **`Format`**

  Gets or sets the data format type. Possible values are defined by the *TimeSpanFormatType* enumeration.

- `public string` **`InputMask`**

  Gets or sets the pattern that indicates the allowed characters in a field value. By default, the property is null, and the attribute determines the input mask by the `Format` value.

Constructors

- `public PXDBTimeSpanLongAttribute()`

  Initializes a new instance of the attribute.

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBTimeSpanLong(Format = TimeSpanFormatType.LongHoursMinutes)]
[PXUIField(DisplayName = "Estimation")]
public virtual Int32? TimeEstimated { get; set; }
```

TimeSpanFormatType Enumeration

Defines data format types for the *PXDBTimeSpanLongAttribute* and *PXTimeSpanLongAttribute* attributes.

Members

- `DaysHoursMinites = 0`
- `DaysHoursMinitesCompact`
- `LongHoursMinutes`
- `ShortHoursMinutes`
- `ShortHoursMinutesCompact`

PXDBTimestamp Attribute

Maps a DAC field of `byte[]` type to the database column of `timestamp` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXRowPersistedSubscriber`
- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBTimestampAttribute : PXDBFieldAttribute,
                                      IPXRowSelectingSubscriber,
                                      IPXCommandPreparingSubscriber,
                                      IPXRowPersistedSubscriber,
                                      IPXFieldUpdatingSubscriber,
                                      IPXFieldSelectingSubscriber
```

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

The attribute binds the field to a timestamp column in the database. The database timestamp is a counter that is incremented for each insert or update operation performed on a table with a `timestamp` column. The counter tracks a relative time within a database (not an actual time that can be associated with a clock). You can use the `timestamp` column of a data record to easily determine whether any value in the data record has changed since the last time it was read.

Examples

```
[PXDBTimestamp()]
```

```
public virtual byte[] tstamp { get; set; }
```

PXDBBinary Attribute

Maps a DAC field of `byte[]` type to the binary database column of either fixed or variable length.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBBinaryAttribute : PXDBFieldAttribute
```

Properties

- `public bool` **`IsFixed`**

  Gets or sets an indication that the binay value has a fixed length. This property should be set to `true` if the database column has a fixed length type (`binary`) and to `false` if the database column has a variable length type (`varbinary`). The default value is `false`.

- `public int` **`Length`**

  Gets the maximum length of the binary value.

  The default value is -1 (the length is not limited). A different value can be set in the constructor.

Constructors

Constructor	Description
PXDBBinaryAttribute()	Initializes a new instance of the attribute
PXDBBinaryAttribute(int)	Initializes a new instance with the given maximum length

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBBinary]
[PXUIField(Visible = false)]
public virtual byte[] NewValue { get; set; }
```

PXDBBinary Attribute Constructors

The *PXDBBinary* attribute exposes the following constructors.

PXDBBinaryAttribute()

Initializes a new instance of the attribute.

Syntax:

```
public PXDBBinaryAttribute()
```

PXDBBinaryAttribute(int)

Initializes a new instance with the given maximum length.

Syntax:

```
public PXDBBinaryAttribute(int length)
```

PXDBVariant Attribute

Maps a DAC field of `byte[]` type to the database column of a variant type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBBinaryAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBVariantAttribute : PXDBBinaryAttribute,
                                    IPXFieldUpdatingSubscriber,
                                    IPXFieldSelectingSubscriber
```

Constructors

Constructor	Description
PXDBVariantAttribute()	Initializes a new instance of the attribute
PXDBVariantAttribute(int)	Initializes a new instance with the given maximum length

Static Methods

Method	Description
GetValue(byte[])	
SetValue(object)	

Remarks

The attribute is added to the value declaration of a DAC field. The field becomes bound to the database column with the same name.

Examples

```
[PXDBVariant]
[PXUIField(DisplayName = "Value")]
public virtual byte[] Value { get; set; }
```

PXDBVariant Attribute Constructors

The *PXDBVariant* attribute exposes the following constructors.

PXDBVariantAttribute()

Initializes a new instance of the attribute.

Syntax:

```
public PXDBVariantAttribute() : base()
```

PXDBVariantAttribute(int)

Initializes a new instance with the given maximum length.

Syntax:

```
public PXDBVariantAttribute(int length) : base(length)
```

PXDBVariant Attribute Methods

The *PXDBVariant* attribute exposes the following static methods.

GetValue(byte[])

Syntax:

```
public static object GetValue(byte[] val)
```

SetValue(object)

Syntax:

```
public static byte[] SetValue(object value)
```

Unbound Field Data Types

The following attributes define a data access class field of a specific type that are not bound to any database columns.

Attribute	C# data type	Comment
PXBool	`bool?`	Boolean value
PXByte	`byte?`	1-byte integer value
PXDate	`DateTime?`	Date and time
PXDateAndTime	`DateTime?`	Date and time values represented by separate input controls in the user interface
PXDecimal	`Decimal?`	16-byte floating point numeric value with a specific precision

Attribute	C# data type	Comment
PXDouble	`double?`	8-byte floating point value
PXFloat	`float?`	4-byte floating point value
PXGuid	`Guid?`	16-byte unique value
PXShort	`short?`	2-byte integer value
PXInt	`int?`	4-byte integer value
PXLong	`int64?`	8-byte integer value
PXString	`string`	String of characters
PXTimeSpan	`int?`	Date and time value represented by minutes passed from 01/01/1900
PXTimeSpanLong	`int?`	Duration in time as the number of minutes
PXVariant	`byte[]`	Arbitrary array of bytes

PXBool Attribute

Indicates a DAC field of `bool?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXBoolAttribute : PXEventSubscriberAttribute,
                               IPXFieldUpdatingSubscriber,
                               IPXFieldSelectingSubscriber,
                               IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXBool()]
[PXDefault(false)]
public virtual bool? Selected { get; set; }
```

PXByte Attribute

Indicates a DAC field of `short?` that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXByteAttribute : PXEventSubscriberAttribute,
                               IPXFieldUpdatingSubscriber,
                               IPXFieldSelectingSubscriber,
                               IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.
- `public int` **`MinValue`**

  Gets or sets the minimum value for the field.
- `public int` **`MaxValue`**

  Gets or sets the maximum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

PXDate Attribute

Indicates a DAC field of `DateTime?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXDateAttribute : PXEventSubscriberAttribute,
                               IPXFieldUpdatingSubscriber,
                               IPXFieldSelectingSubscriber,
                               IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.

- `public string` **`InputMask`**

  Gets or sets the format string that defines how a field value inputted by a user should be formatted. The property takes the same values as `DisplayMask`.

- `public string` **`DisplayMask`**

  Gets or sets the format string that defines how a field value is displayed in the input control. If the property is set to a one-character string, the corresponding *standard date and time format string* is used. If the property value is longer, it is treated as a *custom date and time format string*. A particular pattern depends on the culture set by the application.

- `public string` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public string` **`MaxValue`**

  Gets or sets the maximum value for the field.

- `public bool` **`UseTimeZone`**

  Gets or sets the value that indicates whether the attribute should convert the time to UTC, using the local time zone. If `true`, the time is converted. By default, `true`.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXDate()]
public virtual DateTime? NextEffDate { get; set; }
```

PXDateAndTime Attribute

Indicates a DAC field of `DateTime?` type that is not mapped to a database column and is represented in the user interface by two controls to input date and time values separately.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDateAttribute
```

Syntax

```
public class PXDateAndTimeAttribute : PXDateAttribute
```

Static Methods

Method	Description
SetDateEnabled(PXCache, object, string, bool)	Enables or disables the input control that represents the date part of the field value.
SetDateEnabled<Field>(PXCache, object, bool)	Enables or disables the input control that represents the date part of the field value.
SetTimeEnabled(PXCache, object, string, bool)	Enables or disables the input control that represents the time part of the field value.
SetTimeEnabled<Field>(PXCache, object, bool)	Enables or disables the input control that represents the time part of the field value.

Nested Classes

- `public class now : Constant<DateTime>`

  Represents the local date and time in BQL.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXDateAndTime]
public virtual DateTime? StartDate { get; set; }
```

PXDateAndTime Attribute Methods

The *PXDateAndTime* attribute exposes the following static methods.

SetDateEnabled(PXCache, object, string, bool)

Enables or disables the input control that represents the date part of the field value.

Syntax:

```
public static void SetDateEnabled(PXCache cache, object data,
                                  string name, bool isEnabled)
```

Parameters:

- `cache`

  The cache object to search for `PXDateAndTime` attributes.

- `data`

The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The name of the field the attribute is attached to.

- `isEnabled`

  The value indicating whether the input control is enabled.

SetDateEnabled<Field>(PXCache, object, bool)

Enables or disables the input control that represents the date part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetDateEnabled<Field>(PXCache cache, object data,
                                         bool isEnabled)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for `PXDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isEnabled`

  The value indicating whether the input control is enabled.

SetTimeEnabled(PXCache, object, string, bool)

Enables or disables the input control that represents the time part of the field value.

Syntax:

```
public static void SetTimeEnabled(PXCache cache, object data,
                                  string name, bool isEnabled)
```

Parameters:

- `cache`

  The cache object to search for `PXDateAndTime` attributes.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The name of the field the attribute is attached to.

- `isEnabled`

  The value indicating whether the input control is enabled.

SetTimeEnabled<Field>(PXCache, object, bool)

Enables or disables the input control that represents the time part of the field value. The field is specified as the type parameter.

Syntax:

```
public static void SetTimeEnabled<Field>(PXCache cache, object data,
                                          bool isEnabled)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for `PXDateAndTime` attributes.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `isEnabled`

  The value indicating whether the input control is enabled.

PXDecimal Attribute

Indicates a DAC field of `decimal?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXDecimalAttribute : PXEventSubscriberAttribute,
                                  IPXFieldUpdatingSubscriber,
                                  IPXFieldSelectingSubscriber,
                                  IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.
- `public double` **`MinValue`**

  Gets or sets the minimum value for the field.
- `public double` **`MaxValue`**

Gets or sets the maximum value for the field.

Constructors

Constructor	Description
PXDecimalAttribute()	Initializes a new instance with the default precision, which equals 2
PXDecimalAttribute(int)	Initializes a new instance with the given precision
PXDecimalAttribute(Type)	Initializes a new instance with the precision calculated at runtime using a BQL query

Static Methods

Method	Description
SetPrecision(PXCache, string, int?)	Sets the precision in the attribute intance that marks the field with the specified name in all data records in the cache object
SetPrecision(PXCache, object, string, int?)	Sets the precision in the attribute intance that marks the field with the specified name in a particular data record
SetPrecision<Field>(PXCache, int?)	Sets the precision in the attribute intance that marks the specified field in all data records in the cache object
SetPrecision<Field>(PXCache, object, int?)	Sets the precision in the attribute intance that marks the specified field in a particular data record

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXDecimal(0)]
[PXUIField(DisplayName = "SignBalance")]
public virtual Decimal? SignBalance { get; set; }
```

PXDecimal Attribute Constructors

The *PXDecimal* attribute exposes the following constructors.

PXDecimalAttribute()

Initializes a new instance with the default precision, which equals 2.

Syntax:

```
public PXDecimalAttribute()
```

PXDecimalAttribute(int)

Initializes a new instance with the given precision.

Syntax:

```
public PXDecimalAttribute(int precision)
```

PXDecimalAttribute(Type)

Initializes a new instance with the precision calculated at runtime using a BQL query.

Syntax:

```
public PXDecimalAttribute(Type type)
```

Parameters:

- `type`

  A BQL query based on a class derived from `IBqlSearch` or `IBqlField`. For example, the parameter can be set to `typeof(Search<...>)`, or `typeof(Table.field)`.

PXDecimal Attribute Methods

The *PXDecimal* attribute exposes the following static methods.

SetPrecision(PXCache, string, int?)

Sets the precision in the attribute intance that marks the field with the specified name in all data records in the cache object.

Syntax:

```
public static void SetPrecision(PXCache cache, string name, int? precision)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBDecimal` type.

- `name`

  The name of the field that is be marked with the attribute.

- `precision`

  The new precision value.

SetPrecision(PXCache, object, string, int?)

Sets the precision in the attribute intance that marks the field with the specified name in a particular data record.

Syntax:

```
public static void SetPrecision(PXCache cache, object data, string name, int?
            precision)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDecimal` type.

- `data`

  The data record the method is applied to.

- `name`

The name of the field that is be marked with the attribute.

- `precision`

  The new precision value.

SetPrecision<Field>(PXCache, int?)

Sets the precision in the attribute intance that marks the specified field in all data records in the cache object.

Syntax:

```
public static void SetPrecision<Field>(PXCache cache, int? precision) where Field :
            IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBDecimal` type.

- `precision`

  The new precision value.

SetPrecision<Field>(PXCache, object, int?)

Sets the precision in the attribute intance that marks the specified field in a particular data record.

Syntax:

```
public static void SetPrecision<Field>(PXCache cache, object data, int? precision)
            where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBDecimal` type.

- `data`

  The data record the method is applied to.

- `precision`

  The new precision value.

PXDouble Attribute

Indicates a DAC field of `double?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXDoubleAttribute : PXEventSubscriberAttribute,
                                 IPXFieldUpdatingSubscriber,
                                 IPXFieldSelectingSubscriber,
                                 IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.
- `public double` **`MinValue`**

  Gets or sets the minimum value for the field.
- `public double` **`MaxValue`**

  Gets or sets the maximum value for the field.

Constructors

Constructor	Description
PXDoubleAttribute()	Initializes a new instance of the attribute with default parameters
PXDoubleAttribute(int)	Initializes a new instance of the attribute with the given precision

Static Methods

Method	Description
SetPrecision(PXCache, string, int)	
SetPrecision(PXCache, object, string, int)	
SetPrecision<Field>(PXCache, int)	
SetPrecision<Field>(PXCache, object, int)	

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXDouble]
[PXUIField(Visible = false)]
public virtual Double? OriginalShift { get; set; }
```

PXDouble Attribute Constructors

The *PXDouble* attribute exposes the following constructors.

PXDoubleAttribute()

Initializes a new instance of the attribute with default parameters.

Syntax:

```
public PXDoubleAttribute()
```

PXDoubleAttribute(int)

Initializes a new instance of the attribute with the given precision. The precision is the number of digits after the comma. If a user enters a value with greater number of fractional digits, the value will be rounded.

Syntax:

```
public PXDoubleAttribute(int precision)
```

Parameters:

- `precision`

  The value to use as the precision.

PXDouble Attribute Methods

The *PXDouble* attribute exposes the following static methods.

SetPrecision(PXCache, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, string name, int precision)
```

SetPrecision(PXCache, object, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, object data, string name, int
 precision)
```

SetPrecision<Field>(PXCache, int)

Syntax:

```
public static void SetPrecision<Field>(PXCache cache, int precision) where Field :
 IBqlField
```

SetPrecision<Field>(PXCache, object, int)

Syntax:

```
public static void SetPrecision<Field>(PXCache cache, object data, int precision)
 where Field : IBqlField
```

PXFloat Attribute

Indicates a DAC field of `float?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXFloatAttribute : PXEventSubscriberAttribute,
                                IPXFieldUpdatingSubscriber,
                                IPXFieldSelectingSubscriber,
                                IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.
- `public float` **`MinValue`**

  Gets or sets the minimum value for the field.
- `public float` **`MaxValue`**

  Gets or sets the maximum value for the field.

Constructors

Constructor	Description
PXFloatAttribute()	Initializes a new instance of the attribute with default parameters
PXFloatAttribute(int)	Initializes a new instance of the attribute with the given precision

Static Methods

Method	Description
SetPrecision(PXCache, string, int)	
SetPrecision(PXCache, object, string, int)	
SetPrecision<Field>(PXCache, int)	
SetPrecision<Field>(PXCache, object, int)	

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

PXFloat Attribute Constructors

The *PXFloat* attribute exposes the following constructors.

PXFloatAttribute()

Initializes a new instance of the attribute with default parameters.

Syntax:

```
public PXFloatAttribute()
```

PXFloatAttribute(int)

Initializes a new instance of the attribute with the given precision. The precision is the number of digits after the comma. If a user enters a value with greater number of fractional digits, the value will be rounded.

Syntax:

```
public PXFloatAttribute(int precision)
```

Parameters:

- `precision`

  The value to use as the precision.

PXFloat Attribute Methods

The *PXFloat* attribute exposes the following static methods.

SetPrecision(PXCache, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, string name, int
            precision)
```

SetPrecision(PXCache, object, string, int)

Syntax:

```
public static void SetPrecision(PXCache cache, object data, string name, int
            precision)
```

SetPrecision<Field>(PXCache, int)

Syntax:

```
public static void SetPrecision<Field>(PXCache cache, int precision) where
            Field : IBqlField
```

SetPrecision<Field>(PXCache, object, int)

Syntax:

```
public static void SetPrecision<Field>(PXCache cache, object data, int
            precision) where Field : IBqlField
```

PXGuid Attribute

Indicates a DAC field of `Guid?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXGuidAttribute : PXEventSubscriberAttribute,
                               IPXFieldUpdatingSubscriber,
                               IPXFieldSelectingSubscriber,
                               IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXGuid]
[PXSelector(typeof(EPEmployee.userID),
            SubstituteKey = typeof(EPEmployee.acctCD),
            DescriptionField = typeof(EPEmployee.acctName))]
[PXUIField(DisplayName = "Custodian")]
public virtual Guid? Custodian { get; set; }
```

PXImage Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXStringAttribute
```

Syntax

```
public class PXImageAttribute : PXStringAttribute
```

Properties

- `public string` **HeaderImage**

  Get, set.

PXInt Attribute

Indicates a DAC field of `int?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXIntAttribute : PXEventSubscriberAttribute,
                              IPXFieldUpdatingSubscriber,
                              IPXFieldSelectingSubscriber,
                              IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **IsKey**

  Gets or sets the value that indicates whether the field is a key field.

- `public int` **MinValue**

  Gets or sets the minimum value for the field.

- `public int` **MaxValue**

  Gets or sets the maximum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXInt()]
[PXUIField(DisplayName = "Documents", Visible = true)]
public virtual int? DocCount { get; set; }
```

PXLong Attribute

Indicates a DAC field of `long?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXLongAttribute : PXEventSubscriberAttribute,
                               IPXFieldUpdatingSubscriber,
                               IPXFieldSelectingSubscriber,
                               IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.

- `public Int64` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public Int64` **`MaxValue`**

  Gets or sets the maximum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXLong(IsKey = true)]
[PXUIField(DisplayName = "Transaction Num.")]
public virtual Int64? TranID { get; set; }
```

PXShort Attribute

Indicates a DAC field of `short?` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXShortAttribute : PXEventSubscriberAttribute,
                                IPXFieldUpdatingSubscriber,
                                IPXFieldSelectingSubscriber,
                                IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.
- `public int` **`MinValue`**

  Gets or sets the minimum value for the field.
- `public int` **`MaxValue`**

  Gets or sets the maximum value for the field.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXShort()]
[PXDefault((short)0)]
[PXUIField(DisplayName = "Overdue Days", Enabled = false)]
public virtual short? OverdueDays { get; set; }
```

PXString Attribute

Indicates a DAC field of `string` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXFieldState))]
public class PXStringAttribute : PXEventSubscriberAttribute,
```

```
                                                   IPXFieldUpdatingSubscriber,
                                                   IPXFieldSelectingSubscriber,
                                                   IPXCommandPreparingSubscriber
```

Properties

- `public virtual bool` **`IsKey`**

  Gets or sets the value that indicates whether the field is a key field.

- `public int` **`Length`**

  Gets the maximum length of the string value. If a string value exceeds the maximum length, it will be trimmed. If `IsFixed` is set to `true` and the string length is less then the maximum, it will be extended with spaces. By default, the property is `-1`, which means that the string length is not limited.

- `public string` **`InputMask`**

  Gets or sets the pattern that indicates the allowed characters in a field value. The user interface will not allow the user to enter other characters in the input control associated with the field.

  The default value for the key fields is '`>aaaaaa`'.

  Control characters:

  - '`>`': the following chars to upper case
  - '`<`': the following chars to lower case
  - '`&`', '`C`': any character or a space
  - '`A`', '`a`': a letter or digit
  - '`L`', '`?`': a letter
  - '`#`', '`0`', '`9`': a digit

  Examples:

  ```
  InputMask = ">LLLLL"
  ```

  ```
  InputMask = ">aaaaaaaaaa"
  ```

  ```
  InputMask = ">CC.00.00.00"
  ```

- `public bool` **`IsFixed`**

  Gets or sets an indication that the string has a fixed length. This property should be set to `true` if the database column has a fixed length type (`char` or `nchar`). The default value is `false`.

- `public bool` **`IsUnicode`**

  Gets or sets an indication that the string consists of Unicode characters. This property should be set to `true` if the database column has a Unicode string type (`nchar` or `nvarchar`). The default value is `false`.

Constructors

Constructor	Description
PXStringAttribute()	Initializes a new instance with default parameters
PXStringAttribute(int)	Initializes a new instance with the given maximum length of a field value

Static Methods

Method	Description
SetInputMask(PXCache, string, string)	Sets the input mask for the string field with the specified name for all data records in the cache object
SetInputMask(PXCache, object, string, string)	Sets the input mask for the string field with the specified name
SetInputMask<Field>(PXCache, string)	Sets the input mask for the specified string field for all data records in the cache object
SetInputMask<Field>(PXCache, object, string)	Sets the input mask for the specified string field
SetLength(PXCache, string, int)	Sets the maximum length for the string field with the specified name for all data records in the cache object
SetLength(PXCache, object, string, int)	Sets the maximum length for the string field with the specified name
SetLength<Field>(PXCache, int)	Sets the maximum length for the specified string field for all data records in the cache object
SetLength<Field>(PXCache, object, int)	Sets the maximum length for the specified string field

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

It is possible to specify the maximum length and input validation mask for the string.

You can modify the `Length` and `InputMask` properties at run time by calling the static methods.

Examples

The attribute below defines an unbound field taking as a value Unicode strings of 5 uppercase characters that are strictly aphabetical letters.

```
[PXString(5, IsUnicode = true, InputMask = ">LLLLL")]
public virtual String FinChargeCuryID { get; set; }
```

PXString Attribute Constructors

The *PXString* attribute exposes the following constructors.

PXStringAttribute()

Initializes a new instance with default parameters.

Syntax:

```
public PXStringAttribute()
```

PXStringAttribute(int)

Initializes a new instance with the given maximum length of a field value.

Syntax:

```
public PXStringAttribute(int length)
```

Parameters:

- `length`

  The maximum length value assigned to the `Length` property.

PXString Attribute Methods

The *PXString* attribute exposes the following static methods.

SetInputMask(PXCache, string, string)

Sets the input mask for the string field with the specified name for all data records in the cache object.

Syntax:

```
public static void SetInputMask(PXCache cache, string name, string mask)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXString` type.

- `name`

  The field name.

- `mask`

  The value that is assigned to the `InputMask` property.

SetInputMask(PXCache, object, string, string)

Sets the input mask for the string field with the specified name.

Syntax:

```
public static void SetInputMask(PXCache cache, object data, string name, string
            mask)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXString` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

- `mask`

  The value that is assigned to the `InputMask` property.

SetInputMask<Field>(PXCache, string)

Sets the input mask for the specified string field for all data records in the cache object.

Syntax:

```
public static void SetInputMask<Field>(PXCache cache, string mask) where Field :
            IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXString` type.

- `mask`

  The value that is assigned to the `InputMask` property.

SetInputMask<Field>(PXCache, object, string)

Sets the input mask for the specified string field.

Syntax:

```
public static void SetInputMask<Field>(PXCache cache, object data, string mask)
 where
           Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXString` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `mask`

  The value that is assigned to the `InputMask` property.

SetLength(PXCache, string, int)

Sets the maximum length for the string field with the specified name for all data records in the cache object.

Syntax:

```
public static void SetLength(PXCache cache, string name, int length)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXString` type.

- `name`

  The field name.

- `length`

  The value that is assigned to the `Length` property.

SetLength(PXCache, object, string, int)

Sets the maximum length for the string field with the specified name.

Syntax:

```
public static void SetLength(PXCache cache, object data, string name, int
           length)
```

Parameters:

- `cache`

The cache object to search for the attributes of `PXString` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

- `length`

  The value that is assigned to the `Length` property.

SetLength<Field>(PXCache, int)

Sets the maximum length for the specified string field for all data records in the cache object.

Syntax:

```
public static void SetLength<Field>(PXCache cache, int length) where Field :
            IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXString` type.

- `length`

  The value that is assigned to the `Length` property.

SetLength<Field>(PXCache, object, int)

Sets the maximum length for the specified string field.

Syntax:

```
public static void SetLength<Field>(PXCache cache, object data, int length) where
            Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXString` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `length`

  The value that is assigned to the `Length` property.

PXTimeSpan Attribute

Indicates a DAC field of `int?` type that represents a date value as minutes passed from 01/01/1900 and that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntAttribute
```

Syntax

```
public class PXTimeSpanAttribute : PXIntAttribute
```

Properties

- `public string` **`InputMask`**

  Gets or sets the pattern that indicates the allowed characters in a field value. The user interface will not allow the user to enter other characters in the input control associated with the field.

- `public string` **`DisplayMask`**

  Get, set.

- `public new string` **`MinValue`**

  Gets or sets the minimum value for the field.

- `public new string` **`MaxValue`**

  Gets or sets the maximum value for the field.

Constructors

- `public PXTimeSpanAttribute()`

  Initializes a new instance with default parameters.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

PXTimeSpanLong Attribute

Indicates a DAC field of `int?` type that represents a duration in time as the number of minutes and that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntAttribute
```

Syntax

```
public class PXTimeSpanLongAttribute : PXIntAttribute
```

Properties

- `public TimeSpanFormatType` **`Format`**

  Gets or sets the data format type. Possible values are defined by the *TimeSpanFormatType* enumeration.

- `public string` **`InputMask`**

  Gets or sets the pattern that indicates the allowed characters in a field value. By default, the property is null, and the attribute determines the input mask by the `Format` value.

Constructors

- `public PXTimeSpanLongAttribute()`

Initializes a new instance of the attribute.

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

Examples

```
[PXTimeSpanLong(Format = TimeSpanFormatType.LongHoursMinutes)]
public virtual int? InitResponse { get; set; }
```

PXVariant Attribute

Indicates a DAC field of `byte[]` type that is not mapped to a database column.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXVariantAttribute : PXEventSubscriberAttribute,
                                  IPXFieldUpdatingSubscriber,
                                  IPXFieldSelectingSubscriber
```

Constructors

- `public PXVariantAttribute() : base() { }`

  Initializes a new instance with default parameters.

Static Methods

Method	Description
GetValue(byte[])	

Remarks

The attribute is added to the value declaration of a DAC field. The field is not bound to a table column.

PXVariant Attribute Methods

The *PXVariant* attribute exposes the following static methods.

GetValue(byte[])

Syntax:

```
public static object GetValue(byte[] val)
```

UI Field Configuration

To configure the user interface layout of input controls and buttons, you should use

- *PXUIField*

  Configures the properties of the input control representing a DAC field in the user interface, or the button representing an action.

The attribute is mandatory for all DAC fields displayed in the user interface. You should add the attribute to the field value declaration in the DAC, for example:

```
[PXDBDate()]
[PXUIField(DisplayName = "Pay Date")]
public virtual DateTime? PayDate { get; set; }
```

PXUIField Attribute

Configures the properties of the input control representing a DAC field in the user interface, or the button representing an action. The attribute is mandatory for all DAC fields that are displayed in the user interface.

See *Remarks* for more details. See *Examples* for examples of usage.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXInterfaceField`
- `IPXExceptionHandlingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldSelectingSubscriber`
- `IPXFieldVerifyingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Method |
                AttributeTargets.Class)]
[PXAttributeFamily(typeof(PXUIFieldAttribute))]
public class PXUIFieldAttribute : PXEventSubscriberAttribute,
                                  IPXInterfaceField,
                                  IPXExceptionHandlingSubscriber,
                                  IPXCommandPreparingSubscriber,
                                  IPXFieldSelectingSubscriber,
                                  IPXFieldVerifyingSubscriber
```

Properties

- `public virtual bool` **`Required`**

Gets or sets the value that indicates whether an asterisk sign is shown beside the field in the user interface. Note that this property *does not* check that the field value is specified and add any restriction of this kind. This is done by the *PXDefault* attribute.

The default value is `false`.

- `public virtual bool` **`Visible`**

  Get, set. Allows to show/hide field edit control or grid column in user interface. To control, whether form designer should generate template for this field, use Visibility property instead.

  The default value is `true`.

- `public virtual PXErrorHandling` **`ErrorHandling`**

  Gets or sets the *PXErrorHandling* value that specifies the way the attribute treats an error related to the field. The error is either indicated only when the field is visible, always indicated, or never indicated.

  The default value is `PXErrorHandling.WhenVisible`.

- `public virtual bool` **`Enabled`**

  Gets or sets the value that indicates whether the field input control is enabled. If the field is disabled, the control does not allow the user to edit and select the field value. Compare to the `IsReadOnly` property.

  The default value is `true`.

- `public virtual bool` **`IsReadOnly`**

  Gets or sets the value that indicates whether the field input control allows editing. If the property is set to `true`, the user cannot edit the value, but can still select and copy the value. Compare to the `Enabled` property.

  The default value is `false`.

- `public virtual string` **`DisplayName`**

  Gets or sets the field name displayed in the user interface. This name is rendered as the input control label on a form or as the grid column header.

  The default value is the field name.

- `public virtual PXUIVisibility` **`Visibility`**

  Gets or sets the *PXUIVisibility* value that indicates whether the webpage layout designer should generate a template for this field. You can specify whether the template is generated for a form and grid, is generated for a form, grid, and lookup controls, or never appear in the user interface. The default value is `PXUIVisibility.Visible`.

- `public virtual int` **`TabOrder`**

  Gets or sets the order in which the field input control gets the focus when the user moves it by pressing the TAB key.

- `public virtual PXCacheRights` **`MapViewRights`**

  Gets or sets the *PXCacheRights* value that specifies the access on a cache for a cache to see the button in the user interface. The property is used when the `PXUIField` configures an action button.

- `public virtual PXCacheRights` **`MapEnableRights`**

  Gets or sets the *PXCacheRights* that specifies the access rights on a cache to click the button in the user interface. The property is used when the `PXUIField` configures an action button.

- `public virtual string` **`FieldClass`**

Gets or sets the value that indicates whether the field is shown or hidded depending on the features enabled or disabled. By default, the property is set to the segmented field name.

Constructors

- `public PXUIFieldAttribute()`

  Initializes a new instance of the attribute.

Static Methods

Method	Description
GetDisplayName(PXCache, string)	Returns the value of the `DisplayName` property for the field with the specified name
GetDisplayName<Field>(PXCache)	Returns the value of the `DisplayName` property for the specified field
GetError(PXCache, object, string)	Returns the error string displayed for the field with the specified name
GetError<Field>(PXCache, object)	Returns the error string displayed for the specified field
GetErrors(PXCache, object)	Finds all fields with non-empty error strings and returns a dictionary with field names as the keys and error messages as the values
GetItemName(PXCache)	Returns the user-friendly name of the specified cache object
SetDisplayName(PXCache, string, string)	Sets the display name of the field with the specified name
SetDisplayName<Field>(PXCache, string)	Sets the display name of the specified field
SetEnabled(PXCache, object, bool)	Enables or disables the input controls for all fields in the specific data record or all data records by setting the `Enabled` property
SetEnabled(PXCache, object, string)	Enables the input control for the field with the specified name by setting the `Enabled` property to `true`
SetEnabled(PXCache, string, bool)	Enables or disables the input control for the field with the specified name by setting the `Enabled` property
SetEnabled(PXCache, object, string, bool)	Enables or disables the input control for the field with the specified name by setting the `Enabled` property
SetEnabled<Field>(PXCache, object)	Enables the specified field of the specific data record in the cache object by setting the `Enabled` property to `true`
SetEnabled<Field>(PXCache, object, bool)	Enables or disables the input control for the specified field by setting the `Enabled` property
SetError(PXCache, object, string, string)	Sets the error string to display as a tooltip for the field with the specified name
SetError(PXCache, object, string, string, string)	Sets the error string to display as a tooltip and the error value to display in the input control for the field with the specified name

Method	Description
SetError<Field>(PXCache, object, string)	Sets the error string to display as a tooltip for the specified field
SetError<Field>(PXCache, object, string,)	Sets the error string to display as a tooltip and the error value to display in the input control for the specified field
SetReadOnly(PXCache, object)	Makes the input controls for all fields read-only by setting the `IsReadOnly` property to `true`
SetReadOnly(PXCache, object, string)	Makes the input control for the field with the specified name read-only by setting the `IsReadOnly` property to `true`
SetReadOnly(PXCache, object, bool)	Makes the input controls for all field read-only or not read-only by setting the `IsReadOnly` property
SetReadOnly(PXCache, object, string, bool)	Makes the input control for the field with the specified name read-only or not-read-only by setting the `IsReadOnly` property
SetReadOnly<Field>(PXCache, object)	Makes the input control for the specified field read-only by setting the `IsReadOnly` property to `true`
SetReadOnly<Field>(PXCache, object, bool)	Makes the input control for the specified field read-only or not-read-only by setting the `IsReadOnly` property
SetRequired(PXCache, string, bool)	Sets the `Required` property for the field with the specified name for all data records in the cache object
SetRequired<Field>(PXCache, bool)	Sets the `Required` property for the specified field for all data records in the cache object
SetVisibility(PXCache, string, PXUIVisibility)	Sets the visibility status of the input control for the field with the specified name by setting the `Visibility` property
SetVisibility(PXCache, object, string,)	Sets the visibility status of the input control for the field with the specified name by setting the `Visibility` property
SetVisibility<Field>(PXCache, object,)	Sets the visibility status of the input control for the specified field by setting the `Visibility` property
SetVisible(PXCache, object, string)	Makes the input control for the field with the specified name visible in the user interface by setting the `Visible` property to `true`
SetVisible(PXCache, string, bool)	Shows or hides the input control for the field with the specified name in the user interface for all data record by setting the `Visible` property
SetVisible(PXCache, object, string, bool)	Shows or hides the input control for the field with the specified name in the user interface by setting the `Visible` property
SetVisible<Field>(PXCache, object)	Makes the input control for the specified field visible in the user interface by setting the `Visible` property to `true`

Method	Description
SetVisible<Field>(PXCache, object, bool)	Shows or hides the input control for the specified field in the user interface by setting the `Visible` property
SetWarning(PXCache, object, string, string)	Sets the error string to display as a tooltip for the field with the specified name
SetWarning<Field>(PXCache, object, string)	Sets the error string to display as a tooltip for the specified field

Remarks

The attribute is added:

- To a DAC field declaration to configure the field input control
- To the declaration of the method that implements an action to configure the action button

The attribute's properties configure the control layout in the user interface. You can set the display name, specify whether the control is visible or hidden, enable or disable the control, set the error marker, and specify access rights to view and use the control.

You can use the static methods to set the properties at run time. The `PXUIFieldAttribute` static methods can be called either in the business logic container constructor or the `RowSelected` event handlers.

The `RowSelected` event handler is raised when the user interface controls are prepared to be displayed. This happens each type the webpage sends a request to the server.

For input controls enclosed in a form, the properties can be set in any `RowSelected` event handler. For a grid column, the `Visible` and `Required` properties should be set only in the `RowSelected` event handler corresponding to the primary view DAC. For example, on a master-detail webpage, the detail grid column layout shoudl be configured in the `RowSelected` event handler of the master DAC type.

Also, if the grid column layout is configured at runtime, the `data` parameter should be set to `null`. This will indicate that the property should be set for all data records shown in the grid. If a specific data record is passed to the method rather than null, the method invocation will have no effect.

Examples

Configuring the input control for a DAC field:

```
[PXDBDecimal(2)]
[PXUIField(DisplayName = "Documents Total",
           Visibility = PXUIVisibility.SelectorVisible,
           Enabled = false)]
public virtual decimal? CuryDocsTotal { get; set; }
```

Changing the layout configuration properties at runtime:

```
protected virtual void APInvoice_RowSelected(PXCache cache,
                                             PXRowSelectedEventArgs)
{
    APInvoice doc = e.Row as APInvoice;

    // Disable the field input control
    PXUIFieldAttribute.SetEnabled<APInvoice.taxZoneID>(
        cache, doc, false);

    // Showing or hiding a 'required' mark beside a field input control
    PXUIFieldAttribute.SetRequired<APInvoice.dueDate>(
        cache, (doc.DocType != APDocType.DebitAdj));
```

```
    // Making a field visible.
    // The data parameter is set to null to set the property for all
    // APTran data records.
    PXUIFieldAttribute.SetVisible<APTran.projectID>(
        Transactions.Cache, null, true);
}
```

Note in the `SetEnabled` method, the first parameter is set to the `cache` variable. This is a `PXCache` object keeping `APInvoice` data records. The second parameter is set to such a data record obtained from `e.Row`.

On the other hand, the `SetVisible` method is called for the `APTran` DAC field, and therefore a different cache object should be passed to the method. The appropriate cache is specified using the `Cache` property of the `Transactions` view, which can be defined as something like this:

```
public PXSelect<APTran,
    Where<APTran.tranType, Equal<Current<APInvoice.docType>>,
        And<APTran.refNbr, Equal<Current<APInvoice.refNbr>>>>>
```

Configuring the action button:

```
// The action declaration
public PXAction<APDocumentFilter> viewDocument;
// The action method declaration
[PXUIField(DisplayName = "View Document",
           MapEnableRights = PXCacheRights.Select,
           MapViewRights = PXCacheRights.Select)]
[PXButton]
public virtual IEnumerable ViewDocument(PXAdapter adapter)
{
    ...
}
```

Related Types

- *PXUIVisibility Enumeration*
- *PXErrorHandling Enumeration*
- *PXErrorLevel Enumeration*
- *PXCacheRights Enumeration*

PXUIField Attribute Methods

The *PXUIField* attribute exposes the following static methods.

GetDisplayName(PXCache, string)

Returns the value of the `DisplayName` property for the field with the specified name.

Syntax:

```
public static string GetDisplayName(PXCache cache, string name)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `name`

  The field name.

GetDisplayName<Field>(PXCache)

Returns the value of the `DisplayName` property for the specified field.

Syntax:

```
public static string GetDisplayName<Field>(PXCache cache)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

GetError(PXCache, object, string)

Returns the error string displayed for the field with the specified name.

Syntax:

```
public static string GetError(PXCache cache, object data, string name)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

GetError<Field>(PXCache, object)

Returns the error string displayed for the specified field.

Syntax:

```
public static string GetError<Field>(PXCache cache, object data)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

GetErrors(PXCache, object)

Finds all fields with non-empty error strings and returns a dictionary with field names as the keys and error messages as the values.

Syntax:

```
public static Dictionary<string, string> GetErrors(PXCache cache, object data)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record whose fields are checked for error strings. If `null`, the method takes into account all data records in the cache object.

GetItemName(PXCache)

Returns the user-friendly name of the specified cache object. The name is set using the *PXCacheName* attribute.

Syntax:

```
public static string GetItemName(PXCache sender)
```

Parameters:

- `cache`

  The cache object the method is applied to.

SetDisplayName(PXCache, string, string)

Sets the display name of the field with the specified name.

Syntax:

```
public static void SetDisplayName(PXCache cache, string name, string displayName)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `name`

  The field name.

- `displayName`

  The new display name.

SetDisplayName<Field>(PXCache, string)

Sets the display name of the specified field.

Syntax:

```
public static void SetDisplayName<Field>(PXCache cache, string displayName)
    where  Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `displayName`

  The new display name.

SetEnabled(PXCache, object, bool)

Enables or disables the input controls for all fields in the specific data record or all data records by setting the `Enabled` property.

Syntax:

```
public static void SetEnabled(PXCache cache, object data, bool isEnabled)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isEnabled`

  The value that is assigned to the `Enabled` property.

SetEnabled(PXCache, object, string)

Enables the input control for the field with the specified name by setting the `Enabled` property to `true`.

Syntax:

```
public static void SetEnabled(PXCache cache, object data, string name)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

SetEnabled(PXCache, string, bool)

Enables or disables the input control for the field with the specified name by setting the `Enabled` property.

Syntax:

```
public static void SetEnabled(PXCache cache, string name, bool isEnabled)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `name`

  The field name.

- `isEnabled`

  The value that is assigned to the `Enabled` property.

SetEnabled(PXCache, object, string, bool)

Enables or disables the input control for the field with the specified name by setting the `Enabled` property.

Syntax:

```
public static void SetEnabled(PXCache cache, object data,
                              string name, bool isEnabled)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `name`

  The field name.
- `isEnabled`

  The value that is assigned to the `Enabled` property.

SetEnabled<Field>(PXCache, object)

Enables the specified field of the specific data record in the cache object by setting the `Enabled` property to `true`.

Syntax:

```
public static void SetEnabled<Field>(PXCache cache, object data)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

SetEnabled<Field>(PXCache, object, bool)

Enables or disables the input control for the specified field by setting the `Enabled` property.

Syntax:

```
public static void SetEnabled<Field>(PXCache cache, object data,
                                     bool isEnabled)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `data`

The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isEnabled`

  The value that is assigned to the `Enabled` property.

SetError(PXCache, object, string, string)

Sets the error string to display as a tooltip for the field with the specified name.

Syntax:

```
public static void SetError(PXCache cache, object data,
                            string name, string error)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `name`

  The field name.
- `error`

  The string that is set as the error message string.

SetError(PXCache, object, string, string, string)

Sets the error string to display as a tooltip and the error value to display in the input control for the field with the specified name.

Syntax:

```
public static void SetError(PXCache cache, object data, string name,
                            string error, string errorValue)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.
- `name`

  The field name.
- `error`

  The error string displayed as a tooltip on the field input control.
- `errorValue`

  The string displayed in the field input control (is not assigned to the field).

SetError<Field>(PXCache, object, string)

Sets the error string to display as a tooltip for the specified field.

Syntax:

```
public static void SetError<Field>(PXCache cache, object data, string error)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `error`

  The error string displayed as a tooltip on the field input control.

SetError<Field>(PXCache, object, string, string)

Sets the error string to display as a tooltip and the error value to display in the input control for the specified field. The error level is set to `PXErrorLevel.Error`.

Syntax:

```
public static void SetError<Field>(PXCache cache, object data,
                                    string error, string errorValue)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

- `error`

  The error string displayed as a tooltip on the field input control.

- `errorValue`

  The string displayed in the field input control (is not assigned to the field).

SetReadOnly(PXCache, object)

Makes the input controls for all fields read-only by setting the `IsReadOnly` property to `true`.

Syntax:

```
public static void SetReadOnly(PXCache cache, object data)
```

Parameters:

- `cache`

The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

SetReadOnly(PXCache, object, string)

Makes the input control for the field with the specified name read-only by setting the `IsReadOnly` property to `true`.

Syntax:

```
public static void SetReadOnly(PXCache cache, object data, string name)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

SetReadOnly(PXCache, object, bool)

Makes the input controls for all field read-only or not read-only by setting the `IsReadOnly` property.

Syntax:

```
public static void SetReadOnly(PXCache cache, object data, bool isReadOnly)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isReadOnly`

  The value that is assigned to the `IsReadOnly` property.

SetReadOnly(PXCache, object, string, bool)

Makes the input control for the field with the specified name read-only or not-read-only by setting the `IsReadOnly` property.

Syntax:

```
public static void SetReadOnly(PXCache cache, object data,
                               string name, bool isReadOnly)
```

Parameters:

- `cache`

The cache object to search for the attributes of PXUIField type.

- data

  The data record the method is applied to. If null, the method is applied to all data records in the cache object.

- name

  The field name.

- isReadOnly

  The value that is assigned to the IsReadOnly property.

SetReadOnly<Field>(PXCache, object)

Makes the input control for the specified field read-only by setting the IsReadOnly property to true.

Syntax:

```
public static void SetReadOnly<Field>(PXCache cache, object data)
    where Field : IBqlField
```

Parameters:

- cache

  The cache object to search for the attributes of PXUIField type.

- data

  The data record the method is applied to. If null, the method is applied to all data records in the cache object.

SetReadOnly<Field>(PXCache, object, bool)

Makes the input control for the specified field read-only or not-read-only by setting the IsReadOnly property.

Syntax:

```
public static void SetReadOnly<Field>(PXCache cache, object data,
                                       bool isReadOnly)
    where Field : IBqlField
```

Parameters:

- cache

  The cache object to search for the attributes of PXUIField type.

- data

  The data record the method is applied to. If null, the method is applied to all data records in the cache object.

- isReadOnly

  The value that is assigned to the IsReadOnly property.

SetRequired(PXCache, string, bool)

Sets the Required property for the field with the specified name for all data records in the cache object.

Syntax:

```
public static void SetRequired(PXCache cache, string name, bool required)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `name`

  The field name.
- `required`

  The value assigned to the `Required` property.

SetRequired<Field>(PXCache, bool)

Sets the `Required` property for the specified field for all data records in the cache object.

Syntax:

```
public static void SetRequired<Field>(PXCache cache, bool required)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `required`

  The value assigned to the `Required` property.

SetVisibility(PXCache, string, PXUIVisibility)

Sets the visibility status of the input control for the field with the specified name by setting the `Visibility` property.

Syntax:

```
public static void SetVisibility(PXCache cache, string name,
                                 PXUIVisibility visibility)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.
- `name`

  The field name.
- `visibility`

  The value that is assigned to the `Enabled` property.

SetVisibility(PXCache, object, string, PXUIVisibility)

Sets the visibility status of the input control for the field with the specified name by setting the `Visibility` property.

Syntax:

```
public static void SetVisibility(PXCache cache, object data,
                                 string name, PXUIVisibility visibility)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

- `visibility`

  The value that is assigned to the `Visibility` property.

SetVisibility<Field>(PXCache, object, PXUIVisibility)

Sets the visibility status of the input control for the specified field by setting the `Visibility` property.

Syntax:

```
public static void SetVisibility<Field>(PXCache cache, object data,
                                          PXUIVisibility visibility)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `visibility`

  The value that is assigned to the `Visibility` property.

SetVisible(PXCache, object, string)

Makes the input control for the field with the specified name visible in the user interface by setting the `Visible` property to `true`.

Syntax:

```
public static void SetVisible(PXCache cache, object data, string name)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

SetVisible(PXCache, string, bool)

Shows or hides the input control for the field with the specified name in the user interface for all data record by setting the `Visible` property.

Syntax:

```
public static void SetVisible(PXCache cache, string name, bool isVisible)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `name`

  The field name.

- `isVisible`

  The value that is assigned to the `Enabled` property.

SetVisible(PXCache, object, string, bool)

Shows or hides the input control for the field with the specified name in the user interface by setting the `Visible` property.

Syntax:

```
public static void SetVisible(PXCache cache, object data,
                              string name, bool isVisible)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

- `isVisible`

  The value that is assigned to the `Enabled` property.

SetVisible<Field>(PXCache, object)

Makes the input control for the specified field visible in the user interface by setting the `Visible` property to `true`.

Syntax:

```
public static void SetVisible<Field>(PXCache cache, object data)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

SetVisible<Field>(PXCache, object, bool)

Shows or hides the input control for the specified field in the user interface by setting the `Visible` property.

Syntax:

```
public static void SetVisible<Field>(PXCache cache, object data, bool isVisible)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isVisible`

  The value that is assigned to the `Visible` property.

SetWarning(PXCache, object, string, string)

Sets the error string to display as a tooltip for the field with the specified name. The error level is set to `PXErrorLevel.Warning`.

Syntax:

```
public static void SetWarning(PXCache cache, object data,
                              string name, string error)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `name`

  The field name.

- `error`

  The error string displayed as a tooltip on the field input control.

SetWarning<Field>(PXCache, object, string)

Sets the error string to display as a tooltip for the specified field. The error level is set to `PXErrorLevel.Warning`.

Syntax:

```
public static void SetWarning<Field>(PXCache cache, object data,
                                     string error)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXUIField` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `error`

  The error string displayed as a tooltip on the field input control.

PXUIVisibility Enumeration

This enumeration is used to define:

- The visibility of an input control or a grid column in the webpage layout designer.
- The default set of columns displayed in the pop-up of the `PXSelector` input control.
- The set of columns automatically added to the `PXGrid` control with the `AutoGenerateColumns` property set to `AppendDynamic`, when no appropriate columns are defined for the `PXGrid` control.

Syntax

```
public enum PXUIVisibility
```

Members

- `Undefined`

  The visibility of a field input control or column is not defined.

- `Invisible = 1`

  The field input control or column is not displayed in the webpage layout designer.

- `Visible`

  The field input control or column is displayed in the webpage layout designer.

- `SelectorVisible = 4 | Visible`

  The field input control or column is displayed in the webpage layout designer. Also, the column that corresponds to the field is added to the `PXSelector` lookup control when the `PXSelector` attribute does not define the set of columns explicitly.

- `Dynamic`

  The field input control or column is displayed in the webpage layout designer., but the field colum is automatically added to the `PXGrid` control with the `AutoGenerateColumns` property value set to `AppendDynamic`, when the control has no appropriate column defined for this field.

PXErrorLevel Enumeration

This enumeration specifies the level of the `PXSetPropertyException` exception. Depending on the level, different error or warning signs are attached to UI controls associated with particular fields or rows.

Syntax

```
public enum PXErrorLevel
```

Members

- `Undefined`

  The `Error` sign is attached to the input controls or cells of the DAC fields whose `PXFieldState Error` property values are not null.
- `RowInfo`

  The `Information` sign is attached to a DAC row within the `PXGrid` control.
- `Warning`

  The `Warning` sign is attached to a DAC field input control or cell.
- `RowWarning`

  The `Warning` sign is attached to a DAC row within the `PXGrid` control.
- `Error`

  The `Error` sign is attached to a DAC field input control or a cell.
- `RowError`

  The `Error` sign is attached to a DAC row within the `PXGrid` control.

PXCacheRights Enumeration

Maps the user role's access rights for a specific `PXCache` object.

Syntax

```
public enum PXCacheRights
```

Members

- `Denied`

  Matches the roles for whom access to a `PXCache` object is denied.
- `Select`

  Matches the roles that are allowed to read data records of the DAC type corresponding to the `PXCache` object.
- `Update`

  Matches the roles that are allowed to update data records of the DAC type corresponding to the `PXCache` object.
- `Insert`

  Matches the roles that are allowed to insert data records of the DAC type corresponding to the `PXCache` object.
- `Delete`

  Matches the roles that are allowed to delete data records of the DAC type corresponding to the `PXCache` object.

Examples

Using the enumeration value to confiuge access rights for the button representing a graph action in the user interface:

```
public PXAction<ApproveBillsFilter> ViewDocument;
[PXUIField(DisplayName = "View Document",
           MapEnableRights = PXCacheRights.Update,
           MapViewRights = PXCacheRights.Select)]
[PXButton]
public virtual IEnumerable viewDocument(PXAdapter adapter)
{
    ...
}
```

The user with the select rights for the `ApproveBillsFilter` cache will see the **View Document** button in the user interface. For the user with the update rights for the `ApproveBillsFilter` cache, the **View Document** button will also be enabled.

Default Values

You can set the default values to DAC fields using the following attributes:

- *PXDefault* sets the default value and validates the field value on saving to the database. Derived attributes:
    - *PXUnboundDefault* behaves in the same way as `PXDefault`, but the default value is assigned to the field when a data record is retrieved from the database.
    - *PXDefaultValidate*
- *PXDBDefault* sets the default value using the value of some source field and updates the value if the source field value changes in the database before the data record is saved.

Differences

The first choice to set the default value to a DAC field, is the `PXDefault` attribute. You can set a constant as the default value or provide a BQL query to obtain a value from the database or data records from the cache. The default value is assigned to the field when a data record holding this field is inserted into the cache.

You can use the `PXDefault` just to make the field mandatory for input, by using the attribute without parameters.

The `PXDefault` attribute is not suitable when the default value is taken from a field that can be auto-generated by the database (such as the identity field). In this case, you should use the `PXDBDefault` attribute. It updates the value assigned to the field as default with the value generated by the database,

For example, if you implement a master-detail relationship, you should use the `PXDBDefault` attribute to bind the detail data record fields to master data record key fields. If the master data record is new, its identity field will be set to a real value by the database, when the master record is saved. So if a detail data record is created before the master data record is saved, the detail data record field will be set to the temprorary value of the master identity field. However, the `PXDBDefault` attribute will replace it with the real one on saving of the detail data record to the database.

You can use the `PXUnboundDefault` attribute to set the default value to an unbound field. The value is assigned when a data record is retrieved from the database (on the `RowSelecting` event).

PXDefault Attribute

Sets the default value for a DAC field.

See *Remarks* for more details. See *Examples* for examples of usage.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldDefaultingSubscriber`
- `IPXRowPersistingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
[PXAttributeFamily(typeof(PXDefaultAttribute))]
public class PXDefaultAttribute : PXEventSubscriberAttribute,
                                  IPXFieldDefaultingSubscriber,
                                  IPXRowPersistingSubscriber,
                                  IPXFieldSelectingSubscriber
```

Properties

- `public virtual bool` **`SearchOnDefault`**

  Gets or sets the value that indicates whether the BQL query specified calculate the default value is executed or ignored. By default, is `true` (the BQL query is executed).

- `public virtual PXPersistingCheck` **`PersistingCheck`**

  Gets or sets the *PXPersistingCheck* value that defines how to check the field value for null before saving a data record to the database. If a value doesn't pass a check, the attribute will throw the `PXRowPersistingException` exception. As a result, the save action will fail and the user will get an error message.

  By default, the property equals `PXPersistingCheck.Null`, which disallows null values. Note that for fields that are displayed in the user interface, this setting also disallows blank values (containing only whitespce characters).

- `public virtual Type` **`MapErrorTo`**

  Gets or sets the value that redirects the error from the field the attribute is placed on (source field) to another field. If an error happens on the source field, the error message will be displayed over the input control of the other field. The property can be set to a type derived from `IBqlField`. The BQL query is set in a constructor.

  Examples:

  ```
  [PXDefault(MapErrorTo = typeof(PMRegister.date))]
  public virtual String TranPeriodID { get; set; }
  ```

- `public virtual object` **`Constant`**

  Gets or sets a constant value that will be used as the default value.

- `public virtual Type` **`SourceField`**

  Gets or sets the field whose value will be taken from the BQL query result and used as the default value. The property can be set to a type derived from `IBqlField`. The BQL query is set in a constructor.

Examples:

```
[PXDefault(
    typeof(
        Select<VendorClass,
            Where<VendorClass.vendorClassID,
                  Equal<Current<Vendor.vendorClassID>>>>),
    SourceField = typeof(VendorClass.allowOverrideRate))]
public virtual Boolean? AllowOverrideRate { get; set; }
```

Constructors

Constructor	Description
PXDefaultAttribute()	Initializes a new instance that does not provide the default value, but checks whether the field value is not null before saving to the database
PXDefaultAttribute(Type)	Initializes a new instance that calculates the default value using the provided BQL query
PXDefaultAttribute(object)	Initializes a new instance that defines the default value as a constant value
PXDefaultAttribute(object, Type)	Initializes a new instance that calculates the default value using the provided BQL query and uses the constant value if the BQL query returns nothing
PXDefaultAttribute(TypeCode, string)	Converts the provided string to a specific type and Initializes a new instance that uses the conversion result as the default value
PXDefaultAttribute(TypeCode, string, Type)	Initializes a new instance that determines the default value using either the provided BQL query or the constant if the BQL query returns nothing

Static Methods

Method	Description
Select(PXGraph, BqlCommand, Type, string, object)	
SetDefault(PXCache, string, object)	Sets the new default value of the field with the specified name for all data records in the cache
SetDefault(PXCache, object, string, object)	Sets the new default value of the field with the specified name for a particular data record
SetDefault<Field>(PXCache, object)	Sets the new default value of the specified field for all data records in the cache
SetDefault<Field>(PXCache, object, object)	Sets the new default value of the specified field for a particular data record
SetPersistingCheck(PXCache, string, object,)	Sets the `PersistingCheck` property for the field with the specified name in a particular data record
SetPersistingCheck<Field>(PXCache, object,)	Sets the `PersistingCheck` property for the specifed field in a particular data record

Remarks

The `PXDefault` attribute provides the default value for a DAC field. The default value is assigned to the field when the cache raises the `FiedlDefaulting` event. This happens when the a new row is inserted in code or through the user interface.

A value specified as default can be a constant or the result of a BQL query. If you provide a BQL query, the attribute will execute it on the `FieldDefaulting` event. You can specify both, in which case the attribute first executes the BQL query and uses the constant if the BQL query returns an empty set. If you provide a DAC field as the BQL query, the attribute takes the value of this field from the cache object's `Current` property. The attribute uses the cache object of the DAC type in which the field is defined.

The `PXDefault` attribute also checks that the field value is not `null` before saving to the database. You can adjust this behavior using the `PersistingCheck` property. Its value indicates whether the attribute should check that the value is not `null`, check that the value is not `null` or a blank string, or not check.

The attribute can redirect the error that happened on the field to another field if you set the `MapErrorTo` property.

You can use the static methods to change the attribute properties for a particular data record in the cache or for all data record in the cache.

Examples

The attribute below sets a constant as the default value.

```
[PXDefault(false)]
public virtual bool? IsActive { get; set; }
```

The attribute below provides a string constants that is converted to the default value of the specific type.

```
[PXDefault(TypeCode.Decimal, "0.0")]
public virtual Decimal? AdjDiscAmt { get; set; }
```

The attribute below will take the default value from the `ARPayment` cache object and won't check the field value on saving of the changes to the database.

```
[PXDefault(typeof(ARPayment.adjDate),
           PersistingCheck = PXPersistingCheck.Nothing)]
public virtual DateTime? TillDate { get; set; }
```

The attribute below only prevents the field from being null and does not set a default value.

```
[PXDefault]
public virtual string BAccountAcctCD { get; set; }
```

The attribute below will execute the `Search` BQL query and take the `CAEntryType.ReferenceID` field value from the result.

```
[PXDefault(typeof(
    Search<CAEntryType.referenceID,
        Where<CAEntryType.entryTypeId,
            Equal<Current<AddTrxFilter.entryTypeID>>>>))]
public virtual int? ReferenceID { get; set; }
```

The attribute below will execute the `Select` BQL query and take the `VendorClass.AllowOverrideRate` field value from the result or will use `false` as the default value if the BQL query returns an empty set.

```
[PXDefault(
    false,
    typeof(
```

```
        Select<VendorClass,
            Where<VendorClass.vendorClassID,
                  Equal<Current<Vendor.vendorClassID>>>>),
    SourceField = typeof(VendorClass.allowOverrideRate))]
public virtual Boolean? AllowOverrideRate { get; set; }
```

Setting a new default value to a field at run time:

```
// The view declaration in a graph
public PXSelect<ARAdjust> Adjustments;
...
// The code executed in some graph method
PXDefaultAttribute.SetDefault<ARAdjust.adjdDocType>(Adjustments.Cache, "CRM");
```

Changing the way the attribute checks the field value on saving of the changes to the database:

```
protected virtual void ARPayment_RowSelected(PXCache cache, PXRowSelectedEventArgs
 e)
{
    ARPayment doc = e.Row as ARPayment;
    ...
    PXDefaultAttribute.SetPersistingCheck<ARPayment.depositAfter>(
        cache, doc,
        isPayment && (doc.DepositAsBatch == true)?
            PXPersistingCheck.NullOrBlank : PXPersistingCheck.Nothing);
    ...
}
```

Related Types

- *PXPersistingCheck Enumeration*

PXDefault Attribute Constructors

The *PXDefault* attribute exposes the following constructors.

PXDefaultAttribute()

Initializes a new instance that does not provide the default value, but checks whether the field value is not null before saving to the database.

Syntax:

```
public PXDefaultAttribute()
```

PXDefaultAttribute(Type)

Initializes a new instance that calculates the default value using the provided BQL query.

Syntax:

```
public PXDefaultAttribute(Type sourceType)
```

Parameters:

- `sourceType`

  The BQL query that is used to calculate the default value. Accepts the types derived from: `IBqlSearch`, `IBqlSelect`, `IBqlField`, `IBqlTable`.

PXDefaultAttribute(object)

Initializes a new instance that defines the default value as a constant value.

Syntax:

```
public PXDefaultAttribute(object constant)
```

Parameters:

- `constant`

  Constant value that is used as the default value.

PXDefaultAttribute(object, Type)

Initializes a new instance that calculates the default value using the provided BQL query and uses the constant value if the BQL query returns nothing. If the BQL query is of `Select` type, you should also explicitly set the `SourceField` property. If the BQL query is a DAC field, the attribute will take the value from the `Current` property of the cache object corresponding to the DAC.

Syntax:

```
public PXDefaultAttribute(object constant, Type sourceType) : this(sourceType)
```

Parameters:

- `constant`

  Constant value that is used as the default value.

- `sourceType`

  The BQL query that is used to calculate the default value. Accepts the types derived from: `IBqlSearch`, `IBqlSelect`, `IBqlField`, `IBqlTable`.

PXDefaultAttribute(TypeCode, string)

Converts the provided string to a specific type and Initializes a new instance that uses the conversion result as the default value.

Syntax:

```
public PXDefaultAttribute(TypeCode converter, string constant)
```

Parameters:

- `converter`

  The type code that specifies the type to covert the string to.

- `constant`

  The string representation of the constant used as the default value.

PXDefaultAttribute(TypeCode, string, Type)

Initializes a new instance that determines the default value using either the provided BQL query or the constant if the BQL query returns nothing.

Syntax:

```
public PXDefaultAttribute(TypeCode converter, string constant, Type sourceType) :
            this(sourceType)
```

Parameters:

- `converter`

  The type code that specifies the type to convert the string constant to.

- `constant`

  The string representation of the constant used as the default value if the BQL query returns nothing.

- `sourceType`

  The BQL command that is used to calculate the default value. Accepts the types derived from: `IBqlSearch`, `IBqlSelect`, `IBqlField`, `IBqlTable`.

PXDefault Attribute Methods

The *PXDefault* attribute exposes the following static methods.

Select(PXGraph, BqlCommand, Type, string, object)

Syntax:

```
public static object Select(PXGraph graph, BqlCommand Select, Type sourceType,
 string sourceField, object row)
```

SetDefault(PXCache, string, object)

Sets the new default value of the field with the specified name for all data records in the cache.

Syntax:

```
public static void SetDefault(PXCache cache, string field, object def)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDefault` type.

- `field`

  The name of the field to set the default value to.

- `def`

  The new default value.

SetDefault(PXCache, object, string, object)

Sets the new default value of the field with the specified name for a particular data record.

Syntax:

```
public static void SetDefault(PXCache cache, object data, string field, object def)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDefault` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `field`

  The name of the field to set the default value to.

- `def`

The new default value.

SetDefault<Field>(PXCache, object)

Sets the new default value of the specified field for all data records in the cache.

Syntax:

```
public static void SetDefault<Field>(PXCache cache, object def)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDefault` type.
- `def`

  The new default value.

SetDefault<Field>(PXCache, object, object)

Sets the new default value of the specified field for a particular data record.

Syntax:

```
public static void SetDefault<Field>(PXCache cache, object data, object def)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDefault` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.
- `def`

  The new default value.

SetPersistingCheck(PXCache, string, object, PXPersistingCheck)

Sets the `PersistingCheck` property for the field with the specified name in a particular data record.

Syntax:

```
public static void SetPersistingCheck(PXCache cache, string field,
                                      object data, PXPersistingCheck check)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDefault` type.
- `field`

  The field name.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `def`

  The value that is set to the property.

SetPersistingCheck<Field>(PXCache, object, PXPersistingCheck)

Sets the `PersistingCheck` property for the specifed field in a particular data record.

Syntax:

```
public static void SetPersistingCheck<Field>(PXCache cache, object data,
                                              PXPersistingCheck check)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDefault` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `def`

  The value that is set to the property.

PXPersistingCheck Enumeration

Defines different ways the `PXDefault` attribute checks the field value before a data record with this field is saved to the database.

Syntax

```
public enum PXPersistingCheck
```

Members

- `Null`

  Check that the field value is not `null`.

  Note that the user interface (UI) trims string values, so for fields displayed in the UI, values containing only whitespace characters will also be rejected.

- `NullOrBlank`

  Check that the field value is not `null` and is not a string that contains only whitespace characters.

- `Nothing`

  Do not check the field value.

PXDBDefault Attribute

Sets the default value for a DAC field. Use to assign a value from the auto-generated key field.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldDefaultingSubscriber`

- `IPXRowPersistingSubscriber`
- `IPXRowPersistedSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
                AttributeTargets.Class, AllowMultiple = true)]
public class PXDBDefaultAttribute : PXEventSubscriberAttribute,
                                    IPXFieldDefaultingSubscriber,
                                    IPXRowPersistingSubscriber,
                                    IPXRowPersistedSubscriber
```

Properties

- `public virtual PXPersistingCheck` **`PersistingCheck`**

  Gets or sets the *PXPersistingCheck* value that defines how to check the field value before saving a data record to the database. The attribute either checks that the value is not `null`, checks that the value is `null` or a blank string (contains only whitespace characters), or doesn't check the value. If the attribute discovers that the value is in fact `null` or blank, it will throw the `PXRowPersistingException` exception. As a result, the save action will fail and the user will get an error message.

- `public bool` **`DefaultForUpdate`**

  Gets or sets the value that indicates whether the default value is reassigned on a database update operation.

- `public bool` **`DefaultForInsert`**

  Gets or sets the value that indicates whether the default value is reassigned on a database insert operation.

Constructors

- `public PXDBDefaultAttribute(Type sourceType)`

  Initializes a new instance of the attribute. Obtains the default value using the provided BQL query.

  Parameters:

  - `sourceType`

    The BQL query that is used to calculate the default value. Accepts the types derived from: `IBqlSearch`, `IBqlSelect`, `IBqlField`, `IBqlTable`.

Static Methods

Method	Description
SetDefaultForInsert<Field>(PXCache, object, bool)	Sets the `DefaultForInsert` property for a particular data record
SetDefaultForUpdate<Field>(PXCache, object, bool)	Sets the `DefaultForUpdate` property for a particular data record

Examples

Setting the default value that will be taken from the current `POReceipt` cache object and reassigned only on insertion of the data record to the database:

```
public partial class LandedCostTran : PX.Data.IBqlTable
{
    ...
    [PXDBString(3, IsFixed = true)]
    [PXDBDefault(typeof(POReceipt.receiptType),
                 DefaultForUpdate = false)]
    public virtual string POReceiptType { get; set; }
    ...
}
```

Changing the `SetDefaultForUpdate` property:

```
PXDBDefaultAttribute.SetDefaultForUpdate<SOOrderShipment.shipAddressID>(
    OrderList.Cache, null, false);
```

The method sets the property for the `ShipAddressID` field in all data records in the cache object associated with the `OrderList` view:

PXDBDefault Attribute Methods

The *PXDBDefault* attribute exposes the following static methods.

SetDefaultForInsert<Field>(PXCache, object, bool)

Sets the `DefaultForInsert` property for a particular data record.

Syntax:

```
public static void SetDefaultForInsert<Field>(
    PXCache cache, object data, bool def)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PDBXDefault` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `def`

  The new value for the property.

SetDefaultForUpdate<Field>(PXCache, object, bool)

Sets the `DefaultForUpdate` property for a particular data record.

Syntax:

```
public static void SetDefaultForUpdate<Field>(
    PXCache cache, object data, bool def)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXDBDefault` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `def`

  The new value for the property.

PXUnboundDefault Attribute

Sets the default value to an unbound DAC field. The value is assigned to the field when the data record is retrieved from the database.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDefaultAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`

Syntax

```
public class PXUnboundDefaultAttribute : PXDefaultAttribute,
                                          IPXRowSelectingSubscriber
```

Constructors

Constructor	Description
PXUnboundDefaultAttribute()	Initializes a new instance that does not provide the default value, but checks whether the field value is not null before saving to the database
PXUnboundDefaultAttribute(Type)	Initializes a new instance that calculates the default value using the provided BQL query
PXUnboundDefaultAttribute(object)	Initializes a new instance that defines the default value as a constant value
PXUnboundDefaultAttribute(object, Type)	Initializes a new instance that calculates the default value using the provided BQL query and uses the constant value if the BQL query returns nothing
PXUnboundDefaultAttribute(TypeCode, string)	Converts the provided string to a specific type and initializes a new instance that uses the conversion result as the default value
PXUnboundDefaultAttribute(TypeCode, string, Type)	Initializes a new instance that determines the default value using either the provided BQL query or the constant if the BQL query returns nothing

Remarks

This attributes is similar to the `PXDefault` attribute, but, unlike the `PXDefault` attribute, it assigns the provided default value to the field when a data record is retrieved from the database (on the `RowSelecting` event). The `PXDefault` attribute assigns the default value to the field when a data record is inserted to the cache object.

Examples

```
[PXDecimal(4)]
[PXUnboundDefault(TypeCode.Decimal, "0.0")]
public virtual Decimal? DocBal { get; set; }
```

PXUnboundDefault Attribute Constructors

The *PXUnboundDefault* attribute exposes the following constructors.

PXUnboundDefaultAttribute()

Initializes a new instance that does not provide the default value, but checks whether the field value is not null before saving to the database.

Syntax:

```
public PXUnboundDefaultAttribute()
```

PXUnboundDefaultAttribute(Type)

Initializes a new instance that calculates the default value using the provided BQL query.

Syntax:

```
public PXUnboundDefaultAttribute(Type sourceType) : base(sourceType)
```

Parameters:

- `sourceType`

  The BQL query that is used to calculate the default value. Accepts the types derived from: `IBqlSearch`, `IBqlSelect`, `IBqlField`, `IBqlTable`.

PXUnboundDefaultAttribute(object)

Initializes a new instance that defines the default value as a constant value.

Syntax:

```
public PXUnboundDefaultAttribute(object constant) : base(constant)
```

Parameters:

- `constant`

  Constant value that is used as the default value.

PXUnboundDefaultAttribute(object, Type)

Initializes a new instance that calculates the default value using the provided BQL query and uses the constant value if the BQL query returns nothing. If the BQL query is of `Select` type, you should also explicitly set the `SourceField` property. If the BQL query is a DAC field, the attribute will take the value from the `Current` property of the cache object corresponding to the DAC.

Syntax:

```
public PXUnboundDefaultAttribute(object constant, Type sourceType)
    : base(constant, sourceType)
```

Parameters:

- `constant`

  Constant value that is used as the default value.

- `sourceType`

  The BQL query that is used to calculate the default value. Accepts the types derived from: `IBqlSearch`, `IBqlSelect`, `IBqlField`, `IBqlTable`.

PXUnboundDefaultAttribute(TypeCode, string)

Converts the provided string to a specific type and initializes a new instance that uses the conversion result as the default value.

Syntax:

```
public PXUnboundDefaultAttribute(TypeCode converter, string constant)
    : base(converter, constant)
```

Parameters:

- `converter`

  The type code that specifies the type to covert the string to..

- `constant`

  The string representation of the constant used as the default value.

PXUnboundDefaultAttribute(TypeCode, string, Type)

Initializes a new instance that determines the default value using either the provided BQL query or the constant if the BQL query returns nothing.

Syntax:

```
public PXUnboundDefaultAttribute(TypeCode converter, string constant, Type
 sourceType)
    : base(converter, constant, sourceType)
```

Parameters:

- `converter`

  The type code that specifies the type to convert the string constant to.

- `constant`

  The string representation of the constant used as the default value if the BQL query returns nothing.

- `sourceType`

  The BQL command that is used to calculate the default value. Accepts the types derived from: `IBqlSearch`, `IBqlSelect`, `IBqlField`, `IBqlTable`.

PXDefaultValidate Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDefaultAttribute
```

Syntax

```
public class PXDefaultValidateAttribute : PXDefaultAttribute
```

Constructors

- `public PXDefaultValidateAttribute(Type sourceType, Type validateExists) : base(sourceType)`

Complex Input Controls

You should use attributes to configure complex input controls such as dropdown lists and lookup control.

Dropdown Lists

The following attributes configure a dropdown list that will represent a DAC field in the user interface:

- *PXStringList*

  Configures the dropdown list that will let a user select from a fixed set of strings.
- *PXIntList*

  Configures the dropdown list that will let a user select from a fixed set of values. The control displays strings, while the field is assigned the integer value corresponding to the selected string.
- *PXDecimalList*

  Configures the dropdown list that will let a user select from a fixed set of strings converted to decimal values.
- *PXImagesList*
- *PXDBIntList*
- *PXDBStringList*

Lookup Controls

The following attributes configure a lookup control that will represent a field in the user interface:

- *PXSelector*

  Configures the lookup control for a DAC field that references a data record from a particular table by holding its key.
- *PXCustomSelector*

  The base class to derive custom attributes configuring lookup controls.
- *PXRestrictor*

  Adds a restriction to a BQL command that selects data for a lookup control and displays the error message when the value entered does not fit the restriction.

Segmented Key Controls

A segmented key value is a string value that identifies a data record and consists of one or several segments. A segmented key is an entity identified by a string (referred to as *dimension*). A segmented key is associated with segments. For each segment, you can define the list of possible values. You can create a new segment when the data records identified by the segmented key already exist in the database.

The following attributes configure a control to input a segmented key value in the user interface:

- *PXDimension*

Configures the input control that formats an input as a segmented key value and displays the list of allowed values for each key segment.

- *PXDimensionSelector*

  Configures the input control that combines functionality of the `PXDimenstion` attribute and the `PXSelector` attribute. A user can observe the data set defined by the attribute and select a data record from this data set to assign its segmented key value to the field or to substitute it with the surrogate key.

- *PXDimensionWildcard*

  Behaves as the `PXDimensionSelector` attribute, but additionally allows the *?* character treated as a wildcard.

PXStringList Attribute

Sets a dropdown list as the input control for a DAC field. The control will let a user select from a fixed set of strings or input a value manually.

See *Remarks* for more details. See *Examples* for examples of usage.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`
- `IPXLocalizableList`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Parameter |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXBaseListAttribute))]
public class PXStringListAttribute : PXEventSubscriberAttribute,
                                     IPXFieldSelectingSubscriber,
                                     IPXLocalizableList
```

Properties

- `public Dictionary<string, string>` **`ValueLabelDic`**

  Gets the dictionary of allowed value-label pairs.

- `public virtual bool` **`IsLocalizable`**

  Gets or sets the value that indicates whether the values and labels used by the attribute are localizable.

- `public bool` **`ExclusiveValues`**

  Gets or sets the value that indicates whether a user can input a value not present in the list of allowed values. If `true`, it is prohibited. By default, the property is set to `true`, which means that the user can select only from the values in the dropdown list.

- `public virtual Type` **`BqlField`**

  Returns `null` on get. Sets the BQL field representing the field in BQL queries.

Constructors

Constructor	Description
PXStringListAttribute()	Initializes a new instance with empty lists of allowed values and labels
PXStringListAttribute(string)	Initializes a new instance with the list of allowed values obtained from the provided string
PXStringListAttribute(string[], string[])	Initializes a new instance with the specified lists of allowed values and corresponding labels

Static Methods

Method	Description
AppendList(PXCache, object, string, string[], string[])	Extends the lists of allowed values and labels in the attribute instance that marks the field with the specified name in a particular data record
AppendList<Field>(PXCache, object, string[], string[])	Extends the lists of allowed values and labels in the attribute instance that marks the specified field in a particular data record
SetList(PXCache, object, string,)	Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the field with the specified name in a particular data record
SetList(PXCache, object, string, string[], string[])	Sets the lists of allowed values and labels for the attribute instance that marks the field with the specified name in a particular data record
SetList<Field>(PXCache, object, PXStringListAttribute)	Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the specified field in a particular data record
SetList<Field>(PXCache, object, string[], string[])	Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the specified field in a particular data record

Remarks

The attribute configures a dropdown list that will represent the DAC field in the user interface. You should provide the list of allowed string values and the list of the corresponding labels in the attribute constructor.

You can reconfigure the dropdown list at run time by calling the static methods. You can set a different list of values of labels or extend the list.

Examples

The attribute is added to the DAC field definition as follows.

```
[PXStringList(
    new[] { "N", "P", "I", "F" },
    new[] { "New", "Prepared", "Processed", "Partially Processed" }
)]
[PXDefault("N")]
public virtual string Status { get; set; }
```

The attribute below obtains the list of values from the provided string.

```
[PXStringList("Dr.,Miss,Mr,Mrs,Prof.")]
public virtual string TitleOfCourtesy { get; set; }
```

The attribute below obtains the lists of values and labels from the provided string. The user will select from *Import* and *Export*. While the field will be set to *I* or *E*.

```
[PXStringList("I;Import,E;Export")]
public virtual string TitleOfCourtesy { get; set; }
```

The example below demonstrates an invocation of a `PXStringListAttribute` static method.

```
List<string> values = new List<string>();
List<string> labels = new List<string>();
... // Fill the values and labels lists
// Specify as arrays of values and labels of the dropdown list
PXStringListAttribute.SetList<AUSchedule.actionName>(
    Schedule.Cache, null, values.ToArray(), labels.ToArray());
```

The method called in the example will set the new lists of values and labels for all data records in the cache object the `Schedule.Cache` variable references. The method will assign the lists to the `PXStringList` attribute instances attached to the `ActionName` field.

PXStringList Attribute Constructors

The *PXStringList* attribute exposes the following constructors.

PXStringListAttribute()

Initializes a new instance with empty lists of allowed values and labels.

Syntax:

```
public PXStringListAttribute() : base()
```

PXStringListAttribute(string)

Initializes a new instance with the list of allowed values obtained from the provided string. The string should contain either values separated by a comma, or value-label pairs where the value and label are separated by a semicolon and different pairs are separated by a comma. In the first case labels are set to value strings.

Syntax:

```
public PXStringListAttribute(string list) : this()
```

Parameters:

- `list`

  The string containing the list of values or value-label pairs.

PXStringListAttribute(string[], string[])

Initializes a new instance with the specified lists of allowed values and corresponding labels. When a user selects a label in the user interface, the corresponding value is assigned to the field marked by the instance. The two lists must be of the same length.

Syntax:

```
public PXStringListAttribute(string[] allowedValues, string[] allowedLabels) :
            this()
```

Parameters:

- `allowedValues`

  The list of values assigned to the field when a user selects the corresponding labels..

- `allowedLabels`

  The list of labels displayed in the user interface when a user expands the control.

PXStringList Attribute Methods

The *PXStringList* attribute exposes the following static methods.

AppendList(PXCache, object, string, string[], string[])

Extends the lists of allowed values and labels in the attribute instance that marks the field with the specified name in a particular data record.

Syntax:

```
public static void AppendList(PXCache cache, object data, string field,
                              string[] allowedValues, string[] allowedLabels)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXStringList` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `allowedValues`

  The list of values that is appended to the existing list of values.

- `allowedLabels`

  The list of labels that is appended to the existing list of labels.

AppendList<Field>(PXCache, object, string[], string[])

Extends the lists of allowed values and labels in the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static void AppendList<Field>(PXCache cache, object data,
                                     string[] allowedValues,
                                     string[] allowedLabels)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXStringList` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `allowedValues`

  The list of values that is appended to the existing list of values.

- `allowedLabels`

  The list of labels that is appended to the existing list of labels.

SetList(PXCache, object, string, PXStringListAttribute)

Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the field with the specified name in a particular data record.

Syntax:

```
public static void SetList(PXCache cache, object data, string field,
                           PXStringListAttribute listSource)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXStringList` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `field`

  The name of the field that is be marked with the attribute.

- `listSource`

  The attribute instance from which the lists of allowed values and labels are obtained.

SetList(PXCache, object, string, string[], string[])

Sets the lists of allowed values and labels for the attribute instance that marks the field with the specified name in a particular data record.

Syntax:

```
public static void SetList(PXCache cache, object data, string field,
                           string[] allowedValues, string[] allowedLabels)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXStringList` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `field`

  The name of the field that is be marked with the attribute.

- `allowedValues`

  The new list of values.

- `allowedLabels`

  The new list of labels.

SetList<Field>(PXCache, object, PXStringListAttribute)

Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static void SetList<Field>(PXCache cache, object data,
                                  PXStringListAttribute listSource)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXStringList` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `listSource`

  The attribute instance from which the lists of allowed values and labels are obtained.

SetList<Field>(PXCache, object, string[], string[])

Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static void SetList<Field>(PXCache cache, object data,
                                  string[] allowedValues,
                                  string[] allowedLabels)
    where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXStringList` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `allowedValues`

  The new list of values.

- `allowedLabels`

  The new list of labels.

PXDecimalList Attribute

Sets a dropdown list as the input control for a DAC field of `decimal` type.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXStringListAttribute
```

Syntax

```
public class PXDecimalListAttribute : PXStringListAttribute
```

Constructors

- `public PXDecimalListAttribute(string[] values, string[] labels) : base(values, labels)`

  Initializes a new instance with the provided lists of allowed values and labels. When a user selects a label in the user interface, the corresponding value is converted to `decimal` type and assigned to the field marked by the instance. The two lists must be of the same length.

  Parameters:

  - `values`

    The array of string values the user will be able to select from. A string value is converted by the attribute to the `decimal` value.
  - `labels`

    The array of labels corresponding to `values` and displayed in the user interface.

Remarks

The user will be able to select a value from the predefined values list. Values are specified in the constructor as strings, because the attribute derives from `PXStringList`. The attribute converts a selected value to the `decimal` type that is assigned to the field.

The DAC field data type must be defined using the *PXDBDecimalString* attribute.

Examples

```
[PXDecimalList(
    new string[] { "0.1", "0.5", "1.0", "10", "100" },
    new string[] { "0.1", "0.5", "1.0", "10", "100" })]
public virtual decimal? InvoicePrecision { get; set; }
```

PXImagesList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXStringListAttribute
```

Syntax

```
public class PXImagesListAttribute : PXStringListAttribute
```

Properties

- `public override bool` **`IsLocalizable`**

Constructors

Constructor	Description
PXImagesListAttribute()	

Constructor	Description
PXImagesListAttribute(string[], string[], string[])	

PXImagesList Attribute Constructors

The *PXImagesList* attribute exposes the following constructors.

PXImagesListAttribute()

Syntax:

```
public PXImagesListAttribute()
```

PXImagesListAttribute(string[], string[], string[])

Syntax:

```
public PXImagesListAttribute(string[] allowedValues, string[] allowedLabels,
 string[] allowedImages) : base(allowedValues, allowedLabels)
```

PXIntList Attribute

Sets a dropdown list as the input control for a DAC field. The control will let a user select from a fixed set of integer values represented in the dropdown list by string labels.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`
- `IPXLocalizableList`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Parameter |
                AttributeTargets.Method)]
[PXAttributeFamily( typeof(PXBaseListAttribute))]
public class PXIntListAttribute : PXEventSubscriberAttribute,
                                  IPXFieldSelectingSubscriber,
                                  IPXLocalizableList
```

Properties

- `public virtual bool` **`IsLocalizable`**

  Gets or sets the value that indicates whether the labels used by the attribute are localizable.
- `public Dictionary<int, string>` **`ValueLabelDic`**

  Gets the dictionary of allowed value-label pairs.

Constructors

Constructor	Description
PXIntListAttribute()	Initializes a new instance with empty lists of allowed values and labels
PXIntListAttribute(string)	Initializes a new instance with the list of allowed values obtained from the provided string
PXIntListAttribute(Type)	Initializes a new instance, extracting the list of allowed values and labels from the provided enumeration
PXIntListAttribute(int[], string[])	Initializes a new instance with the specified lists of allowed values and corresponding labels

Static Methods

Method	Description
SetList<Field>(PXCache, object, int[],)	Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the specified field in a particular data record

Remarks

The attribute configures a dropdown list that will represent the DAC field in the user interface. You should provide the list of allowed integer values and the list of the corresponding labels in the attribute constructor.

You can reset the lists of values and labels at run time by calling the `SetList<>` static method.

Examples

```
[PXIntList(
    new int[] { 0, 1 },
    new string[] { "Apply Credit Hold", "Release Credit Hold" })]
public virtual int? Action { get; set; }
```

PXIntList Attribute Constructors

The *PXIntList* attribute exposes the following constructors.

PXIntListAttribute()

Initializes a new instance with empty lists of allowed values and labels.

Syntax:

```
public PXIntListAttribute()
```

PXIntListAttribute(string)

Initializes a new instance with the list of allowed values obtained from the provided string. The string should contain either values separated by a comma, or value-label pairs where the value and label are separated by a semicolon and different pairs are separated by a comma. In the first case labels are set to value strings. Values are converted from strings into integers.

Syntax:

```
public PXIntListAttribute(string list) : this()
```

Parameters:

- `list`

  The string containing the list of values separated by comma.

PXIntListAttribute(Type)

Initializes a new instance, extracting the list of allowed values and labels from the provided enumeration. Uses the enumeration values as allowed values and enumeration values names as the corresponding labels.

Syntax:

```
public PXIntListAttribute(Type enumType) : this()
```

Parameters:

- `enumType`

  The `enum` type that defines the lists of allowed values and labels.

PXIntListAttribute(int[], string[])

Initializes a new instance with the specified lists of allowed values and corresponding labels. When a user selects a label in the user interface, the corresponding value is assigned to the field marked by the instance. The two lists must be of the same length.

Syntax:

```
public PXIntListAttribute(int[] allowedValues, string[] allowedLabels) : this()
```

Parameters:

- `allowedValues`

  The list of values assigned to the field when a user selects the corresponding labels..

- `allowedLabels`

  The list of labels displayed in the user interface when a user expands the control.

PXIntList Attribute Methods

The *PXIntList* attribute exposes the following static methods.

SetList<Field>(PXCache, object, int[], string[])

Sets the lists of allowed values and labels from the provided instance to the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static void SetList<Field>(PXCache cache, object data, int[] allowedValues,
            string[] allowedLabels) where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXIntList` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.

- `allowedValues`

  The new list of values.
- `allowedLabels`

  The new list of labels.

PXDBIntList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXBaseListAttribute
```

Syntax

```
public sealed class PXDBIntListAttribute : PXBaseListAttribute
```

Constructors

- `public PXDBIntListAttribute(Type table, Type valueField, Type descriptionField) : base(new PXDBIntAttributeHelper(table, valueField, descriptionField))`

PXDBStringList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXBaseListAttribute
```

Syntax

```
public sealed class PXDBStringListAttribute : PXBaseListAttribute
```

Constructors

- `public PXDBStringListAttribute(Type table, Type valueField, Type descriptionField) : base(new PXDBStringAttributeHelper(table, valueField, descriptionField))`

PXSelector Attribute

Configures the lookup control for a DAC field that references a data record from a particular table by holding its key field.

See *Remarks* for more details. See *Examples* for examples of usage.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldVerifyingSubscriber`

- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                 AttributeTargets.Class |
                 AttributeTargets.Parameter |
                 AttributeTargets.Method)]
[PXAttributeFamily(typeof(PXSelectorAttribute))]
public class PXSelectorAttribute : PXEventSubscriberAttribute,
                                    IPXFieldVerifyingSubscriber,
                                    IPXFieldSelectingSubscriber
```

Properties

- `public virtual Type` **`DescriptionField`**

  Gets or sets the field from the referenced table that contains the description.

- `public virtual Type` **`SubstituteKey`**

  Gets or sets the field from the referenced table that substitutes the key field used as internal value and is displayed as a value in the user interface (natural key).

- `public virtual Type` **`Field`**

  Gets the field that identifies a referenced data record (surrogate key) and is assigned to the field annotated with the `PXSelector` attribute. Typically, it is the first parameter of the BQL query passed to the attribute constructor.

- `public virtual bool` **`DirtyRead`**

  Gets or sets a value that indicates whether the attribute should take into account the unsaved modifications when displaying data records in control. If `false`, the data records are taken from the database and not merged with the cache object. If `true`, the data records are merged with the modification stored in the cache object.

- `public virtual bool` **`Filterable`**

  Gets or sets the value that indicates whether the filters defined by the user should be stored in the database.

- `public virtual bool` **`CacheGlobal`**

  Gets or sets the value that indicates whether the attribute should cache the data records retrieved from the database to show in the lookup control. By default, the attribute does not cache the data records.

- `public virtual string[]` **`Headers`**

  Gets or sets the list of labels for column headers that are displayed in the lookup control. By default, the attribute uses display names of the fields.

- `public BqlCommand` **`PrimarySelect`**

  Gets the BQL query that is used to retrieve data records to show to the user.

- `public int` **`ParsCount`**

  Get.

Constructors

Constructor	Description
PXSelectorAttribute(Type)	Initializes a new instance that will use the specified BQL query to retrieve the data records to select from
PXSelectorAttribute(Type, params Type[])	Initializes a new instance that will use the specified BQL query to retrieve the data records to select from, and display the provided set of columns

Static Methods

Method	Description
ClearGlobalCache(Type)	Clears the internal cache of the `PXSelector` attribute, removing the data records retrieved from the specified table
ClearGlobalCache<Table>()	Clears the internal cache of the `PXSelector` attribute, removing the data records retrieved from the specified table
GetField(PXCache, object, string, object,)	Returns a value of the field from a foreign data record
GetItem(PXCache, PXSelectorAttribute, object, object)	Returns the foreign data record by the specified key
GetItemType(PXCache, string)	Returns the data access class referenced by the attribute instance that marks the field with specified name
GetSelectorFields(Type)	
Select(PXCache, object, string)	Returns the data record referenced by the attribute instance that marks the field with the specified name in a particular data record
Select(PXCache, object, string, object)	Returns the referenced data record that holds the specified value
Select<Field>(PXCache, object)	Returns the data record referenced by the attribute instance that marks the specified field in a particular data record
Select<Field>(PXCache, object, object)	Returns the referenced data record that holds the specified value
SelectAll(PXCache, string, object)	Returns all data records kept by the attribute instance the marks the field with the specified name in a particular data record
SelectAll<Field>(PXCache, object)	Returns all data records kept by the attribute instance the marks the specified field in a particular data record
SelectFirst(PXCache, object, string)	Returns the first data record retrieved by the attribute instance that marks the field with the specified name in a particular data record
SelectFirst<Field>(PXCache, object)	Returns the first data record retrieved by the attribute instance that marks the specified field in a particular data record

Method	Description
SelectLast(PXCache, object, string)	Returns the last data record retrieved by the attribute instance that marks the field with the specified name in a particular data record
SelectLast<Field>(PXCache, object)	Returns the last data record retrieved by the attribute instance that marks the specified field in a particular data record
SetColumns(PXCache, string, string[], string[])	Sets the list of columns and column headers for all attribute instances that mark the field with the specified name in all data records in the cache object
SetColumns(PXCache, object, string, string[], string[])	Sets the list of columns and column headers to display for the attribute instance that marks the field with the specified name in a particular data record
SetColumns<Field>(PXCache, Type[], string[])	Sets the list of columns and column headers for all attribute instances that mark the specified field in all data records in the cache object
SetColumns<Field>(PXCache, object, Type[],)	Sets the list of columns and column headers to display for the attribute instance that marks the specified field in a particular data record
StoreCached<Field>(PXCache, object, object)	

Remarks

The attribute configures the input control for a DAC field that references a data record from a particular table. Such field holds a key value that identifies the data record in this table.

The input control will be of "lookup" type (may also be called a "selector"). A user can either input the value for the field manually or select from the list of the data records. If a value is inserted manually, the attribute checks if it is included in the list. You can specify a complex BQL query to define the set of data records that appear in the list.

The key field usually represents a database identity column that may not be user-friendly (surrogate key). It is possible to substitute its value with the value of another field from the same data record (natural key). This field should be specified in the `SubstituteKey` property. In this case, the table, and the DAC, have two fields that uniquely identify a data record from this table. For example, the `Account` table may have the numeric `AccountID` field and the user-friendly string `AccountCD` field. On a field that references `Account` data records in another DAC, you should place the `PXSelector` attribute as follows.

```
[PXSelector(typeof(Search<Account.accountID>),
            SubstituteKey = typeof(Account.accountCD))]
```

The attribute will automatically convert the stored numeric value to the displayed string value and back. Note that only the `AccountCD` property should be marked with `IsKey` property set to `true`.

It is also possible to define the list of columns to display. You can use an appropriated constructor and specify the types of the fields. By default, all fields that have the `PXUIField` attribute's `Visibility` property set to `PXUIVisibility.SelectorVisible`.

Along with a key, some other field can be displayed as the description of the key. This field should be specified in the `DescriptionField` property. The way the description is displayed in the lookup control is configured in the webpage layout through the `DisplayMode` property of the `PXSelector` control. The default display format is *ValueField – DescriptionField*. It can be changed to display the description only.

To achieve better performance, the attribute can be configured to cache the displayed data records.

Examples

The example below shows the simplest `PXSelector` attribute declaration. All `Category` data records will be available for selection. Their `CategoryCD` field values will be inserted without conversion.

```
[PXSelector(typeof(Category.categoryCD))]
public virtual string CategoryCD { get; set; }
```

The attribute below configures the lookup control to let the user select from the `Customer` data records retrieved by the `Search` BQL query. The displayed columns are specified explicitly: `AccountCD` and `CompanyName`.

```
[PXSelector(
    typeof(Search<Customer.accountCD,
               Where<Customer.companyType, Equal<CompanyType.customer>>>),
    new Type[]
    {
        typeof(Customer.accountCD),
        typeof(Customer.companyName)
    })]
public virtual string AccountCD { get; set; }
```

The `Customer.accountCD` field data will be inserted as a value without conversion.

The attribute below let the user select from the `Branch` data records. The attribute displays the `Branch.BranchCD` field value in the user interface, but actually assigns the `Branch.BranchID` field value to the field.

```
[PXSelector(typeof(Branch.branchID),
            SubstituteKey = typeof(Branch.branchCD))]
public virtual int? BranchID { get; set; }
```

The example below shows the `PXSelector` attribute in combination with other attributes.

```
[PXDBString(10, IsUnicode = true, InputMask = ">aaaaaaaaaa")]
[PXUIField(DisplayName = "Class ID")]
[PXSelector(
    typeof(Search<CRLeadClass.cRLeadClassID,
               Where<CRLeadClass.isActive, Equal<True>>>),
    DescriptionField = typeof(CRLeadClass.description),
    CacheGlobal = true)]
public virtual string ClassID { get; set; }
```

Here, the `PXSelector` attribute configures a lookup field that will let a user select from the data set defined by the `Search` query. The lookup control will display descriptions the data records, taking them from `CRLeadClass.description` field. The attribute will cache records in memory to reduce the number of database calls.

PXSelector Attribute Constructors

The *PXSelector* attribute exposes the following constructors.

PXSelectorAttribute(Type)

Initializes a new instance that will use the specified BQL query to retrieve the data records to select from. The list of displayed columns is created automatically and consists of all columns from the referenced table with the `Visibility` property of the *PXUIField* attribute set to `PXUIVisibility.SelectorVisible`.

Syntax:

```
public PXSelectorAttribute(Type type)
```

Parameters:

- `type`

  A BQL query that defines the data set that is shown to the user along with the key field that is used as a value. Set to a field (type part of a DAC field) to select all data records from the referenced table. Set to a BQL command of `Search` type to specify a complex select statement.

PXSelectorAttribute(Type, params Type[])

Initializes a new instance that will use the specified BQL query to retrieve the data records to select from, and display the provided set of columns.

Syntax:

```
public PXSelectorAttribute(Type type, params Type[] fieldList) : this(type)
```

Parameters:

- `type`

  A BQL query that defines the data set that is shown to the user along with the key field that is used as a value. Set to a field (type part of a DAC field) to select all data records from the referenced table. Set to a BQL command of `Search` type to specify a complex select statement.

- `fieldList`

  Fields to display in the control.

PXSelector Attribute Methods

The *PXSelector* attribute exposes the following static methods.

ClearGlobalCache(Type)

Clears the internal cache of the `PXSelector` attribute, removing the data records retrieved from the specified table. Typically, you don't need to call this method, because the attribute subscribes on the change notifications related to the table and drops the cache automatically.

Syntax:

```
public static void ClearGlobalCache(Type table)
```

Parameters:

- `table`

  The DAC to drop from the attribute's cache.

ClearGlobalCache<Table>()

Clears the internal cache of the `PXSelector` attribute, removing the data records retrieved from the specified table. Typically, you don't need to call this method, because the attribute subscribes on the change notifications related to the table and drops the cache automatically.

Syntax:

```
public static void ClearGlobalCache<Table>() where Table : IBqlTable
```

Type Parameters:

- `Table`

  The DAC to drop from the attribute's cache.

GetField(PXCache, object, string, object, string)

Returns a value of the field from a foreign data record.

Syntax:

```
public static object GetField(PXCache cache, object data, string field, object
 value,
           string foreignField)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `data`

  The data record that contains a reference to the foreign data record.
- `field`

  The name of the field holding the referenced data record key value.
- `value`

  The key value of the referenced data record.
- `foreignField`

  The name of the referenced data record field whose value is returned by the method.

GetItem(PXCache, PXSelectorAttribute, object, object)

Returns the foreign data record by the specified key.

Syntax:

```
public static object GetItem(PXCache cache, PXSelectorAttribute attr, object
           data, object key)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `attr`

  The instance of the `PXSelector` attribute to query for a data record.
- `data`

  The data record that contains a reference to the foreign data record.
- `key`

  The key value of the referenced data record.

GetItemType(PXCache, string)

Returns the data access class referenced by the attribute instance that marks the field with specified name.

Syntax:

```
public static Type GetItemType(PXCache cache, string field)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `field`

  The name of the field that marked with the attribute.

GetSelectorFields(Type)

Syntax:

```
public static List<KeyValuePair<string, Type>> GetSelectorFields(Type table)
```

Select(PXCache, object, string)

Returns the data record referenced by the attribute instance that marks the field with the specified name in a particular data record.

Syntax:

```
public static object Select(PXCache cache, object data, string field)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `data`

  The data record the method is applied to.

- `field`

  The name of the field that is be marked with the attribute.

Select(PXCache, object, string, object)

Returns the referenced data record that holds the specified value. The data record should be referenced by the attribute instance that marks the field with the specified in a particular data record.

Syntax:

```
public static object Select(PXCache cache, object data, string field, object
            value)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `data`

  The data record the method is applied to.

- `field`

  The name of the field that is be marked with the attribute.

- `value`

  The value to search the referenced table for.

Returns:

Foreign record.

Select<Field>(PXCache, object)

Returns the data record referenced by the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static object Select<Field>(PXCache cache, object data) where Field :
            IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `data`

  The data record the method is applied to.

Select<Field>(PXCache, object, object)

Returns the referenced data record that holds the specified value. The data record is searched among the ones referenced by the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static object Select<Field>(PXCache cache, object data, object value)
            where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `data`

  The data record the method is applied to.

- `value`

  The value to search the referenced table for.

SelectAll(PXCache, string, object)

Returns all data records kept by the attribute instance the marks the field with the specified name in a particular data record.

Syntax:

```
public static List<object> SelectAll(PXCache cache, string fieldname, object
            data)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `fieldname`

  The name of the field that should be marked with the attribute.

- `data`

  The data record the method is applied to.

SelectAll<Field>(PXCache, object)

Returns all data records kept by the attribute instance the marks the specified field in a particular data record.

Syntax:

```
public static List<object> SelectAll<Field>(PXCache cache, object data)
            where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `data`

  The data record the method is applied to.

SelectFirst(PXCache, object, string)

Returns the first data record retrieved by the attribute instance that marks the field with the specified name in a particular data record.

Syntax:

```
public static object SelectFirst(PXCache cache, object data, string field)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `data`

  The data record the method is applied to.
- `field`

  The name of the field that is be marked with the attribute.

SelectFirst<Field>(PXCache, object)

Returns the first data record retrieved by the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static object SelectFirst<Field>(PXCache cache, object data) where Field :
            IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `data`

  The data record the method is applied to.

SelectLast(PXCache, object, string)

Returns the last data record retrieved by the attribute instance that marks the field with the specified name in a particular data record.

Syntax:

```
public static object SelectLast(PXCache cache, object data, string field)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `data`

  The data record the method is applied to.

- `field`

  The name of the field that is be marked with the attribute.

SelectLast<Field>(PXCache, object)

Returns the last data record retrieved by the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static object SelectLast<Field>(PXCache cache, object data) where Field :
            IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `data`

  The data record the method is applied to.

SetColumns(PXCache, string, string[], string[])

Sets the list of columns and column headers for all attribute instances that mark the field with the specified name in all data records in the cache object.

Syntax:

```
public static void SetColumns(PXCache cache, string field, string[] fieldList,
 string[]
            headerList)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.

- `field`

  The name of the field marked with the attribute.

- `fieldList`

  The new list of field names.

- `headerList`

  The new list of column headers.

SetColumns(PXCache, object, string, string[], string[])

Sets the list of columns and column headers to display for the attribute instance that marks the field with the specified name in a particular data record.

Syntax:

```
public static void SetColumns(PXCache cache, object data, string field, string[]
            fieldList, string[] headerList)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records kept in the cache object.
- `field`

  The name of the field marked with the attribute.
- `fieldList`

  The new list of field names.
- `headerList`

  The new list of column headers.

SetColumns<Field>(PXCache, Type[], string[])

Sets the list of columns and column headers for all attribute instances that mark the specified field in all data records in the cache object.

Syntax:

```
public static void SetColumns<Field>(PXCache cache, Type[] fieldList, string[]
            headerList) where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `fieldList`

  The new list of field names.
- `headerList`

  The new list of column headers.

SetColumns<Field>(PXCache, object, Type[], string[])

Sets the list of columns and column headers to display for the attribute instance that marks the specified field in a particular data record.

Syntax:

```
public static void SetColumns<Field>(PXCache cache, object data, Type[] fieldList,
            string[] headerList) where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXSelector` type.
- `data`

  The data record the method is applied to.
- `fieldList`

  The new list of field names.
- `headerList`

  The new list of column headers.

StoreCached<Field>(PXCache, object, object)

Syntax:

```
public static void StoreCached<Field>(PXCache cache, object data, object item) where
 Field : IBqlField
```

PXCustomSelector Attribute

The base class for custom selector attributes. Derive the attribute class from this class and implement the `GetRecords()` method.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXSelectorAttribute
```

Syntax

```
public class PXCustomSelectorAttribute : PXSelectorAttribute
```

Constructors

Constructor	Description
PXCustomSelectorAttribute(Type)	Initializes a new instance with the specified BQL query for selecting the data records to show to the user
PXCustomSelectorAttribute(Type, params Type[])	Initializes a new instance that will use the specified BQL query to retrieve the data records to select from, and display the provided set of columns

PXCustomSelector Attribute Constructors

The *PXCustomSelector* attribute exposes the following constructors.

PXCustomSelectorAttribute(Type)

Initializes a new instance with the specified BQL query for selecting the data records to show to the user.

Syntax:

```
public PXCustomSelectorAttribute(Type type) : base(type)
```

Parameters:

- `type`

  A BQL query that defines the data set that is shown to the user along with the key field that is used as a value. Set to a field (type part of a DAC field) to select all data records from the referenced table. Set to a BQL command of `Search` type to specify a complex select statement.

PXCustomSelectorAttribute(Type, params Type[])

Initializes a new instance that will use the specified BQL query to retrieve the data records to select from, and display the provided set of columns.

Syntax:

```
public PXCustomSelectorAttribute(Type type, params Type[] fieldList) : base(type,
            fieldList)
```

Parameters:

- `type`

  A BQL query that defines the data set that is shown to the user along with the key field that is used as a value. Set to a field (type part of a DAC field) to select all data records from the referenced table. Set to a BQL command of `Search` type to specify a complex select statement.

- `fieldList`

  Fields to display in the control.

PXDimension Attribute

Sets up the input control for a DAC field that holds a segmented value. The control formats the input as a segmented key value and displays the list of allowed values for each key segment when the user presses F3 on a keyboard.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`
- `IPXFieldVerifyingSubscriber`
- `IPXFieldDefaultingSubscriber`
- `IPXRowPersistingSubscriber`
- `IPXRowPersistedSubscriber`
- `IPXFieldUpdatingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Parameter |
                AttributeTargets.Method)]
[Serializable]
public class PXDimensionAttribute : PXEventSubscriberAttribute,
                                    IPXFieldSelectingSubscriber,
                                    IPXFieldVerifyingSubscriber,
                                    IPXFieldDefaultingSubscriber,
                                    IPXRowPersistingSubscriber,
                                    IPXRowPersistedSubscriber,
```

```
                                       IPXFieldUpdatingSubscriber
```

Properties

- public virtual bool **ValidComboRequired**

  Gets or sets the value that indicates whether the user can specify only one of the predefined values as a segment or the user can input arbitrary values.

- public virtual string **Wildcard**

  Gets or sets the one character long string that will be treated as a wildcard – a character that matches any symbols. Typically, the property is set when the field to which the attribute is attached is used for filtering. See also the *PXDimensionWildcard* attribute.

Constructors

- public PXDimensionAttribute(string dimension) : base()

  Creates an instance to work with the provided segmented key.

  Parameters:

  - dimension

    The string identifier of the segmented key.

Static methods

- public static string[] GetSegmentValues(string dimensionid, int segmentnumber)

- public static void Clear()

Examples

```
[PXDimension("SUBACCOUNT", ValidComboRequired = false)]
public virtual string SubID { get; set; }
```

PXDimensionSelector Attribute

Sets up the lookup control for a DAC field that holds a segmented key value or references a data record identified by a segmented key. The attribute combines the `PXDimension` and `PXSelector` attributes.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXAggregateAttribute
```

Interfaces

- IPXFieldVerifyingSubscriber
- IPXRowPersistingSubscriber
- IPXRowPersistedSubscriber

Syntax

```
[PXAttributeFamily( typeof(PXSelectorAttribute))]
public class PXDimensionSelectorAttribute : PXAggregateAttribute,
```

```
IPXFieldVerifyingSubscriber,
IPXRowPersistingSubscriber,
IPXRowPersistedSubscriber
```

Properties

- `public virtual Type` **`DescriptionField`**

  Gets or sets the field from the referenced table that contains the description.

- `public virtual bool` **`CacheGlobal`**

  Gets or sets the value that indicates whether the attribute should cache the data records retrieved from the database to show in the lookup control. By default, the attribute does not cache the data records.

- `public virtual bool` **`Filterable`**

  Gets or sets the value that indicates whether the filters defined by the user should be stored in the database.

- `public virtual bool` **`DirtyRead`**

  Gets or sets a value that indicates whether the attribute should take into account the unsaved modifications when displaying data records in control. If `false`, the data records are taken from the database and not merged with the cache object. If `true`, the data records are merged with the modification stored in the cache object.

- `public virtual Type` **`Field`**

  Gets the field that identifies a referenced data record (surrogate key) and is assigned to the field annotated with the `PXSelector` attribute. Typically, it is the first parameter of the BQL query passed to the attribute constructor.

- `public virtual string[]` **`Headers`**

  Gets or sets the list of labels for column headers that are displayed in the lookup control. By default, the attribute uses display names of the fields.

- `public virtual bool` **`ValidComboRequired`**

  Gets or sets the value that indicates whether only the values from the combobox are allowed in segments.

Constructors

Constructor	Description
PXDimensionSelectorAttribute(string, Type)	Initializes a new instance to reference the data records that are identified by the specified segmented key
PXDimensionSelectorAttribute(string, Type, Type)	Initializes a new instance to reference the data records that are identified by the specified segmented key
PXDimensionSelectorAttribute(string, Type, Type,)	Initializes a new instance to reference the data records that are identified by the specified segmented key

Static Methods

Method	Description
SetValidCombo(PXCache, string, bool)	
SetValidCombo<Field>(PXCache, bool)	

Method	Description
SuppressViewCreation(PXCache)	

Examples

The attribute below sets up the control for input of the *BIZACCT* segmented key's values. Since the `AcctCD` field itself is specified as the substitute key it will keep the segmented key value.

```
[PXDimensionSelector(
    "BIZACCT",
    typeof(BAccount.acctCD),    // BQL query for lookup
    typeof(BAccount.acctCD))]  // Substitute key
public virtual string AcctCD { get; set; }
```

In the following example the `RunRateItemID` field references the data records from

```
[PXDimensionSelector(
    InventoryAttribute.DimensionName,
    typeof(
        Search<InventoryItem.inventoryID,
            Where<InventoryItem.itemType, Equal<INItemTypes.nonStockItem>,
                And<Match<Current<AccessInfo.userName>>>>>),
    typeof(InventoryItem.inventoryCD),
    DescriptionField = typeof(InventoryItem.descr))]
public virtual int? RunRateItemID { get; set; }
```

Related Types

- *PXSelector Attribute*
- *PXDimension Attribute*

PXDimensionSelector Attribute Constructors

The *PXDimensionSelector* attribute exposes the following constructors.

PXDimensionSelectorAttribute(string, Type)

Initializes a new instance to reference the data records that are identified by the specified segmented key. Uses the provided BQL query to retrieve the data records.

Syntax:

```
public PXDimensionSelectorAttribute(string dimension, Type type) : base()
```

Parameters:

- `dimension`

  The string identifier of the segmented key.

- `type`

  A BQL query that defines the data set that is shown to the user along with the key field that is used as a value. Set to a field (type part of a DAC field) to select all data records from the referenced table. Set to a BQL command of `Search` type to specify a complex select statement.

PXDimensionSelectorAttribute(string, Type, Type)

Initializes a new instance to reference the data records that are identified by the specified segmented key. Uses the provided BQL query to retrieve the data records and substitutes the field value (surrogate key) with the provided field (natural key).

Syntax:

```
public PXDimensionSelectorAttribute(string dimension, Type type, Type
 substituteKey) :
            base()
```

Parameters:

- `dimension`

  The string identifier of the segmented key.
- `type`

  A BQL query that defines the data set that is shown to the user along with the key field that is used as a value. Set to a field (type part of a DAC field) to select all data records from the referenced table. Set to a BQL command of `Search` type to specify a complex select statement.
- `substituteKey`

  The field to sustitute the surrogate field's value in the user interface.

PXDimensionSelectorAttribute(string, Type, Type, params Type[])

Initializes a new instance to reference the data records that are identified by the specified segmented key. Uses the provided BQL query to retrieve the data records and substitutes the field value (surrogate key) with the provided field (natural key).

Syntax:

```
public PXDimensionSelectorAttribute(string dimension, Type type, Type substituteKey,
            params Type[] fieldList) : base()
```

Parameters:

- `dimension`

  The string identifier of the segmented key.
- `type`

  A BQL query that defines the data set that is shown to the user along with the key field that is used as a value. Set to a field (type part of a DAC field) to select all data records from the referenced table. Set to a BQL command of `Search` type to specify a complex select statement.
- `substituteKey`

  The field to sustitute the surrogate field's value in the user interface.
- `fieldList`

  Fields to display in the control.

PXDimensionSelector Attribute Methods

The *PXDimensionSelector* attribute exposes the following static methods.

SetValidCombo(PXCache, string, bool)

Syntax:

```
public static void SetValidCombo(PXCache cache, string name, bool isRequired)
```

SetValidCombo<Field>(PXCache, bool)

Syntax:

```
public static void SetValidCombo<Field>(PXCache cache, bool isRequired) where
 Field : IBqlField
```

SuppressViewCreation(PXCache)

Syntax:

```
public static void SuppressViewCreation(PXCache cache)
```

PXDimensionWildcard Attribute

Sets up the lookup control for a DAC field that holds a segmented key value and allows the *?* wildcard character. The attribute combines the `PXDimension` and `PXSelector` attributes.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXAggregateAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`

Syntax

```
public class PXDimensionWildcardAttribute : PXAggregateAttribute,
                                            IPXFieldSelectingSubscriber
```

Properties

- `public virtual Type` **`DescriptionField`**

  Gets or sets the field from the referenced table that contains the description.

- `public virtual string` **`Wildcard`**

  Gets or sets the wildcard string that matches any symbol in the segment.

- `public virtual string[]` **`Headers`**

  Gets or sets the list of labels for column headers that are displayed in the lookup control. By default, the attribute uses display names of the fields.

Constructors

Constructor	Description
PXDimensionWildcardAttribute(string, Type)	Creates a selector
PXDimensionWildcardAttribute(string, Type, params Type[])	Creates a selector overriding the columns

PXDimensionWildcard Attribute Constructors

The *PXDimensionWildcard* attribute exposes the following constructors.

PXDimensionWildcardAttribute(string, Type)

Creates a selector.

Syntax:

```
public PXDimensionWildcardAttribute(string dimension, Type type) : base()
```

Parameters:

- `type`

  Referenced table. Should be either IBqlField or IBqlSearch.

PXDimensionWildcardAttribute(string, Type, params Type[])

Creates a selector overriding the columns.

Syntax:

```
public PXDimensionWildcardAttribute(string dimension, Type type, params Type[]
 fieldList) : base()
```

Parameters:

- `type`

  Referenced table. Should be either IBqlField or IBqlSearch.

- `fieldList`

  Fields to display in the selector.

- `headerList`

  Headers of the selector columns.

Referential Integrity and Calculations

The following attributes implement referential integrity and perform calculations over related data at run time:

- *PXParent*

  Creates a reference to a parent data record. When the parent data record is deleted all child data records that reference it are also deleted.

- *PXFormula*

  Calculates a field from other fields of the same data record or sets an aggregation expression to calculate a parent data record field from child data record fields. Calculations happen at run time.

- *PXUnboundFormula*

  Calculates the value from the child data record fields and aggregates all such values computed for the child data records into the parent data record field. Calculations happen at run time.

- *PXDBChildIdentity*

  Indicates that a DAC field references an auto-generated key field from another table and ensures the field value is correct after changes are committed to the database.

- *PXLineNbr*

  Generates unique line numbers that identify child data records in the parent-child relationship.

Note that all the attributes in the list above add run time server-side logic. The referential integrity is implemented on the server side. And the calculations are implemented on the server side. See the *Adhoc SQL for Fields* section for the attributes that enable calculation of fields on the database side.

PXParent Attribute

Creates a reference to the parent record, establishing a parent-child relationship between two tables.

See *Remarks* for more details. See *Examples* for examples of usage.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
                AttributeTargets.Class, AllowMultiple = true)]
public class PXParentAttribute : PXEventSubscriberAttribute
```

Properties

- `public virtual bool` **`ParentCreate`**

  Gets or sets the value that permits or forbids creation of the parent through the *CreateParent(PXCache, object, Type)* method. In particular, the `PXFormula` attribute tries to create a parent data record if it doesn't exist, by invoking this method. By default, the property equals `false`.

- `public virtual bool` **`LeaveChildren`**

  Gets or sets the value that indicates whether the child data records are left or deleted on parent data record deletion. By default, the property equals `false`, which means that child data records are deleted.

- `public virtual Type` **`ParentType`**

  Gets the DAC type of the parent data record. The type is determined in the constructor as the first table referenced in the `Select` query.

- `public virtual bool` **`UseCurrent`**

  Gets or sets the value that indicates at run time whether to take the parent data record from the `Current` property or retrieve it from the database. In both cases the attribute uses the view corresponding to the `Select` query provided in the constructor.

Constructors

- `public PXParentAttribute(Type selectParent)`

  Initializes a new instance that defines the parent data record using the provided BQL query. To provide parameters to the BQL query, use `Current` to pass the values from the child data record that is `Current` for the cache object.

  Parameters:

  - `selectParent`

    The BQL query that selects the parent record. Should be based on a class derived from `IBqlSelect`, such as `Select<>`.

Static Methods

Method	Description
CopyParent(PXCache, object, object, Type)	Makes the parent of the provided data record be the parent of the other provided data record
CreateParent(PXCache, object, Type)	Creates the parent for the provided child data record for the attribute instance that references the provided parent type or a type derived from it
GetParentCreate(PXCache, Type)	Returns the value of the `ParentCreate` property from the attribute instance that references the provided parent type or a type derived from it
GetParentType(PXCache)	Returns the parent type of the first attribute instance found in the cache object
SelectParent(PXCache, object)	Returns the parent data record of the provided child data record
SelectParent(PXCache, object, Type)	Returns the parent data record of the provided child data record
SelectSiblings(PXCache, object)	Returns the child data records that have the same parent as the provided child data record
SelectSiblings(PXCache, object, Type)	Returns the child data records that have the same parent as the provided child data record
SetLeaveChildren<Field>(PXCache, object, bool)	Enables or disables cascade deletion of child data records for the attribute instance in a paricular data record
SetParent(PXCache, object, Type, object)	Sets the provided data record of parent type as the parent of the child data record

Remarks

You can place the attribute on any field of the child DAC. The primary goal of the attribute is to perform cascade deletion of the child data records once a parent data record is deleted.

The parent data record is defined by a BQL query of `Select<>` type. Typically, the query includes a `Where` clause that adds conditions for the parent's key fields to equal child's key fields. In this case, the values of child data record key fields are specified using the `Current` parameter. The business logic controller that provides the interface for working with these parent and child data records should define a view selecting parent data records and a view selecting child data records. These views will againg be connected using the `Current` parameter.

You can use the static methods to retrieve a particular parent data record or child data records, or get and set some attribute parameters.

Once the `PXParent` attribute is added to some DAC field, you can use the *PXFormula* attribute to define set calculations for parent data record fields from child data record fields.

Examples

The attribute below specifies a query for selecting the parent `Document` data record for a given child `DocTransaction` data record.

```
[PXParent(typeof(
    Select<Document,
        Where<Document.docNbr, Equal<Current<DocTransaction.docNbr>>,
```

```
            And<Document.docType, Equal<Current<DocTransaction.docType>>>>>))]
public virtual string DocNbr { get; set; }
```

Another example is given below.

```
[PXParent(typeof(
    Select<ARTran,
        Where<ARTran.tranType, Equal<Current<ARFinChargeTran.tranType>>,
            And<ARTran.refNbr, Equal<Current<ARFinChargeTran.refNbr>>,
            And<ARTran.lineNbr, Equal<Current<ARFinChargeTran.lineNbr>>>>>>))]
public virtual short? LineNbr { get; set; }
```

Obtaining the parent data record at run time:

```
CR.Location child = (CR.Location)e.Row;
BAccount parent =
    (BAccount)PXParentAttribute.SelectParent(sender, child, typeof(BAccount));
```

Setting the parent data record at run time:

```
// Views definitions in a graph
public PXSelect<INRegister> inregister;
public PXSelect<INTran> intranselect;
...
// Code executed in some graph method
INTran tran = (INTran)res;
PXParentAttribute.SetParent(
    intranselect.Cache, tran, typeof(INRegister), inregister.Current);
```

PXParent Attribute Methods

The *PXParent* attribute exposes the following static methods.

CopyParent(PXCache, object, object, Type)

Makes the parent of the provided data record be the parent of the other provided data record. Uses the first attribute instance that references the provided parent type or a type derived from it.

Syntax:

```
public static void CopyParent(PXCache cache, object item, object copy, Type
            ParentType)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.
- `item`

  The child data record whose parent data record is made the parent of another data record.
- `copy`

  The data record that becomes the child of the provided data record's parent.
- `ParentType`

  The DAC type of the parent data record.

CreateParent(PXCache, object, Type)

Creates the parent for the provided child data record for the attribute instance that references the provided parent type or a type derived from it. Does nothing if `ParentCreate` equals `false` in this attribute instance. If the parent is created, it is inserted into the cache object.

Syntax:

```
public static void CreateParent(PXCache cache, object row, Type
            ParentType)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.
- `row`

  The child data record for which the parent is created.
- `ParentType`

  The DAC type of the parent data record.

GetParentCreate(PXCache, Type)

Returns the value of the `ParentCreate` property from the attribute instance that references the provided parent type or a type derived from it.

Syntax:

```
public static bool GetParentCreate(PXCache cache, Type ParentType)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.
- `ParentType`

  The DAC type of the parent data record.

GetParentType(PXCache)

Returns the parent type of the first attribute instance found in the cache object.

Syntax:

```
public static Type GetParentType(PXCache cache)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.

SelectParent(PXCache, object)

Returns the parent data record of the provided child data record. Uses the first attribute instance found in the cache object.

Syntax:

```
public static object SelectParent(PXCache cache, object row)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.

- `row`

  The child data record whose parent data record is retireved.

SelectParent(PXCache, object, Type)

Returns the parent data record of the provided child data record. Uses the first attribute instance that references the provided parent type or a type derived from it.

Syntax:

```
public static object SelectParent(PXCache cache, object row, Type ParentType)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.

- `row`

  The child data record whose parent data record is retireved.

- `ParentType`

  The DAC type of the parent data record.

SelectSiblings(PXCache, object)

Returns the child data records that have the same parent as the provided child data record. Returns an array of zero length if fails to retrieve the parent. Uses the first attribute instance found in the cache object.

Syntax:

```
public static object[] SelectSiblings(PXCache cache, object row)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.

- `row`

  The child data record for which the data records having the same parent are retrieved.

SelectSiblings(PXCache, object, Type)

Returns the child data records that have the same parent as the provided child data record. Returns an array of zero length if fails to retrieve the parent. Uses the first attribute instance that references the provided parent type or a type derived from it.

Syntax:

```
public static object[] SelectSiblings(PXCache cache, object row, Type
            ParentType)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.

- `row`

  The child data record for which the data records having the same parent are retrieved.

- `ParentType`

  The DAC type of the parent data record.

SetLeaveChildren<Field>(PXCache, object, bool)

Enables or disables cascade deletion of child data records for the attribute instance in a paricular data record.

Syntax:

```
public static void SetLeaveChildren<Field>(PXCache cache, object data, bool
            isLeaveChildren) where Field : IBqlField
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.

- `data`

  The data record the method is applied to. If `null`, the method is applied to all data records in the cache object.

- `isLeaveChildren`

  The new value for the `LeaveChildren` property. If `true`, enables cascade deletion. Otherwise, disables it.

SetParent(PXCache, object, Type, object)

Sets the provided data record of parent type as the parent of the child data record.

Syntax:

```
public static void SetParent(PXCache cache, object row, Type ParentType, object
            parent)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXParent` type.

- `row`

  The child data record for which the parent data record is set. Must not be `null`.

- `ParentType`

  The DAC type of the parent data record.

- `parent`

  The new parent data record.

PXFormula Attribute

Calculates a field from other fields of the same data record and sets an aggregation formula to calculate a parent data record field from child data record fields.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowUpdatedSubscriber`
- `IPXRowInsertedSubscriber`
- `IPXRowDeletedSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class, AllowMultiple = true)]
public class PXFormulaAttribute : PXEventSubscriberAttribute,
                                   IPXRowUpdatedSubscriber,
                                   IPXRowInsertedSubscriber,
                                   IPXRowDeletedSubscriber
```

Properties

- `public virtual string` **`FormulaFieldName`**

  Get the name of the field the attribute is attached to.

- `public virtual Type` **`Formula`**

  Gets or sets the BQL query that is used to calculate the value of the field the attribute is attached to. The value should derive from `Constant<>`, `IBqlField`, or `IBqlCreator`.

- `public virtual Type` **`ParentField`**

  Gets or sets the parent data record field the aggregation result is assigned to. The value should derive from `IBqlField`.

- `public virtual Type` **`Aggregate`**

  Gets or sets the BQL query that represents the aggregation formula used to calculate the parent data record field from the child data records fields. The value should derive from `IBqlAggregateCalculator`.

- `public virtual bool` **`Persistent`**

  Gets or sets the value that indicates whether the attribute recalculates the formula for the child field after a saving of changes to the database. You may need recalculation if the fields the formula depends on are updated on the `RowPersisting` event. By default, the property equals `false`.

Constructors

Constructor	Description
PXFormulaAttribute(Type)	Initializes a new instance that calculates the value of the field the attribute is atached to, by the provided formula
PXFormulaAttribute(Type, Type)	Initializes a new instance that calculates the value of the field the attribute is atached to and sets an aggregate function to calculate the value of a field in the parent data record

Static Methods

Method	Description
CalcAggregate<Field>(PXCache, object)	Calculates the fields of the parent data record using the aggregation formula from the attribute instance that marks the specified field
CalcAggregate<Field>(PXCache, object, bool)	Calculates the fields of the parent data record using the aggregation formula from the attribute instance that marks the specified field
SetAggregate<Field>(PXCache, Type)	Sets the new aggregation formula in the attribute instances that mark the specified field, for all data records in the cache object

Remarks

The attribute assigns the computed value to the field the attribute is attached to. The value is also used for aggregated calculation of the parent data record field (if the aggregate expression has been specified in the attribute parameter).

The `PXParent` attribute must be added to some field of the child DAC.

Examples

The attribute below sums two fields and assigns it the field the attribute is attached to.

```
[PXFormula(typeof(
    Add<SOOrder.curyPremiumFreightAmt, SOOrder.curyFreightAmt>))]
public virtual Decimal? CuryFreightTot { get; set; }
```

The attribute below performs more complex calculation.

```
[PXFormula(typeof(
    Switch<
        Case<Where<Add<SOOrder.releasedCntr, SOOrder.billedCntr>,
                   Equal<short0>>,
            SOOrder.curyOrderTotal>,
        Add<SOOrder.curyUnbilledOrderTotal, SOOrder.curyFreightTot>>))]
public decimal? CuryDocBal { get; set; }
```

The attribute below multiplies the `TranQty` and `UnitPrice` fields and assigns the result to the `ExtPrice` field. The attribute also calculates the sum of the computed `ExtPrice` values over all child `DocTransaction` data records and assigns the result to the parent's `TotalAmt` field.

```
[PXUIField(DisplayName = "Line Total", Enabled = false)]
[PXFormula(
    typeof(Mult<DocTransaction.tranQty, DocTransaction.unitPrice>),
    typeof(SumCalc<Document.totalAmt>))]
public virtual decimal? ExtPrice { get; set; }
```

A common practice is to disable the input control for a calculated field. In the example above, the control is disabled using the `PXUIField` attribute.

The attribute below does not provide a formula for calculating the `TranQty` property. The value inputted by a user is assigned to the field. The attribute only sets the formula to calculate the `TotalQty` field in the parent data record as the sum of `TranQty` values over all related child data records.

```
[PXFormula(null, typeof(SumCalc<Document.totalQty>))]
public virtual decimal? TranQty { get; set; }
```

PXFormula Attribute Constructors

The *PXFormula* attribute exposes the following constructors.

PXFormulaAttribute(Type)

Initializes a new instance that calculates the value of the field the attribute is atached to, by the provided formula.

Syntax:

```
public PXFormulaAttribute(Type formulaType)
```

Parameters:

- `formulaType`

  The formula to calculate the field value from other fields of the same data record. The formula can be an expression built from BQL functions such as `Add`, `Sub`, `Mult`, `Div`, `Switch` and *other functions*.

PXFormulaAttribute(Type, Type)

Initializes a new instance that calculates the value of the field the attribute is atached to and sets an aggregate function to calculate the value of a field in the parent data record. The aggregation function is applied to the values calculated by the first parameter for all child data records.

Syntax:

```
public PXFormulaAttribute(Type formulaType, Type aggregateType)
```

Parameters:

- `formulaType`

  The formula to calculate the field value from other fields of the same data record. The formula can be an expression built from BQL functions such as `Add`, `Sub`, `Mult`, `Div`, `Switch` and *other functions*. If `null`, the aggregation function takes into account the field value inputted by the user.

- `aggregateType`

  The aggregation formula to calculate the parent data record field from the child data records fields. Use an *aggregation function* such as `SumCalc`, `CountCalc`, `MinCalc`, and `MaxCalc`.

PXFormula Attribute Methods

The *PXFormula* attribute exposes the following static methods.

CalcAggregate<Field>(PXCache, object)

Calculates the fields of the parent data record using the aggregation formula from the attribute instance that marks the specified field. The calculation is applied to the child data records merged with the modifications kept in the session.

Syntax:

```
public static void CalcAggregate<Field>(PXCache sender, object parent) where Field :
            IBqlField
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXFormula` type.

- `parent`

The parent data record.

CalcAggregate<Field>(PXCache, object, bool)

Calculates the fields of the parent data record using the aggregation formula from the attribute instance that marks the specified field. The calculation is applied to the child data records that are either taken directly from the database or merged with the modifications kept in the session.

Syntax:

```
public static void CalcAggregate<Field>(PXCache sender, object parent, bool
            IsReadOnly) where Field : IBqlField
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXFormula` type.
- `parent`

  The parent data record.
- `IsReadOnly`

  If `true`, the child data records are not merged with the unsaved modification accessible through the cache object. Otherwise, the child data records are merged with the modifications.

SetAggregate<Field>(PXCache, Type)

Sets the new aggregation formula in the attribute instances that mark the specified field, for all data records in the cache object.

Syntax:

```
public static void SetAggregate<Field>(PXCache sender, Type aggregateType) where
            Field : IBqlField
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXFormula` type.
- `aggregateType`

  The new aggregation formula that will be used to calculate the parent data record field from the child data records fields.

Formulas

The classes described below are used as aggregation formulas in the *PXFormula* or *PXUnboundFormula* attribute to compute the parent data record field from the child data records fields. The expression that is calculated for each child data record is set in the first constructor parameters in the attributes.

SumCalc<Field> : IBqlAggregateCalculator, IBqlUnboundAggregateCalculator

Calculates the aggregated sum of expressions over all child data records and assings it to the specified parent data record field. The *PXUnboundFormula* attribute also supports this aggregation function.

Type Parameters:

- `Field : IBqlField`

Examples:

```
[PXFormula(typeof(Mult<INTran.qty, INTran.unitPrice>),
           typeof(SumCalc<INRegister.totalAmount>))]
public virtual Decimal? TranAmt { get; set; }
```

CountCalc<Field> : IBqlAggregateCalculator, ICountCalc

Calculates the number of the child data records and assigns it to the specified parent data record field.

Type Parameters:

- `Field : IBqlField`

Examples:

```
[PXFormula(null, typeof(CountCalc<ARSalesPerTran.refCntr>))]
public virtual Decimal? CuryTranAmt { get; set; }
```

MinCalc<Field> : IBqlAggregateCalculator

Calculates the minimum expression over all child data records and assigns it to the specified parent data record field.

Type Parameters:

- `Field : IBqlField`

MaxCalc<Field> : IBqlAggregateCalculator

Calculates the maximum expression over all child data records and assigns it to the specified parent data record field.

Type Parameters:

- `Field : IBqlField`

Examples:

```
[PXFormula(null,typeof(MaxCalc<CABankStatement.tranMaxDate>))]
public virtual DateTime? TranDate { get; set; }
```

Functions Used in Formulas

To define a formula for the *PXFormula* attribute to calculate a DAC field, you can use the following BQL functions:

- *Arithmetic operations*
- *Switch expression*
- The functions represented by the classes listed below

Row<Field, DependentField> : IBqlOperand, IBqlCreator

Returns the value of the specified field and creates an additional dependency for the formula – on the provided dependency field. Each time the dependency field is updated, the formula is recalculated. The formula also depends on all other field referenced in the formula.

Type Parameters:

- `Field : IBqlField`
- `DependentField : IBqlField`

Examples:

```
[PXFormula(
    typeof(Mult<Row<POLine.baseOrderQty, POLine.orderQty>, POLine.unitWeight>),
    typeof(SumCalc<POOrder.orderWeight>))]
public virtual Decimal? ExtWeight { get; set; }
```

Parent<Field> : IBqlCreator, IBqlOperand

Returns the value of the specified field from the parent data record. The parent data record is defined by the *PXParent* attribute.

Type Parameters:

- `Field : IBqlOperand`

Examples:

```
[PXUnboundFormula(
    typeof(Switch<
        Case<Where<SOLine.operation, Equal<Parent<SOOrder.defaultOperation>>,
                 And<SOLine.lineType, NotEqual<SOLineType.miscCharge>>>,
             SOLine.orderQty>,
        decimal0>),
    typeof(SumCalc<SOOrder.orderQty>))]
public virtual decimal? OrderQty { get; set; }
```

Selector<KeyField, ForeignOperand> : IBqlCreator, IBqlOperand

Searches for the `PXSelector` attribute on the key field and calculates the provided expression for the data record currently referenced by `PXSelector`.

Type Parameters:

- `KeyField : IBqlOperand`

  The key field to which the `PXSelector` attribute should be attached.

- `ForeignOperand : IBqlOperand`

  The expression that is calculated for the data record currently referenced by `PXSelector`.

Examples:

```
[PXFormula(typeof(
    Selector<APPaymentChargeTran.entryTypeID,
             Selector<CAEntryType.accountID, Account.accountCD>>))]
public virtual int? AccountID { get; set; }
```

Validate<V1> : IBqlCreator, IBqlTrigger

Raises the `FieldVerifying` event for the field to which the `PXFormula` attribute is attached once the specified field changes.

Validate<V1,V2> : IBqlCreator, IBqlTrigger

Raises the `FieldVerifying` event for the field to which the `PXFormula` attribute is attached once the specified fields change.

Examples:

```
[PXFormula(typeof(Validate<ContractItem.maxQty, ContractItem.minQty>))]
public decimal? DefaultQty { get; set; }
```

Validate<V1, V2, V3> : IBqlCreator, IBqlTrigger

Raises the `FieldVerifying` event for the field to which the `PXFormula` attribute is attached once the specified fields change.

Validate<V1, V2, V3, V4> : IBqlCreator, IBqlTrigger

Raises the `FieldVerifying` event for the field to which the `PXFormula` attribute is attached once the specified fields change.

Default<V1> : IBqlCreator, IBqlTrigger

Raises the `FieldDefaulting` event for the field to which the `PXFormula` attribute is attached once the specified field changes.

Type Parameters:

- `V1 : IBqlField`

Examples:

```
[PXFormula(typeof(Default<NotificationSource.setupID>))]
public virtual string Format { get; set; }
```

Default<V1, V2> : IBqlCreator, IBqlTrigger

Raises the `FieldDefaulting` event for the field to which the `PXFormula` attribute is attached once the specified fields change.

Type Parameters:

- `V1 : IBqlField`
- `V2 : IBqlField`

Default<V1, V2, V3> : IBqlCreator, IBqlTrigger

Raises the `FieldDefaulting` event for the field to which the `PXFormula` attribute is attached once the specified fields change.

Type Parameters:

- `V1 : IBqlField`
- `V2 : IBqlField`
- `V3 : IBqlField`

Default<V1, V2, V3, V4> : IBqlCreator, IBqlTrigger

Raises the `FieldDefaulting` event for the field to which the `PXFormula` attribute is attached once the specified fields change.

Type Parameters:

- `V1 : IBqlField`
- `V2 : IBqlField`
- `V3 : IBqlField`
- `V4 : IBqlField`

BqlFormula<Op1> : BqlFormula, IBqlCreator

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`

BqlFormula<Op1, Op2> : BqlFormula<Op1>

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`
- `Op2 : IBqlOperand`

BqlFormula<Op1, Op2, O3> : BqlFormula<Op1, Op2>

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`
- `Op2 : IBqlOperand`
- `Op3 : IBqlOperand`

BqlFormula<Op1, Op2, Op3, Op4> : BqlFormula<Op1, Op2, Op3>

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`
- `Op2 : IBqlOperand`
- `Op3 : IBqlOperand`
- `Op4 : IBqlOperand`

BqlFormula<Op1, Op2, Op3, Op4, Op5> : BqlFormula<Op1, Op2, Op3, Op4>

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`
- `Op2 : IBqlOperand`
- `Op3 : IBqlOperand`
- `Op4 : IBqlOperand`
- `Op5 : IBqlOperand`

BqlFormula<Op1, Op2, Op3, Op4, Op5, Op6> : BqlFormula<Op1, Op2, Op3, Op4, Op5>

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`
- `Op2 : IBqlOperand`
- `Op3 : IBqlOperand`
- `Op4 : IBqlOperand`

- `Op5 : IBqlOperand`
- `Op6 : IBqlOperand`

BqlFormula<Op1, Op2, Op3, Op4, Op5, Op6, Op7> : BqlFormula<Op1, Op2, Op3, Op4, Op5, Op6>

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`
- `Op2 : IBqlOperand`
- `Op3 : IBqlOperand`
- `Op4 : IBqlOperand`
- `Op5 : IBqlOperand`
- `Op6 : IBqlOperand`
- `Op7 : IBqlOperand`

BqlFormula<Op1, Op2, Op3, Op4, Op5, Op6, Op7, Op8> : BqlFormula<Op1, Op2, Op3, Op4, Op5, Op6, Op7>

An abstract class used to derive custom BQL functions.

Type Parameters:

- `Op1 : IBqlOperand`
- `Op2 : IBqlOperand`
- `Op3 : IBqlOperand`
- `Op4 : IBqlOperand`
- `Op5 : IBqlOperand`
- `Op6 : IBqlOperand`
- `Op7 : IBqlOperand`
- `Op8 : IBqlOperand`

PXUnboundFormula Attribute

Calculates the value from the child data record fields and computes the aggregation of such values over all child data records.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXFormulaAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class, AllowMultiple = true)]
public class PXUnboundFormulaAttribute : PXFormulaAttribute
```

Properties

- `public override string` **`FormulaFieldName`**

  Get the name of the field the attribute is attached to.

Constructors

- `public PXUnboundFormulaAttribute(Type formulaType, Type aggregateType) :base(formulaType, aggregateType)`

  Initializes a new instance that calculates the value of the field the attribute is atached to and sets an aggregate function to calculate the value of a field in the parent data record. The aggregation function is applied to the values calculated by the first parameter for all child data records.

  Parameters:

  - `formulaType`

    The formula to calculate the field value from other fields of the same data record. The formula can be an expression built from BQL functions such as `Add`, `Sub`, `Mult`, `Div`, `Switch` and *other functions*. If `null`, the aggregation function takes into account the field value inputted by the user.

  - `aggregateType`

    The aggregation formula to calculate the parent data record field from the child data records fields. Currenlty, only `SumCalc` is supported.

Remarks

Unlike the `PXFormula` attribute, this attribute does not assign the computed value to the field the attribute is attached to. The value is only used for aggregated calculation of the parent data record field. Hence, you can place this attribute on declaration of any child DAC field.

The `PXParent` attribute must be added to some field of the child DAC.

Examples

```
[PXUnboundFormula(
    typeof(Mult<APAdjust.adjgBalSign, APAdjust.curyAdjgAmt>),
    typeof(SumCalc<APPayment.curyApplAmt>))]
public virtual decimal? CuryAdjgAmt { get; set; }
```

Several `UnboundFormula` attributes can be placed on the same DAC field definition, as shown in the example below.

```
[PXUnboundFormula(
    typeof(Switch<
        Case<WhereExempt<APTaxTran.taxID>, APTaxTran.curyTaxableAmt>,
        decimal0>),
    typeof(SumCalc<APInvoice.curyVatExemptTotal>))]
[PXUnboundFormula(
    typeof(Switch<
        Case<WhereTaxable<APTaxTran.taxID>, APTaxTran.curyTaxableAmt>,
        decimal0>),
    typeof(SumCalc<APInvoice.curyVatTaxableTotal>))]
public override Decimal? CuryTaxableAmt { get; set; }
```

PXDBChildIdentity Attribute

Indicates that a DAC field references an auto-generated key field from another table and ensures the DAC field's value is correct after changes are committed to the database.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowPersistingSubscriber`
- `IPXRowPersistedSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBChildIdentityAttribute : PXEventSubscriberAttribute,
                                          IPXRowPersistingSubscriber,
                                          IPXRowPersistedSubscriber
```

Constructors

- `public PXDBChildIdentityAttribute(Type sourceType)`

  Initializes a new instance that takes the value for the field the attribute is attached to from the provided source field.

  Parameters:

  - `sourceType`

    The source field type to get the value from, should be nested (defined in a DAC) and implement `IBqlField`.

Remarks

The attribute updates the field value once the source field is assigned a real value by the database.

Examples

```
[PXDBInt()]
[PXDBChildIdentity(typeof(Address.addressID))]
public virtual int? DefPOAddressID { get; set; }
```

PXLineNbr Attribute

Automatically generates unique line numbers that identify for child data records in the parent-child relationship. This attribute does not work without the *PXParent* attribute.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldDefaultingSubscriber`
- `IPXRowDeletedSubscriber`
- `IPXRowInsertedSubscriber`

Syntax

```
public sealed class PXLineNbrAttribute : PXEventSubscriberAttribute,
                                         IPXFieldDefaultingSubscriber,
                                         IPXRowDeletedSubscriber,
                                         IPXRowInsertedSubscriber
```

Properties

- `public short` **`IncrementStep`**

  Gets or sets the number by which the line number is incremented or decremented. By default, the property equals 1.

Constructors

- `public PXLineNbrAttribute(Type sourceType)`

  Initializes a new instance of the attribute. As a parameter you can provide the parent data record field that stores the number of child data records or the parent DAC if there is no such field. In the latter case the attribute will calculate the number of child data records automatically.

  Parameters:

  - `sourceType`

    The parent data record field that stores the number of children or the parent DAC.

Static Methods

Method	Description
NewLineNbr<TField>(PXCache, object)	Returns the next line number for the provided parent data record

Remarks

The attribute should be placed on the child DAC field that stores the line number. The line number is a two-byte integer incremented by the `IncrementStep` property value, which equals 1 by default. The line number uniquely identifies a data record among the child data records related to a given parent data record. The attribute calculates each next value by incrementing the current number of the child data records.

The child DAC field to store the line number typically has the `short?` data type. It also should be a key field. You indicate that the field is a key field by setting the `IsKey` property of the data type attribute to `true`.

As a parameter, you should pass either the parent DAC field that stores the number of related child data records or the parent DAC itself. In the latter case, the attribute will determine the number of related child data records by itself. If the parent DAC field is specified, the attribute automatically updates its value.

Examples

The attribute below takes the number of related child data records from the provided parent field. The `PXParent` attribute must be added to some other field of this DAC.

```
[PXDBShort(IsKey = true)]
[PXLineNbr(typeof(ARRegister.lineCntr))]
public virtual short? LineNbr { get; set; }
```

In the following example, the attribute calculates the number of related child data records by itself.

```
[PXDBShort(IsKey = true)]
[PXLineNbr(typeof(Vendor))]
[PXParent(typeof(
    Select<Vendor,
        Where<Vendor.bAccountID, Equal<Current<TaxReportLine.vendorID>>>>))]
public virtual short? LineNbr { get; set; }
```

PXLineNbr Attribute Methods

The *PXLineNbr* attribute exposes the following static methods.

NewLineNbr<TField>(PXCache, object)

Returns the next line number for the provided parent data record. The returned value should be used as the child identifier stored in the specified field.

Syntax:

```
public static object NewLineNbr<TField>(PXCache cache, object Current) where
 TField :
            class, IBqlField
```

Parameters:

- `cache`

  The cache object to search for the

- `Current`

  The parent data record for which the next child identifier (line number) is returned.

Returns:

The line number as an object. Cast to `short?`.

Adhoc SQL for Fields

The following attributes set database-side calculation of DAC fields that are not bound to particular database columns:

- *PXDBCalced*

  Defines the SQL expression that calculates an unbound field from the fields of the same DAC whose values are taken from the database.

- *PXDBScalar*

  Defines the SQL subrequest that retrieves the value for an unbound DAC field. The subrequest can retrieve data from any bound field from any DAC.

The attributes will add the provided expression and the subrequest into the SQL query that selects data records of the given DAC.

PXDBCalced Attribute

Defines the SQL expression that calculates an unbound field from the fields of the same DAC whose values are taken from the database.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXCommandPreparingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
                AttributeTargets.Class)]
public class PXDBCalcedAttribute : PXEventSubscriberAttribute,
                                   IPXRowSelectingSubscriber,
                                   IPXCommandPreparingSubscriber,
                                   IPXFieldSelectingSubscriber
```

Properties

- `public virtual bool` **`Persistent`**

  Gets or sets the value that indicates whether the field the attribute is attached to is updated after a database commit operation.

Constructors

- `public PXDBCalcedAttribute(Type operand, Type type)`

  Initializes a new instance that uses the provided BQL expression to calculate the value of the field.

  Parameters:

  - `operand`

    The BQL query that is translated into SQL code that retrieves the value of the field. Specify any combination of BQL functions, constants, and the bound fields of the same DAC.

  - `type`

    The data type of the field.

Remarks

You should place the attribute on the field that is not bound to any particular database column.

The attribute will translate the provided BQL query into the SQL code and insert it into the select statement that retrieves data records of this DAC. In the BQL query, you can reference any bound field of the same DAC or an unbound field marked with *PXDBScalar*. You can also use BQL constants, arithmetic operations, equivalents of SQL function (such as `SUBSTRING` and `REPLACE`), and the `Switch` expression.

If, in contrast, you need to calculate the field on the server side at run time, use the *PXFormula* attribute.

Note that you should also annotate the field with an attribute that *indicates an unbound field* of a particular data type. Otherwise, the field may be displayed incorretly in the user interface.

Examples

The attribute below defines the expression to calculate the field of `decimal` type.

```
[PXDBCalced(typeof(Sub<POLine.curyExtCost, POLine.curyOpenAmt>),
            typeof(decimal))]
public virtual decimal? CuryClosedAmt { get; set; }
```

See the following example with the `Switch` expression.

```
[PXDBCalced(
    typeof(Switch<Case<Where<INUnit.unitMultDiv, Equal<MultDiv.divide>>,
        Mult<INSiteStatus.qtyOnHand, INUnit.unitRate>>,
        Div<INSiteStatus.qtyOnHand, INUnit.unitRate>>),
    typeof(decimal))]
public virtual decimal? QtyOnHandExt { get; set; }
```

See the following example with the more complex BQL expression.

```
[Serializable]
public class Product : PX.Data.IBqlTable
{
    ...
    [PXDecimal(2)]
    [PXDBCalced(typeof(
        Minus<Sub<Sub<IsNull<Product.availQty, decimal_0>,
                      IsNull<Product.bookedQty, decimal_0>>,
              Product.minAvailQty>>),
        typeof(decimal))]
    public virtual decimal? Discrepancy { get; set; }
    ...
}
```

This example also shows the enclosing declaration of the `Product` DAC. You can retrieve the records from the `Product` table by executing the following code in some graph.

```
PXSelect<Product>.Select(this);
```

This BQL statement will be translated into the following SQL query.

```
SELECT [other fields],
        -((ISNULL(Product.AvailQty, .0) - ISNULL(Product.BookedQty, .0))
          - Product.MinAvailQty) as Product.Discrepancy
FROM Product
```

PXDBScalar Attribute

Defines the SQL subrequest that will be used to retrieve the value for the DAC field.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
```

Syntax

```
[PXAttributeFamily( typeof(PXDBFieldAttribute))]
public class PXDBScalarAttribute : PXDBFieldAttribute
```

Constructors

- `public PXDBScalarAttribute(Type search)`

  Initializes a new instance that uses the provided `Search` command to retrieve the value of the field the attribute is attached to.

  Parameters:

  - `search`

    The BQL query based on the `Search` class or other class derived from `IBqlSearch`.

Remarks

You should place the attribute on the field that is not bound to any particular database column.

The attribute will translate the provided BQL `Search` command into the SQL subrequest and insert it into the select statement that retrieves data records of this DAC. In the BQL command, you can reference any bound field of any DAC.

Note that you should also annotate the field with an attribute that *indicates an unbound field* of a particular data type. Otherwise, the field may be displayed incorretly in the user interface.

You should not use fields marked with the `PXDBScalar` attribute in BQL parameters (`Current`, `Optional`, and `Required`).

Examples

The attribute below selects the `AcctName` value from the `Vendor` table as the `VendorName` value.

```
[PXString(50, IsUnicode = true)]
[PXDBScalar(typeof(
    Search<Vendor.acctName,
        Where<Vendor.bAccountID, Equal<RQRequestLine.vendorID>>>))]
public virtual string VendorName { get; set; }
```

PXDBUserPassword Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBCalcedAttribute
```

Interfaces

- `IPXFieldUpdatingSubscriber`

Syntax

```
public class PXDBUserPasswordAttribute : PXDBCalcedAttribute,
                                          IPXFieldUpdatingSubscriber
```

Constructors

- `public PXDBUserPasswordAttribute() : base(typeof(Users.password), typeof(string))`

Audit Fields

The following attributes are placed on DAC fields used for data audit. The framework binds these fields to database columns and automatically assigns field values.

- *PXDBCreatedByID*

  Maps a DAC field to the database column and automatically sets the field value to the ID of the user who created the data record.

- *PXDBCreatedByScreenID*

  Maps a DAC field to the database column and automatically sets the field value to the string ID of the application screen that created the data record.

- *PXDBCreatedDateTime*

Maps a DAC field to the database column and automatically sets the field value to the data record's creation date and time.

- *PXDBCreatedDateTimeUtc*

  Maps a DAC field to the database column and automatically sets the field value to the data record's creation UTC date and time.

- *PXDBLastModifiedByID*

  Maps a DAC field to the database column and automatically sets the field value to the ID of the user who was the last to modify the data record.

- *PXDBLastModifiedByScreenID*

  Maps a DAC field to the database column and automatically sets the field value to the string ID of the application screen on which the data record was modified the last time.

- *PXDBLastModifiedDateTime*

  Maps a DAC field to the database column and automatically sets the field value to the data record's last modification date and time.

- *PXDBLastModifiedDateTimeUtc*

  Maps a DAC field to the database column and automatically sets the field value to the data record's last modification date and time in UTC.

PXDBCreatedByID Attribute

Maps a DAC field to the database column and automatically sets the field value to the ID of the user who created the data record.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXAggregateAttribute
```

Interfaces

- `IPXRowInsertingSubscriber`
- `IPXFieldVerifyingSubscriber`

Syntax

```
[Serializable]
public class PXDBCreatedByIDAttribute : PXAggregateAttribute,
                                        IPXRowInsertingSubscriber,
                                        IPXFieldVerifyingSubscriber
```

Properties

- `public Type` **`BqlField`**

  Returns `null` on get. Sets the BQL field representing the field in BQL queries.

- `public bool` **`DontOverrideValue`**

  Gets or sets the value that indicates whether a field update is allowed after the field value is set for the first time.

Constructors

- `public PXDBCreatedByIDAttribute() : this(typeof(Creator.pKID), typeof(Creator.username), typeof(Creator.username),`

  Initializes a new instance of the attribute.

Nested Classes

- `public sealed class Creator : Users`

  The class used internally to represent the creator of a data record.

  Nested classes:

  - `public new abstract class pKID : IBqlField`
  - `public new abstract class username : IBqlField`

  Properties:

  - `public override String` **`Username`**

    Gets or sets the user name.

    Syntax:

    ```
    [PXDBString]
    [PXUIField(DisplayName = "Created By",
               Enabled = false,
               Visibility = PXUIVisibility.SelectorVisible)]
    public override String Username { get; set; }
    ```

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `Guid?`.

The attribute aggregates the *PXDBGuid* and `PXDisplaySelector` (derives from *PXSelector*).

Examples

```
[PXDBCreatedByID()]
public virtual Guid? CreatedByID { get; set; }
```

PXDBCreatedByScreenID Attribute

Maps a DAC field to the database column and automatically sets the field value to the string ID of the application screen that created the data record.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
```

Interfaces

- `IPXRowInsertingSubscriber`

Syntax

```
public class PXDBCreatedByScreenIDAttribute : PXDBStringAttribute,
                                              IPXRowInsertingSubscriber
```

Constructors

- `public PXDBCreatedByScreenIDAttribute() : base(10)`

  Initializes a new instance of the attribute.

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `string`.

Examples

```
[PXDBCreatedByScreenID()]
public virtual string CreatedByScreenID { get; set; }
```

PXDBCreatedDateTime Attribute

Maps a DAC field to the database column and automatically sets the field value to the data record's creation date and time.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDateAttribute
```

Interfaces

- `IPXCommandPreparingSubscriber`
- `IPXRowInsertingSubscriber`

Syntax

```
public class PXDBCreatedDateTimeAttribute : PXDBDateAttribute,
                                            IPXCommandPreparingSubscriber,
                                            IPXRowInsertingSubscriber
```

Constructors

- `public PXDBCreatedDateTimeAttribute() : base()`

  Initializes a new instance of the attribute.

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `DateTime?`.

Examples

```
[PXDBCreatedDateTime()]
public virtual DateTime? CreatedDateTime { get; set; }
```

PXDBCreatedDateTimeUtc Attribute

Maps a DAC field to the database column and automatically sets the field value to the data record's creation UTC date and time.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDateAttribute
      PXDBCreatedDateTimeAttribute
```

Syntax

```
public class PXDBCreatedDateTimeUtcAttribute : PXDBCreatedDateTimeAttribute
```

Constructors

- `public PXDBCreatedDateTimeUtcAttribute() : base()`

  Initializes a new instance of the attribute.

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `DateTime?`.

Examples

```
[PXDBCreatedDateTimeUtc]
[PXUIField(DisplayName = "Date Created", Enabled = false)]
public virtual DateTime? CreatedDateTime { get; set; }
```

PXDBLastModifiedByID Attribute

Maps a DAC field to the database column and automatically sets the field value to the ID of the user who was the last to modify the data record.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXAggregateAttribute
    PXDBCreatedByIDAttribute
```

Interfaces

- `IPXRowUpdatingSubscriber`

Syntax

```
[Serializable]
public class PXDBLastModifiedByIDAttribute : PXDBCreatedByIDAttribute,
                                             IPXRowUpdatingSubscriber
```

Constructors

- `public PXDBLastModifiedByIDAttribute() : base(typeof(Modifier.pKID), typeof(Modifier.username), typeof(Modifier.username),`

  Initializes a new instance of the attribute.

Nested Classes

- `public sealed class Modifier : Users`

The class used internally to represent the user who modified the data record.

Nested classes:

- `public new abstract class pKID : IBqlField`
- `public new abstract class username : IBqlField`

Properties

- `public override String` **`Username`**

  Gets or sets the user name.

  Syntax:

  ```
  [PXDBString]
  [PXUIField(DisplayName = "Last Modified By",
             Enabled = false,
             Visibility = PXUIVisibility.SelectorVisible)]
  public override String Username { get; set; }
  ```

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `Guid?`.

Examples

```
[PXDBLastModifiedByID()]
[PXUIField(DisplayName = "Last Modified By")]
public virtual Guid? LastModifiedByID { get; set; }
```

PXDBLastModifiedByScreenID Attribute

Maps a DAC field to the database column and automatically sets the field value to the string ID of the application screen on which the data record was modified the last time.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBStringAttribute
      PXDBCreatedByScreenIDAttribute
```

Interfaces

- `IPXRowUpdatingSubscriber`

Syntax

```
public class PXDBLastModifiedByScreenIDAttribute :
    PXDBCreatedByScreenIDAttribute,
    IPXRowUpdatingSubscriber
```

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `string`.

Examples

```
[PXDBLastModifiedByScreenID()]
```

```
public virtual string LastModifiedByScreenID { get; set; }
```

PXDBLastModifiedDateTime Attribute

Maps a DAC field to the database column and automatically sets the field value to the data record's last modification date and time.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDateAttribute
      PXDBCreatedDateTimeAttribute
```

Interfaces

- `IPXCommandPreparingSubscriber`
- `IPXRowUpdatingSubscriber`

Syntax

```
public class PXDBLastModifiedDateTimeAttribute :
    PXDBCreatedDateTimeAttribute,
    IPXCommandPreparingSubscriber,
    IPXRowUpdatingSubscriber
```

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `DateTime?`.

Examples

```
[PXDBLastModifiedDateTimeUtc]
[PXUIField(DisplayName = "Last Modified Date", Enabled = false)]
public virtual DateTime? LastModifiedDateTime { get; set; }
```

PXDBLastModifiedDateTimeUtc Attribute

Maps a DAC field to the database column and automatically sets the field value to the data record's last modification date and time in UTC.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDateAttribute
      PXDBCreatedDateTimeAttribute
        PXDBLastModifiedDateTimeAttribute
```

Syntax

```
public class PXDBLastModifiedDateTimeUtcAttribute :
 PXDBLastModifiedDateTimeAttribute
```

Constructors

- `public PXDBLastModifiedDateTimeUtcAttribute()`

Initializes a new instance of the attribute.

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `DateTime?`.

Examples

```
[PXDBLastModifiedDateTimeUtc]
[PXUIField(DisplayName = "Last Modified Date", Enabled = false)]
public virtual DateTime? LastModifiedDateTime { get; set; }
```

PXDBLastChangeDateTime Attribute

Maps a DAC field to the database column and automatically sets the field value to the data record's last modification date and time.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBDateAttribute
```

Interfaces

- `IPXRowUpdatingSubscriber`

Syntax

```
public class PXDBLastChangeDateTimeAttribute : PXDBDateAttribute,
                                               IPXRowUpdatingSubscriber
```

Constructors

- `public PXDBLastChangeDateTimeAttribute(Type monitoredField)`

  Initializes a new instance that will monitor the specified field. On each modification of this field, the attribute will update assign the modification date and time to the field that is marked with the attribute.

  Parameters:

  - `monitoredField`

    The field to monitor. Specify a type that implements `IBqlField`.

Remarks

The attribute is added to the value declaration of a DAC field. The field data type should be `DateTime?`.

Examples

```
[PXDBLastChangeDateTime(typeof(CRCase.status))]
public virtual DateTime? StatusDate { get; set; }
```

Data Projection

The following attributes implement projection of data from one or several data into a single data access class (DAC):

- *PXProjection*

  Binds the DAC to an arbitrary data set. The attribute thus defines a named view, but implemented by the server side rather then the database.

- *PXExtraKey*

  Indicates that the field implements a relationship between two tables. The use of this attribute enables update of the referenced table on update of the projection.

PXProjection Attribute

Binds the DAC to an arbitrary data set defined by the `Select` command. The attribute thus defines a named view, but implemented by the server side rather then the database.

Inheritance Hierarchy

```
Attribute
  PXDBInterceptorAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXProjectionAttribute : PXDBInterceptorAttribute
```

Properties

- `public bool` **`Persistent`**

  Gets or sets the value that indicates whether the instances of the DAC that represents the projection can be saved to the database. If the property equals `true`, the attribute will parse the `Select` command and determine the tables that should be updated. Alternatively, you can specify the list of tables in the constructor. If the property equals `false`, the DAC is readonly.

Constructors

Constructor	Description
PXProjectionAttribute(Type)	Initializes a new instance that binds the DAC to the data set defined by the provided `Select` command
PXProjectionAttribute(Type, Type[])	Initializes a new instance that binds the DAC to the specified data set and enables saving of the DAC instances to the database

Remarks

You can place the attribute on the DAC declaration. The framework doesn't bind such DAC to a database table—that is, doesn't select data from the table having the same name as the DAC. Instead, you specify an arbitrary BQL `Select` command that is executed to retrieve data for the DAC. The `Select` command can select data from one or several comands and include any BQL clauses.

By default, the projection is readonly, but you can make it updatable by setting the `Persistent` property to `true`. The attribute will use the `Select` command to determine which tables needs updating. However, only the first table referenced by the `Select` command is updated by default. If the data should be committed not only into main table, but also to the joined tables, the fields that connect the tables must be marked with the *PXExtraKey* attribute. Additionally, you can use the constructor with two parameters to provide the list of table explicitly. This list should include the tables referenced in the `Select` command. This constructor will also set the `Persistent` property to `true`.

You should explicitly map the projection fields to the column retrieved by the `Select` command. To map a field, set the `BqlField` property of the attribute that binds the field to the database (such as `PXDBString` and `PXDBDecimal`) to the type that represents the column, as follows.

```
[PXDBString(15, IsUnicode = true,
            BqlField = typeof(Supplier.accountCD))]
public virtual string AccountCD { get; set; }
```

Examples

In the following example, the attribute joins data from two table and projects it to the single DAC.

```
[Serializable]
[PXProjection(typeof(
    Select2<Supplier,
        InnerJoin<SupplierProduct,
            On<SupplierProduct.accountID, Equal<Supplier.accountID>>>>))]
public partial class SupplierPrice : IBqlTable
{
    public abstract class accountID : PX.Data.IBqlField
    {
    }
    // The field mapped to the Supplier field (through setting of BqlField)
    [PXDBInt(IsKey = true, BqlField = typeof(Supplier.accountID))]
    public virtual int? AccountID { get; set; }

    public abstract class productID : PX.Data.IBqlField
    {
    }
    // The field mapped to the SupplierProduct field
    // (through setting of BqlField)
    [PXDBInt(IsKey = true, BqlField = typeof(SupplierProduct.productID))]
    [PXUIField(DisplayName = "Product ID")]
    public virtual int? ProductID { get; set; }

    ...
}
```

Note how the DAC declares the fields. The projection defined in the example is readonly. To make it updatable, you should set the `Persistent` property to `true`, changing the attribute declaration to the following one.

```
[PXProjection(
    typeof(Select2<Supplier,
        InnerJoin<SupplierProduct,
            On<SupplierProduct.accountID, Equal<Supplier.accountID>>>>),
    Persistent = true
)]
```

If the projection should be able to update both tables, you should place the *PXExtraKey* attribute on the field that relates the tables—the `AccountID` property—as follows.

```
[PXDBInt(IsKey = true, BqlField = typeof(Supplier.accountID))]
[PXExtraKey]
public virtual int? AccountID { get; set; }
```

PXProjection Attribute Constructors

The *PXProjection* attribute exposes the following constructors.

PXProjectionAttribute(Type)

Initializes a new instance that binds the DAC to the data set defined by the provided `Select` command.

Syntax:

```
public PXProjectionAttribute(Type select)
```

Parameters:

- `select`

  The BQL command that defines the data set, based on the `Select` class or any other class that implements `IBqlSelect`.

PXProjectionAttribute(Type, Type[])

Initializes a new instance that binds the DAC to the specified data set and enables update saving of the DAC instances to the database. The tables that should be updated during update of the current DAC.

Syntax:

```
public PXProjectionAttribute(Type select, Type[] persistent) : this(select)
```

Parameters:

- `select`

  The BQL command that defines the data set, based on the `Select` class or any other class that implements `IBqlSelect`.

- `persistent`

  The list of DACs that represent the tables to update during update of the current DAC.

PXExtraKey Attribute

Indicates that the field implements a relationship between two tables in a projection. The use of this attribute enables update of the referenced table on update of the projection.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXCommandPreparingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXExtraKeyAttribute : PXEventSubscriberAttribute,
                                   IPXCommandPreparingSubscriber
```

Remarks

You can place the attribute on the field declaration in the DAC that represents a *projection*. The attribute is required when the projection combines data from joined tables and more than one table needs to be updated on update of the projection. In this case the attribute should be placed on all fields that implement the relationship between the main and the joined tables.

Examples

The following example shows the declaration of a projection that can update data in two tables.

```
// Projection declaration
[PXProjection(
    typeof(
        Select2<CRCampaignMembers,
            RightJoin<Contact,
                On<Contact.contactID, Equal<CRCampaignMembers.contactID>>>>
    ),
    Persistent = true)]
[Serializable]
public partial class SelCampaignMembers : CRCampaignMembers, IPXSelectable
{
    ...
    // The field connecting the current DAC with the Contact DAC
    [PXDBInt(BqlField = typeof(Contact.contactID))]
    [PXExtraKey]
    public virtual int? ContactContactID { get; set; }
    ...
}
```

Note that the `Select` commands retrieves data from two tables, `CRCampaignMembers` and `Contact`. To make the projection updatable, you set the `Persistent` property to `true`. The projection field that implements relationship between the tables is marked with the `PXExtraKey` attribute.

Access Control

The group mask value indicates access rights a user should have to use a data record. To be able to set access rights for particular data records, you should use the *PXDBGroupMask* attribute to mark the DAC field that holds the group mask value.

PXDBGroupMask Attribute

Marks a DAC field of `byte[]` type that holds the group mask value.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBBinaryAttribute
```

Syntax

```
public class PXDBGroupMaskAttribute : PXDBBinaryAttribute
```

Constructors

Constructor	Description
PXDBGroupMaskAttribute()	Initializes an instance of the attribute with default parameters
PXDBGroupMaskAttribute(int)	Initializes an instance of the attribute with the specified maximum length of the value

Examples

The code below shows definition of a DAC field tha holds a group mask value.

```
[PXDBGroupMask()]
```

```
public virtual Byte[] GroupMask { get; set; }
```

PXDBGroupMask Attribute Constructors

The *PXDBGroupMask* attribute exposes the following constructors.

PXDBGroupMaskAttribute()

Initializes an instance of the attribute with default parameters.

Syntax:

```
public PXDBGroupMaskAttribute() : base()
```

PXDBGroupMaskAttribute(int)

Initializes an instance of the attribute with the specified maximum length of the value.

Syntax:

```
public PXDBGroupMaskAttribute(int length) : base(length)
```

Notes

By using the *PXNote* attribute, you enable a user to attach text notes, files, and activity items to data records.

You should use the `PXNote` attribute in the data access class of these data records to mark the field that will store the identifier of a note in the `Note` table. Basically, notes are used to attach text to data record. This text is stored in the note data record in the `Note` table. Additionally, you can attach files or other entities to a data record through a note. This feature is implemented through additional tables that store identifiers of a note and the attached entity.

The `PXNote` attribute can also be configured to save the specified table fields in a note. In this case, the user will be able to search the data records by the values saved in the note, using the Acumatica Framework application website search.

PXNote Attribute

Binds a DAC field of `long?` type to the database column that keeps note identifiers and enables attachment of text comments, files, and activity items to a data record.

See *Remarks* for more details. See *Examples* for examples of usage.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBLongAttribute
```

Interfaces

- `IPXRowPersistingSubscriber`
- `IPXRowPersistedSubscriber`
- `IPXRowDeletedSubscriber`
- `IPXReportRequiredField`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXNoteAttribute : PXDBLongAttribute,
                               IPXRowPersistingSubscriber,
                               IPXRowPersistedSubscriber,
                               IPXRowDeletedSubscriber,
                               IPXReportRequiredField
```

Properties

- `public Type[]` **`ExtraSearchResultColumns`**

  Gets or sets the list of fields that will be displayed in a separate column when rendering search results.

- `public Type[]` **`Searches`**

  Gets the list of fields whose values will be saved in the note and will be available to the website search. The default value is null. The property is set through the constructor.

- `public Type[]` **`ForeignRelations`**

  Gets or sets the list of fields that connect the current table with foreign tables. The fields from the foreign tables can be specified along with current table fields in the `Searches` list.

- `public bool` **`ShowInReferenceSelector`**

  Gets or sets the value that indicates whether activity items can be associated with the DAC where the `PXNote` attribute is used. If the property equals `true`, the DAC will appear in the list of types in the lookup that selects the related data record for an activity. If the property equals `false`, activity attributes cannot be associated with data records of the DAC. By default the property equals `false`.

- `public Type` **`DescriptionField`**

  Gets or set the field whose value will be displayed as value in the lookup that selects the related data record for an activity.

- `public Type` **`Selector`**

  Gets or sets the BQL expression that selects the data records to be displayed in the pop-up window of the lookup that selects the related data record for an activity. As the BQL expression, you can specify a `Search<>` command or just a field. This field, or the main field of the `Search<>` command, will be the value that identifies a data record in the activity item.

- `public Type[]` **`FieldList`**

  Gets or set the list of columns that will be displayed in the pop-up window of the lookup that selects the related data record for an activity.

Constructors

Constructor	Description
PXNoteAttribute()	Initializes a new instance of the attribute
PXNoteAttribute(params Type[])	Initializes an instance of the attribute that will save values of the provided fields in the note

Static Methods

Method	Description
GetFileNotes(PXCache, object)	Returns the list of identifiers of files that are shown in the **Files** pop-up window
GetNote(PXCache, object)	Returns the text comment of the note attached to the provided object
GetNoteID(PXCache, object, string)	Returns the identifier of the note attached to the provided object and inserts a new note into the cache if the note does not exist
GetNoteID<Field>(PXCache, object)	Returns the identifier of the note attached to the provided object and inserts a new note into the cache if the note does not exist
GetNoteIDNow(PXCache, object)	Returns the identifier of the note attached to the provided object and inserts a new not into the database if the note does not exist
GetNoteIDReadonly(PXCache, object, string)	Returns the identifier of the note attached to the provided object or `null` if the note does not exist
GetNoteIDReadonly<Field>(PXCache, object)	Returns the identifier of the note attached to the provided object or `null` if the note does not exist
SetFileNotes(PXCache, object, params Guid[])	Sets the list of identifiers of files that are shown in the **Files** pop-up window
SetNote(PXCache, object, string)	Sets the text of the note attached to the provided data record
UpdateEntityType(PXCache, object, string, Type)	Sets the DAC type of the data record to which the note is attached

Remarks

The attribute should be placed on the DAC field that will hold the identifier of the related note. A note is a data record in the `Note` database table. A note data record contains the note identifier, the text comment, the DAC name of the related data record, and some other fields.

Only one data record can reference a note. So the identifier of this note can be used as the global identifier of the data record. Thanks to this fact, in addition to adding text comments to a data record notes are used to implement:

- *Full-text search of data records*: A note can be used to store the specified fields of the related data record, which can be found by these fields through the website search.
- *File attachments*: The relationships between files and notes are kept in a separate table, `NoteDoc`, as pairs of a file identifier and note identifier. The `UploadFile` stores general information about files, and the `UploadRevision` stores specific revisions of files.
- *Association of activity items with a data record*.

For any of these features to work, the given DAC should define a field marked with the `PXNote` attribute.

Examples

The attribute below indicates that the DAC field references a note.

```
[PXNote(new Type[0])]
public virtual long? NoteID { get; set; }
```

Here, `new Type[0]` as parameter is used to force creation of the note on saving of a data record even if the used did not create a note manually.

The attribute below indicates that the DAC field holds note identifier, sets the lists of fields (from different tables) that will be saved in the note, and allows association of a data record with activity items. It will be possible to find the `Vendor` data record through the application website search by the values of these fields.

```
[PXNote(
    typeof(Vendor.acctCD),
    typeof(Vendor.acctName),
    typeof(Contact.eMail),
    typeof(Contact.phone1),
    typeof(Contact.fax),
    typeof(Address.addressLine1),
    typeof(Address.city),
    typeof(Address.countryID),
    typeof(Address.postalCode),
    ForeignRelations =
        new Type[] { typeof(Vendor.defContactID),
                     typeof(Vendor.defAddressID) },
    ExtraSearchResultColumns =
        new Type[] { typeof(CR.Contact) },

    ShowInReferenceSelector = true,
    DescriptionField = typeof(Vendor.acctCD),
    Selector = typeof(Vendor.acctCD)
)]
public virtual long? NoteID { get; set; }
```

The first few parameters specify fields to save in the note. The `ForeignRelations` property specifies the `Vendor` fields that reference the related `Contact` and `Address` data records. Fields from these tables are also provided among the field to save in the note.

The `ShowInReferenceSelector` allows attaching activity items to `Vendor` data records. On the activity webpage, the lookup field for selecting a related data record will display the `Vendor.AcctCD` (configured by `DescriptionField`) when a `Vendor` data record is selected and use the same field (due to `Selector`) as the reference value.

PXNote Attribute Constructors

The *PXNote* attribute exposes the following constructors.

PXNoteAttribute()

Initializes a new instance of the attribute that will be used to attach notes to data record but won't save values of the fields in a note.

Syntax:

```
public PXNoteAttribute()
```

PXNoteAttribute(params Type[])

Initializes an instance of the attribute that will save values of the provided fields in the note. The values saved in a note will be updated each time the data record is saved.

If you don't need to save fields in the note, but need to have a note automatically created for each data record of the current DAC type, provide an empty array as the parameter:

```
[Note(new Type[0])]
```

Syntax:

```
public PXNoteAttribute(params Type[] searches)
```

Examples:

- `params searches`

  The fields to save within the note to enable full-text search of a data record by these fields.

PXNote Attribute Methods

The *PXNote* attribute exposes the following static methods.

GetFileNotes(PXCache, object)

Returns the list of identifiers of files that are shown in the **Files** pop-up window.

Syntax:

```
public static Guid[] GetFileNotes(PXCache sender, object data)
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

GetNote(PXCache, object)

Returns the text comment of the note attached to the provided object.

Syntax:

```
public static string GetNote(PXCache sender, object data)
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

GetNoteID(PXCache, object, string)

Returns the identifier of the note attached to the provided object and inserts a new note into the cache if the note does not exist.

Syntax:

```
public static long GetNoteID(PXCache cache, object data, string name)
```

Parameters:

- `sender`

The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

- `name`

  The name of the field that stores note identifier. If `null`, the method will search attributes on all fields and use the first `PXNote` attribute it finds.

GetNoteID<Field>(PXCache, object)

Returns the identifier of the note attached to the provided object and inserts a new note into the cache if the note does not exist. The field that stores note identifier is specified in the type parameter.

Syntax:

```
public static long GetNoteID<Field>(PXCache cache, object data)
    where Field : IBqlField
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

GetNoteIDNow(PXCache, object)

Returns the identifier of the note attached to the provided object and inserts a new not into the database if the note does not exist.

Syntax:

```
public static long? GetNoteIDNow(PXCache cache, object data)
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

GetNoteIDReadonly(PXCache, object, string)

Returns the identifier of the note attached to the provided object or `null` if the note does not exist.

Syntax:

```
public static long? GetNoteIDReadonly(PXCache cache, object data, string name)
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

- `name`

  The name of the field that stores note identifier. If `null`, the method will search attributes on all fields and use the first `PXNote` attribute it finds.

GetNoteIDReadonly<Field>(PXCache, object)

Returns the identifier of the note attached to the provided object or `null` if the note does not exist. The field that stores note identifier is specified in the type parameter.

Syntax:

```
public static long? GetNoteIDReadonly<Field>(PXCache cache, object data)
    where Field : IBqlField
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

SetFileNotes(PXCache, object, params Guid[])

Sets the list of identifiers of files that are shown in the **Files** pop-up window.

Syntax:

```
public static void SetFileNotes(PXCache cache, object data,
                                params Guid[] fileIDs)
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

- `fileIDs`

  The indetifiers of files to display.

SetNote(PXCache, object, string)

Sets the text of the note attached to the provided data record.

Syntax:

```
public static void SetNote(PXCache sender, object data, string note)
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.

- `data`

  The data record the method is applied to.

- `note`

The text to place in the note.

UpdateEntityType(PXCache, object, string, Type)

Sets the DAC type of the data record to which the note is attached. The full name of the DAC is saved in the database in the note record. This information is used, for example, to determine the webpage to open to show full details of the data record associated with a note.

Syntax:

```
public static void UpdateEntityType(PXCache cache, object data,
                                    string noteFieldName, Type newEntityType)
```

Parameters:

- `sender`

  The cache object to search for the attributes of `PXNote` type.
- `data`

  The data record the method is applied to.
- `noteFieldName`

  The name of the field that stores note identifier.
- `newEntityType`

  New DAC type to associate with the note.

Report Optimization

The value of an unbound DAC field can be calculated in the property getter. The calculation can involve other fields of the same DAC. However, at the time when the value of the DAC field is requested, other fields are not guaranteed to be calculated or assigned their values. Such situations are normal when the Integration Services or Copy-Paste functionality is used, or when the field is used in reports.

To ensure that the fields referenced in the property getter have values at the time when it is executed, you should use the *PXDependsOnFields* attribute.

PXDependsOnFields Attribute

Used for calculated DAC fields that contain referenses to other fields in their property getters. The attribute allows such fields to work properly in reports and Integration Services.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property, AllowMultiple = false)]
public sealed class PXDependsOnFieldsAttribute : Attribute
```

Constructors

- `public PXDependsOnFieldsAttribute(params Type[] fields)`

  Initializes an instance of the attribute that makes the field the attribute is attached to depend on the provided DAC fields.

Examples

The code below shows definition of a calculated DAC field.

```
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Balance")]
public virtual Decimal? ActualBalance
{
    [PXDependsOnFields(typeof(docBal), typeof(taxWheld))]
    get
    {
        return this.DocBal - this.TaxWheld;
    }
}
```

The property getter involves two fields, `DocBal` and `TaxWheld`. These two fields should be specified as parameters of the `PXDependsOnFields` attribute.

Attributes on DACs

You can place the following attributes on the data access class (DAC) declaration:

- *PXPrimaryGraph Attribute*

  Sets the graph that is used by default to edit a data record.
- *PXCacheName Attribute*

  Sets the user-friendly name of the DAC. The name is displayed in the user interface.
- *PXTable*

  Binds a DAC that derives from another DAC to the table having the name of the derived table. Without the attribute, the derived DAC will be bound to the same table as the DAC that starts the inheritance hierarchy.
- *PXAccumulator Attribute*

  Updates values of a data record in the database according to specified policies.
- *PXHidden Attribute*

  Allows the developer to hide a DAC, a graph, or a view from the selectors of DACs and graphs and the Web Service API (in particular, from reports).
- *PXEMailSource Attribute*

The *PXProjection* and *PXTable* attributes can also mark a DAC. See *Data Projection* for more details.

PXPrimaryGraph Attribute

Sets the primary graph for the DAC. The primary graph determines the default page where a user is redirected for editing a data record.

Inheritance Hierarchy

```
PXPrimaryGraphBaseAttribute
```

Syntax

```
public class PXPrimaryGraphAttribute : PXPrimaryGraphBaseAttribute
```

Constructors

Constructor	Description
PXPrimaryGraphAttribute(Type)	Initializes a new instance that will use the provided graph to edit a data record
PXPrimaryGraphAttribute(Type[], Type[])	Initializes a new instance that will use the graph corresponding to the first satisfied condition

Static Methods

Method	Description
FindPrimaryGraph(PXCache, out)	Finds the primary graph of the DAC the cache object corresponds to

Remarks

The attribute can be placed on the following declarations:

- On the DAC to specify the primary graph for this DAC.
- On the graph to indicate that it is the primary graph for the specified DACs.

The second methods overrides the primary graph set by the first method.

You can specify several graphs and a set of the correspond conditions. In this case, the first graph for which the condition holds true at run time is considered the primary graph. A condition is a BQL query based on either the `Where` class or the `Select` class.

Examples

In the example below, the attribute specifies the primary graph for a DAC.

```
[PXPrimaryGraph(typeof(SalesPersonMaint))]
public partial class SalesPerson : PX.Data.IBqlTable
{
    ...
}
```

In the example below, the attribute specifies the graph that is used as the primary graph for a DAC if the condition holds true for the data in the cache.

```
[PXPrimaryGraph(
    new Type[] { typeof(ShipTermsMaint)},
    new Type[] { typeof(Select<ShipTerms,
        Where<ShipTerms.shipTermsID, Equal<Current<ShipTerms.shipTermsID>>>>)
    })]
public partial class ShipTerms : PX.Data.IBqlTable
{
    ...
}
```

In the example below, the attribute specifies the graph that is used as the primary graph for a DAC if the `Select` statement retrieves a non-empty data set.

```
[PXPrimaryGraph(
    new Type[] { typeof(CountryMaint)},
    new Type[] { typeof(Select<State,
        Where<State.countryID, Equal<Current<State.countryID>>,
            And<State.stateID, Equal<Current<State.stateID>>>>>)
    })]
public partial class State : PX.Data.IBqlTable
```

```
{
    ...
}
```

In the example below, the attribute specifies two graphs and the corresponding `Select` statements. The first graph for which the `Select` statement returns a non-empty data set is used as the primary graph for the DAC.

```
[PXPrimaryGraph(
    new Type[] {
        typeof(APQuickCheckEntry),
        typeof(APPaymentEntry)
    },
    new Type[] {
        typeof(Select<APQuickCheck,
            Where<APQuickCheck.docType, Equal<Current<APPayment.docType>>,
                And<APQuickCheck.refNbr, Equal<Current<APPayment.refNbr>>>>>),
        typeof(Select<APPayment,
            Where<APPayment.docType, Equal<Current<APPayment.docType>>,
                And<APPayment.refNbr, Equal<Current<APPayment.refNbr>>>>>)
    })]
public partial class APPayment : APRegister, IInvoice
{
    ...
}
```

PXPrimaryGraph Attribute Constructors

The *PXPrimaryGraph* attribute exposes the following constructors.

PXPrimaryGraphAttribute(Type)

Initializes a new instance that will use the provided graph to edit a data record.

Syntax:

```
public PXPrimaryGraphAttribute(Type type)
```

Parameters:

- `type`

  The business logic controller (graph) or the DAC. The graph should derive from `PXGraph`. The DAC should implement `IBqlTable`.

PXPrimaryGraphAttribute(Type[], Type[])

Initializes a new instance that will use the graph corresponding to the first satisfied condition. Provide the array of graphs and the array of corresponding conditions.

Syntax:

```
public PXPrimaryGraphAttribute(Type[] types, Type[] conditions)
```

Parameters:

- `type`

  The array of business logic controllers (graphs) or DACs. A graph should derive from `PXGraph`. A DAC should implement `IBqlTable`.

- `conditions`

  The array of conditions that correspond to the graphs or DACs specified in the first parameter. Specify BQL queries, either `Where` expressions or `Select` commands.

PXPrimaryGraph Attribute Methods

The *PXPrimaryGraph* attribute exposes the following static methods.

FindPrimaryGraph(PXCache, out)

Finds the primary graph of the DAC the cache object corresponds to. Sets the discovered graph type to the out parameter and returns the attribute instance.

Syntax:

```
public static PXPrimaryGraphBaseAttribute FindPrimaryGraph(PXCache cache, out
            Type graphType)
```

Parameters:

- `cache`

  The cache object to search for the attributes of `PXPrimaryGraph` type.

- `(out) graphType`

  The discovered primary graph type.

PXCacheName Attribute

Sets the user-friendly name of the data access class (DAC).

Inheritance Hierarchy

```
Attribute
  PXNameAttribute
```

Syntax

```
public class PXCacheNameAttribute : PXNameAttribute
```

Constructors

- `public PXCacheNameAttribute(string name) : base(name)`

  Initializes a new instance that assigns the specified name to the DAC.

Remarks

The attribute is added to the DAC declaration. The name can be obtained at run time through the *GetItemName(PXCache)* static method of the `PXUIField` attribute.

Examples

```
[PXCacheName("Currency Info")]
public partial class CurrencyInfo : PX.Data.IBqlTable
{
    ...
}
```

PXTable Attribute

Binds a DAC that derives from another DAC to the table having the name of the derived DAC. Without the attribute, the derived DAC will be bound to the same table as the DAC that starts the inheritance hierarchy.

Inheritance Hierarchy

```
Attribute
  PXDBInterceptorAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXTableAttribute : PXDBInterceptorAttribute
```

Properties

- `public bool` **`IsOptional`**

  Gets or sets the value that indicates whether the base DAC data record can exist without the extension DAC data record. This situation corresponds to the use of the attribute on the extension DAC that is bound to a separate database table. By default, the value is `false`, and the data record in the extension table is always created for a data record of the base table.

Constructors

Constructor	Description
PXTableAttribute()	Initializes a new instance of the attribute
PXTableAttribute(params Type[])	Initializes a new instance of the attribute when the base DAC has a pair of surrogate and natural keys

Remarks

The attribute is placed on the declaration of a DAC.

The attribute can be used in customizations. You place it on the declaration of a DAC extension to indicate that the extension fields are bound to a separate table.

Examples

The `PXTable` attribute below indicates that the `APInvoice` DAC is bound to the `APInvoice` table. Without the attribute, it would be bound to the `APRegister` table.

```
[System.SerializableAttribute()]
[PXTable()]
public partial class APInvoice : APRegister, IInvoice
{
    ...
}
```

The `PXTable` attribute below indicates that the `FSxLocation` extension of the `Location` DAC is bound to a separate table and the `Location` DAC can include data records that do not have the corresponding data records in the extension table.

```
[PXTable(typeof(Location.bAccountID),
         typeof(Location.locationID),
         IsOptional = true)]
public class FSxLocation : PXCacheExtension<Location>
{
    ...
}
```

Here, you specify the key fields of the `Location` DAC, because it includes a surrogate-natural pair of key fields, `LocationID` (which is the database key as well) and `LocationCD` (human-readable value). In the `PXTable` attribute, you specify the surrogate `LocationID` field.

PXTable Attribute Constructors

The *PXTable* attribute exposes the following constructors.

PXTableAttribute()

Initializes a new instance of the attribute.

Syntax:

```
public PXTableAttribute()
```

PXTableAttribute(params Type[])

Initializes a new instance of the attribute when the base DAC has a pair of surrogate and natural keys. In this case, in the parameters, you should specify all key fields of the base DAC. From the pair of the surrogate and natural keys, you include only the surrogate key.

Syntax:

```
public PXTableAttribute(params Type[] links) : this()
```

Parameters:

- `links`

  The list of key fields of the base DAC.

PXAccumulator Attribute

Updates values of a data record in the database according to specified policies. You can derive a custom attribute from this attribute and override the `PrepareInsert()` method to set other assignment behavior for target values (such as taking the maximum instead of summarizing).

Inheritance Hierarchy

```
Attribute
  PXDBInterceptorAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXAccumulatorAttribute : PXDBInterceptorAttribute
```

Properties

- `public virtual bool` **`SingleRecord`**

  Gets or sets the value that indicates whether the attribute always updates only a single data record.

Constructors

Constructor	Description
PXAccumulatorAttribute()	Empty default constructor

Constructor	Description
PXAccumulatorAttribute(Type[], Type[])	Initializes an instance of the attribute with the source fields and destination fields

PrepareInsert(PXCache, object, PXAccumulatorCollection)

The method to override in a successor of the `PXAccumulator` attribute and set policies for fields.

The method is invoked by the `PersistInserted()` method of the `PXAccumulator` attribute.

Typically, when you override this method, you call the base version of the method and set the policies for fields by calling the `Update<>()` method of the `columns` parameter.

Syntax:

```
protected virtual bool PrepareInsert(PXCache sender, object row,
                                     PXAccumulatorCollection columns)
```

Parameters:

- `sender`

  The cache object into which the data record is inserted.
- `row`

  The data record to insert into the cache.
- `columns`

  The object representing columns.

PersistInserted(PXCache, object)

The method that will be executed by the cache instead of the standard *PersistInserted(object)* method. If the attribute is attached to the cache, the cache will discover that a successor of the `PXInterceptor` attribute is attached, invoke the attribute's method from the standard method, and quit the standard method.

If you only need to set insertion policies for some DAC field, you should override only the `PrepareInsert()` method. Overriding the `PersistInserted()` method is needed to tweak the persist operation—for example, to catch and process errors.

Syntax:

```
public override bool PersistInserted(PXCache sender, object row)
```

Parameters:

- `sender`

  The cache object into which the data record is inserted.
- `row`

  The inserted data record to be saved to the database.

Remarks

You can use the attribute on its own or derive a custom attribute. Both a successor of `PXAccumulator` and the `PXAccumulator` attribute itself should be placed on the definition of a DAC.

To define custom policy for fields of the specified DAC, you should derive a custom class from this attribute and override the `PrepareInsert()` method. The method is called within the

`PersistInserted()` method of the `PXAccumulator`. You can override the `PersistInserted()` method as well.

With default settings, the attribute doesn't work with tables that contain an identity column. To use the attribute on these tables, you should set to true the `UpdateOnly` property of the `columns` parameter in the `PrepareInsert()` method.

The logic of the `PXAccumulator` attribute works on saving of the inserted data records to the database. This process is implemented in the *PersistInserted()* method of the cache. This methods detects the `PXAccumulator`-derived attribute and calls the `PersistInserted()` method defined in this attribute.

When you update a data record using the attribute, you typically initialize a new instance of the DAC, set the key fields to the key values of the data record you need to update, and insert it into the cache. When a user saves changes on the webpage, or you save changes from code, your custom attribute processes these inserted data records in its own way, updating database records instead of inserting new redords and applying the policies you specify.

By deriving from this attribute, you can implement an attribute that will prevent certain fields from further updates once they are initialized with values.

Examples

The code below shows how the attribute can be used directly. When a data record is saved, value of every field from the first array will be added to the previously saved value of the corresponding field from the second array. That is, `FinYtdBalance` values will be accumulated in the `FinBegBalance` value, `TranYtdBalance` values in the `TranBegBalance` value, and so on.

```
[PXAccumulator(
    new Type[] {
        typeof(CuryAPHistory.finYtdBalance),
        typeof(CuryAPHistory.tranYtdBalance),
        typeof(CuryAPHistory.curyFinYtdBalance),
        typeof(CuryAPHistory.curyTranYtdBalance)
    },
    new Type[] {
        typeof(CuryAPHistory.finBegBalance),
        typeof(CuryAPHistory.tranBegBalance),
        typeof(CuryAPHistory.curyFinBegBalance),
        typeof(CuryAPHistory.curyTranBegBalance)
    }
)]
[Serializable]
public partial class CuryAPHist : CuryAPHistory
{ ... }
```

In the following example, the class derived from `PXAccumulatorAttribute` overrides the `PrepareInsert()` method and specifies the assignment behavior for several fields.

```
public class SupplierDataAccumulatorAttribute : PXAccumulatorAttribute
{
    public SupplierDataAccumulatorAttribute()
    {
        base._SingleRecord = true;
    }

    protected override bool PrepareInsert(PXCache sender, object row,
                                          PXAccumulatorCollection columns)
    {
        if (!base.PrepareInsert(sender, row, columns))
            return false;

        SupplierData bal = (SupplierData)row;
        columns.Update<SupplierData.supplierPrice>(
            bal.SupplierPrice, PXDataFieldAssign.AssignBehavior.Initialize);
        columns.Update<SupplierData.supplierUnit>(
            bal.SupplierUnit, PXDataFieldAssign.AssignBehavior.Initialize);
```

```
        columns.Update<SupplierData.conversionFactor>(
            bal.ConversionFactor, PXDataFieldAssign.AssignBehavior.Initialize);
        columns.Update<SupplierData.lastSupplierPrice>(
            bal.LastSupplierPrice, PXDataFieldAssign.AssignBehavior.Replace);
        columns.Update<SupplierData.lastPurchaseDate>(
            bal.LastPurchaseDate, PXDataFieldAssign.AssignBehavior.Replace);

        return true;
    }
}
```

The custom attribute is then applied to a DAC as follows.

```
[System.SerializableAttribute()]
[SupplierDataAccumulator]
public class SupplierData : PX.Data.IBqlTable
{ ... }
```

Related Types

- *PXDataFieldAssign.AssignBehavior Enumeration*

PXAccumulator Attribute Constructors

The *PXAccumulator* attribute exposes the following constructors.

PXAccumulatorAttribute()

Empty default constructor.

Syntax:

```
public PXAccumulatorAttribute()
```

PXAccumulatorAttribute(Type[], Type[])

Initializes an instance of the attribute with the source fields and destination fields.

For example, a source field may be the transaction amount and the destination field the current balance.

Syntax:

```
public PXAccumulatorAttribute(Type[] source, Type[] destination)
```

Parameters:

- `source`

  Fields whose values are summarized in the corresponding destination fields.
- `destination`

  Fields that store sums of source fields from the data records inserted into the database previously to the current data record.

PXDataFieldAssign.AssignBehavior Enumeration

Defines possible policies of assigning a value to a DAC field. The enumeration declaration nests in the `PXDataFieldAssign` class.

Syntax

```
public class PXDataFieldAssign : PXDataFieldParam
{
```

```
    public enum AssignBehavior {...}
}
```

Members

- `Replace`

  The new value is inserted into the data field, and the previous value is overwritten.
- `Summarize`

  The new value is added to the value stored in the database.
- `Maximize`

  The maximum of the new value and the value from the database is saved in the database.
- `Minimize`

  The minimum of the new value and the value from the database is saved in the database.
- `Initialize`

  The new value is saved in the database as the value if the field does not have a value in the database. If the data field is not null, the new value is discarded.

Remarks

The enumeration is typically used in the methods of the *PXAccumulator* attribute and its successors.

PXHidden Attribute

Hides the data access class (DAC), the business logic controller (graph), or the view from the selectors of DACs and graphs and from the Web Service API clients.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Field |
                AttributeTargets.Assembly, AllowMultiple = true)]
public sealed class PXHiddenAttribute : Attribute
```

Properties

- `public bool` **`ServiceVisible`**

  Gets or sets the value that indicates whether the object marked with the attribute is visible to the Web Service API (in particular, to the Report Designer). By default, default the property equals `false`, and the object is hidden from all selectors.

Remarks

You can the attribute either on the declaration of a DAC, a graph, or a view. You can hide the object from everything but the Web Service API by placing the attribute on the object declaration and setting the `ServiceVisible` property to `true`.

Examples

In the example below, the attribute is placed on the DAC declaration.

```
[Serializable]
[PXHidden]
public partial class ActivitySource : IBqlTable { ... }
```

In the example below, the attribute is placed on the graph declaration.

```
[PXHidden()]
public class CAReleaseProcess : PXGraph<CAReleaseProcess> { ... }
```

In the example below, the attribute is placed on the view declaration in some graph.

```
[PXHidden]
public PXSelect<CurrencyInfo> CurrencyInfoSelect;
```

Attributes on Actions

The following attributes set up the button that will represent an action in the user interface:

- *PXButton*

  The base attribute for all other attributes that configure buttons. The successor attributes only set base class properties to specific values.
- *PXSaveButton*
- *PXSaveCloseButton*
- *PXCancelButton*
- *PXCancelCloseButton*
- *PXInsertButton*
- *PXDeleteButton*
- *PXFirstButton*
- *PXPreviousButton*
- *PXNextButton*
- *PXLastButton*
- *PXSendMailButton*
- *PXReplyMailButton*
- *PXForwardMailButton*
- *PXTemplateMailButton*
- *PXLookupButton*
- *PXProcessButton*

Also, you can use the *PXUIField* attribute to configure the button layout and set access rights.

PXButton Attribute

Sets up a button that is used to initiate the action in the user interface.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`

Syntax

```
public class PXButtonAttribute : PXEventSubscriberAttribute,
                                 IPXFieldSelectingSubscriber
```

Properties

- `public bool` **`ShortcutCtrl`**

  Gets or sets the value that indicates whether the keyboard shortcut for the button includes the *Ctrl* key.
- `public bool` **`ShortcutShift`**

  Gets or sets the value that indicates whether the keyboard shortcut for the button includes the *Shift* key.
- `public char` **`ShortcutChar`**

  Gets or sets the character that is used as the keyboard shorcut for the button. Setting additionally the `ShortcutCtrl` and `ShortcutShift` properties adds or removes *Ctrl* and *Shift* keys to and from the shortcut.
- `public PXSpecialButtonType` **`SpecialType`**

  Gets or sets the *PXSpecialButtonType* value that indicates whether a button has a special type, such as `Save`, `Cancel`, or `Refresh`, or does not have. A button of a special type may be searched, for instance, by graph methods in special occassions (the `PressSave()` method searches visible buttons of `Save` type and selects the first of them). By default, the property is set to `PXSpecialButtonType.Default`.
- `public PXSpecialButtonType` **`OnClosingPopup`**

  Gets or sets the special type of the button that will be triggerred on closing of an application webpage that is opened in popup mode.
- `public bool` **`ClosePopup`**

  Gets or sets the value that indicates whether the enclosing popup is closed once the button logic is executed.
- `public bool` **`PopupVisible`**

  Gets or sets the value that indicates whether the button is visible when the enclosing webpage is opened in popup mode.
- `public bool` **`CommitChanges`**

  Gets or sets the value that indicates whether a button press posts modifications to the server.
- `public string` **`ImageSet`**

  Gets or sets the value that identifies the image set. Forms the first part of the button image URL.
- `public string` **`ImageKey`**

Gets or sets the value that identifies the button image within the set specified by `ImageSet`. Forms the second part of the button image URL.

- `public string` **`ImageUrl`**

  Gets or sets the URL of the image displayed on the button when it is enabled.

- `public string` **`DisabledImageUrl`**

  Gets or sets the URL of the image displayed on the button when it is disabled.

- `public string` **`HoverImageUrl`**

  Gets or sets the URL of the image displayed on the enabled button on hover.

- `public string` **`Tooltip`**

  Gets or sets the string displayed as a tooltip for the button.

- `public PXConfirmationType` **`ConfirmationType`**

  Gets or sets the *PXConfirmationType* value that indicates in what cases the confirmation message is shown to a user on a button press. By default, the property is set to `PXConfirmationType.Unspecified`.

- `public string` **`ConfirmationMessage`**

  Gets or sets the confirmation message that can be shown to a user on a button press. The cases when the configramtion message is shown depend on `ConfirmationType`.

- `public bool` **`MenuAutoOpen`**

  Gets or sets the value that indicates whether a button press only expands the menu with other buttons. If `true`, the button press opens the menu and does not trigger button's action.

Constructors

- `public PXButtonAttribute() : base()`

  Create an instance of the attribute.

Remarks

This attribute should be placed on the declaration of the method that implements the action.

Through attribute's parameters, you can configure some properties of the button, such `ImageUrl`, `ShortcutChar`, and `Tooltip`. To configure other layout properties, use the *PXUIField* attribute, such as `DisplayName`, `Visible`, or `Enabled`. Still some other properties can be set only on an ASPX page.

A number of other attributes derive from the `PXButton` attributes. These attribute do not implement additional logic and only set certain properties to specific values.

Examples

An example of using the attribute without parameters is given below.

```
// Action declaration in a graph
public PXAction<ApproveBillsFilter> ViewDocument;

// Action implementation in a graph
[PXUIField(DisplayName = "View Document",
           MapEnableRights = PXCacheRights.Update,
           MapViewRights = PXCacheRights.Update)]
[PXButton]
public virtual IEnumerable viewDocument(PXAdapter adapter) { ... }
```

In the example below the button is disabled by default (it can be *enabled* in code). Also, the `ImageKey` property sets a specific image to be displayed on the button.

```
public PXAction<VendorR> viewCustomer;

[PXUIField(DisplayName = Messages.ViewCustomer,
           Enabled = false, Visible = true,
           MapEnableRights = PXCacheRights.Select,
           MapViewRights = PXCacheRights.Select)]
[PXButton(ImageKey = PX.Web.UI.Sprite.Main.Process)]
public virtual IEnumerable ViewCustomer(PXAdapter adapter) { ... }
```

In the example below, the attribute provides specific URLs of the images displayed on the button by default (`ImageUrl`) when it is disabled (`DisabledImageUrl`). The tooltip is also set.

```
public PXAction<EPActivity> CancelSending;

[PXUIField(DisplayName = EP.Messages.CancelSending, MapEnableRights =
 PXCacheRights.Select)]
[PXButton(ImageUrl = "~/Icons/Cancel_Active.gif",
          DisabledImageUrl = "~/Icons/Cancel_NotActive.gif",
          Tooltip = EP.Messages.CancelSendingTooltip)]
public virtual void cancelSending() { ... }
```

Related Types

- *PXSpecialButtonType Enumeration*
- *PXConfirmationType Enumeration*

PXSpecialButtonType Enumeration

Defines possible special types of a button. The enumeration is used to set `PXButton` attribute properties.

Members

- `Default`

  The button does not have a special type.
- `Save`

  The button has the **Save** button type. In particular, a graph searches buttons of this type when the graph's`Actions.PressSave()` method is invoked.
- `Cancel`

  The button has the **Cancel** button type. In particular, a graph searches buttons of this type when the graph's `Actions.PressCancel()` method is invoked.
- `Refresh`

  The button has the **Refresh** button type.

PXConfirmationType Enumeration

Defines values that indicate cases when the confirmation message is shown on a button press. The message box typically asks a user to confirm the action.

Members

- `Always`

  Always show the message box.
- `IfDirty`

Show the message box when there are unsaved changes on the webpage.

- `Unspecified`

  Whether to show the message box is not specified.

PXSaveButton Attribute

Sets up a button with the properties of the **Save** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXSaveButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXSaveButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute.

  - `CommitChanges` to `true`
  - `SpecialType` to `PXSpecialButtonType.Save`
  - *Ctrl* + *S* as the keyboard shortcut

  Also sets the image and the tooltip.

Examples

```
public PXAction<INPIHeader> save;

[PXSaveButton]
protected virtual IEnumerable Save(PXAdapter adapter) { ... }
```

PXSaveCloseButton Attribute

Sets up a button with the properties of the **Save and Close** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
    PXSaveButtonAttribute
```

Syntax

```
public class PXSaveCloseButtonAttribute : PXSaveButtonAttribute
```

Constructors

- `public PXSaveCloseButtonAttribute()`

  Create an instance of the attribute. In addition to properties that are set by the base *PXSaveButton* attribute, extends the keyboard shortcut with *Shift* ans sets the different tooltip.

PXInsertButton Attribute

Sets up a button with the properties of the **Add New Record** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXInsertButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXInsertButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute.

  - `PopupVisible` to `false`
  - *Ctrl* + - as the keyboard shortcut

  Also sets the image, the tooltip, and the confirmation message.

Examples

```
public PXAction<INPIHeader> Insert;

[PXInsertButton]
protected virtual IEnumerable insert(PXAdapter adapter) { ... }
```

PXCancelButton Attribute

Sets up a button with the properties of the **Cancel** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXCancelButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXCancelButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute:

  - `ClosePopup` to `false`
  - `SpecialType` to `PXSpecialButtonType.Cancel`
  - `ConfirmationType` to `PXConfirmationType.IfDirty`
  - *Ctrl* + - as the keyboard shortcut

  Also sets the image, the tooltip, and the confirmation message.

Examples

```
public PXAction<CashAccount> cancel;

[PXUIField(DisplayName = ActionsMessages.Cancel, MapEnableRights =
 PXCacheRights.Select)]
[PXCancelButton]
protected virtual IEnumerable Cancel(PXAdapter adapter) { ... }
```

PXCancelCloseButton Attribute

Sets up a button with the properties of the **Cancel and Close** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
    PXCancelButtonAttribute
```

Syntax

```
public class PXCancelCloseButtonAttribute : PXCancelButtonAttribute
```

Constructors

- `public PXCancelCloseButtonAttribute() : base()`

  Create an instance of the attribute. In addition to properties that are set by the base *PXCancelButton* attribute, sets the different tooltip.

PXDeleteButton Attribute

Sets up a button with the properties of the **Delete** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXDeleteButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXDeleteButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute:

  - `ClosePopup` to `true`
  - `ConfirmationType` to `PXConfirmationType.Always`
  - *Ctrl + .* as the keyboard shortcut

  Also sets the image, the tooltip, and the confirmation message.

Examples

```
public PXAction<CARecon> delete;
```

```
[PXDeleteButton]
[PXUIField]
protected virtual IEnumerable Delete(PXAdapter a) { ... }
```

PXFirstButton Attribute

Sets up a button with the properties of the **Go to First Record** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXFirstButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXFirstButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute:

  - `PopupVisible` to `false`
  - *Ctrl* + *!* as the keyboard shortcut

  Also sets the image, the tooltip, and the confirmation message.

Examples

```
public PXAction<CuryRateFilter> first;

[PXFirstButton]
[PXUIField]
protected virtual IEnumerable First(PXAdapter a) { ... }
```

PXPreviousButton Attribute

Sets up a button with the properties of the **Go to Previous Record** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXPreviousButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXPreviousButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute:

  - `PopupVisible` to `false`
  - *!* as the keyboard shortcut

Also sets the image, the tooltip, and the confirmation message.

Examples

```
public PXAction<APDocumentFilter> previousPeriod;

[PXUIField(DisplayName = "Prev",
           MapEnableRights = PXCacheRights.Select,
           MapViewRights = PXCacheRights.Select)]
[PXPreviousButton]
public virtual IEnumerable PreviousPeriod(PXAdapter adapter) { ... }
```

PXNextButton Attribute

Sets up a button with the properties of the **Go to Next Record** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXNextButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXNextButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute:

  - `PopupVisible` to `false`
  - '' as the keyboard shortcut

  Also sets the image, the tooltip, and the confirmation message.

Examples

```
public PXAction<APDocumentFilter> nextPeriod;

[PXUIField(DisplayName = "Next",
           MapEnableRights = PXCacheRights.Select,
           MapViewRights = PXCacheRights.Select)]
[PXNextButton]
public virtual IEnumerable NextPeriod(PXAdapter adapter) { ... }
```

PXLastButton Attribute

Sets up a button with the properties of the **Go to Last Record** button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXLastButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXLastButtonAttribute() : base()`

  Create an instance of the attribute, setting the properties of the *PXButton* attribute:

  - `PopupVisible` to `false`
  - *Ctrl* + *"* as the keyboard shortcut

  Also sets the image, the tooltip, and the confirmation message.

Examples

```
public PXAction<AP1099YearMaster> lastVendor;

[PXUIField(DisplayName = "Last",
           MapEnableRights = PXCacheRights.Select,
           MapViewRights = PXCacheRights.Select)]
[PXLastButton]
public virtual IEnumerable LastVendor(PXAdapter adapter) { ... }
```

PXSendMailButton Attribute

Sets up a button with the properties of the button that sends an email.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXSendMailButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXSendMailButtonAttribute() : base()`

  Create an instance of the attribute, setting the specific tooltip.

Examples

```
public PXAction<EPActivity> Send;

[PXUIField(DisplayName = Messages.Send,
           MapEnableRights = PXCacheRights.Select)]
[PXSendMailButton]
protected virtual IEnumerable send(PXAdapter adapter) { ... }
```

PXReplyMailButton Attribute

Sets up a button with the properties of the button that replies to an email.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXReplyMailButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXReplyMailButtonAttribute() : base()`

  Create an instance of the attribute, setting the specific tooltip.

Examples

```
public PXAction<EmailFilter> Reply;

[PXUIField(DisplayName = Messages.Reply)]
[PXReplyMailButton]
protected void reply() { ... }
```

PXForwardMailButton Attribute

Sets up a button with the properties of the button that forwards an email.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXForwardMailButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXForwardMailButtonAttribute() : base()`

  Create an instance of the attribute, setting the specific tooltip.

Examples

```
public PXAction<EmailFilter> Forward;

[PXUIField(DisplayName = Messages.Forward)]
[PXForwardMailButton]
protected void forward() { ... }
```

PXTemplateMailButton Attribute

Sets up a button with the specific properties.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXTemplateMailButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXTemplateMailButtonAttribute()`

  Create an instance of the attribute, setting the image and the tooltip.

PXLookupButton Attribute

Sets up a button with the properties of the lookup button.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXLookupButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXLookupButtonAttribute() : base()`

  Create an instance of the attribute, setting the image.

Examples

```
public PXAction<APInvoice> newVendor;

[PXUIField(DisplayName = "New Vendor",
           MapEnableRights = PXCacheRights.Select,
           MapViewRights = PXCacheRights.Select)]
[PXLookupButton]
public virtual IEnumerable NewVendor(PXAdapter adapter) { ... }
```

PXProcessButton Attribute

Sets up a button with the properties of buttons that are used on processing screens.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXButtonAttribute
```

Syntax

```
public class PXProcessButtonAttribute : PXButtonAttribute
```

Constructors

- `public PXProcessButtonAttribute() : base()`

  Create an instance of the attribute, setting the `CommitChanges` property of the *PXButton* attribute to `true`.

Examples

```
public PXAction<APInvoice> createSchedule;

[PXUIField(DisplayName = "Assign to Schedule",
```

```
            MapEnableRights = PXCacheRights.Update,
            MapViewRights = PXCacheRights.Update)]
[PXProcessButton(ImageKey = PX.Web.UI.Sprite.Main.Shedule)]
public virtual IEnumerable CreateSchedule(PXAdapter adapter) { ... }
```

Attributes on Data Views

You can place the following attributes on the declaration of a data view in a graph:

- *PXFilterable*

  Adds the control that lets a user create filters and save them in the database. The control is added to the grid that uses the data view to retrieve data.
- *PXImport*

  Adds the grid toobar button that allows the user to load data from the file to the grid. The attribute is placed on the data view the grid uses to retrieve the data.
- *PXPreview*
- *PXEmailLoadTemplate*
- *PXHidden*

  Hides the data view from the selectors of DACs and graphs and from the Web Service API clients.
- *PXCopyPasteHiddenView*

  Indicates that the cache corresponding to the primary DAC of the data view is not copied when the copy-paste feature is utilized on the webpage.
- *PXCopyPasteHiddenFields*

  Indicates that the specific fields of the primary DAC of the data view are not copied when the copy-paste feature is utilized on the webpage.

PXFilterable Attribute

Placed on the view declaration, adds the control that lets a user create filters and save them in the database. The control is added to the grid that uses the view to retrieve data.

Inheritance Hierarchy

```
PXViewExtensionAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class PXFilterableAttribute : PXViewExtensionAttribute
```

Constructors

- `public PXFilterableAttribute(params Type[] autoFill)`

  Initializes a new instance of the attribute. The parameters are optional and are not used in most cases (you can specify the DACs whose `Current` objects will be used to fill the filter parameters before showing it to the user).

Remarks

The attribute is placed on the view declaration.

If you specify this view as the data member of a grid control, the grid will include a control which can be used to create filters and save them in the database. A filter is a set of conditions checked for the fields selected by the view. When a grid applies a filter it displays only the data records that satisfy the filter's conditions.

Examples

```
[PXFilterable]
public PXSelect<APInvoice> APDocumentList;
```

PXViewName Attribute

Defines the user-friendly name of the view.

Inheritance Hierarchy

```
Attribute
  PXNameAttribute
```

Syntax

```
public class PXViewNameAttribute : PXNameAttribute
```

Constructors

- `public PXViewNameAttribute(string name) : base(name)`

  Initializes a new instance of the attribute that sets the provided string as the view name.

  Parameters:

  - `name`

    The string used as the user-friendly name of the view.

Remarks

The attribute is added to the view declaration.

Examples

```
[PXViewName(Messages.Orders)]
public PXSelectReadonly<SOOrder,
    Where<SOOrder.customerID, Equal<Current<BAccount.bAccountID>>>>
    Orders;
```

Here `Messages.Orders` is a constant defined by the application.

PXImport Attribute

Adds the grid toobar button that allows the user to load data from the file to the grid. The attribute is placed on the view the grid uses to retrieve the data.

Inheritance Hierarchy

```
PXViewExtensionAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class PXImportAttribute : PXViewExtensionAttribute
```

Constructors

Constructor	Description
PXImportAttribute(Type)	Initializes a new instance of the attribute
PXImportAttribute(Type, IPXImportWizard)	Initializes a new instance of the attribute

Static Methods

Method	Description
SetEnabled(PXGraph, string, bool)	Enables or disables the control which the attribute adds to the grid

IPXPrepareItems Interface

Defines methods that can be implemented by the graph to control the data import.

Syntax:

```
public interface IPXPrepareItems
```

Methods:

- `bool PrepareImportRow(string viewName, IDictionary keys, IDictionary values)`

  Prepares a record from the imported file for convertion into a DAC instance.

  Parameters:

  - `viewName`

    The name of the view that is marked with the attribute.

  - `keys`

    The keys of the data to import.

  - `values`

    The values corresponding to the keys.

- `bool RowImporting(string viewName, object row)`

  Implements the logic executed before the insertion of a data record into the cache.

  Parameters:

  - `viewName`

    The name of the view that is marked with the attribute.

  - `row`

    The record to import as a DAC instance.

- `bool RowImported(string viewName, object row, object oldRow)`

  Implements the logic executed after the insertion of a data record into the cache.

Parameters:

- `viewName`

  The name of the view that is marked with the attribute.
- `row`

  The imported record as a DAC instance.

- `void PrepareItems(string viewName, IEnumerable items)`

  Verifying the imported items before they are saved in the database.

  Parameters:

  - `viewName`

    The name of the view that is marked with the attribute.
  - `items`

    The collection of objects to import as instances of the DAC.

Remarks

The attribute placed on the view declaration in the graph. As a result, a grid that uses the view as a data provider will include a button that opens the data import wizard. Using this wizard, the user can load data from an Excel or *.cvs* file to the grid.

You can control all steps of data import by having the graph implement the `PXImportAttribute.IPXPrepareItems` interface.

Examples

The attibute below adds the upload button to the toolbar of the grid that will use the `Transactions` view as a data provider.

```
// Primary view declaration
public PXSelect<INRegister,
    Where<INRegister.docType, Equal<INDocType.adjustment>>> adjustment;
...

[PXImport(typeof(INRegister))]
public PXSelect<INTran,
          Where<INTran.docType, Equal<Current<INRegister.docType>>,
              And<INTran.refNbr, Equal<Current<INRegister.refNbr>>>>>
       Transactions;
```

In this example, the primary view DAC is `INRegister`, and it is passed to the attribute as a parameter.

In the following example, the graph implements the `PXImportAttribute.IPXPrepareItems` interface to control the data import.

```
public class APInvoiceEntry : APDataEntryGraph<APInvoiceEntry, APInvoice>,
                              PXImportAttribute.IPXPrepareItems
{
    ...
    // The attribute is placed on the view declaration
    [PXImport(typeof(APInvoice))]
    public PXSelectJoin<APTran,
        LeftJoin<POReceiptLine,
            On<POReceiptLine.receiptNbr, Equal<APTran.receiptNbr>,
            And<POReceiptLine.lineNbr, Equal<APTran.receiptLineNbr>>>>,
        Where<APTran.tranType, Equal<Current<APInvoice.docType>>,
            And<APTran.refNbr, Equal<Current<APInvoice.refNbr>>>>,
        OrderBy<Asc<APTran.tranType,
                Asc<APTran.refNbr, Asc<APTran.lineNbr>>>>>
```

```
        Transactions;
    ...

    // Implementation of the IPXPrepareItems methods
    public virtual bool PrepareImportRow(
        string viewName, IDictionary keys, IDictionary values)
    {
        if (string.Compare(viewName, "Transactions", true) == 0)
        {
            if (values.Contains("tranType")) values["tranType"] =
                Document.Current.DocType;
            else values.Add("tranType", Document.Current.DocType);
            if (values.Contains("tranType")) values["refNbr"] =
                Document.Current.RefNbr;
            else values.Add("refNbr", Document.Current.RefNbr);
        }
        return true;
    }

    public bool RowImporting(string viewName, object row)
    {
        return row == null;
    }

    public bool RowImported(string viewName, object row, object oldRow)
    {
        return oldRow == null;
    }

    public virtual void PrepareItems(string viewName, IEnumerable items) { }
    ...
}
```

PXImport Attribute Constructors

The *PXImport* attribute exposes the following constructors.

PXImportAttribute(Type)

Initializes a new instance of the attribute. The parameter is set the primary view DAC.

Syntax:

```
public PXImportAttribute(Type primaryTable)
```

Parameters:

- `primaryTable`

  The first DAC that is referenced by the primary view of the graph where the current view is declared.

PXImportAttribute(Type, IPXImportWizard)

Initializes a new instance of the attribute. The first parameter is the primary table of the view the attribute is attached to.

Syntax:

```
public PXImportAttribute(Type primaryTable, IPXImportWizard importer) :
            this(primaryTable)
```

Parameters:

- `primaryTable`

  The first table that is referenced in the view (primary table).

- `importer`

  The object implementing the `IPXImportWizard` interface.

PXImport Attribute Methods

The *PXImport* attribute exposes the following static methods.

SetEnabled(PXGraph, string, bool)

Enables or disables the control which the attribute adds to the grid.

Syntax:

```
public static void SetEnabled(PXGraph graph, string viewName, bool isEnabled)
```

Parameters:

- `graph`

  The graph where the view marked with the attribute is defined.

- `viewName`

  The name of the view that is marked with the attribute.

- `isEnabled`

  The value that indicates whether the method enables or disables the control.

PXPreview Attribute

Inheritance Hierarchy

```
PXViewExtensionAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class PXPreviewAttribute : PXViewExtensionAttribute
```

Constructors

Constructor	Description
PXPreviewAttribute(Type)	
PXPreviewAttribute(Type, Type)	

PXPreview Attribute Constructors

The *PXPreview* attribute exposes the following constructors.

PXPreviewAttribute(Type)

Syntax:

```
public PXPreviewAttribute(Type primaryViewType) : this(primaryViewType, null) { }
```

PXPreviewAttribute(Type, Type)

Syntax:

```
public PXPreviewAttribute(Type primaryViewType, Type previewType)
```

PXEmailLoadTemplate Attribute

Inheritance Hierarchy

```
PXViewExtensionAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class PXEmailLoadTemplateAttribute : PXViewExtensionAttribute
```

Properties

- `public Type` **`ContentField`**

  Get, set.
- `public Type` **`ReferenceField`**

  Get, set.
- `public Type` **`PrimaryView`**

  Get.

Constructors

- `public PXEmailLoadTemplateAttribute(Type primaryView)`

Miscellaneous

This chapter includes the following attributes, which are not related to other groups of attributes:

- *PXDisableCloneAttributes*
- *PXDynamicAggregate*
- *PXDynamicMask*
- *CloseBrackets*
- *DashboardType*
- *DashboardVisible*
- *IncomingMailProtocols*
- *OpenBrackets*
- *OperationList*
- *PXAggregate*
- *PXAttributeFamily*
- *PXAutomationMenu*
- *PXAutoSave*
- *PXBreakInheritance*

- *PXCheckUnique*
- *PXCompositeKey*
- *PXCopyPasteHiddenFields*
- *PXCopyPasteHiddenView*
- *PXCultureSelector*
- *PXCustomization*
- *PXCustomStringList*
- *PXDACDescription*
- *PXDBDataLength*
- *RowCondition*
- *RowNbr*
- *SSlRequest*
- *TypeDelete*
- *PXEMailAccountIDSelector*
- *PXEMailSource*
- *PXEntityName*
- *PXEnumDescription*
- *PXExtension*
- *PXFeature*
- *PXFontList*
- *PXFontSizeList*
- *PXFontSizeStrList*
- *PXLineNbrMarker*
- *PXName*
- *PXNotCleanable*
- *PXNoteText*
- *PXNotPersistable*
- *PXNoUpdate*
- *PXNubmerSeparatorListAttribure*
- *PXOffline*
- *PXOverride*
- *PXPhoneValidation*
- *PXRefNote*
- *PXRefNoteSelector*
- *PXRateSync*
- *PXShortCut*
- *PXSplitRow*

- *PXStandartDateTimeFormatSelector*
- *PXSubstitute*
- *PXSuppressEventValidation*
- *PXSurrogate*
- *PXTableName*
- *PXTimeZone*
- *PXVirtual*
- *PXVirtualDAC*
- *PXZipValidation*
- *ReportView*

PXDisableCloneAttributes Attribute

Inheritance Hierarchy

```
Attribute
  PXClassAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXDisableCloneAttributesAttribute : PXClassAttribute
```

PXDynamicAggregate Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowSelectingSubscriber`
- `IPXRowSelectedSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public sealed class PXDynamicAggregateAttribute :
    PXEventSubscriberAttribute,
    IPXRowSelectingSubscriber,
    IPXRowSelectedSubscriber,
```

PXDynamicMask Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Parameter |
                AttributeTargets.Method)]
public class PXDynamicMaskAttribute : PXEventSubscriberAttribute,
                                      IPXFieldSelectingSubscriber
```

Properties

- `public virtual string` **`DefaultMask`**

  Get, set.

Constructors

- `public PXDynamicMaskAttribute(Type maskSearch)`

CloseBrackets Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public sealed class CloseBracketsAttribute : PXIntListAttribute
```

Properties

- `public static string[]` **`Labels`**

  Get.

- `public static int[]` **`Values`**

  Get.

Constructors

- `public CloseBracketsAttribute() : base(Values, Labels)`

DashboardType Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
```

```
public class DashboardTypeAttribute : Attribute
```

Properties

- `public enum` **`Type`**

Constructors

- `public DashboardTypeAttribute(params int[] type)`

DashboardVisible Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public sealed class DashboardVisibleAttribute : PXEventSubscriberAttribute
```

Properties

- `public bool` **`Visible`**

  Get.

Constructors

Constructor	Description
DashboardVisibleAttribute()	
DashboardVisibleAttribute(bool)	

DashboardVisible Attribute Constructors

The *DashboardVisible* attribute exposes the following constructors.

DashboardVisibleAttribute()

Syntax:

```
public DashboardVisibleAttribute() : this(true) { }
```

DashboardVisibleAttribute(bool)

Syntax:

```
public DashboardVisibleAttribute(bool visible)
```

IncomingMailProtocols Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public class IncomingMailProtocolsAttribute : PXIntListAttribute
```

Constructors

- `public IncomingMailProtocolsAttribute() : base(`

OpenBrackets Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public sealed class OpenBracketsAttribute : PXIntListAttribute
```

Properties

- `public static string[]` **`Labels`**

  Get.

- `public static int[]` **`Values`**

  Get.

Constructors

- `public OpenBracketsAttribute() : base(Values, Labels)`

OperationList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public class OperationListAttribute: PXIntListAttribute
```

Constructors

- `public OperationListAttribute(): base`

PXAggregate Attribute

The type used to combine multiple attributes in one, which is derived from this attribute.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Syntax

```
public class PXAggregateAttribute : PXEventSubscriberAttribute
```

Properties

- `public override Type` **`BqlTable`**

  Gets or sets the DAC associated with the attribute. The setter also sets the provided value as `BqlTable` in all attributes combined in the current attribute.
- `public override string` **`FieldName`**

  Gets or sets the name of the field associtated with the attribute. The setter also sets the provided value as `FieldName` in all attributes combined in the current attribute.
- `public override int` **`FieldOrdinal`**

  Gets or sets the index of the field associtated with the attribute. The setter also sets the provided value as `FieldOrdinal` in all attributes combined in the current attribute.

Fields

- `protected List<PXEventSubscriberAttribute>` **`_Attributes`**

  The collection of the attributes combined in the current attribute.

Constructors

- `public PXAggregateAttribute()`

  Initializes a new instance of the attribute; pulls the `PXEventSubscriberAttribute`-derived attributes placed on the current attribute and adds them to the collection of aggregated attributes.

PXAttributeFamily Attribute

Allows to specify rules, which attributes can not be combined together.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited = false)]
public class PXAttributeFamilyAttribute: Attribute
```

Constructors

Constructor	Description
FromType(Type)	
PXAttributeFamilyAttribute(Type)	

Static Methods

Method	Description
CheckAttributes(PropertyInfo, PXEventSubscriberAttribute[])	
GetRoots(Type)	

PXAttributeFamily Attribute Constructors

The *PXAttributeFamily* attribute exposes the following constructors.

FromType(Type)

Syntax:

```
public static PXAttributeFamilyAttribute FromType(Type t)
```

PXAttributeFamilyAttribute(Type)

Syntax:

```
public PXAttributeFamilyAttribute(Type rootType)
```

PXAttributeFamily Attribute Methods

The *PXAttributeFamily* attribute exposes the following static methods.

CheckAttributes(PropertyInfo, PXEventSubscriberAttribute[])

Syntax:

```
public static void CheckAttributes(PropertyInfo prop, PXEventSubscriberAttribute[]
 attributes)
```

GetRoots(Type)

Syntax:

```
public static Type[] GetRoots(Type t)
```

PXAutomationMenu Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXAggregateAttribute
```

Interfaces

- `IPXRowSelectedSubscriber`

Syntax

```
[PXDBString]
[PXDefault(Undefined)]
[PXUIField(DisplayName = "Action")]
[PXStringList(new string[]
{ Undefined }, new string[]
{ Undefined })]
public class PXAutomationMenuAttribute : PXAggregateAttribute,
                                          IPXRowSelectedSubscriber
```

Properties

- `public string` **`DisplayName`**

  Get, set.
- `public bool` **`Visible`**

  Get, set.

Constructors

- `public PXAutomationMenuAttribute() : base()`

Nested Classes

- `public class undefinded : Constant<string> : base(Undefined)`

  Constructors

  - `public undefinded()`

PXAutoSave Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class PXAutoSaveAttribute : Attribute
```

PXBreakInheritance Attribute

When placed on a derived data access class (DAC), indicates that the cache objects corresponding to the base DACs should not be instantiated.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
public sealed class PXBreakInheritanceAttribute : Attribute
```

Examples

In the example below, the attribute prevents instantiation of the `INItemStats` cache during instantiation of the `INItemStatsTotal` cache.

```
[PXBreakInheritance]
[Serializable]
public partial class INItemStatsTotal : INItemStats
{
    ...
}
```

PXCheckUnique Attribute

Ensures that a DAC field has distinct values in all data records in a given context.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowInsertingSubscriber`
- `IPXRowUpdatingSubscriber`
- `IPXRowPersistingSubscriber`

Syntax

```
public class PXCheckUnique : PXEventSubscriberAttribute,
                             IPXRowInsertingSubscriber,
                             IPXRowUpdatingSubscriber,
                             IPXRowPersistingSubscriber
```

Constructors

- `public PXCheckUnique(params Type[] fields)`

  Initializes a new instance of the attribute. The parameter is optional.

Remarks

The attribute is placed on the declaration of a DAC field, and ensures that this field has a unique value within the current context.

The functionality of the attribute can be implemented through other ways. The use of the attribute for imposing constraint of a key field is obsolete. You should use the `IsKey` property of the data type attribute for this purpose.

Examples

```
[PXDBString(30, IsKey = true)]
[PXUIField(DisplayName = "Mailing ID")]
[PXCheckUnique]
public override string NotificationCD { get; set; }
```

PXCompositeKey Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowSelectingSubscriber//`
- `IPXFieldVerifyingSubscriber`

Syntax

```
public class PXCompositeKeyAttribute : PXEventSubscriberAttribute,
                                       IPXRowSelectingSubscriber//,
                                       IPXFieldVerifyingSubscriber
```

PXCopyPasteHiddenFields Attribute

Indicates that the specific fields of the primary DAC of the data view are not copied when the copy-paste feature is utilized on the webpage.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class PXCopyPasteHiddenFieldsAttribute: Attribute
```

Constructors

- `public PXCopyPasteHiddenFieldsAttribute(params Type[] fields )`

  Initializes a new instance of the attribute that prevent the specified DAC fields from

Static Methods

Method	Description
IsDefined(PXGraph, string, string)	Determines whether the provided graph defines a data view with the given name and this data view is marked with the `PXCopyPasteHiddenFields` attribute referencing the field.

Remarks

See the *PXCopyPasteHiddenView* attribute for more detail.

Examples

The code below prevents only the `InvoiceNbr` field of the `APInvoice` DAC from copying when a user clicks **Copy** on the webpage.

```
[PXCopyPasteHiddenFields(typeof(APInvoice.invoiceNbr))]
public PXSelectJoin<APInvoice,
```

```
    LeftJoin<Vendor, On<Vendor.bAccountID, Equal<APInvoice.vendorID>>>>
        Document;
```

Multiple fields can be listed, as the following code shows.

```
[PXCopyPasteHiddenFields(typeof(GLTranDoc.parentLineNbr),
                         typeof(GLTranDoc.curyDiscAmt),
                         typeof(GLTranDoc.extRefNbr))]
public PXSelect<GLTranDoc,
    Where<GLTranDoc.module, Equal<Current<GLDocBatch.module>>,
        And<GLTranDoc.batchNbr, Equal<Current<GLDocBatch.batchNbr>>>>,
    OrderBy<Asc<GLTranDoc.groupTranID, Asc<GLTranDoc.lineNbr>>>>
        GLTranModuleBatNbr;
```

PXCopyPasteHiddenFields Attribute Methods

The *PXCopyPasteHiddenFields* attribute exposes the following static methods.

IsDefined(PXGraph, string, string)

Determines whether the provided graph defines a data view with the given name and this data view is marked with the `PXCopyPasteHiddenFields` attribute referencing the field.

Syntax:

```
public static bool IsDefined(PXGraph g, string viewName, string fieldName)
```

Parameters:

- `g`

  The graph object to check.
- `viewName`

  The name of the data view to check.
- `fieldName`

  The name of the field to search.

PXCopyPasteHiddenView Attribute

Indicates that the cache corresponding to the primary DAC of the data view is not copied when the copy-paste feature is utilized on the webpage.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class PXCopyPasteHiddenViewAttribute: Attribute
```

Static Methods

Method	Description
IsDefined(PXGraph, string)	Returns the value indicating whether the attribute is attached to the specified data view in the graph

Remarks

The attribute is placed on the definition of a data view in a graph to prevent the cache of the first DAC type referenced by the data view to be copied and pasted. The copy-paste feature allows a user to copy all caches related to the graph of the current webpage, add a new data record, and paste all copied caches to the new data record. The `PXCopyPasteHiddenView` attribute hides a cache from this feature.

To hide only a specific field from the copy-paste feature, use the *PXCopyPasteHiddenFields* attribute.

Examples

The code below shows the use of the attribute on the definition of a data view in a graph.

```
[PXCopyPasteHiddenView()]
public PXSelectJoin<APAdjust,
    InnerJoin<APPayment, On<APPayment.docType, Equal<APAdjust.adjgDocType>,
        And<APPayment.refNbr, Equal<APAdjust.adjgRefNbr>>>>> Adjustments;
```

As a result, the `APAdjust` cache is not copied when the user clicks **Copy** on the webpage bound to the graph where the data view is defined.

PXCopyPasteHiddenView Attribute Methods

The *PXCopyPasteHiddenView* attribute exposes the following static methods.

IsDefined(PXGraph, string)

Returns the value indicating whether the attribute is attached to the specified data view in the graph.

Syntax:

```
public static bool IsDefined(PXGraph g, string viewName)
```

Parameters:

- `g`

  The graph where the data view is defined.
- `viewName`

  The name of the data view.

PXCultureSelector Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXSelectorAttribute
    PXCustomSelectorAttribute
```

Syntax

```
public class PXCultureSelectorAttribute : PXCustomSelectorAttribute
```

Constructors

- `public PXCultureSelectorAttribute() : base(typeof(PX.SM.Locale.localeName),`

PXCustomization Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXCustomizationAttribute : Attribute
```

PXCustomStringList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXStringListAttribute
```

Syntax

```
public class PXCustomStringListAttribute : PXStringListAttribute
```

Properties

- `public string[]` **AllowedValues**

  Get.
- `public string[]` **AllowedLabels**

  Get.

Constructors

- `public PXCustomStringListAttribute(string[] AllowedValues, string[] AllowedLabels) : base(AllowedValues, AllowedLabels)`

PXDACDescription Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Assembly, AllowMultiple = true)]
public class PXDACDescriptionAttribute : Attribute
```

Properties

- `public Type` **Target**

  Get.
- `public Attribute` **Attribute**

  Get.

Constructors

- `public PXDACDescriptionAttribute(Type target, Attribute attribute)`

PXDBDataLength Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXCommandPreparingSubscriber`
- `IPXRowSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXDBDataLengthAttribute : PXEventSubscriberAttribute,
                                       IPXCommandPreparingSubscriber,
                                       IPXRowSelectingSubscriber
```

Constructors

Constructor	Description
PXDBDataLengthAttribute(Type)	
PXDBDataLengthAttribute(string)	

PXDBDataLength Attribute Constructors

The *PXDBDataLength* attribute exposes the following constructors.

PXDBDataLengthAttribute(Type)

Syntax:

```
public PXDBDataLengthAttribute(Type targetField)
```

PXDBDataLengthAttribute(string)

Syntax:

```
public PXDBDataLengthAttribute(string targetFieldName)
```

RowCondition Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBByteAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property)]
public sealed class RowConditionAttribute : PXDBByteAttribute
```

RowNbr Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldDefaultingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property)]
public sealed class RowNbrAttribute : PXEventSubscriberAttribute,
                                      IPXFieldDefaultingSubscriber
```

SSlRequest Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public class SSlRequestAttribute : PXIntListAttribute
```

Constructors

- `public SSlRequestAttribute() : base(`

TypeDelete Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public class TypeDeleteAttribute : PXIntListAttribute
```

Constructors

- `public TypeDeleteAttribute() : base(`

PXEMailAccountIDSelector Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXSelectorAttribute
    PXCustomSelectorAttribute
```

Syntax

```
public class PXEMailAccountIDSelectorAttribute : PXCustomSelectorAttribute
```

Properties

- `public override Type` **`DescriptionField`**

  Get, set.

Constructors

Constructor	Description
PXEMailAccountIDSelectorAttribute()	
PXEMailAccountIDSelectorAttribute(Boolean)	
PXEMailAccountIDSelectorAttribute(Boolean, Boolean)	

Static Methods

Method	Description
GetRecords(PXGraph)	

PXEMailAccountIDSelector Attribute Constructors

The *PXEMailAccountIDSelector* attribute exposes the following constructors.

PXEMailAccountIDSelectorAttribute()

Syntax:

```
public PXEMailAccountIDSelectorAttribute() :
 base(typeof(EMailAccount.emailAccountID))
```

PXEMailAccountIDSelectorAttribute(Boolean)

Syntax:

```
public PXEMailAccountIDSelectorAttribute(Boolean _needOwner) :
 base(typeof(EMailAccount.emailAccountID))
```

PXEMailAccountIDSelectorAttribute(Boolean, Boolean)

Syntax:

```
public PXEMailAccountIDSelectorAttribute(Boolean _needOwner, Boolean
 _onlyremoveempty) : base(typeof(EMailAccount.emailAccountID))
```

PXEMailAccountIDSelector Attribute Methods

The *PXEMailAccountIDSelector* attribute exposes the following static methods.

GetRecords(PXGraph)

Syntax:

```
public static IEnumerable GetRecords(PXGraph graph)
```

PXEMailSource Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXEMailSourceAttribute : Attribute
```

Properties

- `public Type[]` **`Types`**

  Get.

Constructors

Constructor	Description
PXEMailSourceAttribute()	
PXEMailSourceAttribute(params Type[])	

Remarks

The attribute is placed on the declaration of a DAC.

Examples

The code below shows the use of the attribute on the declaration of a DAC.

```
[System.SerializableAttribute()]
[PXPrimaryGraph(typeof(ARStatementUpdate))]
[PXEMailSource]
public partial class ARStatement : PX.Data.IBqlTable
{ ... }
```

PXEMailSource Attribute Constructors

The *PXEMailSource* attribute exposes the following constructors.

PXEMailSourceAttribute()

Syntax:

```
public PXEMailSourceAttribute() { }
```

PXEMailSourceAttribute(params Type[])

Syntax:

```
public PXEMailSourceAttribute(params Type[] types)
```

PXEntityName Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXStringListAttribute
```

Syntax

```
public class PXEntityNameAttribute : PXStringListAttribute
```

Constructors

- `public PXEntityNameAttribute(Type refNoteID)`

PXEnumDescription Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field)]
public sealed class PXEnumDescriptionAttribute : Attribute
```

Properties

- `public string` **`Category`**

  Get, set.
- `public Type` **`EnumType`**

  Get, set.
- `public string` **`Field`**

  Get, set.
- `public string` **`DisplayName`**

  Get.

Constructors

- `public PXEnumDescriptionAttribute(string displayName, Type keyType) : base()`

Static Methods

Method	Description
GetFullInfo(Type, bool)	
GetInfo(Type, object)	
GetNames(Type)	
GetValueNamePairs(Type, bool)	
GetValueNamePairs(Type, string, bool)	

PXEnumDescription Attribute Methods

The *PXEnumDescription* attribute exposes the following static methods.

GetFullInfo(Type, bool)

Syntax:

```
public static IDictionary<object, KeyValuePair<string, string>> GetFullInfo(Type
 @enum, bool localize = false)
```

GetInfo(Type, object)

Syntax:

```
public static KeyValuePair<string, string> GetInfo(Type @enum, object value)
```

GetNames(Type)

Syntax:

```
public static string[] GetNames(Type @enum)
```

GetValueNamePairs(Type, bool)

Syntax:

```
public static IDictionary<object, string> GetValueNamePairs(Type @enum, bool
 localize = true)
```

GetValueNamePairs(Type, string, bool)

Syntax:

```
public static IDictionary<object, string> GetValueNamePairs(Type @enum, string
 categoryName, bool localize = true)
```

PXExtension Attribute

Not used.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

```
  PXSelectorAttribute
```

Syntax

```
public class PXExtensionAttribute : PXSelectorAttribute
```

Constructors

- `public PXExtensionAttribute(Type type) : base(type)`

  Creates an extension.

  Parameters:

  - `type`

    Referenced table. Should be either IBqlField or IBqlSearch.

PXFeature Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
public class PXFeatureAttribute : Attribute
```

Constructors

- `public PXFeatureAttribute(Type feature)`

PXFontList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXStringListAttribute
```

Syntax

```
public sealed class PXFontListAttribute : PXStringListAttribute
```

Constructors

- `public PXFontListAttribute() : base(_values, _labels)`

PXFontSizeList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public sealed class PXFontSizeListAttribute : PXIntListAttribute
```

Constructors

- `public PXFontSizeListAttribute() : base(_values, _labels)`

PXFontSizeStrList Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXIntListAttribute
```

Syntax

```
public sealed class PXFontSizeStrListAttribute : PXIntListAttribute
```

Constructors

- `public PXFontSizeStrListAttribute() : base(PX.Common.FontFamilyEx.FontSizes, PX.Common.FontFamilyEx.FontSizesStr)`

PXLineNbrMarker Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Method, AllowMultiple = false)]
public class PXLineNbrMarkerAttribute : PXEventSubscriberAttribute
```

PXName Attribute

The base class for *PXCacheName* and *PXViewName* attributes. Do not use this attribute directly.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
public class PXNameAttribute : Attribute
```

Properties

- `public string` **`Name`**

  Gets the value specified as the name in the constructor.

Constructors

- `public PXNameAttribute(string name)`

  Initializes a new instance of the attribute that assigns the provided name to the object.

  Parameters:

  - `name`

    The value used as the name of the object.

PXNotCleanable Attribute

Inheritance Hierarchy

```
PXCacheExtensionAttribute
```

Syntax

```
public sealed class PXNotCleanableAttribute : PXCacheExtensionAttribute
```

PXNoteText Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`

Syntax

```
public class PXNoteTextAttribute : PXEventSubscriberAttribute,
                                   IPXFieldSelectingSubscriber
```

PXNotPersistable Attribute

Inheritance Hierarchy

```
PXCacheExtensionAttribute
```

Syntax

```
public sealed class PXNotPersistableAttribute : PXCacheExtensionAttribute
```

PXNoUpdate Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property)]
public class PXNoUpdateAttribute : PXEventSubscriberAttribute
```

PXNubmerSeparatorListAttribure Attribute

Inheritance Hierarchy

```
```

Syntax

```
```

PXOffline Attribute

Inheritance Hierarchy

```
PXDBInterceptorAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXOfflineAttribute : PXDBInterceptorAttribute
```

PXOverride Attribute

Indicates that the method defined in a graph extension overrides a virtual method defined in the graph. The attribute is used in the scope of the Acumatica Extensibility Framework.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
public class PXOverrideAttribute: Attribute
```

Remarks

The attribute is placed on the declaration of a method in a graph extension. As a result, the method overrides the graph method with the same signature—that is, the method is executed instead of the graph method whenever the graph method is invoked. The graph extension is a class that derives from the `PXGraphExtension` generic class, where the type parameter is set to the graph to extend.

Examples

The example below shows the declaration of a graph extension and the method that overrides the graph method.

```
// The  definition of the JournalWithSubEntry graph extension
public class JournalWithSubEntryExtension :
    PXGraphExtension<JournalWithSubEntry>
{
    [PXOverride]
    public void PrepareItems(string viewName, IEnumerable items)
```

```
    {
        ...
    }
}
```

PXPhoneValidation Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class)]
public class PXPhoneValidationAttribute : PXEventSubscriberAttribute,
                                          IPXFieldSelectingSubscriber
```

Properties

- `public virtual Type` **`PhoneValidationField`**

  Get, set.
- `public virtual string` **`PhoneMask`**

  Get, set.
- `public virtual Type` **`CountryIdField`**

  Get, set.

Constructors

- `public PXPhoneValidationAttribute(Type phoneValidationField)`

Static Methods

Method	Description
Clear<Table>()	

PXPhoneValidation Attribute Methods

The *PXPhoneValidation* attribute exposes the following static methods.

Clear<Table>()

Syntax:

```
public static void Clear<Table>() where Table : IBqlTable
```

PXRefNote Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXDBFieldAttribute
    PXDBLongAttribute
```

Syntax

```
public class PXRefNoteAttribute : PXDBLongAttribute
```

Properties

- `public bool` **`FullDescription`**

  Get, set.

Remarks

PXRefNoteSelector Attribute

Inheritance Hierarchy

```
PXViewExtensionAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class PXRefNoteSelectorAttribute : PXViewExtensionAttribute
```

Constructors

- `public PXRefNoteSelectorAttribute(Type primaryViewType, Type refNoteIDField)`

Static Methods

Method	Description
SetEnabled(PXView, bool)	

PXRefNoteSelector Attribute Methods

The *PXRefNoteSelector* attribute exposes the following static methods.

SetEnabled(PXView, bool)

Syntax:

```
public static void SetEnabled(PXView view, bool enabled)
```

PXRateSync Attribute

Synchronizes CuryRateID with the field to which this attribute is applied.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXRowInsertingSubscriber`
- `IPXRowSelectedSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Parameter |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXRateSyncAttribute : PXEventSubscriberAttribute,
                                   IPXRowInsertingSubscriber,
                                   IPXRowSelectedSubscriber
```

PXRestrictor Attribute

Adds a restriction to a BQL command that selects data for a lookup control and displays the error message when the value entered does not fit the restriction.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldVerifyingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Parameter |
                AttributeTargets.Method, AllowMultiple = true)]
public class PXRestrictorAttribute : PXEventSubscriberAttribute,
                                     IPXFieldVerifyingSubscriber
```

Properties

- `public bool` **`ReplaceInherited`**

  Gets or sets the value indicating whether the current `PXRestrictor` attribute should override the inherited `PXRestrictor` attributes.

Constructors

- `public PXRestrictorAttribute(Type where, string message, params Type[] pars)`

  Initializes a new instance of the attribute.

  The message string

  Parameters:

  - `where`

The `Where<>` BQL clause used as the additional restriction for a BQL command.

- `message`

  The error message that is displayed when a value violating the restriction is entered. The error message can reference the fields specified in the third parameter, as *{0}–{N}*. The attribute will take the values of these fields from the data record whose identifier was entered as the value of the current field.

- `pars`

  The types of fields that are referenced by the error message.

Remarks

The attribute is used on DAC fields represented by lookup controls in the user interface. For example, such fields can have the *PXSelector* attribute attached to them. The attribute adds the `Where<>` clause to the BQL command that selects data for the control. As a result, the control lists the data records that satisfy the BQL command and the new restriction. If the user enters a value that is not in the list, the error message configured by the attribute is displayed.

A typical example of attribute's usage is specifiying condition that checks whether a referenced data record is active. This condition could be specified in the `PXSelector` attribute. But in this case, if an active data record once selected through the lookup control becomes inactive, saving the data record that includes this lookup field will result in an error. Adding the condition through `PXRestrictor` attribute prevents this error. The lookup field can still hold a reference to the inactive data record. However, the new value can be selected only among the active data records.

Examples

The code below shows the use of the attribute on a lookup field.

```
[PXDBString(10, IsUnicode = true)]
[PXUIField(DisplayName = "Tax Category")]
[PXSelector(typeof(TaxCategory.taxCategoryID),
            DescriptionField = typeof(TaxCategory.descr))]
[PXRestrictor(typeof(Where<TaxCategory.active, Equal<True>>),
              "Tax Category '{0}' is inactive",
              typeof(TaxCategory.taxCategoryID))]
public virtual string TaxCategoryID { get; set; }
```

Note that the error message includes *{0}*, which will be replaced with the value of the `TaxCategoryID` field when the error message is displayed.

PXShortCut Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public sealed class PXShortCutAttribute : PXEventSubscriberAttribute,
                                          IPXFieldSelectingSubscriber
```

Properties

- `public HotKeyInfo` **`HotKey`**

  Get.

Constructors

Constructor	Description
PXShortCutAttribute(bool, bool, bool, PX.Export.KeyCodes)	
PXShortCutAttribute(bool, bool, bool, params char[])	

PXShortCut Attribute Constructors

The *PXShortCut* attribute exposes the following constructors.

PXShortCutAttribute(bool, bool, bool, PX.Export.KeyCodes)

Syntax:

```
public PXShortCutAttribute(bool ctrl, bool shift, bool alt, PX.Export.KeyCodes
 key) : this(ctrl, shift, alt, (int)key, null) { }
```

PXShortCutAttribute(bool, bool, bool, params char[])

Syntax:

```
public PXShortCutAttribute(bool ctrl, bool shift, bool alt, params char[] chars) :
 this(ctrl, shift, alt, 0, HotKeyInfo.ConvertChars(chars)) { }
```

PXSplitRow Attribute

Inheritance Hierarchy

```
PXDBInterceptorAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXSplitRowAttribute : PXDBInterceptorAttribute
```

Constructors

- `public PXSplitRowAttribute(params Type[] fields)`

PXStandartDateTimeFormatSelector Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXSelectorAttribute
    PXCustomSelectorAttribute
```

Syntax

```
public class PXStandartDateTimeFormatSelectorAttribute : PXCustomSelectorAttribute
```

Constructors

- `public PXStandartDateTimeFormatSelectorAttribute(Char code) : base(typeof(PX.SM.StandartDateTimeFormat.pattern))`

PXSubstitute Attribute

Indicates that the derived DAC should replace its base DACs in a specific graph or all graphs.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class PXSubstituteAttribute: Attribute
```

Properties

- `public Type` **`GraphType`**

  Gets or sets the specific graph in which the derived DAC replaces base DACs.
- `public Type` **`ParentType`**

  Gets or sets the base DAC type up to which all types in the inheritance hierarchy are substituted with the derived DAC. By default, the property has the null value, which means that all base DACs are substituted with the derived DAC.

Constructors

- `public PXSubstituteAttribute()`

  Initializes a new instance of the attribute. Without explicitly set properties, the attribute will cause all base DACs to be replaced with the derived DAC in all graphs.

Remarks

The attribute is placed on the definition of a DAC that is derived from another DAC. The attribute is used primarily to make the declarative references of the base DAC in definitions of calculations and links from child objects to parent objects be interpreted as the references of the derived DAC.

Examples

The code below shows the use of the `PXSubstitute` attributes on the `APInvoice` DAC.

```
[System.SerializableAttribute()]
[PXSubstitute(GraphType = typeof(APInvoiceEntry))]
[PXSubstitute(GraphType = typeof(TX.TXInvoiceEntry))]
[PXPrimaryGraph(typeof(APInvoiceEntry))]
public partial class APInvoice : APRegister, IInvoice
{ ... }
```

PXSuppressEventValidation Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Method)]
public class PXSuppressEventValidationAttribute : Attribute
```

PXSurrogate Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
public class PXSurrogateAttribute: Attribute
```

PXTableName Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class)]
public class PXTableNameAttribute : Attribute
```

PXTimeZone Attribute

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXStringListAttribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public sealed class PXTimeZoneAttribute : PXStringListAttribute
```

Properties

- `public override bool` **`IsLocalizable`**

Constructors

- `public PXTimeZoneAttribute() : base(_values, _labels) { }`

PXUnboundKey Attribute

Marks the property as a key one.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class)]
public class PXUnboundKeyAttribute : PXEventSubscriberAttribute,
                                     IPXFieldSelectingSubscriber
```

PXVirtual Attribute

Prevents the data records of a specific DAC from saving to the database. The attribute is placed on the definition of this DAC.

Inheritance Hierarchy

```
Attribute
```

Syntax

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public sealed class PXVirtualAttribute : Attribute
```

Examples

```
[PXVirtual]
[PXCacheName(Messages.TimeCardDetail)]
[Serializable]
public partial class EPTimeCardSummary : IBqlTable
{ ... }
```

PXVirtualDAC Attribute

Prevents the data view from selecting data records from the database.

Inheritance Hierarchy

```
Attribute
  PXViewExtensionAttribute
    PXCacheExtensionAttribute
```

Syntax

```
public sealed class PXVirtualDACAttribute : PXCacheExtensionAttribute
```

Remarks

The attribute can be placed on data views defined in a graph. The data view will not try to select data records from the database. You should define the optional method for this data view to form the resultset which the data view will return.

Examples

```
[PXVirtualDAC]
public PXSelect<PMProjectBalanceRecord,
    Where<PMProjectBalanceRecord.recordID, IsNotNull>,
    OrderBy<Asc<PMProjectBalanceRecord.sortOrder>>> BalanceRecords;
```

PXVirtualSelector Attribute

Suppress GUI selector, used in formula.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
  PXSelectorAttribute
```

Syntax

```
public class PXVirtualSelectorAttribute : PXSelectorAttribute
```

Constructors

- `public PXVirtualSelectorAttribute(Type type) : base(type)`

  Creates an virtual selector.

  Parameters:

  - `type`

    Referenced table. Should be either IBqlField or IBqlSearch.

PXZipValidation Attribute

Implements validation of a value for DAC fields that hold a ZIP postal code.

Inheritance Hierarchy

```
PXEventSubscriberAttribute
```

Interfaces

- `IPXFieldVerifyingSubscriber`
- `IPXFieldSelectingSubscriber`

Syntax

```
[AttributeUsage(AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Method)]
public class PXZipValidationAttribute : PXEventSubscriberAttribute,
                                        IPXFieldVerifyingSubscriber,
                                        IPXFieldSelectingSubscriber
```

Properties

- `public virtual Type` **`ZipValidationField`**

  Gets or sets the DAC field that holds the ZIP validation information in a country data record.

- `public virtual Type` **`CountryIdField`**

  Gets or sets the DAC field that holds the identifier of a country data record.

Constructors

Constructor	Description
PXZipValidationAttribute(Type)	Initializes a new instance of the attribute that does not know the field holding the ZIP mask.
PXZipValidationAttribute(Type, Type)	Initializes a new instance of the attribute that uses the specified fields to retrieve the ZIP validation information and ZIP masks per country.

Static Methods

Method	Description
Clear<Table>()	Clears the internal slots that ara used to keep ZIP definitions and ZIP mask definitions.

Examples

The code below shows a typical usage of the attribute. The constructor with two parameters, which are set to the fields from the `Country` DAC, is used. The `CountryIdField` property is set to a field from the `ARAddress` DAC where the `PostalCode` is defined.

```
[PXDBString(20)]
[PXUIField(DisplayName = "Postal Code")]
[PXZipValidation(typeof(Country.zipCodeRegexp),
                 typeof(Country.zipCodeMask),
                 CountryIdField = typeof(ARAddress.countryID))]
public virtual string PostalCode { ... }
```

PXZipValidation Attribute Constructors

The *PXZipValidation* attribute exposes the following constructors.

PXZipValidationAttribute(Type)

Initializes a new instance of the attribute that does not know the field holding the ZIP mask.

Syntax:

```
public PXZipValidationAttribute(Type zipValidationField)
    : this(zipValidationField, null)
```

PXZipValidationAttribute(Type, Type)

Initializes a new instance of the attribute that uses the specified fields to retrieve the ZIP validation information and ZIP masks per country.

Syntax:

```
public PXZipValidationAttribute(Type zipValidationField, Type zipMaskField)
```

PXZipValidation Attribute Methods

The *PXZipValidation* attribute exposes the following static methods.

Clear<Table>()

Clears the internal slots that ara used to keep ZIP definitions and ZIP mask definitions.

Syntax:

```
public static void Clear<Table>()
    where Table : IBqlTable
```

ReportView Attribute

Inheritance Hierarchy

```
Attribute
```

Syntax

```
public sealed class ReportViewAttribute : Attribute
```

Alphabetical Index

The list below contains all `PX.Data` attributes described in this reference:

- *CloseBrackets*
- *DashboardType*
- *DashboardVisible*
- *IncomingMailProtocols*
- *OpenBrackets*
- *OperationList*
- *PXAccumulator*
- *PXAggregate*
- *PXAttributeFamily*
- *PXAutoSave*
- *PXAutomationMenu*
- *PXBool*
- *PXBreakInheritance*
- *PXButton*
- *PXByte*
- *PXCacheName*
- *PXCancelButton*
- *PXCancelCloseButton*
- *PXCheckUnique*
- *PXCompositeKey*
- *PXCopyPasteHiddenFields*
- *PXCopyPasteHiddenView*

- *PXCultureSelector*
- *PXCustomSelector*
- *PXCustomStringList*
- *PXCustomization*
- *PXDACDescription*
- *PXDB3DesCryphString*
- *PXDBBinary*
- *PXDBBool*
- *PXDBByte*
- *PXDBCalced*
- *PXDBChildIdentity*
- *PXDBCreatedByID*
- *PXDBCreatedByScreenID*
- *PXDBCreatedDateTime*
- *PXDBCreatedDateTimeUtc*
- *PXDBCryptString*
- *PXDBDataLength*
- *PXDBDate*
- *PXDBDateAndTime*
- *PXDBDecimal*
- *PXDBDecimalString*
- *PXDBDefault*
- *PXDBDouble*
- *PXDBEmail*
- *PXDBField*
- *PXDBFloat*
- *PXDBGroupMask*
- *PXDBGuid*
- *PXDBIdentity*
- *PXDBInt*
- *PXDBIntList*
- *PXDBLastModifiedByID*
- *PXDBLastModifiedByScreenID*
- *PXDBLastModifiedDateTime*
- *PXDBLastModifiedDateTimeUtc*
- *PXDBLocalString*
- *PXDBLong*

- *PXDBLongIdentity*
- *PXDBScalar*
- *PXDBShort*
- *PXDBString*
- *PXDBStringList*
- *PXDBText*
- *PXDBTime*
- *PXDBTimeSpan*
- *PXDBTimeSpanLong*
- *PXDBTimestamp*
- *PXDBUserPassword*
- *PXDBVariant*
- *PXDate*
- *PXDateAndTime*
- *PXDecimal*
- *PXDecimalList*
- *PXDefault*
- *PXDefaultValidate*
- *PXDeleteButton*
- *PXDependsOnFields*
- *PXDimension*
- *PXDimensionSelector*
- *PXDimensionWildcard*
- *PXDisableCloneAttributes*
- *PXDouble*
- *PXDynamicAggregate*
- *PXDynamicMask*
- *PXEMailAccountIDSelector*
- *PXEMailSource*
- *PXEmailLoadTemplate*
- *PXEntityName*
- *PXEnumDescription*
- *PXExtension*
- *PXExtraKey*
- *PXFeature*
- *PXFilterable*
- *PXFirstButton*

- *PXFloat*
- *PXFontList*
- *PXFontSizeList*
- *PXFontSizeStrList*
- *PXFormula*
- *PXForwardMailButton*
- *PXGuid*
- *PXHidden*
- *PXImport*
- *PXInsertButton*
- *PXInt*
- *PXIntList*
- *PXLastButton*
- *PXLineNbr*
- *PXLineNbrMarker*
- *PXLong*
- *PXLookupButton*
- *PXName*
- *PXNextButton*
- *PXNoUpdate*
- *PXNotCleanable*
- *PXNotPersistable*
- *PXNote*
- *PXNoteText*
- *PXNubmerSeparatorListAttribure*
- *PXOffline*
- *PXOverride*
- *PXParent*
- *PXPhoneValidation*
- *PXPreview*
- *PXPreviousButton*
- *PXPrimaryGraph*
- *PXProcessButton*
- *PXProjection*
- *PXRSACryptString*
- *PXRateSync*
- *PXRefNote*

- *PXRefNoteSelector*
- *PXReplyMailButton*
- *PXRestrictor*
- *PXSaveButton*
- *PXSaveCloseButton*
- *PXSelector*
- *PXSendMailButton*
- *PXShort*
- *PXShortCut*
- *PXSplitRow*
- *PXStandartDateTimeFormatSelector*
- *PXString*
- *PXStringList*
- *PXSubstitute*
- *PXSuppressEventValidation*
- *PXSurrogate*
- *PXTable*
- *PXTableName*
- *PXTemplateMailButton*
- *PXTimeSpan*
- *PXTimeSpanLong*
- *PXTimeZone*
- *PXUIField*
- *PXUnboundDefault*
- *PXUnboundFormula*
- *PXVariant*
- *PXViewName*
- *PXVirtual*
- *PXVirtualDAC*
- *PXZipValidation*
- *ReportView*
- *RowCondition*
- *RowNbr*
- *SSIRequest*
- *TypeDelete*

Common Types

This chapter describes the common types that are used in more than one component of the Acumatica Framework.

The following types are described:

- *PXEntryStatus Enumeration*
- *PXErrorHandling Enumeration*
- *PXDbType Enumeration*
- *PXDBOperation Enumeration*

PXEntryStatus Enumeration

This enumeration specifies the status of a data record. The status of a data record changes as a result of manipulations with the data record: inserting, updating, or deleting.

Syntax

```
public enum PXEntryStatus
```

Members

- `Notchanged`

  The data record has not been modified since it was placed in the `PXCache` object or since the last time the `Save` action was invoked (triggering execution of BLC's `Actions.PressSave()`).

- `Updated`

  The data record has been modified, and the `Save` action has not been invoked. After the changes are saved to the database, the data record status changes to `Notchanged`.

- `Inserted`

  The data record is new and has been added to the `PXCache` object, and the `Save` action has not been invoked. After the changes are saved to the database, the data record status changes to `Notchanged`.

- `Deleted`

  The data record is not new and has been marked as `Deleted` within the `PXCache` object. After the changes are saved, the data record is deleted from the database and removed from the `PXCache` object.

- `InsertedDeleted`

  The data record is new and has been added to the `PXCache` object and then marked as `Deleted` within the `PXCache` object. After the changes are saved, the data record is removed from the `PXCache` object.

- `Held`

  An `Unchanged` data record can be marked as `Held` within the `PXCache` object to avoid being collected during memory cleanup. `Updated`, `Inserted`, `Deleted`, `InsertedDeleted`, or `Held` data records are never collected during memory cleanup. Any `Notchanged` data record can be removed from the `PXCache` object during memory cleanup.

Transitions Between Statuses

The table below shows how the status of the data record changes on invocation of different `PXCache` methods.

Original Status	Status Before	PXCache Method Invoked	Status After
-	-	Insert() / Insert(object)	Inserted
-	Inserted	Update(object)	Inserted
-	Inserted	Delete(object)	InsertedDeleted
Inserted	InsertedDeleted	Insert(object) / Update(object)	Inserted
-	Notchanged	Update(object)	Updated
-	Notchanged	Delete(object)	Deleted
Notchanged	Deleted	Insert(object) / Update(object)	Updated
-	Updated	Delete(object)	Deleted
Updated	Deleted	Insert(object) / Update(object)	Updated

PXErrorHandling Enumeration

This enumeration is used in the *PXUIField* attribute to specify when to handle the `PXSetPropertyException` exception related to the field. If the exception is handled, the user gets a message box with the error description, and the field input control is marked as causing an error.

Syntax

```
public enum PXErrorHandling
```

Members

- `WhenVisible`

  The exception is reported only when the `PXUIField` attribute with the `Visible` property set to `true` is attached to a DAC field.
- `Always`

  The exception is always reported by the `PXUIField` attribute attached to a DAC field.
- `Never`

  The exception is never reported by the `PXUIField` attribute attached to a DAC field.

PXDbType Enumeration

This enumeration specifies the SQL Server-specific data type of a field property for use in `System.Data.SqlClient.SqlParameter`.

Syntax

```
public enum PXDbType
```

Members

- `BigInt = 0`

  `System.Int64`. A 64-bit signed integer.

- `Binary = 1`

  `System.Array` of type `System.Byte`. A fixed-length stream of binary data ranging between 1 and 8000 bytes.
- `Bit = 2`

  `System.Boolean`. An unsigned numeric value that can be 0, 1, or null.
- `Char = 3`

  `System.String`. A fixed-length stream of non-Unicode characters ranging between 1 and 8000 characters.
- `DateTime = 4`

  `System.DateTime`. Date and time data ranging in value from January 1, 1753 to December 31, 9999 to an accuracy of 3.33 milliseconds.
- `Decimal = 5`

  `System.Decimal`. A fixed precision and scale numeric value between $-10^{38}-1$ and $10^{38}-1$.
- `Float = 6`

  `System.Double`. A floating point number within the range of -1.79E+308 through 1.79E+308.
- `Image = 7`

  `System.Array` of type `System.Byte`. A variable-length stream of binary data ranging from 0 to $2^{31}-1$ (or 2,147,483,647) bytes.
- `Int = 8`

  `System.Int32`. A 32-bit signed integer.
- `Money = 9`

  `System.Decimal`. A currency value ranging from -2^{63} (or -922,337,203,685,477.5808) to $2^{63}-1$ (or +922,337,203,685,477.5807) with an accuracy to a ten-thousandth of a currency unit.
- `NChar = 10`

  `System.String`. A fixed-length stream of Unicode characters ranging between 1 and 4000 characters.
- `NText = 11`

  `System.String`. A variable-length stream of Unicode data with a maximum length of $2^{30}-1$ (or 1,073,741,823) characters.
- `NVarChar = 12`

  `System.String`. A variable-length stream of Unicode characters ranging between 1 and 4000 characters. Implicit conversion fails if the string is greater than 4000 characters. Explicitly set the object when you're working with strings longer than 4000 characters.
- `Real = 13`

  `System.Single`. A floating point number within the range of -3.40E+38 through 3.40E+38.
- `UniqueIdentifier = 14`

  `System.Guid`. A globally unique identifier (GUID).
- `SmallDateTime = 15`

  `System.DateTime`. Date and time data ranging in value from January 1, 1900 to June 6, 2079 to an accuracy of one minute.
- `SmallInt = 16`

System.Int16. A 16-bit signed integer.

- SmallMoney = 17

  System.Decimal. A currency value ranging from -214,748.3648 to +214,748.3647 with an accuracy to a ten-thousandth of a currency unit.

- Text = 18

  System.String. A variable-length stream of non-Unicode data with a maximum length of 2^{31}-1 (or 2,147,483,647) characters.

- Timestamp = 19

  System.Array of type System.Byte. Automatically generated binary numbers, which are guaranteed to be unique within a database. The timestamp is typically used as a mechanism for version-stamping table rows. The storage size is 8 bytes.

- TinyInt = 20

  System.Byte. An 8-bit unsigned integer.

- VarBinary = 21

  System.Array of type System.Byte. A variable-length stream of binary data ranging between 1 and 8000 bytes. Implicit conversion fails if the byte array is greater than 8000 bytes. Explicitly set the object when you are working with byte arrays larger than 8000 bytes.

- VarChar = 22

  System.String. A variable-length stream of non-Unicode characters ranging between 1 and 8000 characters.

- Variant = 23

  System.Object. A special data type that can contain numeric, string, binary, or date data, as well as the SQL Server values EMPTY and NULL, which is assumed if no other type is declared.

- Xml = 25

  An XML value. Obtain the XML as a string by using the System.Data.SqlClient.SqlDataReader.GetValue(System.Int32) method or the System.Data.SqlTypes.SqlXml.Value property, or as System.Xml.XmlReader—by calling the System.Data.SqlTypes.SqlXml.CreateReader() method.

- Udt = 29

  An SQL Server user-defined type (UDT).

- Unspecified = 100

  Unspecified value type that is implicitly converted by SQL Server into an appropriate database column type.

- DirectExpression = 200

  A string constant containing a T-SQL statement being embedded into the final statement.

PXDBOperation Enumeration

This enumeration specifies the type of a T-SQL statement generated by the Acumatica Data Access Layer.

The enumeration is used to indicate the type of the operation and the option set for the operation. PXDBOperation supports the FlagsAttribute attribute, which allows PXDBOperation members to be represented as bit fields in the enumeration value.

Syntax

```
public enum PXDBOperation
```

Members

`PXDBOperation` members can be divided into two groups:

Command

Member	Value	Description
Select	0	SELECT operation
Update	1	UPDATE operation
Insert	2	INSERT operation
Delete	3	DELETE operation

Option

Member	Value	Description
Normal	0	The operation has no options set.
GroupBy	4	This specifies an aggregate operation.
Internal	8	The result of the operation cannot be used to prepare the external representation.
External	12	The operation contains a sorting, filter, or search query across any DAC field visible in the UI.
Second	16	The operation is changing system data visibility and transferring it from the system data segment to the customer data segment.

Examples

Getting the type of an operation:

```
protected virtual void DACName_FieldName_CommandPreparing(
    PXCache sender,
    PXCommandPreparingEventArgs e)
{
    PXDBOperation operationKind = e.Operation & PXDBOperation.Command;
}
```

Getting the option set for an operation:

```
protected virtual void DACName_FieldName_CommandPreparing(
    PXCache sender,
    PXCommandPreparingEventArgs e)
{
    PXDBOperation operationOptions = e.Operation & PXDBOperation.Option;
}
```

Report Designer

This section provides the information on how to create report forms and printed pages by using the Report Designer tool.

- *Acumatica Report Designer Report Designer User Interface*
- *Creating and Modifying the Reports*
- *Selecting Data for the Report*
- *Data Grouping and Sorting*
- *Using Expressions*
- *Creating the Report Content*
- *Using Variables*
- *Using the External Parameter Collection Editor*
- *Saving and Publishing the Reports*

Acumatica Report Designer Report Designer User Interface

The Acumatica Report Designer provides visual tools that you can use to design custom reports. From the Acumatica Report Designer screen, you can select the report data from the Acumatica ERP system database, create the report content, and save the report in a detached file with the .rpx format. This file stores the report description as XML data.

Accessing the Report Designer

To view the Acumatica Report Designer main window, navigate as follows: **Start** > **Programs** > **Acumatica** > **Report Designer**.

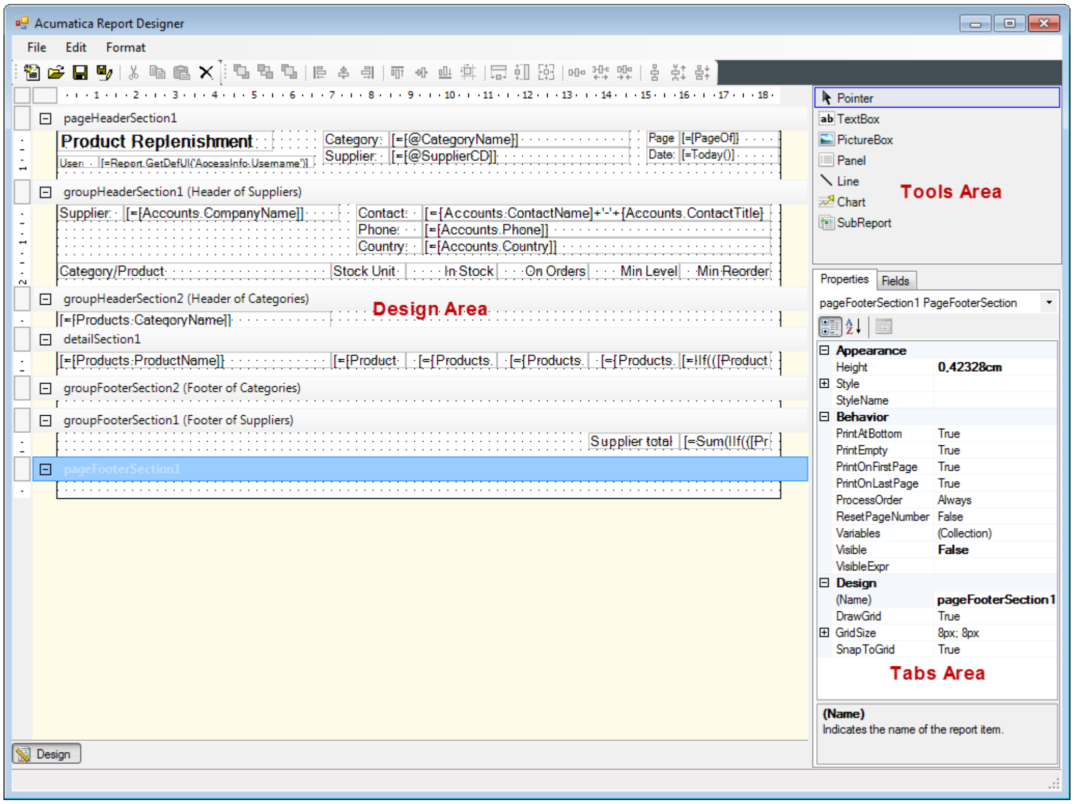


Figure: Report Designer Main Window

The main window of the Acumatica Report Designer includes three areas:

- The **Design** area displays the report layout, which users can change.
- The **Tools** area provides access to the tools that can be used to design the report layout and add the report content.
- The **Tabs** area includes the following tabs:
    - **Properties**: Displays the properties of the report element selected in the Design area.
    - **Fields**: Lists the names of all data access class (DAC) fields selected as the source of data for the report.

Main Window Menu

The Main Window menu of the Acumatica Report Designer includes the commands described below.

Main Window Menu Commands

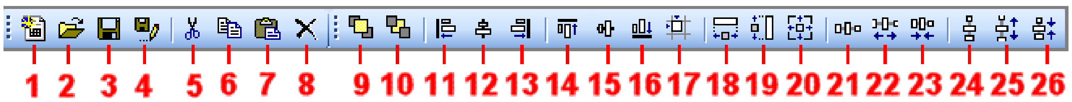
Command	Description
File	The commands under the **File** menu, listed below, provide access to the main operations with the report file and allow you to access the database schema:
	New: Creates a new report file.
	Open: Opens an existing report file.
	Open From Server: Opens an existing report file located on the Acumatica ERP application server.
	Save: Saves the current report file.

Command	Description
	Save As: Saves the current report in a new file. This command can be used to rename a report file or to save it to a new location.
	Save On Server: Saves the report on the Acumatica ERP application server.
	Build Schema: Runs the **Schema Builder** wizard.
	Exit: Closes the Report Designer main window.
Edit	The commands under the **Edit** menu, listed below, allow you to perform basic editing operations with the objects placed in the Design area.
	Cut: Removes the selected items from the Design area and places a copy of them on the clipboard.
	Copy: Places a copy of the selected items on the clipboard.
	Paste: Places the items from the clipboard in the Design area.
	Delete: Completely removes the selected items from the Design area.
Format	The commands under the **Format** menu, listed below, let you perform basic formatting operations on the objects placed in the Design area.
	Bring To Front: Changes the layering of the objects placed in the Design area, placing the selected items in front of all the other items in the area.
	Send To Back: Changes the layering of the objects placed in the Design area, placing the selected items behind all the other items in the area.
	Align: Aligns the selected objects as follows: • *Left*, *Center*, and *Right* dictate how the selected items in the Design area will be horizontally aligned. • *Top*, *Middle*, and *Bottom* determine how the selected items in the Design area will be vertically aligned. • *To Grid* snaps the selected items in the Design area to the grid.
	Make Same Size: Adjusts the size of the selected items in the Design area as follows: • *Width*: Makes the selected objects the same width. • *Height*: Makes the selected objects the same height. • *Both*: Makes the selected objects the same width and height.
	Horizontal Spacing: Changes the horizontal spacing between the selected items in the Design area as follows: • *Make Equal*: Sets equal horizontal spacing between the selected objects. • *Increase*: Increases the horizontal spacing between the selected objects. • *Decrease*: Decreases the horizontal spacing between the selected objects. • *Remove*: Sets a zero horizontal spacing between the selected objects.
	Vertical Spacing: Changes the vertical spacing between the selected items in the Design area as follows: • *Make Equal*: Sets equal vertical spacing between the selected objects.

Command	Description
	• *Increase*: Increases the vertical spacing between the selected objects. • *Decrease*: Decreases the vertical spacing between the selected objects. • *Remove*: Sets a zero vertical spacing between the selected objects.

Main Window Toolbar

The Main Window toolbar of the Acumatica Report Designer provides single-click access to the menu buttons, as shown and described described below.



Main Window Toolbar Buttons

Number	Button Name	Description
1	**New Report**	Invokes the *New* command from the **File** menu
2	**Open Report**	Invokes the *Open* command from the **File** menu
3	**Save Report**	Invokes the *Save* command from the **File** menu
4	**Save Report As**	Invokes the *Save As* command from the **File** menu
5	**Cut**	Invokes the *Cut* command from the **Edit** menu
6	**Copy**	Invokes the *Copy* command from the **Edit** menu
7	**Paste**	Invokes the *Paste* command from the **Edit** menu
8	**Delete**	Invokes the *Delete* command from the **Edit** menu
9	**Bring To Front**	Invokes the *Bring To Front* command from the **Format** menu
10	**Send To Back**	Invokes the *Send To Back* command from the **Format** menu
11	**Align Left**	Invokes the *Align > Left* command from the **Format** menu
12	**Align Center**	Invokes the *Align > Center* command from the **Format** menu
13	**Align Right**	Invokes the *Align > Rights* menu command from the **Format** menu
14	**Align Top**	Invokes the *Align > Top* command from the **Format** menu
15	**Align Middle**	Invokes the *Align > Middle* command from the **Format** menu
16	**Align Bottom**	Invokes the *Align > Bottom* command from the **Format** menu
17	**Align To Grid**	Invokes the *Align > To Grid* command from the **Format** menu
18	**Make Same Width**	Invokes the *Make Same Size > Width* command from the **Format** menu

Number	Button Name	Description
19	**Make Same Height**	Invokes the *Make Same Size > Height* command from the **Format** menu
20	**Make Same Size**	Invokes the *Make Same Size > Both* command from the **Format** menu
21	**Make Horizontal Spacing Equal**	Invokes the *Horizontal Spacing > Make Equal* command from the **Format** menu
22	**Increase Horizontal Spacing**	Invokes the *Horizontal Spacing > Increase* command from the **Format** menu
23	**Decrease Horizontal Spacing**	Invokes the *Horizontal Spacing > Decrease* command from the **Format** menu
24	**Make Vertical Spacing Equal**	Invokes the *Vertical Spacing > Make Equal* command from the **Format** menu
25	**Increase Vertical Spacing**	Invokes the *Vertical Spacing > Increase* command from the **Format** menu
26	**Decrease Vertical Spacing**	Invokes the *Vertical Spacing > Decrease* command from the **Format** menu

Creating and Modifying the Reports

By using the Acumatica Report Designer, you can create new reports or modify existing ones. Both options are briefly described below.

Creating a New Report

To create a new report, on the **File** menu, select *New*. The Report Designer creates a new report file that includes the page header section, the page footer section, and the detail section. You can then create the layout or add content, as described in the *Composing the Report Layout* and *Creating the Report Content* sections of this guide. After modification, the report file can be saved in a local folder or saved to the Web server.

Modifying an Existing Report

To modify an existing report, you can open a locally stored file or load the file from the Acumatica ERP website.

To open a locally stored file, on the **File** menu, select *Open*, and then select the report file to be modified. The selected file is displayed in the Design area of the Report Designer. Once you modify it, you can save it locally or on the server.

To open a report file located on the Acumatica ERP website, on the **File** menu of the Report Designer, select *Open*.

Selecting Data for the Report

When you create a report, you define the rules used to select the necessary data to be displayed in the report. This data is retrieved from the system database via an appropriate data access class (DAC).

To define what data is selected from the database, the Report Designer provides the Schema Builder wizard. Using this wizard, users can load the database schema, set the report parameters, and define the data selection criteria, data filtering, and sorting and grouping rules.

In This Section

This section contains the following articles:

- *Loading the Database Schema*
- *Building the Database Request*

Loading the Database Schema

The Acumatica Report Designer accesses the database through the data access classes (DACs) defined in Acumatica ERP. To select the necessary data for the report, you need to load the WSDL file generated by the Acumatica ERP application server. The WSDL file contains the definition of all available DACs.

Connecting to the Application Server

To connect to the Application Server, perform the following steps:

1. Start the Report Designer: **Start > All Programs > Acumatica > Report Designer**.
2. Create a new blank report form by clicking **New Report** on the toolbar (or by accessing this option on the **File** menu).

   When you open the Report Designer for the first time, the blank report form is displayed by default.

3. On the **File** menu, select *Build schema*, as shown on the screenshot below. The Schema Builder wizard appears.

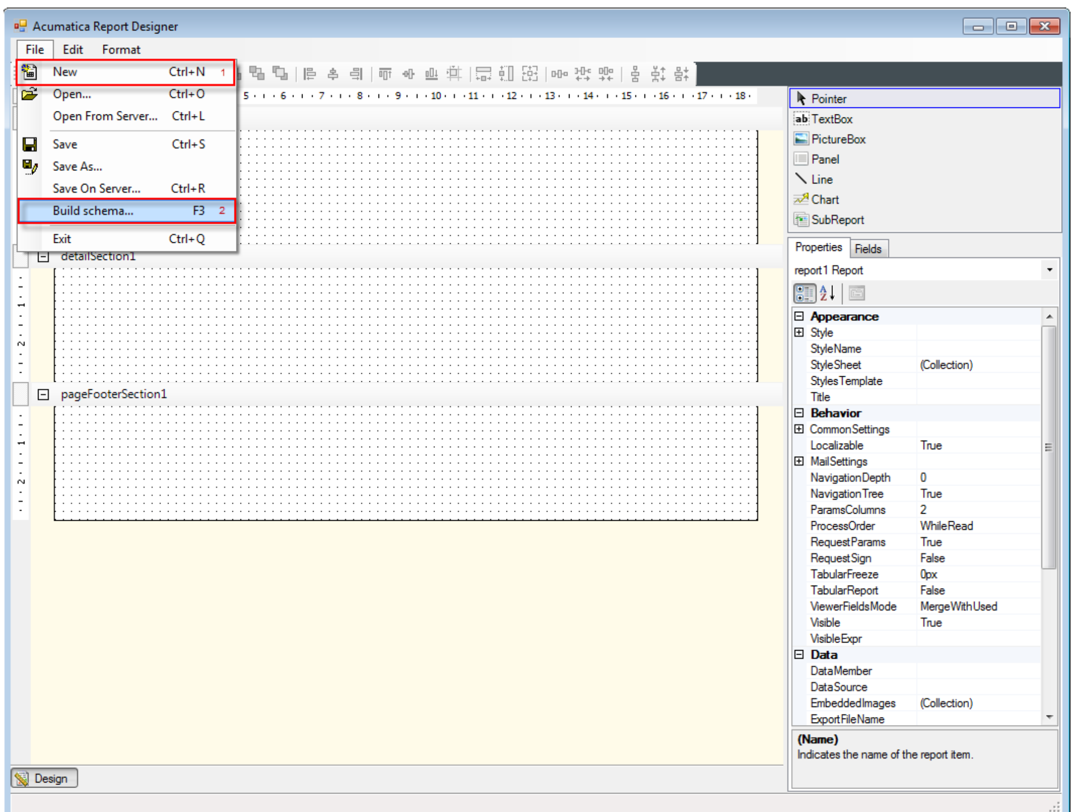


Figure: Accessing the Schema Builder

4. To load the Acumatica ERP WSDL definition file, enter the connection string (as shown in the second screenshot below, in the area left of the red 1):

```
http://{domain}
```

Here, you must replace with the actual URL to your application; you may also need to replace http with https. If access to the WSDL definition is restricted, provide the user ID and password (see item 2 in the screenshot below). A typical connection string for an application launched from Microsoft Visual Studio on a local computer looks like the following:

```
http://localhost:64971/Site
```

5. Click the **Load schema** button (item 3). The Report Designer connects to the application server and loads the Acumatica ERP schema definition. When the WSDL file is retrieved, notice the list of all data access classes (DACs) defined in your application, as shown in the screenshot below.

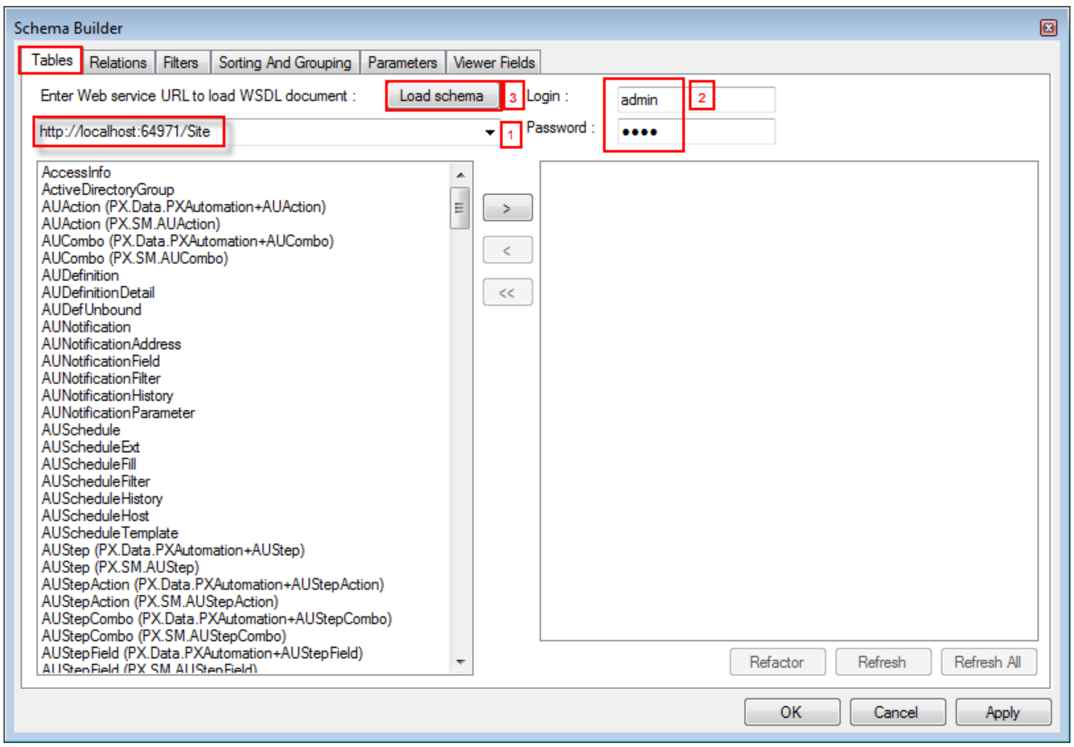


Figure: Loading the DAC schema

When you load the schema definition from the application in Visual Studio, make sure that the application has been started and is accessible through the Web browser.

The Acumatica Report Designer receives all the meta information required for report creation from the Acumatica ERP WSDL file. You don't need to install Acumatica ERP locally to develop the report, instead you can just connect to the remote server by using the appropriate URL.

Building the Database Request

Data access classes (DACs), which are used to access the data in the system database from the report engine, must be defined for each report. To specify what data will be displayed in the designed report, you should perform the following steps, each of which is described in detail below:

- Select the DACs from the list of available ones displayed on the **Tables** tab of the Schema Builder wizard. The selected DACs specify the tables in the system database from which the data will be selected.

- Specify the relations between the selected DACs on the **Relations** tab of the Schema Builder wizard. The DAC relations provide the necessary information to build the SQL request to the database.

Selecting the DACs for the Report

To select the DACs to be included in the report, perform the following steps:

1. In the list of available DACs on the **Tables** tab of the Schema Builder wizard, select the DAC name to select the data from the database table related to this DAC.
2. Click the button to move the DAC to the list of the selected DACs.
3. Repeat Steps 1 and 2 for each DAC to be selected. The selected DACs will appear in the list of the selected DACs in the right side of the **Tables** tab.

   To remove a DAC from the list of selected DACs, select the DAC by name and click the button (see screenshot). To remove all the DACs from the list of selected DACs, click the button.

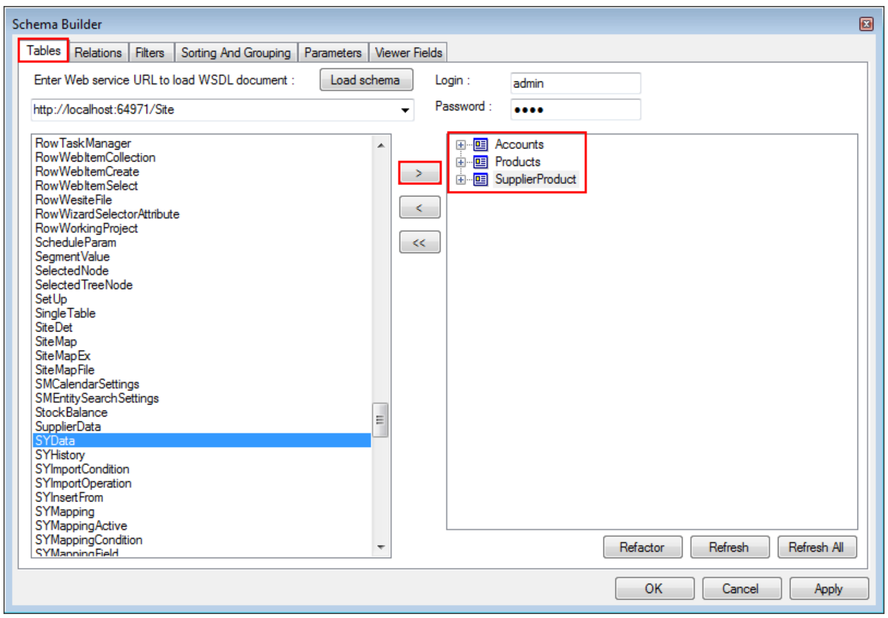


Figure: Selecting DACs

The list of the selected DACs displays the DACs and their attributes, which match the fields in the database table related to the DAC.

Specifying the Relations Between DACs

The **Relations** tab of the **Schema Builder** wizard allows you to specify the relations between DACs. The relations between the DACs specify how the relevant tables will be joined in the generated SQL request.

To define a relation between two DACs, you must specify the DAC related to the parent table and the DAC related to the child table in the relation, and specify the DAC attributes related to the data fields to be used as the relevant table joining criteria. Any report can include one or multiple relations between the two DACs.

To set the relations between DACs, repeat the following steps for each relation to be used in the report:

1. Click the empty line in the grid **Enter the report table relations here**.
2. In the **Parent Table** box, select the name of the parent table in the relation.
3. In the **Join Type** box, select the type of table join: *Left*, *Right*, *Inner*, *Full*, or *Cross*.

4. In the **Child Table** box, select the name of the child table in the relation.
5. Enter the aliases for the parent and child tables (**Parent Alias** and **Child Alias**), if required.

For each relation between the DACs, you should also specify the data field links. Repeat the following steps for each data link to be used in the relation between the tables:

1. Click the empty line in the grid **Enter the data field links for active relation**.
2. In the **Parent Field** box, select the name of the parent field for the data link.
3. In the **Link Condition** box, select the condition for linking the fields: *Equal*, *NotEqual*, *Greater*, *GreaterOrEqual*, *Less*, or *LessOrEqual*. You can also select the *IsNull* or *IsNotNull* items; in such a case, you should not add a child field.
4. In the **Child Field** box, select the name of the child field for the data link.
5. If more than one relation expression will be used for joining the data tables, select the operator: *And* or *Or*.
6. Select the **Braces** if they are required in the data link expressions.

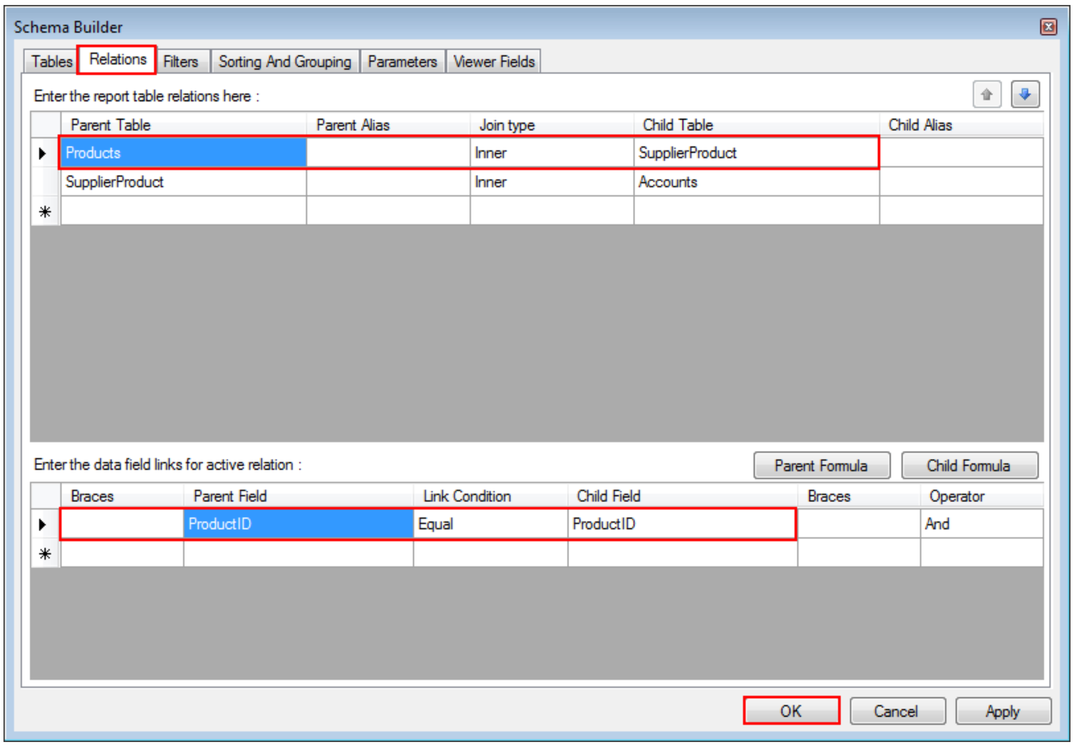


Figure: Configuring Relation 1

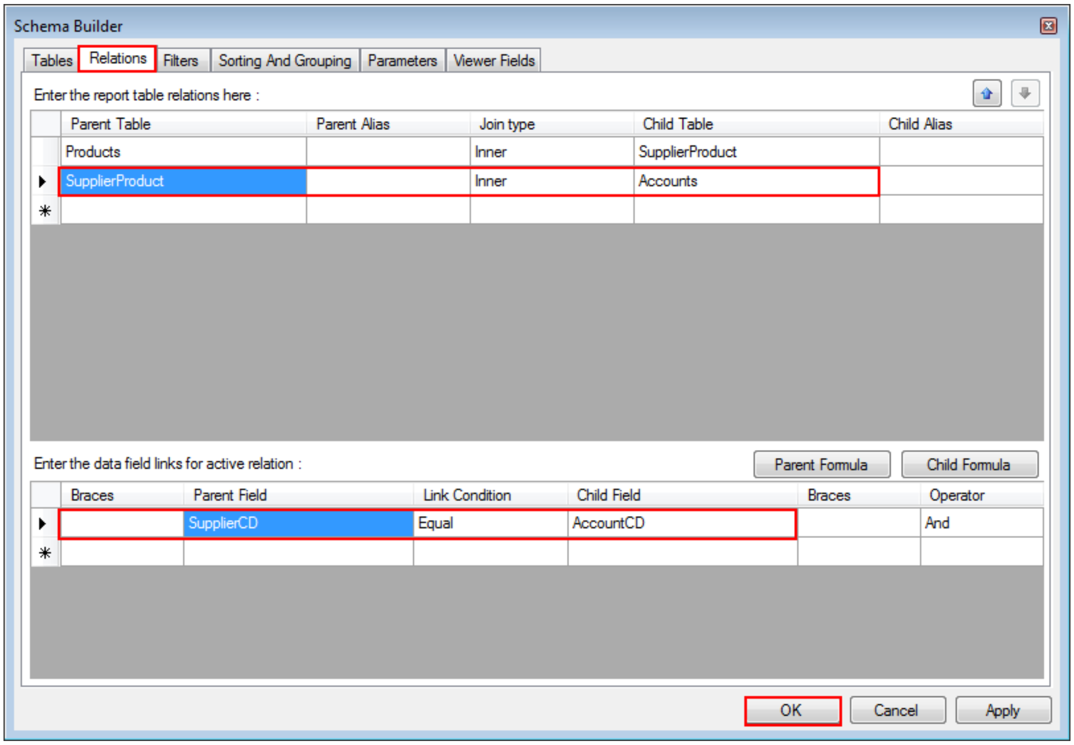


Figure: Configuring Relation 2

The DACs relations and data field links you defined can be deleted: Simply click the relevant line in the grid, and press the DELETE key.

Composing the Report Layout

The report layout determines the visual presentation of the data. To design the report layout, you should perform the following tasks:

- Define what sections will be included in the report
- Set up the headers and footers for the report and each report section
- Set the appearance parameters for each report section
- Define the behavior parameters for each report section
- Add visual elements to the report

In This Section

This section includes the following articles:

- *Adding and Removing Report Sections*
- *Defining the Appearance of a Report Section*
- *Defining the Behavior Settings of a Report Section*
- *Adding and Removing Visual Elements in the Report*

Adding and Removing Report Sections

By default, when you create a new report, it includes three sections: the page header section, one page detail section (others can be added), and the page footer section. The sections can display various content, and the values of variables used to calculate and display report values can be reset in each new section.

You can add a new report section or delete any section. You can also copy the style of one section and apply it to another.

Adding a Report Section

To add a report section by duplicating an existing one, proceed as follows:

1. Select the report section you wish to duplicate, and right-click it.
2. Choose **Duplicate Section** in the pop-up menu, and the new section will be added to the report. The new section will have the same type as the parent section (header of the relevant group, footer of the relevant group, detail section, page footer).

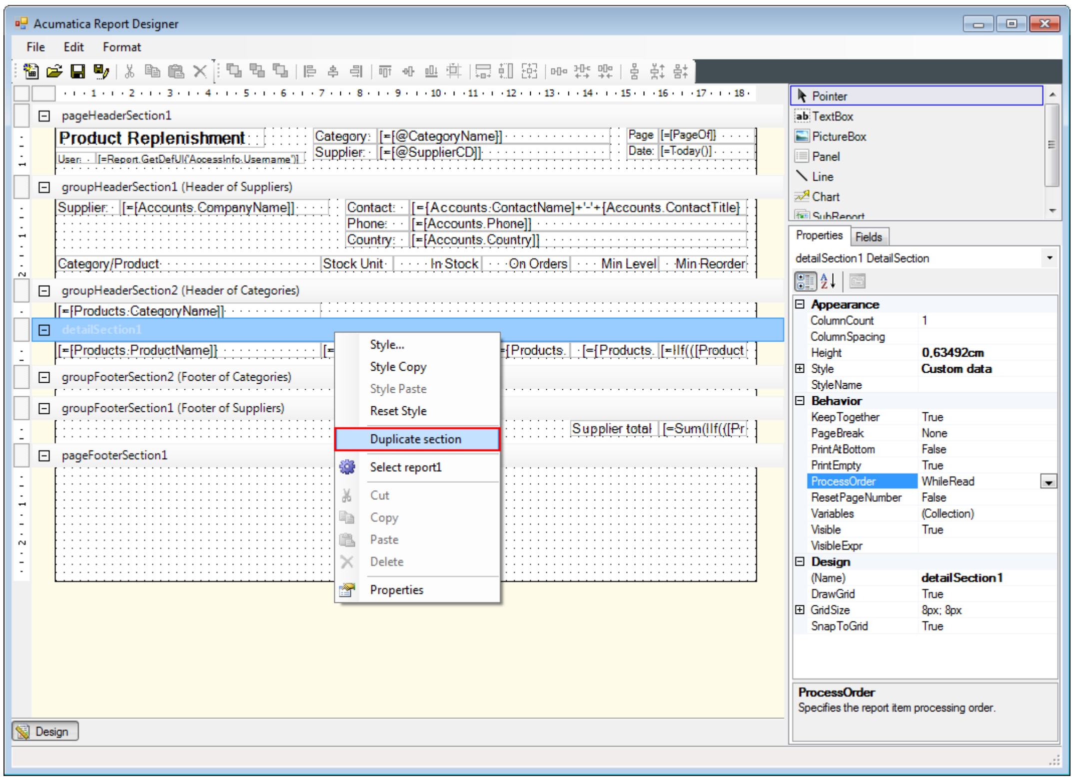


Figure: Duplicating a report section

Removing a Report Section

To remove an existing section from a report, do the following steps:

1. Right-click the section.
2. Choose **Delete** in the pop-up menu. The selected section will be removed from the report.

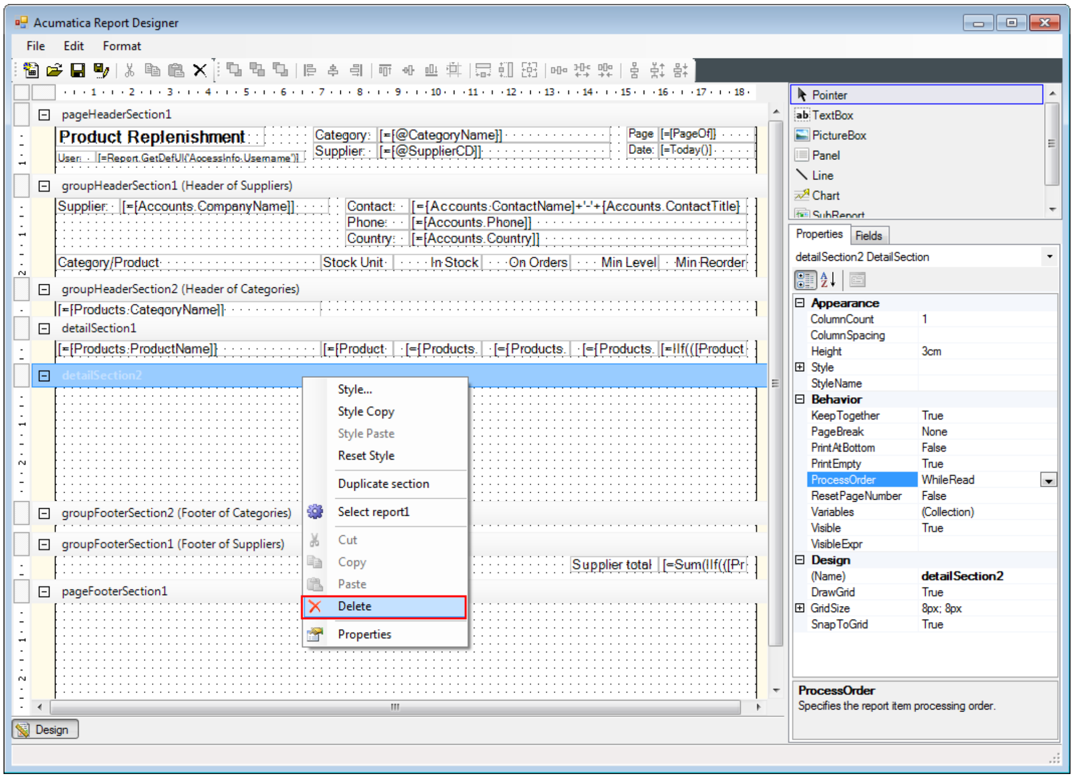


Figure: Deleting a report section

Copying the Style Between the Report Sections

The style defined for one report section can be applied to another section. To copy the style between the sections, perform the following steps:

1. Right-click the report section from which the style should be copied, and choose **Style Copy** from the pop-up menu.

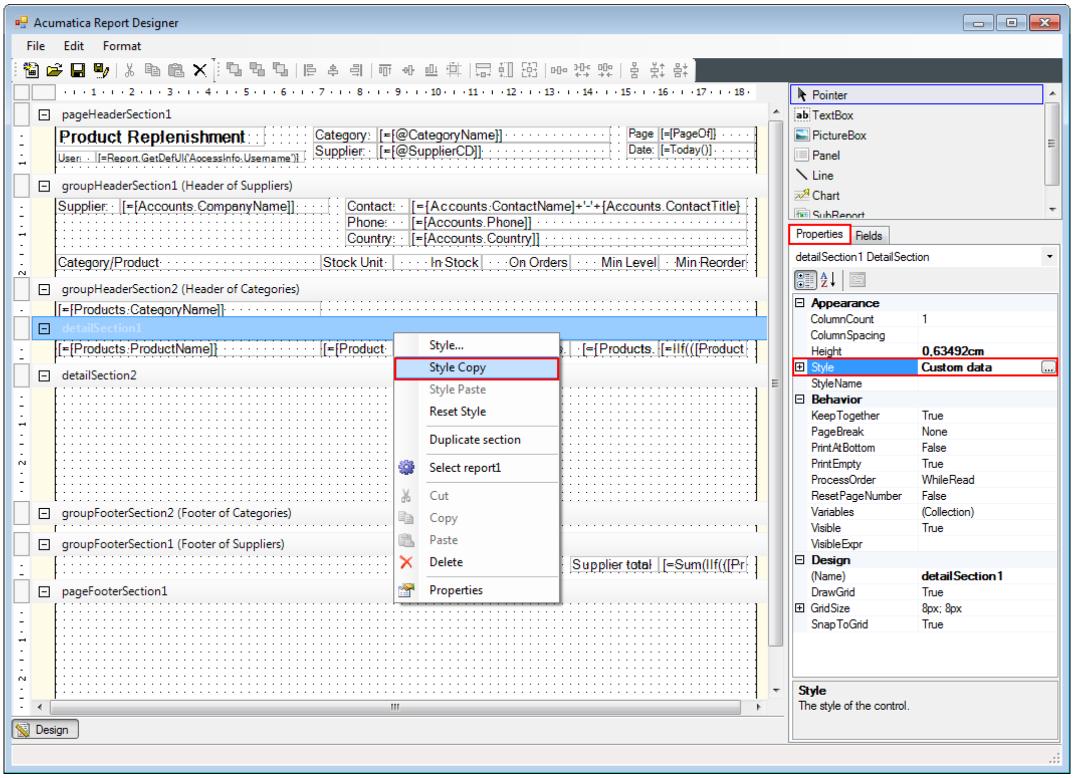


Figure: Copying the style from a section

2. Right-click the report section to which the style should be applied, and choose **Style Paste** from the pop-up menu. The selected style will be applied to this section.

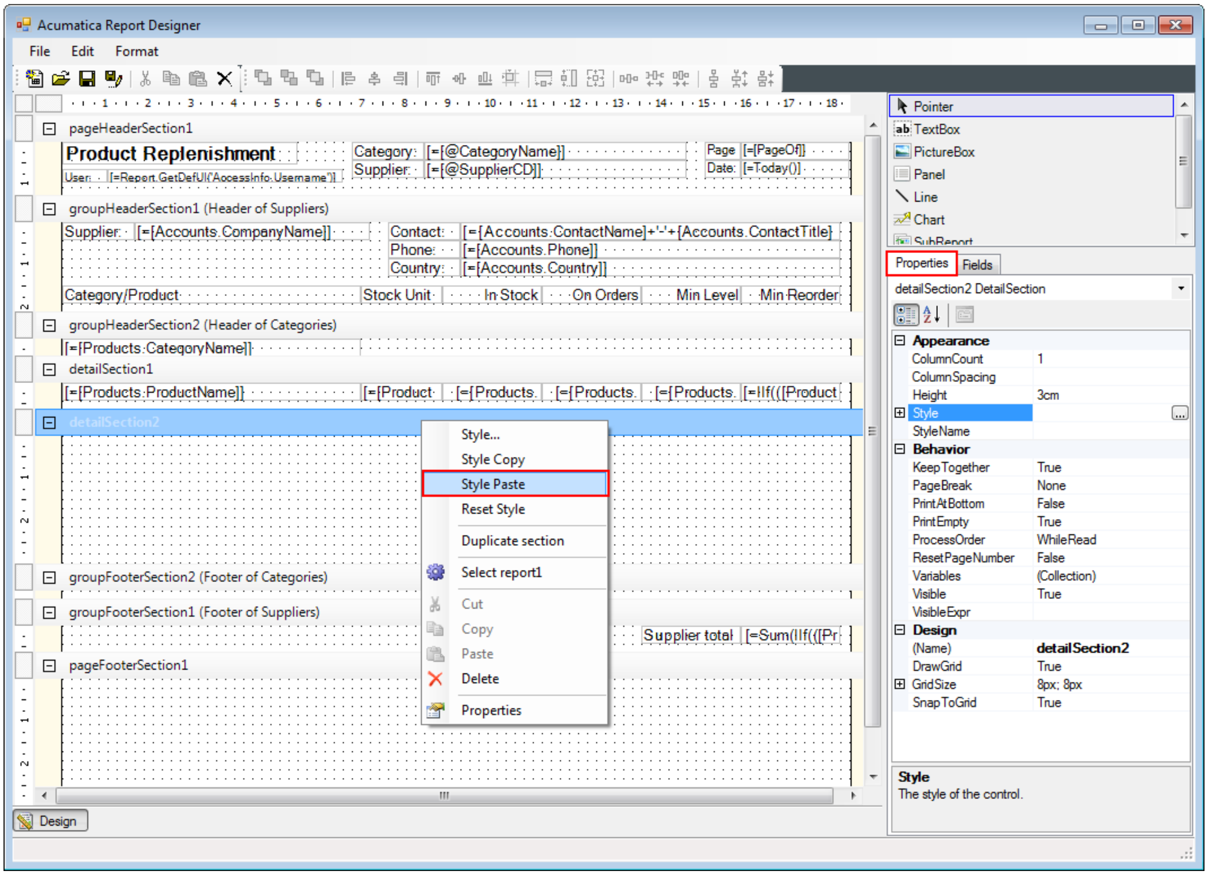


Figure: Applying the copied style to another section

We recommend that you use a special report template instead of defining styles—see the *Defining the Behavior Settings of a Report Section* article.

Defining the Appearance of a Report Section

Acumatica Framework supports report styling with two files: *TemplateReport.rpx* (for preparation of common reports) and *TemplateForm.rpx* (for preparation of printing Web pages). Using report templates enables users to print reports and documents that share a uniform style. You can create report and document templates yourself or edit existing Acumatica Framework templates through Microsoft Visual Studio. (Template files are XML files that define a set of styles.) Using style templates is the most sensible way to prepare well-styled reports and documents.

If you decide not to use templates, programmers can manually adjust for a group of users font types, font colors, font sizes, and other settings for each field and label. (In the second screenshot below, you can see the **Style** group of parameters, which can be adjusted for fields and labels.) This method is labor-consuming, however; that's why using report and document templates is recommended.

Using Template Files

To use a template file, proceed as follows:

1. In Acumatica Report Designer, select the top level of the report. On the **Properties** tab, locate the **Styles Template**. Open the list of report files, choose the *TemplateReport.rpx* file, and click the **Open** button, as shown in the first screenshot below.

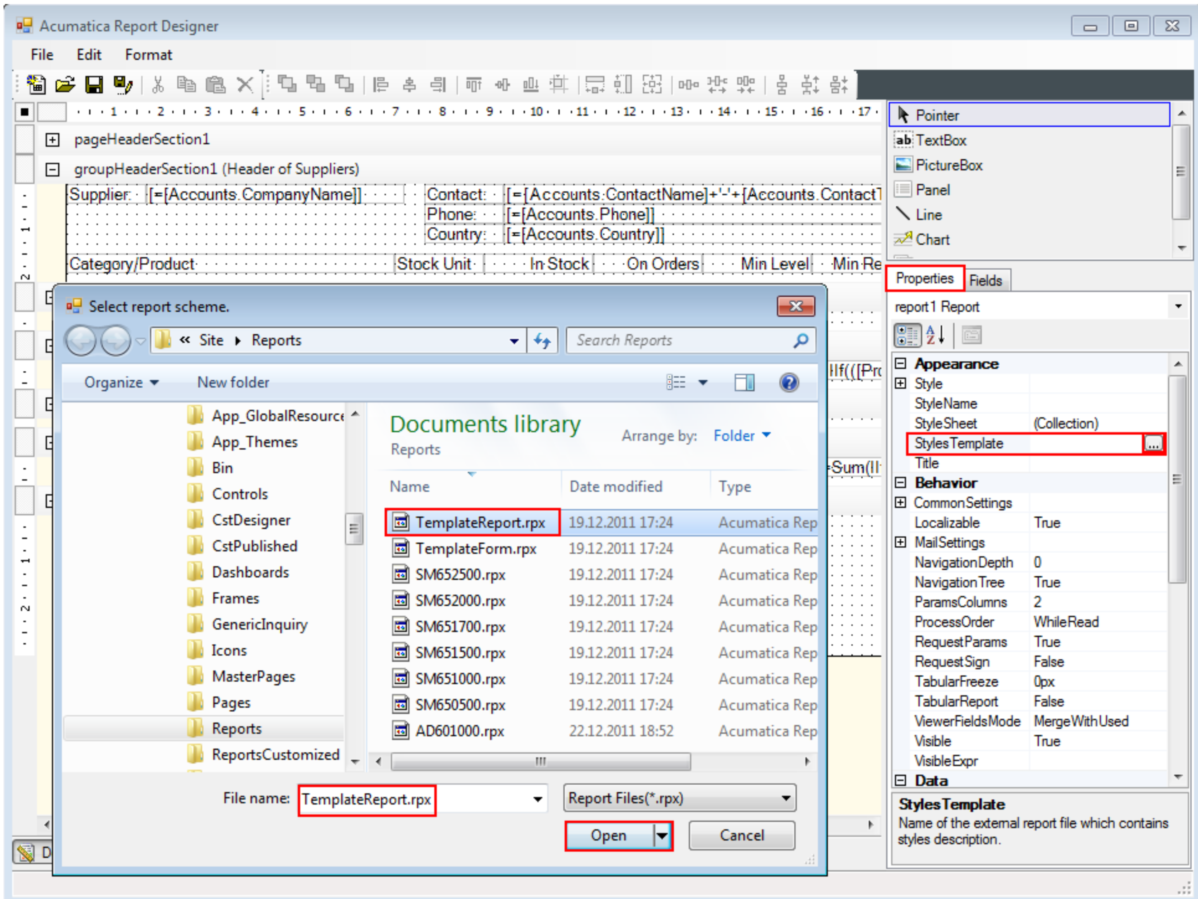


Figure: Selecting the template report

2. Select any report field and set the required **StyleName** property. (The second screenshot below illustrates this with the **Contact** data field.)

3. Try to set appropriate **StyleName** properties for the most fields and labels, save the report, and then open and execute the *Product Replenishment* report. The report will change its appearance according to the styles predefined for the fields and labels.

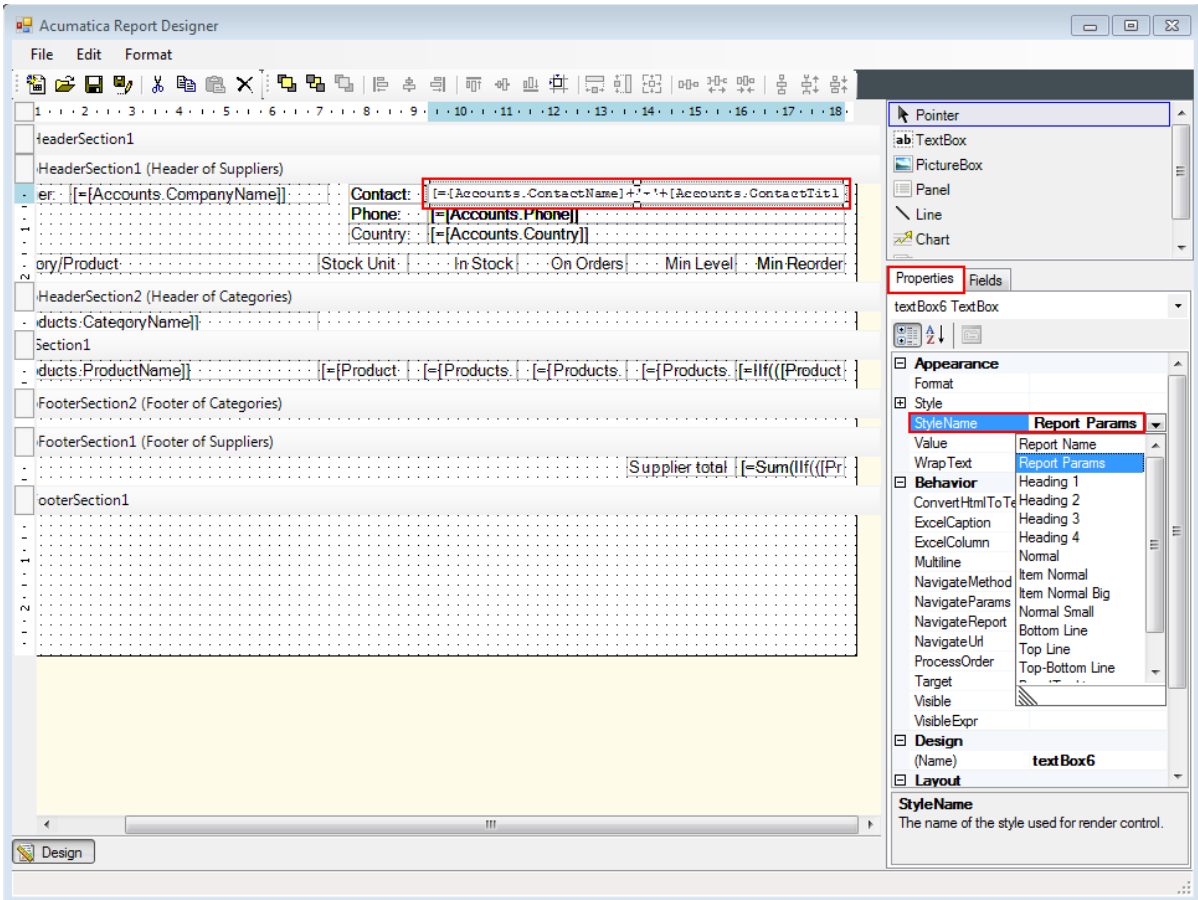


Figure: Setting a style parameter for a field

Defining a Report Section's Appearance Settings

You can define the appearance settings of each report section, which determine how the report section will be printed. Appearance settings include the following:

- The number of columns
- The space between the columns
- The style of the section, which includes its text properties, border settings, and background color and image

To define the appearance settings for a report section, perform the following steps:

1. Click the section within the report to select it, as shown in the screenshot below.

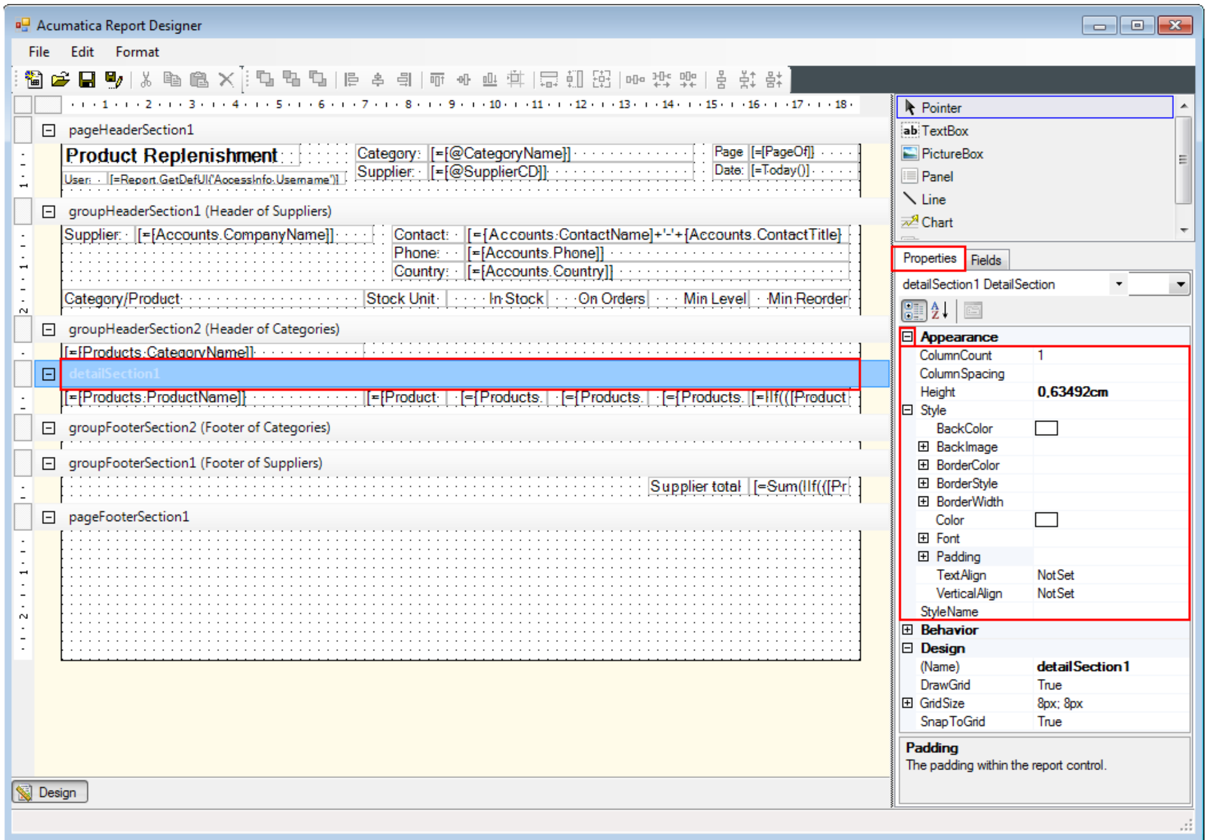


Figure: Styling adjustment

2. On the **Properties** tab, in the **Appearance** group, choose settings for the fields described below.

Appearance Settings

Field	Description
ColumnCount	The number of columns within the report section.
ColumnSpacing	The spacing between the columns (in pixels).
Height	The height of the section (in centimeters).
Style	The printing style for the report section, set by the values in the following fields.
	BackColor The background color for the report section.

	BackImage The background image parameters for the report section. Enter desired values in the following fields:
	• **Source** - Specify the source of the image.
	• **Image** - Define the image to be used as the background:
	    • For an embedded image, select the image name.
	    • For an external image, enter the path to the image file.
	    • For an image retrieved from the database, enter the name of the data field where the image is stored.

Field	Description
	• **Repeat** - Select the appropriate value specifying the repeating pattern for the chosen image: ◦ *NoRepeat* - Adds the specified image with no repeating ◦ *RepeatX* - Repeats the image horizontally to fill the width of the report section ◦ *RepeatY* - Repeats the image vertically to fill the height of the report section ◦ *Repeat* - Repeats the image horizontally and vertically to fill both the width and height of the report section
	BorderColor The border color for the report section. You can define the color for the bottom, left, right, and top border of the section, and set the default border color, which will be applied if no special settings are defined for the specific borders.
	BorderStyle The border line style. You can define the style for the bottom, left, right, and top border of the section, and set the default border style, which will be applied if no special settings are defined for the specific borders.
	BorderWidth The border line width for the report section (in pixels). You can define the width of the bottom, left, right, and top border of the section, and set the default border width, which will be applied if no special settings are defined for the specific borders.
	Font The font settings for the report section. You can select the font name and size and specify whether the following font attributes are applied: bold, italic, strikeout, and underline.
	Padding The padding setting for the report section, which you can specify in pixels for the left side, right side, top, and bottom of the report section.
	TextAlign The text alignment for the report section: *Left*, *Center*, *Right*, or *Not Set*.
	VerticalAlign The content vertical alignment for the report section: *Not Set*, *Top*, *Middle*, or *Bottom*.
StyleName	The name of the style defined for the report section. To assign a descriptive name to a style you have defined for a report section, enter the name. To apply an existing style to the report section, select its style name.

Defining the Behavior Settings of a Report Section

Each section has its own behavior settings that define the following:

- How the section data is processed
- How the section position on the page is controlled
- How the section's data is displayed in the report
- What variables are defined within the report section

Defining Behavior Settings for Section

To define the behavior settings for a report section, perform the following steps:

1. Click the section within the report to select it, as shown in the screenshot below (one of the header groups had been selected).

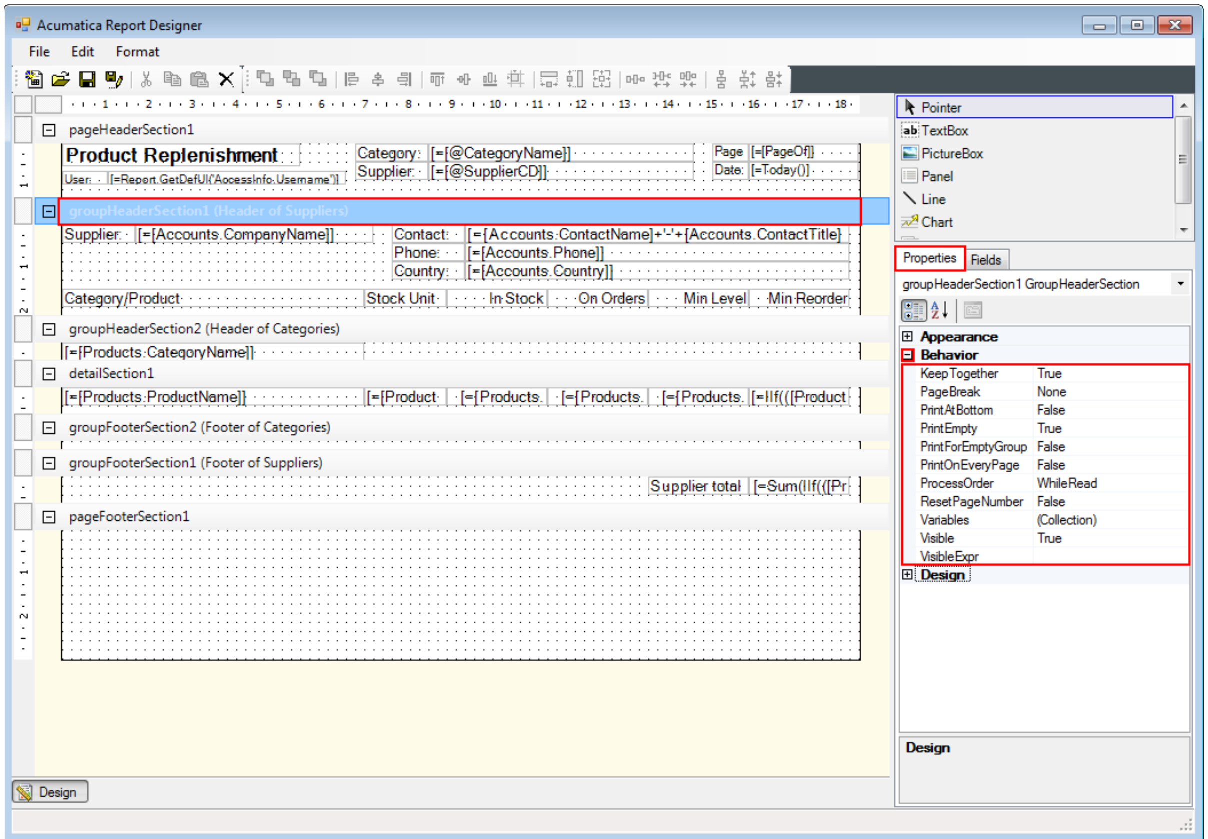


Figure: Defining the Behavior Settings of a Report Section

2. On the **Properties** tab, in the **Behaviors** group, specify the appropriate settings. The properties are listed and described below based on the section type.

Behavior Settings of the Report Header and Report Footer Sections

Property	Description
KeepTogether	A setting that defines whether the lines in this section should be printed on the same page.
PageBreak	A specification of where in this section the page break should be added: *Before*, *After*, or *BeforeAndAfter*.
PrintAtBottom	A setting that defines whether the lines in this report section are printed at the bottom of the page.
PrintEmpty	A setting that specifies whether empty lines are printed in this report section.
ProcessOrder	The processing order of the data within the section.
ResetPageNumber	A setting specifying whether page numbering is reset when a new section starts.
Variables	A listing of the variables defined for the section. These variables are visible within the whole report, but are calculated within the sections where they are defined.

Property	Description
Visible	The report section's visibility property (*False* or *True*). The invisible (hidden) sections are not printed in the report.
VisibleExpr	The expression that calculates the report section visibility property. This value overrides the *Visible* property value if it was set explicitly.

Behavior Settings of the Page Header and Page Footer Sections

Property	Description
PrintAtBottom	A setting that defines whether the lines in the report section are printed at the bottom of the page.
PrintEmpty	A setting that specifies whether empty lines are printed in this report section.
PrintOnFirstPage	A setting that defines whether the page header data is printed on the first page of the report.
PrintOnLastPage	A setting determining whether the page header data is printed on the last page of the report.
ProcessOrder	The processing order of the data within the section.
ResetPageNumber	A setting specifying whether page numbering is reset when a new section starts.
Variables	A listing of the variables defined for the section. These variables are visible within the whole report, but are calculated within the sections where they are defined.
Visible	The report section's visibility property (*False* or *True*). The invisible (hidden) sections are not printed in the report.
VisibleExpr	The expression that calculates the report section visibility property. This value overrides the *Visible* property value if it was set explicitly.

Behavior Settings of the Group Header and Group Footer Sections

Property	Description
KeepTogether	A setting that defines whether the lines in this section should be printed on the same page.
PageBreak	A specification of where in this section the page break should be added: *Before*, *After*, or *BeforeAndAfter*.
PrintAtBottom	A setting that defines whether the lines in the report section are printed at the bottom of the page.
PrintEmpty	A setting that specifies whether empty lines are printed in this report section.
PrintForEmptyGroup	A setting defining whether empty data groups are printed in the report section.
PrintOnEveryPage	A setting determining whether the section data is printed on every page of the report.
ProcessOrder	The processing order of the data within the section.

Property	Description
ResetPageNumber	A setting specifying whether page numbering is reset when a new section starts.
Variables	A listing of the variables defined for the section. These variables are visible within the whole report, but are calculated within the sections where they are defined.
Visible	The report section's visibility property (*False* or *True*). The invisible (hidden) sections are not printed in the report.
VisibleExpr	The expression that calculates the report section visibility property. This value overrides the *Visible* property value if it was set explicitly.

Behavior Settings of the Detail Section

Property	Description
KeepTogether	A setting that defines whether the lines in this section should be printed on the same page.
PageBreak	A specification of where in this section the page break should be added: *Before*, *After*, or *BeforeAndAfter*.
PrintAtBottom	A setting that defines whether the lines in the report section are printed at the bottom of the page.
PrintEmpty	A setting that specifies whether empty lines are printed in this report section.
ProcessOrder	The processing order of the data within the section.
ResetPageNumber	A setting specifying whether page numbering is reset when a new section starts.
Variables	A listing of the variables defined for the section. These variables are visible within the whole report, but are calculated within the sections where they are defined.
Visible	The report section's visibility property (*False* or *True*). The invisible (hidden) sections are not printed in the report.
VisibleExpr	The expression that calculates the report section visibility property. This value overrides the *Visible* property value if it was set explicitly.

References

- *Using Variables*

Adding and Removing Visual Elements in the Report

The Tools area on the Acumatica Report Designer form (in the upper right) displays the visual elements that can be added to the report. You can add any of these visual elements to a report section or remove it from the section.

Adding a Visual Element

To add a visual elements to a report section, select the element in the Tools area, and place it in the desired position within the report by dragging and dropping it. You can resize the element by dragging its borders.

After a visual element is added on the screen, you can do the following actions to it:

- Define the style of the element, and reset the style if desired
- Copy and paste the style between visual elements
- Define the order of visual elements on the screen by bringing them to the front or sending them to the back
- Cut, copy, and paste visual elements and their content to other areas within the report

To perform these actions with a visual element, use the commands available in the Report Designer toolbar, or right-click the visual element and select the relevant command from the pop-up menu.

Removing a Visual Element

To remove a visual element, you select the element in the report section by clicking it, and press the DELETE key.

Data Grouping and Sorting

The data in reports can be divided into several groups, each of which displays the data sorted in the order selected for the group. The sorting criteria are defined separately for every report group and for the report as a whole.

To set up the data grouping and sorting rules, you should define the following:

- The data groups to be included in the report and their grouping rules
- The data sorting rules for the report
- The report's parameters

In This Section

This section includes the following articles:

- *Defining the Data Groups and Grouping and Sorting Rules for a Report*
- *Defining Parameters for a Report*
- *Using Filters*

Defining the Data Groups and Grouping and Sorting Rules for a Report

Data groups are used to structure and logically group data in a report. You can add new data groups to the report and define the behavior properties for each group. The groups' data will be displayed on the pages of the generated report.

To define the data groups in a report, perform the following steps:

1. Select the whole report as an object for which properties will be set by clicking the icon in the top left corner of the Report Designer screen.
2. On the **Properties** tab, click the button next to the **Groups** collection. The **Group Collection Editor** dialog appears; using the dialog, you can add, remove, or modify the data groups.
3. Select the top level of the report (click the icon left of the red 1 in the screenshot below). On the **Properties** tab, locate the **Groups** property under the **Data** section (see item 2). Open the **GroupExp Collection Editor** window by clicking the button next to the **Groups** (Collection) property.

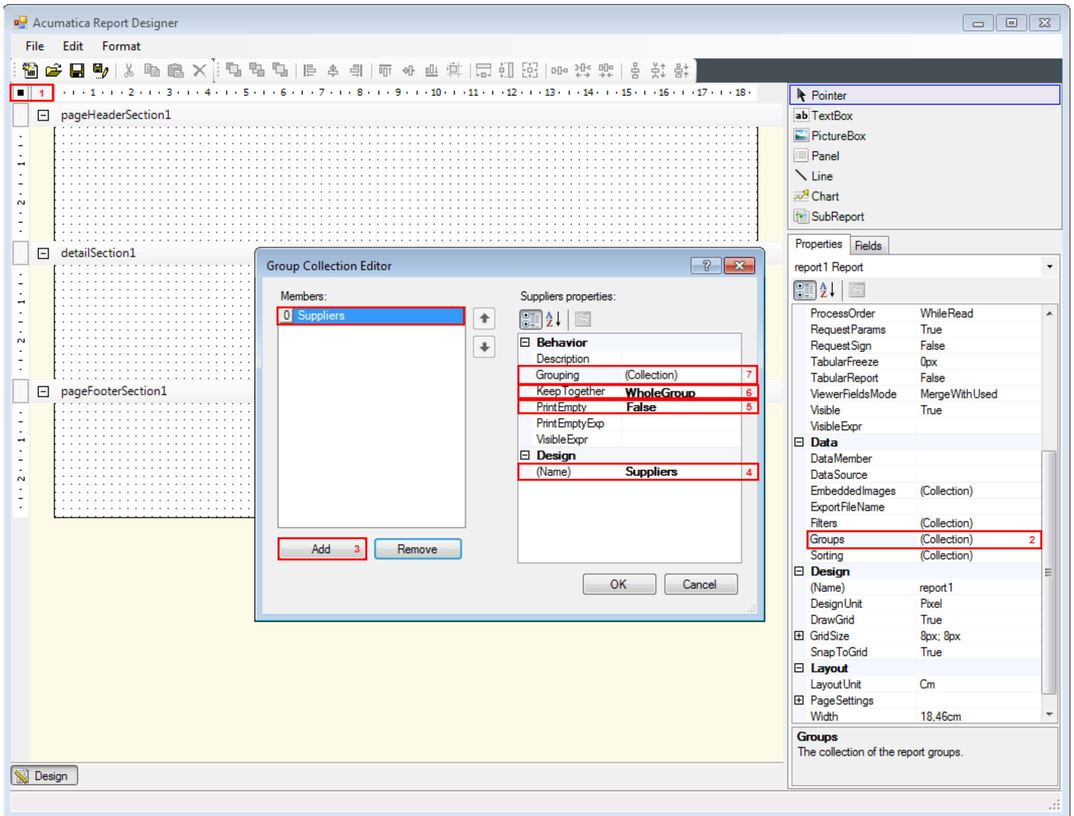


Figure: Adding the Suppliers group

4. Click the **Add** button (item 3) to create the new group section.
5. Change the group name to required (item 4), set the **PrintEmpty** property (item 5), and the **KeepTogether** property (item 6) as you need.
6. To specify the group description, enter the **Description** value in the **Behavior** set of group properties. The description will be displayed in the group header. To define the group description, you can use the **Expression Editor** dialog, as documented in the *Using the Expression Editor* article.
7. Locate the **Grouping** property and click the button (item 7) to open the **GroupExp Collection Editor** window.
8. Click the **Add** button to create a new grouping expression (see item 1 in the screenshot below). Specify the appropriate **DataField** property (item 2) and its **SortOrder** property as *Ascending* or *Descending*.

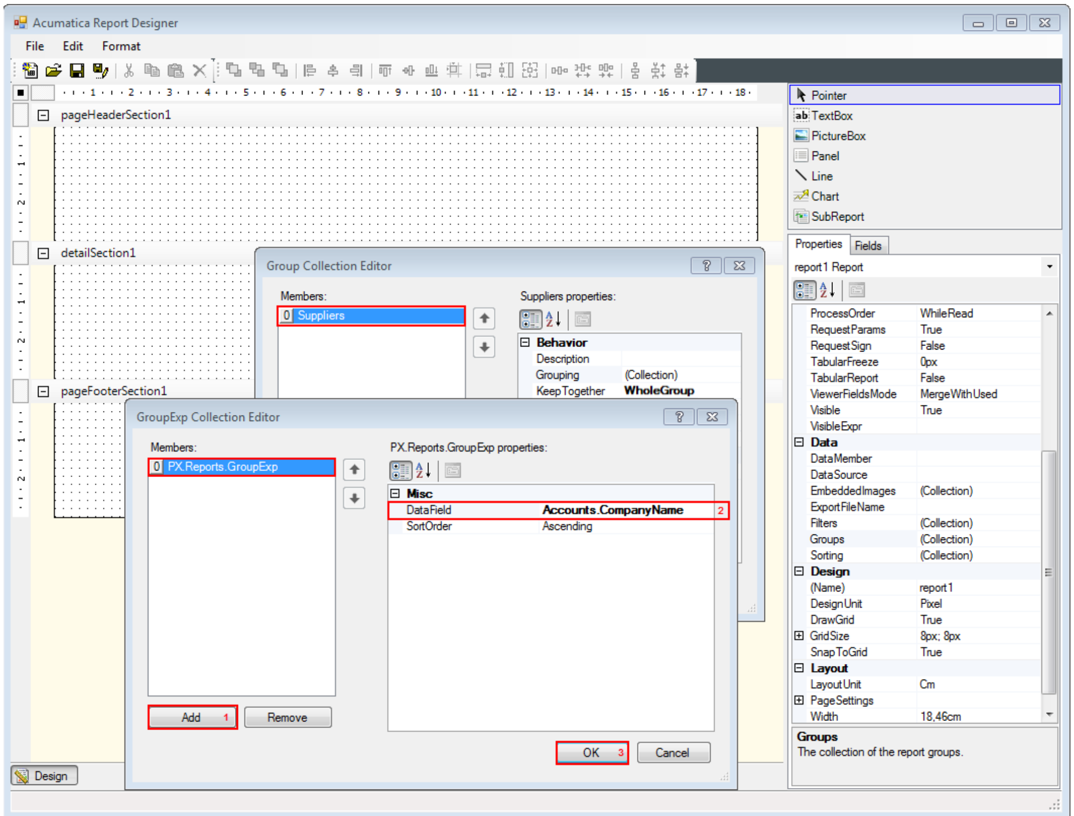


Figure: Configuring the Suppliers group

9. Click **OK** (item 3) to close the **GroupExp Collection Editor** window.
10. If it's required, repeat the appropriate actions above to add the second group section with grouping on another field (see the screenshots below).
11. Click **OK** to close the **GroupExp Collection Editor** window. Both group sections have been added to the report page.
12. Click **Save**.

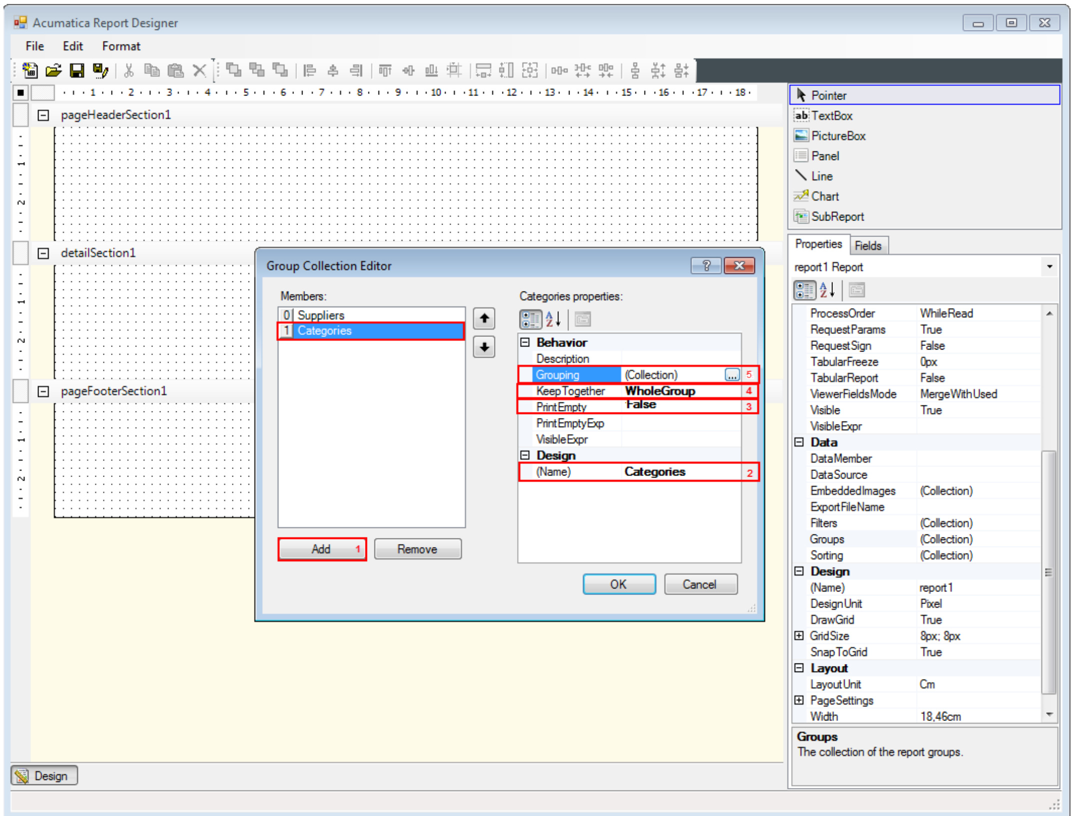


Figure: Adding the Categories group

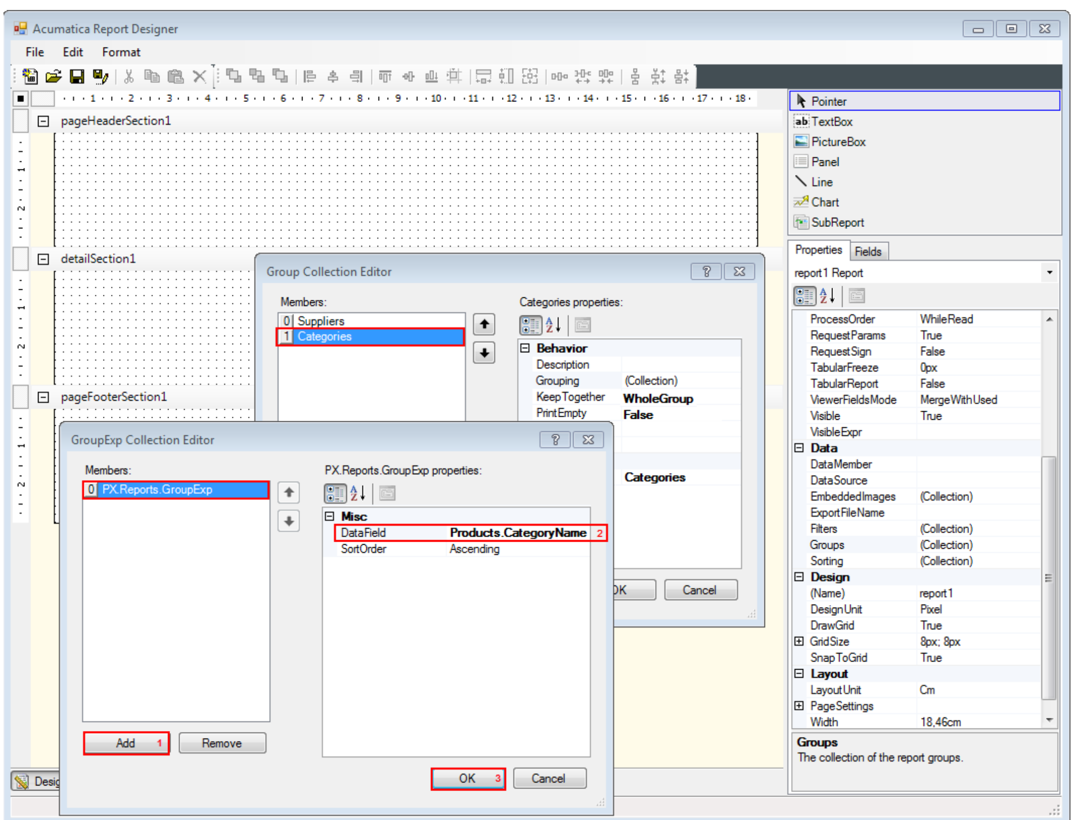


Figure: Configuring the Categories group

By defining groups, you specify sorting conditions for the SQL that is generated by the report, as well as adding the group footer and header section onto the report form in the designer.

References

- *Using the Expression Editor*

Defining Parameters for a Report

You can use parameters to share values between two or more reports, or in expressions and formulas to calculate values for multiple fields within the same report. Parameters are variables that are requested from the user before the report is executed. Based on the parameter, the report engine creates a variable within the report, which can be referred to as a database field can. When referred to from code, a parameter starts with the @ symbol.

To define a parameter for a report, perform the following steps:

1. Start the Schema Builder wizard by selecting the *Build Schema* command from the **File** menu.
2. Open the **Parameters** tab. The list of parameters defined for the report is displayed in the left area of the tab.
3. Click **Add** (see the item with the red 1 in the first screenshot below) to add a new parameter to the parameters list.Alternatively, to change properties of an existing parameter, click its name in the parameters list.
4. In the **Name** box, enter the parameter name (item 2).
5. In the **Input Mask** box, define the input mask for the parameter, if necessary.
6. In the **Data Type** box, select the data type for the parameter (item 3).
7. In the **Prompt** box, enter the prompt for the parameter (the label to be displayed on the screen —see item 4).
8. In the **Default Value** box, enter the default value for the parameter. You can use expressions and formulas to define parameters' default values.
9. In the **Column Span** box, set the column span to display the parameter.
10. Set the appropriate check boxes for the parameter (item 5):
    - **Allow Null** - To indicate that the parameter can have *Null* values
    - **Visible** - To display the parameter on the screen
    - **Required** - To indicate that the parameter is required for the report
11. In the **View Name** box, enter the view formula used to retrieve data for the parameter (item 6). The **View name** property specifies the lookup window that will open to help the user select the parameter. The *Report.GetView()* function creates the lookup field by using the **PXSelector** attribute declared on the DAC field; the DAC field is passed as a function parameter.

    You can also use any field of any existing outside DAC, if it has an attribute with appropriate lookup columns for the report parameter being adjusted. You can create a special DAC with needed lookup fields if you haven't found the appropriate field or fields in the existing DACs.
12. In the **Value** column of the **Available Values** table, you can enter the value of the expression. If more than one value may be used for the parameter, add another value to the list of available values in a separate row.
13. Add the label that will be displayed when the parameter has the corresponding value.

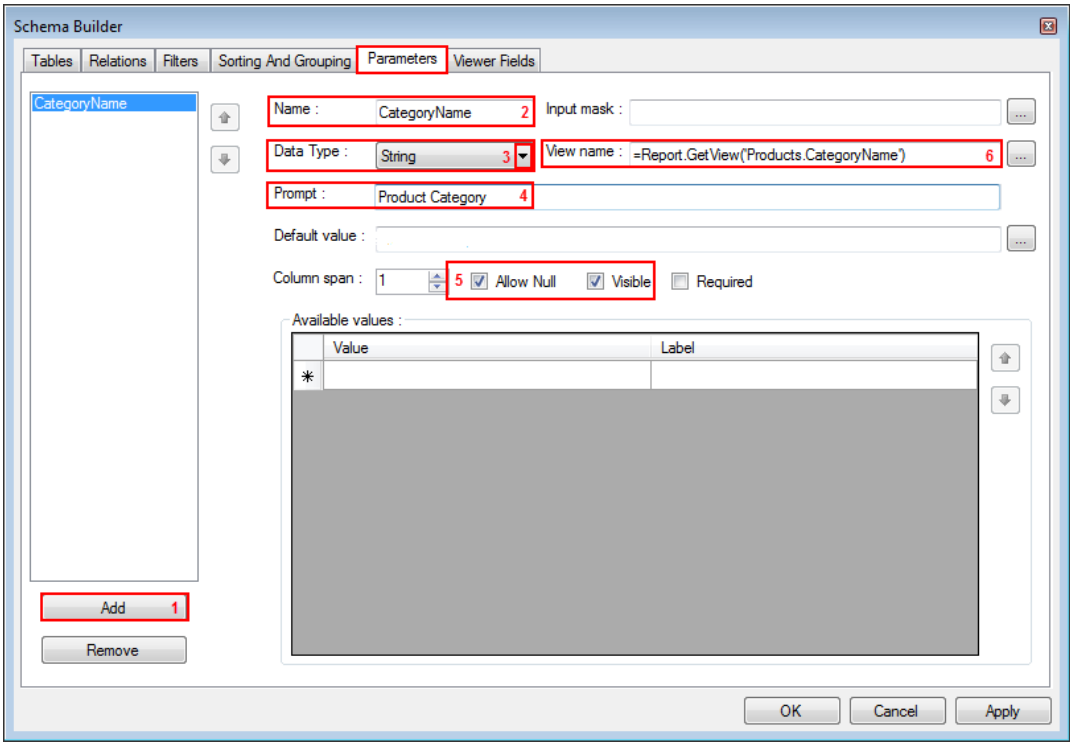


Figure: Adding the first parameter

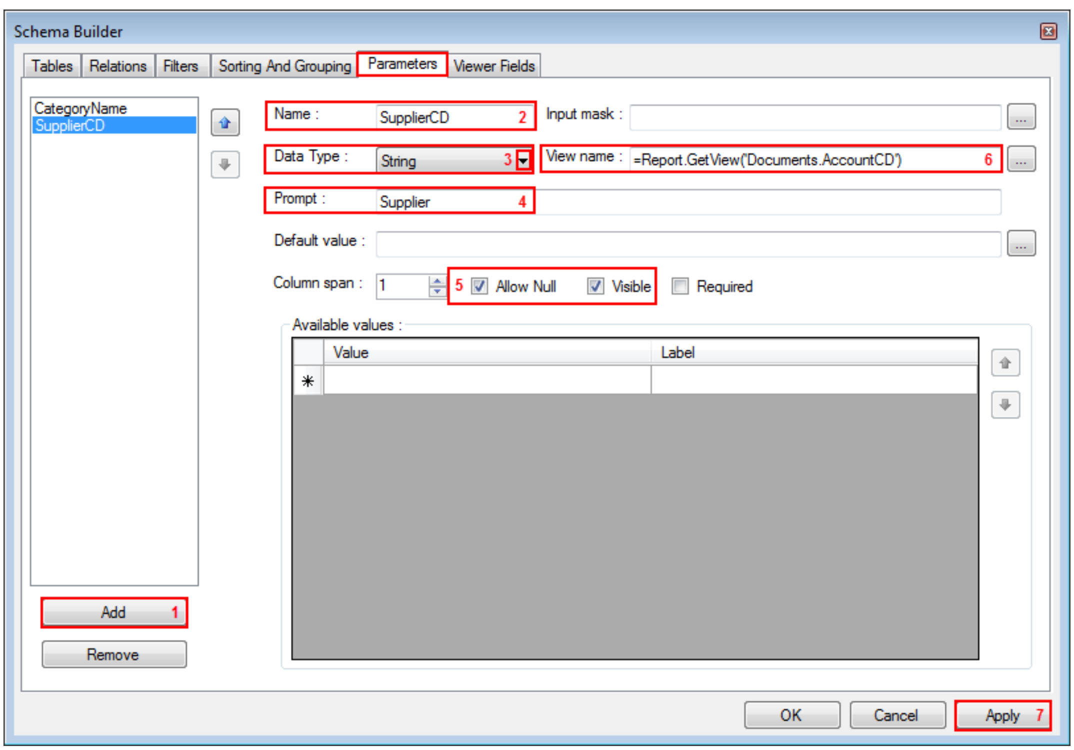


Figure: Adding the second parameter

14. To apply the changes, click the **Apply** button.

15. To save the parameters defined for the report, and their values, click **OK**; otherwise, click **Cancel**.

Using Filters

Filters allow you to limit the volume of data selected for the reports, set more specific criteria for selecting data from data tables, and remove unnecessary data as a result of the table joining. The **Filters** tab of the Schema Builder wizard lists the data filtering rules defined for the current report, which you can modify. Data filtering rules can also be set on the **Properties** tab.

Using the Schema Builder Wizard

Filter expressions use the data field names and parameters to set the criteria for data processing. To set a filter using the Schema Builder wizard, perform the following steps:

1. In the grid on the **Filters** tab, click the empty line to add a new expression to the filter.
2. In the **Data Field** field, select a data field or parameter name.
3. In the **Condition** field, select the appropriate condition for the expression: *Equal*, *NotEqual*, *Greater*, *GreaterOrEqual*, *Less*, *LessOrEqual*, *Like*, *RLike*, *LLike*, *Between*, *IsNull*, or *IsNotNull*.
4. In the **Value1** and **Value2** fields, enter the value or values for the expression.
5. If more than one data filtering expression will be used for filtering data, in the **Operator** field, select the operator: *And* or *Or*.
6. Select the braces in the **Braces** column if they are required in the data filtering expressions.
7. Repeat these steps for each expression to be used in the data filtering rule.

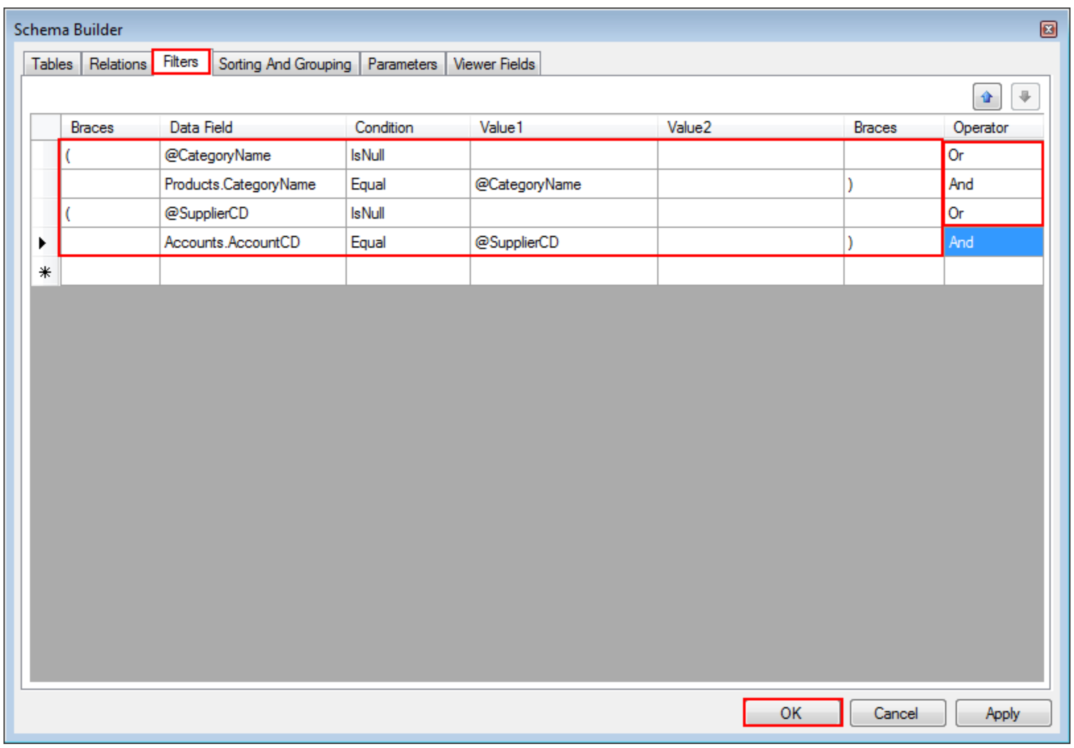


Figure: Configuring the filter

8. Click **Apply** to apply the changes.
9. Click **OK** to save the changes and close the Schema Builder wizard, or **Cancel** to discard the changes.

Any defined expressions can be deleted. To delete an expression, click the relevant line in the grid, and press the DELETE key. On the **Filters** tab, you can add additional filtering conditions to be transformed to the SQL *Where* condition.

Using the Properties Tab

The **Properties** tab allows you to define the data filters as well. To set a filter and define the data filtering criteria, perform the following steps:

1. Select the whole report as an object for which the properties will be set by clicking the ■ icon in the top left corner of the Report Designer window. The **Properties** tab displays the report properties.
2. On the **Properties** tab, click the ... button next to the **Filters** collection. The **FilterExp Collection Editor** dialog appears, allowing you to edit the filter expressions.

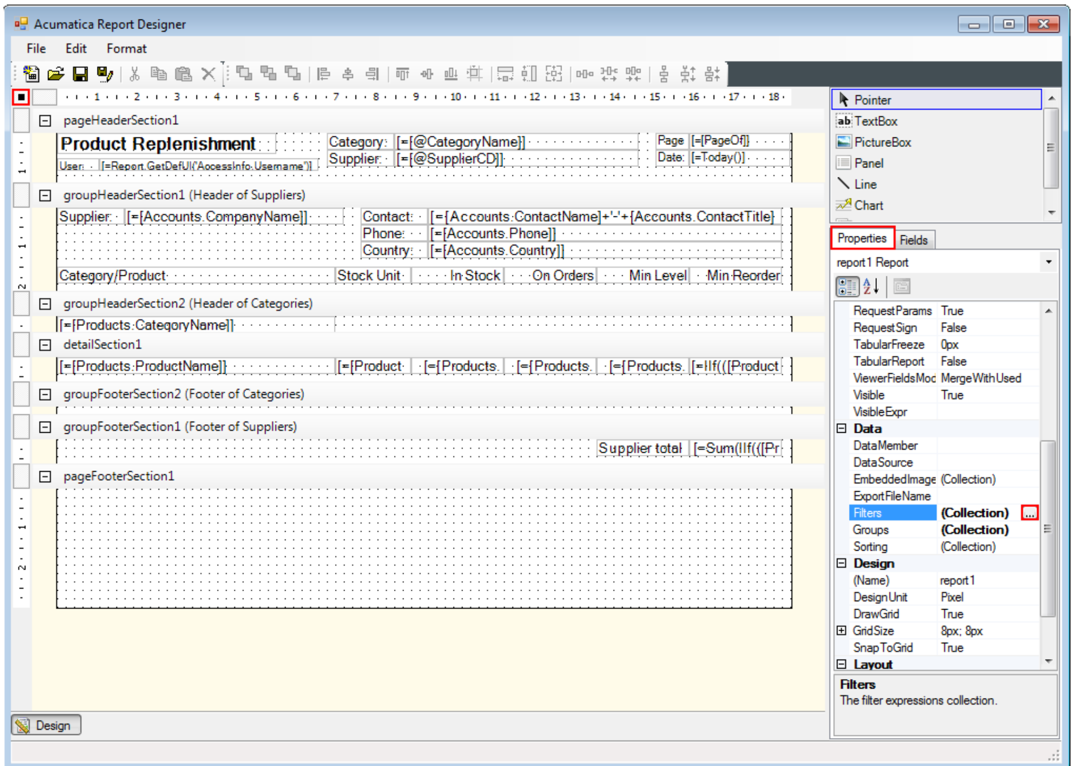


Figure: Select Filters

3. To add a new expression to the filter, click the **Add** button under the **Members** list. The new expression will be added to the list of filter expressions and selected for editing.
4. In the **Data Field** field, select the data field or parameter name.
5. In the **Condition** field, select the condition for the expression: *Equal*, *NotEqual*, *Greater*, *GreaterOrEqual*, *Less*, *LessOrEqual*, *Like*, *RLike*, *LLike*, *Between*, *IsNull*, or *IsNotNull*.
6. In the **Value** and **Value2** fields, enter the value or values for the expression.
7. If more than one data filtering expression will be used for filtering data, in the **Operator** field, select the operator: *And* or *Or*.
8. In the **Open Braces** field, enter the number of the opening braces to be added before the new expression.
9. In the **Close Braces** field, enter the number of the closing braces to be added after the new expression.
10. Repeat Steps 3–9 for each expression to be used in the data filtering rule.

11. Click **OK** to save the changes and close the **FilterExp Collection Editor** dialog, or **Cancel** to discard the changes.

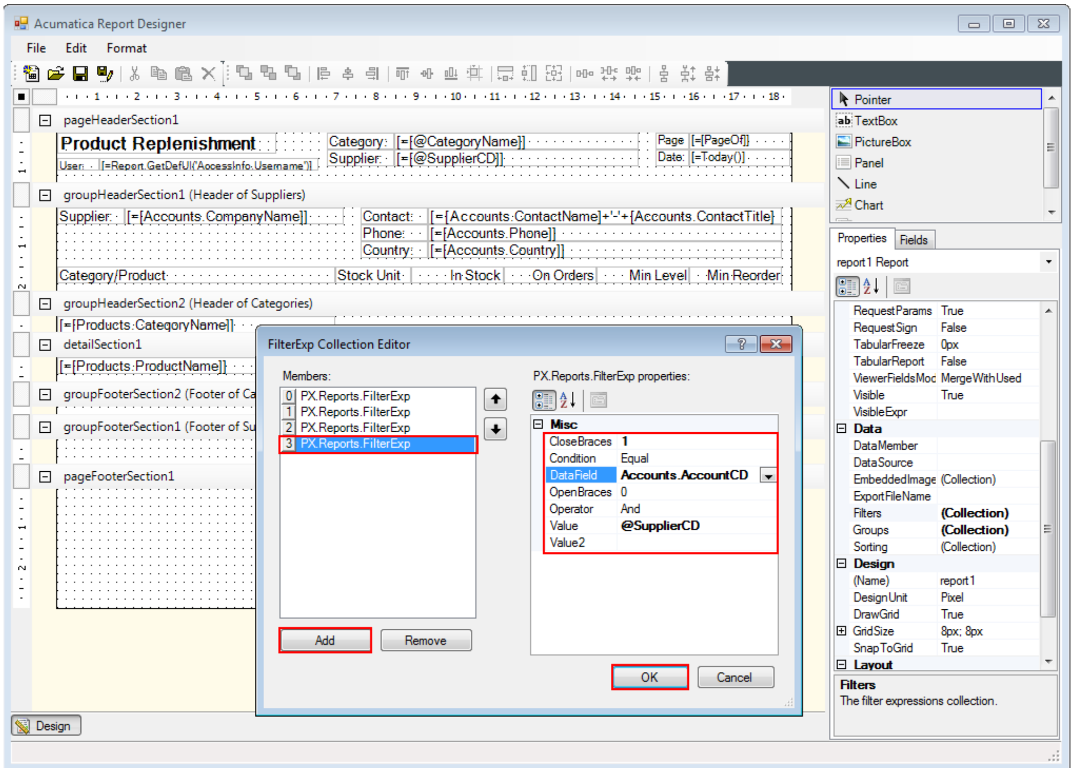


Figure: Define the filtering rules

The defined expressions can be deleted. To delete an expression, click the relevant item in the **Members** list, and click the **Remove** button.

Using Expressions

Expressions are used to define the data values to be displayed in the report or the internal variables used to set the report properties, including report visibility, the group description, and the parameter determining whether the empty lines will be printed in the report.

To help you define expressions, the Report Designer provides the **Expression Editor** dialog.

In This Section

This section includes the following articles:

- *Using the Expression Editor*
- *Using Globals, Parameters, and Local Variables*
- *Using Operators in Expressions*
- *Using Functions in Expressions*

Using the Expression Editor

To define an expression for a report parameter, you use the **Expression Editor** dialog, which you invoke by clicking the ... button on the **Properties** tab for a property, as shown in the screenshot below. (The most common example is setting the **Value** property for a text box inserted in the report.)

Using the **Expression Editor** dialog, you can enter the expression directly or compose it by selecting the appropriate values, global variables, report variables, parameters, operators, and functions.

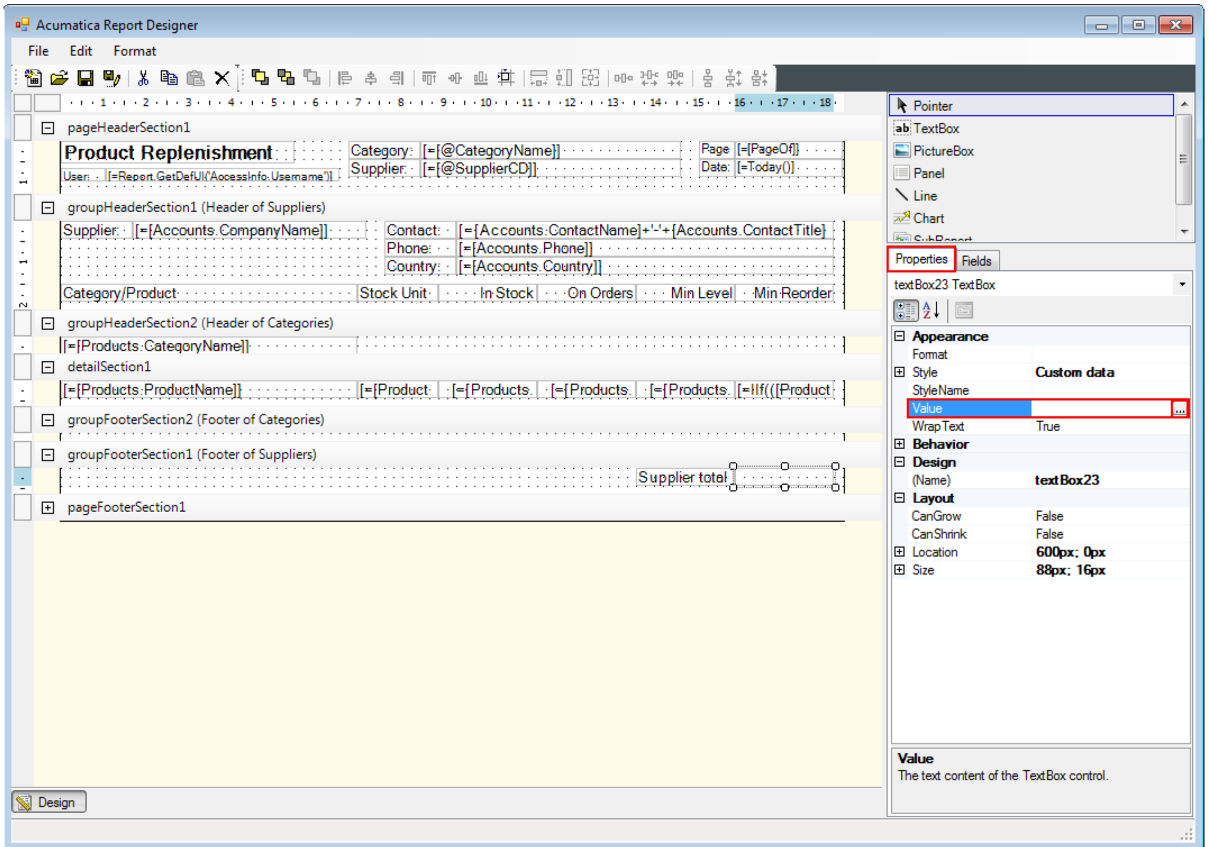


Figure: Invoking the Expression Editor dialog

The **Expression Editor** dialog consists of four areas:

- Report Attributes area (left area of the dialog) - This area displays the list of the attributes defined for this report by the data schema it uses.
- Parameters, Variables, Operators, and Functions area (middle area of the dialog) - This area lists the parameters, operators, functions, and variables available in the report.
- Parameter, Variable, Operator, and Function Selection area (right area of the dialog) - This area allows selecting the specific parameters, operators, and functions to be used in expressions.
- Expression Editing area (bottom area of the dialog) - This area displays the expression you have composed and allows you to edit it.

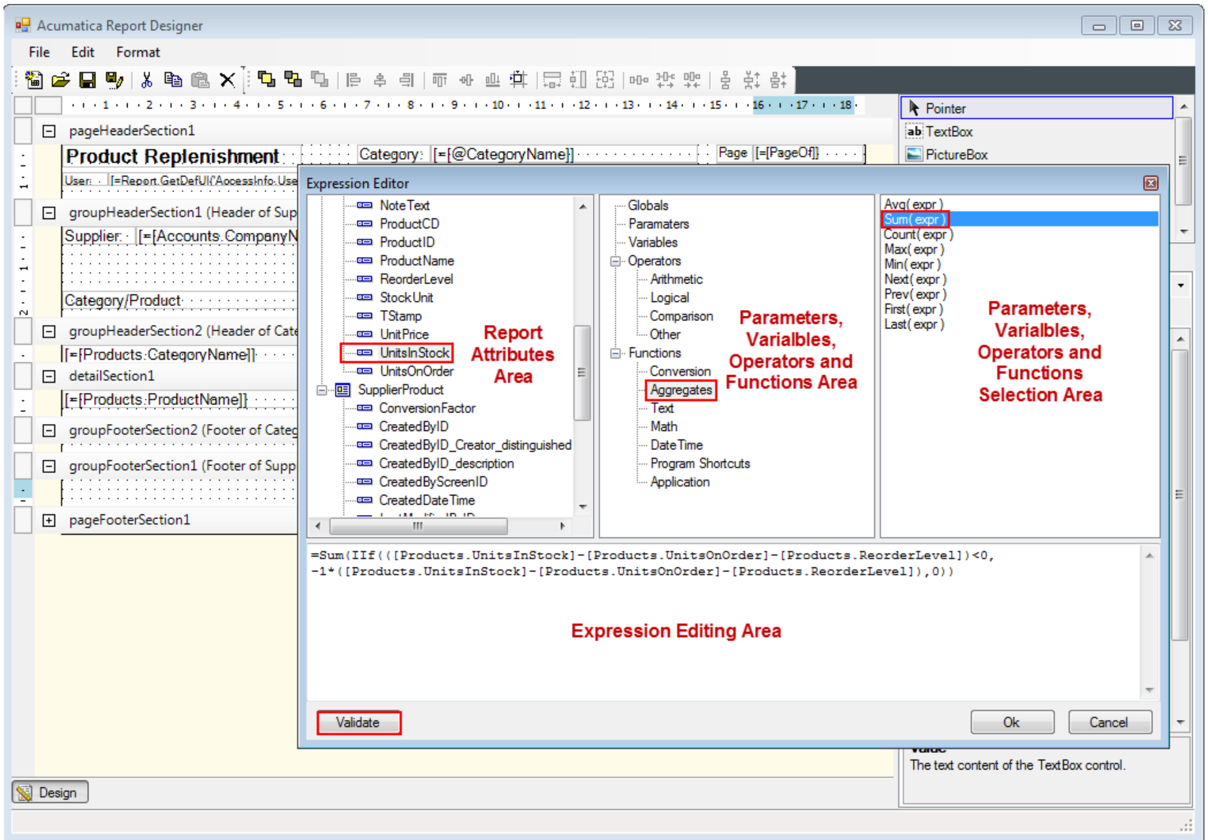


Figure: The Expression Editor window

To enter the expression using the **Expression Editor** dialog, use the following steps:

1. In the Parameters, Variables, Operators, and Functions area, expand the hierarchical structure of the existing entities, and click the link of the group of parameters, variables, functions, or operators to display the list of available items in the selection area.
2. In the Parameter, Variable, Operator, and Function Selection area, select the required item and double-click it to insert the item into the report.
3. In the Expression Editing area, edit the expression.
4. To validate the expression, click the **Validate** button in the lower left.
5. Click **OK** to save the expression or **Cancel** to discard the changes.

Using Globals, Parameters, and Local Variables

Expressions can use global variables, parameters, and local variables to define the data that will be used to calculate the values displayed in the report. These variables and parameters are links to the calculated data, selected from the available data set or defined in the report.

Globals

Global variables (sometimes referred to as *globals*) are available in all reports. Globals can be inserted into a report as values or included in expressions.

Globals

Global	Description
PageIndex	Substitutes into the expression the page index value selected in the current report data source definition.

Global	Description
PageCount	Substitutes into the expression the page count value for the current report.
PageOf	Substitutes into the expression the page number and total page count values for the current report.

Parameters

The parameters defined in the report can be used to substitute values into the expression. Every report has its own set of parameters defined by the user creating or modifying the report. The parameters, defined on the report level, can be modified using the Schema Builder wizard.

Parameters have the `@param_name` format, where *param_name* is the name of the parameter defined in the report.

Here is an example of expressions that use the report parameters.

```
([Categories.CategoryName]=[@CategoryName])
```

In the above example, **Categories.CategoryName** is an attribute available from the data schema, and *@CategoryName* is a report parameter; this is the example of a simple condition. } Here, **ARStatementCycle.AgeDays02** is an attribute available from the data schema, and *@AgeDate* is a report parameter; this is the example of an arithmetic operation.

Variables

The local variables you define for a report can be used to substitute values into the expression. Local variables are defined separately for each report data group, but the visibility of the variables is not limited by the group where the variable is defined. To define a new variable, use the **Properties** page of the report data group.

The variables have the `$variable_name` format, where *variable_name* is the name of the variable defined in the report.

Examples

See below for examples of expressions using local variables:

```
=$Age02
```

Here, *$Age02* is a local report variable.

```
=Assign( '$RowNumber', $RowNumber + 1 )
```

In this example, the row number is calculated; *$RowNumber* is a local report variable.

Using Operators in Expressions

Operators are used to perform certain operations with the data attributes, globals, parameters, and variables or to modify the data before it is inserted into the report.

To add operators in the expressions, you can enter them directly in the expression editing area or select them from the list of operators provided by the Expression Editor, described in the *Using the Expression Editor* article.

You can use the following groups of operators in the expressions.

Arithmetic Operators

Arithmetic operators are used to perform familiar arithmetic operations that involve the calculation of numeric values. The arithmetic operators group includes the following operators.

Arithmetic Operators

Operator	Description and Examples
+ (addition)	Adds the operands and returns the result. **Example:** `Sum([OrderDetails.ExPrice]+[Orders.Freight]` Here, **OrderDetails.ExPrice** and **Orders.Freight** are attributes from the database scheme.
- (subtraction)	Subtracts the second operand from the first and returns the result. **Example:** `[ARPayment.UnappliedBal]-$AgeBal00` Here, **ARPayment.UnappliedBal** is an attribute from the database scheme, and `$AgeBal00` is a report variable.
*** (multiplication)**	Multiplies the two operands and returns the result. **Example:** `[OrderDetails.Quantity]*[OrderDetails.UnitPrice]` In this example, **OrderDetails.Quantity** and **OrderDetails.UnitPrice** are attributes from the database scheme.
/ (division)	Yields the quotient of the operands, which is the first operand divided by the second. **Example:** StCycCustomerTot/$CustomerTot*100}} Here, `$StCycCustomerTot` and `$CustomerTot` are the report variables.
Mod (modulus)	Divides the first integer operand by the second integer operand and returns the remainder, rounded to the nearest integer. **Example:** `[ARStatementCycle.AgeDays02]Mod(7)` In this example, **ARStatementCycle.AgeDays02** is the attribute from the database scheme.

Logical operators

Logical operators evaluate one or two Boolean expressions and return a Boolean result (`True` or `False`). Because these operators evaluate only Boolean expressions, you must use fields whose only values are `True` and `False` (typically check boxes and radio buttons). The logical operators are listed below.

Logical Operators

Operator	Description and Examples
And	Performs logical conjunction on two Boolean expressions: returns `True` if and only if both expressions evaluate to `True`; in other cases, returns `False`. **Example:** `([ARStatementCycle.Day00]<>0)And([ARStatementCycle.Day01]<>0))` In this example, **ARStatementCycle.Day00** and **ARStatementCycle.Day01** are attributes from the database scheme.
Or	Performs logical disjunction on two Boolean expressions: returns `True` if at least one expression evaluates to `True`; returns `False` if neither expression evaluates to *True*. **Example:** `($CurrBal=0)Or([Terms.DayDue00]<[@AgeDate])` Here, `$CurrBal` is the report variable and **Terms.DayDue00** is an attribute from the database scheme.
Not	Performs logical negation on a Boolean expression: returns `True` if and only if the operand is `False`. Logical negation is an unary operator. **Example:** }} In this example, `$CurrBal` is a report variable.

Comparison Operators

Comparison operators compare two expressions and return a Boolean value (`True` or `False`) that represents the result of the comparison. This group of operators includes the following operators.

Comparison operators

Operator	Description and Examples
=	Returns `True` if operands are equal. **Example:** `([Terms.DayDue00]=$DueDate))` In this example, **Terms.DayDue00** is an attribute from the database scheme, and `$DueDate` is a report variable.
<>	Returns `True` if operands are not equal. **Example:** `([RowTerms.CreatedDateTime]<>DueDate)` Here, **RowTerms.CreatedDateTime** is an attribute from the database scheme, and `$DueDate` is a report variable.
<	Returns `True` if the first operand is less than the second one. **Example:** `([Terms.CreatedDateTime]<$DueDate)` Here, **Terms.CreatedDateTime** is an attribute from the database scheme, and `$DueDate` is a report variable.
>	Returns `True` if the first operand is greater than the second one. **Example:** `([Terms.CreatedDateTime]>$DueDate)` In this example, **Terms.CreatedDateTime** is an attribute from the database scheme, and `$DueDate` is a report variable.
<=	Returns `True` if the first operand is less than or equal to the second operand. **Example:** `([Terms.CreatedDateTime]<=$DueDate)` Here, **Terms.CreatedDateTime** is an attribute from the database scheme, and `$DueDate` is a report variable.
>=	Returns `True` if the first operand is greater than or equal to the second operand. **Example:** `([Terms.CreatedDateTime]>=$DueDate)` Here, **Terms.CreatedDateTime** is an attribute from the database scheme, and `$DueDate` is the report variable.

Other Operators

This miscellaneous group of operators includes the following operators and constants.

Other Operators

Operator	Description and Examples
In	A binary operator that returns `True` if the second operand (a string) contains the first operand (which is also a string). **Example** `$AgeTot01 In (100, 501, 579)` In this example, `$AgeTot01` is a report variable.
True	A binary constant used as an operand in logical expressions. **Example** `$AgeTot01 <>0=True` Here, `$AgeTot01` is a report variable.
False	A binary constant used as an operand in logical expressions. **Example:** `$AgeTot01 <>0=False` Here, `$AgeTot01` is the report variable.
Null	A special value, used as an operand in logical expressions, that designates an undefined value. **Example:** `([Terms.Descr]=Null` In this example, **Terms.Descr** is an attribute from the database scheme.

References

- *Using Globals, Parameters, and Variables*
- *Using the Expression Editor*

Using Functions in Expressions

Functions enable you to perform specific tasks that facilitate the processing of data for the reports. Many functions available in the Expression Editor window process the data and return the values you can use in reports.

To use functions in expressions, you can enter them manually in the expression editing area or select them from the list of functions provided by Expression Editor. You can use the following groups of functions in expressions.

Type Conversion Functions

The type conversion functions enable you to convert data from one data type to another. Listed below are the type conversion functions available in the *Conversion* subnode of the *Functions* node in Expression Editor.

Function	Description and Examples
CBool(x)	Converts the expression used as the function argument into a Boolean expression. Returns `False` if the Boolean value is 0; otherwise, returns `True`. **Example:** `CBool($CurrCompanyTot - $CompanyTot)` In this example, `CurrCompanyTot` and `CompanyTot` are report variables.
CDate(x)	Converts the expression used as the function argument into a value of the *Date* type. The argument should be a valid date expression according to the locale selected for the import or export scenario. **Example:** `CDate($DueDate - 1)` In this example, `DueDate` is a report variable.
CStr(x)	Converts the expression used as the function argument into a string. If the argument is `Null`, the function returns a run-time error; otherwise, it returns a string. **Example:** `CStr($PrintDoc)` Here, `PrintDoc` is a report variable.
CDbl(x)	Converts the expression defined in the function argument into a value of the *Double* type. **Example:** `CDbl($CurrBal/$CurrTot)` Here, `CurrBal` and `CurrTot` are report variables.
CSng(x)	Converts the expression used as the function argument into a value of the *Single* type. If the expression has a value outside the acceptable range for the *Single* type, this function returns an error. **Example:** `CSng($StCycCurrTot/$CompanyTot)` In this example, `StCycCurrTot` and `CompanyTot` are report variables.
CDec(x)	Converts the expression used as the function argument into a value of the *Decimal* type. **Example:** `CDec($CompanyTot)` In this example, `CompanyTot` is a report variable.
CInt(x)	Converts the expression used as the function argument into a value of the *Integer* type. **Example:** `CInt([ARPayment.ExtRefNbr])`

Function	Description and Examples
	In this example, `ARPayment.ExtRefNbr` is an attribute from the database scheme.
CShort(x)	Converts a numeric value to a value of the *Short* type. **Example:** `CShort([ARPayment.ImpRefNbr])` `ARPayment.ImpRefNbr` is an attribute from the database scheme.
CLong(x)	Converts a numeric value to a value of the *Long* type. **Example:** `CLong($CurrTot)` In this example, `CurrTot` is a report variable.

Aggregate Functions

Aggregate functions perform a calculation on a set of values and return a single value. Listed below are the aggregate functions available in the *Aggregates* subnode of the *Functions* node in Expression Editor.

Function	Description and Examples
Avg(*expression*)	Returns the average of all non-null values of the specified expression. **Example:** `Avg($StCycAgeTot00, $StCycAgeTot01)` In this example, `StCycAgeTot00` and `StCycAgeTot01` are report variables.
Sum(*expression*)	Returns a sum of the values of the specified expression. **Example:** `Sum([ARInvoice.TaxTotal], $CurrTot)` In this example, `ARInvoice.TaxTotal` is an attribute from the database scheme, and `CurrTot` is a report variable.
Count(*expression*)	Returns a count of the values from the specified expression. **Example:** `Count($AgeBal00, $AgeBal01)` In this example, `AgeBal00` and `AgeBal01` are report variables.
Max(*expression*)	Returns the maximum value from all non-null values of the specified expression. **Example:** `Max($CurrCompanyTot, $CompanyTot)` In this example, `CurrCompanyTot` and `CompanyTot` are report variables.
Min(*expression*)	Returns the minimum value from all non-null values of the specified expression. **Example:** `Min($CurrCompanyTot, $CompanyTot)` In this example, `CurrCompanyTot` and `CompanyTot` are report variables.
Next(*expression*)	Returns the next value (from the current one) in the specified expression. **Example:** `Next([ARInvoice.LineTotal],[ARInvoice.TaxTotal])` In this example, `ARInvoice.LineTotal` and `ARInvoice.TaxTotal` are attributes from the database scheme.
Prev(*expression*)	Returns the previous value (from the current one) in the specified expression. **Example:** `Prev([ARInvoice.LineTotal],[ARInvoice.TaxTotal])` `ARInvoice.LineTotal` and `ARInvoice.TaxTotal` are attributes from the database scheme.

Function	Description and Examples
First(*expression*)	Returns the first value in the specified expression. **Example:** `First([ARInvoice.LineTotal],[ARInvoice.TaxTotal])` In this example, `ARInvoice.LineTotal` and `ARInvoice.TaxTotal` are attributes from the database scheme.
Last(*expression*)	Returns the last value in the specified expression. **Example:** `Last([ARInvoice.LineTotal],[ARInvoice.TaxTotal])` In this example, `ARInvoice.LineTotal` and `ARInvoice.TaxTotal` are attributes from the database scheme.

String Functions

String functions, perform an operation on a string input value and return a string or numeric value. Listed below are the string functions available in the *Text* subnode of the *Functions* node in Expression Editor.

Function	Description and Examples
LTrim(*string*)	Removes all leading spaces or parsing characters from the specified character expression, or all leading 0 bytes from the specified binary expression. **Example:** `LTrim(CStr([Contact.LastName]))` In this example, `Contact.LastName` is an attribute from the database scheme.
RTrim(*string*)	Removes all trailing spaces or parsing characters from the specified character expression, or all trailing 0 bytes from the specified binary expression. **Example:** `RTrim(CStr([Contact.LastName]))` In this example, `Contact.LastName` is an attribute from the database scheme.
Trim(*string*)	Removes all trailing spaces or parsing characters from the specified character expression, or all trailing 0 bytes from the specified binary expression. **Example:** `Trim(CStr([Contact.FirstName]+[Contact.MidName]+[Contact.LastName]))` In this example, `Contact.FirstName`, `Contact.MidName`, and `Contact.LastName` are attributes from the database scheme.
Format(*format, argument(s)*)	Replaces the format item in a specified formatting string (*format*) with the text equivalent of the arguments (*arguments*). **Example:** `Format('Curr. Balance: . . . . . . . . {0:N}; Total Amount: . . . . . . . . {1:N}', $CurrBal, $CurrTot)` In this example, `CurrBal` and `CurrBal` are report variables; 0 and 1 are specifiers indicating where the arguments will be inserted; C is the *currency* format specifier; and N is the *number* format specifier.
UCase(*string*)	Returns a string that has been converted to uppercase. The *string* argument is any valid string expression. If *string* contains *Null*, *Null* is returned. **Example:** `UCase(CStr([RowContact.MidName]))`

Function	Description and Examples
	In this example, `RowContact.MidName` is an attribute from the database scheme.
LCase(*string*)	Returns a string that has been converted to lowercase. The *string* argument is any valid string expression. If *string* contains *Null*, *Null* is returned. **Example:** `LCase(CStr([Contact.Email]))` In this example, `Contact.Email` is an attribute from the database scheme.
InStr(*string*, *findString*)	Returns the position of the first occurrence of one string (*findString*) within another (*string*). **Example:** `InStr(CStr([Contact.Email]), '@')` In this example, `Contact.Email` is an attribute from the database scheme.
InStrRev(*string*, *findString*)	Returns the position of the last occurrence of one string (*findString*) within another (*string*), starting from the right side of the string. **Example:** `InStrRev(CStr([Contact.Email]), '@')` In this example, `Contact.Email` is an attribute from the database scheme.
Len(*string*)	Returns an integer containing either the number of characters in a string or the nominal number of bytes required to store a variable. **Example:** `Len(CStr([Contact.Email]))` In this example, `Contact.Email` is an attribute from the database scheme.
Left(*string*, *length*)	Returns a string containing a specified number of characters from the left side of a string. If *string* contains *Null*, *Null* is returned. **Example:** `Left(CStr([Contact.Email]), 7)` In this example, **Contact.Email** is an attribute from the database scheme.
Right(*string*, *length*)	Returns a string containing a specified number of characters from the right side of a string. If *string* contains *Null*, *Null* is returned. **Example:** `Right(CStr([Contact.Email]), 10)` In this example, **Contact.Email** is an attribute from the database scheme.
Replace(*string*, *oldValue*, *newValue*)	Returns a string in which a specified substring (*oldValue*) has been replaced with another substring (*newValue*). **Example:** `Replace(CStr([Contact.Email]), '@.', '@')` In this example, **Contact.Email** is an attribute from the database scheme.
PadLeft(*string*, *width*, *paddingChar*)	Right-aligns the characters in a specified string (*string*), padding with the specified characters (*paddingChar*) on the left for a specified total width (*width*). **Example:** `PadLeft(CStr([Contact.Email]), 7, '@')` In this example, **Contact.Email** is an attribute from the database scheme.
PadRight(*string*, *width*, *paddingChar*)	Left-aligns the characters in a specified string (*string*), padding with the specified characters (*paddingChar*) on the right for a specified total width (*width*).

Function	Description and Examples
	Example: `PadRight(CStr([Contact.Email]), 10, '@')` In this example, **Contact.Email** is an attribute from the data scheme.

Mathematical Functions

Mathematical functions perform calculations, usually based on input values provided as arguments, and return numeric values. Listed below are the mathematical functions available in the *Math* subnode of the *Functions* node in Expression Editor.

Function	Description and Examples
Abs(x)	Returns the absolute value of a number. **Example:** `Abs($CurrBal - $CurrTot)` In this example, `CurrBal` and `CurrTot` are the report variables.
Floor(x)	Returns the largest integer that is not greater than the argument. **Example:** `Floor([Contact.NoteID])` In this example, `Contact.NoteID` is an attribute from the database scheme.
Ceiling(x)	Returns the smallest integer that is not less than the argument. **Example:** `Ceiling([Contact.NoteID])` In this example, `Contact.NoteID` is an attribute from the database scheme.
Round(x, *decimals*)	Returns a numeric expression, rounded to the specified precision (*decimals*). **Example:** `Round($CurrTot, 2)` In this example, `CurrTot` is a report variable.
Min(x, y)	Returns the smaller of two values. **Example:** `Min($CurrTot, $CurrCompanyTot)` In this example, `CurrTot` and `CurrCompanyTot` are report variables.
Max(x, y)	Returns the greater of two values. **Example:** `Max($CurrTot, $CurrCompanyTot)` In this example, `CurrTot` and `CurrCompanyTot` are report variables
Pow(x, *power*)	Computes the value of x raised to the specified power (*power*). **Example:** `Pow(([Contact.NoteID], 2))` In this example, `Contact.NoteID` is an attribute from the database scheme.

Date and Time Functions

The date and time functions perform operations on system-generated values and return values of the following types: string, numeric, or *Date/Time*. Listed below are the string functions available in the *DateTime* subnode of the *Functions* node in Expression Editor.

Function	Description and Examples
Now()	Returns the current date and time according to the system date and time settings on the local computer. **Example:** `Now()`
Today()	Returns the current date according to the system date and time settings on the local computer. **Example:** `Today()`

DateAdd(*date*, *interval*, *number*)	Returns a new date calculated by adding the specified number (`nbr`) of time intervals (`int`) to the date (`dt`). The `int` argument specifies the type of time interval, and is one of the following options:
	• `yyyy` - A number (`nbr`) of years will be added to the specified date (`dt`).
	• `q` - A number (`nbr`) of quarters will be added to the specified date (`dt`).
	• `m` - A number (`nbr`) of months will be added to the specified date (`dt`).
	• `y` - Same as *d*; see below.
	• `d` - A number (`nbr`) of days will be added to the specified date (`dt`).
	• `w` - A number (`nbr`) of weekdays will be added to the specified date (`dt`).
	• `ww` - A number (`nbr`) of weeks will be added to the specified date (`dt`).
	• `h` - A number (`nbr`) of hours will be added to the specified date (`dt`).
	• `n` - A number (`nbr`) of minutes will be added to the specified date (`dt`).
	• `s` - A number (`nbr`) of seconds will be added to the specified date (`dt`).
	Examples:
	`DateAdd($DueDate, 'm', -2)`
	`DateAdd(CDate('31/01/1995'), 'm', -2)`
	`DateAdd($DueDate, 'y', -2) DateAdd(Cdate($DueDate), 'd', -2)`
	In these examples, `DueDate` is a report variable.

Year(*date*)	Returns the year, as an integer, extracted from the specified date (`date`).
	Examples:
	`Year([ARPayment.ClearDate])`
	`Year(Cdate($DueDate)) Year($DueDate)`
	`Year(CDate('31/01/1995'))`

Function	Description and Examples
	In these examples, `DueDate` is a report variable, and `ARPayment.ClearDate` is an attribute from the database scheme.

Month(*date*)	Returns the month, as an integer, extracted from the specified date (`date`).
	Examples:
	`=Month([ARPayment.ClearDate])`
	`=Month($DueDate) =Month(Cdate($DueDate))`
	`=Month(CDate('31/01/1995'))`
	In this example, `DueDate` is a report variable, and `ARPayment.ClearDate` is an attribute from the database scheme.

Day(*date*)	Returns the day (as an integer) extracted from the specified date (`date`).
	Examples:
	`Day([ARPayment.ClearDate])`
	`Day($DueDate) Day(Cdate($DueDate))`
	`Day(CDate('31/01/1995'))`
	In these examples, `DueDate` is a report variable, and `ARPayment.ClearDate` is an attribute from the database scheme.

DayOfWeek(*date*)	Returns the day of the week associated with the specified date (`date`) as an integer.
	Examples:
	`DayOfWeek([ARPayment.ClearDate])`
	`DayOfWeek($DueDate)`
	`DayOfWeek(Cdate($DueDate))`
	`DayOfWeek(CDate('31/01/1995'))`
	In this example, `DueDate` is a report variable, and `ARPayment.ClearDate` is an attribute from the database scheme.

DayOfYear(*date*)	Returns the day of the year calculated for the specified date (`date`).
	Examples:
	`DayOfYear([ARPayment.ClearDate])`
	`DayOfYear($DueDate)`
	`DayOfYear(Cdate($DueDate))`
	`DayOfYear(CDate('31/01/1995'))`
	In these examples, `DueDate` is a report variable, and `ARPayment.ClearDate` is an attribute from the database scheme.

Minute(*date*)	Returns the number of minutes extracted from the specified date (`date`).
	Examples:
	`Minute([ARPayment.ClearDate])`
	`Minute($DueDate)`

Function	Description and Examples

	`Minute(Cdate($DueDate))`
	`Minute(CDate('31/01/1995'))`
	In this example, `DueDate` is a report variable, and `ARPayment.ClearDate` is an attribute from the database scheme.

Second(*date*)	Returns the seconds extracted from the specified date (`date`) as an integer.
	Examples:
	`Second([ARPayment.ClearDate])`
	`Second($DueDate) Second(Cdate($DueDate))`
	`Second(CDate('31/01/1995'))`
	In this example, `DueDate` is a report variable, and `ARPayment.ClearDate` is an attribute from the database scheme.

Shortcut Functions

The shortcut functions perform miscellaneous operations. Listed below are the string functions available in the *Math* subnode of the *Program Shortcut* node in Expression Editor.

Function	Description and Examples
IIf(*expression*, *truePart*, *falsePart*)	Returns one of two values, depending on the evaluation of the expression: If the expression evaluates to `True`, the function returns the *truePart* value; otherwise, it returns the *falsePart* value. **Example:** `IIf(($CurrTot-$CurrBal)<>0), CStr([ARRegister.DocBal]), 'No data available')` In this example, `CurrTot` and `CurrBal` are report variables, and **ARRegister.DocBal** is an attribute from the database scheme.
Switch(*expression_1*, *value_1*, *expression_2*, *value_2*, ...)	Returns the value *value_n* that corresponds to the first expression *expression_n* that evaluates to `True`. *expression_1*, *expression_2*, and so on are Boolean expressions. **Example:** `Switch((($CurrTot-$CurrBal)<0), $CurrBal, (($CurrTot-$CurrBal)>0), $CurrTot)` In this example, `CurrTot` and `CurrBal` are report variables.
IsNull(*value*, *nullValue*)	Replaces `NULL` with the specified replacement value. The *value* argument is checked for `NULL`. **Example:** `IsNull($PrintDoc, 'NULL')` In this example, `PrintDoc` is a report variable.
Assign('*$name*', *expression*)	Assigns the result of the expression calculation to the variable specified as the parameter. The function can be used to assign a value to an existing variable, or a new variable can be created with the expression calculation value assigned to it. **Example:** `Assign(PrintDoc, (IsNull([RowARRegister.CustomerID])))`

Function	Description and Examples
	In this example, PrintDoc is a report variable, and **ARRegister.CustomerID** is an attribute from the data scheme).
Assign('*$name*', *expression*, *resetExpression*)	Assigns the result of the expression calculation to the variable specified as the parameter. The *expression* value is assigned to the variable when the variable is set, and the *resetExpression* defines when the variable value should be reset. The function can be used to assign a value to an existing variable, or a new variable can be created and the expression calculation value is assigned to it. **Example:** Assign(<nowiki>'PrintDoc'</ nowiki>, (IsNull([ARRegister.CustomerID])), IsNull([APPayment.AdjFinPeriodID])) In this example, PrintDoc is a report variable, and **ARRegister.CustomerID** is an attribute from the database scheme).

Application-Specific Functions

The application-specific functions are specific for the application in which you will run the report. That is why these functions are not listed the Expression Editor windows. You will need to enter these functions manually.

The following table includes the application-specific functions available in Acumatica Report Designer.

Function	Description and Examples
GetAPPaymentInfo(*accountCD, paymentMethodID, detailID, acctCD*)	Returns the value of the specified AP payment attribute (*detailID*) for specific cash account (*accountCD*), payment method (*paymentMethodID*), and vendor (*acctCD*). The function returns the attribute value as it is specified in the **Payment Instructions** section on the **Payment Settings** tab of the *Vendors* (AP.30.30.00) form. If the specified record is not available, the function returns an empty string. **Example:** Payments.GetAPPaymentInfo('102000','FEDWIRE','INSTRUCTIONS','V0
GetARPaymentInfo(*accountCD, paymentMethodID, detailID, pMInstanceID*)	Returns the value of the specified AR payment attribute (*detailID*) for specific cash account (*accountCD*), payment method (*paymentMethodID*), and customer (*acctCD*). The function returns the attribute value as it is specified on the **Payment Method Details** tab of the *Customer Payment Methods* (AR.30.30.10) form. If the specified record is not available, the function returns an empty string. **Example:** Payments.GetARPaymentInfo('102000','FEDWIRE','ACCOUNTNO','C0003
GetRemitPaymentInfo(*accountCD, paymentMethodID, detailID*)	Returns the value of the specified payment attribute (*detailID*) for specific cash account (*accountCD*), payment method (*paymentMethodID*), and vendor or customer (*acctCD*). The function returns the attribute value as it is specified on the **Remittance Settings** tab of the *Cash Accounts* (CA.20.20.00) form. If the specified record is not available, the function returns an empty string.

Function	Description and Examples
	Example: `Payments.GetRemitPaymentInfo('102000','FEDWIRE','ACCOUNTNO')`

Creating the Report Content

The report content includes visual elements that can contain text, data, and graphics. The visual elements are placed within the report sections, and their appearance and behavior properties are determined by both the properties of the visual elements themselves and the properties of their report section. Adding content to the report generally involves three steps: adding visual elements to the report, linking them with the data to be displayed in the report, and setting the visual elements' properties.

In This Section

The following articles cover the types of content you can add:

- *Adding a Text Box to the Report Section*
- *Adding a Picture Box to the Report Section*
- *Adding a Panel to the Report Section*
- *Adding a Line to the Report Section*
- *Adding Graphics on the Report*
- *Adding a Subreport to the Report*

Adding a Text Box to the Report Section

Text boxes are used to display text or data in the report. Descriptive captions (labels) and data items are placed within the text boxes. The text to be displayed on the label and the data to be displayed in the text box are defined by the **Value** property of the *TextBox* visual element. To display a label in the text box, enter the label text in the **Value** property on the **Properties** tab. To retrieve data from the database, the text boxes use expressions that include the links to the data from the data scheme. (For more details, see *Using Expressions*.)

To add a text box to the report section and define it appropriately, perform the following steps:

1. Add the *TextBox* visual element to the report section, and position it in the desired location. *Adding and Removing Visual Elements in the Report* describes how to add visual elements.
2. Change the name of the text box if necessary (**Name** on the **Properties** tab).
3. Define the text box's properties on the **Properties** tab, as described in the remainder of this article.

Defining the Appearance Properties of the Text Box

Use the following properties, found in the **Appearance** group on the **Properties** tab, to define the appearance of the text box.

Appearance Properties

Property	Description
Format	The format of the data in the text box. You can use the **Expression Editor** dialog to define the data format; for more information, see *Using the Expression Editor*.
Style	The printing style for the text box, set by the the following values:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td rowspan="10"></td><td>BackColor: The background color for the text box.</td></tr>
<tr><td>BackImage: The background image settings for the text box. Enter desired values for the following:
BarCode Type: The required bar code type, selected from the drop-down list with a restricted quantity of bar code types.
Source - The source of the image.
Image: The specific image to be used as the background:
For an embedded image, select the image name.
For an external image, enter the path to the image file.
For an image retrieved from the database, enter the name of the data field where the image is stored.
Repeat: The appropriate value specifying the repeating pattern for the chosen image:
NoRepeat: Adds the specified image with no repeating
RepeatX: Repeats the image horizontally to fill the width of the report section
RepeatY: Repeats the image vertically to fill the height of the report section
Repeat: Repeats the image horizontally and vertically to fill both the width and height of the report section</td></tr>
<tr><td>BorderColor: The border color of the text box. You can define the color for the bottom, left, right, and top border, and set the default border color, which will be applied if no special settings are defined for the specific borders.</td></tr>
<tr><td>BorderStyle: The border line style for the text box. You can define the style for the bottom, left, right, and top border of the text box, and set the default border style, which will be applied if no special settings are defined for the specific borders.</td></tr>
<tr><td>BorderWidth: The border line width for the text box (in pixels). You can define the width of the bottom, left, right, and top border of the text box, and set the default border width, which will be applied if no special settings are defined for the specific borders.</td></tr>
<tr><td>Font: The font settings for the text box. You can select the font name and size and specify whether the following font attributes are applied: bold, italic, strikeout, and underline.</td></tr>
<tr><td>Padding: The padding setting for the text box, which you can specify in pixels for the left side, right side, top, and bottom of the text box.</td></tr>
<tr><td>TextAlign: The text alignment for the text box: Left, Center, Right, or Not Set.</td></tr>
<tr><td>VerticalAlign: The content vertical alignment for the text box: Not Set, Top, Middle, or Bottom.</td></tr>
<tr><td></td></tr>
<tr><td>StyleName</td><td>The name of the style defined for the text box. To assign a descriptive name to a style you have defined for a text box, enter the name. To apply an existing style to the text box, select its name.</td></tr>
</table>

Property	Description
Value	The value to be displayed in the text box. Enter the text here if the text box will display a data label in the report, or use the **Expression Editor** dialog to define the value to be displayed in the text box.
WrapText	The text wrapping for the text box. To wrap the text across a text box, set this value to *True*.

Defining the Behavior Properties of the Text Box

The following properties, found in the **Behavior** group on the **Properties** tab, let you define the data processing order, navigation settings, and visibility settings of the text box.

Behavior Properties

Property	Description
ConvertHtmlToText	A setting that defines whether the data within the text box must be converted to the plain text format. This property is used if a field value may contain formulas with tags.
ExcelCaption	A setting that is used to export a report to Excel when an original report's structure is rather complicated. In such cases, distortions of the Excel format report can take place. Export to Excel becomes simpler if both this and the **ExcelColumn** property is defined (see the next item below) for each data field that is to be exported; the other data fields are not exported to Excel. The **ExcelCaption** property defines column's caption.
ExcelColumn	A setting that is used to export a report to Excel when an original report's structure is rather complicated. Export to Excel becomes simpler if both this and the **ExcelCaption** property is defined (see the previous item) for each data field that is to be exported; the other data fields are not exported to Excel. The **ExcelColumn** property defines the Excel column to which data from the field is to be entered after the export process is done.
Multiline	A setting that defines whether the data within the text box can be displayed in multiple lines.
NavigateMethod	The navigation method for the text box. This setting is used if navigation to a URL should be performed when the user clicks the value displayed in the text box. To use the client for navigation, select *Client*. To use the server for navigation, select *Server*.
NavigateParams	The navigation parameters for the text box, which are used if navigation to a different URL should be performed. To define these parameters, click the button in the box displaying the parameter collection name, and use the External Parameter Collection Editor to define the set of parameters and their values. (For more details, see *Using the External Parameter Collection Editor*.)
NavigateURL	The URL for navigation, used if navigation should be performed when the user clicks the value displayed in the text box.
ProcessOrder	The processing order for the data associated with the text box, which defines when the expression value is calculated: • To process the data while reading, select *WhileRead*. • To process the data while printing, select *WhilePrint*.

Property	Description
	• To process the data while reading and printing, select *Always*.
Target	The command or application to be invoked when the user clicks the value within the text box.
Visible	The text box's visibility property (*False* or *True*). The invisible (hidden) visual elements are not printed in the report.
VisibleExpr	The expression that calculates the text box visibility property. This value overrides the **Visible** property value if it was set explicitly.

Defining the Layout Properties of the Text Box

Use the following group of properties to define the position of the text box on the report page.

Layout Properties

Property	Description
CanGrow	An option that defines whether the text box size can grow if the text or data does not fit into its current size.
CanShrink	An option that defines whether the text box size can shrink to fit the size of the text box content.
Location	The position of the text box on the report page (in pixels). The **Location** values include the horizontal (*x*) and vertical (*y*) coordinates of the text box on the page.
Size	The size of the text box (in pixels). The **Size** values include the width and height of the text box.

Adding a Picture Box to the Report Section

Picture boxes are used to display graphical elements in the report. These graphics can be selected from the set of embedded images, retrieved from the external sources, or selected from the database.

To add a picture box to the report section and define it appropriately, proceed as follows:

1. Add the *PictureBox* visual element to the report section, and position it in the desired location. The *Adding and Removing Visual Elements in the Report* article describes how to add visual elements.
2. Change the name of the picture box if necessary (**Name** on the **Properties** tab).
3. Define the picture box's properties on the **Properties** tab, as described in this article.

Defining the Appearance Properties of a Picture Box

Use the following properties, found in the **Appearance** group on the **Properties** tab, to define the appearance of the picture box.

Appearance Properties

Property	Description
Style	The printing style for the picture box, set by the following values:
	BackColor: The background color for the picture box.
	BackImage: The background image settings for the picture box. Enter desired values for:

Property	Description

	• **Source**: The source of the image.
	• **Image**: The specific image to be used as the background:
	    • For an embedded image, select the image name.
	    • For an external image, enter the path to the image file.
	    • For an image retrieved from the database, enter the name of the data field where the image is stored.
	• **Repeat**: The appropriate value specifying the repeating pattern for the chosen image:
	    • *NoRepeat*: Adds the specified image with no repeating
	    • *RepeatX*: Repeats the image horizontally to fill the width of the report section
	    • *RepeatY*: Repeats the image vertically to fill the height of the report section
	    • *Repeat*: Repeats the image horizontally and vertically to fill both the width and height of the report section

	BorderColor: The border color for the picture box. You can define the color for the bottom, left, right, and top border of the section, and set the default border color, which will be applied if no special settings are defined for the specific borders.
	BorderStyle: The border line style for the picture box. You can define the style for the bottom, left, right, and top border of the picture box, and set the default border style, which will be applied if no special settings are defined for the specific borders.
	BorderWidth: The border line width for the picture box (in pixels). You can define the width of the bottom, left, right, and top border of the picture box, and set the default border width, which will be applied if no special settings are defined for the specific borders.
	Font: The font settings of the picture box. You can select the font name and size and specify whether the following font attributes are applied: bold, italic, strikeout, and underline.
	Padding: The padding setting for the picture box, which you can specify in pixels for the left side, right side, top, and bottom of the report section.
	TextAlign: The text alignment for the picture box: *Left*, *Center*, *Right*, or *Not Set*.
	VerticalAlign: The content vertical alignment for the picture box: *Not Set*, *Top*, *Middle*, or *Bottom*.
StyleName	The name of the style defined for the picture box. To assign a descriptive name to a style you have defined, enter the name. To apply an existing style, select its name.

Defining the Behavior Properties of the Picture Box

The following properties, found in the **Behavior** group on the **Properties** tab, let you define the data processing order, navigation settings, and visibility settings of the picture box.

Behavior Properties

Property	Description
BarcodeSettings	The barcode settings for the picture box. Enter desired values for the following:

Property	Description
	• **AddCheckDigit**: By setting this property to *True*, you allow to print the check digit for the barcode. • **BarHeight**: The barcode height. • **BarWidth**: The barcode width. • **LeftMargin**: The barcode left margin. • **TextMargin**: The barcode text margin. • **TopMargin**:The barcode top margin. • **With ratio**: The value of a bar code ration.
ProcessOrder	The processing order for the data associated with the picture box, which defines when the expression value is calculated: • To process the data while reading, select *WhileRead*. • To process the data while printing, select *WhilePrint*. • To process the data while reading and printing, select *Always*.
Visible	The picture box's visibility property (*False* or *True*). The invisible (hidden) visual elements are not printed in the report.
VisibleExpr	The expression that calculates the picture box visibility property. This value overrides the **Visible** property value if it was set explicitly.

Defining the Data Properties of the Picture Box

These properties allow you to define the source and type of the data for the picture box and to select what image will be displayed.

Data Properties

Property	Description
BarCode Type	The required barcode type, selected from the drop-down list with a restricted quantity of types.
MimeType	The type of media data for the picture box.
Source	The type of data source of the image to be displayed in the picture box. Select one of the available values: • *Embedded*: An embedded image • *External*: An external image • *Database*: A data field
Value	The actual source of data for the picture box: • To define the source of data for an embedded image, select the embedded image name. • To define the source of data for an external image, enter the path to the external image file (with the file name included). • To define the source of data for an image stored in the database, enter the data field name.

Defining the Layout Properties of the Picture Box

Use these properties to define the size and location of the picture box.

Layout Properties

Property	Description
Location	The position of the picture box on the report page (in pixels). The **Location** values include the horizontal (*x*) and vertical (*y*) coordinates of the picture box on the page.
Size	The size of the picture box (in pixels). The **Size** values include the width and height of the picture box.
Sizing	The method of placing and fitting the selected image in the picture box. Select one of the available options: • *AutoSize*: Automatically selects the image size as the size of the picture to be placed in the picture box • *Center*: Places the image in the center of the picture box • *Normal*: Places the image in the left top corner of the picture box • *Fit*: Stretches or shrinks the image to completely fit into the picture box size • *Scale*: Scales the image to fit the picture box size

Adding a Panel to the Report Section

Visual elements are placed on a *panel* to make a new group of elements located and processed together.

To add a panel to a report section and define it appropriately, proceed as follows:

1. Add the *Panel* visual element, and position it in the desired location. The *Adding and Removing Visual Elements in the Report* article describes how to add visual elements.
2. Change the name of the panel if necessary (**Name** on the **Properties** tab).
3. Define the panel's properties on the **Properties** tab, as described in the rest of this article.

Defining the Appearance Properties of the Panel

Use the following properties, found in the **Appearance** group on the **Properties** tab, to define the appearance of the panel.

Appearance Properties

Property	Description
Style	The printing style for the panel, set by the following values:
	BackColor: The background color for the panel.

	BackImage: The background image settings for the panel. Enter desired values for the following:
	• **Source**: The source of the image.
	• **Image**: The image to be used as the background:
	    • For an embedded image, select the image name.
	    • For an external image, enter the path to the image file.

Property	Description
	• For an image retrieved from the database, enter the name of the data field where the image is stored. • **Repeat**: The appropriate value specifying the repeating pattern for the chosen image: • *NoRepeat*: Adds the specified image with no repeating • *RepeatX*: Repeats the image horizontally to fill the width of the report section • *RepeatY*: Repeats the image vertically to fill the height of the report section • *Repeat*: Repeats the image horizontally and vertically to fill both the width and height of the report section
	BorderColor: The border color of the panel. You can define the color for the bottom, left, right, and top border, and set the default border color, which will be applied if no special settings are defined for the specific borders.
	BorderStyle: The border line style for the panel. You can define the style for the bottom, left, right, and top border of the panel, and set the default border style, which will be applied if no special settings are defined for the specific borders.
	BorderWidth: The border line width for the panel (in pixels). You can define the width of the bottom, left, right, and top border of the panel, and set the default border width, which will be applied if no special settings are defined for the specific borders.
	Font: The font settings of the panel; definition of this setting does not change the panel.
	Padding: The padding setting for the panel, which you can specify in pixels for the left side, right side, top, and bottom of the panel.
	TextAlign: The text alignment of the panel; definition of this setting does not affect the panel.
	VerticalAlign: The text alignment of the panel; defining this setting does not affect the panel.
StyleName	The name of the style defined for the panel. To assign a descriptive name to a style you have defined for a text, enter the name. To apply an existing style to the panel, select its name.

Defining the Behavior Properties of the Panel

These properties, found under the **Behavior** group on the **Properties** tab, let you define the data processing order and visibility properties of the panel.

Behavior Properties

Property	Description
ProcessOrder	The processing order for the data associated with the panel, which defines when the expression value is calculated: • To process the data while reading, select *WhileRead*. • To process the data while printing, select *WhilePrint*.

Property	Description
	• To process the data while reading and printing, select *Always*.
Visible	The panel's visibility property (*False* or *True*). The invisible (hidden) visual elements are not printed in the report.
VisibleExpr	The expression that calculates the panel's visibility property. This value overrides the **Visible** property value if it was set explicitly.

Defining the Layout Properties of the Panel

Use these properties to define the panel's size and location properties.

Layout Properties

Property	Description
Location	The position of the panel on the report page (in pixels). The **Location** values include the horizontal (*x*) and vertical (*y*) coordinates of the panel on the page.
Size	The size of the panel (in pixels). The **Size** values include the width and height of the panel.

Adding a Line to the Report Section

Lines are used to divide the report space, direct the eye, or visually separate elements in the report. You can add lines to improve the look and readability of the report.

To add a line to a report section, perform the following steps:

1. Add the *Line* visual element, and position it in the desired location. The *Adding and Removing Visual Elements in the Report* article describes how to add visual elements.
2. Change the name of the line if necessary: Enter it as the **Name** on the **Properties** tab.
3. Define the line's properties, described in this article, on the **Properties** tab.

Defining the Appearance Properties of the Line

Use the following properties, found in the **Appearance** section on the **Properties** tab, to define the appearance of the line.

Appearance Properties

Property	Description
Direction	The direction of the line on the screen: *Horizontal*, *Vertical*, or *Diagonal*.
LineColor	The color of the line.
LineStyle	The style of the line: *Solid*, *Dashed*, or *Dotted*.
LineWidth	The width of the line (in pixels).
Style	The printing style for the line, set by the following:
	BackColor The background color; this setting does not apply to the line.
	BackImage The background image; this setting does not affect the line.
	BorderColor The border color; this setting does not apply to the line.
	BorderStyle The border style; this setting does not affect the line.

Property	Description
	BorderWidth The border width; this setting does not apply to the line.
	Font The font; this setting does not affect the line.
	Padding The padding setting for the line, which you can specify in pixels for the left side, right side, top, and bottom of the line.
	TextAlign The text alignment; this setting does not apply to the line.
	VerticalAlign The vertical alignment; this setting does not apply to the line.
StyleName	The name of the style defined for the line. To assign a descriptive name to a style you have defined for a line, enter the name. To apply an existing style to the line, select its name.

Defining the Behavior Properties of the Line

The following properties, found in the **Behavior** section on the **Properties** tab, let you define the data processing order and visibility properties of the line.

Behavior Properties

Property	Description
ProcessOrder	The processing order for the data associated with the line, which defines when the expression value is calculated: • To process the data while reading, select *WhileRead*. • To process the data while printing, select *WhilePrint*. • To process the data while reading and printing, select *Always*.
Visible	The line's visibility property (*False* or *True*). The invisible (hidden) visual elements are not printed in the report.
VisibleExpr	The expression that calculates the line's visibility property. This value overrides the **Visible** property value if it was set explicitly.

Defining the Layout Properties of the Line

Use these properties to define the line's size and location.

Layout Properties

Property	Description
Location	The position of the line on the report page (in pixels). The **Location** parameter values include the horizontal (*x*) and vertical (*y*) coordinates of the line on the page.
Size	The size of the line (in pixels). The **Size** parameter values include the width and height of the line.

Adding Graphics to a Report

The graphics in the report can be used as background images or illustrations to catch the user's attention or organize information.

To add a graphic to an Acumatica ERP report, you can embed the image file into the report, select the image from an external file, or select a data field and load an image from it. External files are stored on external resources, such as websites or local hosts, accessible from the Acumatica ERP application site where the reports are published; the report stores only the link to the external file where the image file is located. Embedded images, conversely, are stored together with the report file, and are included in the report as its inner elements.

Embedding an Image in the Report

To embed an image in the report, perform the following steps:

1. Select the whole report as an object for which the properties will be set by clicking the icon in the left top corner of the Acumatica Report Designer window.
2. On the **Properties** tab, which displays the report properties, click the button next to the **EmbeddedImages** collection. The **Embedded Images** dialog box appears, which you can use to add or remove the embedded images for the report.

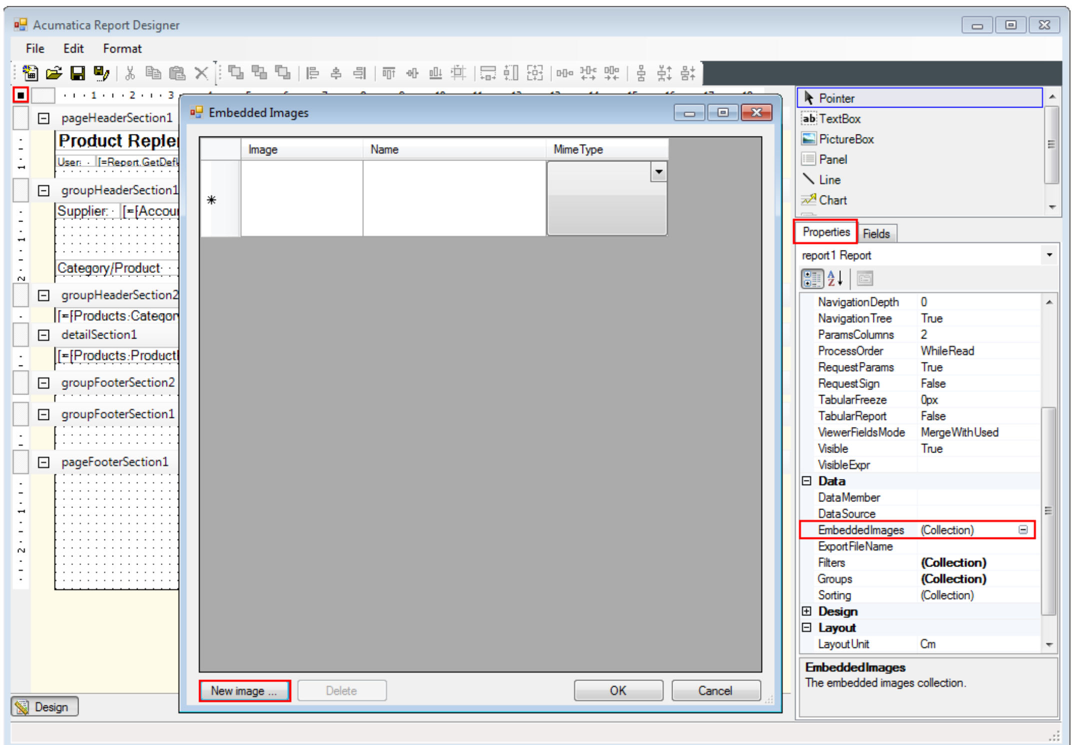


Figure: Embedding a new image

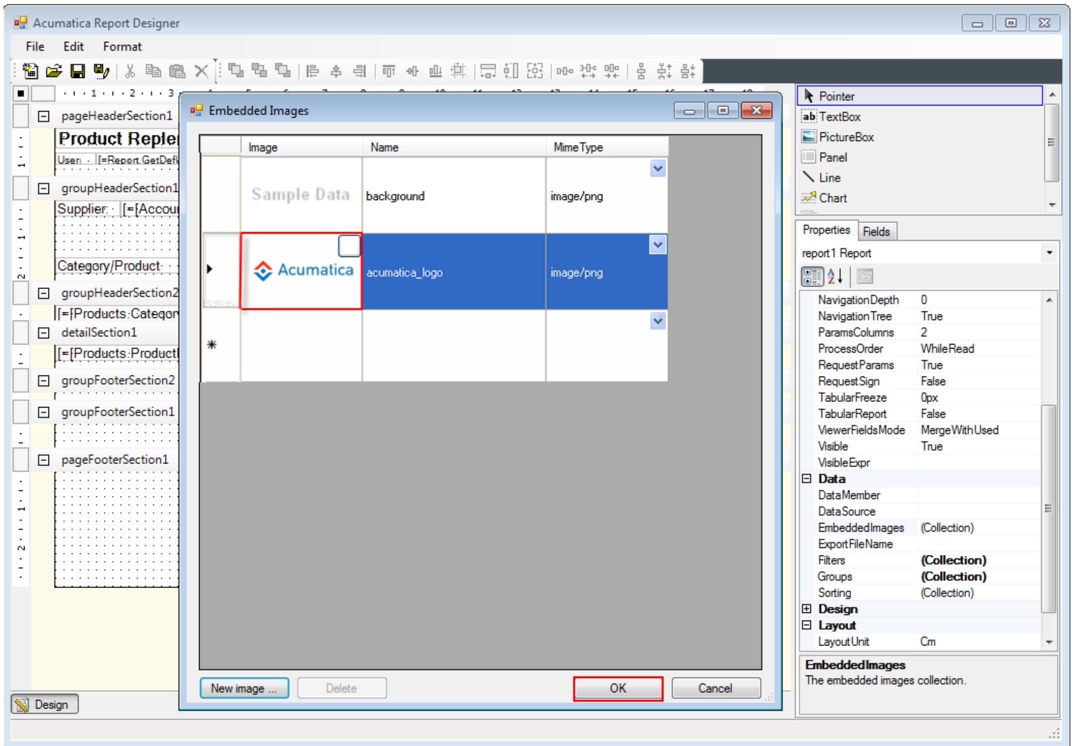


Figure: Saving the embedded images

3. To add a new image, on the **Embedded Images** dialog, click the **New Image** button, or click the button in the empty line of the embedded images list. Select the image to be imported into the report, and add it to the report. To replace the existing image in the report with a new one, click the button next to the image to be replaced, and select a new image to be embedded into the report.
4. To delete an embedded image from the report, click the image in the **Image** list, and click **Delete**.
5. Click **OK** to save your changes.

Adding a Subreport to the Report

Subreports allow you to include data from other reports in the current report. You can add one report or multiple subreports to a single master report.

Adding a Subreport to the Master Report

To include a subreport in the master report, you use the *SubReport* visual element. You can link the subreport to the master report and define the subreport's appearance, behavior, design, and layout properties.

The name of the subreport to be included in the master report is defined by the **ReportName** property of the *Subreport* visual element. If the subreport uses parameters, you need define them in the master report to pass the parameters' values from the master report to the linked report you add to the master report.

To add a subreport to the master report section and define it appropriately, perform the following steps:

1. Add the *SubReport* visual element to the report section, and position it within the report section. (*Adding and Removing Visual Elements in the Report* describes how to add visual elements.) You can add a *SubReport* visual element to only a report header or detail section.

2. Change the name of the subreport if necessary (**Name** on the **Properties** tab).
3. Define the subreport's properties on the **Properties** tab.

Defining the Appearance Properties for the Subreport

Use the following properties, found in the **Appearance** group on the **Properties** tab, to define the appearance of the subreport to be included in the master report.

Appearance Properties

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td rowspan="9">Style</td><td>The printing style for the subreport, set by the following values:</td></tr>
<tr><td>BackColor: The background color for the subreport.</td></tr>
<tr><td>BackImage: The background image settings for the subreport. Enter desired values for the following:

Source - The source of the image.
Image: The image to be used as the background:

For an embedded image, select the image name.
For an external image, enter the path to the image file.
For an image retrieved from the database, enter the name of the data field where the image is stored.

Repeat: The repeating pattern for the chosen image:

NoRepeat: Adds the specified image with no repeating
RepeatX: Repeats the image horizontally to fill the width of the report section
RepeatY: Repeats the image vertically to fill the height of the report section
Repeat: Repeats the image horizontally and vertically to fill both the width and height of the report section

</td></tr>
<tr><td>BorderColor The border color of the subreport. You can define the color for the bottom, left, right, and top border, and set the default border color, which will be applied if no special settings are defined for the specific borders.</td></tr>
<tr><td>BorderStyle: The border line style for the subreport. You can define the style for the bottom, left, right, and top border, and set the default border style, which will be applied if no special settings are defined for the specific borders.</td></tr>
<tr><td>BorderWidth: The border line width for the subreport (in pixels). You can define the width of the bottom, left, right, and top border of the subreport, and set the default border width, which will be applied if no special settings are defined for the specific borders.</td></tr>
<tr><td>Font: The font settings for the subreport. You can select the font name and size and specify whether the following font attributes are applied: bold, italic, strikeout, and underline.</td></tr>
<tr><td>Padding: The padding setting for the subreport, which you can specify in pixels for the left side, right side, top, and bottom of the subreport.</td></tr>
<tr><td>TextAlign: The text alignment for the subreport: Left, Center, Right, or Not Set.</td></tr>
</table>

Property	Description
	VerticalAlign: The content vertical alignment for the subreport: *Not Set*, *Top*, *Middle*, or *Bottom*.
StyleName	The name of the style defined for the subreport. To assign a descriptive name to a style you have defined for a subreport, enter the name. To apply an existing style to the subreport, select its name.
Value	The value to be displayed in the subreport. Enter the text here if the subreport will display a data label in the report, or use the **Expression Editor** dialog to define the value to be displayed in the subreport.
WrapText	The text wrapping for the ssubreport. To wrap the text across a subreport, set this value to *True*.

Defining the Behavior Properties of the Subreport

The following properties, found in the **Behavior** group on the **Properties** tab, let you define the parameters to be passed from the master report to the subreport, specify the data processing order, set the link to subreport in the master report, and define the visibility properties for the subreport.

Behavior Properties

Property	Description
Parameters	The collection of parameters to be used in both master report and the subreport. To add a parameter to the collection, use the External Parameter Collection Editor; for details, see *Using the External Parameter Collection Editor*. If any parameters are defined for the subreport, the number of parameters defined for the master report and subreport must be equal. The names of the parameters used in the master report and subreport should also be the same.
Process Order	The data processing method for the subreport. Choose one of the following options: • *WhileRead*: The subreport data is processed when the subreport is invoked from the master report. • *WhilePrint*: The subreport data is processed when the master report is printed. • *Always*: The subreport data is processed when the master report is active.
ReportName	The subreport name. To select the subreport for inserting it into the master report, click the button in the box where the subreport name is displayed, and select the file of the report to be used as a subreport. The subreport file and the master report file should be located in the same folder.
Visible	The subreport's visibility property (*False* or *True*). The invisible (hidden) visual elements are not printed in the report.
VisibleExpr	The expression that calculates the text box visibility property. This value overrides the **Visible** property value if it was set explicitly.

Defining the Layout Properties of the Subreport

Use the following group of properties to define the position of the subreport on the report page.

Layout Properties

Property	Description
Location	The position of the subreport on the report page (in pixels). The **Location** parameter values include the horizontal (*x*) and vertical (*y*) coordinates of the subreport area on the master report page.
Size	The size of the subreport area (in pixels). The **Size** parameter values include the width and height of the subreport.

Using Variables

Variables are used in reports to calculate values based on the expressions defined for them, store these values, and make them available in all sections of the report.

Adding a Variable to the Report Section

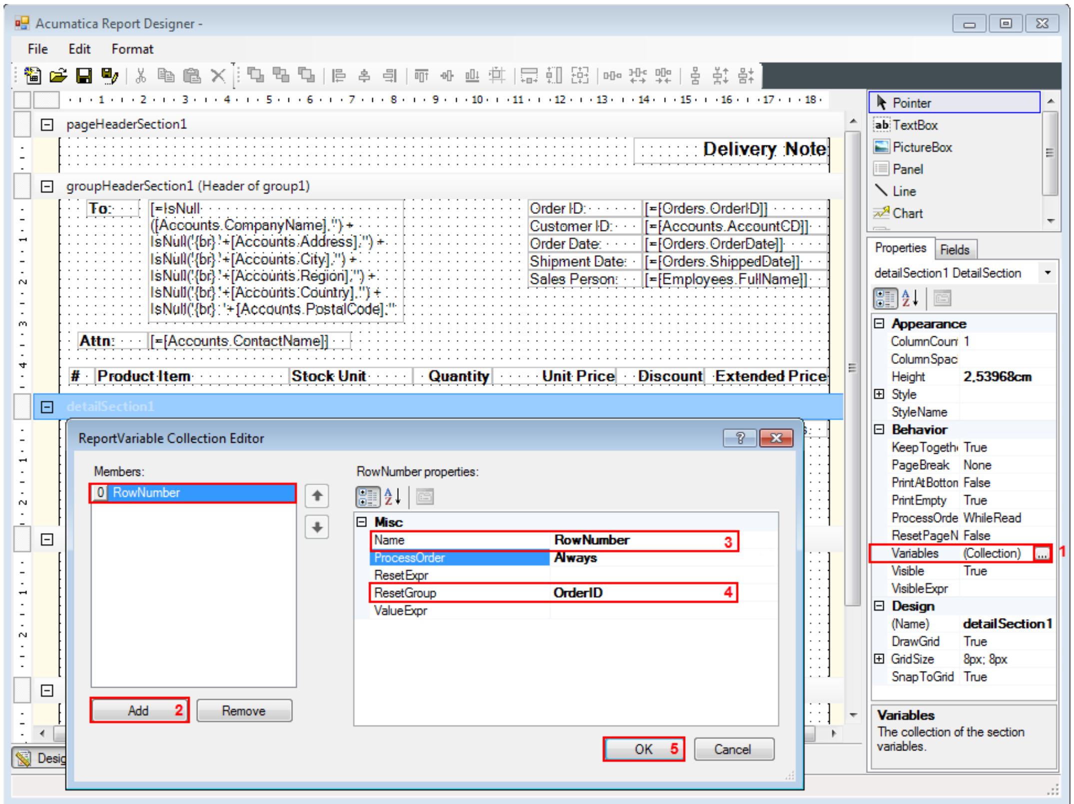
To add a variable to the report section, perform the following steps:

1. Select the report section where you want to add the variable.
2. Click within the **Variables** edit box on the **Properties** tab (shown left of the red 1 in the screenshot below), and the **ReportVariable Collection Editor** window appears. You can use this window to add variables to the report and define their properties.
3. Click **Add** (item 2 in the screenshot). The new variable will be displayed in the **Members** list of the **ReportVariable Collection Editor** dialog.
4. In the **Name** field in the **Misc** section of the **ReportVariable Collection Editor** window, enter the name of the variable (item 3).
5. In the **ProcessOrder** field, select the process order for the variable, which defines how it is processed: Choose *While Read* to direct the system to process the values of the variables while reading, *While Print* to direct the system to process the values of the variables while printing, and *Always* to direct the system to process the values of the variables while reading and printing.
6. In the **ResetExpr** field, define the reset expression for the variable, if it is required.
7. In the **ResetGroup** field, select the group where the variable value will be reset (item 4).

   In the `ResetGroup` property, you can specify the id of the group, in which the variable should be calculated locally. If you have set this property, for each instance of the specified group the variable has an independent value. At the end of each group, the variable is reset. If you have two or more nested groups, you can calculate variables individually for each group by setting the `ResetGroup` property.

   Use this property to calculate some values within a group. For instance, if you have the `Vendor` group inside the `Account` group and you want to calculate the account balance and each vendor balance within the account. For the `VendorBalance` variable, set `ResetGroup` to the `Vendor` group. For the `AccountBalance` variable set `ResetGroup` to the `Account` group. If the `ResetGroup` property is not set, the value will be accumulated from one vendor to another.
8. In the **ValueExpr** field, define the expression used to set the variable value, if it is required.

   To delete a variable from the list of existing variables, invoke the **ReportVariable Collection Editor** dialog, click this variable in the **Members** list, and click the **Remove** button.
9. Click **OK** (item 5) to save the changes and close the window.



References

- *Using Expressions*

Using the External Parameter Collection Editor

The External Parameter Collection Editor lets you define the parameters for a text box or subreport visual object.

For a text box, you can add navigation parameters by invoking the **ExternalParameter Collection Editor** dialog from the **NavigateParams** field on the **Properties** tab, and for a subreport, you can define the external parameters shared by the master report and the subreport from the **Parameters** field on the **Properties** tab. The existing parameters are listed in the **Members** list in the left area of the dialog.

To add a new parameter or change the properties of the existing one, perform the following steps (a text box is used as an example):

1. Select the text box and click the button for the **NavigateParams (Collection)** in the **Properties** tab (shown left of the red 1 in the screenshot below).
2. Click the **Add** button in the bottom left of the dialog, or click the existing parameter's name in the **Members** list (item 2 in the screenshot below).

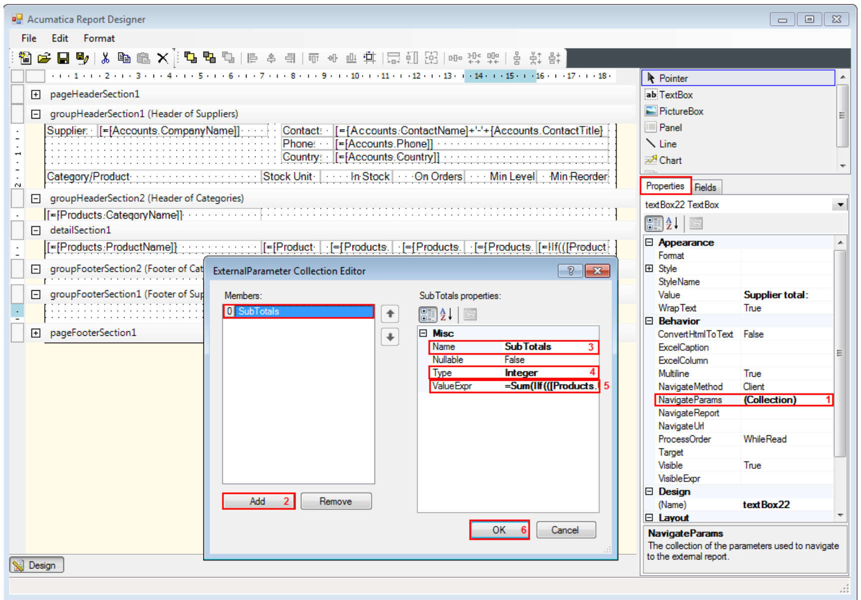


Figure: The ExternalParameter Collection Editor window

3. In the **Name** field in the parameter's properties table (located on the right side of the dialog), enter the parameter's name (item 3).
4. In the **Nullable** field, set the nullability property for the parameter: *True* or *False*. If the parameter's nullability property is set to *True*, the parameter can accept null values.
5. In the **Type** field, select the parameter's data type (item 4), which can be *Boolean*, *DateTime*, *Float* , *Integer*, or *String*.
6. In the **ValueExpr** field, define the expression to be used to calculate the parameter's value (item 5). Use the **Expression Editor** dialog to define the expression.
7. Click **OK** (item 6) to save the changes made to the external parameters, or click **Cancel** to discard the changes.

References

- *Using the Expression Editor*

Saving and Publishing the Reports

A custom report you design can be saved on your system or network drive. To make the report available for other Acumatica ERP users, you need to publish the report on the Acumatica ERP server.

Saving a Report

You can save custom reports locally or on the server. The decision about where to save the reports depends on various factors, including the stage of the report designing process, the Internet connection bandwidth, and the desired availability of the report to other users participating in the report development and review process.

- **Saving a Report Locally**: To save the designed report locally, use the *Save* or *Save As* command on the **File** menu, with a folder on a local system or network drive specified as the destination folder.
- **Saving a Report on the Server**: To save the designed report on the server, select the *Save on Server* command on the **File** menu, and provide the following information in the **Save Report on Server** dialog box:

1. **Specify Web Site URL**:The connection string to the server where the designed report will be stored
2. **Select report to load**: The locally stored custom report to be uploaded on the server
3. **Login**: The login to connect to the server
4. **Password**: The password to connect to the server

Publishing a Report

You must publish the designed custom report on the Acumatica ERP site to make it available to other Acumatica ERP application users. To publish a report on the site, use the *Site Map* (SM.20.05.20) form.

To publish a report, take the following steps:

1. Upload or copy the created report file to the appropriate folder on the Acumatica ERP website. By default, the `Reports/` folder, located in the root of the appropriate module on the Acumatica ERP website, is used.
2. From Acumatica ERP, navigate to the Site Map form: **System Management > Site Management > Site Map**.
3. Add a new node or expand the relevant module's hierarchical structure, and select **Reports**.
4. Add a new record to the list of expanded node items for the new report. Specify the following information:
   - **Title**: The title of the custom report.
   - **Icon**: The path to the icon for the custom report (optional).
   - **URL**: The URL of the custom report on the site. Use the following format for the URL specification:

     `~/Frames/ReportLauncher.aspx?ID=ReportName.rpx`
5. Click **Save** to save your changes. The added report will become visible with the site map.

For more information about the site management procedures, see the *System Management Module* section of this guide.

After the report is published, users who will generate the report must be granted access rights to this report.

References

- *Site Map* (SM.20.05.20)

Recommendations

This document describes some recommendations and best practices of report design for the Acumatica ERP application. These recommendations focus on the creation of visually consistent and easy-to-comprehend reports. You can also refer to an *example* of a simple report that illustrates the best practices described here.

Header Layout

A report can include two types of headers: The *report header* appears on the first page of the report, and the *page header* appears on the pages of the report. By default, the page header appears on all pages of the report, but you can configure it to appear on pages starting from the second one. You should always insert both the report header and the page header into your report. If either of them is absent, you can right-click the report area outside of any section and select **Report Header** or **Page Header**.

The report header and the page header should each consist of two sections. To split any section into two sections, right-click the section and select **Duplicate section**.

To make the page header appear on pages starting from the second one (rather than on all pages), you should set the **PrintOnFirstPage** property to *False* on all sections that represent the page header and footer.

The first section of the report header should have the following layout:

- On the left side of the report header, you should place the name of the report and the following mandatory fields below it, with each field represented by two text boxes:
    - *Company*
    - *Ledger* (if it is included in your report parameters)
    - *Branch* (if it is included in your report parameters)
- On the right side of the report header, you should place the following mandatory fields, with each field represented by two text boxes:
    - *User*
    - *Date*
    - *Page*
- If additional fields from the report parameters should be printed on the report header, put the fields in the middle part of the header in one column or two columns.

For information about how to set the values of the mandatory fields, see the *Parameter Values* section of this document.

The figure below shows an example of the layout of a report header.

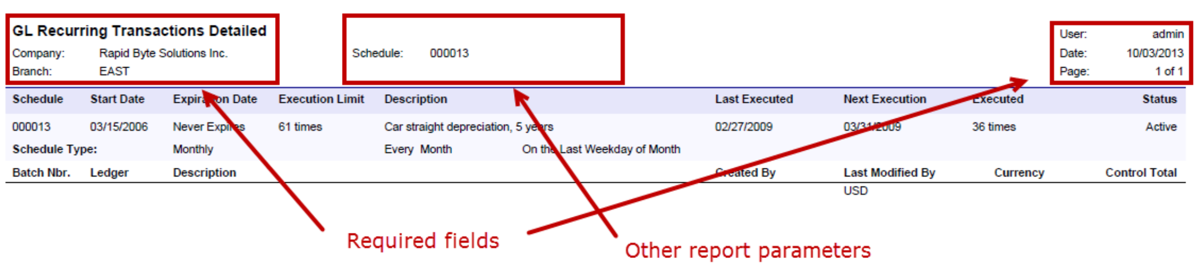


Figure: Example of a report header

The first section of the page header should have the following layout:

- On the left side of the report, you should put the name of the report.
- On the right side of the report, you should put the *Page* field.
- No report parameters are displayed on the page header.

The second section of both the report header and page header should contain text boxes with labels for columns.

The following screenshot shows the view of the report header in the Acumatica Report Designer.

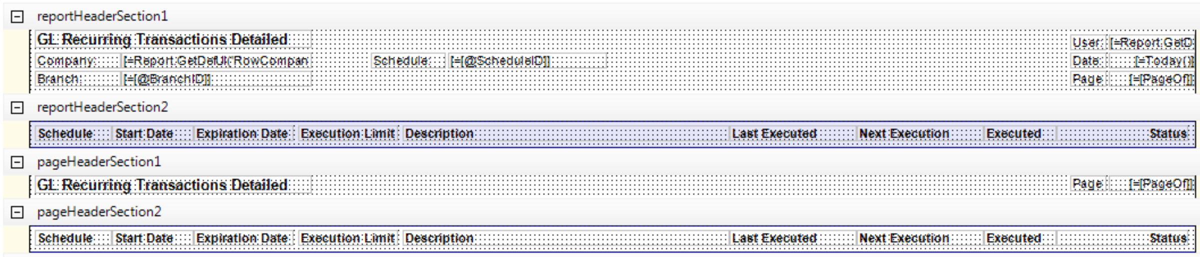


Figure: Example of the report header in the Report Designer

General Layout Properties

The table below shows the recommended properties for the layout of the whole report and all controls the report includes.

Description	Value
StylesTemplate property of the report	TemplateReport.rpx[1]
NavigationTree property of the report	False
LayoutUnit property of the report	Pixel
Width property of the report	1026px
Margin from the left border of the report	4px
Margin from the right border of the report	4px
Vertical margin between two text boxes	4px
Horizontal margin between the text box with the label and the text box with the value	0px
Height of the text box with the report name	16px
Height of the other text boxes	14px

[1]The template file should reside in the same directory as the report.

Recommended Predefined Styles

For any visual element of the report, you can set one of the predefined styles. You should assign specific predefined styles to the elements listed in the following table. To use the predefined styles, you should specify the template for the report by setting the **StylesTemplate** property to *TemplateReport.rpx*. This file is located in the same folder that contains the default reports provided with Acumatica ERP. To display report properties in the **Properties** view, click the little square in the upper left corner of the designer area.

Element	Style name
Text box with the report name	Report Name
Text boxes for both labels and values of report parameters in the header	Report Params
The report or page header section with column names	ColumnsHeaderSection
The group header sections with information on the grouping item	GroupHighlight

Element	**Style name**
The group header section with column names for the display of detail records	GroupL1Highlight
Text boxes for column names	Heading 1
Text boxes for total amounts of a group	Heading 1
Text boxes for displaying regular data	Normal

Abbreviations for Column Names

The following table shows the recommended abbreviations for column names.

Full column name	**Short column name**
Beginning Balance	Beg. Balance
Ending Balance	End. Balance
Financial Period	Fin. Period
Subaccount	Sub.
Reference Number	Ref. Nbr.
Batch Number	Batch Nbr.
Document	Doc.
Currency	Cur.
Original	Orig.
Transaction	Tran.

Currency Column Before an Amount Column

In any details view, any column representing an amount should be preceded with the currency column. If a column representing an amount immediately follows another such column and the two columns have the same currency (such as debit amount and credit amount in journal transactions), you should insert only one currency column—before the first of these two columns.

Parameter Names

When any of the following fields is used as a report parameter to specify a range of values, the name should start with *From* or *To*.

The name of a report parameter is set on the **Parameters** tab of **Schema Builder** in the **Prompt** field. If you don't specify the name in the **Prompt** field, the parameter won't be shown on the report webpage.

Field	**Display name of the parameter**	**Display name of the parameter**
Period	From Period	To Period
Date	From Date	To Date
Account	From Account	To Account
Subaccount	From Subaccount	To Subaccount

When the name of a field ends with *ID*, the name of the corresponding parameters should not include *ID*. The fields to which this rule is applied are listed in the table below.

Field	Display name of the parameter
Vendor ID	Vendor
Customer ID	Customer
Branch ID	Branch
Tax Agency ID	Tax Agency
Account ID	Account

Parameter Values

The table belows describes the recommended way to display the values of the mandatory fields displayed in the header.

First text box—Value	Second text box—Value
Company:	=Report.GetDefUI('RowCompanyBAccount.AcctName')
Ledger:	=[@LedgerID][1]
Branch:	=[@BranchID][1]
User:	=Report.GetDefUI('RowAccessInfo.DisplayName')
Date:	=Today()
Page:	=PageOf()

[1]Insert the actual name of the parameter that you specified in the **Schema Builder**.

Sample Report

This example illustrates best practices in report design for the Acumatica ERP application. To implement the sample report, you need to have the Acumatica Report Designer and an instance of the Acumatica ERP application installed.

The report will display data records of a scheduled batch with their details—journal transactions. By *scheduled batch*, we mean a batch that is processed according to the related schedule. The report will select batches by the **Scheduled** field, which equals *true* when a schedule is associated with the batch. By using the parameters of the report, you can filter batches by a ledger, branch, or batch number (to display details of a specific batch).

Building the Data Schema for the Report

1. In the **Schema Builder** window, load the schema of the website by specifying the URL of the application and valid credentials, and add the Batch table and GLTran (PX.Objects.GL.GLTran) table to the report (see the screenshot below).

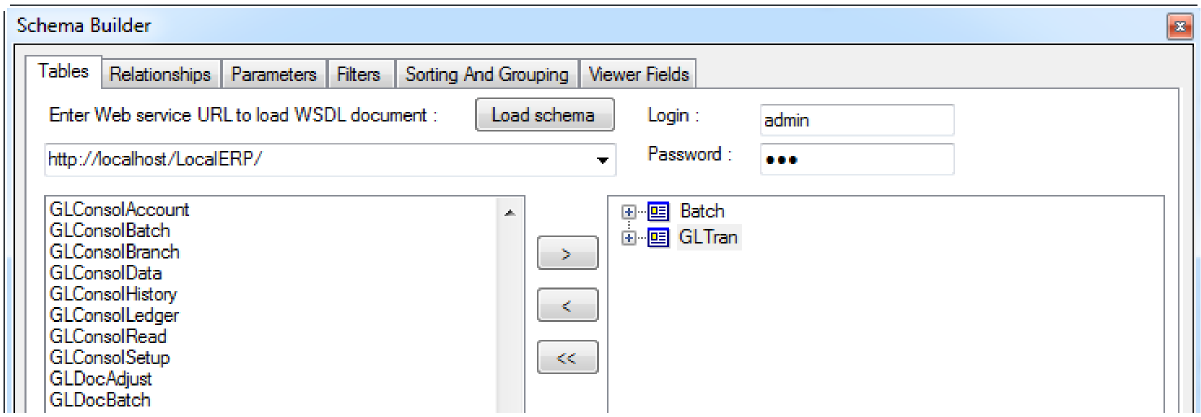


Figure: Loading schema and selecting tables for the report

2. Configure the relationship between two tables with the following propeties:

- **Parent Table**: *Batch*
- **Join Type**: *Left*
- **Child Table**: *GLTran*
- **Parent Field**: *BatchNbr*
- **Link Condition**: *Equal*
- **Child Field**: *BatchNbr*

3. On the **Parameters** tab, add three parameters (Branch, Ledger, and Batch) with the following properties.

Property	Branch parameter	Ledger parameter	Batch parameter
Name	BranchID	LedgerID	BatchID
Data Type	String	String	String
View Name	=[Batch.BranchID]	=[Batch.LedgerID]	=[Batch.BatchNbr]
Prompt	Batch	Ledger	Batch
Column Span	2	2	2
Allow Null	True	True	True
Visible	True	True	True

In the **View Name** property, you specify the data field from which the report should take the display options for the parameter (such as the type of the control for entering a value).

4. Specify filtering conditions to restrict the set of data (selecting only scheduled batches) and use the report parameters.

Braces	Data Field	Condition	Value1	Braces	Operator
	Batch.Scheduled	Equal	True		And
(	Batch.BranchID	Equal	@BranchID		Or
	@BranchID	IsNull		)	And
(	Batch.LedgerID	Equal	@LedgerID		Or

Braces	Data Field	Condition	Value1	Braces	Operator
	@LedgerID	IsNull		)	And
(	Batch.BatchNbr	Equal	@BatchID		Or
	@BatchID	IsNull		)	And

You can use the parameters of your report to build filtering conditions in any way you need. Typically, as the example above shows, you check whether some field value equals the parameter value or the parameter value is null (not specified).

Specifying General Settings for the Report

To specify general report settings, click the square button at the upper left corner of the designer and set the following properties for the report:

- **StylesTemplate**: *TemplateReport.rpx*
- **NavigationTree**: *False*
- **GridSize**: *4px*; *4px*
- **Excel Mode**: *Manual*
- **LayoutUnit**: *Pixel*
- **Width**: *1026px*

Preparing the Header

1. Add the report header and page header to the report, and split each header into two sections by using the **Duplicate Section** command.
2. Set the following properties for the sections that represent the report header and footer and the page header and footer.

Section	StyleName	PrintOnFirstPage	Height
reportHeaderSection1			56px
reportHeaderSection2	ColumnsHeaderSection		24px
pageHeaderSection1		False	20px
pageHeaderSection2	ColumnsHeaderSection	False	24px

3. Add and align the text boxes for the report name, mandatory parameters, and other report parameters as described in the *recommendations*. The table below gives an example of the settings for the text boxes.

Value	StyleName	Location	Size
Scheduled Batches	Report Name	4px; 0px	244px; 16px
Company:	Report Params	4px; 20px	76px; 14px
=Report.GetDefUI('RowCompanyBAccount. AcctName')	Report Params	80px; 20px	168px; 14px
Ledger:	Report Params	4px; 38px	76px; 14px
=[@LedgerID]	Report Params	80px; 38px	168px; 14px
Branch:	Report Params	4px; 56px	76px; 14px

Value	StyleName	Location	Size
=[@BranchID]	Report Params	80px; 56px	168px; 14px
Batch:	Report Params	340px; 20px	76px; 14px
=[@BatchID]	Report Params	76px;14px	168px; 14px
User:	Report Params	916px; 20px	32px; 14px
=Report.GetDefUI('RowAccessInfo.DisplayName')	Report Params	948px; 20px	76px; 14px
Date:	Report Params	916px; 38px	32px; 14px
=Today()	Report Params	948px; 38px	76px; 14px
Page:	Report Params	916px; 56px	32px; 14px
=PageOf	Report Params	948px; 56px	76px; 14px

You can copy a group of controls and paste them into the same section or another section. To select multiple controls, click them one by one while pressing the Shift key. You can also set a property for all selected controls at once.

For the label text boxes of the Ledger, Branch, and Batch parameters, set the **VisibleExpr** property to the following values:

- *=([@LedgerID]<>Null)*
- *=([@BranchID]<>Null)*
- *=([@BatchID]<>Null)*

As a result, these text boxes will be displayed only when a user specifies parameter values for the report and runs it.

4. Add text boxes with the properties shown in the following table to the section named *reportHeaderSection2*. The text boxes will represent column headers for batch records.

Value	StyleName	Style—Text Align	Location	Size
Batch Nbr.	Heading 1		4px; 4px	68px; 14px
Ledger	Heading 1		72px; 4px	72px; 14px
Description	Heading 1		144px; 4px	272px; 14px
Created By	Heading 1		616px; 4px	112px; 14px
Last Modified By	Heading 1		728px; 4px	112px; 14px
Currency	Heading 1	Right	840px; 4px	64px; 14px
Control Total	Heading 1	Right	904px; 4px	116px; 14px

You can use a predefined style and specify additional display properties in the **Style** group of properties.

The same column headers should be placed in *pageHeaderSection2*. To copy column headers from the report header, select all text boxes in *reportHeaderSection2*, right-click them, select **Copy**, right-click *pageHeaderSection2*, and click **Paste**.

Preparing the Main Part of the Report

1. Add one group by right-clicking the report outside of any section and selecting **Add New Group**. Duplicate the group header and the group footer. Open the **Schema Builder**, open the **Sorting and Grouping** tab, select *group1*, and specify the following properties for the grouping:
   - **Data Field**: *Batch.BatchNbr*
   - **Sort Direction**: *Ascending*

   You can duplicate group headers and footers any number of times. You can use additional group headers and footers to add spacing between rows. The numbers of headers and footers doesn't have to be the same. However, you add a new group only to add a new level of grouping data.

2. Set the following properties for the group headers, footers, and detail section.

Section	StyleName	Height
groupHeaderSection1	GroupHighlight	16px
groupHeaderSection2	GroupL1Highlight	20px
groupFooterSection1		20px
groupFooterSection2		16px
detailSection1		16px

3. Copy the text boxes with column names from the report or page header to *groupHeaderSection1*, shift them to the top of the section, and set the **StyleName** property to *Normal*. Set **Value** to the corresponding Batch data fields:
   - *=[Batch.BatchNbr]*
   - *=[Batch.LedgerID]*
   - *=[Batch.Description]*
   - *=[Batch.CreatedByID]*
   - *=[Batch.LastModifiedByID]*
   - *=[Batch.CuryID]*
   - *=[Batch.CuryControlTotal]*

4. Add text boxes with the following properties to *groupHeaderSection2* to represent the column headers for journal transaction records.

Value	StyleName	Style—Text Align	Location	Size
Branch	Heading 1		4px; 4px	68px; 14px
Account	Heading 1		72px; 4px	72px; 14px
Sub.	Heading 1		144px; 4px	136px; 14px
Ref. Nbr.	Heading 1		280px; 4px	116px; 14px
Description	Heading 1		396px; 4px	332px; 14px
Currency	Heading 1	Left	728px; 4px	60px; 14px
Debit	Heading 1	Right	788px; 4px	116px; 14px
Credit	Heading 1	Right	904px; 4px	116px; 14px

5. Copy the text boxes with column names from *groupHeaderSection2* to *detailSection1*, shift them to the top of the section, and set the **StyleName** property to *Normal* for all of them. Set **Value** to the corresponding GLTran data fields:

- *=[GLTran.BranchID]*
- *=[GLTran.AccountID]*
- *=[GLTran.SubID]*
- *=[GLTran.RefNbr]*
- *=[GLTran.TranDesc]*
- *=[GLTran.CuryID]*
- *=[GLTran.CuryDebitAmt]*
- *=[GLTran.CuryCreditAmt]*

6. In the first group footer, add four text boxes to *groupFooterSection1* and set the following properties for them. These text boxes will be used to display total amounts for a batch right under the **Debit** and **Credit** columns in the first group footer.

Value	StyleName	Style—Text Align	Location	Size
Batch Total:	Heading 1		616px; 4px	112px; 14px
=[Batch.CuryID]	Heading 1	Left	728px; 4px	60px; 14px
=[Batch.CuryDebitTotal]	Heading 1	Right	788px; 4px	116px; 14px
=[Batch.CuryCreditTotal]	Heading 1	Right	904px; 4px	116px; 14px

You can use aggregation functions to perform calculations over grouped items. For example, you could replace *=[Batch.CuryDebitTotal]* with *=Sum([GLTran.CuryDebitAmt])*, which would calculate a sum over all child journal transactions for each parent batch. However, here we use the data field of the parent, because it already contains the sum.

To draw a line above the total amounts, add a line to the *groupFooterSection1* and properly align it. You can set the following properties for the line:

- **Location**: *612px; 2px*
- **Size**: *414px; 2px*

The X coordinate of the location added to the width should be less or equal to the overall width of the report for the line to not extend beyond the report.

Publishing the Report

To make the report accessible through the website, you should add it to the Site Map (**System** > **Customization** > **Manage** > **Site Map**) of your Acumatica ERP application. For example, you can add this report to the **Finance** > **General Ledger** > **Reports** > **Audit** section of the site. You can add a new node with the following properties:

- **ScreenID**: *GL.69.00.11*
- **Title**: *Scheduled Batches*
- **Url**: *~/Frames/ReportLauncher.aspx?ID=<YouReportName>.rpx*

Website Management

In this chapter, you will get acquainted with the standard Site Map of the Acumatica ERP application, as well as learn how to configure or modify the Site Map for your own purposes. Topics of this chapter also contain descriptions of how to register webpages, how to grant access rights to the registered webpages, as well as how to manage the Help Wiki.

Content

This chapter covers the following topics:

Configuring the Site Map

You use the site map of the Acumatica Framework application for adjusting the multilevel menu structure and for registering webpages. See *Site Map* (SM.20.05.20) for details.

In the first section of this topic, the typical multilevel structure of the Acumatica Framework application site map is described. The second section gives the common rules of site map configuration.

The Typical Structure of the Acumatica Framework Application Site Map

If you start an Acumatica Framework application instance and then navigate to **System** > **Customization** > **Manage** > **Site Map**, you will see the site map tree. This tree displays the menu and sub-menu structure of the typical Acumatica Framework application. As the screenshot below illustrates, this structure consists of different levels, beginning with the topmost level (the common solution level) and two upper levels that represent the main menu and sub-menu items, and ending with the lowermost level, which includes various webpages (forms).

If you expand a menu item by clicking the node icon left of it, you will see the second-level node names (sub-menu) in the tree, which mostly include the names of application modules (see again the screenshot below).

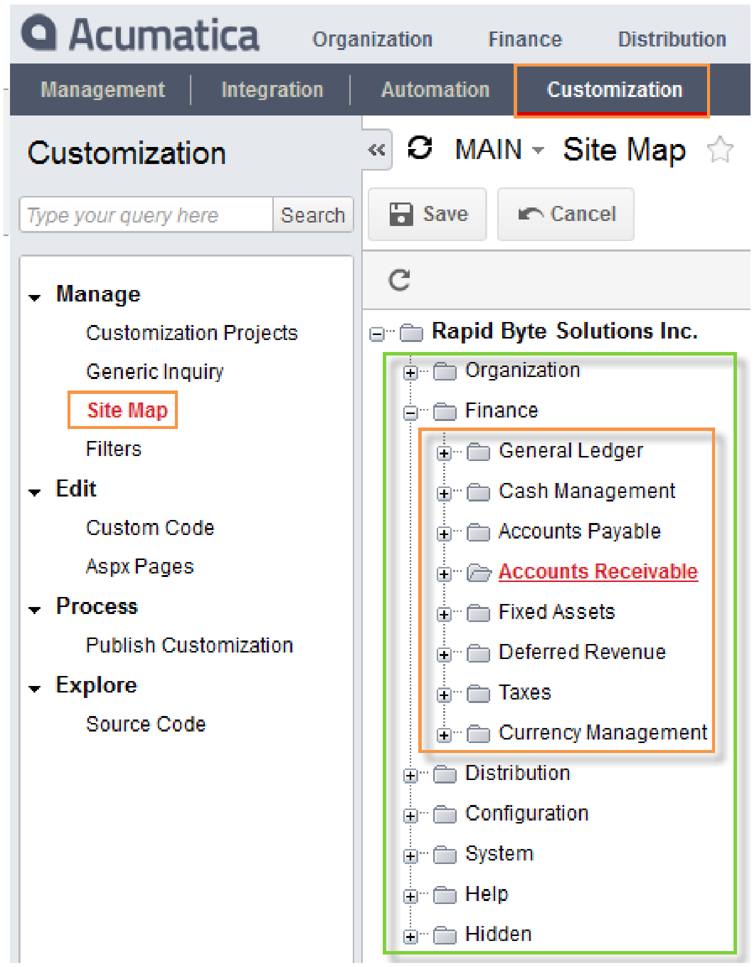


Figure: Opening the site map

If you select a sub-menu item that represents an application module, you will see the third-level node names in the table right of the tree (see the screenshot below), which holds the settings of the nodes (see the screenshot below). Most modules include up to four nodes that provide access to the webpages on the lowermost level:

- The **Work Area** node includes data entry, maintenance, and inquiry webpages.
- The **Processes** node includes processing webpages.
- The **Reports** node includes report webpages.
- The **Configuration** node includes setup webpages, analytical reports, and some maintenance webpages.

Each node represents a tab that is displayed below the Search box at the top of the navigation pane when a user is viewing the module.

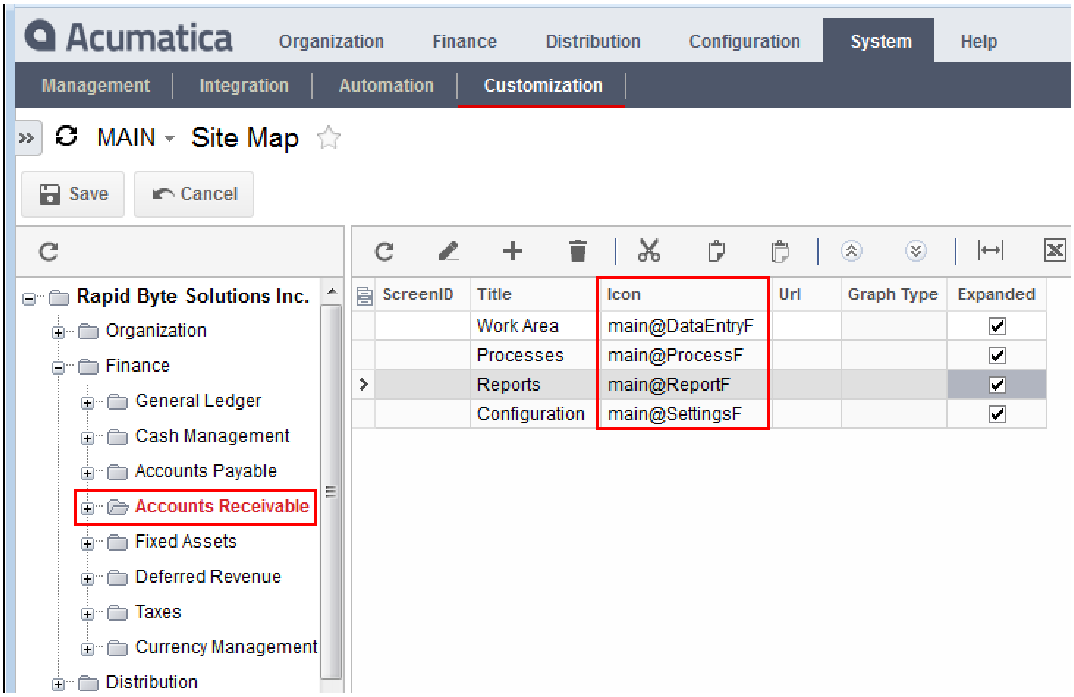


Figure: An example of settings for the third-level nodes

The fourth-level nodes can be used for additional grouping of webpages in the navigation pane. There are no system restrictions on how to name these groups and how many groups may be added.

After selecting a fourth-level node item, you can see the corresponding webpages within that group and their settings (see the screenshot below).

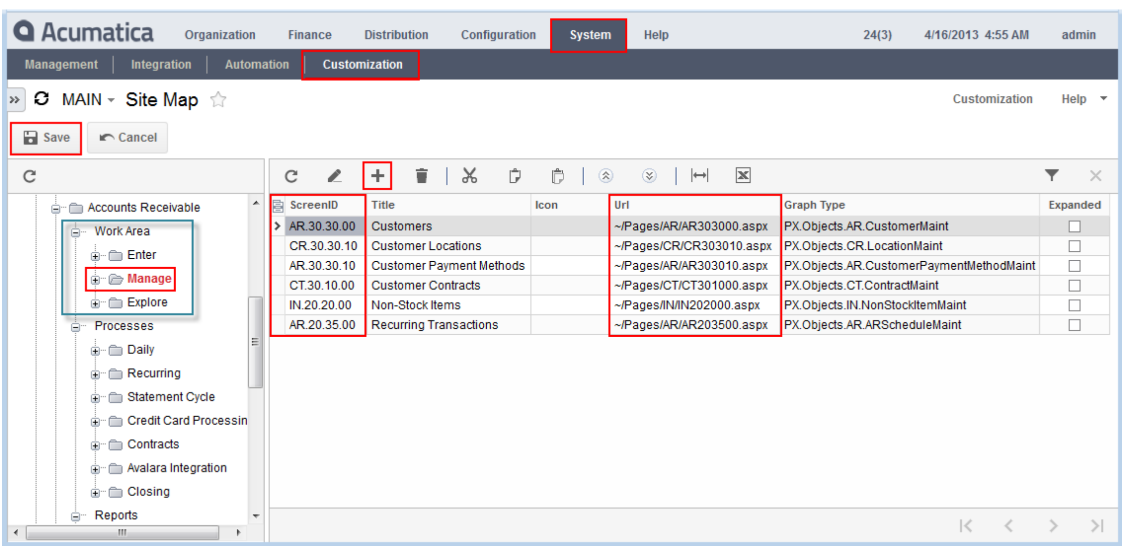


Figure: An example of settings for the fifth-level items (webpages)

Common Rules of Configuring the site map

As you can see in the screenshot below, navigation in the standard Acumatica Framework application instance represents the sequence of selected items on different levels, from the main menu down to the item on the lowermost level of the site map, to open the required webpage (form). For instance, to open at run time the *Update Base Prices* webpage, you should click **Distribution** (the first level and top line of the main menu) and then **Inventory** (the second level and sub-menu, or bottom line of the main menu). Then click the **Processes** (the third level, with the icon name) tab, and beneath the **Recurring** (the fourth level, with a sub-section of the navigation pane) group, click **Update Base Prices** (the fifth level, which is the required webpage).

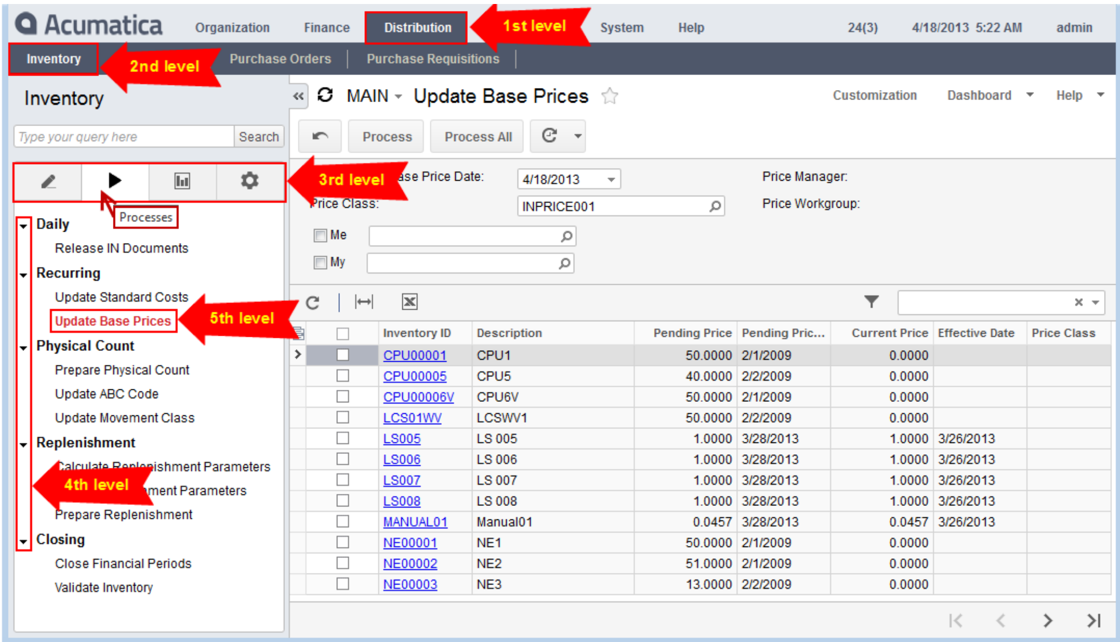


Figure: Navigation at run time through the different levels of the site map

You can construct a site map structure for your own application, taking into account the following rules of site map design:

- All of the site map levels are mandatory except for the third and fourth level. You should include at least one needed node for each required level of the site map. In such a case, you can register the webpage after selecting the appropriate item of the second- (or the third-) level node.
- The top-level node represents the common solution; you can add first-level (main menu) items and adjust their properties after you select this level.
- The first-level node defines different sub-menu items; for each menu item, you must add and adjust at least one sub-menu item.
- By selecting each node on levels from the second to fourth, you can adjust the appropriate item properties of the level beneath the node, including adding and adjusting new items.

  Notice that if you create a node with only one item as the second or third sub-node, this item will be invisible unless you add a second item on the same level.

- To register a newly developed page as a webpage, you should select the respective item of the fourth level, if it exists; otherwise, you have to first add and adjust properties of this level (and each level above it). The process of adding new items is described in *Registering the Page as a New Webpage*.

After you register a newly developed page as a webpage, you need to assign access rights to it, as described in *Granting Access Rights to a Registered Webpage*.

Registering the Page as a New Webpage

To give the end user access to a page you have developed, debugged, and tested, you must register this page as a webpage on the *Site Map* (SM.20.05.20) form and then grant appropriate access rights to each webpage by using the *Access Rights by Role* (SM.20.10.25) form. The guidelines in this topic will help you register the page. To learn how to grant access rights, see *Granting Access Rights to a Registered Webpage*.

If your site map structure is not yet ready, you should first create and adjust nodes with appropriate items for the upper levels of the site map upper levels. See *Configuring the Site Map* for details.

When the site map is ready, all the nodes are configured and most items are properly registered.

Adding Items to the Site Map and Adjusting Their Properties

This section describes the creation of an additional branch of the site map. (To illustrate the case when you do not need the fourth level of the site map, which you can use to divide the navigation pane into sections, this branch will have four levels instead of the maximum of five. You can decrease the number of levels if you have only a few webpages to be registered.) To resolve this task, you should perform the following instructions:

1. Start your project application.
2. Navigate to **System** > **Customization** > **Manage** > **Site Map**, and then select the top-level folder (*Acumatica Company*).
3. Above the table on the right, click **Add Row** to add a node for the *RB* folder of the main menu. Specify the following settings (see also the screenshot below).
    - **Screen ID**: `RB.00.00.00`
    - **Title**: *RB*
    - **Icon**: None
    - **URL**: *~/Frames/Default.aspx*
    - **GraphType**: Empty
    - **Expanded** (check box): Cleared
4. By clicking the **Move Row Up** button several times, move the item to the needed position within the first-level menu item, and then click **Save**.

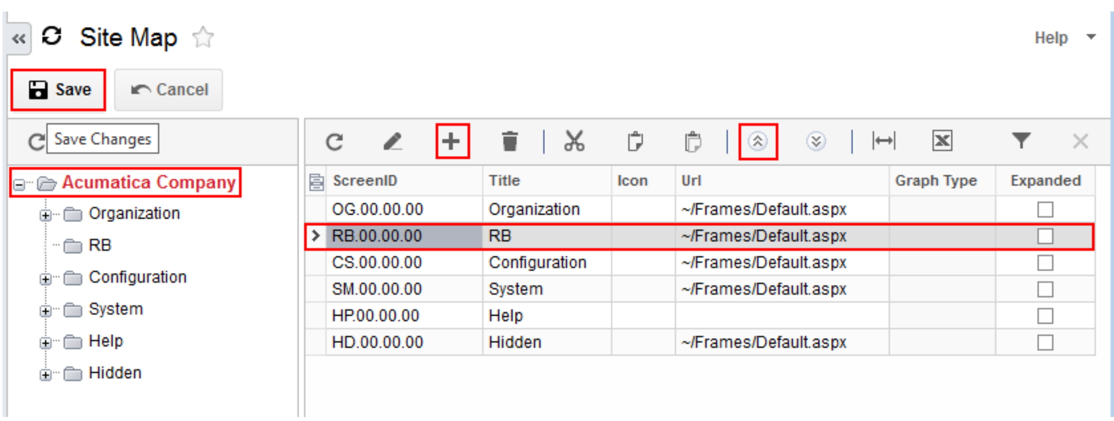


Figure: Adding and adjusting properties of the first-level menu item

5. Select the *RB* folder. In the table on the right, click **Add Row** to add a sub-menu item for the *RB* menu item you added in Instruction 3. Specify the following settings, and then save your changes (see also the screenshot below):
    - **Screen ID**: `RB.00.00.00`
    - **Title**: *RapidByte*

- **Icon**: Empty
- **URL**: *~/Frames/Default.aspx*
- **GraphType**: Empty
- **Expanded** (check box): Cleared

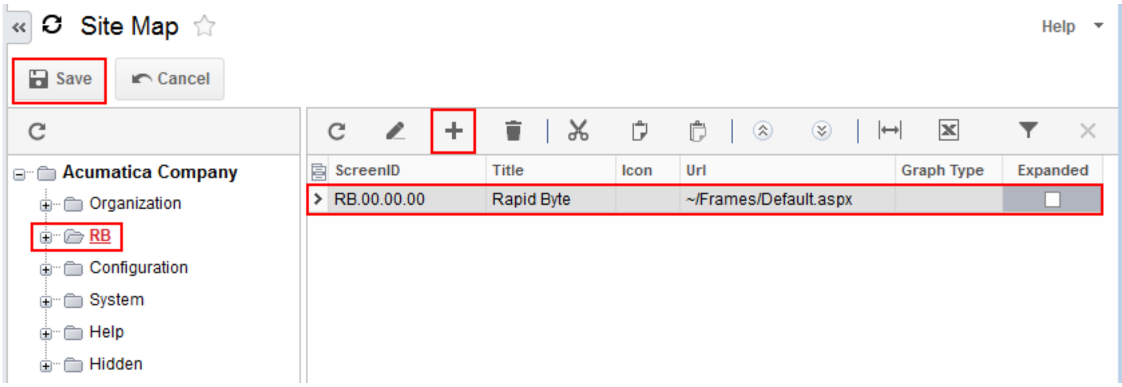


Figure: Adding and adjusting properties of the second-level item (sub-menu item)

6. Select the *RapidByte* folder, and add the third-level nodes to group the webpage types you will use. Specify the following settings, keeping the sub-nodes in the order shown in the table below, and then save your changes (see the screenshot below):

Screen ID	Title	Icon	URL	Expanded
Empty	`Work Area`	*main@DataEntryF*	Empty	Selected
Empty	`Processes`	*main@ProcessF*	Empty	Selected
Empty	`Reports`	*main@ReportF*	Empty	Selected
Empty	`Configuration`	*main@SettingsF*	Empty	Selected

In this example of adding a site map branch, you do not need to specify a screen ID for the third-level nodes. By selecting the **Expanded** check box, you provide automatic expansion of any third-level node with its webpages during site startup. The **GraphType** column also should be empty.

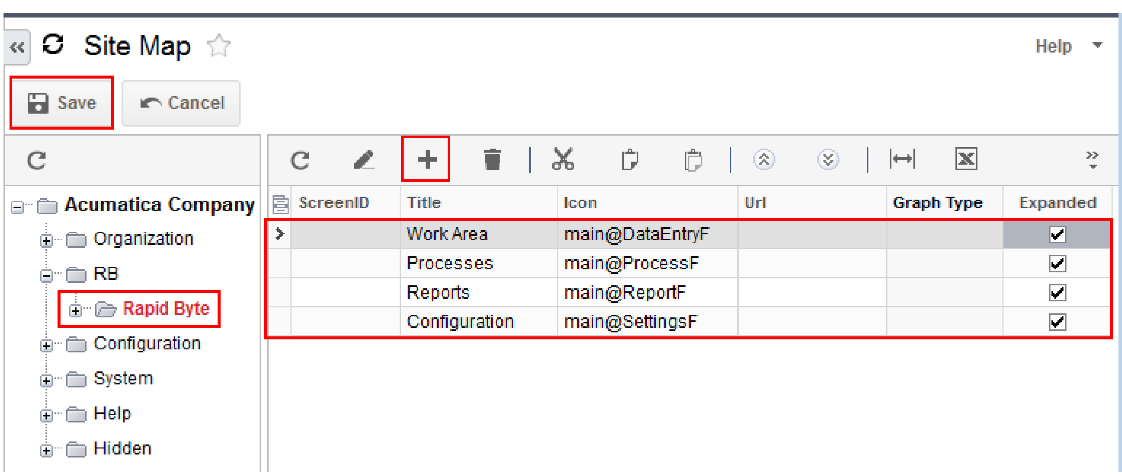


Figure: Adding and adjusting properties of the third-level node items (for webpage grouping)

Registering a New Page as a Webpage

Select the **Work Area** item, and then click **Add Row** to register the new developed pages as webpages (see the screenshot below). Here is the example of registering the *Employees* page. Make the appropriate specifications, and save your changes:

- **Screen ID**: `RB.20.20.00`
- **Title**: *Employees*
- **Icon**: Empty
- **GraphType**: *RB.RapidByte.EmployeeMaint* (added automatically)
- **URL**: *~/Pages/RapidByte/RB202000.aspx*
- **Expanded** (check box): Cleared

Notice that the system automatically defines the **Graph Type** setting for webpages.

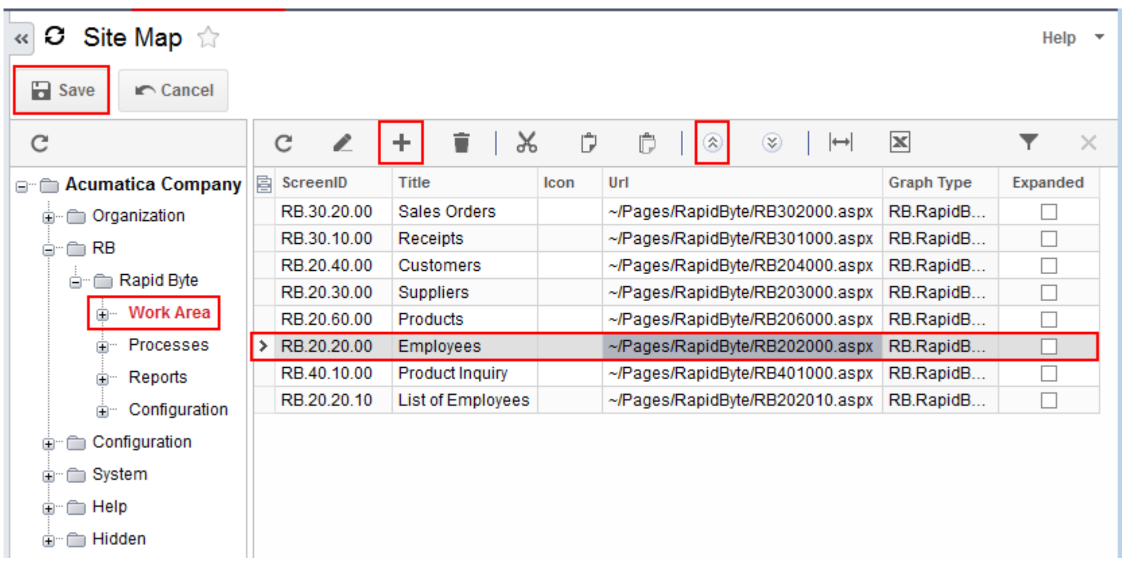


Figure: Registering the page as a webpage

You should register all the new developed pages as webpages similarly.

As was mentioned earlier, once you register webpages, you will grant access rights, as described in *Granting Access Rights to a Registered Webpage*.

Granting Access Rights to a Registered Webpage

After you register newly developed pages as webpages, you should grant appropriate access rights to them on the *Access Rights by Role* (SM.20.10.25) form, as this topic describes.

To grant access rights to the new registered webpage, proceed as follows:

- Navigate to **Configuration** > **User Security** > **Manage** > **Access Rights By Role**.
- In the **Role Name** box, select **Administrator**.
- In the System Tree pane of the form (lower left), click the node of the **RB** subfolder.
- On the Access Rights pane (lower right table), for the *Employees* page, select the *Delete* access rights, as shown in the screenshot below.
- Save your changes.

The *Delete* access rights encompass the *View*, *Edit*, and *Insert* rights. For more information, see *Levels of Access Rights*. If you need to cancel previous access rights to the webpage for the specified role, you should select the *Revoked* rights.

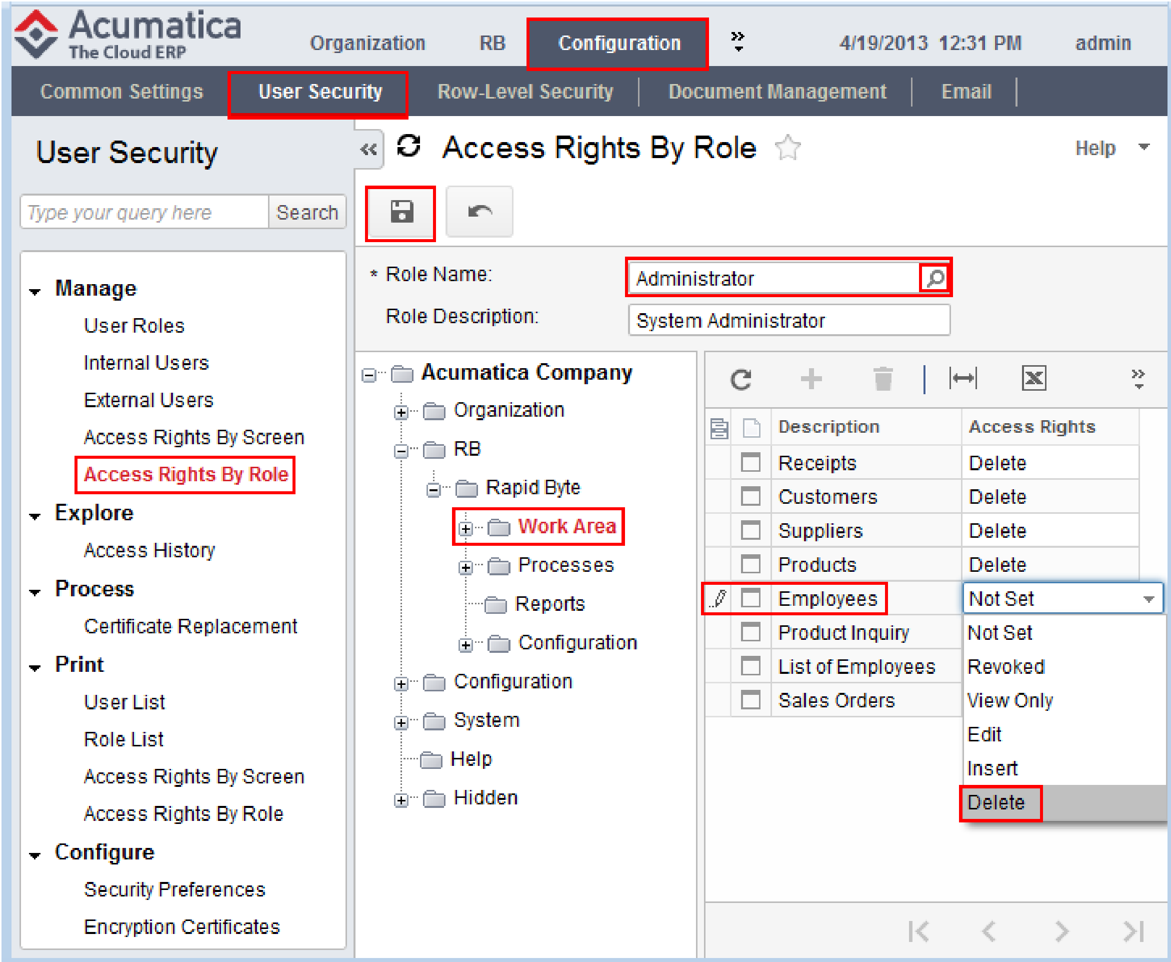


Figure: Granting access rights to the Employees webpage

Generally, the access rights granted to the webpage user interface (UI) elements and actions for a role are inherited from the access rights the role has to the webpage. Therefore, you should first give the role a permissive level of access to the system object that supports the webpage functionality. Then you can set access rights to the UI elements, as shown in the screenshot below (which shows an example with another solution and another webpage).

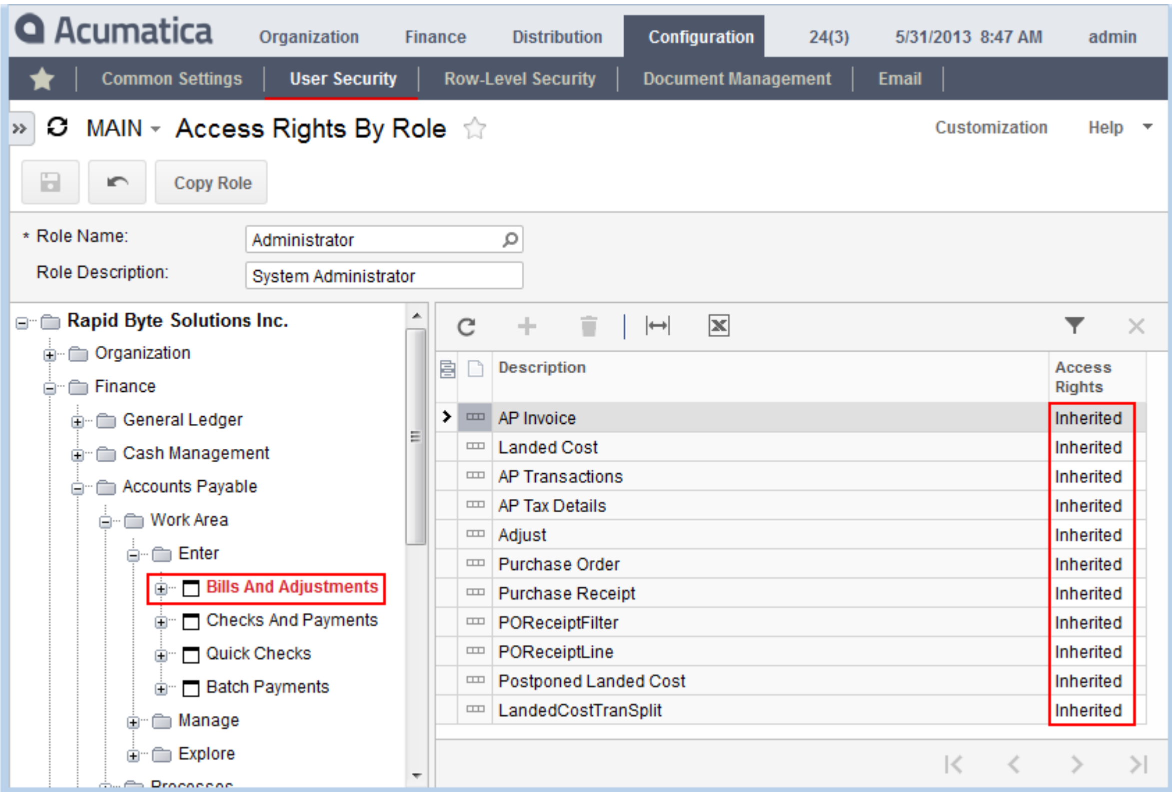


Figure: Default access rights to the UI elements of the Bills and Adjustments webpage

The *Inherited* rights indicate that access rights to the element are inherited from the access rights the role has to the webpage.

Managing the Help Wiki

Acumatica ERP includes a built-in wiki-based content management system that consists of topics (or articles) organized within second-level menu items (submenus) and folders. By using this system, you can create Help for any application you have developed with Acumatica Framework, in addition to the Help already provided by Acumatica ERP.

The first section of this topic analyzes the main features of the Help wiki, while the second section considers the configuration of the standard Help wiki. The third section describes how to create a new topic within the Help wiki, and the fourth section illustrates how to create a new wiki and a topic within it.

Exploring the Structure and Usage of the Help Wiki

The Acumatica Framework deliverable database contains a few wiki-based submenus with the standard Help content. If you start an Acumatica Framework application and click the **Help** menu item on the right side of the form title bar, you can see the available topics of the wiki-based Help for programmers and IT specialists within the following submenu items:

- Getting Started
- Installation
- Customization
- Acumatica Framework

Acumatica ERP also has two Help wiki submenus that hold topics for end users:

- User Guide
- Implementation

To describe the webpages you develop and to cover other topics, you can easily add new topics to any folder (or as a root item) of each Help wiki submenu item, and you can create your own folders and subfolders and then add new topics. You can also create your own wiki menu and submenu items, but new reference topics for webpages will still be added to the main Help wiki submenu items. You can add only separate topics to the new wiki submenu items.

When you first install Acumatica ERP or Acumatica Framework, each topic in the Help wiki submenu items has a single record in its history (with the full text copy of last topic version stored in the relevant database table). You can give users appropriate access rights to edit any topic, including those originally drafted by Acumatica Inc. Each time the user saves changes to any topic, the system adds (along with the full text version of the topic) a new record to the history table that includes such information as when the version was created and whether it was published (to make its text visible to users who open the topic).

When you upgrade the Acumatica Framework or Acumatica ERP application instance, topics existing in Help sections are updated if they were changed by Acumatica Inc., and new topics are added. If your site has modified at least one version of a wiki topic and published the version, this version cannot be replaced during the upgrade; however, you can access the updated text by restoring the appropriate history record as a current published topic version without needing to remove your site's version. Thus, you can refer the reader to the version updated by Acumatica Inc. and to your site's version when it is necessary.

Exploring the Configuration of the Standard Help Wiki

Each wiki submenu holds the Acumatica Inc.-supplied Help topics, grouped by a common theme, and can hold any Help topics you develop for webpages.

Proceed as follows to explore the setup of a standard Help wiki submenu:

1. Start the application, and navigate to the *Wiki Setup* (SM.20.20.05) form: **Configuration** > **Document Management** > **Manage** > **Wiki**.
2. In the **ID** box, select **HelpRoot_Studio**. (The *HelpRoot_* prefix is reserved for the Help wiki submenus that are provided by Acumatica Inc.; this wiki section holds most of the topics devoted to the development of an application by using Acumatica Framework.) Once you select this ID, the system retrieves the settings of the other fields. Notice the following fields:
    - **Name**: This field contains the name of the wiki item that you see as the second-level menu (or submenu) item.
    - **Style**: Open the lookup window of this field to see the style options for the existing Help wiki (as shown in the screenshot below).
    - **Site Map Location**: Open the lookup window with the Site Map tree, and then click the **Help** node to see the location of the Help wiki (its position) among the other menu items.
    - **Site Map Title**: This field holds the name of the selected Help wiki submenu item (the same as the name of the wiki ID).
    - **Article Type**: This field indicates the type of article (topic) that makes up the content of the wiki. (**Article** is the appropriate setting for a wiki that holds information that will be accessed by users over time.)
3. In the table on the **Access Rights** tab, notice the role names and their access rights, which you (as an administrator) can give to the roles assigned to users.

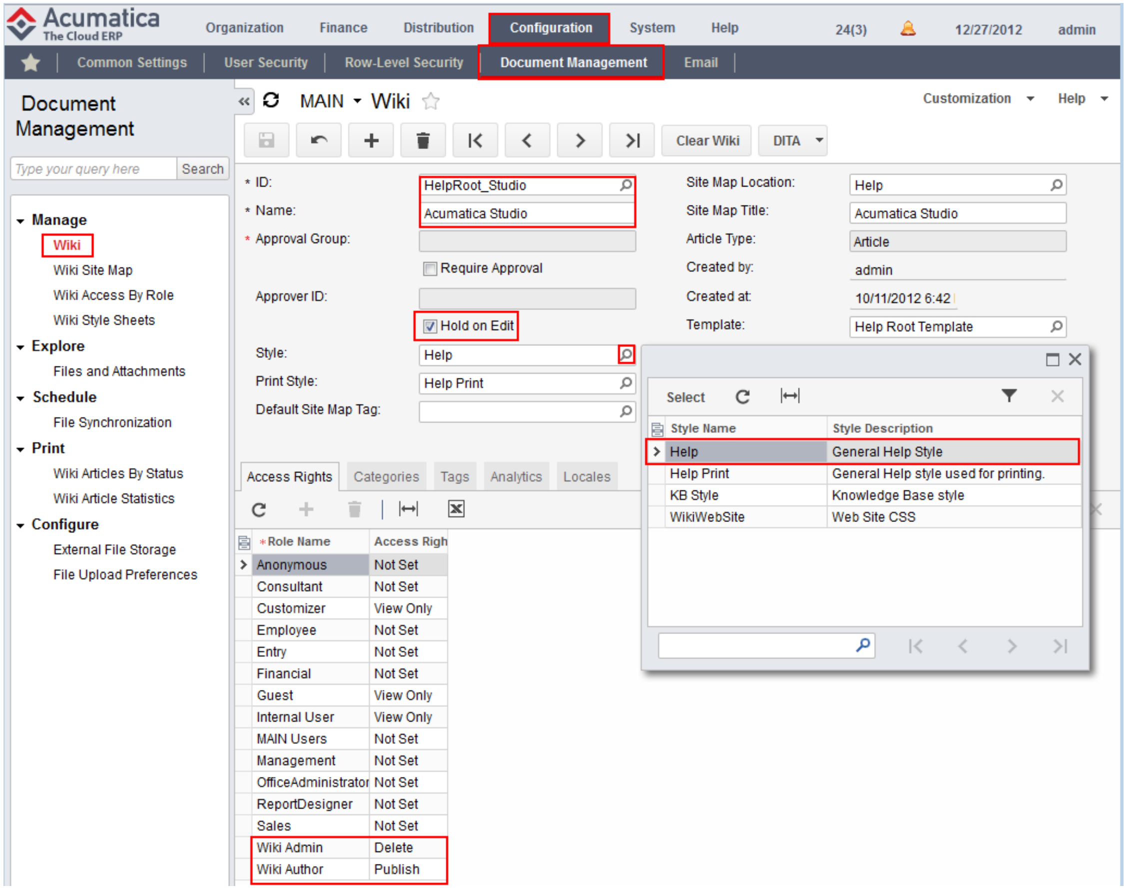


Figure: Verifying the configuration of the Help wiki

Adding a Help Topic for a Webpage

In this step, you will learn how to add a reference topic to the Help wiki that documents a new developed webpage.

All the forms (webpages) delivered with Acumatica Framework and Acumatica ERP already have detailed descriptions. If your application instance has no newly developed webpage, you should read rather than perform the instructions below.

To add a reference topic, you would perform the following tasks:

1. Open the webpage that you want to describe.
2. Click **Help** in the top right corner of the webpage, on the Main toolbar. You will see the following error message: *The article does not exist or you don't have enough rights to see it.*
3. On the Wiki toolbar, click **Edit**, and a new topic will be created. Notice that the system generates the **Article ID** based on the structure of the page number (**Screen ID**). For instance, the SM_20_05_20 article ID corresponds to the SM.20.05.20 page number (**Screen ID** on the user interface), which means SM (System Management) module, maintenance webpage (20), serial number (05), and subnumber (20).
4. In the **Article Name** field, specify the name of the webpage that is displayed in the navigation pane (that is, the same title you specified when you registered the webpage).
5. Add Help content for the page. Clear the **Hold** check box if you are ready to publish the topic.
6. Click **Save**. The new topic is now created.

You can repeat these steps for other webpages and reports that you have created.

Creating a New Wiki With a Topic

If you want to create a new wiki—for instance, to hold a group of not-yet-developed topics that describe your application add-on—complete the guidelines of this section. Again, keep in mind that you won't be able to add to the new wiki reference topics describing any webpage you develop. However, topics in a new wiki can contain references to webpage-related topics that are stored in the built-in Help wiki's submenu items; similarly, webpage-related topics in the Help wiki can have references to topics in the new wiki.

Before you proceed, you should know that you can register the new wiki in the application site map either as a separate menu item (that is, an additional help item) or as a submenu item of the **Help** menu. For the first case, topics of your own wiki menu are accessible through the main menu, while for the second case, you first have to select the **Help** submenu item. In both cases, the additional help will have aforementioned restriction concerning webpage descriptions. You take similar actions for both cases; we consider the first case. Proceed as follows:

1. Navigate to the *Wiki Setup* (SM.20.20.05) form: **Configuration** > **Document Management** > **Manage** > **Wiki**.
2. In the **Wiki** area, specify settings for the new wiki: On the toolbar, click **Add New Record** (if another wiki ID record is still active). Then type `AdHelp` as the **ID** and `Additional Help` as the **Name**, and select **Help** as the **Style**, **Help Print** as the **Print Style**, and **Article** as the **Article Type**. Open the **Site Map Location** window with the site map tree, and select the Company root node (to create a new local wiki group of topics you should select the **Help** submenu item); the **Site Map Title** value, added automatically, is the same as the **Name** value. Clear the **Require Approval** check box, and select the **Hold on Edit** check box. (See the screenshot below.)

   If you clear the **Hold on Edit** check box, the **Hold** check box will be cleared by default when an author edits a topic in the wiki, causing the topic to be published when it is saved. All readers can see the text of a published topic. With the **Hold on Edit** check box cleared, the author can still select the **Hold** check box before saving a topic, so that it is on hold. In this case, changes to the topic will be seen only by users with the *Publish* or *Delete* access rights (see the next instruction).

3. On the **Access Rights** tab table, grant the *Publish* access rights to **Wiki Author**, and grant the *Delete* access rights to **Administrator** and **Wiki Admin**, as shown in the screenshot below. Other role names have the *Not Set* access rights by default; you can change some of these rights.

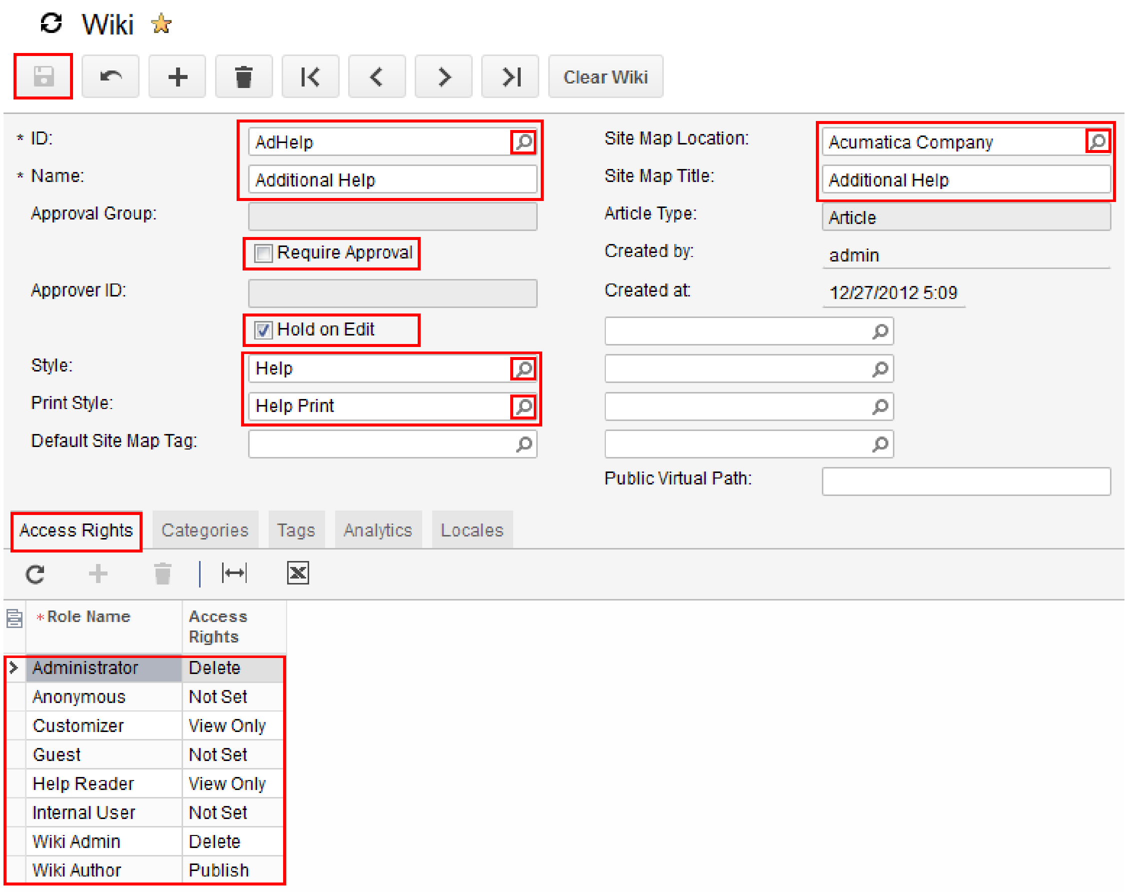


Figure: Adding a new wiki

4. Save your changes. The new wiki appears on the main menu as the rightmost menu item.
5. On the main menu, click **Additional Help**. The **Additional Help** main folder opens, with the **Create a new article** and **Deleted Items** links.
6. Click the **Create a new article** link to open the Wiki Editor for the new topic.
7. In the upper area of the **Content** tab, type `Intro` in the **Article ID** field and `Introduction` in the **Name** field. Keep **Additional Help** selected as the **Parent Folder**. See the screenshot below.

   If you want to create subtopics under the created folder or the higher-level topic, select the **Folder** check box. You can have any number of sublevels and topics at each level of a wiki. However, a folder cannot be placed directly under a topic or between topics.

8. Type any text into the editing area.
9. Be sure the **Hold** check box is selected. If you want the text of the topic to be visible to users now, however, clear the check box.
10. Click **Save** to save the new topic, **Introduction**, with the added text. The topic appears in the navigation pane. Any user with *View Only* access rights or higher can open the application and read this topic.

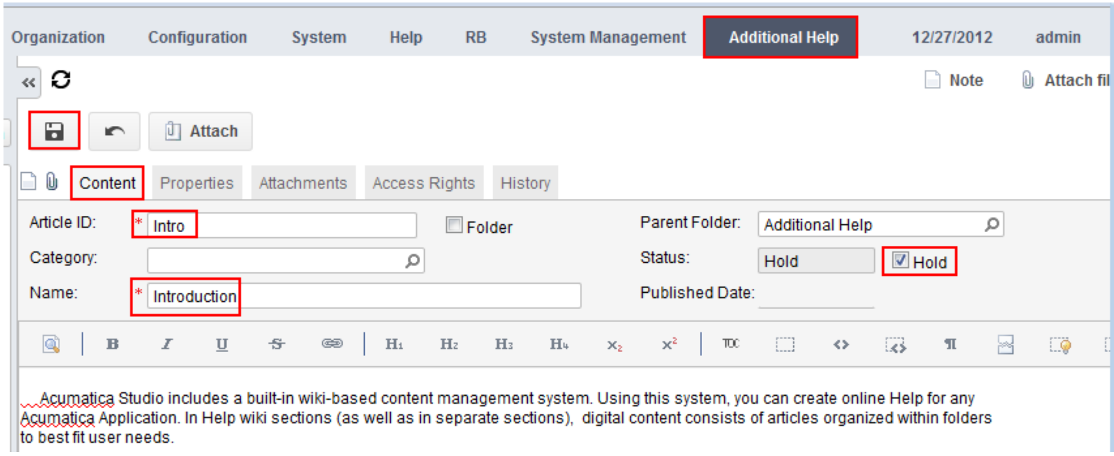


Figure: Adding a new topic

For more details, see *Setting Up a New Wiki*.

Web Services API Developer Guide

The Acumatica ERP Web Services Application Programming Interface (API) provides a fast, reliable, and convenient way of exposing business functionality and data managed by an Acumatica ERP application for integration with any external business and operation support system. The Acumatica ERP API is based on web service standards, such as SOAP and WSDL, and can be accessed with almost any current programming environment or integration tool. By using the development environment you are familiar with, you can easily create a client application that accesses the Acumatica Framework application through standard web services protocols to do any of the following:

- Authorize the programmer with the server running the Acumatica ERP application
- Get query and access information from the Acumatica ERP application
- Import information into the Acumatica ERP application
- Create, update, and delete objects in the Acumatica ERP application
- Execute some long-running processes and perform administrative tasks

Every operation that uses the Acumatica ERP API is executed through the same business logic layer as the user interface.

Web Services API Overview

Acumatica Inc. introduces a simple, streamlined way of interacting with its web services. The system automatically generates a WSDL file describing the operations (services) and list of parameters and objects; you can access this file through the *Web Services* (SM.20.70.40) form.

You can implement advanced integration scenarios involving operations on one or more forms by using the new web services configuration form to generate custom WSDL files.

All the functionality of the application is available through the Web Services API; however, the functionality and information that will be exposed and available to the web services client depends on the access rights granted to the user signed in as a client to the Acumatica Inc. instance.

Web Services Calls

To execute the API call, you need to prepare the SOAP message and send it to the remote server that provides web services by using the HTTP/HTTPS protocol.

To simplify this process, most development environments (such as Microsoft Visual Studio and NetBeans) support importing of the WSDL definition file and provide automation tools for the creation of proxy classes. This approach enables you to access the object model in a convenient and familiar way, while ensuring compile-time verification of the web services calls.

Web Services API Objects

Interaction with the API is made through an object called **Screen**. This object acts as a gateway between the web services client and Acumatica Inc., so that you can sign in and retrieve, insert, update, or delete data, as well as perform any action that may be exposed by the form.

The preparation and execution of web services calls is facilitated by the **Content** object, which you can retrieve by calling the *GetSchema()* API function. This function returns an object that closely matches the way the form is presented to the end user. Each area on the form is mapped to an object in the **Content** object. For example, the Account Settings area in the **General Info** tab of the *Customers* (AR.30.30.00) form is defined in the **GeneralInfoAccountSettings** object. This object exposes a public property for every field in this area. Actions that can be performed in the form are exposed in a property called **Actions**. The class diagram below illustrates the relationship between the **Screen** and **Content** objects and associated areas of the **Content** object.

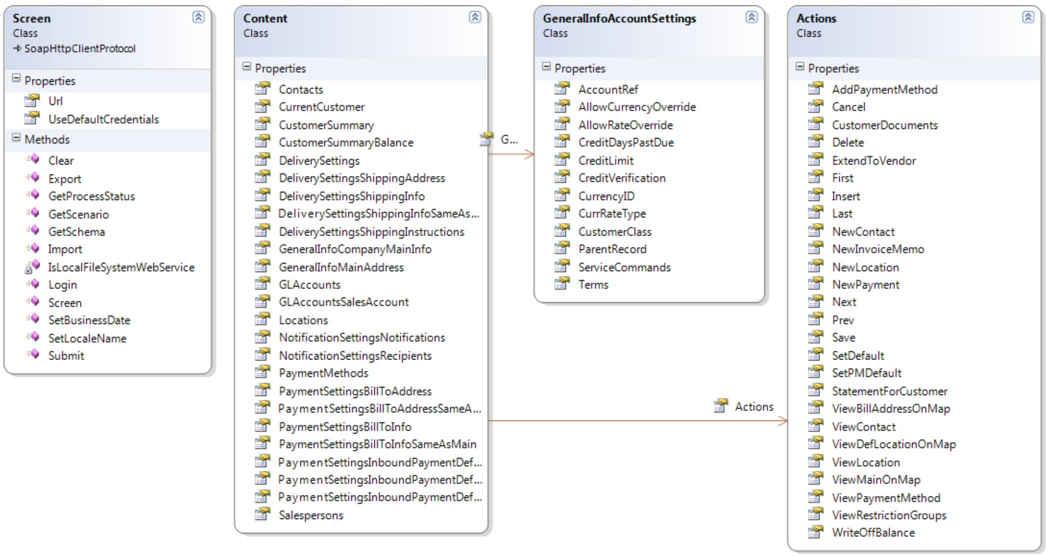


Figure: Sample web service class diagram

To execute an API call, you must build an array of commands and submit it to the form by calling the *Submit()* function. To process batch import and export operations, you define a scenario and use the *Import()* and *Export()* functions.

Quick Start

This mini-tutorial will help you get started with the Acumatica ERP Web Services Application Programming Interface (API). To begin working with the Web Services API, perform the following steps:

- *Generate and Locate the WSDL File of the Web Services*
- *Import the WSDL File of the Web Services Into the Development Environment*
- *Review and Use the Code From the Sample Project*

Step 1. Generate and Locate the WSDL File of the Web Services

Acumatica ERP automatically generates a WSDL file describing the operations (services) and an XML description of parameters and objects for a form or multiple forms. You can access this file through the *Web Services* (SM.20.70.40) form of Acumatica ERP.

For more information about the WSDL standard, see *Web Services Description Language (WSDL) 1.1*.

To create a WSDL file for multiple forms, perform the following actions:

1. On the Web Services form, click **Add New Record** on the form toolbar, and type the **Service ID** name (for instance, *APITEST*, as shown in the figure below).
2. Keep the **Import**, **Export**, and **Submit** check boxes selected (as they are by default), and leave the **Include Untyped** check box cleared. Click **Save**.

   If you also want to use untyped data to make it possible to manipulate string arrays instead of structured data, select the **Include Untyped** check box. The generated untyped operations have the *Untyped* prefix in their names—for instance, *UntypedSetSchema*, *UntypedExport*, and *UntypedSubmit*. The untyped operations cannot be used with specific forms. For instance, you can't generate the *UntypedGL301000Submit* operation, but you can generate the *GL301000Submit* operation.

3. Click **Add Row** on the table toolbar, and then add the value for the **Screen ID** column by using the lookup window and finding the *Payments and Applications* (AR.30.20.00) form.

4. Repeat the previous step to add each of the following forms, as shown in the figure below: *Customers* (AR.30.30.00), *Transactions* (CA.30.40.00), *Leads* (CR.30.10.00), *Contacts* (CR.30.20.00), *Business Account* (CR.30.30.00), *Opportunities* (CR.30.40.00), *Journal Transactions* (GL.30.10.00), *Stock Items* (IN.20.25.00), *Warehouses* (IN.20.40.00), *Transfers* (IN.30.40.00), *Purchase Receipts* (PO.30.20.00), *Sales Orders* (SO.30.10.00), and *Shipments* (SO.30.20.00). Click **Save** again.

The collection of forms you added above is necessary for using a single WSDL file in various kinds of examples that illustrate the use of the Web Services API. You can perform the instructions in these examples to learn the rules of syntax and the semantics of the API code, and then use the obtained experience in your work when you need to include a client application along with Acumatica ERP.

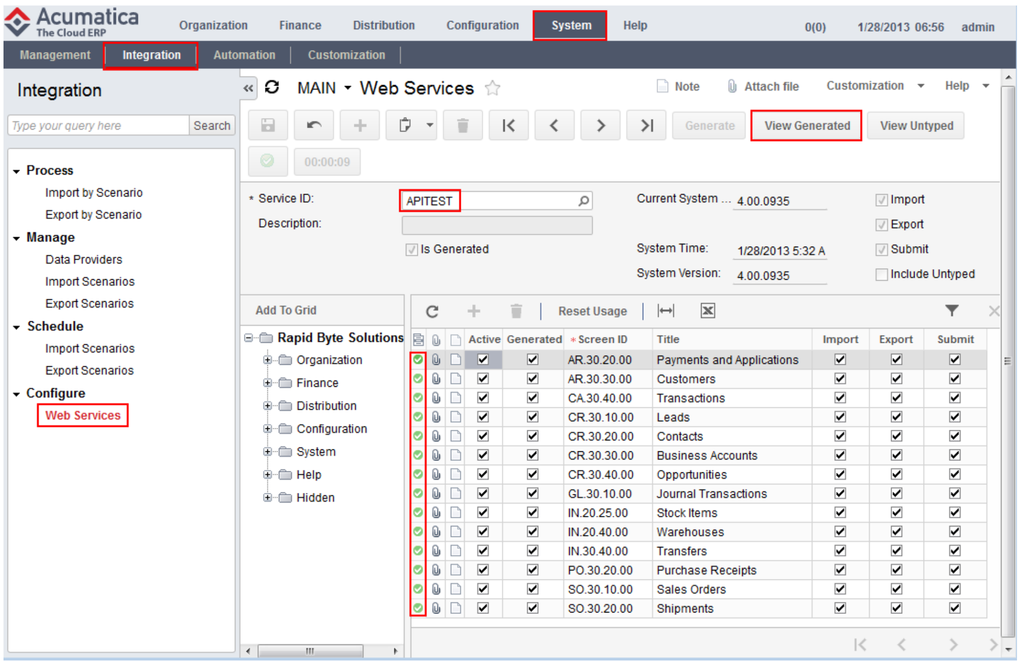


Figure: Creating the WSDL file

5. On the form toolbar, click **Generate** to start the process of generating the WSDL file. After the process is successfully completed, you can see the green flags in the leftmost column for each table row (that is, for each form).

6. Optional: Click **View Generated** to open the new window with the list of operations that are supported by the Acumatica ERP Web Services API, as illustrated in the figure below. Note that some operations are bound with specific forms, because these operations support the particular structure of the appropriate form. To see the examples of SOAP client requests and HTTP server responses that can be implemented by using the appropriate operation, click any item.

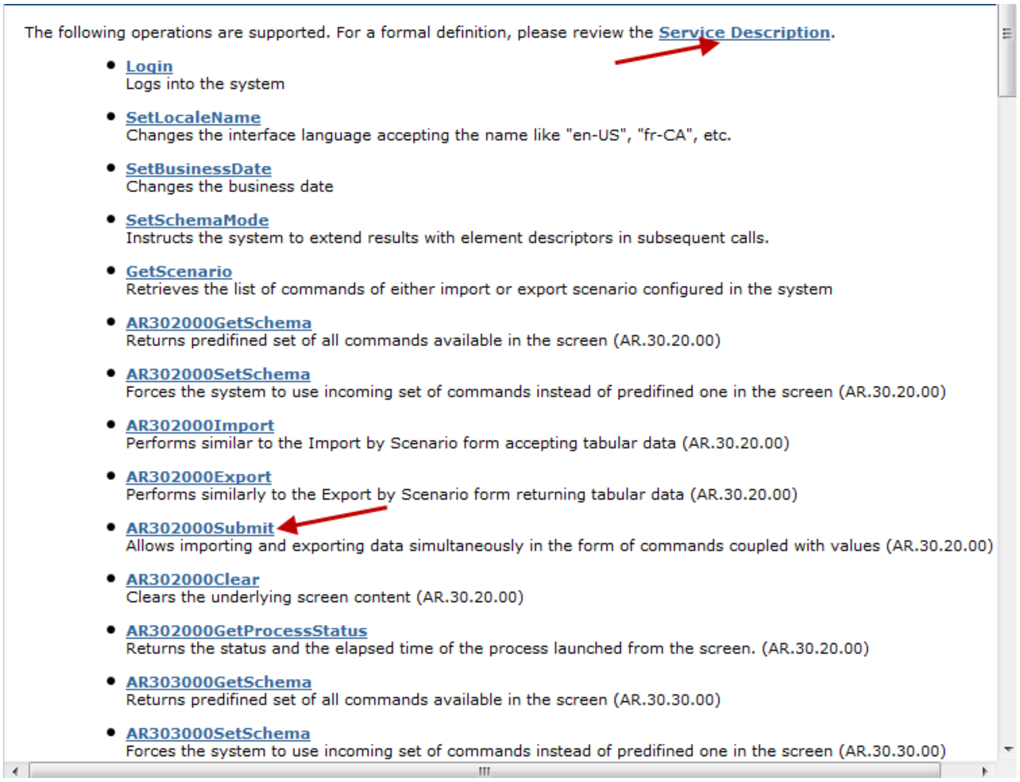


Figure: The list of available operations

7. Optional: Return to the previous screen, and click the **Service Description** reference to see the XML description of the generated WSDL file. A fragment of this file is shown in the figure below.
8. Close the window and return to the application.

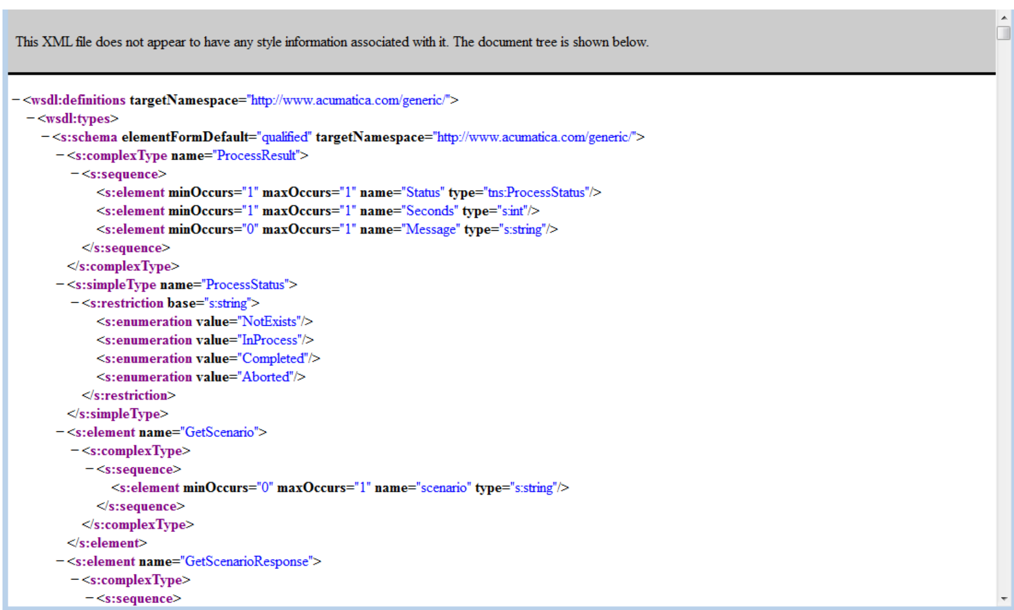


Figure: The XML description of the generated WSDL file

To find the latest version of the WSDL file, use the following URL:

```
http://{domain}/Soap/{name}.asmx?WSDL
```

Replace *domain* with the actual URL path to your application and *name* with the ID of the web service. For example, the valid URL to access the *Customers* form could be either of the following, with the latter for the local Acumatica ERP instance:

```
http://localhost:8080/Demo/Soap/APITEST.asmx?WSDL
http://localhost/WebAPIVirtual/Soap/APITEST.asmx?WSDL
```

The WSDL file automatically generated by the system includes all the changes implemented to the application logic and its database structure through the customization. If you made any customization that affects the business logic or database structure that you use through the API support of the form, make sure that you have retrieved the latest version of the WSDL file after the customization is published. You may generate the WSDL file any number of times.

Step 2. Import the WSDL File of the Web Services Into the Development Environment

When the WSDL file is generated, you must import it into your development environment to generate proxy classes. If necessary, see the documentation of your development environment to find out the correct way of building the proxy classes based on the WSDL definition.

Programming languages supported by Microsoft Visual Studio.NET can access the Web Services API through the proxy classes created by using the WSDL description for corresponding server-side objects. Below you will find instructions on how to implement the proxy classes by using Visual Studio 2008 or later and NetBeans 6.9.

To generate proxy classes from the WSDL definition by using Visual Studio 2008 or later:

1. Start MS Visual Studio and select **File** > **New** > **Project**.
2. In the **New Project** window that appears, select the required template; most examples of Acumatica ERP Web Service API implementation are based on the Visual C# *Console Application* template, although you can use any another template.
3. Define the name of the project and solution, as shown in the figure below, and click **OK**. (Although you can use any name for the project and solution, we recommend that you use a project name that is identical to the name of the solution that includes it.)

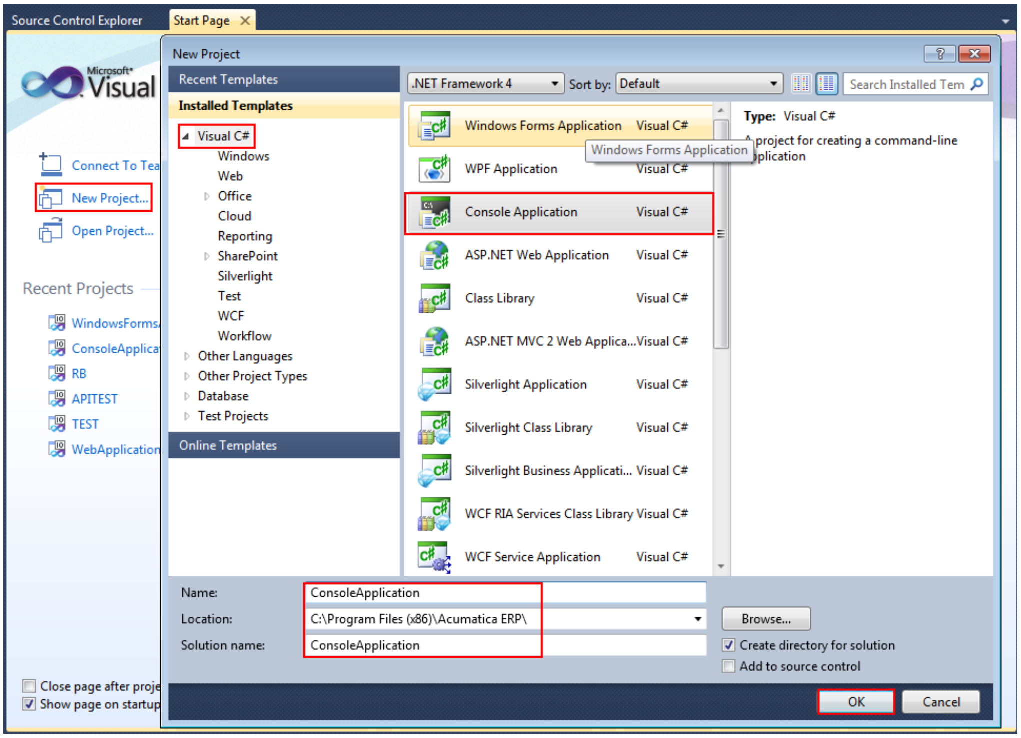


Figure: Creating the new project

4. Open the **Project** menu and select *Add Service Reference*.
5. In the dialog box that appears, click **Advanced**.
6. In the second dialog box that appears, click **Add Web Reference**.
7. In the third dialog box, type the path to Web Service WDSL descriptor file for the URL, as shown in the figure below. You can either use the local version of the WSDL file or provide the URL reference to the remote server.

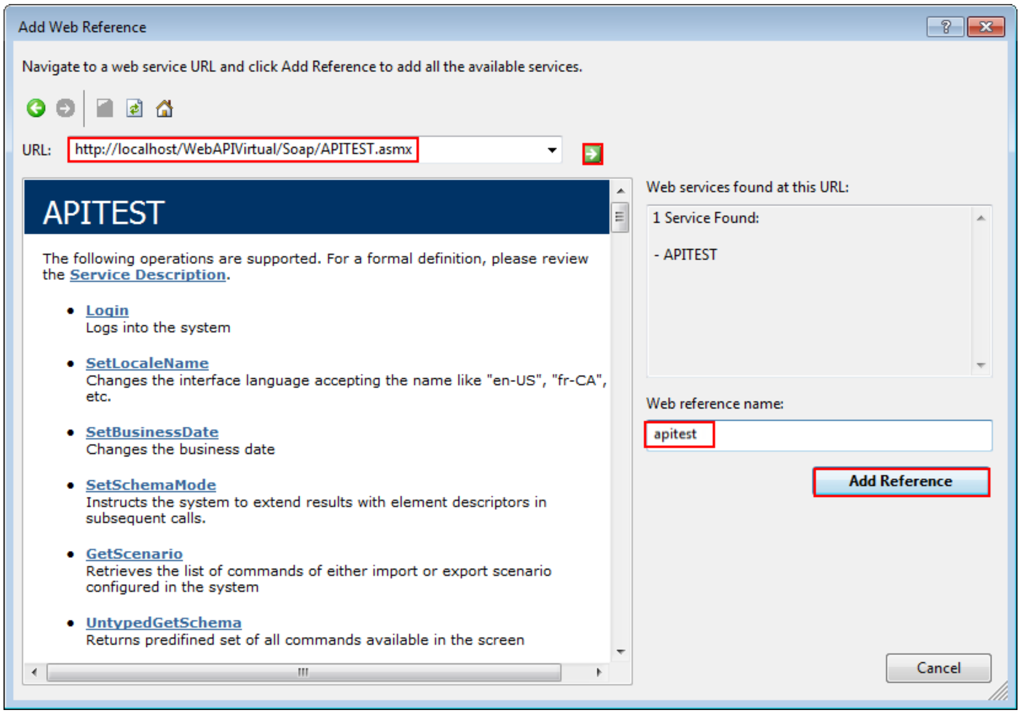


Figure: Specifying the URL of the WSDL file for the web reference

8. Click **GO** to continue.
9. Specify the **Web reference name**: *apitest*, for instance (see the figure above). This name will be used as a namespace for the generated web service proxy classes.
10. Click **Add Reference** to complete the creation process. As a result, in the Solution Explorer window, you can see the *Web References* folder with the reference to the WSDL file generated in Step 1, as shown in the figure below.

The new Visual Studio project now consists of the *Program* proxy class, which can be used for communication between the client application and Acumatica ERP Web Services. The communication program code must be added within the body of the *Program* proxy class.

Because you may access multiple web services in the same Acumatica ERP instance, we recommend that you name web references according to the original name of the WSDL file, but without capitalization: *apitest*.

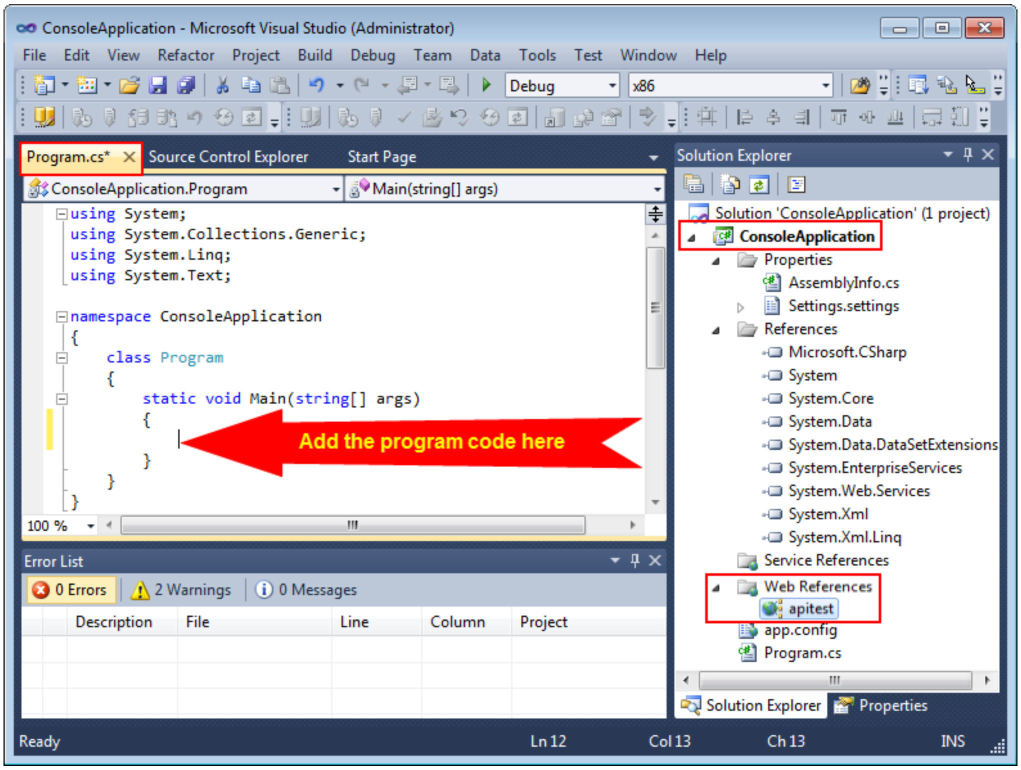


Figure: The *apitest* web reference and the *Program* proxy class

Java API for XML Web Services (JAX WS) supports the SOAP protocol and may be used with Acumatica Framework.

To generate proxy classes from the WSDL definition by using NetBeans 6.9 or later:

1. Right-click on your project, and select **New** > **Web Service Client**.
2. In the dialog box, for the URL input line, specify the path to the web service WDSL descriptor file.
3. Enter a package name.
4. Click **Finish** to complete the process.

NetBeans will process the specified WSDL definition and create a proxy class. This proxy class will be used for communication between the client application and the Acumatica ERP Web Service.

Step 3. Review and Use the Code From the Sample Project

Once you have imported the WSDL file and created the proxy class, you can start development of your client application. The fastest way to learn how to develop a client application by using the Web Services API is to learn and use the client application code from the sample project. The first typical solution can be found in *Exporting Warehouse Data*.

To avoid possible errors, pay attention to the following points:

1. To avoid unexpected code conflicts, create each example of the client application code within the project of the new empty solution. Otherwise, you should replace all previous code lines within the same project before starting to test the results of each code example.
2. Before adding the client application code, add to the proxy class code one line that contains the *using* command (as the figure below shows):

```
using ConsoleApplication.apitest;
```

Here *ConsoleApplication* is the name of your client application and *apitest* is the name of the bound web service.

3. Optional: Before you debug the client application, replace the URL of the WSDL file with the URL that corresponds to your file name and location. (In the figure below, you can see the example of the command line with the highlighted URL in the client application code that is to be replaced with the URL of your WSDL file.) This step is optional because if you don't specify the URL of the WSDL file, the system will use the URL set in the *App.Config* file.

4. Optional: Before debugging the client application, ensure that you have created the proper support of the authorization process; otherwise, you may need to make changes as follows (also shown in the figure below):

   - If your installation of Acumatica ERP includes the common company, use the simplest authorization code line:

     ```
     LoginResult result = context.Login("admin", "E618");
     ```

     Instead of *admin*, you may have another user name, but you should have enough rights to work with Web Services API services. Replace the password in the appropriate code line (*E618* by default) with the password that you had specified for the Acumatica ERP instance.

     In all the topics with examples, we use the common company and the simplest authorization code line.

   - If you work with more than one company but with the common branch, use the following modified authorization code line:

     ```
     LoginResult result = context.Login("user@CompanyCD", "E618");
     ```

     In the code line above, **Company CD** represents the required company short (CD) name.

   - If you work with more than one company and the company that you need has various branches, you should use the following modified authorization code line:

     ```
     LoginResult result = context.Login("user@CompanyCD:BranchCD", "E618");
     ```

     In the code line above, **CompanyCD** represents the required company short (CD) name, and **BranchCD** is the short branch name—that is, the CD name of the branch (for instance, *MAIN*, *NORTH*, or *SOUTH*) within the selected company.

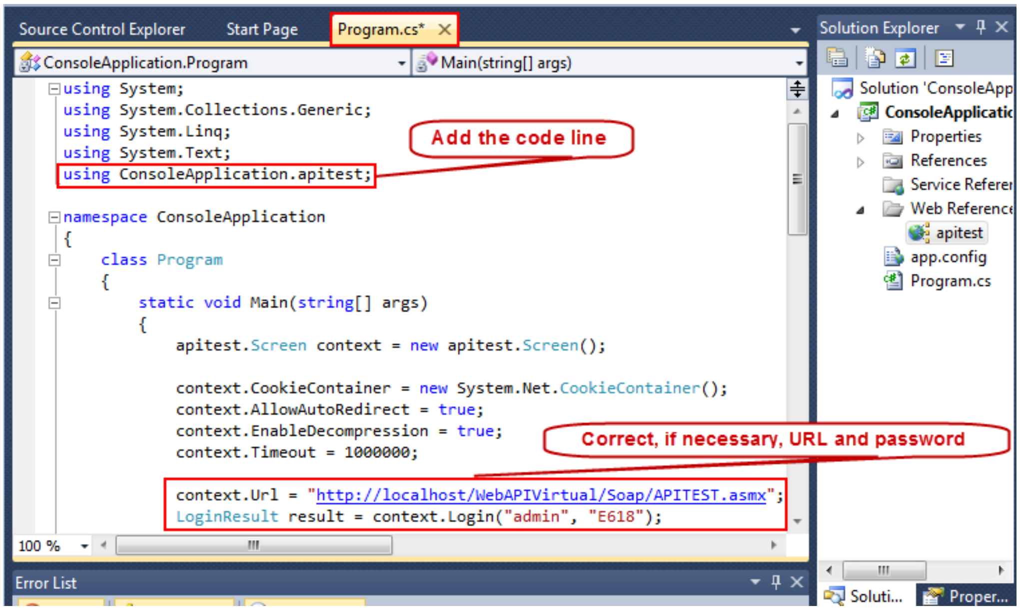


Figure: Correcting the code of the client application

Examples of the Web Service API Implementation

The examples in this section demonstrate how to use the following objects and properties of the Web Services API:

- **Screen**, an intermediary object that you will use for implementing the Web Services communication layer.
- The **CookieContainer** property, which preserves the session state between round trips. This property must be enabled in all client applications.
- **Content**, an object that defines the schema of the current form.

You can use the following links to directly access the examples of the Web Services API implementation:

- *Exporting Warehouse Data*
- *Exporting Stock Items*
- *Simulating the Behavior of Add Buttons on the Purchase Receipts Form*
- *Copying a Sales Order*
- *Adding a New Cash Transaction Document*
- *Adding Records to the Business Accounts and Opportunities Forms*
- *Importing of Data With an Image Into the Journal Transactions Form*
- *Exporting of Data With an Image From the Journal Transactions Form*

Exporting Warehouse Data

In this example, you create, run, and test a client application that exports to a string array required record fields from the *Warehouses* (IN.20.40.00) maintenance form of the Inventory module. The system filters exported data by the fixed **Warehouse ID** field value.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.

2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        IN204000Content IN204000 = context.IN204000GetSchema();
+        context.IN204000Clear();
+        string[][] IN204000result = context.IN204000Export
+        (
+           new Command[]
+           {
+             IN204000.WarehouseSummary.WarehouseID,
+             IN204000.LocationTableLocationTable.LocationID,
+             new Field { FieldName = "LocationID", ObjectName =
+                   IN204000.LocationTableLocationTable.LocationID.ObjectName }
+           },
+            new Filter[]
+            {
+               new Filter()
+               {
+                 Field = new Field() { FieldName = IN204000.WarehouseSummary.
+                                WarehouseID.FieldName, ObjectName = IN204000.
+                                   WarehouseSummary.WarehouseID.ObjectName },
+                 Condition = FilterCondition.Equals,
+                 Value = "GIT",
+                 Operator = FilterOperator.And
+               }
+           },
+           0, false, false
+        );
     }
   }
 }
```

This code implements the following process flow:

1. Using the *Export* method to export data from the form.
2. Using the *Filter* method to constrain the exported data by two fields of one record from the Warehouses form.

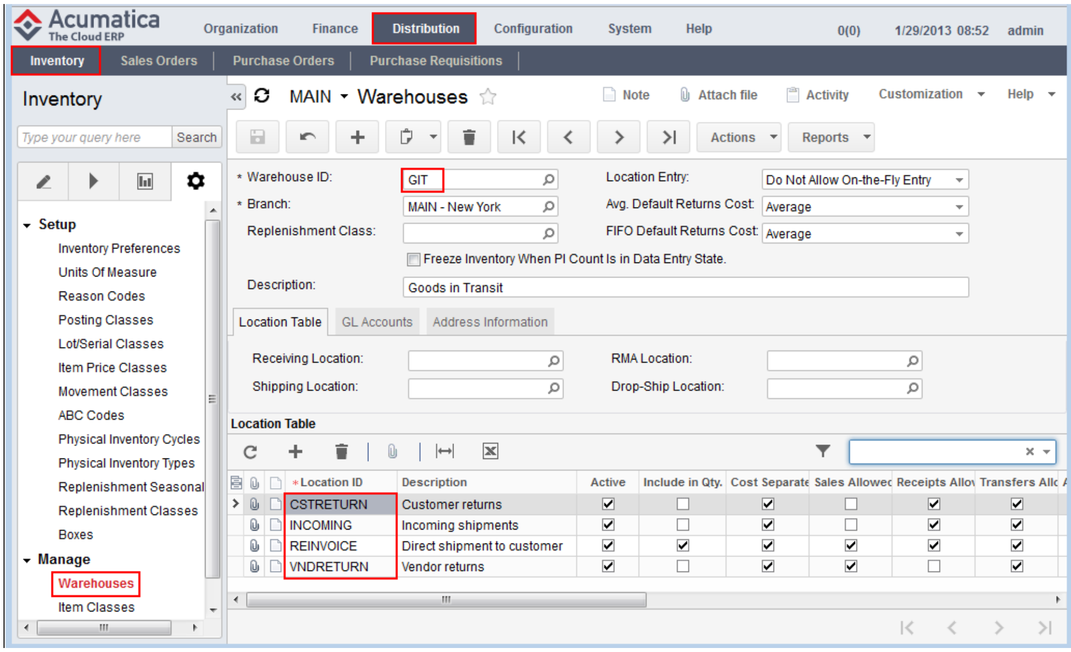


Figure: Exploring the Warehouses form

After you prepare the code, you should build the solution. Start the Acumatica ERP application instance with the WSDL file, navigate to **Distribution** > **Inventory**, select the **Configuration** submenu, and then select the **Manage** > **Warehouses** form. Select *GIT* as the **Warehouse ID**, and note the **Location ID** column values, as shown in the figure above. In Visual Studio, set appropriate breakpoints and then press F5 to run the client application in *Debug* mode. Use step-by-step debugging to ensure that the array contains exported data. (The figure below illustrates the test results.)

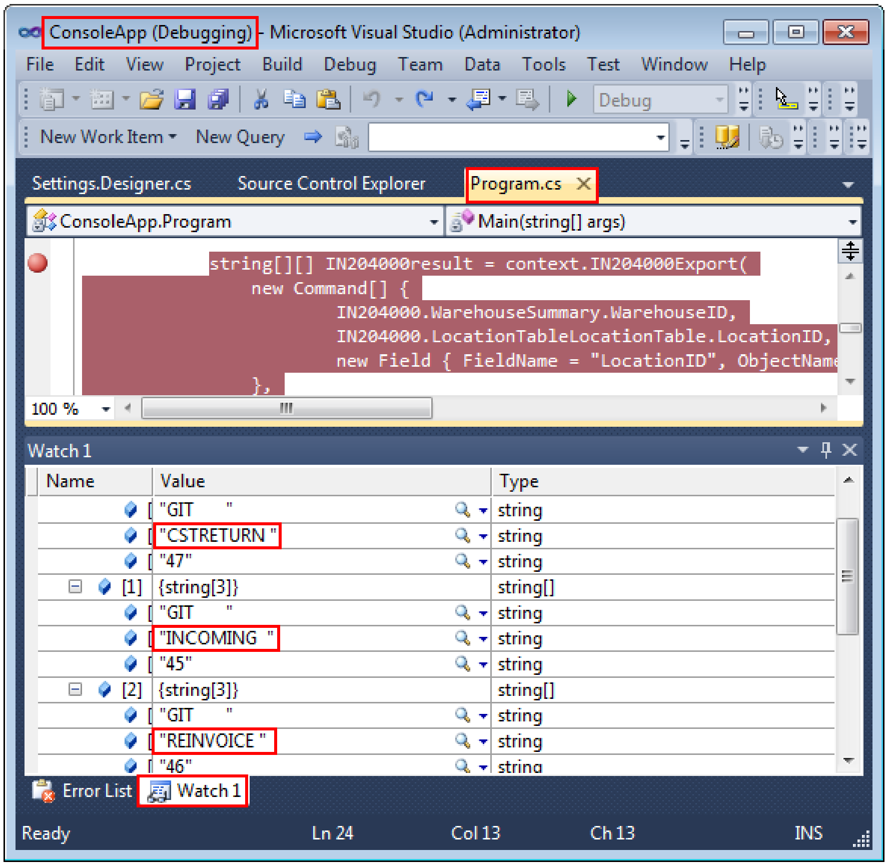


Figure: Checking the results in debug mode

Exporting Stock Items

In this example, you create, run, and test a client application that exports to a string array required record fields from the *Warehouses* (IN.20.25.00) maintenance form of the Inventory module. The system filter exports data by the hidden field **LastModifiedDateTime**. The date and time of the last modification of the Stock Items form must be fewer than 100 days before the current date.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add two *using* operators, as shown in the code below. (The added code lines are preceded by +.)

```
using System;
```

```
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using System.Globalization;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        context.SetLocaleName(CultureInfo.CurrentCulture.Name);
+        DateTime lastSyncDate = DateTime.UtcNow;
+        lastSyncDate = lastSyncDate.AddDays(-100);
+        IN202500Content IN202500 = context.IN202500GetSchema();
+        context.IN202500Clear();
+        string[][] IN202500data = context.IN202500Export
+        (
+           new Command[]
+           {
+             IN202500.StockItemSummary.ServiceCommands.EveryInventoryID,
+             IN202500.StockItemSummary.InventoryID,
+             IN202500.WarehouseDetails.Warehouse,
+             IN202500.WarehouseDetails.QtyOnHand,
+             new Field
+              {
+                ObjectName = IN202500.StockItemSummary.InventoryID.ObjectName,
+                                                FieldName = "LastModifiedDateTime"
+              }
+           },
+             new Filter []
+              {
+               new Filter
+                {
+                 Field = new Field { ObjectName =
+                          IN202500.StockItemSummary.InventoryID.ObjectName,
+                                         FieldName = "LastModifiedDateTime" },
+                 Condition = FilterCondition.Greater,
+                 Value = lastSyncDate.ToLongDateString()
+               }
+           },
+           0, false, false
+       );
     }
   }
 }
```

This code implements the following process flow:

1. Using the *Export* method to export data from the form.
2. Using the *Filter* method to limit the exported data by the date and time of the last modification of the Stock Items form. (The date and time of the last modification must be fewer than 100 days before the current date.)

   You defined the *lastSyncDate* variable, which is used to limit the quantity of data being exported depending on the current date and time.

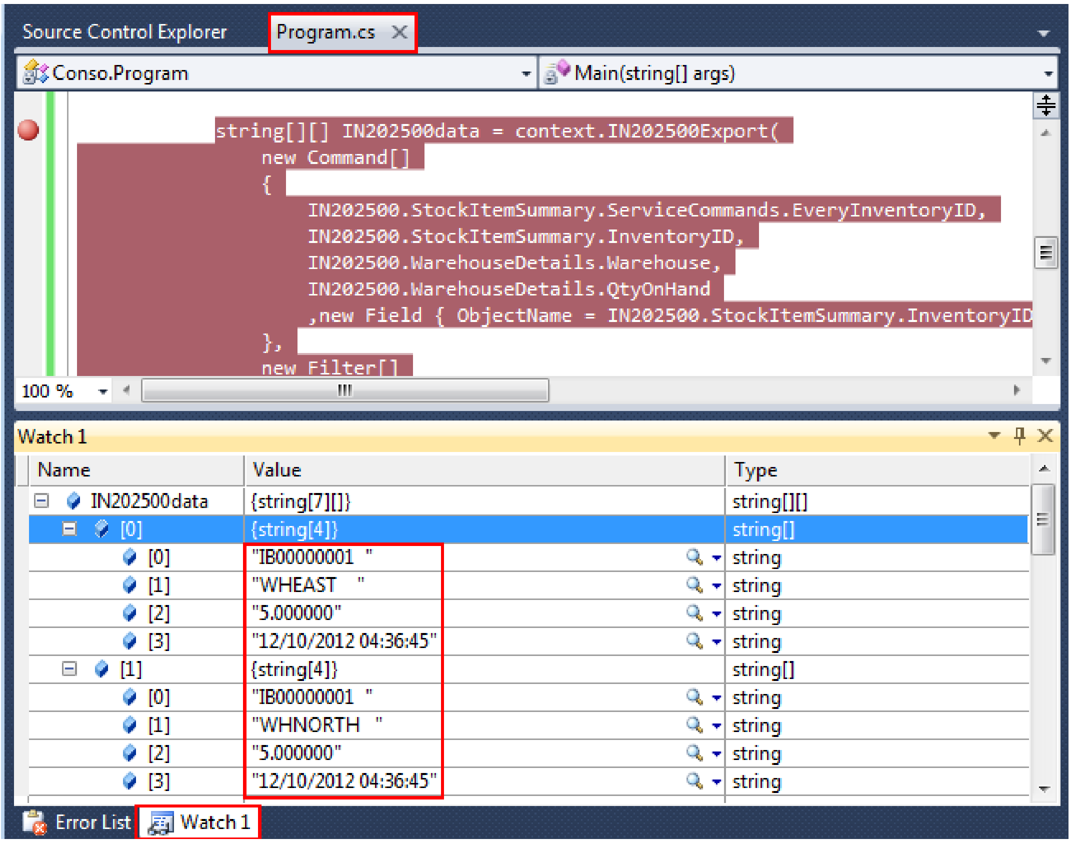


Figure: Checking the results in debug mode

After preparing the code, you should build the solution. Set appropriate breakpoints and then press F5 to run the client application in *Debug* mode. Use step-by-step debugging to ensure that the array contains exported data. (The figure above illustrates the test results.)

Optionally, you can start the Acumatica ERP application instance with the WSDL file and navigate to the **Distribution** > **Inventory** > **Manage** > **Stock Items** form. Select *IB00000001* as the **Inventory ID**, open the **Warehouse Detail** tab, and note the **Warehouse** and **Qty On Hand**column values, as shown in the figure below. Compare the column values with the values in the string array, displayed in the **Watch** window of Visual Studio in debug mode.

If no data has been exported, increase the number of subtracted days in the `lastSyncDate = lastSyncDate.AddDays(-100);` code line and repeat the data export.

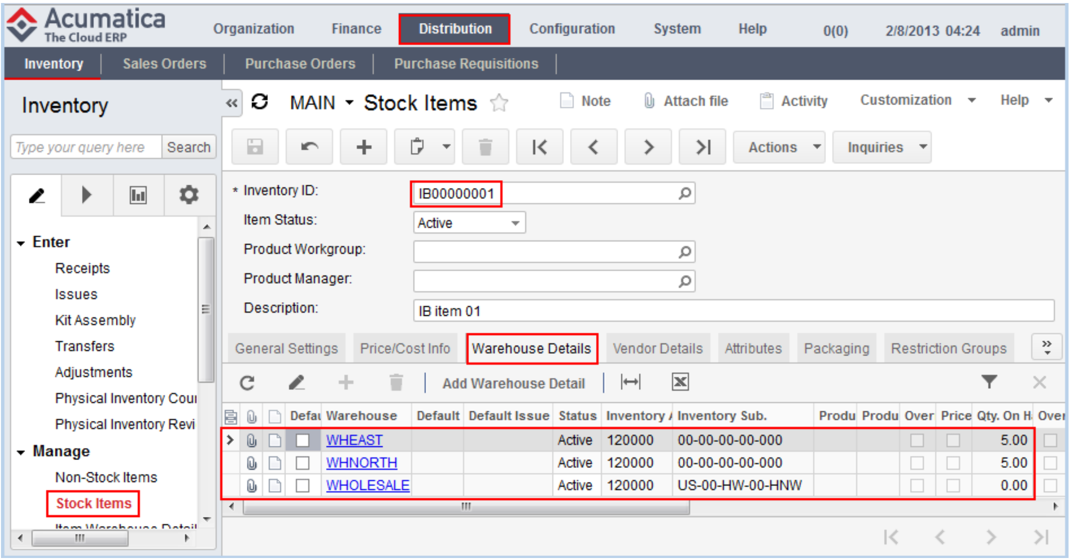


Figure: Exploring the Warehouse Details tab of the Stock Items form

Simulating the Behavior of Add Buttons on the Purchase Receipts Form

In this example, you create, run, and test a command-line client application that adds lines to the details table of the *Purchase Receipts* (PO.30.20.00) form from the details table of the *Purchase Orders* (PO.30.10.00)form. (Both forms are located in the Purchase Orders module.) The client application will add lines from all purchase orders that have the same **VendorCD** field value. The application will imitate a user clicking the **Add PO** (the *AddPOOrder* action is called) button on the Purchase Receipts form and the user's next few steps.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

In the steps below, before you create the code, you will add a purchase order and a purchase receipt. These tasks are necessary to test the result of running the client application when we know the values of key fields and use them in the code lines.

Do the following actions:

1. Start Acumatica ERP, and navigate to the **Distribution** > **Purchase Orders** > **Enter** > **Purchase Orders** form. Add a purchase order with the *Normal* type and the following values: *PORG000084* as the **Order Nbr**, *ACITAISYST* as the **Vendor**, and *MAIN* as the **Location**. Click **Save**.
2. Add to the **Document Details** tab three lines with any **Inventory ID**, **Order Qty**, and **Unit Cost** column values (as an example, see the figure below). Add *0* as the **Subitem** value (this

column cannot be empty). Add the **Control Total** value (if this field appears in your system) so that it equals the **Order Total** value, and fill in the other mandatory fields (designated with asterisks); otherwise, the purchase order will not be saved. Click **Save**.

3. Clear the **Hold** check box and click **Save**.

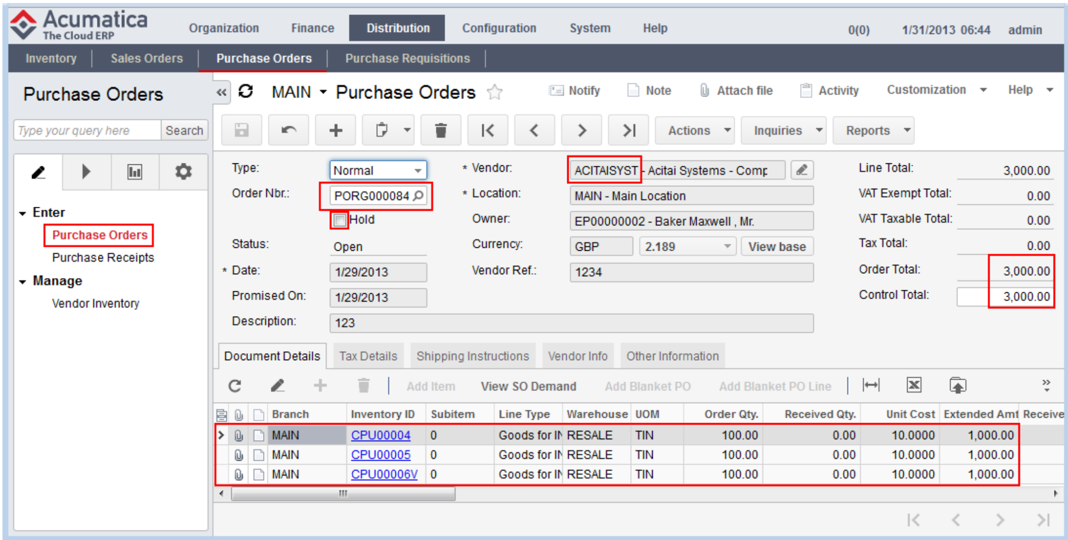


Figure: Creating a new purchase order

4. Navigate to the **Distribution** > **Purchase Orders** > **Enter** > **Purchase Receipts** form, and add a receipt with the *Receipt* type and the following values: *PORE000079* as the **Receipt Nbr.**, *ACITAISYST* as the **Vendor**, and *MAIN* as the **Location**. Click **Save**.

5. Click **Add PO** on the table toolbar of the **Document Details** tab.

6. In the table of the **Add Purchase Order** dialog box that appears, notice one line with the field values of the purchase order added before. (In other cases, more than one line or no lines may be displayed.) Select the unlabeled check box and click **Add & Close**, as shown in the figure below. Notice that the **Add Purchase Order** window is closed, while on the **Document Details** tab, the three lines have been added. (You can see this in the figure in the end of this article.) You will implement this scenario in the C# client application code. Click **Cancel** to not save the added lines.

If you implement within one client application another scenario, based on the **Add PO Line** button, you should obtain the same result. You can prepare the code for the second scenario independently. This code is shown at the end of this topic; see *code of the second scenario*.

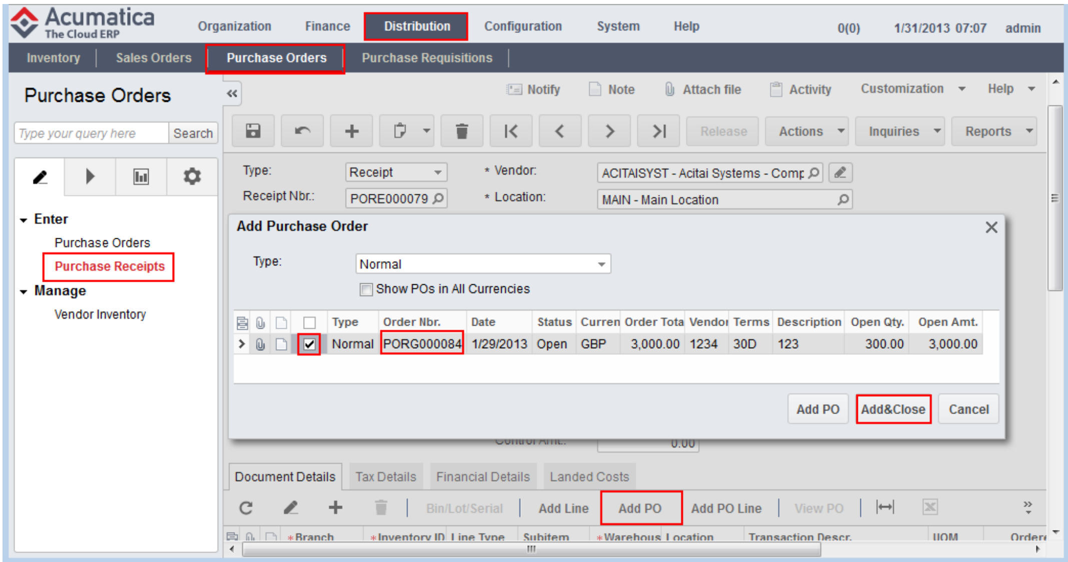


Figure: Adding new lines to the details table of the purchase receipt

7. Add code lines to the *Program* proxy class code but previously add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        PO302000Content PO302000 = context.PO302000GetSchema();
+        context.PO302000Clear();
+        PO302000.Actions.AddPOOrder.Commit = true;
+        PO302000.Actions.AddPOOrder2.Commit = true;
+        PO302000.AddPurchaseOrder.Selected.LinkedCommand = null;
+        PO302000.DocumentDetails.InventoryID.LinkedCommand = null;

+        PO302000Content[] PO302000result = context.PO302000Submit
+        (
+         new Command[]
+        {
+         new Value { Value = "PORE000079", LinkedCommand =
+
 PO302000.DocumentSummary.ReceiptNbr },
+         new Value { Value = "OK", LinkedCommand =
+          PO302000.AddPurchaseOrder.ServiceCommands.DialogAnswer, Commit =
 true },
+          //uncomment the next two lines if you want to use multicurrency
 orders
+          //new Value { Value = "True", LinkedCommand =
```

```
+           //    PO302000.AddPurchaseOrderPOSelection.AnyCurrency, Commit =
 true },
+             PO302000.Actions.AddPOOrder,
+             new Key { Value = "='Normal'", FieldName =
+
 PO302000.AddPurchaseOrder.OrderType.FieldName,
+                    ObjectName =
 PO302000.AddPurchaseOrder.OrderType.ObjectName },
+             new Key { Value = "='PORG000084'", FieldName =
+
 PO302000.AddPurchaseOrder.OrderNbr.FieldName,
+                     ObjectName =
 PO302000.AddPurchaseOrder.OrderNbr.ObjectName },
+             new Value { Value = "True", LinkedCommand =
+                            PO302000.AddPurchaseOrder.Selected, Commit =
 true },
+             PO302000.Actions.AddPOOrder2
+         }
+         );
      }
   }
 }
```

If you created a purchase order with another **Order Nbr.** value or a receipt with another **Receipt Nbr.** value, use the real document ID value.

8. Build the solution, open the application, and press F5 to run the client application in *Debug* mode.
9. Again open Acumatica ERP, refresh the form, and ensure that the three lines have been added as a result of running the client application. (The figure below illustrates the test results.)

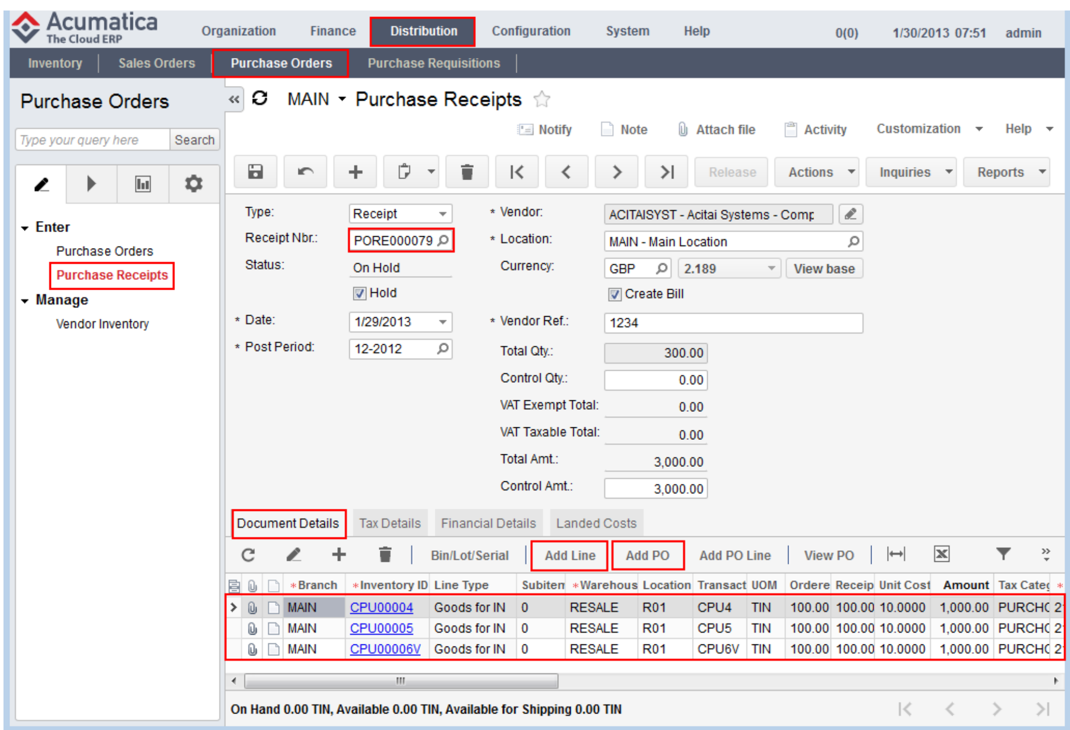


Figure: Three added lines as a result of running the client application code

This code implements the following process flow:

1. Activating the *AddPOOrder* and *AddPOOrder2* actions.

2. Invoking the *AddPOOrder* and *AddPOOrder2* actions to imitate adding lines to the details table by using the scenario that had been implemented for the **Add PO** button in Acumatica ERP: selecting all the records in the table of the **Add Purchase Order** dialog box (after invoking the *Add PO Line* button, you can also specify through the code the required purchase order number), and clicking the **Add & Close** button.

The code for the second scenario follows.

```
apitest.Screen context = new apitest.Screen();
          context.CookieContainer = new System.Net.CookieContainer();
          context.AllowAutoRedirect = true;
          context.EnableDecompression = true;
          context.Timeout = 1000000;
          context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
          LoginResult result = context.Login("admin", "E618");

          PO302000.Actions.AddPOOrderLine.Commit = true;
          PO302000.Actions.AddPOOrderLine2.Commit = true;
          PO302000.AddPurchaseOrderLine.Selected.LinkedCommand = null;
          PO302000.DocumentDetails_.InventoryID.LinkedCommand = null;
          PO302000result = context.PO302000Submit(
          new Command[]
          {
            new Value { Value = "PORE000079", LinkedCommand =
                                    PO302000.DocumentSummary.ReceiptNbr},
            new Value { Value = "OK", LinkedCommand =
                   PO302000.AddPurchaseOrderLine.ServiceCommands.DialogAnswer,
                                                        Commit = true },
            PO302000.Actions.AddPOOrderLine,
            new Key { Value = "='PORG000084'", FieldName =
                  PO302000.AddPurchaseOrderLine.OrderNbr.FieldName, ObjectName =
                             PO302000.AddPurchaseOrderLine.OrderNbr.ObjectName },
            new Key { Value = "='CPU00004'", FieldName =
              PO302000.AddPurchaseOrderLine.InventoryID.FieldName, ObjectName =
                          PO302000.AddPurchaseOrderLine.InventoryID.ObjectName },
            new Value{ Value = "True", LinkedCommand =
                         PO302000.AddPurchaseOrderLine.Selected, Commit = true },
                                               PO302000.Actions.AddPOOrderLine2
            new Key{ Value = "='CPU00004'", FieldName =
                 PO302000.DocumentDetails_.InventoryID.FieldName, ObjectName =
                               PO302000.DocumentDetails_.InventoryID.ObjectName},
            new Value{ Value = "1.00", LinkedCommand =
                           PO302000.DocumentDetails_.ReceiptQty, Commit = true},
           // the next part of code is needed if you use Serial items
           PO302000.BinLotSerialNumbers.ServiceCommands.NewRow,
           new Value { Value = "R01", LinkedCommand =
                                 PO302000.BinLotSerialNumbers.Location },
           new Value { Value = "1.00", LinkedCommand =
                        PO302000.BinLotSerialNumbers.Quantity, Commit = true },
           new Value { Value = "25.00", LinkedCommand =
                           PO302000.DocumentDetails_.UnitCost, Commit = true },
           new Key { Value = "='CPU00004'", FieldName =
                  PO302000.DocumentDetails_.InventoryID.FieldName, ObjectName =
                              PO302000.DocumentDetails_.InventoryID.ObjectName },
           new Value { Value = "0.00", LinkedCommand =
                          PO302000.DocumentDetails_.ReceiptQty, Commit = true }
           PO302000.Actions.Save
                     }
                );
```

If you created a purchase order with another **Order Nbr.** value or a receipt with another **Receipt Nbr.** value, use the real document ID value.

Copying a Sales Order

In this example, you create, run, and test a simple command-line client application that copies key field and column values from an existing *Sales Orders* (SO.30.10.00) form of the Sales Orders module and pastes the values into an added sales order.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also use your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

In the steps below, before you create the code example, you will ensure that a particular sales order with a specific **Order Nbr.** value exists—the order we plan to copy—and check some of its values. This step is necessary so you can later make sure the copying operation worked appropriately.

Do the following actions:

1. Start Acumatica ERP, and navigate to **Distribution** > **Sales Orders** > **Enter** > **Sales Orders**. In the **Order Type** field, select *SO*, and in the **Order Nbr.** field, select (by using the lookup window) *000097*. Note the values of the **Inventory ID** column in the details table on the **Document Details** tab (for the three rows) and the **Order Total** field in the main area of the form. (See the figure below).

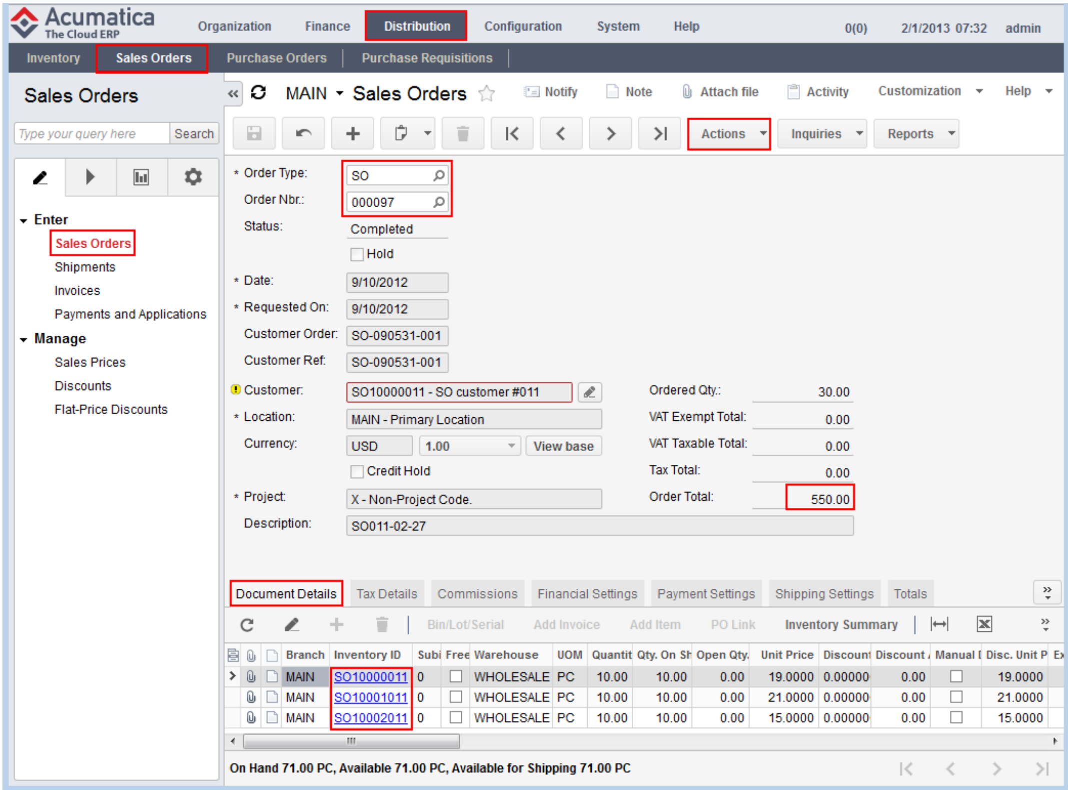


Figure: The existing sales order

If you select *Copy Order* on the **Actions** menu, you can create a new order by using the internal Acumatica ERP *Copy Order* operation. This example imitates the copying operation by using the external client application code.

2. Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        SO301000Content SO301000 = context.SO301000GetSchema();
+        context.SO301000Clear();
+        SO301000.Actions.CopyOrder.Commit = true;
+        SO301000Content[] SO301000Content = context.SO301000Submit
+        (
+        new Command[]
```

```
+           {
+              new Value { Value = "SO", LinkedCommand =
+
 SO301000.OrderSummary.OrderType },
+              new Value { Value = "000097", LinkedCommand =
+
 SO301000.OrderSummary.OrderNbr },
+              new Value { Value = "OK", LinkedCommand =
+
 SO301000.CopyTo.ServiceCommands.DialogAnswer },
+              new Value { Value = "QT", LinkedCommand =
+
 SO301000.CopyTo.OrderType},
+           SO301000.Actions.CopyOrder,
+           SO301000.Actions.Save,
+           SO301000.OrderSummary.OrderNbr
+           }
+         );
      }
   }
 }
```

3. Build the solution, open the application, and press F5 to run the client application in *Debug* mode.
4. Again open Acumatica ERP and navigate to the Sales Order form. Select *QT* in the **Order Type** field, and select the new sales order (with the highest **Order Nbr.** value).Ensure that same three lines that existed in sales order *000097* have been added to the details table after you ran the client application, and make sure the **Order Total** field has the same value that you noted in sales order *000097*. (The figure below illustrates the test results.)

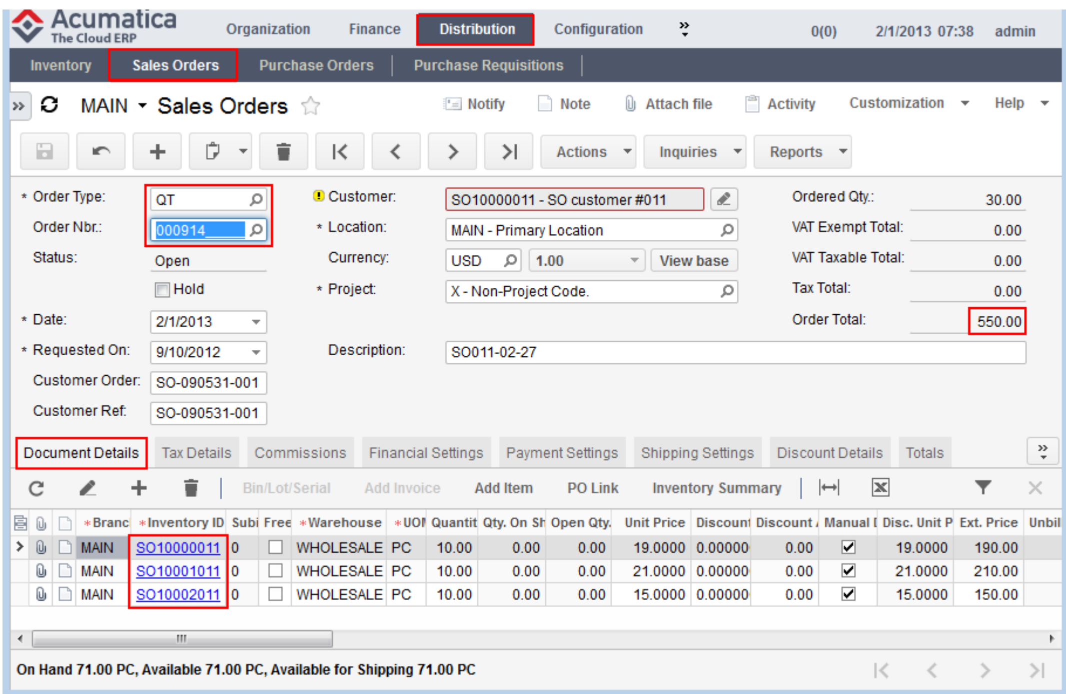


Figure: The added sales order as a result of running the client application code

As the introduction mentions, this code represents the simple example of a client application that is used for inserting a new sales order by copying many of its settings from an existing one. This code implements the following process flow:

1. Using the *Submit* method to provide the copying operation.

2. Invoking the *CopyOrder* action to imitate the selection of the *Copy Order* option on the **Actions** menu of the form.
3. Using the *SO301000.OrderSummary.OrderNbr* command to invoke the document autonumbering method implemented in Acumatica ERP.

Adding a New Cash Transaction Document

In this example, you create, run, and test a simple command-line client application that adds a new cash transaction document to the *Transactions* (CA.30.40.00) form of the Cash Management module.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        try
+        {
+           CA304000Content CA304000 = context.CA304000GetSchema();
+           context.CA304000Clear();
+           CA304000Content[] CA304000result = context.CA304000Submit
+           (
+           new Command[]
+           {
+             new Value { Value = "100000", LinkedCommand =
+                                  CA304000.TransactionSummary.CashAccount },
+             new Value { Value = "PETTYEXP", LinkedCommand =
+                                        CA304000.TransactionSummary.EntryType },
+             new Value { Value = "111", LinkedCommand =
```

```
+                                    CA304000.TransactionSummary.DocumentRef },
+             new Value { Value = "true", LinkedCommand =
+                                       CA304000.TransactionSummary.Approved },
+             CA304000.TransactionDetails.ServiceCommands.NewRow,
+             new Value { Value = "408000", LinkedCommand =
+                                  CA304000.TransactionDetails.OffsetAccount },
+             new Value { Value = "00-00-00-00-000", LinkedCommand =
+                               CA304000.TransactionDetails.OffsetSubaccount },
+             new Value { Value = "1", LinkedCommand =
+                                        CA304000.TransactionDetails.Quantity },
+             new Value { Value = "100", LinkedCommand =
+                            CA304000.TransactionDetails.Price, Commit = true },
+             new Value { Value = "100", LinkedCommand =
+                    CA304000.TransactionSummary.ControlTotal, Commit = true },
+             CA304000.Actions.Save, CA304000.TransactionSummary.ReferenceNbr
+          }
+          );
+        }
+        catch (Exception ex)
+        {
+          Console.WriteLine(ex.Message);
+        }
     }
  }
 }
```

As the introduction mentions, this code represents the simple example of a client application that is used for inserting a new cash transaction document into the Transactions form of the Cash Management module. This code implements the following process flow:

1. Using the *Submit* method to add data to the form.
2. Invoking the *Save* action in the form.
3. Using the *CA304000.TransactionSummary.ReferenceNbr* command to invoke the document autonumbering method implemented in Acumatica ERP.

After preparing the code, you can build the solution and then press F5 to run the client application in *Debug* mode. Start the Acumatica ERP application instance with the WSDL file, and navigate to **Finance** > **Cash Management** > **Enter** > **Transactions** to open the Transactions form. In the **Reference Nbr.** field, select the added transaction item (which has the highest reference number). Ensure that the item has been added with the needed values. (See the figure below.)

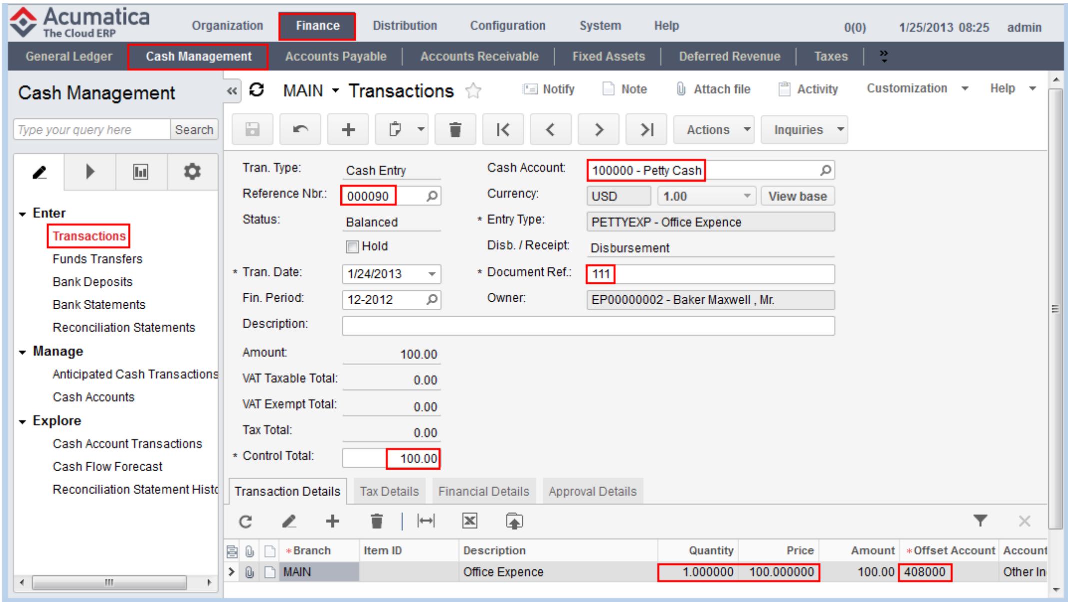


Figure: Testing the result of running the client application

Adding Records to the Business Accounts and Opportunities Forms

In this example, you create, run, and test a simple command-line client application that adds new records to the *Business Accounts* (CR.30.30.00) and *Opportunities* (CR.30.40.00) forms of the Customer Management module.

As with the previous example, we make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
```

```
+       apitest.Screen context = new apitest.Screen();
+       context.CookieContainer = new System.Net.CookieContainer();
+       context.AllowAutoRedirect = true;
+       context.EnableDecompression = true;
+       context.Timeout = 1000000;
+       context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+       LoginResult result = context.Login("admin", "E618");

+       CR303000Content CR303000 = context.CR303000GetSchema();
+       context.CR303000Clear();
+       CR303000Content[] CR303000Content = context.CR303000Submit
+      (
+       new Command[]
+          {
+            new Value { Value = "TEST123", LinkedCommand =
+                                   CR303000.AccountSummary.BusinessAccount },
+            new Value { Value = "TEST123", LinkedCommand =
+                                   CR303000.AccountSummary.BusinessAccountName },
+            new Value { Value = "US", LinkedCommand =
+                                          CR303000.DetailsMainAddress.Country },
+            new Value { Value = "Industry", LinkedCommand =
                                                 CR303000.Attributes.Attribute },
+            new Value { Value = "Banking", LinkedCommand =
+                                     CR303000.Attributes.Value, Commit = true },
+            CR303000.Actions.Save
+          }
+       );

+       CR304000Content CR304000 = context.CR304000GetSchema();
+       context.CR304000Clear();
+       CR304000Content[] CR304000Content = context.CR304000Submit
+       (
+         new Command[]
+         {
+           new Value { Value = "TEST123", LinkedCommand =
+                                 CR304000.OpportunitySummary.BusinessAccount },
+           new Value { Value = "MAIN", LinkedCommand =
+                                        CR304000.OpportunitySummary.NoteText },
+           new Value { Value = "INSIDE", LinkedCommand =
+                                                  CR304000.Details.ClassID },
+           new Value { Value = "DESCRIPTION", LinkedCommand =
+                                       CR304000.OpportunitySummary.Subject },
+           CR304000.Actions.Save
+        }
+      );
    }
  }
}
```

As the introduction mentions, this code represents the simple example of a client application that adds new records to the Business Accounts and Opportunities forms of the Customer Management module. This code implements the following process flow:

1. Using the *Submit* method to add data to the forms.
2. Invoking the *Save* action in the form.

After preparing the code, you can build the solution and then press F5 to run the client application in *Debug* mode. Perform the following actions:

- Start the Acumatica ERP application instance with the WSDL file. Navigate to **Organization** > **Cash Management** > **Manage** > **Business Account** to open the Business Account form, and in the **Business Account** field, find the added record by using a quick search and select it. (The new record has the number *TEST123*.) Ensure that the record has been added with the needed values. (See the figure below.)

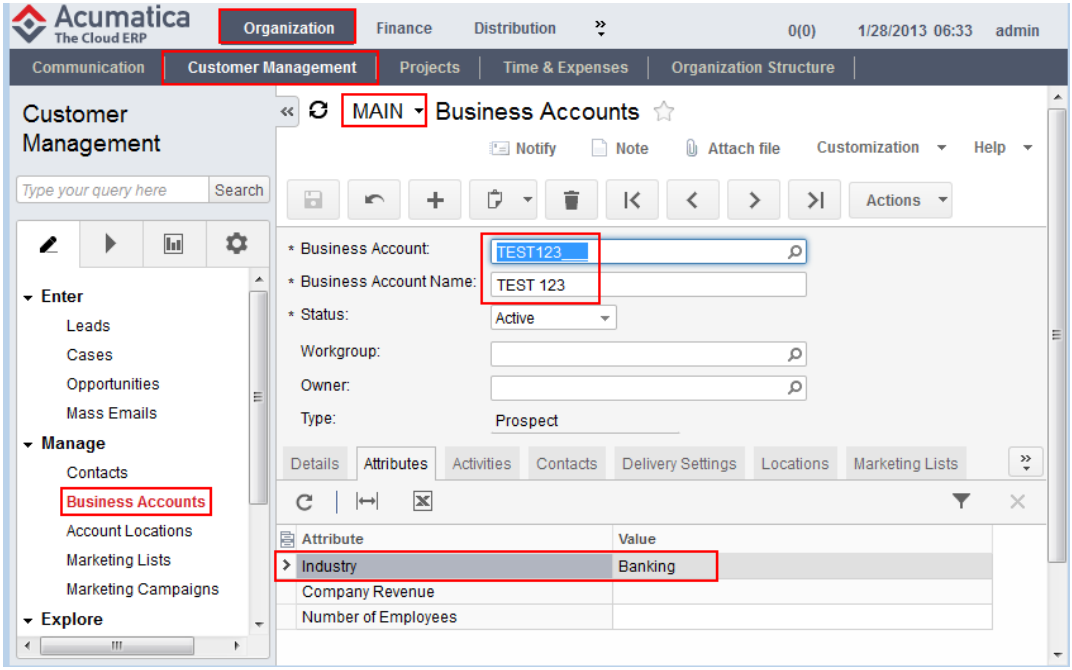


Figure: Testing the first result of running client application

- Navigate to **Organization** > **Cash Management** > **Manage** > **Opportunities** to open the Opportunities form, and in the **Opportunity ID** field, select the added record, which has the highest reference number. Ensure that the record has been added with the needed values. (See the figure below.)

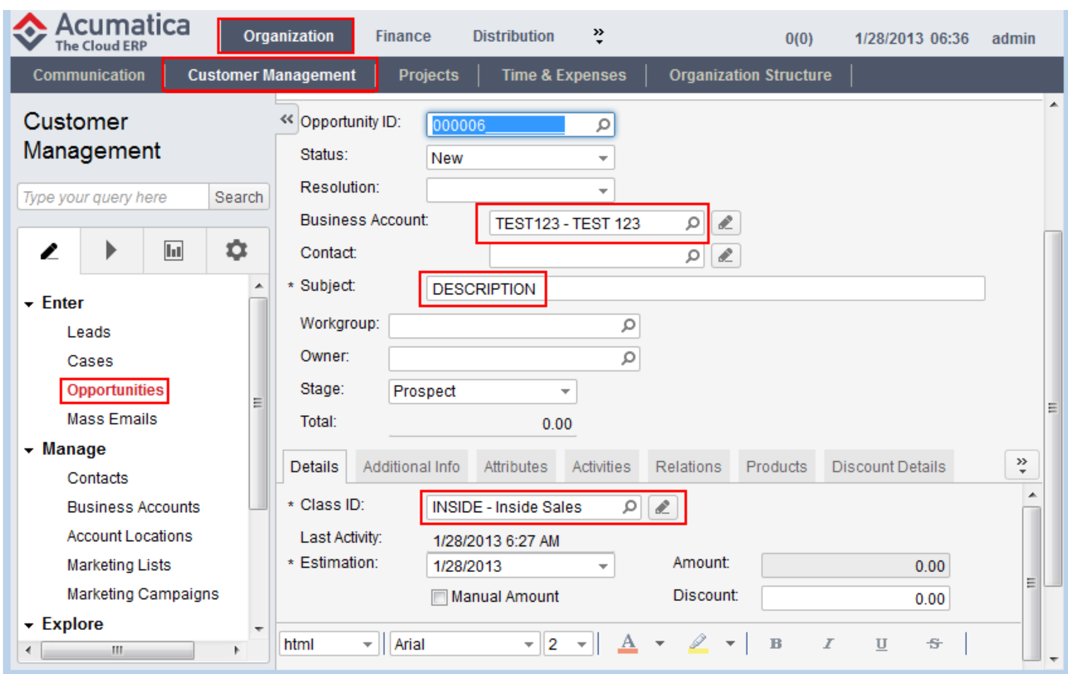


Figure: Testing the second result of running the client application

Importing of Data With an Image Into the Journal Transactions Form

In this example, you create, run, and test a client application that enables the import data with an image into the *Journal Transactions* (GL.30.10.00) form of the General Ledger module.

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        byte[] filedata;
+        using (System.IO.FileStream file =
+                System.IO.File.Open(@"D:\01.jpg", System.IO.FileMode.Open))
+          {
+            filedata = new byte[file.Length];
+            file.Read(filedata, 0, filedata.Length);
+          }
+        GL301000Content GL301000 = context.GL301000GetSchema();
+        context.GL301000Clear();
+        GL301000ImportResult[] GL301000ImportResult = context.GL301000Import
+        (
+           new Command[]
+            {
+            new Value
+             {
+               Value = "GL", LinkedCommand = GL301000.BatchSummary.Module },
+               GL301000.BatchSummary.BatchNumber,
+               GL301000.BatchSummary.ControlTotal,
+               new Value
+                {
+                    FieldName = "01.jpg", LinkedCommand =
-                         GL301000.BatchSummary.ServiceCommands.Attachment
+                }
+              GL301000.TransactionDetails.Account,
+              GL301000.TransactionDetails.Subaccount,
```

```
+                GL301000.TransactionDetails.RefNumber,
+                GL301000.TransactionDetails.CreditAmount,
+                GL301000.TransactionDetails.DebitAmount,
+                GL301000.Actions.Save
+              },
+              null,
+              new string [][]
+               { new string[] { "00003849", "10", Convert.ToBase64String(filedata),
+                               "100000", "US-00-00-00-000", "REF", "10,0", "0,0" },
+                new string[] { "00003849", "10", Convert.ToBase64String(filedata),
+                               "101000", "US-00-00-00-000", "REF", "0,0", "10,0" },
+               },
+               false, false, true);
+         );
      }
    }
  }
```

This code implements the following process flow:

1. Using the *Import* method to import data into the form.
2. Using the standard .Net classes (*FileStream*, *File*, and *FileMode*) to open and read the byte content of the external file.
3. Using the *Attachment* service command to attach the external file to the form.

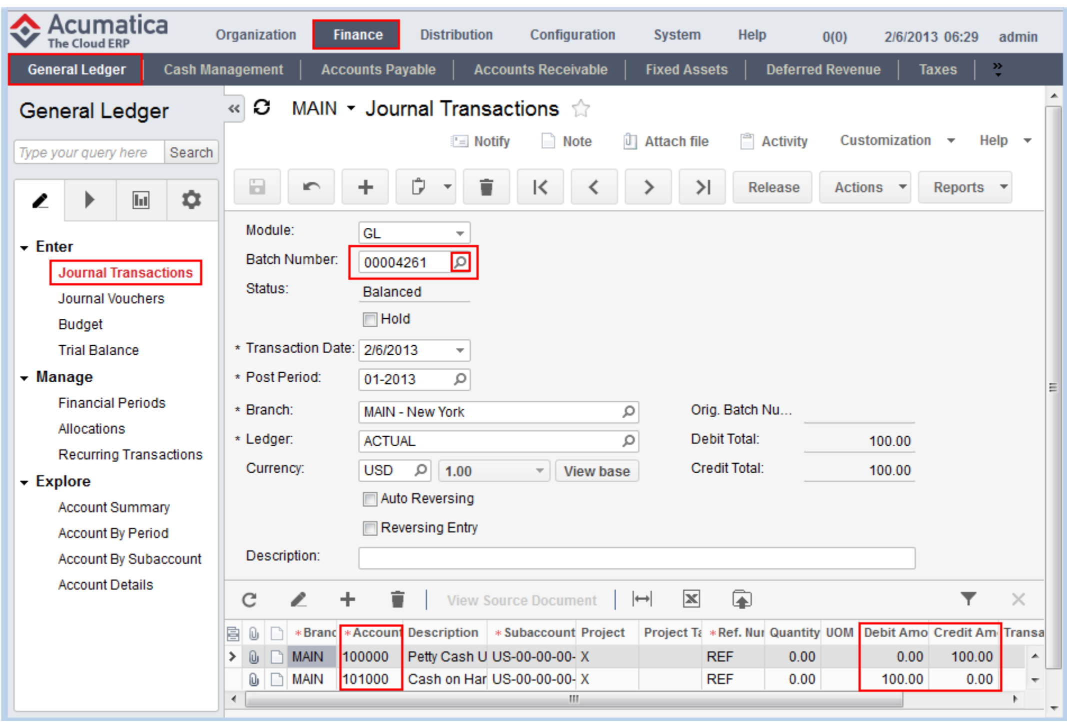


Figure: Exploring the Journal Transactions form

Test the results of data importing as follows:

- After preparing the code, build the solution and then press F5 to run the application in debug mode. Start the Acumatica ERP application instance with the WSDL file, and navigate to the **Finance** > **General Ledger** > **Enter** > **Journal Transactions** form. In the **Batch Number** lookup field, find and select the largest batch number, and note the transaction values of the two transactions in the details table (these values must equal those used in the code lines), as shown in the figure above.

- To see the attached file, click **Attach file** on the title bar and select the attached file name. (The figure below illustrates the process of opening the attached file.)

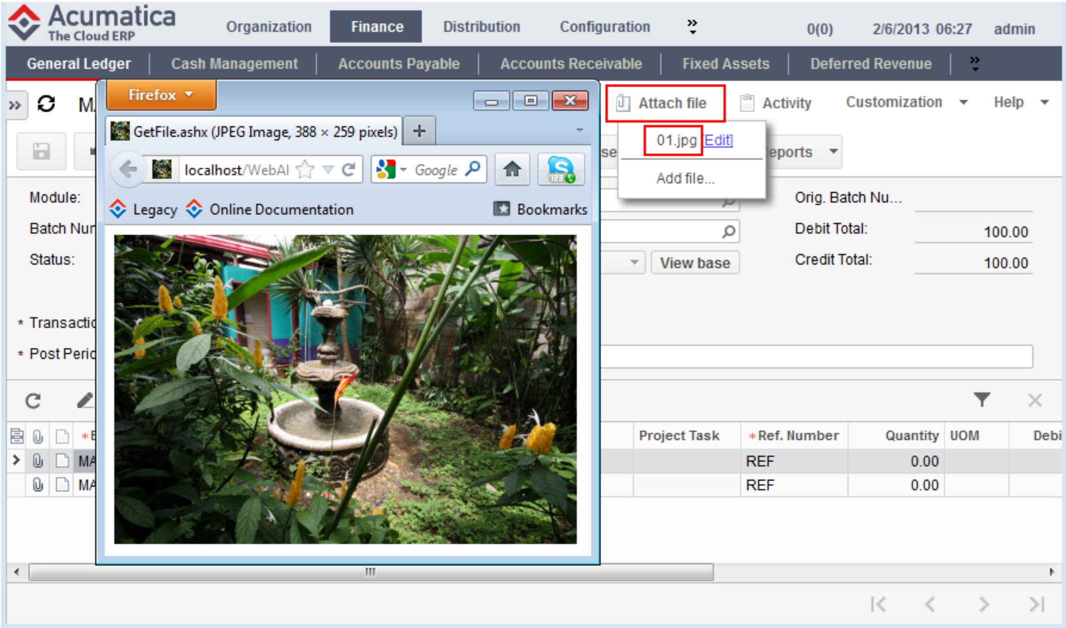


Figure: Opening the attached file

Exporting of Data With an Image From the Journal Transactions Form

In this example, you create, run, and test a client application that exports data with an image from the *Journal Transactions* (GL.30.10.00) form of the General Ledger module to a string array and limits exported data with filter conditions.

Before performing the actions of this example, import data with an image, as described in the previous example (see *Importing of Data With an Image Into the Journal Transactions Form*).

We make the following assumptions in this example:

1. You have installed the local client application instance (named *WEBAPIVirtual*) with the standard ERP demo application database. If you will use another application instance name, you should correct appropriate code lines in the code example shown in the next section.
2. You have created the Web Services WSDL definition file. (See *Quick Start, Step 1*.)
3. You have imported the Web Services WSDL definition file and generated the proxy class in the *ConsoleApplication.apitest* namespace. (See *Quick Start, Step 2*.) If you will use another WDSL file name, location, or namespace, you should correct appropriate code lines in the code example shown in the next section. You should also add your own password if it is different from the one used in the authorization code line in the code example. (See *Quick Start, Step 3*.)
4. You have primary information about the objects and properties of the Web Services API that the code lines of the example use. See the brief definitions in *Examples of the Web Service API Implementation*.

Create, Correct, and Run the Code Example

Add the code lines to the *Program* proxy class code and add the *using* operator, as shown in the code below. (The added code lines are preceded by +.)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
+using ConsoleApplication.apitest;

 namespace ConsoleApplication
 {
  class Program
  {
     static void Main(string[] args)
     {
+        apitest.Screen context = new apitest.Screen();
+        context.CookieContainer = new System.Net.CookieContainer();
+        context.AllowAutoRedirect = true;
+        context.EnableDecompression = true;
+        context.Timeout = 1000000;
+        context.Url = "http://localhost/WebAPIVirtual/Soap/APITEST.asmx";
+        LoginResult result = context.Login("admin", "E618");

+        GL301000Content GL301000 = context.GL301000GetSchema();
+        context.GL301000Clear();
+        string[][] export = context.GL301000Export
+        (
+           new Command[]
+            {
+            new Value
+             {
+               Value = "GL", LinkedCommand = GL301000.BatchSummary.Module },
+               GL301000.BatchSummary.ServiceCommands.EveryBatchNumber,
+               new Field
+               {
+                  ObjectName = GL301000.BatchSummary.BatchNumber.ObjectName,
-                        FieldName = "LastModifiedDateTime", Value = "TS"
+               }
+              GL301000.BatchSummary.BatchNumber,
+              GL301000.BatchSummary.ControlTotal,
+              new Value {
+                         FieldName = "01.jpg", LinkedCommand =
+                         GL301000.BatchSummary.ServiceCommands.Attachment
+                        },
+              GL301000.TransactionDetails.Account,
+              GL301000.TransactionDetails.Subaccount,
+              GL301000.TransactionDetails.RefNumber,
+              GL301000.Transactionetails.CreditAmount,
+              GL301000.TransactionDetails.DebitAmount
+            },
+            new Filter[]
+            { new Filter { Field = GL301000.BatchSummary.TransactionDate,
+              Condition = FilterCondition.GreaterOrEqual, Value = DateTime.Today }
+             },
+             0, true, true);
+       );
     }
   }
 }
```

This code implements the following process flow:

1. Using the *Export* method to export data from the form to the string array.
2. Using the **Filter** object with the *FilterCondition* property to filter exported data to the string array. (This exports only transactions from the current day.)
3. Using the *Attachment* service command to identify and download the attached file from the form.

Test the results of data exporting as follows:

- After preparing the code, build the solution and then press F5 to run the application in debug mode. Start the Acumatica ERP application instance with the WSDL file, navigate to the **Finance** > **General Ledger** > **Enter** > **Journal Transactions** form. In the **Batch Number** lookup field, find and select the largest batch number, change the **Transaction Date** field value to the current date value (if necessary), and note the transaction values of the two transactions in the details table (which must equal the values that will be obtained in the watch window of Visual Studio), as shown in the figure above. Compare the transaction values with the debugging results, as shown in the figure below.

- In Visual Studio, set appropriate breakpoints and then press F5 to run the client application in *Debug* mode. Use step-by-step debugging to ensure that the array contains exported data with the attached image file code. (The figure below illustrates the test results.)

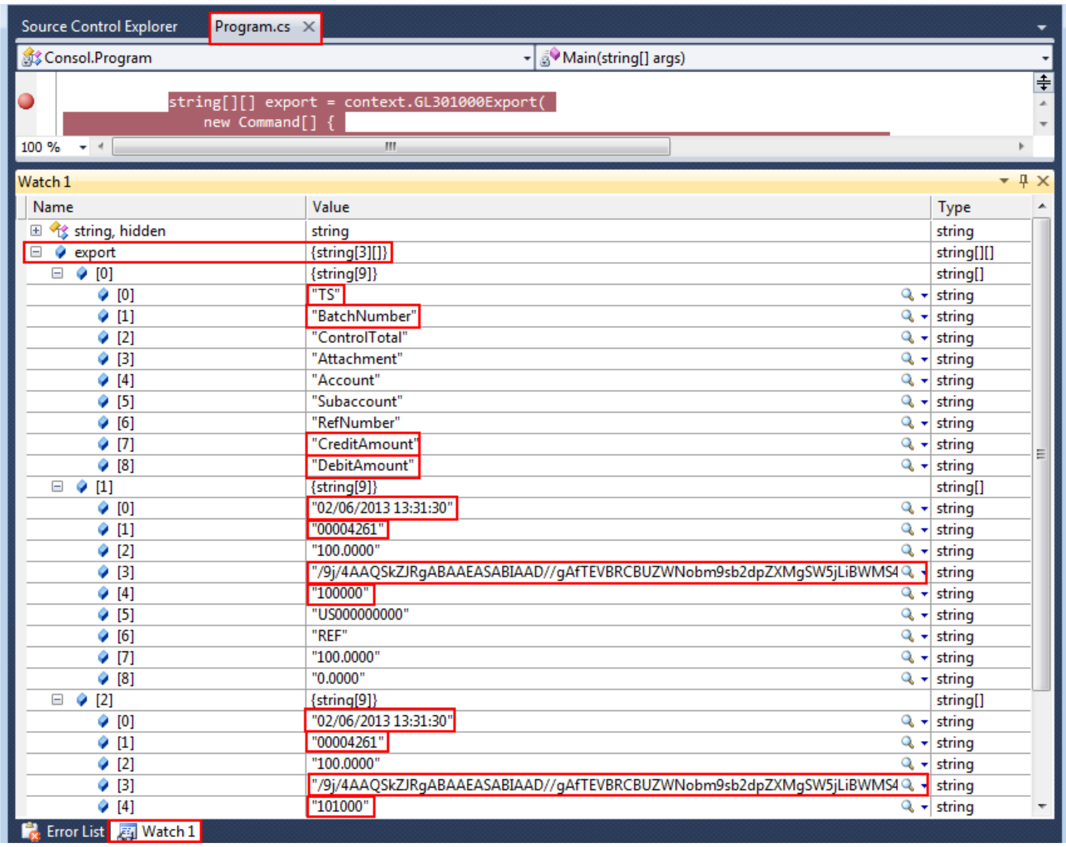


Figure: Checking the results in debug mode