

Contents

Copyright.....	9
About the Guide.....	10
Introduction.....	11
Acumatica Customization Platform.....	12
Customization Project.....	12
Types of Items in a Customization Project.....	13
Deployment of Customization.....	17
Simultaneous Use of Multiple Customizations.....	17
Customization of a Multi-Company Site.....	18
Customization Tools.....	19
Customization Projects Form.....	20
Customization Menu.....	21
Element Inspector.....	23
Customization Project Editor.....	25
Source Code Browser.....	51
Customization Framework.....	53
Changes in Webpages (ASPX).....	54
Changes in the Application Code (C#).....	55
Changes in the Database Schema.....	80
Custom Processes During Publication of a Customization.....	81
Performing Customization.....	83
To Assign the Customizer Role to a User Account.....	83
User Access Rights for Customization.....	84
To Detect Whether a Customization Project Is Applied to the Application.....	85
Exploring the Source Code.....	86
To Explore the C# Code of a BLC.....	87
To Explore the C# Code of a DAC.....	88
To Explore the ASPX Code of a Page.....	88
To Find a Customization of the ASPX Code.....	89
To Find Source Code by a Fragment.....	91
To Develop a Customization Project.....	92
To Perform Final Testing of a Customization.....	93
To Deploy a Customization Project.....	93
Managing Customization Projects.....	95
To Create a New Project.....	96
To Select an Existing Project.....	98
To Open a Project.....	99

To Update a Project.....	99
To Delete a Project.....	100
To Export a Project.....	101
To Import a Project.....	103
To Replace the Content of a Project from a Package.....	104
To Merge Multiple Projects.....	106
To Manipulate Customization Projects from the Code.....	107
GetPackage() Method.....	108
PublishPackages() Method.....	109
UnpublishAllPackages() Method.....	110
UploadPackage() Method.....	111
Publishing Customization Projects.....	112
To Prepare a Project for Publication.....	112
To Publish a Single Project.....	113
To Publish Multiple Projects.....	113
Performing the Publication Process.....	114
Validating Customization Code.....	115
To Publish the Current Project.....	115
To Publish the Current Project with a Cleanup Operation.....	115
To Publish a Customization for a Multi-Company Site.....	116
To View a Published Customization.....	117
To Unpublish a Customization.....	118
Unpublishing Customization Projects.....	119
Managing Items in a Project.....	120
Customized Screens.....	121
To Add a Page Item for an Existing Form.....	121
To Delete a Page Item from a Project.....	124
To Add a New Custom Form to a Project.....	124
To Delete a Custom Form from a Project.....	125
To Delete Items from the Project on the Edit Project Items Page.....	125
Customized Data Classes.....	126
To Add a DAC Item for an Existing Data Access Class to a Project.....	127
To Delete a DAC Item from a Project.....	129
To Convert a DAC Item to a Code Item.....	130
To Upgrade Technology for Legacy DAC Customization.....	131
Code.....	134
To Create a Custom Business Logic Controller.....	136
To Create a Custom Data Access Class.....	136
To Customize an Existing Business Logic Controller.....	138
To Customize an Existing Data Access Class.....	141
To Add Custom Code to a Project.....	142
To Add a Customization Plug-In to a Project.....	143
To Delete a Code Item From a Project.....	144
To Move a Code Item to the Extension Library.....	144

Custom Files.....	145
To Add a Custom File to a Project.....	145
To Update a File Item in a Project.....	146
To Delete a Custom File From a Project.....	149
Generic Inquiries.....	149
To Add a Generic Inquiry to a Project.....	149
To Delete a Generic Inquiry from a Project.....	151
To Update Generic Inquiry Items in a Project.....	151
To Redirect to the Generic Inquiry Form.....	152
Custom Reports.....	153
To Add a Custom Report to a Project.....	154
To Delete a Custom Report from a Project.....	155
To Update a Custom Report in a Project.....	155
Site Map.....	156
To Add a Site Map Node to a Project.....	156
To Delete a Site Map Item from a Project.....	158
To Update a Site Map Node in a Project.....	158
To Redirect to the Site Map Form.....	158
Database Scripts.....	158
To Add a Custom Table to a Project.....	159
To Update Custom Tables in the Project.....	160
To Add a Custom SQL Script to a Project.....	161
To Edit a Custom SQL Script.....	166
To Delete an Sql Item From a Project.....	166
System Locales.....	167
To Add a System Locale to a Project.....	167
To Delete a System Locale from a Project.....	169
To Update a Custom System Locale in a Project.....	169
To Redirect to the System Locales Form.....	170
Import and Export Scenarios.....	170
To Add an Integration Scenario to a Project.....	171
To Delete an Integration Scenario from a Project.....	172
To Update an Integration Scenario in a Project.....	172
To Redirect to the Import Scenarios Form.....	173
Shared Filters.....	173
To Add a Shared Filter to a Project.....	174
To Delete a Shared Filter from a Project.....	175
To Update a Shared Filter in a Project.....	175
To Redirect to the Filters Form.....	176
Access Rights.....	176
To Add Access Rights to a Project.....	177
To Delete Access Rights from a Project.....	178
To Update Access Rights in a Project.....	178
To Redirect to the Access Rights by Screen Form.....	179
Wikis.....	179

To Add a Custom Wiki to a Project.....	180
To Delete a Custom Wiki from a Project.....	181
To Update a Custom Wiki in a Project.....	181
To Redirect to the Wiki Form.....	182
Web Service Endpoints.....	182
To Add a Custom Web Service Endpoint to a Project.....	183
To Delete a Custom Web Service Endpoint from a Project.....	184
To Update a Custom Web Service Endpoint in a Project.....	184
To Redirect to the Web Service Endpoints Form.....	184
Analytical Reports.....	185
To Add a Custom Analytical Report to a Project.....	186
To Delete a Custom Analytical Report from a Project.....	187
To Update a Custom Analytical Report in a Project.....	187
To Redirect to the Report Definitions Form.....	188
Customizing Elements of the User Interface.....	189
Custom Form.....	192
To Develop a Custom Form.....	193
To Create a Custom Form Template.....	194
To Delete a Custom Form from a Project.....	197
Existing Form.....	197
To Start a Customization of a Form.....	198
To Delete a Customization of a Form.....	199
To Add a Form Container.....	199
To Add a Grid Container.....	200
To Add a Tab Container.....	200
To Add a Dialog Box.....	203
To Delete a Container.....	204
Form Container (PXFormView).....	205
To Open a Container in the Layout Editor.....	207
To Set a Container Property.....	207
To Add a Nested Container.....	211
To Add a Box for a Data Field.....	212
To Add a Layout Rule.....	214
To Add Another Supported Control.....	216
To Reorder Child UI Elements.....	217
To Delete a Child UI Element.....	218
Grid Container (PXGrid).....	219
To Add a Column for a Data Field.....	221
To Add a Control to the Form View of a Grid.....	224
Tab Container (PXTab).....	226
Tab Item Container (PXTabItem).....	227
To Conditionally Hide a Tab Item.....	228
Dialog Box (PXSmartPanel).....	228
To Open a Smart Panel in the Layout Editor.....	229
Box (Control for a Data Field).....	230

To Select a Box in the Layout Editor.....	231
To Set a Box Property.....	232
To Change the Type of a Box.....	233
Layout Rule (PXLayoutRule).....	235
To Select a Layout Rule in the Layout Editor.....	238
To Set a Layout Rule Property.....	239
Panel (PXPanel).....	247
Group Box (PXGroupBox).....	248
To Open a Group Box in the Layout Editor.....	248
To Create a Group Box for a Drop-Down Field.....	249
To Set a Group Box Property.....	250
Label (PXLabel).....	252
Radio Button (PXRadioButton).....	252
To Bind a Radio Button to a Value in the List of a Data Field.....	253
Button (PXButton).....	253
To Use a Button in a Dialog Box.....	254
To Use a Button to Invoke a Method.....	255
Java Script (PXJavaScript).....	257
Toolbars, Action Buttons, and Menus.....	258
Other Control Types.....	258

Customizing Business Logic..... 259

Data Access Class.....	259
To Start the Customization of a Data Access Class.....	259
To Add a Custom Data Field.....	261
To Create a New DAC.....	263
To Create a DAC Extension.....	265
Data Field.....	266
To Customize a Field on the DAC Level.....	267
To Customize a Field on the Graph Level.....	269
To Set a Default Value.....	269
To Change the Label of a Field.....	271
To Make a Field Mandatory.....	272
To Customize the Table of a Selector Field.....	274
To Add an Event Handler for a Field.....	275
To Provide Multi-Language Support for a Field.....	276
Graph.....	278
To Start the Customization of a Graph.....	279
To Create a Custom Graph.....	282
To Add a New Member.....	282
To Add an Action.....	283
To Add an Event Handler.....	284
To Override an Event Handler.....	286
To Override a Virtual Method.....	288
Data View.....	291
To Override a Data View.....	292

To Add a Data View Delegate.....	292
To Override a Data View Delegate.....	293
Action.....	294
To Start the Customization of an Action.....	294
To Override an Action Delegate Method.....	296
To Rename an Action Button.....	296
To Disable or Enable an Action.....	298
To Hide or Show an Action.....	298
Customizing the Database Schema.....	301
To Create a Custom Table.....	301
To Create a Custom Column in an Existing Table.....	303
To Create an Extension Table.....	303
Requirements for an Extension Table Schema.....	303
DAC Extension Mapped to an Extension Table.....	304
To Add a Custom SQL Script to a Customization Project.....	310
Integrating the Project Editor with Microsoft Visual Studio.....	312
To Work with a Code Item.....	312
To Work with Data Access Classes.....	313
To Debug the Customization Code.....	313
To Synchronize Code Changes with the Customization Project.....	314
Integrating the Project Editor with a Version Control System.....	315
To Save a Project to a Local Folder.....	316
To Update the Content of a Project from a Local Folder.....	318
To Configure a Connection String.....	319
To Integrate the Customization Project Editor with TFS.....	320
To Integrate the Customization Project Editor with Git.....	321
Troubleshooting Customization.....	322
To Discover the Method That Has Thrown an Exception.....	322
To Write to the Trace Log from the Code.....	324
To Log All Exceptions to a File.....	326
To Debug the Customization Code.....	326
To Validate a BQL Statement.....	327
To Measure the Execution Time of a BQL Statement.....	330
To Discover the Cause of Performance Degradation.....	334
To Force the Platform to Execute Database Scripts.....	338
To Resolve Issues While Upgrading a Customized Website.....	338
To Validate the Compatibility of the Published Customization with a New Version Before an Upgrade.....	339
To Resolve an Issue Discovered During the Validation.....	341
To Use the Technical Release Notes to Find the Breaking Changes.....	343
To Use an Ignore List for the Validation Errors.....	345
Examples.....	346
Examples of User Interface Customization.....	346

Dragging, Moving, and Deleting UI Controls and Grid Columns.....	346
Adding Input Controls.....	354
Adding Advanced Controls.....	358
Adding Columns to a Grid.....	365
Modifying Columns in a Selector.....	368
Adding PXLayouRule Components.....	373
Examples of Functional Customization.....	389
Adding Data Fields.....	390
Customizing DAC Attributes.....	400
Modifying a BLC Action.....	404
Modifying a BLC Data View.....	408
Declaring or Altering a BLC Data View Delegate.....	411
Extending BLC Initialization.....	415
Altering the BLC of a Processing Form.....	419
Adding or Altering BLC Event Handlers.....	423
Altering BLC Virtual Methods.....	433
Appendix.....	440
Reports.....	440
Report Form.....	440
Report.....	444
Form Toolbar.....	445
Table Toolbar.....	447
Glossary.....	450

Copyright

© 2017 Acumatica, Inc.
ALL RIGHTS RESERVED.

No part of this document may be reproduced, copied, or transmitted without the express prior consent of Acumatica, Inc.

11235 SE 6th, Suite 140
Bellevue, WA 98004

Restricted Rights

The product is provided with restricted rights. Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in the applicable License and Services Agreement and in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (c)(2) of the Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable.

Disclaimer

Acumatica, Inc. makes no representations or warranties with respect to the contents or use of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Acumatica, Inc. reserves the right to revise this document and make changes in its content at any time, without obligation to notify any person or entity of such revisions or changes.

Trademarks

Acumatica is a registered trademark of Acumatica, Inc. HubSpot is a registered trademark of HubSpot, Inc. Microsoft Exchange and Microsoft Exchange Server are registered trademarks of Microsoft Corporation. All other product names and services herein are trademarks or service marks of their respective companies.

Software Version: 6.1

Last updated: July 20, 2017

About the Guide

This guide describes the scope of the Acumatica Customization Platform and provides guidelines on how to use the platform capabilities for the customization of Acumatica ERP.

The guide is intended to answer the following questions:

- How does the Acumatica Customization Platform work? ([Acumatica Customization Platform](#))
- How to start, develop, test, and deploy a customization project? ([Performing Customization](#))
- How to create, open, update, delete, publish, unpublish, export, and import a customization project? ([Managing Customization Projects](#))
- How to create, add, update, and delete each type of item (such as screens, data access classes, code, files, and reports) in a customization project? ([Managing Items in a Project](#))
- How to publish a single customization project or multiple project for a single company or multi-company site? ([Publishing Customization Projects](#))
- How to create a custom form and customize the look and behavior of an existing form of Acumatica ERP? ([Customizing Elements of the User Interface](#))
- How to create custom code or extensions for an existing data access class or business logic controller? ([Customizing Business Logic](#))
- How to customize an existing table or create a custom table in the database? ([Customizing the Database Schema](#))
- How to use Microsoft Visual Studio to develop the customization code? ([Integrating the Project Editor with Microsoft Visual Studio](#))
- How to integrate with TFS, Git, or another version control system? ([Integrating the Project Editor with a Version Control System](#))
- What can I do to solve an issue that occurs while I am developing or applying a customization? ([Troubleshooting Customization](#))
- Which examples can I use to enhance my understanding of the customization tasks I will perform? ([Examples](#))

Introduction

You can change the user interface and business logic of Acumatica ERP, as well as build custom application modules that can be added to the system. The following diagram illustrates the different types of changes that you can make to the system within the scope of the customization process.

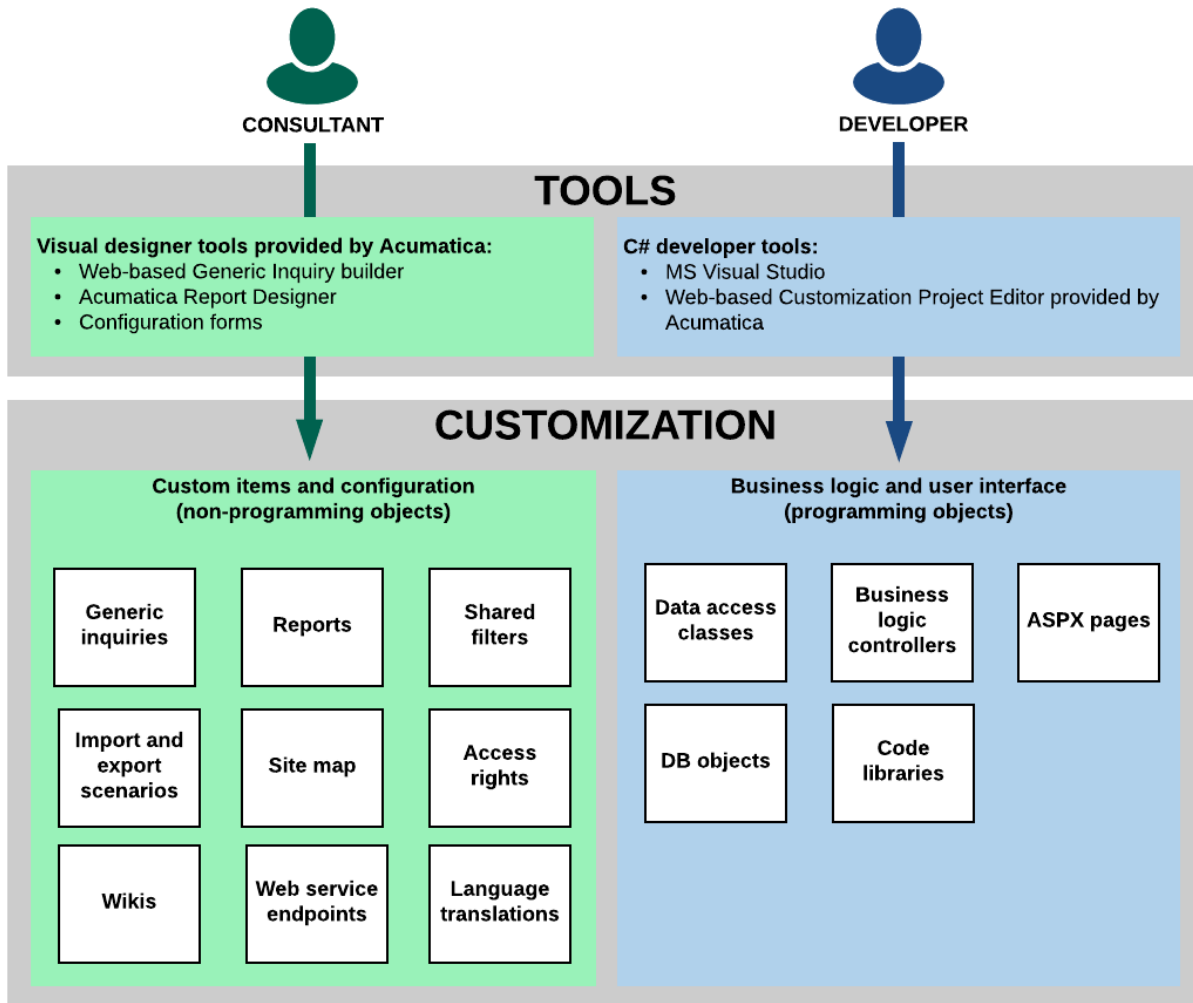


Figure: Customization of Acumatica ERP

As a value-added reseller (VAR), you can deliver end-user customization that might be very specific to each particular customer. At this level, you might want to add custom reports and data filters, and configure generic inquiries for users. These changes do not involve programming unless you also want to modify the business logic and the user interface of the application. For more information on the web-based Generic Inquiry builder and the Acumatica Report Designer, see [Managing Generic Inquiries](#) and [Report Designer](#).

As an independent software vendor (ISV), you can develop vertical solutions and add-ons to the core functionality of the system. At this level, you might want to modify the business logic and the UI of the application, which you can do by using the [Acumatica Customization Platform](#). To develop custom application modules, you can use the API that is provided by the Acumatica Framework.

As an original equipment manufacturer (OEM), you can build your own cloud ERP products based on high-level application objects of Acumatica ERP and the underlying Acumatica Framework technology. This type of customization may involve intensive changes at all levels of the system: modifications of the business logic and UI, development of custom modules, and report building.

Acumatica Customization Platform

To customize Acumatica ERP, you can use the following parts of the Acumatica Customization Platform:

- The web-based [Customization Tools](#) to customize the UI and business logic and to gather the changes into a distributable package that can be deployed to and applied on a target system
- The [Customization Framework](#) to develop customization code that changes the business logic of Acumatica ERP

Since an instance of Acumatica ERP consists of the website and the database, you can use the platform to customize both of these components. The website customization can include custom DLLs, custom and modified ASPX files, and files with custom C# code that modifies the UI and business logic of the product. The database customization can include changes in both the data and the schema of the database.

This part describes in detail the technologies implemented in the Acumatica Customization Platform.

In This Part

- [Customization Project](#)
- [Customization Tools](#)
- [Customization Framework](#)

Customization Project

When you use the tools provided by the Acumatica Customization Platform, the platform uses a customization project as a container that holds each change you make during the customization.

An Acumatica Customization Project is a set of changes to the user interface, configuration data, and functionality of Acumatica ERP. As the following diagram shows, a customization project might include any of the following:

- New custom forms and modifications to the existing forms of Acumatica ERP
- Custom C# code
- Custom database scripts
- Custom or modified reports (Acumatica Report Designer reports, generic inquiries, and analytical reports)
- Changes in the application configuration (site map changes, new system locales, integration scenarios, shared reusable filters, access rights of roles to forms, changes of wikis) that are saved in the database for the current company
- Additional files that you need for Acumatica ERP customization

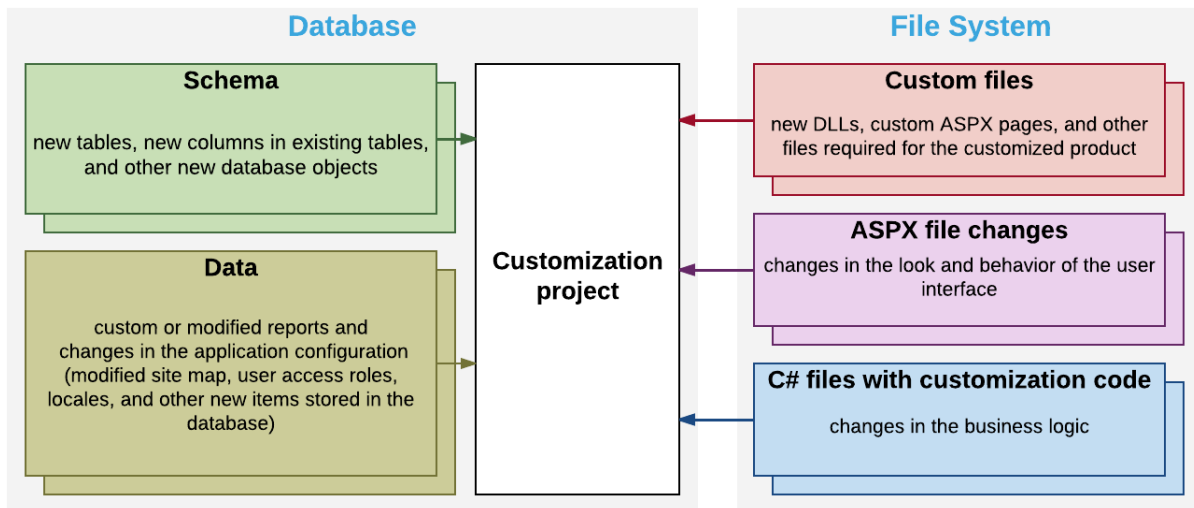


Figure: Content of a customization project

You develop and maintain customization projects by using the tools of the Acumatica Customization Platform (see [Customization Tools](#) for details). This platform provides the mechanisms to develop, upgrade, publish, unpublish (that is, cancel publication), export, and import a customization project. The content of a customization project is stored in the database of the Acumatica ERP instance.

To perform any customization of the UI or to extend the business logic, you have to create a new customization project or modify an existing one. (See [To Create a New Project](#) and [To Select an Existing Project](#) for details.)

Once you have selected a customization project for development, the customizations you perform will be added to this project. The customization project holds each change you make during the customization; however before the project is published, the changes exist only in the project and are not yet applied to the product. To apply the content of a customization project to Acumatica ERP, you have to publish the project.

When you need to apply the developed customization to a target environment, you should add all the changes and additional files to the customization project and export the project as a deployment package—a complete redistributable customization package. Then in the target environment, you import the package and publish the project.

You can create as many customization projects as you need and independently develop and maintain each customization project for a specific customization task.

In This Chapter

- [Types of Items in a Customization Project](#)
- [Deployment of Customization](#)
- [Simultaneous Use of Multiple Customizations](#)
- [Customization of a Multi-Company Site](#)

Types of Items in a Customization Project

When you customize an instance of Acumatica ERP by using the [Customization Project Editor](#), the platform keeps all items of a customization project as records in the `CustObject` table of the database. Each record in this table keeps the data of an item, and includes the XML code of the item in a special field. When you add an item to a customization project, the platform adds the new record to the table, creates the XML code of the item, and stores the code within the `Content` field of the record.



: You can view the content of an item of a customization project by using the [Item XML Editor](#) of the Customization Project Editor, as the following screenshot shows.

Edit Project Items

Object Name	Type	Descriptio	Exclui	Creat By	Creation Date	Last Modified By	Last Modified On
Pages\AR\AR409000.aspx.cs	File		<input type="checkbox"/>	admin	6/23/2016	admin	6/23/2016
Active Contracts	GenericInquiryScr...		<input type="checkbox"/>	admin	12/22/2015	admin	6/23/2016
~/pages/ar/ar303000.aspx	Page		<input type="checkbox"/>	admin	11/6/2015	admin	6/23/2016
~/pages/ar/ar409000.aspx	Page		<input type="checkbox"/>	admin	6/23/2016	admin	6/23/2016
~/pages/so/so301000.aspx	Page		<input type="checkbox"/>	admin	6/23/2016	admin	6/23/2016
~/pages/so/so303000.aspx	Page		<input type="checkbox"/>	admin	12/22/2015	admin	6/23/2016
AR650660.RPX	Report		<input type="checkbox"/>	admin	12/22/2015	admin	12/22/2015
Subscription Billing Details	SiteMapNode		<input type="checkbox"/>	admin	12/22/2015	admin	12/22/2015
Active Contracts	SiteMapNode		<input type="checkbox"/>	admin	12/22/2015	admin	12/22/2015
Subscription Usage Details	SiteMapNode		<input type="checkbox"/>	admin	6/23/2016	admin	6/23/2016
ARTran	Table		<input type="checkbox"/>	admin	12/22/2015	admin	12/22/2015
Contact	Table		<input type="checkbox"/>	admin	11/6/2015	admin	6/23/2016
Contract Usage	XportScenario		<input type="checkbox"/>	admin	12/22/2015	admin	6/23/2016

Source

```
<Page path="~/pages/ar/ar303000.aspx" ControlId="5" pageSource="7T3bctvGks8+VecfWnzKPtNwcc5iSnnVyItmxVdl"
  <PXFormView ID="DefContact" ParentId="phG_tab_Items#0_DefContact" TypeFullName="PX.Web.UI.PXFormView"
    <Children Key="Template">
      <AddItem>
        <PXDropDown TypeFullName="PX.Web.UI.PXDropDown">
          <Prop Key="Virtual:ApplyStyleSheetSkin" />
          <Prop Key="ID" Value="CstPXDropDown3" />
          <Prop Key="DataField" Value="UsrPersonalIDType" />
        </PXDropDown>
      </AddItem>
      <AddItem>
        <PXTextEdit TypeFullName="PX.Web.UI.PXTextEdit">
          <Prop Key="Virtual:ApplyStyleSheetSkin" />
          <Prop Key="ID" Value="CstPXTextEdit2" />
        </PXTextEdit>
      </AddItem>
    </Children>
  </PXFormView>
</Page>
```

Figure: Viewing the content of an item of a customization project in the Item XML Editor

Different item types have XML code that is structured differently. For example, if you create the `UsrPersonalID` bound custom field in the `CR.Contact` DAC (which is mapped to the `SOOrder` table) and the control for the field on the *Customers* (AR.30.30.00) form, the customization project might contain the following new objects:

- An item with the following XML code to add the column to the DB table.

```
<Table TableName="Contact">
  <Column TableName="Contact" ColumnName="UsrPersonalID" ColumnType="string"
    AllowNull="True" MaxLength="15" DecimalPrecision="2"
    DecimalLength="15" IsNewColumn="True" IsUnicode="True" />
</Table>
```

- An item with the following XML code to add the field to the DAC.

```
<DAC type="PX.Objects.CR.Contact">
  <Field FieldName="UsrPersonalID" TypeName="string" MapDbTable="Contact"
    TextAttributes="#CDATA" StorageName="AddColumn">
    <CDATA name="TextAttributes">
      <![CDATA[[PXDBString(15)]]><PXUIField(DisplayName="Personal ID")]]></CDATA>
    </Field>
</DAC>
```

- An item with the following XML code to add the control to the form.

```
<Page path="~/pages/ar/ar303000.aspx" ControlId="5"
  pageSource="...binary content of the page..."
  <PXFormView ID="DefContact" ParentId="phG_tab_Items#0_DefContact"
    TypeFullName="PX.Web.UI.PXFormView">
    <Children Key="Template">
```

```

<AddItem>
  <PXTextEdit TypeFullName="PX.Web.UI.PXTextEdit">
    <Prop Key="Virtual:ApplyStylesheetSkin" />
    <Prop Key="ID" Value="CstPXTextEdit2" />
    <Prop Key="DataField" Value="UsrPersonalID" />
    <Prop Key="CommitChanges" Value="True" />
  </PXTextEdit>
</AddItem>
</Children>
</PXFormView>
</Page>

```

Note that the items in these code blocks differ by type (`Table`, `DAC`, and `Page`) and structure. All the possible types of items in a customization project are described in the following table.

Item Types Within a Customization Project

Item Type (XML Tag, if differs)	Object	Description	Editing Form	Is Unpublished
<i>Page</i>	Custom form or changes to an existing form	For an existing form, layout change instructions that have to be applied by the platform to the ASPX code of the form during the project publication; for a custom form, the content of the form and path to the <code>.aspx</code> file of the form (the path is required for the system to detect changes of the file on the file system in the development environment and to deploy the file while publishing the project)	<i>Layout Editor</i>	Yes
<i>DAC</i>	Changes to an existing data access class	The data required to create the corresponding extension for the original data access class	<i>Data Class Editor</i>	Yes
<i>Table</i>	Changes to the schema of an existing database table	Description of custom columns added to a table for bound custom fields created in the appropriate DAC	-	-
<i>Code (Graph)</i>	Custom C# code	A custom DAC or BLC, an extension for an existing DAC or BLC, a customization plugin, or any custom class: the listed class types differ by the base class—correspondingly, <code>PX.Data.IBqlTable</code> , <code>PXGraph<></code> , <code>PXCacheExtension<></code> , <code>PXGraphExtension<></code> , <code>CustomizationPlugin</code> , and no or any base class for a custom class	<i>Code Editor</i>	Yes
<i>File</i>	Custom file	Path to a custom file and GUID of the file content in the file storage of the database (the path is relative to the website folder; new custom forms are added to the project as custom files)	<i>File Editor</i>	Yes

Item Type (XML Tag, if differs)	Object	Description	Editing Form	Is Unpublished
<i>GenericInquiryScreen</i>	Custom or customized generic inquiry	Data set of a custom or customized generic inquiry form	Generic Inquiry form (SM.20.80.00)	-
<i>Report</i>	Custom Acumatica Report Designer report	Data set of a custom report created by using the Acumatica Report Designer	Acumatica Report Designer	-
<i>SiteMapNode</i>	Custom or customized site map node	Data set of a custom or customized site map node; you should create a custom site map node for each custom form, generic inquiry, or report included in the customization project	Site Map form (SM.20.05.20)	-
<i>Sql</i>	Custom SQL script	Custom database table definition or custom SQL script that has to be executed while the customization project is published	SQL Script Editor	-
<i>Locale</i>	Custom locale	Data set of a custom system locale, which is a set of parameters that defines the language and other local preferences—such as how to display numbers, dates, and times in the user interface—for a group of users	System Locales form (SM.20.05.50)	-
<i>XportScenario</i>	Custom integration scenario	Data set of a custom export or import scenario used to perform data migration between a legacy application and Acumatica ERP	Import Scenarios form (SM.20.60.25) and Export Scenarios form (SM.20.70.25)	-
<i>SharedFilter</i>	Custom shared filter	Data set of a custom reusable shared filter created on a processing or inquiry form	Filters form (CS.20.90.10)	-
<i>ScreenWithRights</i>	Custom access rights to a form	Data set of the custom access rights of roles to a form, down to the control of form elements, such as buttons, text boxes, and check boxes	Access Rights by Screen form (SM.20.10.20)	-
<i>WikiArticle</i>	Custom wiki module	Data set of a custom wiki and all the articles created within this wiki	Wiki form (SM.20.20.05)	-
<i>EntityEndpoint</i>	Custom web service endpoint	Data set of a custom web service endpoint	Web Service Endpoints form (SM.20.70.60)	-
<i>ReportDefinition</i>	Custom analytical report	Data set of a custom analytical report, including the data of a predefined sets of rows, columns, and units	Report Definitions form (CS.20.60.00)	-

Deployment of Customization

Once you have finished a customization project, you can export the project as a deployment package that can be then imported and published as a customization on the end-user systems, as shown in the following diagram.

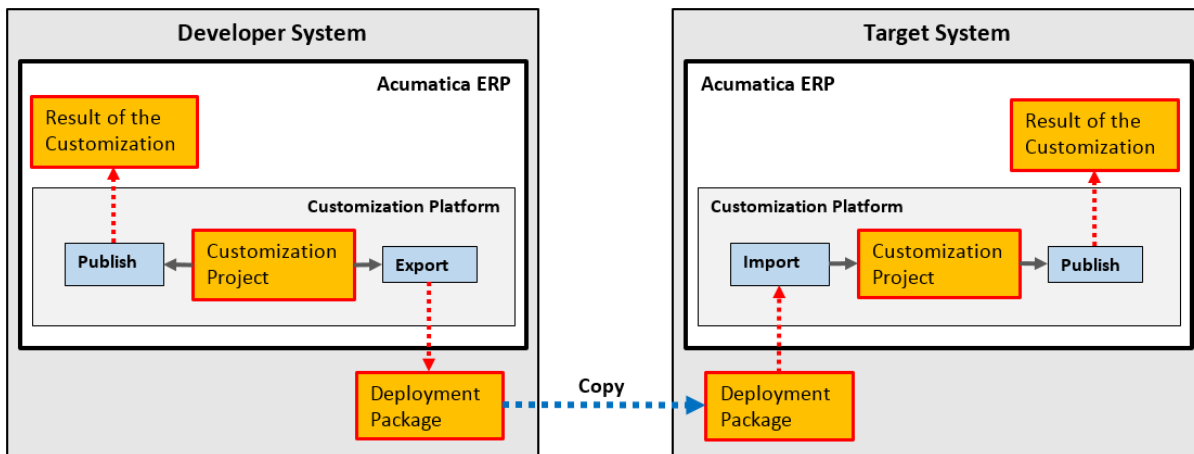


Figure: Deployment of customization to target systems

A deployment package is a redistributable .zip file that includes the full content of a customization project. A deployment package consists of the `project.xml` file and any custom files that you have added to the project, such as external assemblies and custom ASPX pages. You can manually edit the `project.xml` file in an XML Editor in the file system. However we recommend that you modify the project items in the easiest and most reliable way: by using the [Customization Project Editor](#).

When the project is finished, you can download the deployment package to deploy the customization to the target system (see [To Export a Project](#) for details). If you have finished the project, we recommend that you publish the project and test the customization before downloading the deployment package, to ensure that you have no issues. Also, you can download the package to have a backup copy of the customization project you are working on.

You can import a deployment package to work with the customization project or to publish the final customization on the target website (see [To Import a Project](#) for details).



: In MySQL, the maximum size of one packet that can be transmitted to or from a server is 4MB by default. If you use MySQL and want to manage a customization project with the size that is larger than the default maximum value, you have to increase the `max_allowed_packet` variable in the server. The largest possible packet size is 1GB.

Simultaneous Use of Multiple Customizations

With the Acumatica Customization Platform, you can simultaneously manage multiple customizations by using the [Customization Projects](#) (SM.20.45.05) form, also described in [Customization Projects Form](#) . You can publish multiple customization projects to an Acumatica ERP instance at once.

When you publish more than one customization project, the platform merges the content of all projects into a single customization project. If different projects include customization for the same object, the platform tries to merge the changes by using the following approach:

- If the changes can be merged, the platform merges them. For example, the platform can merge different properties of the same control in an ASPX page.
- If the changes cannot be merged, as with the same report being in different customization projects, the platform stops the process and displays an error message.

On the [Customization Projects](#) form, you can specify an optional number (level) for each customization project, assigning the highest number to the most important change. The level can be used to resolve conflicts that arise while you are publishing customization projects if multiple modifications of the same objects of an Acumatica ERP instance are merged. As a result, the customization from the project with the highest level is added to the merged project.

Multilevel customizations might be required when you develop an off-the-shelf customization solution that is distributed in multiple editions, or applications that extend the functionality of Acumatica ERP or other software based on Acumatica Framework in multiple markets. You may have a customization that contains a solution common to all markets as well as multiple market-specific customizations, each of which is deployed along with the base customization. Moreover, you can later apply a high-level project to customize deployed solutions for the end user.

However if you have multiple customizations of the same object and have no market requirements to keep multilevel customizations, we recommend that you merge the customizations into a single customization project, as described in [To Merge Multiple Projects](#).

Customization of a Multi-Company Site

The data of each company that uses the same instance of Acumatica ERP is isolated in the database.

Because a customization project is stored in the database, by default, the project data belongs to the company in which the project is created or imported. When the project is published, the customization applies to both the website files and the database. Because a customization project can contain different types of items, the platform uses a specific approach to apply each type of items to the website, database schema, and database data.

Initially the *Page*, *DAC*, *Table*, *SQL*, and *Code* items of a customization project are stored in the database for a single company. However, while the project is being published, the platform creates certain files for these items in the website folder or changes the database schema, if required. The new files and the changed database schema are available from other companies. As a result, the listed item types are applied for multiple companies.

During the publication of the project, for each *File* item, the platform creates the file in the file system, so *File* items are shared for multiple companies as well.

For example, suppose you have logged in to the *MyCompany* company and have a customization project that contains only items of the *Page*, *DAC*, *Table*, *Code*, and *File* types. After you have published the project, the customization is applied to each company in the instance of Acumatica ERP. But the customization project data is still available only in the *MyCompany* company.

However if you publish a customization project that contains items of other types (such as generic inquiries and access rights of roles to forms), the customization does not apply to the website files or to the database schema. This data is stored only in the database, and the application server uses the data from the database at run time. Because this data is company-specific, it is available in only the company where it was published. Moreover, for all companies that use the same site, on the Login form and in the **About Acumatica** box, you can see that the website is customized, but you won't see any published project on the [Customization Projects](#) (SM.20.45.05) form. This form, also described in [Customization Projects Form](#), displays the customization projects that have been uploaded to the current company; therefore, no projects are displayed if they have been uploaded and published under another company.

You can publish multiple customization projects for multiple companies at once. On the [Customization Projects](#) form, you can select the customization projects that you need to publish for multiple companies and use the **Publish to Multiple Companies** action to open the **Publish to Multiple Companies** dialog box. You then select the required companies and apply the selected customization projects to the selected companies. (See [To Publish a Customization for a Multi-Company Site](#) for details.)

The following table shows the differences in applying a customization to a multi-company website if the customization that has been published from another company is shared or not shared for your company.

Question	Customization published for a multi-company site from another company is shared for your company	Customization published for a multi-company site from another company is NOT shared for your company
Is the customization applied to the application instance (see To Detect Whether a Customization Project Is Applied to the Application for more information) for your company?	Yes	Yes
Do changes in the file system (<i>Page</i> , <i>DAC</i> , and <i>Code</i> items) exist for your company?	Yes	Yes
Do changes of the database schema (<i>Table</i> and <i>SQL</i> items) exist for your company?	Yes	Yes
Do custom reports and configuration (such items as <i>Report</i> , <i>SiteMapNode</i> , and <i>SharedFilter</i>) exist for your company?	Yes	No
Does the project list on the Customization Projects form contain the project? (That is, can you access the customization project data from your company?)	No	No
Is it necessary to import the customization package to access the customization project data from your company?	Yes	Yes
After the customization package is imported for your company and added to the list on the Customization Projects form, is the project displayed in the list as an already published one?	Yes	No

Customization Tools

Depending on the complexity of the particular customization task, you can employ any the tools implemented in the Acumatica Customization Platform.

For every type of customization, you can use the tools described in the following topics:

- [Customization Projects Form](#)
- [Customization Menu](#)
- [Element Inspector](#)
- [Customization Project Editor](#)
 - [Layout Editor](#)
 - [ASPX Editor](#)
 - [Data Class Editor](#)
 - [Code Editor](#)
 - [File Editor](#)
 - [SQL Script Editor](#)
 - [XML Editors](#)
 - [Project XML Editor](#)
 - [Item XML Editor](#)

- [Source Code Browser](#)

Customization Projects Form

The Acumatica Customization Platform stores the data of the customization projects that were created or imported in the database of the instance of Acumatica ERP.

You can manage customization projects on the [Customization Projects](#) (SM.20.45.05) form, which is shown in the following screenshot.

Published	Project Name	Level	Screen Names	Description	Created By	Last Modified On
<input type="checkbox"/>	EcommerceSept17		IN202500		admin	10/27/2016
<input type="checkbox"/>	HideSSN		CR303010		admin	7/18/2014
<input type="checkbox"/>	POstatus		PO301000		admin	7/18/2014
<input type="checkbox"/>	StockImage				admin	8/13/2014
<input type="checkbox"/>	StockItemImage		AR303000...		admin	10/27/2016
<input checked="" type="checkbox"/>	YogiFon		AR303000...		admin	11/3/2016

Figure: Viewing the Customization Projects form

On this form, you can add a new customization project, open a customization project for editing in [Customization Project Editor](#), publish any number of customization projects, cancel the publication of customization projects, export a customization project as the deployment package, import a customization project from an existing deployment package, and delete a customization project. (See [Managing Customization Projects](#) for instructions.)

Published Customization Page

The Published Customization page of the [Customization Projects](#) (SM.20.45.05) form shows the merged XML code of the customization projects that are currently published. On the Published Customization page, you can:

- View the code.
- Download the deployment package that contains the code.

The following screenshot shows the merged project, which contains customizations introduced by two different projects for the forms with the CR303010 and PO301000 identifiers.

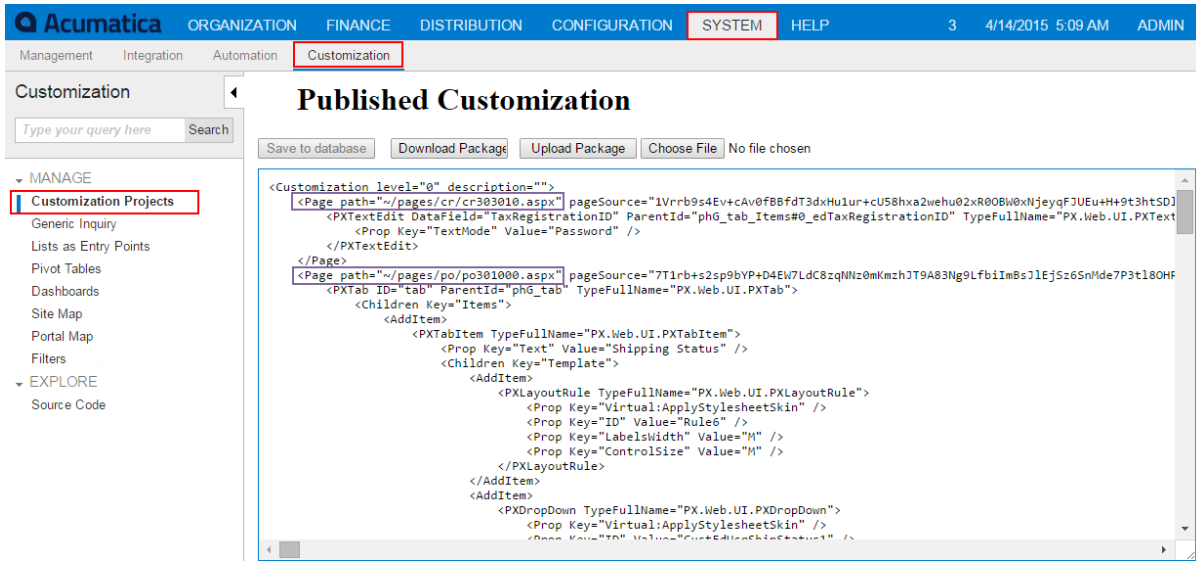


Figure: Viewing the merged XML code of the published customization projects

The Published Customization page includes a toolbar and a text area for viewing XML code. The text area displays the XML content of a merged customization project. This area is not used for editing the XML code.

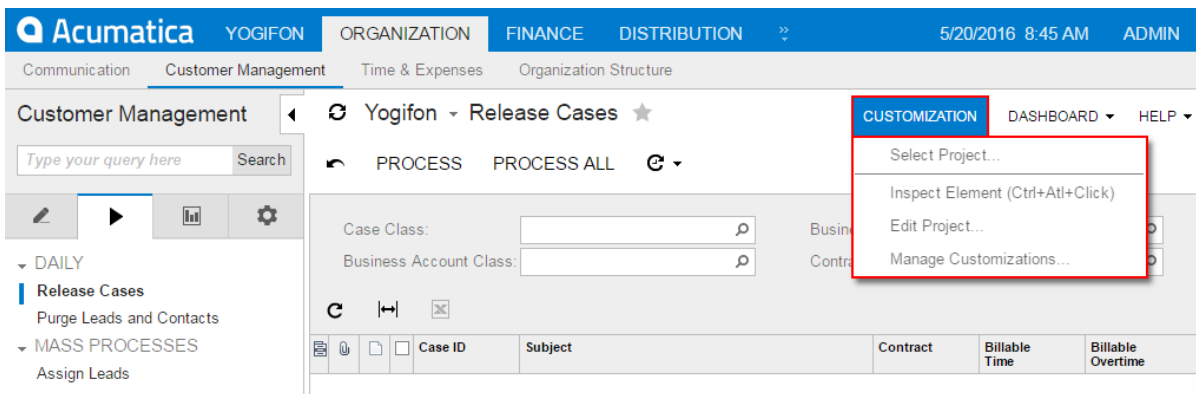
The toolbar buttons of this page are described in the following table.

Button	Description
Save to database	Is not used.
Download Package	Downloads the <code>Customization.zip</code> file, which includes the full content of the merged customization project. You can use this file as a joint deployment package to work with the customization project or to publish the final customization on the target website. See To Merge Multiple Projects for details.
Upload Package	Is not used.
Choose File	Is not used.


Customization Menu

You can access the **Customization** menu on an opened Acumatica ERP form if you have the *Customizer* role. (For details, see [To Assign the Customizer Role to a User Account](#).)

Click **Customization** on the form to access the **Customization** menu associated with the form, as shown in the following screenshot.



You can use the following customization-related menu commands.

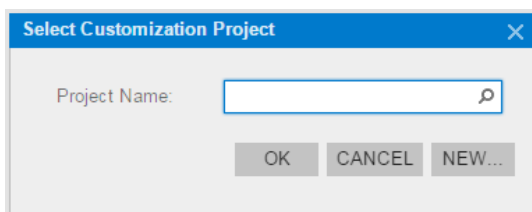
Command	Description
Select Project	Opens the Select Customization Project Dialog Box , which you use to select an existing customization project or to create a new project for all modifications that you are going to perform.
Inspect Element	<p>Launches the Element Inspector, which gives you an option to select a UI control on the current form and opens the Element Properties Dialog Box for the selected control. You use the dialog box to inspect and customize the control.</p> <p> : You can use the keyboard shortcut Ctrl+Alt+Click to inspect elements on pop-up windows and dialog boxes.</p> <p>If you have selected a customization project, all the modifications that you initiate in the inspector will be added to this current project. Otherwise, the inspector opens the Select Customization Project dialog box.</p>
Edit Project	Opens the Customization Project Editor for the currently selected customization project.
Manage Customizations	Opens the Customization Projects (SM.20.45.05) form.

Select Customization Project Dialog Box

You use the **Select Customization Project** dialog box to select an existing customization project or to create a new project.

You open the dialog box, shown in the following screenshot, in the following ways:

- From the **Customization** menu of a form—by selecting the **Select Project** command
- From the **Customization** menu of a form—by selecting the **Edit Project** command if there is no currently selected customization project
- From the [Element Properties Dialog Box](#), which you access by clicking the **Customize** button if there is no currently selected customization project



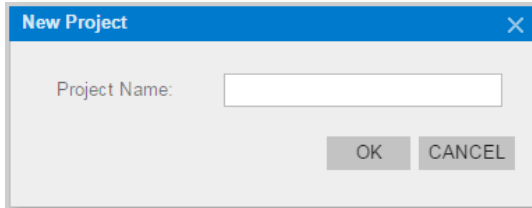
The dialog box contains the following UI controls.

Control	Description
Project Name	Provides the ability to select an existing customization project. The box contains a selector, which you can use to find an existing customization project by the name or by a part of the name.
OK	Confirms your selection and exits the dialog box.
Cancel	Cancels the operation and exits the dialog box.
New	Opens the New Project Dialog Box , where you can create a new project.



If you have selected a customization project, all customizations that you initiate will be added to this current project until you select another project or sign out.

New Project Dialog Box

You use the **New Project** dialog box, shown in the following screenshot, to create a new customization project.



The dialog box contains the following UI controls.


Control	Description
Project Name	Provides the ability to enter a name of the customization project.  : The customization project name is used as the <code>namespace</code> if you create an extension library from the project. The first character of the name must be a letter or the underscore symbol.
OK	Creates an empty customization project with the specified name and closes the dialog box.  : As soon as you click the OK button, the platform creates a new customization project in the database.
Cancel	Closes the dialog box.

Element Inspector

You can use the Element Inspector for the following purposes:

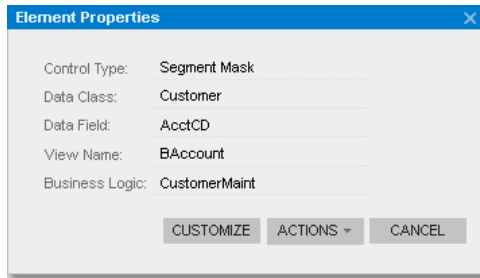
- To view the following reference information for any visual element on a form:
 - Control type
 - Data access class (if applied)
 - Data field (if applied)
 - Data view (if applied)
 - Business logic controller
 - Action name (if applied)
- To view the source code of the ASPX page that contains the UI control for the inspected element
- To view the source code of the data access class that provides data for the inspected element
- To view the source code of the business logic executed for the inspected element
- To start the customization of the inspected element

For a form, you can activate the Element Inspector from the [Customization Menu](#). If you need to activate the inspector for a pop-up panel, dialog box, or other UI element that opens in modal mode and makes the **Customization** menu unavailable for selection, you can press Control-Alt.

After the inspector is activated, the  cursor indicates that you can select a UI element to inspect. If you select an element, the Element Inspector opens the [Element Properties Dialog Box](#).

Element Properties Dialog Box


The Element Inspector opens the **Element Properties** dialog box, shown below, when you have selected a UI element to inspect.



In this dialog box, you can perform the following:

- Inspect properties of the UI element selected on the form
- With a single click, launch the [Layout Editor](#) for the form to customize the inspected element
- Select an action to do any of the following:
 - Launch the [Code Editor](#) to develop a graph extension for the business logic controller (BLC) bound to the form
 - Launch the [Data Class Editor](#) for the data access class (DAC) that contains the data field of the inspected element to customize the DAC
 - Open the [Source Code Browser](#) to view the following:
 - The ASPX code of the inspected page
 - The source code of the BLC bound to the form
 - The source code of the DAC that contains the data field of the inspected element

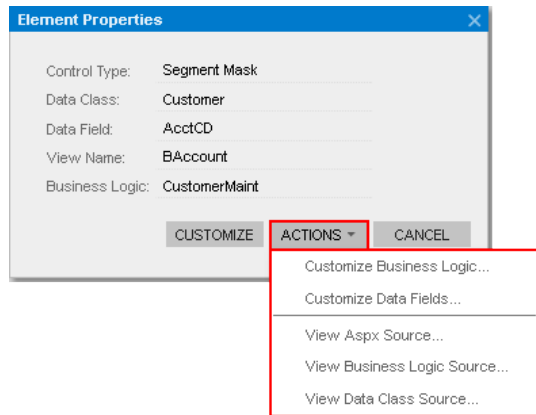
The **Element Properties** dialog box contains the following controls.

Control	Description
Control Type	The type of the inspected UI element.
Data Class	The name of the DAC to which the field for the inspected element belongs.
Data Field	The string value of the <code>DataField</code> property of the inspected UI element. (It corresponds to the name of the field in the DAC.)
View Name	The name of the data view that provides data for the inspected UI element.
Business Logic	The name of the BLC bound to the form.
Action Name	The name of the action of the inspected toolbar button.
Customize	<p>A button that launches the Customization Project Editor, which opens on the Layout Editor page for the form that contains the inspected element.</p> <p> : The successive use of the Element Inspector and the Customize button is the easiest way to change a property of a UI control on a form, because the Layout Editor opens with the Properties tab displayed for the element that is currently being inspected.</p>
Actions	Opens the Actions menu with commands to customize or revise the element source code using the Customization Project Editor (to customize) or the Source Code browser (to view).
Cancel	Cancels the operation and closes the dialog box.

The Actions Menu

The **Actions** menu, shown in the screenshot below, contains commands you can invoke to customize or revise the code of the following:

- The inspected element
- The current ASPX page on which the element is selected
- The DAC to which the field for the inspected element belongs
- The BLC (graph class) that is bound to the form



The menu contains the following commands.

Command	Description
Customize Business Logic	Creates a graph extension template for the BLC that is bound to the form, adds the template code to the customization project, and opens the Customization Project Editor on the Code Editor page, which loads the BLC extension template so you can edit it.
Customize Data Fields	Opens the Customization Project Editor on the Data Class Editor page in the Edit Attributes section so you can customize the inspected element attributes.
View ASPX Source	Opens the Source Code Browser on the Screen ASPX tab with the source code of the current form.
View Business Logic source	Opens the Source Code browser on the Business Logic tab with the source code of the BLC that is bound to the form.
View Data Class Source	Opens the Source Code browser on the Data Access tab with the source code of the DAC to which the field for the inspected element belongs.



: If you have opened the Customization Project Editor or the Source Code browser, you can access any source code of the website—not only the source code of the inspected element and the current form, DAC, and BLC.

Customization Project Editor


You can use the Customization Project Editor (Project Editor) to develop and manage the content of a customization project. The editor contains separate pages to add and manage items of the following types in the currently selected customization project:

- Screen (form)
- Data access class (DAC)

- Code (C#)
- File
- Generic inquiry
- Acumatica Report Designer report
- Site map
- Database script
- Translation (language locale)
- Integration scenario
- Shared reusable filter
- Access rights of roles to forms
- Wiki changes
- Analytical report

(See [Managing Items in a Project](#) for instructions.)

To help you work with screens, data access classes, code, custom files, and database scripts in the scope of a customization project, the Project Editor includes the following editors, which are displayed as its pages:

- [Layout Editor](#), which is the visual editor for the source code of an .aspx page
- [Data Class Editor](#), which is used to create, develop, manage, and view in XML format the content of extensions for data access classes
 -  : When the project is published, the Acumatica Customization Platform transforms the content of extensions for data access classes from XML format to C# code and saves the code in .cs files in the App_RuntimeCode folder of the website.
- [Code Editor](#), which is used to manage, develop, and view the customization code (C#) added to the project, including extensions for business logic controllers (BLCs) and DACs
- [File Editor](#), which is a text editor used to edit and review the content of text files included in the customization project
- [SQL Script Editor](#), which is used to add and edit custom SQL scripts and add custom tables to the customization project

Other types of items are custom data that can be added to the customization project from the database. For these items, the Project Editor provides you the capabilities to add to the project, delete from the project, and reload from the database. To create or edit these items, you can use dedicated forms of Acumatica ERP. For example, to create or edit a generic inquiry in the database, you can use the [Generic Inquiry](#) (SM.20.80.00) form.

Also, the Project Editor includes the following editors:

- [ASPX Editor](#), which is used to edit the ASPX code of a page customized by means of the Layout Editor
- [XML Editors](#), which are used to edit and review the XML code of the customization project ([Project XML Editor](#)) or a separate item of the project ([Item XML Editor](#))

You can launch the Customization Project Editor only if you have selected a customization project. You can launch the editor from any form of Acumatica ERP by using the [Customization Menu](#) or from the [Customization Projects](#) (SM.20.45.05) form, also described in [Customization Projects Form](#).

The Project Editor, shown in the following screenshot, looks like a regular webpage that consists of the following parts:

- The main menu for working with the customization project
- A navigation pane, which displays the list of pages used to manage the corresponding project items

- The main area, which displays the list of project items or provides a work area to edit these items


The screenshot shows the Acumatica interface with the 'Custom Files' view. The navigation pane on the left includes categories like SCREENS, DATA ACCESS, CODE, and Files (5). The main area displays a table of files with columns for Object Name, Description, Last Modified By, and Last Modified On.

Object Name	Description	Last Modified By	Last Modified On
App_Data\Mobile\includes\YF401000.xml.inc		admin	4/18/2016
App_Data\Mobile\YogiFonMSM.xml		admin	4/18/2016
Bin\YogiFon_Code.dll		admin	4/18/2016
Pages\YF\YF401000.aspx		admin	4/15/2016
Pages\YF\YF401000.aspx.cs		admin	4/15/2016

In the navigation pane, a node with capitalized name can be expanded to get direct access to items of the appropriate type.

The main menu contains the following items and commands.

Item	Command	Description
File	Manage Customization Projects	Opens the <i>Customization Projects</i> (SM.20.45.05) form.
	Edit Project XML	Opens the Project XML page, which you can use to edit the XML source code of the current customization project, save it to the database, download the project package, and upload a deployment package to replace the project content.
	Edit Project Items	Opens the Edit Project Items page, which you can use to edit the XML source code of a project item.
	Export Project Package	Exports the deployment package of the project—that is, the ZIP file that contains the project. The file has the same name as the exported customization project has.
	Replace from Package	Initiates the import of a previously exported deployment package from a ZIP file. Provides the Open Package dialog box with the Choose File button and the Upload button to replace the current project content.
Publish	Publish Current Project	Initiates the process of publishing the current customization project. Launches a publication process that opens the Compilation window to output a log with information about the process. If both the validation and compilation of the project are successful, the process makes the Publish button available. This button is used to finalize the publishing and to update the website.
	Multiple Projects	Opens the <i>Customization Projects</i> in a new window.
	Publish with Cleanup	Initiates the process of publishing the current customization project as the Publish Current Project command does,

Item	Command	Description
		but with the following difference: When the Acumatica Customization Platform publishes a project that contains a database script, the platform executes the script and tries to avoid executing the script at every publication of the project for optimization purposes. Therefore, the platform keeps information about each script that has been executed at least once and has not since been changed in the database, and omits the repeated execution of such scripts. If you run the Publish with Cleanup operation, the platform cleans all the information about previously executed scripts of the customization project and executes this scripts once more while publishing the project.
Extension Library	Create New	<p>Creates a solution for Microsoft Visual Studio in which you can develop an extension library for the customization project. The solution contains the website and extension library projects. This action also downloads the <code>OpenSolution.bat</code> file. The file contains the absolute path to the <code>.sln</code> file in the file system; you can use this file to open the solution in Visual Studio.</p> <p> : By default, the system uses the <code>App_Data\Projects</code> folder of the website as the parent folder for the solution project. If the website folder is outside of the <code>C:\Program Files (x86)</code> and <code>C:\Program Files</code> folders, such as <code>C:\AcumaticaSites\MySite</code>, we recommend that you not change it. Otherwise, we recommend that you specify a parent folder outside these folders to avoid an issue with permission to access files.</p>
	Bind to Existing	Specifies the extension library project in the file system to which the customization code will be moved if you click Move to Extension Lib on the toolbar of the Code Editor.
	Open in Visual Studio	Downloads the <code>OpenSolution.bat</code> file, which is used to open the existing solution in Visual Studio.
	Show Active Extensions	Starts the verification of extensions for data access classes and business logic controllers, and opens the Validate Extensions pop-up window to display the validation log. In the log, every error is highlighted in red. We recommend that you verify extensions if you have upgraded legacy DAC customization.
Source Control	Save Project to Folder	Saves the customization project as a set of files to a local folder that is used for integration with source control systems. Invoking this action opens the Saves Project to Folder dialog box (see To Save a Project to a Local Folder for instructions) so that you can select the name and location of the folder inside a repository.
	Open Project from Folder	Loads the customization project from the repository. (See To Update the Content of a Project from a Local Folder for instructions.)
	Setup Source Control	Opens the Source Control Setup dialog box, which you can use to specify a configuration string for connection to a version control system, if required. (See To Configure a Connection String for details.) For example, to control versions, you should set up the configuration string for the Team Foundation Server (TFS), but it is not needed for Git. (See To Integrate the Customization Project Editor with TFS for instructions.)

Layout Editor

The Layout Editor is the visual editor for the source code of an ASPX page. Use it to configure ASP.NET containers, such as forms and grids, as well as to specify the properties of UI controls.


By using the editor, you can perform such customizations of a form as the following:

- Add a custom form to Acumatica ERP
- Add a container to a form
- Add a custom field to a data access class (DAC)
- Add a control for a field to a container
- View and modify the properties of a control
- Change the order of controls in a container
- Customize the attributes of a field in a data access class
- View the modifications made to the original declaration of the form
- Immediately preview in the browser the changes made during the customization of the layout

During customization, you can open the Layout Editor in the following ways:

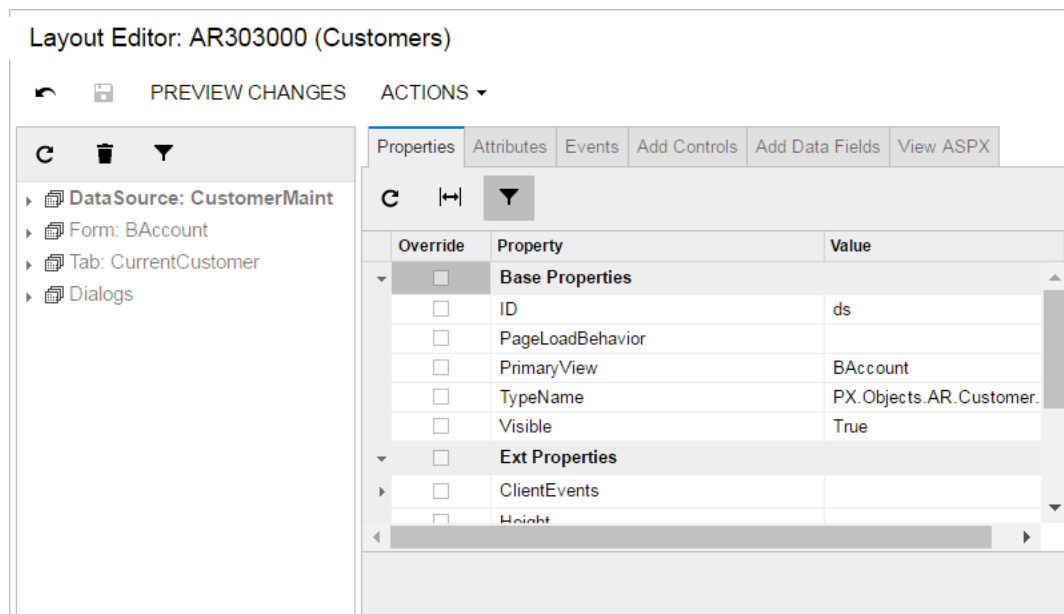
- From the *Element Properties Dialog Box*, which you access by clicking the **Customize** button on the form and selecting **Inspect Element**



: For users' convenience, when the Layout Editor is opened, the control tree displays only the node of the object selected with the Element Inspector. You can click **Show/Hide All Controls** () on the tree toolbar to view all the controls of the form in the tree.

- From the Customized Screens page of the *Customization Project Editor*, by selecting the Screen ID of a screen
- From the navigation pane of the Customization Project Editor, by clicking an item within the **Screens** folder

The title of the Layout Editor page includes the ID and name (in parentheses) of the customized form. The page also has a toolbar, control tree pane, and tabs, as the following screenshot shows.



Page Toolbar

The page toolbar includes standard and page-specific buttons. The page-specific buttons are described below.

Button	Description
Save	Saves to the customization project the difference between the modified and original code of the ASPX page.
Preview Changes	Opens the customized form in a new browser window.
Actions	Provides the following actions: <ul style="list-style-type: none"> • Edit Aspx: Opens the ASPX Editor with the webpage source code. • Open Screen: Opens the original form of Acumatica ERP. • Customize Business Logic: Opens the Code Editor with the extension class template of the business logic container (BLC) that is bound to the form. • Customize Data Class: Opens the Data Class Editor for customization of the DAC that contains the field selected in the Control Tree.

Control Tree

The Control Tree displays the hierarchical structure of controls on the webpage. In the tree, you can:

- Select a container or control for review or customization.
- Change the order of controls in a container.
- Delete any selected item from the webpage.



: We do not recommend that you remove the **DataSource** control.

The toolbar of the Control Tree includes the following buttons.

Button	Description
Refresh	Refreshes the Control Tree.
Delete	Deletes the selected item from the form.
Show/Hide All Controls	Hides or shows all containers in the tree except of the one with the selected item.

To change the order of controls in a container, manually drag controls on the Control Tree within their containers.

Properties Tab

On the **Properties** tab, you can review and modify properties of the form controls.

The tab contains a toolbar and a table. The tab toolbar includes the **Hide Advanced Properties** button (▼) to hide or show advanced properties of the selected control. The tab table consists of the following columns.

Column	Description
Override	A check box that indicates whether the property value was changed. It is selected automatically after you have changed the property.

Column	Description
Property	The property name or the name of the property group.
Value	The value of the property. Default values are not displayed.

Attributes Tab

You use the **Attributes** tab to review and customize the attributes of the DAC field that is bound to the control currently selected in the Control Tree.

The tab consists of the following:

- A summary area with the DAC field information
- Buttons to customize the field attributes

The summary area contains the following elements.

Element	Description
Field Name	The name of the field in the DAC. It is the name of the public virtual property in the public abstract class of the field.
Data Class	The name of the DAC to which the field belongs.
Original Attributes	The original attributes of the field in the Acumatica ERP.

The **Attributes** tab provides the following buttons.

Button	Description
Customize Attributes	Opens the Data Class Editor to customize the field attributes in the DAC.
View Source	Opens the Source Code Browser on the Data Access tab that displays the DAC source code.
Override On Screen Level	Opens the Code Editor with the BLC extension class template. The template includes the field attributes and the template of the <code>DACName_FieldPropertyName_CacheAttached()</code> method, which you can use to replace the attributes within the BLC.

Events Tab

On the **Events** tab, you can view and add event handlers for the selected control.

The tab consists of a summary area, a toolbar, and a table. The summary area contains the following elements.

Element	Description
Data Class	The name of the DAC to which the field belongs.
Field Name	The name of the field in the DAC.
Business Logic	The name of the BLC bound to the form.

The tab toolbar includes the following specific buttons.

Button	Description
Add Handler	For the selected event, opens the Code Editor with the BLC extension class template. The class template includes a code template for the event handler, so you should implement only the body of the handler. Provides the following menu commands: <ul style="list-style-type: none"> • Keep Base Method: Creates the event handler with two parameters, as it is defined in the base BLC. As a result, the event handler is added to the appropriate event handler collection. • Override Base Method: Creates the event handler with an additional parameter to replace the base BLC event handler collection.
View Source	Opens the Source Code browser with the source code of the BLC bound to the form.

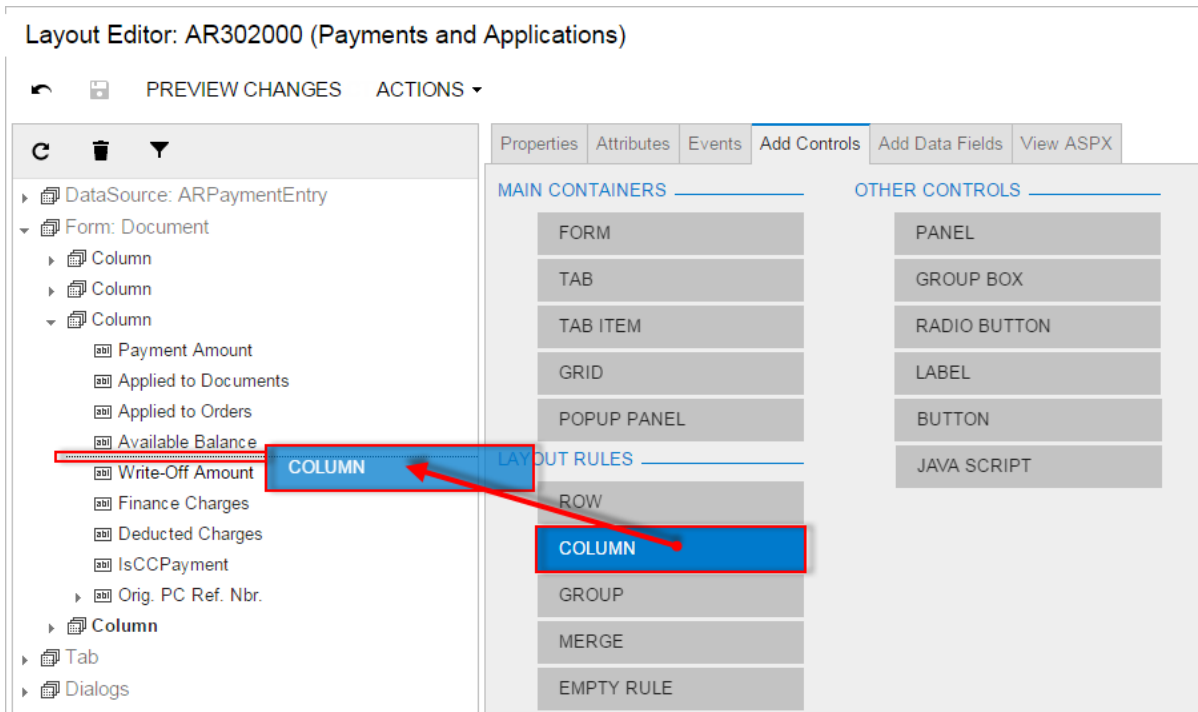
The table holds information about the event handlers used for the control. It contains the following columns.

Column	Description
Event	The event name.
Handled in Source	A check box that indicates whether the event handler is implemented within Acumatica ERP.
Customized	A check box that indicates whether the event handler is customized. This check box is selected automatically after you have added the event handler.

Add Control Tab

On the **Add Control** tab, you can add new containers, controls, and layout rules to the form.

The tab consists of three groups of controls. To add a control, drag and drop it to the needed position in the Control Tree, as the screenshot below shows. After you have added the control, you can configure its properties on the **Properties** tab.




Add Data Fields Tab

You can use the **Add Data Fields** tab to manage controls in the form container that is currently selected in the Control Tree. On this tab, you can:

- Create a control for a DAC field and add it to the selected container.
- Create new custom field in a DAC, create a control for the field, and add it to the selected container.

The tab contains the **Data View** drop-down list, a toolbar, and a table. In the **Data View** drop-down list, you can select a DAC to view its fields in the table. The list includes all `DACName (DataViewName)` pairs of the BLC bound to the form. The tab toolbar contains the following buttons.

Button	Description
Create Controls	Adds the selected fields to the container selected in the Control Tree.
New Field	<p>Opens the Create New Field Dialog Box so you can add a new custom field to the DAC that is referenced in the data view selected in the Data View box.</p> <p> : Once you have created one or more fields, it is necessary to publish the customization project before creating a control for the new fields.</p>

The table displays fields that belong to the selected DAC. The table includes the following columns.

Column	Description
Included	An unlabeled check box that you can use to select a field, for which the Create Controls operation will create a control in the object that is currently selected in the Control Tree.
Used	A check box that indicates whether a control for the field exists in the object that is currently selected in the Control Tree. This check box is selected automatically after you have created a control for the field.

Column	Description
Field Name	The name and <code>DisplayName</code> (in parentheses) of the field in the DAC.
Control	The type of the UI control.

You can use the following filters of DAC fields displayed in the table.

Filter	Description
All	Fields that are not represented by controls on the form.
Visible	Fields that are visible due to the <code>Visibility</code> field attribute in the DAC.
Custom	New custom fields of the DAC.

View ASPX Tab

The **View ASPX** tab displays the ASPX code of the control selected in the Control Tree. The changes in the code of the original page are highlighted in yellow, as the following screenshot shows.

ASPX Editor

You can use the ASPX Editor to edit the ASPX code of a page that has been customized by means of the Layout Editor. You can use this editor instead of the Layout Editor if you prefer a text editor rather than a visual tool.

You can launch the ASPX Editor by clicking **Actions > Edit Aspx** on the toolbar of the Layout Editor.

The ASPX Editor page contains the following parts (see the screenshot below):

- A toolbar with the single **Generate Customization Script** button
- The work area of the editor

Aspx Editor: AR302000 (Payments and Applications)

GENERATE CUSTOMIZATION SCRIPT

```

1 <%@ Page Language="C#" MasterPageFile="~/MasterPages/FormDetail.master" AutoEventWireup="true" ValidateRequest="false" CodeF
2
3 <%@ MasterType VirtualPath="~/MasterPages/FormDetail.master" %>
4 <asp:Content ID="cont1" ContentPlaceHolderID="phDS" runat="Server">
5   <px:PXDataSource ID="ds" runat="server" Visible="True" Width="100%" TypeName="PX.Objects.AR.ARPaymentEntry" PrimaryView=
6     <CallbackCommands>
7       <px:PXDSCallbackCommand CommitChanges="True" Name="Save" PopupVisible="true" />
8       <px:PXDSCallbackCommand Name="Insert" PostData="Self" />
9       <px:PXDSCallbackCommand Name="First" PostData="Self" StartNewGroup="True" />
10      <px:PXDSCallbackCommand Name="Last" PostData="Self" />
11      <px:PXDSCallbackCommand StartNewGroup="True" Name="Release" PopupVisible="true" CommitChanges="true" />
12      <px:PXDSCallbackCommand Visible="false" Name="ViewBatch" />
13      <px:PXDSCallbackCommand Name="Action" CommitChanges="True" StartNewGroup="True" />
14      <px:PXDSCallbackCommand Name="Inquiry" RepaintControls="All" CommitChanges="True" />
15      <px:PXDSCallbackCommand Name="Report" CommitChanges="true" />
16      <px:PXDSCallbackCommand Name="CurrencyView" Visible="False" />
17      <px:PXDSCallbackCommand Name="NewCustomer" Visible="False" />
18      <px:PXDSCallbackCommand Visible="false" Name="EditCustomer" />
19      <px:PXDSCallbackCommand Visible="false" Name="CustomerDocuments" />
20      <px:PXDSCallbackCommand Visible="false" CommitChanges="true" Name="LoadInvoices" />
21      <px:PXDSCallbackCommand Visible="false" CommitChanges="true" Name="LoadOrders" />
22      <px:PXDSCallbackCommand Visible="false" Name="ReverseApplication" CommitChanges="True" DependOnGrid="detgrid2" />
23      <px:PXDSCallbackCommand Name="ViewDocumentToApply" DependOnGrid="detgrid" Visible="false" />
24      <px:PXDSCallbackCommand Name="ViewSODocumentToApply" DependOnGrid="detgrid3" Visible="false" />
25      <px:PXDSCallbackCommand Visible="false" Name="ViewApplicationDocument" DependOnGrid="detgrid2" />
26      <px:PXDSCallbackCommand Visible="false" Name="ViewCurrentBatch" DependOnGrid="detgrid2" />
27      <px:PXDSCallbackCommand Visible="false" Name="ViewVoucherBatch" />
28      <px:PXDSCallbackCommand Visible="false" Name="ViewWorkBook" />
29      <px:PXDSCallbackCommand Visible="false" CommitChanges="true" Name="CaptureCCPayment" />
30      <px:PXDSCallbackCommand Visible="false" CommitChanges="true" Name="AuthorizeCCPayment" />
31

```

You use the **Generate Customization Script** button instead of a **Save** button because this action saves to the customization project not the modified ASPX code, but the difference between the modified code and the original code of the page.

Data Class Editor

You use the Data Class Editor to develop the content of extensions for data access classes (DACs). By using the editor, for example, you can do the following:

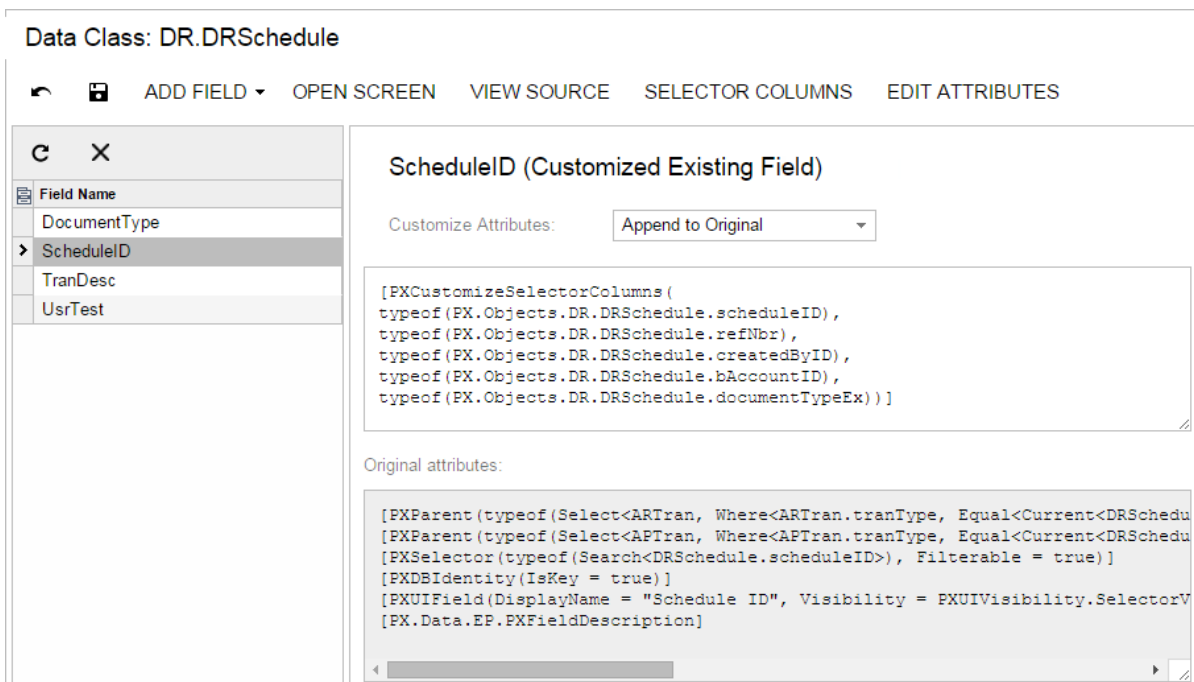
- Customize the attributes of the existing fields of a DAC
- Add custom fields to a DAC
- For a selector field, add, delete, and sort the columns of the selector table
- Review the modifications made to an original class
- View the source code of an original data access class
- Navigate to the form on which a field of a class is used

While performing the customization, you can open the editor in the following ways:


- From the **Customization Menu > Inspect Element > Element Properties** dialog box, by selecting the **Customize Data Fields** command on the **Actions** menu
- From the Customized Data Classes page of the Customization Project Editor, by selecting the **Name** field of a DAC
- From the navigation pane of the Customization Project Editor, by clicking an item in the **Data Access** folder

The Data Class Editor page includes the following elements (see the screenshot below):

- The title with the name of the customized class
- A toolbar with standard and page-specific buttons
- A list of the customized fields of the class
- A work area to customize the attributes of the field that is currently selected in the list



The page toolbar contains the following page-specific buttons.

Button	Description
Add Field	Provides the following actions: <ul style="list-style-type: none"> • Create New Field: Opens the Create New Field Dialog Box, which you can use to add a new custom field to the DAC. • Change Attributes of Base Field: Opens the Change Existing Field Dialog Box, which you can use to add an existing field of the DAC to the customization project.
Open Screen	For the DAC, opens the primary form of Acumatica ERP.  : If there is a primary business logic controller (BLC) for the class, then the primary form is the form bound to this container. Otherwise, the primary form is the form bound to the BLC in which the class is the main DAC of the primary view. See the Acumatica Framework training documentation for details.
View Source	Opens the Source Code Browser on the Data Access tab, which displays the DAC source code.
Selector Columns	(Available for only the fields that are selectors.) Opens the Customize Selector Columns Dialog Box , in which you can modify the columns in the selector table.
Edit Attributes	(Unavailable for new custom fields.) Opens the Customize Attributes Dialog Box , which you can use to customize the attributes of the selected field.

You use the list of the currently customized fields of the DAC to select the field that you can review or change in the work area.

The work area of the page consists of the following:

- The title with the name and type (in parentheses: **New Field** or **Customized Existing Field**) of the selected field
- The **Customize Attributes** text area, where you can edit the attributes of the selected field
- The **Original Attributes** text area, where you can view the original attributes of the selected field

- A drop-down list that is visible for only existing customized fields of the data access class, so you can select one of the following ways of applying the changes to the field attributes:
 - *Keep Original*: The original attributes remain on the field until you select another option in this box.
 - *Replace Original*: The original attributes are replaced on the field with the custom attributes specified in the **Customize Attributes** text area.
 - *Append to Original*: The custom attributes are added to the end of the list of the original attributes of the data field. If you use this option, make sure you do not duplicate attributes on the field.



: The platform gives you advanced possibilities to control the field customization by using additional attributes in the DAC extension. See [Customization of Field Attributes in DAC Extensions](#) of [Customization Framework](#) for details.

If you want to change the original attributes in the **Customize Attributes** text area, before typing any text in the area, select the *Replace Original* option. The original attributes will be copied to the **Customize Attributes** text area.

Create New Field Dialog Box

You can use the **Create New Field** dialog box to add a new data field to the customized DAC.

You can open the dialog box, shown in the following screenshot, in either of the following ways:

- From the **Add Data Fields** tab of the Layout Editor, by clicking the **New Field** button
- From the **Add Field** button of the Data Class Editor, by selecting the **Create New Field** command

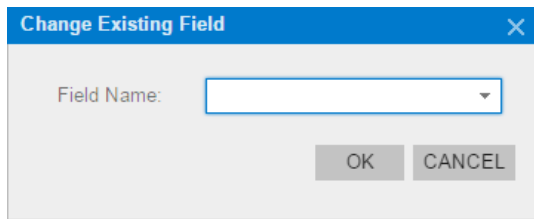
The dialog box controls are described below.

Control	Description
Field Name	The name of the field in the DAC. Because the field is custom, the <i>Usr</i> prefix is added automatically to the name when the box loses focus.
Display Name	The name of the field to be displayed on the form. The specified string is inserted into the <code>DisplayName</code> parameter of the <code>PXUIField</code> attribute of the field.
Storage Type	The storage type for the field, which can be one of the following: <ul style="list-style-type: none"> • <i>NonPersistedField</i>: An unbound field of the DAC. The unbound field is added only to the data access class and is not mapped to the database. • <i>DBTableColumn</i>: An ordinary bound field. The platform adds the database column to the base table by altering the base table schema. • <i>NameValuePair</i>: A "name-value" bound field. The platform stores the field value in a dedicated table of the database without altering the schema of the base table.

Control	Description
Data Type	The data type to be used for a custom fields, which can be one of the following: <i>string</i> , <i>int</i> , <i>bool</i> , <i>decimal</i> , <i>datetime</i> , and <i>guid</i> .
Length	(Available if you have selected the <i>string</i> or <i>decimal</i> type.) For the <i>string</i> type, the maximum number of symbols in the field value; for the <i>decimal</i> type, the precision of the value (the maximum total number of decimal digits that will be stored, both to the left of the decimal point and to the right of it).
Decimal	(Available only if you have selected the <i>decimal</i> type.) The scale of the value (that is, the number of decimal digits that will be stored to the right of the decimal point).
OK	Adds the new field with the specified properties to the data access class and closes the dialog box. The field appears in the list of customized fields of the class in the Data Class Editor.
Cancel	Closes the dialog box.

Change Existing Field Dialog Box

The **Change Existing Field** dialog box, shown below, is used to add an existing field of the data access class (DAC) to the list for customization.



You can open the dialog box by clicking the **Add Field** button of the Data Class Editor and selecting the **Change Attributes of Base Field** command.

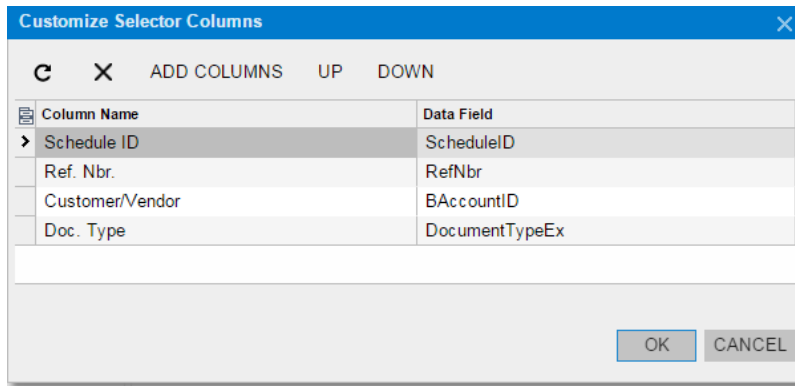
The dialog box controls are described below.

Control	Description
Field Name	The existing field of the DAC you want to change.
OK	Adds the selected field to the list of the customized fields of the data access class, and closes the dialog box.
Cancel	Closes the dialog box.

Customize Selector Columns Dialog Box

The **Customize Selector Columns** dialog box, shown below, is available for only a selector field. In the dialog box, you can add, delete, and sort columns to customize the selector table. The dialog box contains a toolbar, a table, the **OK** and **Cancel** buttons.

You can open the dialog box from the Data Class Editor by clicking the **Selector Columns** button.



The dialog box toolbar includes the buttons described below.

Button	Description
Refresh	Refreshes the list of columns displayed in the table.
Delete	Deletes the selected column from the selector.
Add Columns	Opens the Add Columns to Selector Dialog Box , which you can use to add columns to the table of this dialog box.
Up	Moves up the selected field so the column moves left in the selector.
Down	Moves down the selected field so the column moves right in the selector.

The table contains information on the selector content and includes the following columns.

Column	Description
Column Name	The value of the <code>DisplayName</code> parameter of the <code>PXUIField</code> attribute of the field.
Data Field	The name of the public virtual property of the field in the DAC.

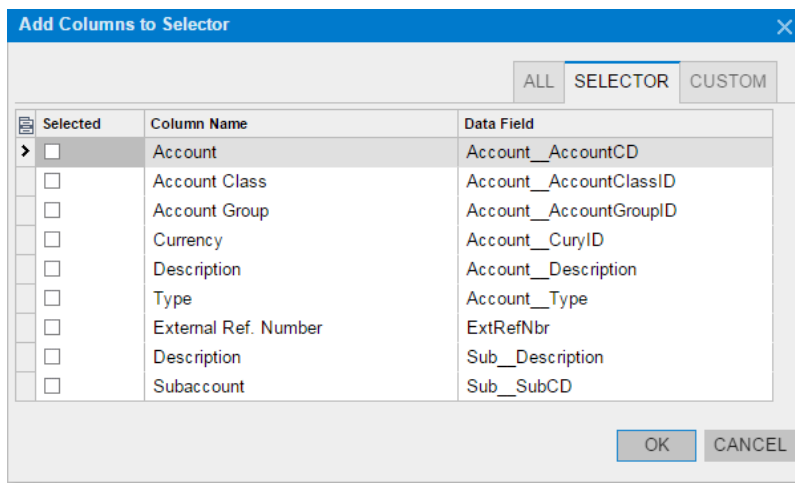
After you click **OK** in the **Customize Selector Columns** dialog box, the system applies the modifications to the selector table. As a result, the `PXCustomizeSelectorColumns` attribute is added to the selector field, and you can view the attribute in the **Customize Attributes** text area of the Data Class Editor. This attribute defines the new set and order of the columns in the selector.

The **Cancel** button closes the dialog box without saving changes to the selector table.

Add Columns to Selector Dialog Box

The **Add Columns to Selector** dialog box, shown in the following screenshot, is used to add one or multiple columns to the selector table at once. The dialog box contains a table with filters and the **OK** and **Cancel** buttons.

You can open the dialog box from the **Customize Selector Columns** dialog box by clicking the **Add Columns** button.



The table includes the following columns.

Column	Description
Selected	A check box you can use to select fields that will be added to the selector.
Column Name	The value of the <code>DisplayName</code> parameter of the <code>PXUIField</code> attribute of the field.
Data Field	The name of the public virtual property of the field in the DAC.

Each tab of the **Add Columns to Selector** dialog box displays the filtered list of fields. The tabs are described below.

Filter	Description
All	All fields.
Selector	Fields that are selectors.
Custom	New custom fields.

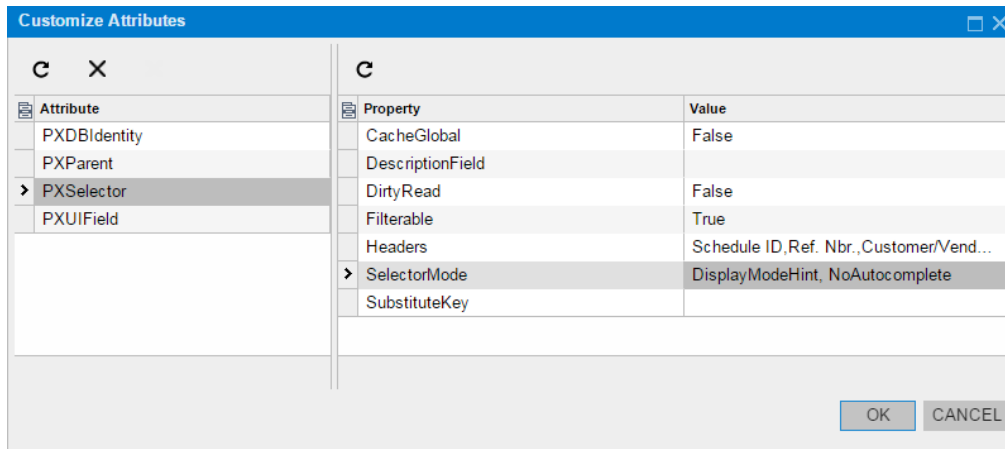
After you click **OK** in the **Add Columns to Selector** dialog box, the system applies the modifications. As a result, the selected columns are added to the table of the **Customize Selector Columns** dialog box.

The **Cancel** button closes the dialog box without saving the changes to the selector table.

Customize Attributes Dialog Box

The **Customize Attributes** dialog box (shown in the following screenshot) provides the easiest way to edit attributes of the field selected in the Data Class Editor. The dialog box contains an attribute pane, a work area, and the **OK** and **Cancel** buttons.

You can open the dialog box from the Data Class Editor by clicking the **Edit Attributes** button.



The attribute pane includes:

- A toolbar with the **Refresh** and **Delete** buttons.
- The list of the field attributes.

In the list, you can select an attribute to be deleted or edited.

You can use the **Delete** button to delete the selected attribute of the field. After you delete the attribute and click **OK**, the `[PXRemoveBaseAttribute(typeof(AttrNameAttribute))]` attribute is added to the field and you can view the attribute in the **Customize Attributes** text area of the Data Class Editor.

The work area contains a table that lists the parameters of the attribute selected in the list. The table columns are described below.

Column	Description
Property	The name of the parameter.
Value	The original value of the parameter.

In the table, you can enter any string as the parameter value without type validation. If the value of the customized parameter has an incorrect type, it causes a validation error during the publication of the project.

After you click **OK** in the **Customize Attributes** dialog box, the platform applies the modifications to the field. As a result, the `[PXCustomizeBaseAttribute(typeof(AttrNameAttribute), "ParameterName", NewValue)]` attribute is added to the field for each modified parameter. You can view the attribute in the **Customize Attributes** text area of the Data Class Editor.

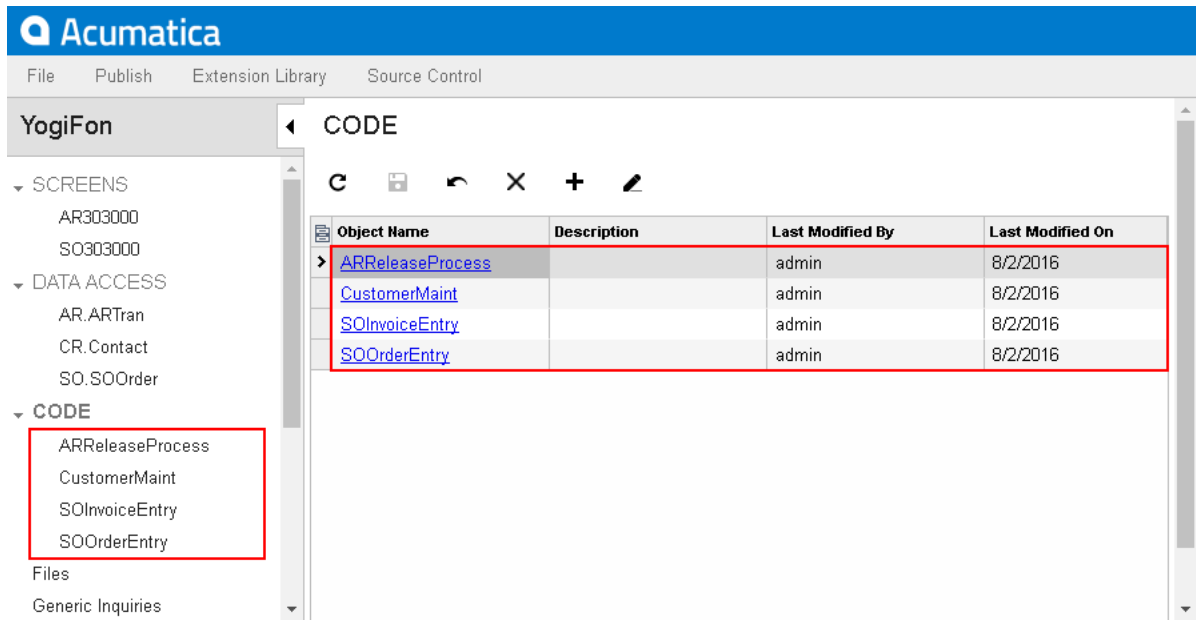
You can click the **Cancel** button to close the dialog box.

Code Editor

With the Code Editor, you can develop, view, and edit the customization code that has been added to the project.

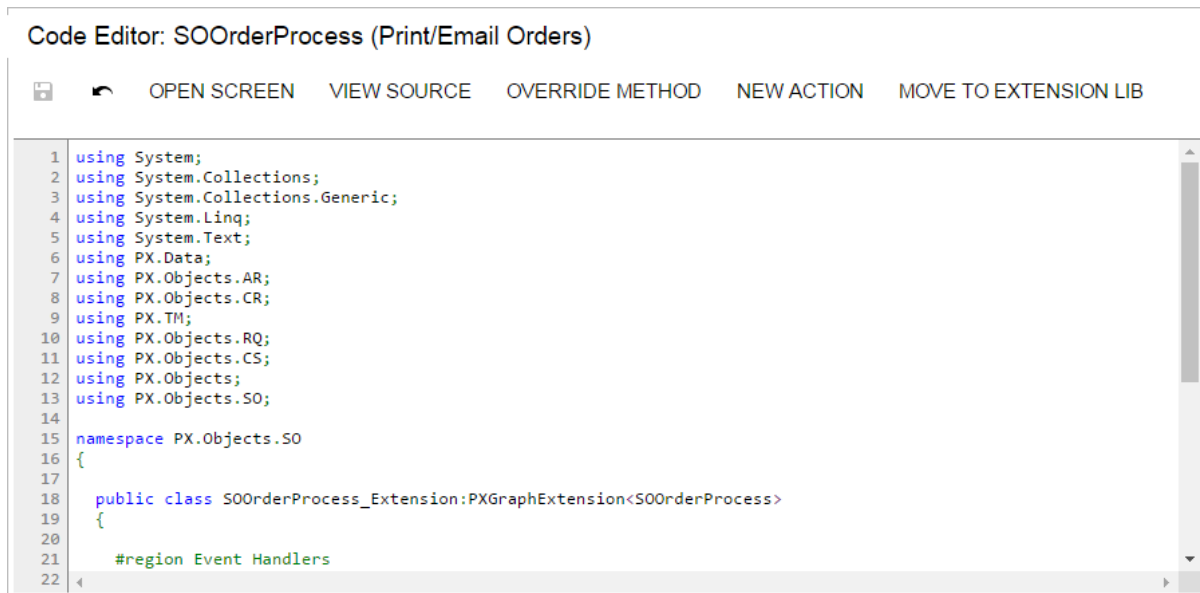
You can open the Code Editor in the following ways:

- From a form of Acumatica ERP by using **Customization > Inspect Element** to open the **Element Properties** dialog box, and then clicking **Actions > Customize Business Logic**
- From the Code page of the Customization Project Editor—by clicking the **Object Name** of an existing `Code` item (see the screenshot below)





- From the navigation pane of the Customization Project Editor, by clicking an item in the **Code** folder

The Code Editor page includes a toolbar and a text area for editing code, as shown in the following screenshot.



The toolbar buttons of the Code Editor page are described below.

Button	Description
Save	Saves the code in the project.  : You can use the Control-S combination on the keyboard to save the code.
Cancel	Cancels unsaved changes in the code.

Button	Description
Open Screen	Opens the form bound to the business logic controller if you are editing the customization code of the business logic executed for a form.
View Source	Opens the Source Code Browser with the original source code of the business logic controller (BLC) if you are editing the customization code of the business logic executed for a form.
Override Method	Opens the Select Methods to Override Dialog Box , which you can use to select multiple virtual methods of the BLC to override.
New Action	Opens the Create Action Dialog Box , which you can use to create a code template for a new action.
Move to Extension Lib	<p>Launches the operation that converts the current <code>Code</code> item into a file of customization code, adds the file to the extension library project in Microsoft Visual Studio, and removes the item from the customization project. See Move to Extension Lib Action for details.</p> <p> : The customization project must be bound to an existing extension library project in Visual Studio before you invoke the Move to Extension Lib operation. See Customization Project Editor for details.</p>

Create Action Dialog Box

If the `Code` item you are viewing by using the Code Editor is a business logic controller (BLC, also referred to as *graph*) extension, you can create a new action in this BLC. To do this, you can click the **New Action** button of the Code Editor to open the **Create Action** dialog box, shown in the following screenshot.

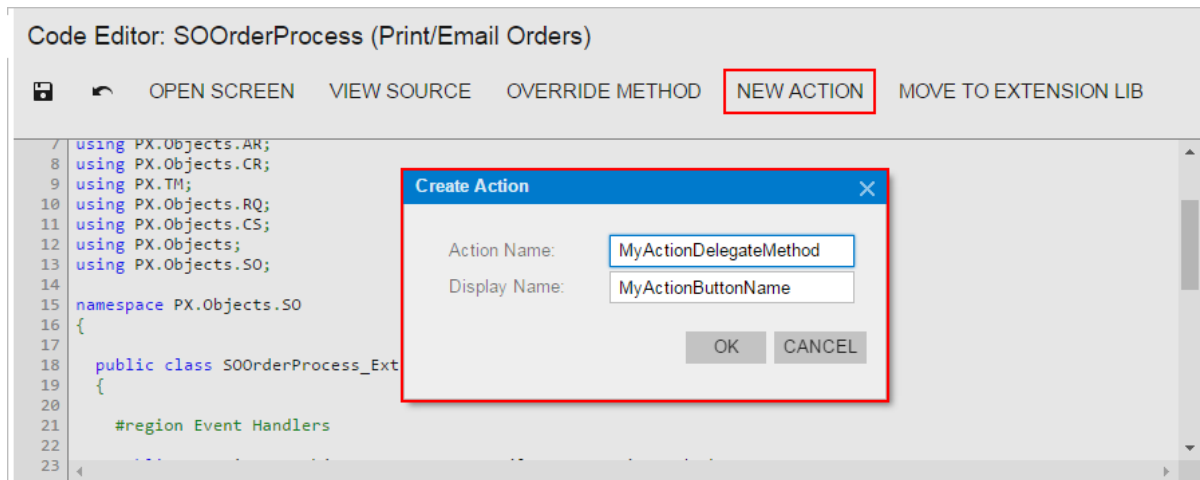


Figure: Opening the New Action dialog box

To create an action declaration in the BLC extension, you should specify the name of the action delegate method in the **Action Name** box and the name of the action button in the **Display Name** box, as shown in the screenshot above, and then click **OK**. The system adds to the graph extension a template of the action declaration that includes the following class members:

- The declaration of the action delegate method
- The declaration of the button attributes to add the button to the form toolbar with the specified name
- The template of the action delegate method

The following example shows the template code for an action.

```
public PXAction<DACName> myActionDelegateMethod;

[PXButton(CommitChanges = true)]
[PXUIField(DisplayName = "MyActionButtonName")]
protected void MyActionDelegateMethod()
{
    // the body of the action delegate method
}
```

Select Methods to Override Dialog Box

If the *Code* item you are viewing in the Code Editor is a business logic controller (BLC) extension, you can override a virtual method of this BLC. To do this, you can use the **Override Method** action of the Code Editor, which opens the **Select Methods to Override** dialog box, shown in the following screenshot.

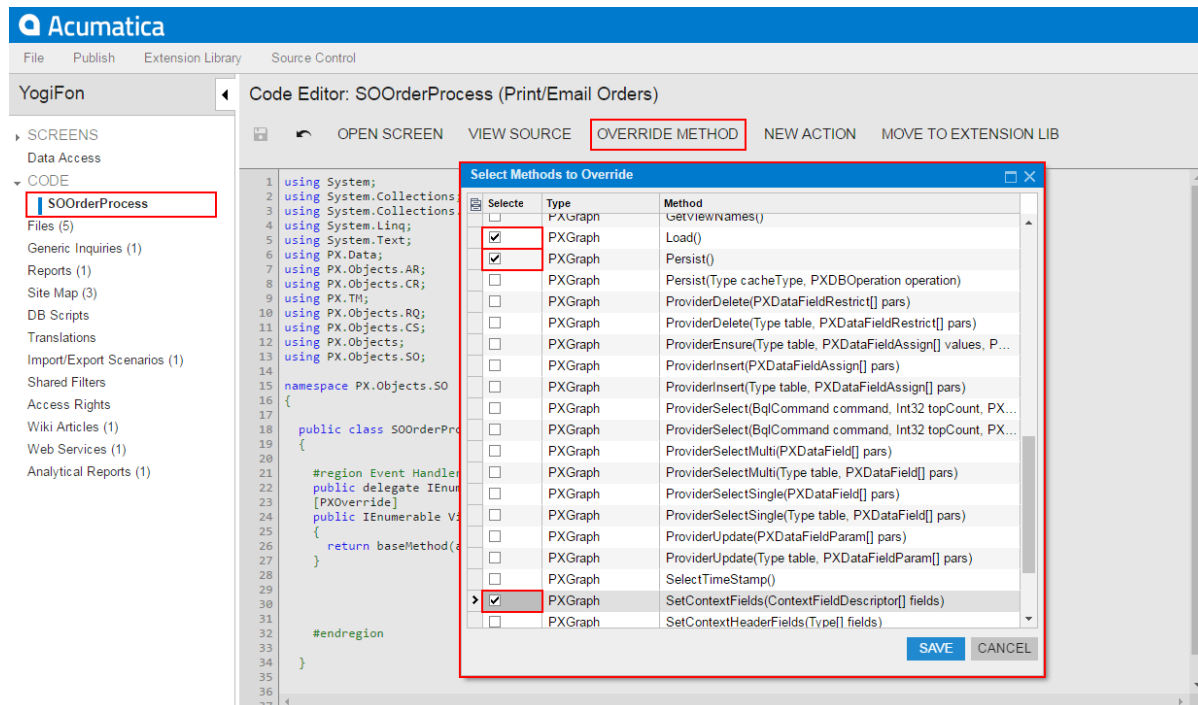


Figure: Opening the Select Methods to Override dialog box

To add a method to the customization, you should select the check box for the method in the table, as shown in the screenshot, and click **Save**. The system adds to the graph extension a template of an overridden method for each method selected in the table.

The dialog box contains a table that lists the virtual methods of the current BLC and its parent classes, and the **Save** and **Cancel** buttons. You use the table, which contains the following columns, to select the virtual methods that you have to override.

Column	Description
Selected	A check box that you can use to select the virtual method to be overridden in the BLC extension.
Type	The identifier of the class type that contains the declaration of the virtual method.
Method	The signature of the virtual method.

To cancel the operation and close the dialog box, click **Cancel**.

Click **Save** to close the dialog box and launch the process that adds a code template to the BLC extension for each item selected in the table.

When you override a virtual method, the system generates an overridden method template that includes the delegate declaration of the base method, the `PXOverride` attribute, and the method declaration that invokes the base method delegate.

The following example shows the template code to override the `MethodName` method.

```
public delegate returnType MethodNameDelegate(...);
[PXOverride]
public returnType MethodName(..., MethodNameDelegate baseMethod)
{
    return baseMethod(...);
}
```

Move to Extension Lib Action

You can develop customization code either as *Code* items in a customization project or as source code included in an extension library project to develop in Microsoft Visual Studio. Some part of a customization may exist in the *Code* items of a customization project, while another part can be included in an extension library that is added to the customization project as a dynamic link library (DLL) file.

If you have a *Code* item in a customization project that you want to move to an extension library to compile it into a DLL, you can use the **Move to Extension Lib** action on the page toolbar of the Code Editor.

Before you launch the operation, be aware that the customization project is bound to an existing extension library. (See [Customization Project Editor](#) for details.)

After the operation is complete, the *Code* item that is currently displayed in the work area of the Code Editor is removed from the customization project, and the file with the same source code is appended to the extension library that is bound to the customization project. The system assigns a similar name to the file: For example, if the *Code* item name was `CodeItemName`, the name of the created file will be `CodeItemName.cs`.

For example, suppose you need to move the `CustomerMaint` *Code* item to the `YogiFon` extension library (see the screenshot below).

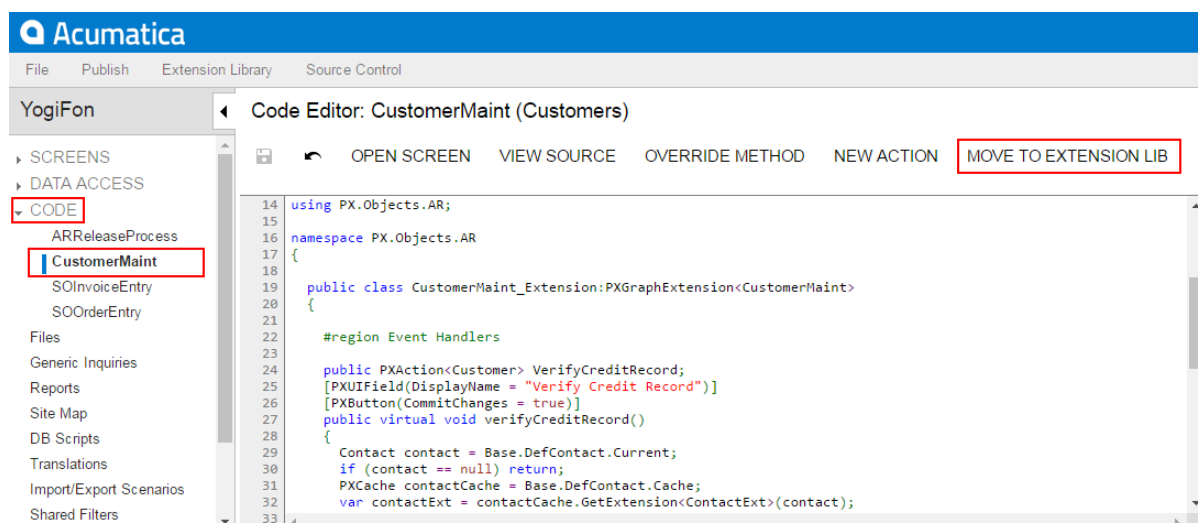


Figure: Viewing the content of the Code item before you move it to the extension library

When the operation is complete, the `CustomerMaint` *Code* item is removed from the customization project, as the screenshot below shows.

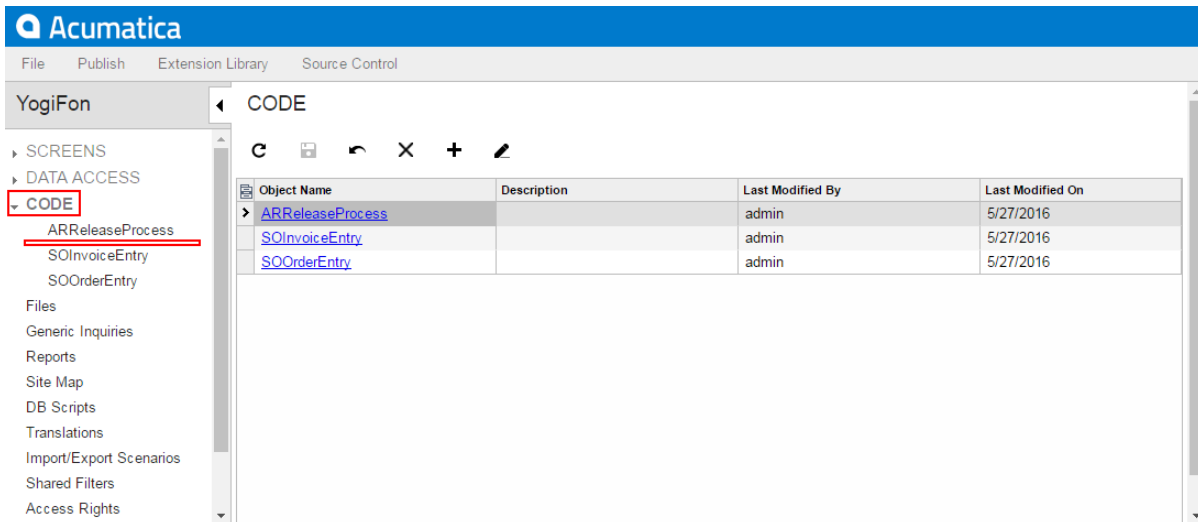


Figure: Viewing the Code item list after the operation was performed

In place of the removed item, the `CustomerMaint.cs` file with the same source code is appended to the bound extension library project, as shown in the screenshot below.

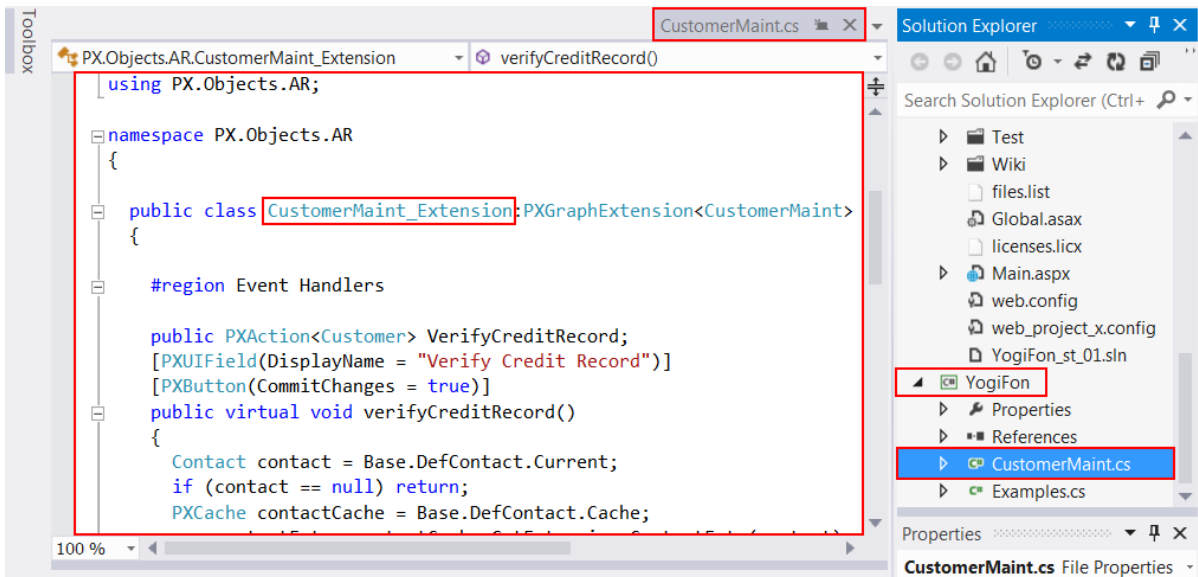


Figure: Viewing the content of the source code file added to the extension library

The operation of moving code to an extension library is irreversible. If you need to move source code from an extension library to a *Code* item of a customization project, use the following approach:

- In Visual Studio (or any text editor), open the file, select the needed source code, and copy it to the clipboard.
- Create a new *Code* item in the customization project.
- Delete the code template from the created item.
- Paste the code from the clipboard, and save the *Code* item to the customization project.
- Delete the source code file from the extension library.

File Editor

You can use the File Editor to view or edit a custom file added to the customization project.

To open a custom file in the editor, on the Custom Files page of the Customization Project Editor, click the file name in the list of custom files.

You can use the editor as follows with custom files included in the project:

- To review and edit a text file
- To review the content of a binary DLL file (see the following screenshot)

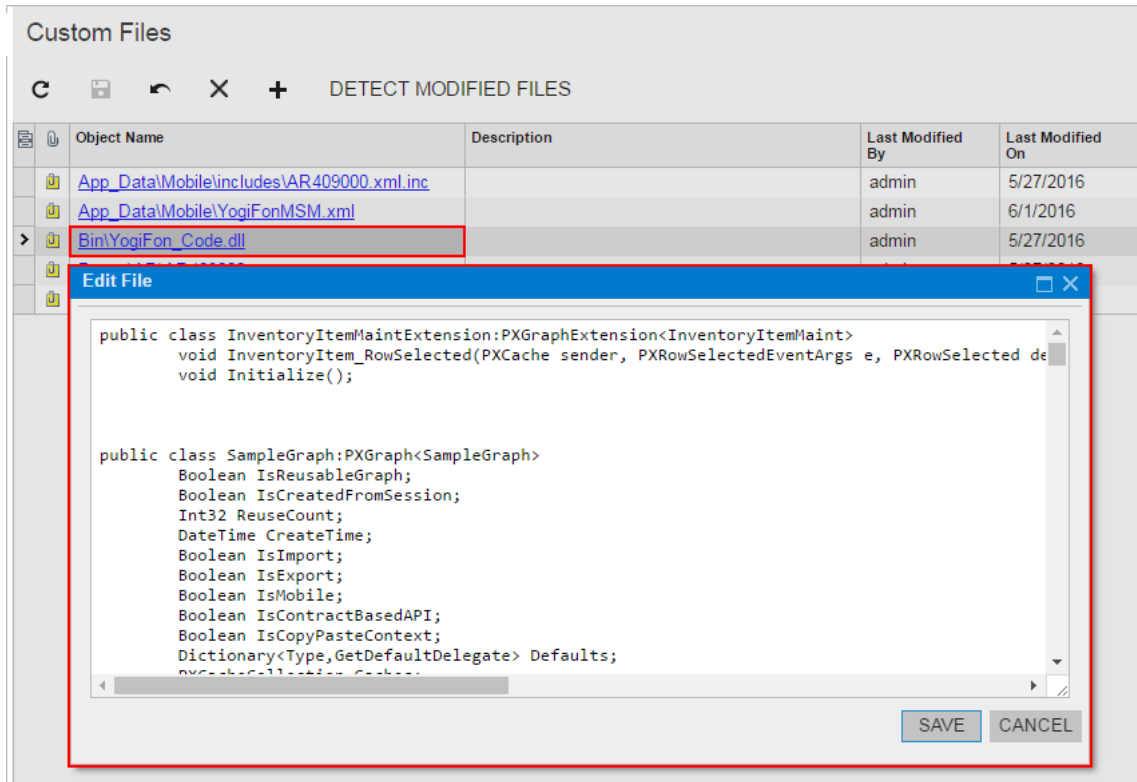


Figure: Viewing the content of a custom DLL file in the File Editor



: In the editor, you cannot save changes in DLL files.

If you have used the File Editor to modify a custom file in a customization project and have saved the changes in the database, the changes are not saved in the original file in the file system. If you then click **Detect Modified Files** on the toolbar of the Custom Files page, the platform does not detect a conflict because the file in the database is newer. The platform automatically updates the original file in the file system during the publication of the customization project.

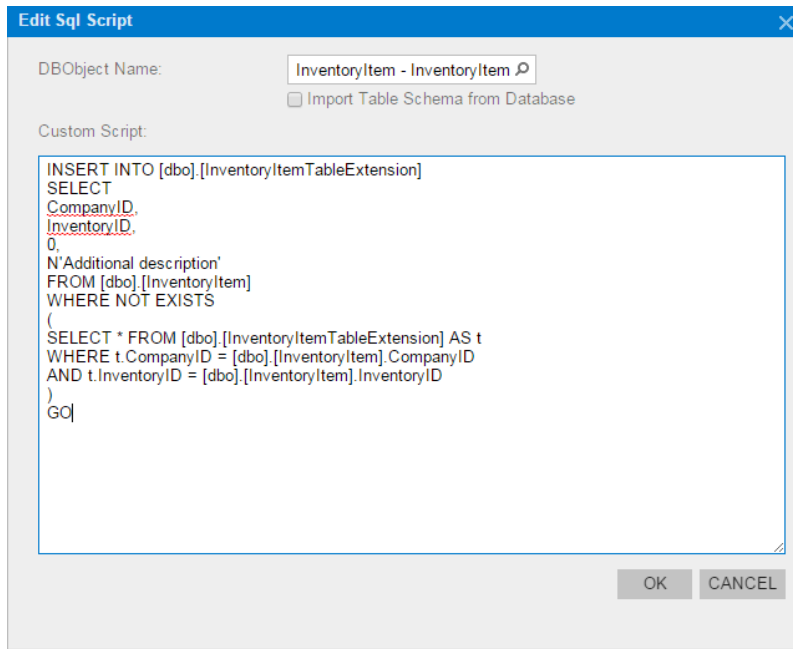
SQL Script Editor

You can use the SQL Script Editor to do the following:


- Add a custom table to the customization project
- Add and edit a custom SQL script

You can open the editor, shown in the following screenshot, from the Database Scripts page of the Customization Project Editor in the following ways:

- By selecting the **Object Name** field of an existing script: To edit the script
- By clicking the **Add New Record (+)** button on the toolbar: To create a new SQL script



The SQL Script Editor includes the following UI controls.

Control	Description
DBObjectName	<p>A text box with a selector that you can use as follows:</p> <ul style="list-style-type: none"> To select an already created custom table to add the table schema to the project To specify the name of the SQL script that you are going to add to the project
Import Table Schema from Database	<p>(This check box appears only if the DBObjectName box holds the name of an existing DB table.) A check box that indicates (if selected) that the schema of the selected table is added to the customization project.</p> <p> : You can view the table schema added to the project in the Source text box of the <i>Item XML Editor</i>.</p>
Custom Script	A text area that you can use to view and edit a custom SQL script.
OK	A button you click to save the custom SQL script to the customization project.
Cancel	A button you click to close the dialog box without saving changes to the script.

See [Database Scripts](#) for details about using the SQL Script Editor.

XML Editors

In the database, the platform keeps each item of the customization project in XML format.

The platform provides the following tools for experienced users to edit the content of a customization project in XML format:

- [Project XML Editor](#)
- [Item XML Editor](#)

Project XML Editor

You can use the Project XML Editor to edit and review the content of a customization project in XML format.

To open the editor, in the Customization Project Editor, click **File > Edit Project XML** on the menu (see the following screenshot).

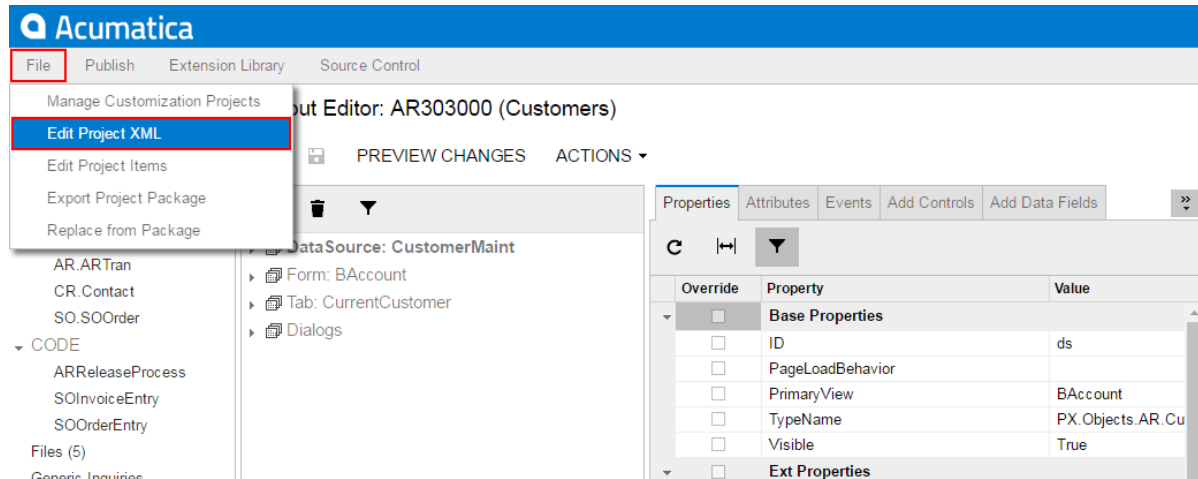


Figure: Opening the Project XML Editor

The editor page, shown in the screenshot below, contains the following UI elements:

- At the top of the page, the name of the customization project that is opened in the editor
- The page toolbar, which includes the file name of a selected deployment package (or the *No file chosen* message) and the following buttons:
 - **Save to database:** To save the XML code of the project to the database
 - **Download Package:** To save the project locally as a deployment package ZIP file that has the same name as the customization project
 - **Upload Package:** To open the deployment package file whose name is currently selected and displayed in the page toolbar
 - **Choose File:** To select a deployment package file
- The work area of the editor

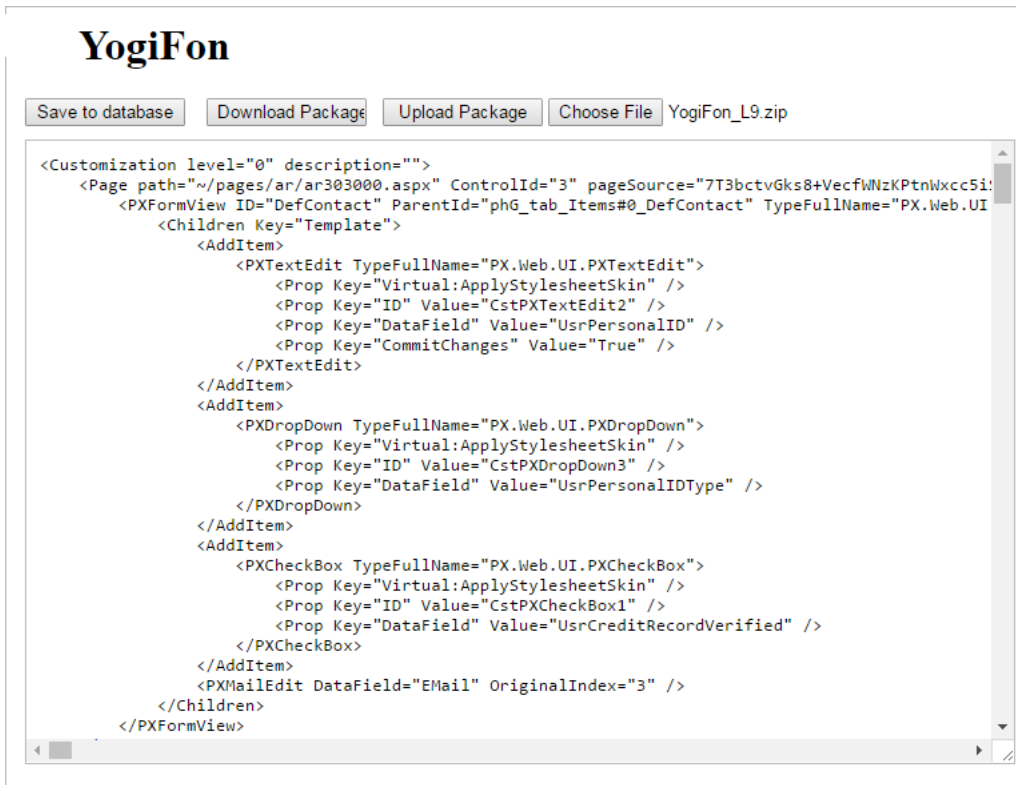


Figure: Viewing the Project XML Editor page

Item XML Editor

You can use the Item XML Editor to edit and review an item of a customization project.

To open the editor, in the Customization Project Editor, click **File > Edit Project Items** on the menu (see the following screenshot).

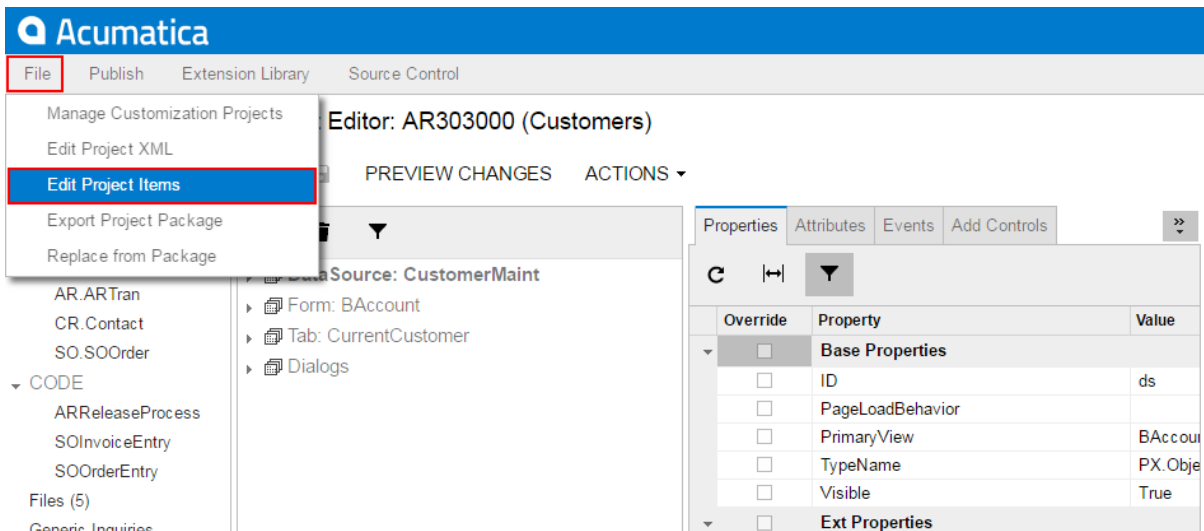


Figure: Opening the Item XML Editor

The editor page, shown in the following screenshot, contains the following UI elements:

- The page toolbar, which contains the **Save** and **Cancel** buttons
- The list of the items in the customization project
- The work area of the editor

Edit Project Items

Object Name	Type	Descr	Exclud	Created By	Creation Date	Last Modified By	Last Modified On
PX.Objects.AR.ARTran	DAC		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
PX.Objects.CR.Contact	DAC		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
PX.Objects.SO.SOOrder	DAC		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
App_Data\Mobile\includes\AR409000.xml.inc	File		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
App_Data\Mobile\YogiFonMSM.xml	File		<input type="checkbox"/>	admin	5/27/2016	admin	6/1/2016
Bin\YogiFon_Code.dll	File		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
Pages\AR\AR409000.aspx	File		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
Pages\AR\AR409000.aspx.cs	File		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
ARReleaseProcess	Code		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016
SOInvoiceEntry	Code		<input type="checkbox"/>	admin	5/27/2016	admin	5/27/2016

Source

```
<DAC type="PX.Objects.AR.ARTran">
  <Field fieldName="UsrSIMCardID" typeName="string" MapDbTable="ARTran" TextAttributes="#CDATA" StorageAttributes="TextAttributes"><![CDATA[[PXDBString(40)
[PXUIField(DisplayName="SIM Card ID")
]]></CDATA>
  </Field>
  <Field fieldName="UsrPhoneNumber" typeName="string" MapDbTable="ARTran" TextAttributes="#CDATA" StorageAttributes="TextAttributes"><![CDATA[[PXDBString(15)
[PXUIField(DisplayName="Phone Number")
]]></CDATA>
  </Field>
  <Field fieldName="UsrContractID" typeName="int" MapDbTable="ARTran" TextAttributes="#CDATA" StorageAttributes="TextAttributes"><![CDATA[[PXDBInt]
[PXUIField(DisplayName="Contract ID" Enabled = false)]]></CDATA>
```

Figure: Viewing the Item XML Editor page

Source Code Browser

You can explore the source code of the Acumatica ERP application on the [Source Code](#) (SM.20.45.70) form, which is shown in the following screenshot.

The screenshot shows the Acumatica customization platform interface. At the top, there is a navigation bar with tabs for Management, Integration, Automation, and Customization. The Customization tab is active, and the 'Source Code' sub-tab is selected. The main area displays the source code for a form with ScreenID 'SM204505 - Customization Projects'. The code is a mix of ASPX and C# code, including page headers, data source definitions, and grid controls. A sidebar on the left contains a 'MANAGE' section with options like Customization Projects, Generic Inquiry, Lists as Entry Points, Pivot Tables, Dashboards, Site Map, Portal Map, and Filters. Below this is an 'EXPLORE' section with 'Source Code' highlighted in a red box.

Figure: Viewing the Source Code form

You can open the browser in the following ways:

- From the *Element Properties Dialog Box*, by selecting one of the following commands in the **Actions** menu:
 - **View ASPX Source:** To explore the source code of current form
 - **View Business Logic Source:** To explore the source code of the business logic controller (BLC) that is bound to the current form
 - **View Data Class Source:** To explore the source code of the data access class (DAC) that contains the data field of the inspected element
- From the *Data Class Editor*, by clicking **View Source** on the page toolbar
- From the *Code Editor*, by clicking **View Source** on the page toolbar
- From the *Layout Editor*, by clicking **View Source** on the toolbar of the **Attributes** or **Events** tab

You can use the Source Code browser for the following purposes:

- To search for the following types of source code:
 - ASPX code, by the screen ID or by the screen title. You can search by any part of an ID or of a title.
 - C# source code of a BLC or DAC, by the class name. You can also type any part of the name and select the name from the list of names that match your search string.
 - Any source code, by a text fragment.
- To download a ZIP file with the source code for the website

Customization Framework

The programming framework of the Acumatica Customization Platform, like Acumatica Framework, is intended for developers experienced in C#.NET. Because the application objects of Acumatica ERP are built on top of Acumatica Framework, the developers need to learn both the programming frameworks to be able to effectively develop quality customizations.

When you have to customize an instance of Acumatica ERP, first you must determine the scope of the customization. For each form that works with data from the database, the instance of Acumatica ERP must contain at least the following objects (see the diagram below):

- An ASPX page: The page must contain, at minimum, the data source control and a control container with controls for data fields.
- A business logic controller (BLC, also referred to as *graph*): The graph must be specified in the `TypeName` property of the data source control of the page. The graph must contain at least one data view, which is specified in the `PrimaryView` property of the data source control as well as in the `DataMember` property of the control container. The graph instance is created on each round trip and initializes the creation of the data view instance based on a BQL statement. The data view provides data manipulations and data flows between the container control, the cache object of the graph, and the corresponding table of the database. The BQL statement contains a reference to at least one data access class that is required to map the database table to data records in the cache object.
- A data access class (DAC): On each round trip, the DAC instance is created in the cache object when the data view processes any operation with the corresponding data.
- A table in the database: The table is mapped to the data access class that defines the data record type in the cache object of the graph instance.

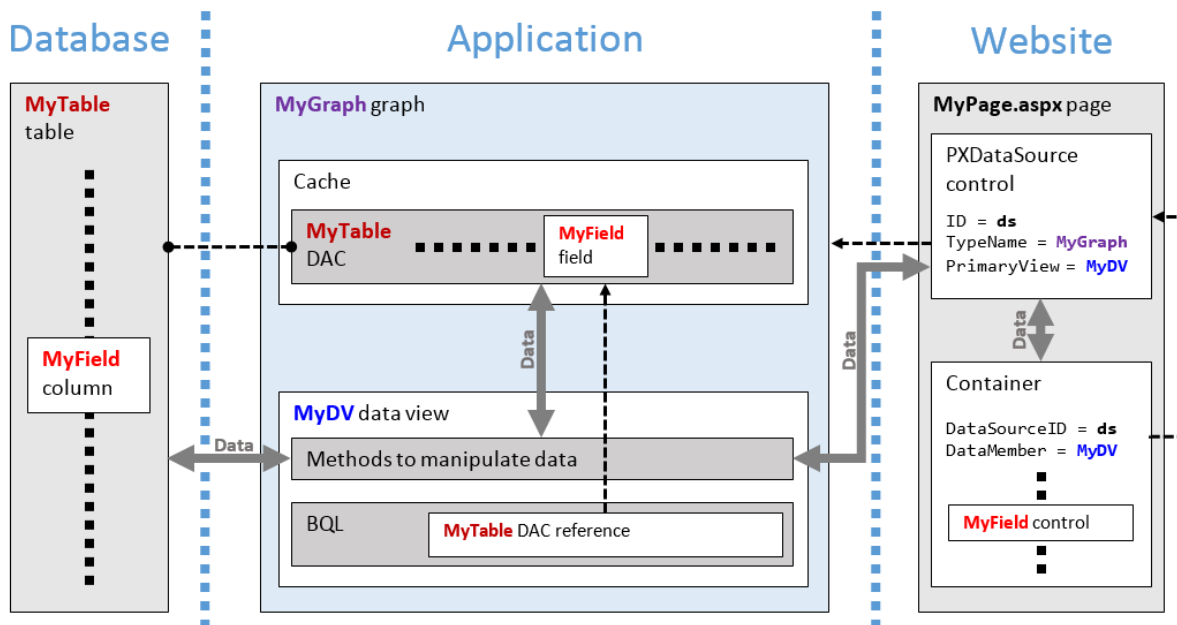


Figure: Objects required for a form that works with data from a database table

By using the Acumatica Customization Platform, you can create or customize each of the object types listed above.

Customization of ASPX Pages

To change the layout and behavior of an Acumatica ERP form, you must customize the corresponding ASPX page. However for customizing an ASPX page, the platform uses an approach that does not require you to change the original ASPX code of the Acumatica ERP form. Instead, the platform can apply the `.aspx` file with the same name from a special subfolder within the website folder, if this file

exists. At run time, while the platform is processing a request to open a form, the platform first tries to find the needed `.aspx` file inside this subfolder to use it instead of the original file. If the file with customized ASPX code is found, the platform opens the customized form. Otherwise, the original form is opened.

To cancel a customization of a page, you need only to delete the file with the appropriate name from the subfolder.

Customization of Application Classes (BLCs and DACs)

To provide the ability to customize the functionality or business logic of a form, the platform uses the technology based on class extensions. With this technology, to customize a BLC or DAC, the platform does not change the original code of Acumatica ERP. Instead, the platform uses an additional C# file for each class extension.

At run time, the platform automatically detects a class extension during the first initialization of the base (original) class. If an extension is found, the platform replaces the base class with the merged result of the base class and the extension that was found.

To cancel a customization of business logic, you need only to delete the file that contains the appropriate class extension.

This approach makes it easy to apply and cancel any customization of business logic in Acumatica ERP.

Customization of the Database Schema

The platform provides the following capabilities that you can use to customize the database schema:

- You can create an [SQL script](#) to execute while the customization is applied to an instance of Acumatica ERP.
- You can create a custom bound field to add the corresponding column to a table while the customization is applied to an instance of Acumatica ERP.
- You can define the `UpdateDatabase()` method of a class derived from the `CustomizationPlugin` class to execute an SQL script or a stored procedure from the C# code after the customization is applied to an instance of Acumatica ERP. (See [Custom Processes During Publication of a Customization](#) for details.)

In This Chapter

- [Changes in Webpages \(ASPX\)](#)
- [Changes in the Application Code \(C#\)](#)
- [Changes in the Database Schema](#)
- [Custom Processes During Publication of a Customization](#)

Changes in Webpages (ASPX)

To customize the look and behavior of an Acumatica ERP form, you need to change the ASPX code of the form. Because you need to include all the changes in a customization project, you have to perform the form customization by using the [Layout Editor](#) of the Customization Project Editor.

To collect all the changes that you make while you customize a form, the Layout Editor creates a *Page* item with a name that corresponds to the form ID, and includes the item in the currently selected customization project. This item contains XML code with instructions that have to be applied to the ASPX code of the form during the publication of the project.

For example, the following fragment of a *Page* item contains the XML code with the `<AddItem>` tag used to add the `UsrSIMCardID` field as a column to the `Transactions` grid.

```
<PXGridLevel DataMember="Transactions"
```

```

        ParentId="phG_tab_Items#0_grid_Levels#0"
        TypeFullName="PX.Web.UI.PXGridLevel">
    <Children Key="Columns">
        <AddItem>
            <PXGridColumn TypeFullName="PX.Web.UI.PXGridColumn">
                <Prop Key="DataField" Value="UsrSIMCardID" />
                <Prop Key="Width" Value="160" />
            </PXGridColumn>
        </AddItem>
    </Children>
</PXGridLevel>

```

In the XML code above, note that the `<Prop>` tag is used to set the `Width` property of the column to `160`.

With the Layout Editor, you can customize any object in the ASPX code of an Acumatica ERP form and save the resulting changeset to the customization project. To apply the customization to the website, you have to publish the project.

For example, at the publication process, the platform transforms the XML code fragment above to the following fragment of ASPX code.

```
<px:PXGridColumn DataField="UsrSIMCardID" Width="160" />
```

During the publication of the project, the platform applies the XML changeset to the appropriate form to create a customized version of the `.aspx` file with the same name in the `pages_xx` subfolder of the `CstPublished` folder of the website. If the form ID contains the `SO` prefix, the customized ASPX code is located in the `\CstPublished\pages_so` folder, as the following screenshot shows.

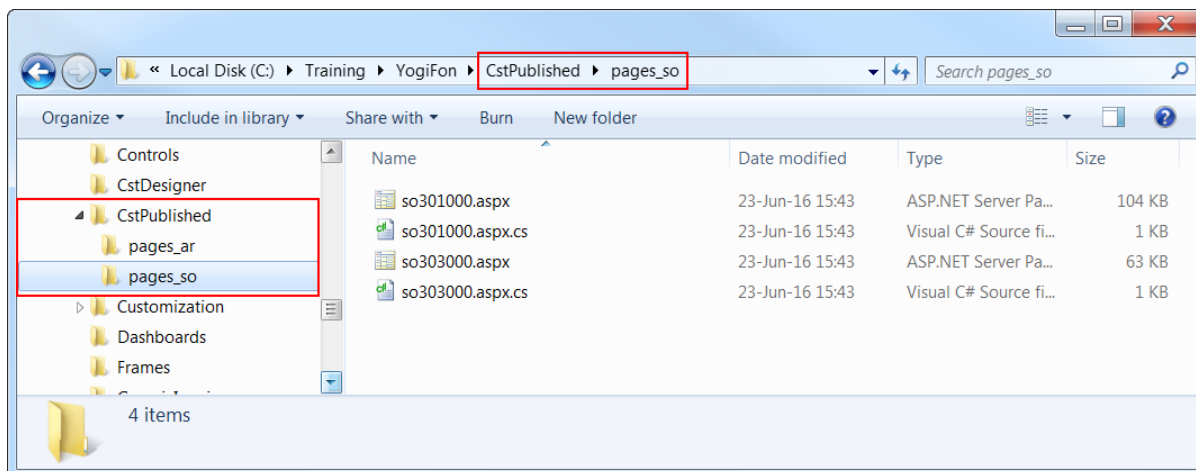


Figure: Viewing the files with customized ASPX code in the CstPublished folder

After the project has been published, when Acumatica ERP has to display a form, first it tries to find the `.aspx` file of the form in the `CstPublished` folder. If it finds it, Acumatica ERP opens the customized version of the form instead of the original one.

Any customization of Acumatica ERP can be unpublished. If you unpublish a form customization, the platform deletes the corresponding file in the `CstPublished` folder of the website.

Changes in the Application Code (C#)

In this section, you can find information about the customization of the application functionality, which is provided by the application business logic (implemented in C# code) of data access classes and business logic controllers.

BLC and DAC Extensions

To give you the ability to customize the functionality or behavior of a form, the Acumatica Customization Platform uses technology based on extension models. When you are resolving a typical

customization task, you generally create extensions for original data access classes and business logic controllers.

An extension for a business logic controller (BLC, also referred to as *graph*) or a data access class (DAC) is a class derived from a generic class defined in the `PX.Data` assembly of Acumatica ERP. To declare an extension for a DAC, you derive a class from the `PXCacheExtension<T>` generic class. To declare an extension of a BLC, you derive a class from the `PXGraphExtension<T>` generic class.

If you have created an extension for a BLC or DAC in a customization project and published the project, the platform applies the extension to the base class at run time. During publication of the project, all the code extensions created for an instance of Acumatica ERP in the project are saved as C# source code files in the `App_RuntimeCode` folder of the application instance (see the diagram below).

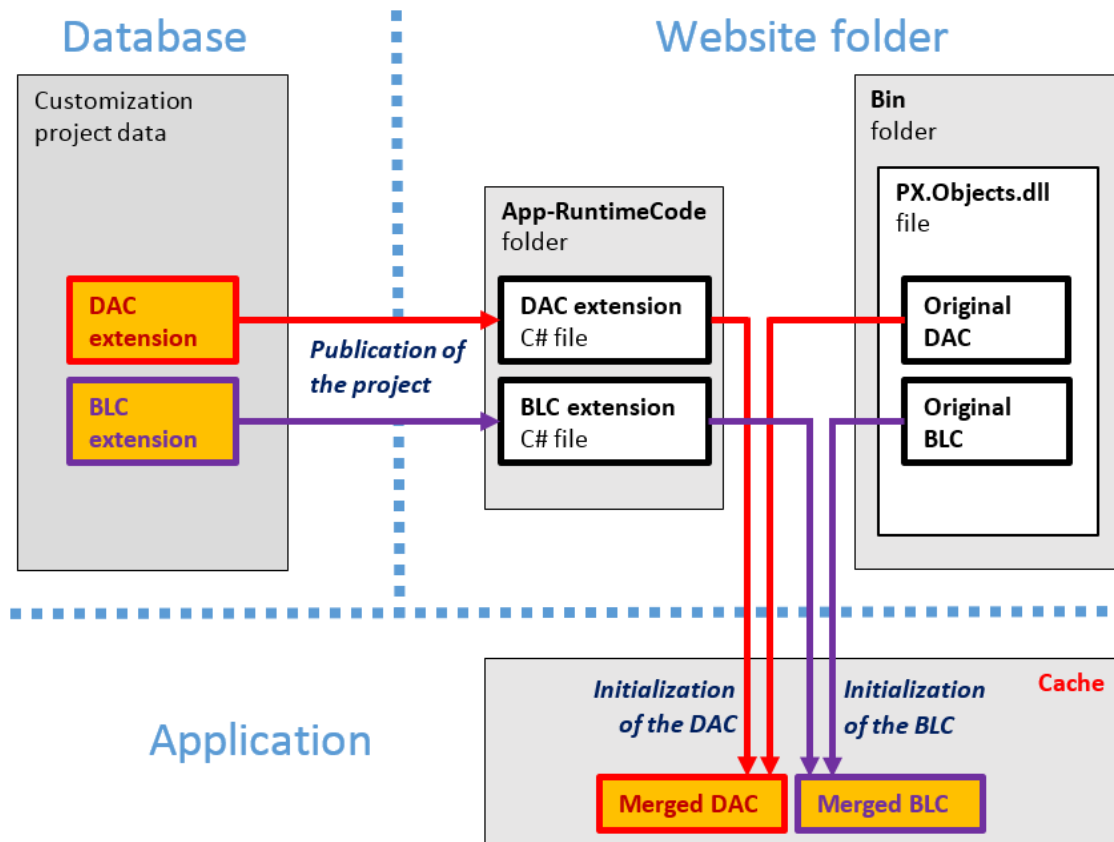


Figure: Use of DAC and BLC extensions for a customization

At run time, during the first initialization of a base class, the Acumatica Customization Platform automatically finds the extension for the class and applies the customization by replacing the base class with the merged result of the base class and the extension it found.

When you unpublish all customization projects, the platform deletes all the code files from the `App_RuntimeCode` folder. As a result, the platform has no custom code to merge at run time.

Multilevel Extensions

The Acumatica Customization Platform supports multilevel extensions, which are required when you develop off-the-shelf software that is distributed in multiple editions. Precompiled extensions provide a measure of protection for your source code and intellectual property.

The figure below illustrates the DAC extension model.


```

> PXCacheExtension<Table>
> PXCacheExtension<Extension1,Table>
> PXCacheExtension<Extension2,Extension1,Table>
> PXCacheExtension<Extension3,Extension2,Extension1,Table>
> PXCacheExtension<Extension4,Extension3,Extension2,Extension1,Table>
> PXCacheExtension<Extension5,Extension4,Extension3,Extension2,Extension1,Table>
> PXCacheExtension<Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Table>
> PXCacheExtension<Extension7,Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Table>
> PXCacheExtension<Extension8,Extension7,Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Table>
> PXCacheExtension<Extension9,Extension8,Extension7,Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Table>
> PXCacheExtensionAttribute

```

Figure: DAC extension levels

The figure below illustrates the BLC extension model.

```

> PXGraphExtension<Graph>
> PXGraphExtension<Extension1,Graph>
> PXGraphExtension<Extension2,Extension1,Graph>
> PXGraphExtension<Extension3,Extension2,Extension1,Graph>
> PXGraphExtension<Extension4,Extension3,Extension2,Extension1,Graph>
> PXGraphExtension<Extension5,Extension4,Extension3,Extension2,Extension1,Graph>
> PXGraphExtension<Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Graph>
> PXGraphExtension<Extension7,Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Graph>
> PXGraphExtension<Extension8,Extension7,Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Graph>
> PXGraphExtension<Extension9,Extension8,Extension7,Extension6,Extension5,Extension4,Extension3,Extension2,Extension1,Graph>

```

Figure: BLC extension levels

The platform can be used to customize the end production application. You can use multilevel extensions to develop applications that extend the functionality of Acumatica ERP or other software based on Acumatica Framework in multiple markets. You may have a base extension that contains the solution common to all markets as well as multiple market-specific extensions. Every market-specific solution is deployed along with the base extension. Moreover, you can later customize deployed extensions for the end user by using DAC and BLC extensions.



: When extensions can be deployed separately, the application developer should use multiple extension levels. Otherwise, we recommend using a single extension level.

The Order in Which Extensions Are Loaded

For each DAC or graph type, the system loads and applies extensions at run time as follows:

1. The system collects extensions for the DAC or graph type.
2. The system sorts the list of extensions in alphabetical order.
3. If there is a subscriber to the `PX.Data.PXBuildManager.SortExtentions` event, the system passes the list of extensions in alphabetical order to this subscriber, which can sort extensions in a custom way. A sample implementation of the subscriber is shown in the following code.

```

using System;
using Autofac;
using System.Web.Compilation;
using PX.Data.DependencyInjection;

namespace MyLib
{
    public class ServiceRegistration : Module

```

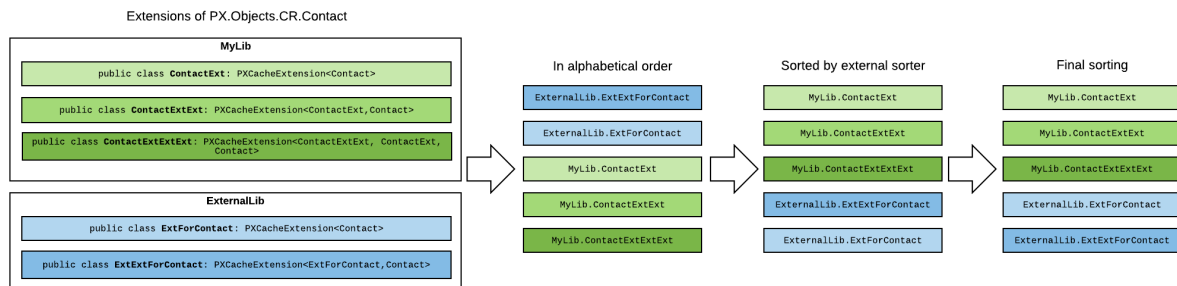
```

{
    protected override void Load(ContainerBuilder builder)
    {
        builder.ActivateOnApplicationStart<ExtensionSorting>();
    }
    private class ExtensionSorting
    {
        public ExtensionSorting()
        {
            PXBuildManager.SortExtensions += (list) =>
            {
                list.Sort((a, b) =>
                {
                    return -string.Compare(a.FullName, b.FullName,
                    StringComparison.InvariantCulture);
                } );
            };
        }
    }
}
}

```

- The system changes the order of dependent extensions (such as `DaclExt: PXCacheExtension<Dacl>` and `DaclExtExt: PXCacheExtension<DaclExt, Dacl>`) so that the higher-level extensions have higher priorities during the merge operation.

Suppose that the website folder contains five extensions of the `Contact` DAC that are available in the `MyLib` and `ExternalLib` namespaces, and the customization project includes an external sorter, which sorts the extensions in a custom way. The following diagram shows how these extensions are sorted. As a result of the sorting in this example, the first extension that is applied is `MyLib.ContactExt`. The `ExternalLib.ExtExtForContact` class has the highest priority during the merge operation.



In This Section

- [DAC Extensions](#)
- [Graph Extensions](#)
- [Run-Time Compilation](#)
- [Extension Library](#)

DAC Extensions

This topic explores the ways provided by the Acumatica Customization Platform to define data access class (DAC) extensions of different levels and shows how different DAC extensions can interact.

To declare a DAC extension, you derive a class from `PXCacheExtension<T>`.

First-Level DAC Extension

The example below shows a declaration of a first-level DAC extension.

```
class BaseDACExtension : PXCacheExtension<BaseDAC>
{
    public void SomeMethod()
    {
        BaseDAC dac = Base;
    }
}
```

The extension class includes the read-only `Base` property, which returns an instance of the base DAC.

Second-Level DAC Extension

The example below shows a declaration of a second-level DAC extension.

```
class BaseDACExtensionOnExtension :
    PXCacheExtension<BaseDACExtension, BaseDAC>
{
    public void SomeMethod()
    {
        BaseDAC dac = Base;
        BaseDACExtension dacExt = Base1;
    }
}
```

The extension class includes the following:

- The read-only `Base` property, which returns the instance of the base DAC
- The read-only `Base1` property, which returns the instance of the first-level DAC extension

Two Variants of a Higher-Level DAC Extension

A definition of an extension of a higher level has two possible variants. In the first variant, you derive the extension class from the `PXCacheExtension` generic class with two type parameters, where the first type parameter is set to an extension of the previous level. In the second variant, you derive the extension class from the `PXCacheExtension` generic class with the same number of type parameters as the level of the extension of the new class. In this case, you set type parameters to extension classes from all lower extension levels, from the previous level down to the base class.

First Variant of a Higher-Level DAC Extension

The example below shows the declaration of a third- or higher-level DAC extension that derives from the `PXCacheExtension` generic class with two type parameters.

```
class BaseDACMultiExtensionOnExtension :
    PXCacheExtension<BaseDACExtensionOnExtension, BaseDAC>
{
    public void SomeMethod()
    {
        BaseDAC dac = Base;
        BaseDACExtensionOnExtension dacExtOnExt = Base1;
    }
}
```

An extension class defined this way includes the following:

- The read-only `Base` property, which returns the instance of the base DAC

- The read-only `Base1` property, which returns the instance of the DAC extension from the previous level

Second Variant of a Higher-Level DAC Extension

The example below shows a declaration of a third- or higher-level DAC extension that derives from the `PXCacheExtension` generic class with three or more type parameters.

```
class BaseDACAdvMultiExtensionOnExtension :
    PXCacheExtension<BaseDACExtensionOnExtension, BaseDACExtension, BaseDAC>
{
    public void SomeMethod()
    {
        BaseDAC dac = Base;
        BaseDACExtension dacExt = Base1;
        BaseDACExtensionOnExtension dacExtOnExt = Base2;
    }
}
```

An extension class defined in this way includes the following:

- The read-only `Base` property, which returns the instance of the base DAC
- The read-only `BaseN` properties for all extension levels below the current level, where *N* is the sequence number of an extension level

Attributes Hierarchy of a DAC Field

Conceptually, a DAC extension is a substitution of the base DAC. The base DAC is replaced at run time with the merged result of the base DAC and every extension discovered.

Each extension on another extension completely overrides the DAC field attributes. The attributes declared on the highest-level extension override all previously declared attributes.

For example, suppose that the attributes of the same DAC field are defined on different extension levels as follows:

- In the base DAC, as shown below

```
[PXUIField(DisplayName = "Base Name")]
[PXDefault("Default base value")]
```

- In the first-level DAC extension, as the following code shows

```
[PXUIField(DisplayName = "Level1 Name", Visible = false)]
```

- In the second-level DAC extension, as shown below

```
[PXUIField(DisplayName = "Level2 Name")]
[PXDefault("Default 2nd level value")]
```

If all the extensions are applied, any field of the second-level DAC extension will have the following parameters specified by attributes:

- Display name: *Level2 Name*
- Visible: *true*
- Default value: *Default 2nd level value*



: You cannot modify the same member of the base DAC in the DAC extensions of the same level. You should use DAC extensions of different levels that successively extend one another.

DAC Field Read and Write Operations

The Acumatica Framework can access the value of a DAC field only through an instance of the base DAC. The instance of the DAC extension is used to get the field value only when the field is declared in the extension and is not present in the base DAC.

To assign a value to a field, the application developer should use the instance of the base DAC. You must use the instance of a DAC extension to assign a field value when the field is declared within the extension and is not present in the base DAC.

In This Section

- [Access to a Custom Field](#)
- [Customization of Field Attributes in DAC Extensions](#)
- [Supported DAC Extension Formats](#)

Access to a Custom Field

You can customize a data access class (DAC) in either of the following ways (see the diagram below):

- By altering the attributes of existing fields. You can use an altered field just as you would any other existing field.
- By declaring new (custom) fields.

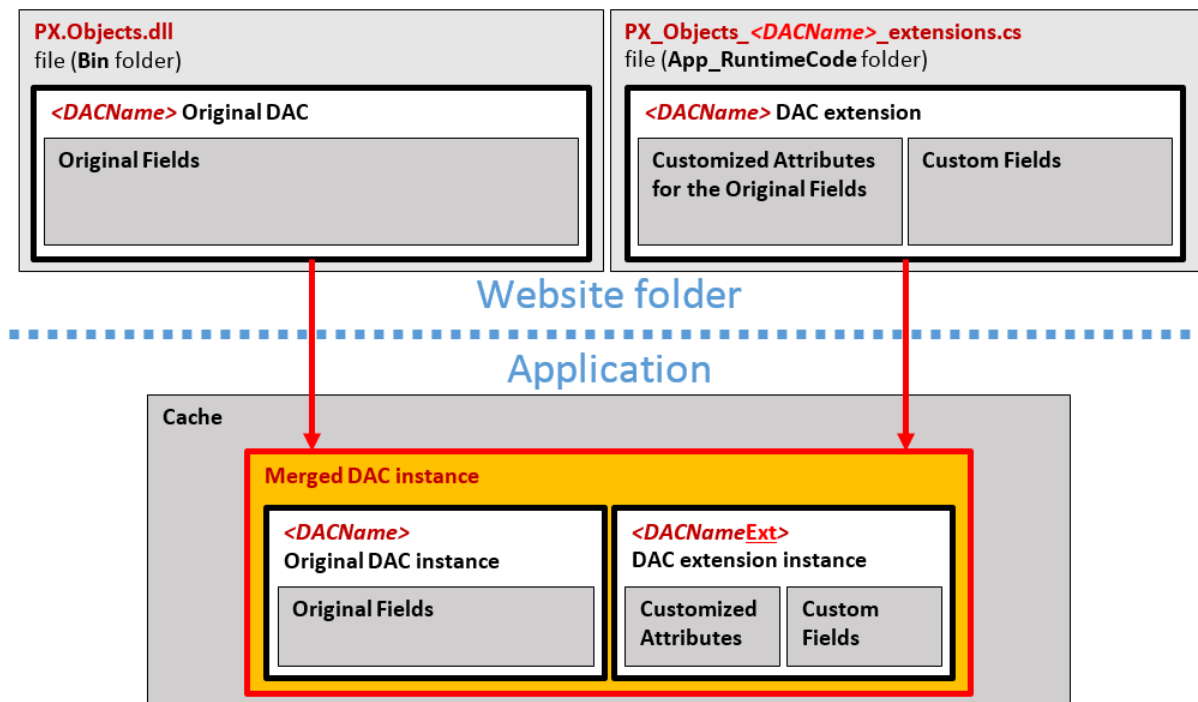


Figure: Analyzing the content of the merged DAC instance in the cache object

Every custom field is declared within the code of a DAC extension; therefore, at run time, the custom field is accessible only through the DAC extension instance of the cache object.

You can access a custom field:

- [From a Method](#)
- [From a BQL Statement](#)
- [From a Field Attribute](#)

From a Method

You can access the extension instance through the base (original) DAC object by using one of the following generic methods:

- The `GetExtension<T>(object)` static generic method of the `PXCache<T>` generic class
- The `GetExtension<T>(object)` static generic method declared within the non-generic `PXCache` class



: There are no differences between these generic methods. You can use either one.

For example, you might access an instance of a DAC extension as follows. This example uses the `row` instance of the base DAC to retrieve the `rowExt` DAC extension object.

```
DACNameExt rowExt = PXCache<DACName>.GetExtension<DACNameExt>(row);
```

To access a custom field, use the instance of a DAC extension that contains the field as follows. The code below illustrates the use of the `GetExtension<T>(object)` static generic method of the `PXCache<T>` generic class to access an instance of a DAC extension used to access a custom field.

```
var fieldValue = PXCache<DACName>.GetExtension<DACNameExt>(row).UsrFieldName;
```

You can retrieve a DAC extension object or particular field from it by using the methods of `PXCache`, as shown in the code below. This code demonstrates how you can access a field by using the `GetExtension<>` method.

```
//Access to the field through the GetExtension<> method
//localCategory gets the value of the usrLocalTaxCategoryID custom field
//defined in the InventoryItemExtension DAC extension for the InventoryItem class
protected void InventoryItem_RowUpdating(PXCache sender, PXRowUpdatingEventArgs e)
{
    InventoryItem row = e.NewRow as InventoryItem;
    InventoryItemExtension rowExt =
    sender.GetExtension<InventoryItemExtension>(row);
    string localCategory = rowExt.UsrLocalTaxCategoryID;
    ...
}
```

In event handlers, you can also use the `GetValue()` and `GetValueExt()` methods to access the custom field by its string name, as the code below shows.

```
//Access to the field through the GetValue() method
//localCategory gets the value of the usrLocalTaxCategoryID custom field
//defined in a DAC extension for the InventoryItem class
protected void InventoryItem_RowUpdating(PXCache sender, PXRowUpdatingEventArgs e)
{
    InventoryItem row = e.NewRow as InventoryItem;
    string localCategory = (string)sender.GetValue(row, "usrLocalTaxCategoryID");
    ...
}
```

For custom fields defined in DAC extensions, you can also use other methods of `PXCache` classes, such as `SetValue()` and `SetDefaultExt()`. For more information, see the `PXCache<Table>` class in the API Reference Documentation of Acumatica Framework.

You can refer to the custom field by its BQL name in any methods of Acumatica Framework. The example below shows how you can refer to the field in the `SetEnabled<T>()` method of the `PXUIFieldAttribute` that you can use to configure the UI representation of the field at run time. For more information on available classes and their methods, see the API Reference Documentation of Acumatica Framework.

```
//The usrLocalTaxCategoryID custom field is defined
//in the InventoryItemExtension DAC extension for the InventoryItem class
protected void InventoryItem_RowSelected(PXCache sender, PXRowSelectedEventArgs e)
```

```
{
    InventoryItem row = e.Row as InventoryItem;

    PXUIFieldAttribute.SetEnabled<InventoryItemExtension.usrLocalTaxCategoryID>(sender,
    row, true);
    ...
}
```

From a BQL Statement

In BQL expressions, you have to refer to the custom field by its BQL name in the extension class.

For example, after you have added the `UsrSearchKeywords` field to the `InventoryItemExtension` class, which is a DAC extension for the `InventoryItem` class, the field is accessible in BQL, as shown in the code below.

```
//The usrLocalTaxCategoryID custom field is defined
//in the InventoryItemExtension DAC extension for the InventoryItem class
PXSelect<InventoryItemExtension,
    Where<InventoryItemExtension.usrLocalTaxCategoryID, IsNotNull>>
```

From a Field Attribute

In DAC extensions, you have to refer to a custom field by its BQL name.

After you have added the `UsrSearchKeywords` field to the `InventoryItemExtension` class, which is a DAC extension for the `InventoryItem` class, the field is accessible in other DAC extensions.

For example, if you need to specify the type of the custom DAC field. For example, in attributes, you can use the `typeof` operator, as shown in the code below.

```
//The usrLocalTaxCategoryID custom field is defined
//in the InventoryItemExtension DAC extension for the InventoryItem class
[PXSelector(typeof(InventoryItemExtension.usrLocalTaxCategoryID))]
```

Customization of Field Attributes in DAC Extensions

If you have a customization that replaces original attributes of a field with custom attributes, after upgrading Acumatica ERP to a new version, a new functionality may become unavailable, as the following diagram shows.

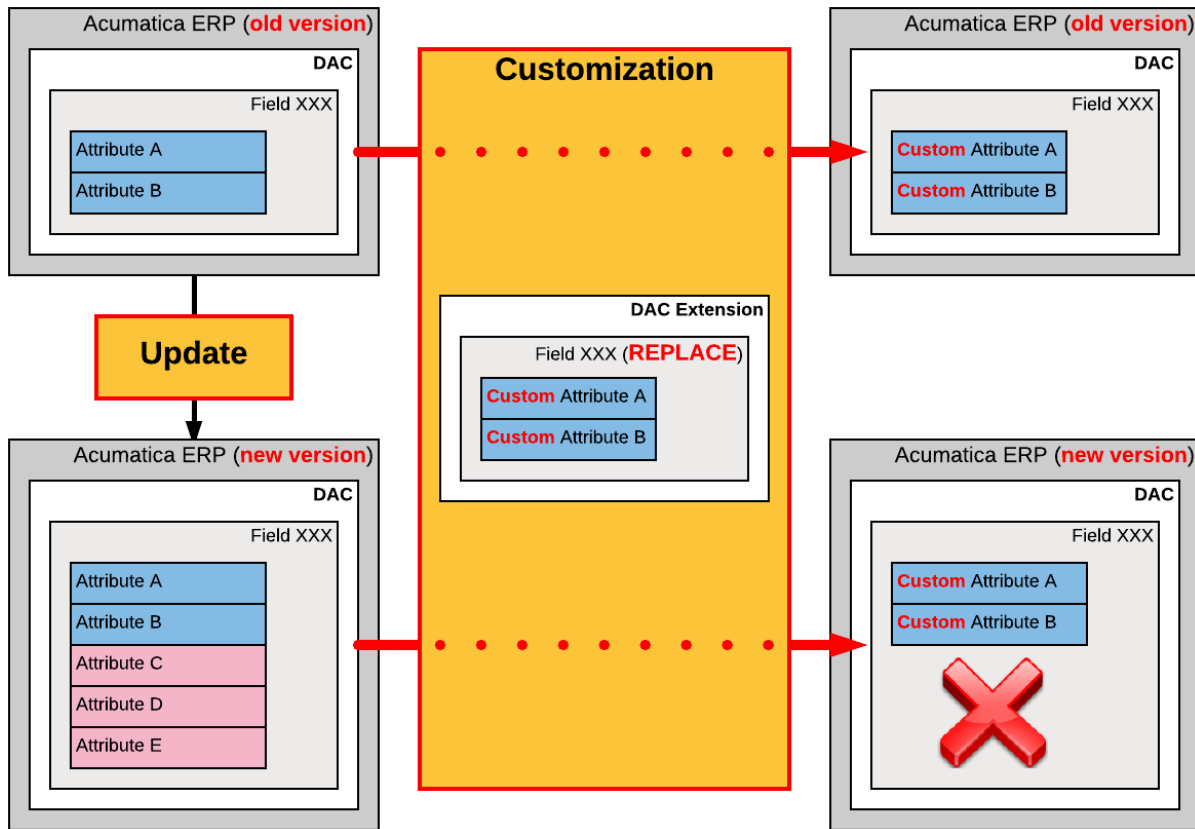


Figure: Possible result of using the Replace (default) method to customize the attributes of a DAC field

To solve this issue, the customization framework provides advanced possibilities to control the field customization by using additional attributes in the DAC extension.

When you customize Acumatica ERP, you can specify how the system should apply the original and custom attributes to the field. Thus you can make the customizations more flexible and use the collections of original attributes that could be updated between Acumatica ERP versions.

To specify the way the system should apply the field attributes in a DAC extension, you can use the following attributes.

Attribute	Description
PXMergeAttributes	Specifies how to apply custom attributes to the existing ones.
PXRemoveBaseAttribute	Removes the specified existing attribute.
PXCustomizeBaseAttribute	Defines a new value for the specified attribute parameter.
PXCustomizeSelectorColumns	Defines the new set and order of the columns in the selector.

PXMergeAttributes Attribute

You use the `PXMergeAttributes` attribute to specify for each field how to apply custom attributes to the existing ones.

There are three options to specify the operation: `Append`, `Replace`, and `Merge`. These options are declared by the `MergeMethod` enumerator as follows.

```
public enum MergeMethod { Append, Replace, Merge }
```

The `Append` option is used to add custom attributes to the existing ones.

The `Replace` option forces the system to use custom attributes instead of the existing ones. This option is used by default if you do not specify the `PXMergeAttributes` attribute on the customized field.

The `Merge` option makes the system apply the union of the custom and existing attributes to the customized field. The resulting collection of attributes includes:

- An existing attribute if the custom attribute collection does not contain the same attribute.
- Each attribute of the custom attribute collection.

To define a merge method for a field, insert the `PXMergeAttributes` attribute of the field in the DAC extension, and define the option as the value of the `Method` parameter, as shown in the examples below.

For example, suppose that for the customized `FieldName` field, you have to add the `PXDefault` attribute and change the value in the `PXUIFieldAttribute`. Suppose the original code of the `FieldName` field is the following.

```
#region FieldName
...
[PXDBDecimal(2)]
[PXUIField(DisplayName = "Display Name")]
public virtual decimal? FieldName
{...}
#endregion
```

In the DAC extension code for the customized field, specify the `Merge` option, as shown below.

```
public class DACName_Extension: PXCacheExtension<DACName>
{
    #region FieldName

    [PXMergeAttributes(Method = MergeMethod.Merge)]

    [PXDefault(TypeCode.Decimal, "0.0")]
    [PXUIField(DisplayName = "Name")]

    public string FieldName{get;set;}

    #endregion
}
```

The system will merge attributes for the `FieldName` field. In the result of the merge, the `FieldName` field will have the following collection of attributes.

```
...
[PXDBDecimal(2)]
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Name")]
public virtual decimal? FieldName
{...}
```

PXRemoveBaseAttribute Attribute

The `PXRemoveBaseAttribute` attribute is added to the field to remove the specified attribute of the field.

For example, to remove the `PXUIField` attribute of the field, add the following attribute to the field region of the DAC extension.

```
[PXRemoveBaseAttribute(typeof(PXUIFieldAttribute))]
```

PXCustomizeBaseAttribute Attribute

The `PXCustomizeBaseAttribute` attribute is added to the field for each modified parameter.

For example, to set the `Enabled` parameter of the `PXUIField` attribute of a field to `false`, add the following attribute to the field region of the DAC extension.

```
[PXCustomizeBaseAttribute(typeof(PXUIFieldAttribute), "Enabled", false)]
```

The following example shows how to change the `Required` parameter of the `PXUIField` attribute for the `MyField` field of the `MyDAC` data access class in the `MyGraph` graph extension by using the `CacheAttached()` event handler.

```
public class MyGraph_Extension:PXGraphExtension<MyGraph>
{
    ...
    [PXCustomizeBaseAttribute(typeof(PXUIFieldAttribute), "Required", true)]
    protected void MyDAC_MyField_CacheAttached(PXCache cache)
    {
    }
    ...
}
```

PXCustomizeSelectorColumns Attribute

This attribute defines a new set and order of the columns in the specified selector.

For example, suppose that you have to add a new column to the `ItemClassID` selector, which is located on the **General Settings** tab of the Stock Items form. The original attributes of the field are the following.

```
#region ItemClassID
public abstract class itemClassID : PX.Data.IBqlField
{
}
protected String _ItemClassID;

[PXDBString(10, IsUnicode = true)]
[PXUIField(DisplayName = "Item Class", Visibility =
PXUIVisibility.SelectorVisible)]
[PXSelector(typeof(Search<INItemClass.itemClassID>))]
public virtual String ItemClassID
{
    get { return this._ItemClassID; }
    set { this._ItemClassID = value; }
}
#endregion
```

The `ItemClassID` selector contains two columns for the `INItemClass.itemClassID` and `INItemClass.descr` fields.

The code below shows how to customize only the `PXSelector` attribute of the field if you need to insert a new column for the `INItemClass.baseUnit` field between the existing columns.

```
public class IN_InventoryItem_Extension:
PXCacheExtension<PX.Objects.IN.InventoryItem>
{
    #region ItemClassID
    [PXMergeAttributes(Method = MergeMethod.Append)]
    [PXCustomizeSelectorColumns(
        typeof(PX.Objects.IN.INItemClass.itemClassID),
        typeof(PX.Objects.IN.INItemClass.baseUnit),
        typeof(PX.Objects.IN.INItemClass.descr))]
    #endregion
}
```

The system will append attributes for the `ItemClassID` field because of the `Append` method in the `PXMergeAttributes` specified in the DAC extension. In the result of the specified merge method, the `ItemClassID` field has all original attributes, and the `PXCustomizeSelectorColumns` attribute is

added through the DAC extension. The selector control of the `ItemClassID` field has three columns, as specified in the `PXCustomizeSelectorColumns` attribute.

Supported DAC Extension Formats

You can keep a DAC customization in a customization project in any of the following ways:

- As a *DAC* item of a customization project in XML format. When you publish the project, the platform creates the DAC extension source code and saves the code in the `<DACItemName>_extensions.cs` file in the `App_RuntimeCode` folder of the website.
 - ☰ : The system creates the file name based on the item name by replacing all symbols except letters and digits with the `_` symbol and adding the `_extensions` suffix. For example, the extension code file for the `PX.Objects.CR.Contact` DAC item has the name `PX_Objects_CR_Contact_extensions.cs`.
- As a *Code* item of a customization project that is the C# code of the DAC extension wrapped in XML format. When you publish the project, the platform saves the code in the `<CodeItemName>.cs` file in the `App_RuntimeCode` folder of the website.
- As a `.dll` file of the extension library in the `Bin` folder of the website. The library contains the binary code of the DAC extension. To deploy the extension library to the production environment along with a customization project, you include the extension library in the project as a *File* item. See [Extension Library](#) for details.

By using the [Customization Project Editor](#), you can:

- Convert a *DAC* item to a *Code* item.
 - ☰ : The system creates the *Code* name based on the *DAC* item name by using the last word of the name and adding the `Extensions` suffix. For example, the `PX.Objects.CR.Contact` DAC item is converted to the `ContactExtensions` *Code* item.
- Move a *Code* item to the extension library included in the customization project as a *File* item.

If you have a customized data access class that is added to the project as a *DAC* item, then you can click **Convert to Extension** on the page toolbar of the Customized Data Classes page to convert the class changes into the class extension code used to complete the extension development in either the Code Editor or Microsoft Visual Studio (see the diagram below).

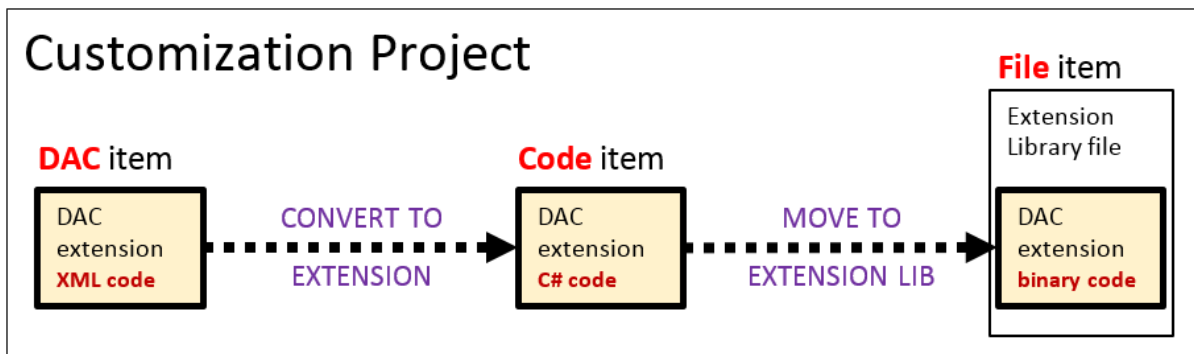


Figure: Converting the code format of a DAC extension in a customization project

For a *Code* item included in a customization project, if you want to move the code to an extension library to compile it into a `.dll` file, you can click the **Move to Extension Lib** button on the page toolbar of the [Code Editor](#). After the operation is complete, the *Code* item that is currently displayed in the work area of the Code Editor is removed from the customization project. The file with the same source code is appended to the extension library that is bound to the customization project. If the *Code* item name was `CodeItemName`, the name of the created file will be `CodeItemName.cs`.

Graph Extensions

This topic explores the ways provided by the Acumatica Customization Platform to define business logic controller (BLC, also referred as *graph*) extensions of different levels, and shows how different BLC extensions can interact.

To declare an extension of a BLC, you derive a class from `PXGraphExtension<T>`.

First-Level BLC Extension

The example below shows a declaration of a first-level BLC extension.

```
class BaseBLCEExtension : PXGraphExtension<BaseBLC>
{
    public void SomeMethod()
    {
        BaseBLC baseBLC = Base;
    }
}
```

The extension class includes the read-only `Base` property, which returns an instance of the base BLC.

Second-Level BLC Extension

The example below shows a declaration of a second-level BLC extension.

```
class BaseBLCEExtensionOnExtension :
    PXGraphExtension<BaseBLCEExtension, BaseBLC>
{
    public void SomeMethod()
    {
        BaseBLC baseBLC = Base;
        BaseBLCEExtension ext = Base1;
    }
}
```

The extension class includes the following:

- The read-only `Base` property, which returns the instance of the base BLC
- The read-only `Base1` property, which returns the instance of the first-level BLC extension

Two Variants of a Higher-Level BLC Extension

A definition of a higher-level BLC extension has two possible variants. In the first variant, you derive the extension class from the `PXGraphExtension` generic class with two type parameters, where the first type parameter is set to an extension of the previous level. In the second variant, you derive the extension class from the `PXGraphExtension` generic class with the same number of type parameters as the level of the extension of the new class. In this case, you set type parameters to extension classes from all lower extension levels, from the previous level down to the base class.

First Variant of a Higher-Level BLC Extension

The example below shows a declaration of a third- or higher-level BLC extension that is derived from the `PXGraphExtension` generic class with two type parameters.

```
class BaseBLCMultiExtensionOnExtension :
    PXGraphExtension<BaseBLCEExtensionOnExtension, BaseBLC>
{
    public void SomeMethod()
    {
        BaseBLC BLC = Base;
        BaseBLCEExtensionOnExtension prevExt = Base1;
    }
}
```

An extension class defined in this way includes the following:

- The read-only `Base` property, which returns the instance of the base BLC

- The read-only `Base1` property, which returns the instance of the BLC extension from the previous level

Second Variant of a Higher-Level BLC Extension

The example below shows a declaration of a third- or higher-level BLC extension that is derived from the `PXGraphExtension` generic class with three or more type parameters.

```
class BaseBLCAdvMultiExtensionOnExtension :
    PXGraphExtension<BaseBLCExtensionOnExtension, BaseBLCExtension, BaseBLC>
{
    public void SomeMethod()
    {
        BaseBLC BLC = Base;
        BaseBLCExtension ext = Base1;
        BaseBLCExtensionOnExtension extOnExt = Base2;
    }
}
```

An extension class defined in this way includes the following:

- The read-only `Base` property, which returns the instance of the base BLC
- The read-only `BaseN` properties for all extension levels below the current level, where *N* is the sequence number of an extension level

Conceptually, a BLC extension is a substitution of the base BLC. The base BLC is replaced at run time with the merged result of the base BLC and every extension the platform found. The higher level of a declaration an extension has, the higher priority it obtains in the merge operation.

In This Section

- [Event Handlers](#)
- [Customization of a Data View](#)
- [Customization of an Action](#)
- [Override of a Method](#)

Event Handlers

The business logic associated with data modifications is implemented through event handlers. Event handlers are methods that are executed when the `PXCache` objects of a particular data access class (DAC) raise data manipulation events.

Every business logic controller (BLC, also referred to as *graph*) instance has a collection of event handlers for each type of data manipulation event. Every collection is filled automatically with event subscribers that are declared within the base (original) BLC and that meet the naming conventions of Acumatica Framework event handlers.

With Acumatica Customization Platform, you can define new event handlers within BLC extensions. You can define an event handler in two possible ways:

- You define the event handler in the same way as it is defined in the base BLC. As a result, the event handler is added to the appropriate event handler collection. Depending on the event type, the event handler is added to either the end of the collection or the start of it. When the event occurs, all event handlers in the collection are executed, from the first to the last one.
- You define the event handler with an additional parameter, which represents the delegate for one of the following:
 - The event handler with an additional parameter from the extension of the previous level, if such an event handler exists.

- The first item in the collection of event handlers, if no handlers with additional parameters declared within lower-level extensions exist. The collection contains event handlers without the additional parameter from extensions discovered at all levels.

In either case, you can decide whether to invoke the delegate.



: The `CacheAttached()` event handler declared in the highest-level BLC extension is used to replace base DAC field attributes. Attributes attached to the `CacheAttached()` event handlers within the base BLC or its extensions are attached to the `PXCache` object, each time completely replacing the previous ones, from the base BLC to the highest extension discovered.

Event Handler Added to the End of the Collection

The following event handlers are added to the end of the collection:

- `FieldUpdated(PXCache sender, PXFieldUpdatedEventArgs e)`
- `RowSelecting(PXCache sender, PXRowSelectingEventArgs e)`
- `RowSelected(PXCache sender, PXRowSelectedEventArgs e)`
- `RowInserted(PXCache sender, PXRowInsertedEventArgs e)`
- `RowUpdated(PXCache sender, PXRowUpdatedEventArgs e)`
- `RowDeleted(PXCache sender, PXRowDeletedEventArgs e)`
- `RowPersisted(PXCache sender, PXRowPersistedEventArgs e)`

The system executes event handlers from the base (original) event handler up to the highest extension level (referred to as the *bubbling strategy*). The lower the BLC extension's level of declaration, the earlier the event subscriber is called. The figure below illustrates this principle.

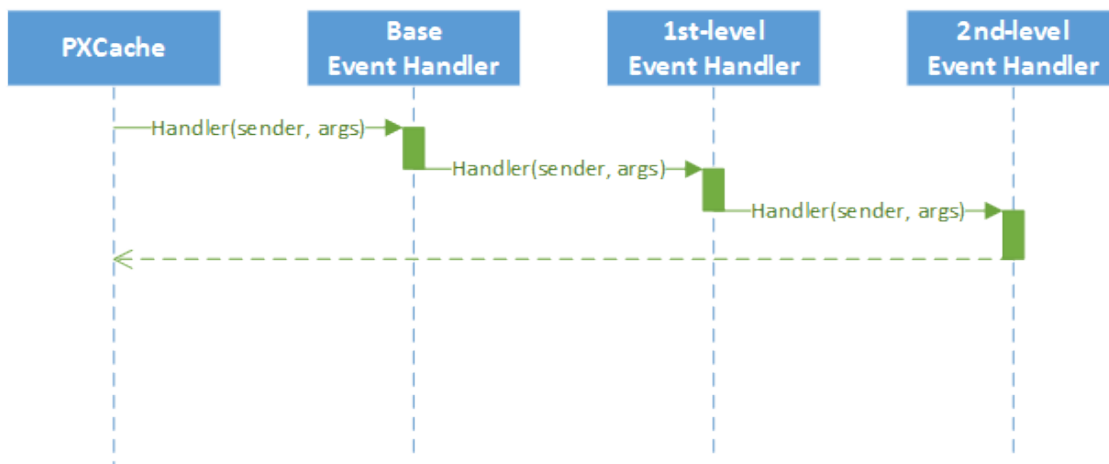


Figure: Event execution with the bubbling strategy

Event Handlers Added to the Beginning of the Collection

The following event handlers are added to the beginning of the collection:

- `FieldSelecting(PXCache sender, PXFieldSelectingEventArgs e)`
- `FieldDefaulting(PXCache sender, PXFieldDefaultingEventArgs e)`
- `FieldUpdating(PXCache sender, PXFieldUpdatingEventArgs e)`
- `FieldVerifying(PXCache sender, PXFieldVerifyingEventArgs e)`
- `RowInserting(PXCache sender, PXRowInsertingEventArgs e)`

- RowUpdating(PXCache sender, PXRowUpdatingEventArgs e)
- RowDeleting(PXCache sender, PXRowDeletingEventArgs e)
- RowPersisting(PXCache sender, PXRowPersistingEventArgs e)
- CommandPreparing(PXCache sender, PXCommandPreparingEventArgs e)
- ExceptionHandling(PXCache sender, PXExceptionHandlingEventArgs e)

Event handlers are added by the system to the beginning of the collection by using the *tunneling strategy*. The system executes event handlers from the highest extension level down to the base event handler. The higher the BLC extension's level of declaration, the earlier the event subscriber is called. The figure below illustrates this principle.

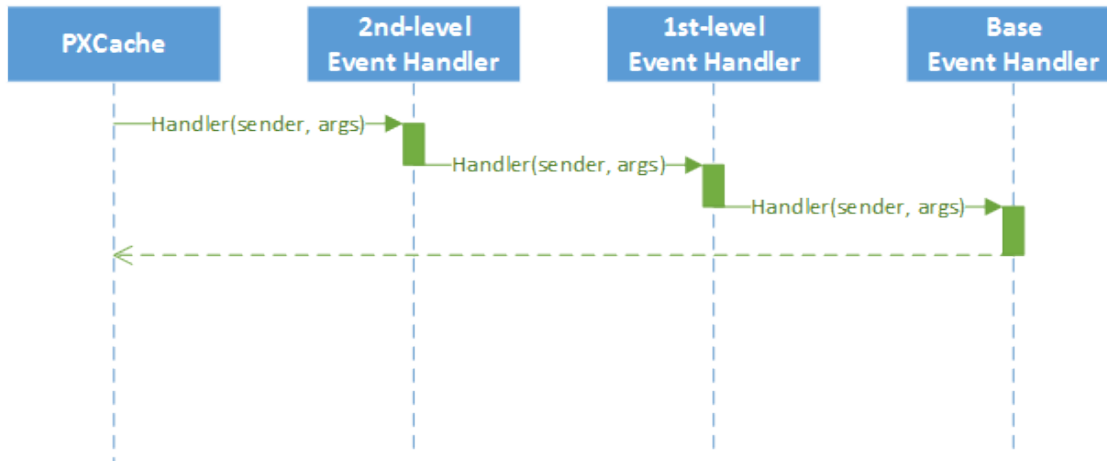


Figure: Event execution with the tunneling strategy

Event Handlers with an Additional Parameter

The event handler with an additional parameter replaces the base BLC event handler collection. When the event is raised, the system calls the event handler with an additional parameter of the highest-level BLC extension. The system passes a link to the event handler with an additional parameter from the extension of the previous level, if such an event handler exists, or to the first item in the event handler collection. You use a delegate as an additional parameter to encapsulate the appropriate event handler.

The Acumatica Framework provides the following delegates to encapsulate event handlers:

- PXFieldSelecting(PXCache sender, PXFieldSelectingEventArgs e)
- PXFieldDefaulting(PXCache sender, PXFieldDefaultingEventArgs e)
- PXFieldUpdating(PXCache sender, PXFieldUpdatingEventArgs e)
- PXFieldVerifying(PXCache sender, PXFieldVerifyingEventArgs e)
- PXFieldUpdated(PXCache sender, PXFieldUpdatedEventArgs e)
- PXRowSelecting(PXCache sender, PXRowSelectingEventArgs e)
- PXRowSelected(PXCache sender, PXRowSelectedEventArgs e)
- PXRowInserting(PXCache sender, PXRowInsertingEventArgs e)
- PXRowInserted(PXCache sender, PXRowInsertedEventArgs e)
- PXRowUpdating(PXCache sender, PXRowUpdatingEventArgs e)
- PXRowUpdated(PXCache sender, PXRowUpdatedEventArgs e)

- `PXRowDeleting(PXCache sender, PXRowDeletingEventArgs e)`
- `PXRowDeleted(PXCache sender, PXRowDeletedEventArgs e)`
- `PXRowPersisting(PXCache sender, PXRowPersistingEventArgs e)`
- `PXRowPersisted(PXCache sender, PXRowPersistedEventArgs e)`
- `PXCommandPreparing(PXCache sender, PXCommandPreparingEventArgs e)`
- `PXExceptionHandling(PXCache sender, PXExceptionHandlingEventArgs e)`

For example, for the `FieldVerifying` event, the event handler with an additional parameter looks like `FieldVerifying(PXCache sender, PXFieldVerifyingEventArgs e, PXFieldVerifying del)`.

You can execute `del()` to invoke the event handler to which `del` points, or you can decide not to invoke it. When `del()` points to the base BLC event handler collection, its invocation causes the execution of the whole collection. All other event handlers in the collection are invoked sequentially after the first handler is executed.

Suppose that you have declared event handlers as follows.

```
public class BaseBLC : PXGraph<BaseBLC, DAC>
{
    protected void DAC_RowUpdated(PXCache cache, PXRowUpdatedEventArgs e)
    {

    }

    protected void DAC_Field_FieldVerifying(PXCache sender,
PXFieldVerifyingEventArgs e)
    {

    }
}

public class BaseBLCExt : PXGraphExtension<BaseBLC>
{
    protected void DAC_RowUpdated(PXCache cache, PXRowUpdatedEventArgs e)
    {

    }

    protected void DAC_Field_FieldVerifying(PXCache sender,
PXFieldVerifyingEventArgs e)
    {

    }

    protected void DAC_RowUpdated(PXCache cache, PXRowUpdatedEventArgs e,
PXRowUpdated del)
    {
        if (del != null)
            del(sender, e);
    }

    protected void DAC_Field_FieldVerifying(PXCache sender,
PXFieldVerifyingEventArgs e, PXFieldVerifying del)
    {
        if (del != null)
            del(sender, e);
    }
}

public class BaseBLCExtOnExt : PXGraphExtension<BaseBLCExt, BaseBLC>
{
    protected void DAC_RowUpdated(PXCache cache, PXRowUpdatedEventArgs e)
```



```

{
}

protected void DAC_Field_FieldVerifying(PXCache sender,
PXFieldVerifyingEventArgs e)
{
}

protected void DAC_RowUpdated(PXCache cache, PXRowUpdatedEventArgs e,
PXRowUpdated del)
{
    if (del != null)
        del(sender, e);
}

protected void DAC_Field_FieldVerifying(PXCache sender,
PXFieldVerifyingEventArgs e, PXFieldVerifying del)
{
    if (del != null)
        del(sender, e);
}
}

```

In this case, the `RowUpdated` and `FieldVerifying` event handlers are invoked in the appropriate sequences, explained below.

The following figure illustrates the order in which the `RowUpdated` events are invoked.

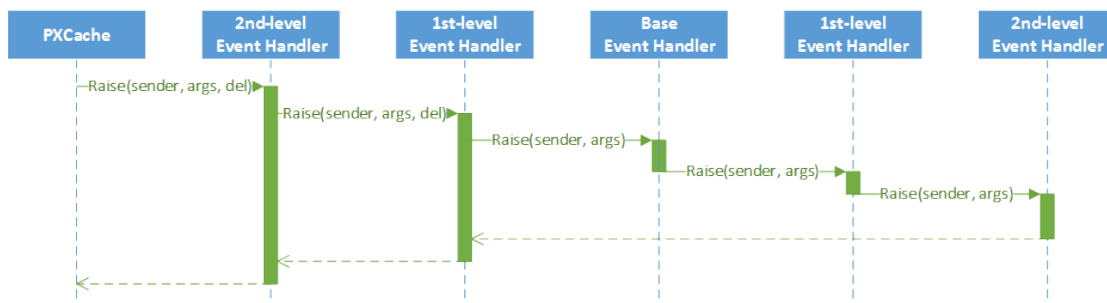


Figure: The order of the `RowUpdated` event handler execution

The following figure illustrates the order in which the `FieldVerifying` events are invoked.

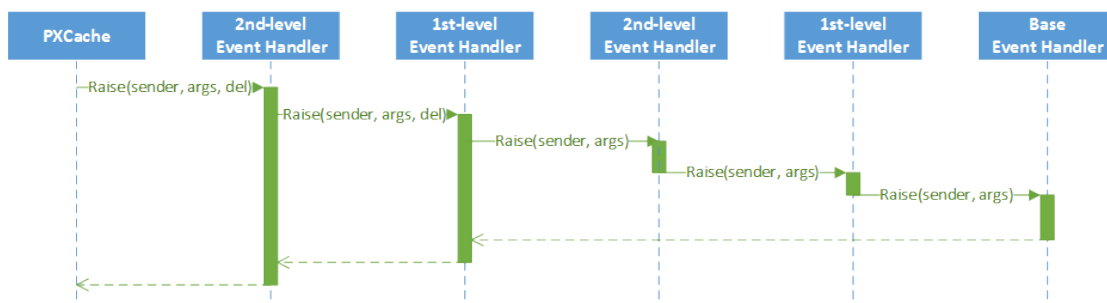


Figure: The order of the `FieldVerifying` event handler invocation

Customization of a Data View

The platform provides a way to alter or extend the data views defined in a BLC.



: A *data view* is a `PXSelect` BQL expression declared in a BLC for accessing and manipulating data. A data view may contain a delegate, which is an optional graph method that executes when the data view is requested. Every data view is represented by the `PXView` object and placed in the `Views` collection of the

appropriate BLC. To construct an instance of the `PXView` class, you use a `PXSelect` BQL expression and an optional delegate from the highest-level extension discovered.

Suppose that you have declared the `Objects` data view within the base BLC, as shown below.

```
public class BaseBLC : PXGraph<BaseBLC, DAC>
{
    public PXSelect<DAC> Objects;
}
```

You can alter a data view within a BLC extension in the following ways:

- By altering the data view within a BLC extension. A data view that is redeclared within a BLC extension replaces the base data view in the `Views` collection of the BLC instance. Consider the following example of a first-level BLC extension.

```
public class BaseBLCExt : PXGraphExtension<BaseBLC>
{
    public PXSelectOrderBy<DAC,
        OrderBy<Asc<DAC.field>>> Objects;
}
```

The `Views` collection of a BLC instance contains the `PXView` object, which uses the `Objects` data view declared within the first-level extension, instead of the data view declared within the base BLC.



: A data view redeclared within a BLC extension completely replaces the base data view within the `Views` collection of a BLC instance, including all attributes attached to the data view declared within the base BLC. You can either attach the same set of attributes to the data view or completely redeclare the attributes.

- By declaring or altering the data view delegate in a BLC extension. The new delegate is attached to the corresponding data view. Consider the following example of a second-level BLC extension.

```
public class BaseBLCExtOnExt : PXGraphExtension<BaseBLCExt, BaseBLC>
{
    protected IEnumerable objects()
    {
        return PXSelect<DAC>.Select(Base);
    }
}
```

The `Views` collection of a BLC instance contains the `PXView` object, which uses the `Objects` data view declared within the first-level extension, with the `objects()` delegate declared within the second-level extension (see the screenshot below).

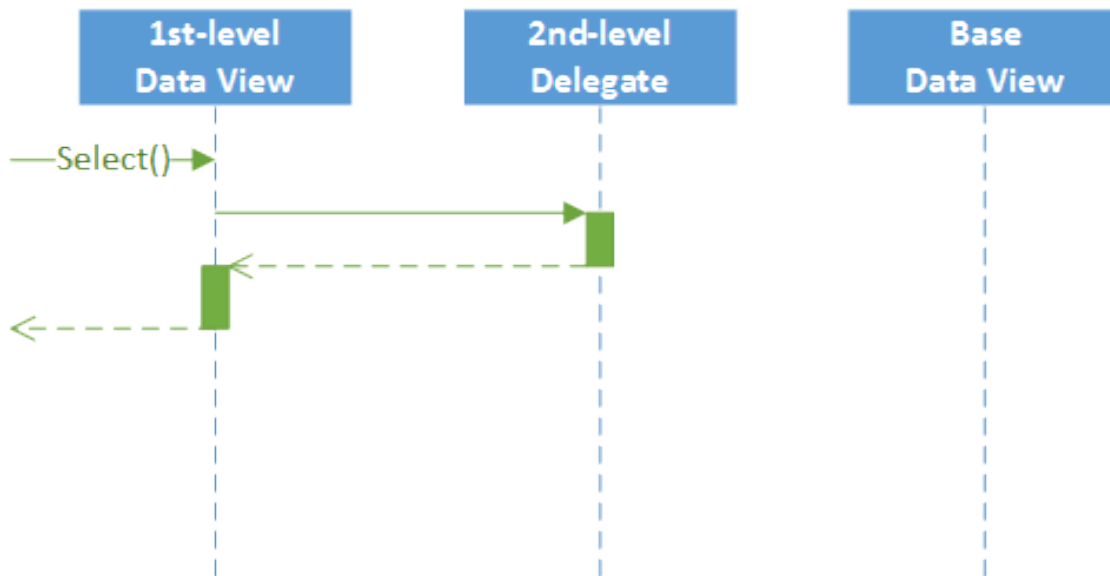


Figure: The interaction among the levels of a BLC extension

To query a data view declared within the base BLC or a lower-level extension from the data view delegate, you should redeclare the data view within a BLC extension. You do not need to redeclare a data member when it is not meant to be used from the data view delegate. Consider the following example of a second-level BLC extension.

```

public class BaseBLCExtOnExt : PXGraphExtension<BaseBLCExt, BaseBLC>
{
    protected IEnumerable objects()
    {
        return Base.Objects.Select();
    }
}
  
```

The new delegate queries the data view declared within the base BLC. Having redeclared the data view within the first-level extension, you prevent the data view execution from an infinite loop (see the following screenshot).

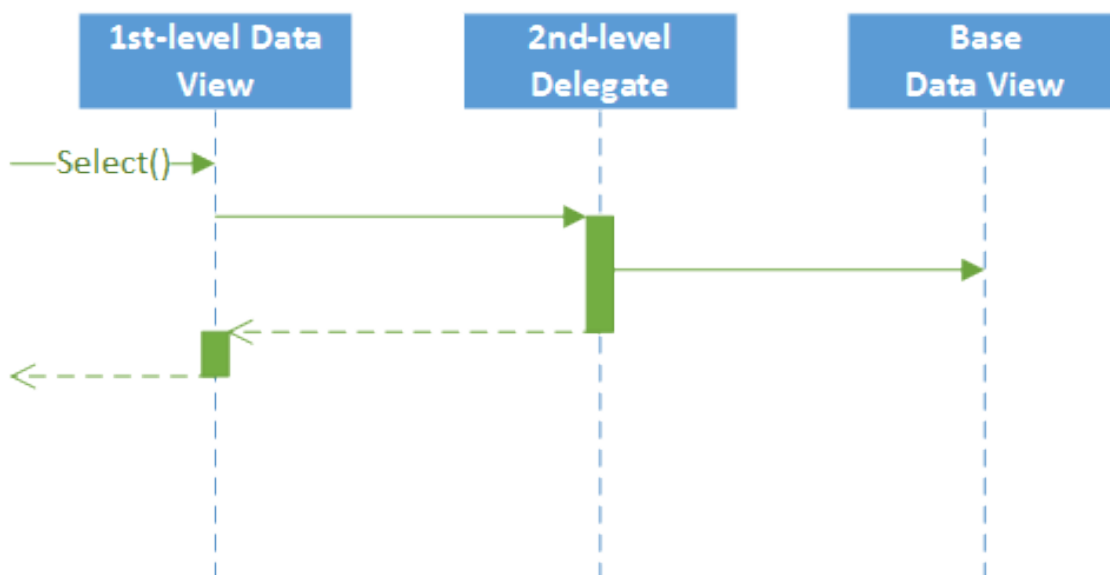


Figure: The view of the base BLC from the second-level delegate



: If a data view declared within the base BLC contains a delegate, this delegate also gets invoked when the data view is queried from the new delegate (see the following figure).

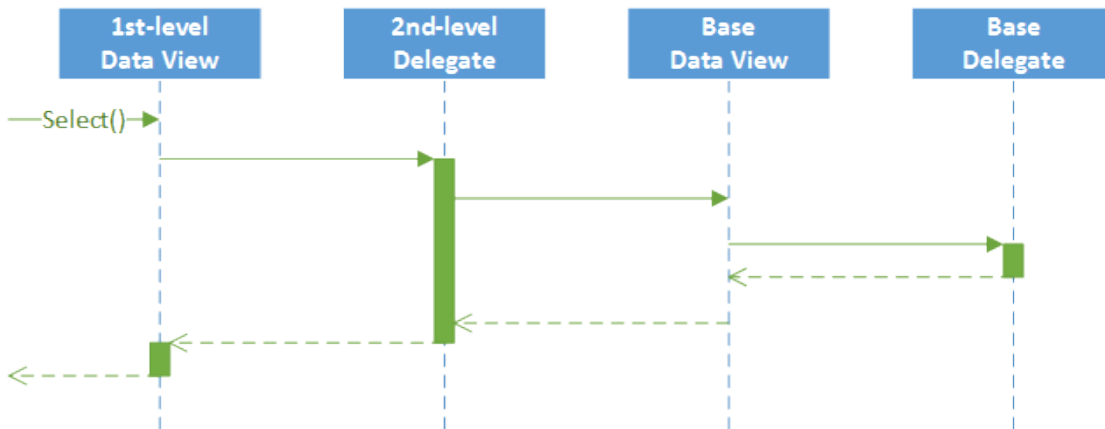


Figure: The view with the delegate of the base BLC from the second-level delegate

Customization of an Action

The platform provides a way to alter or extend an action defined in a business logic controller (BLC, also referred as *graph*).



: An *action* is a BLC member of the `PXAction` type. An action always has the delegate defined. Every action is represented by the `PXAction` object and placed in the `Actions` collection of the appropriate BLC. To construct an instance of the `PXAction` class, you use a BLC member of the `PXAction` type and a delegate from the highest-level extension discovered.

Suppose that you have declared the `Objects` data view and the `ValidateObjects` action within the base BLC, as shown below.

```

public class BaseBLC : PXGraph<BaseBLC, DAC>
{
    public PXSelect<DAC> Objects;

    public PXAction<DAC> ValidateObjects;
    [PXButton]
    [PXUIField(DisplayName = "Validate Objects")]
    protected virtual void validateObjects()
    {
    }
}
  
```

You can alter actions within BLC extensions in the following ways:

- By overriding action attributes within a BLC extension.

To override action attributes in a BLC extension, you should declare both the BLC member of the `PXAction` type and the delegate. You should attach a new set of attributes to the action delegate. Also, you need to invoke the `Press` method on the base BLC action. Having redeclared the member of `PXAction<>`, you prevent the action delegate execution from an infinite loop. Consider the following example of a first-level BLC extension.

```

public class BaseBLCExt : PXGraphExtension<BaseBLC>
{
    public PXAction<DAC> ValidateObjects;
    [PXButton]
    [PXUIField(DisplayName = "Validate All Objects")]
    protected void validateObjects()
    {
        Base.ValidateObjects.Press();
    }
}
  
```

```

    }
}

```

The *Actions* collection of a BLC instance contains the `ValidateObjects` action, which consists of the `PXAction` type member and the delegate, both of which were declared within the first-level extension (see the following figure).



: When overriding an action delegate within a BLC extension, you completely redeclare the attributes attached to the action. Every action delegate must have `PXButtonAttribute` (or the derived attribute) and `PXUIFieldAttribute` attached.

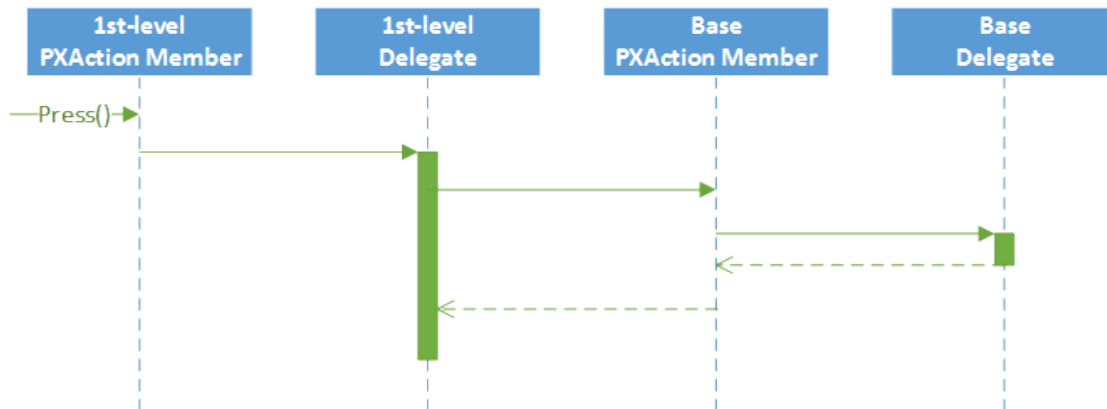


Figure: The interaction among the first-level and base actions

- By overriding the action delegate within the BLC extension.

The new delegate is used by the appropriate action. Consider the following example of a second-level BLC extension.

```

public class BaseBLCExtOnExt : PXGraphExtension<BaseBLCExt, BaseBLC>
{
    [PXButton]
    [PXUIField(DisplayName = "Validate Objects")]
    protected void validateObjects()
    {
    }
}

```

The *Actions* collection of a BLC instance contains the `ValidateObjects` action, which consists of the `PXAction<>` type member declared within the first-level extension and the delegate declared within the second-level extension. To use an action declared within the base BLC or the lower-level extension from the action delegate, you should redeclare the action within a BLC extension. You do not need to redeclare an action when it is not meant to be used from the action delegate.

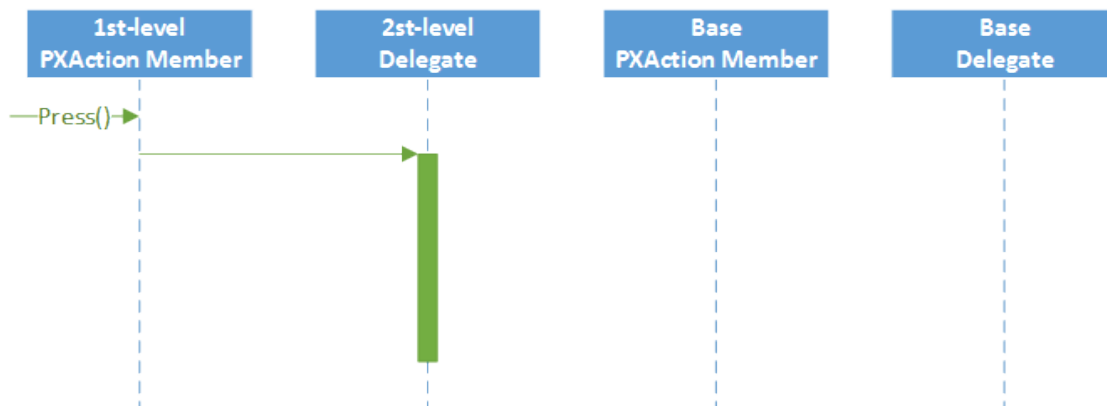


Figure: The case when you don't need to redeclare an action



: To modify the same member of the base BLC or any BLC extension, you should use extensions of higher levels.

Override of a Method

The platform provides a way to override a method of a BLC.

Suppose that you need to override the `Persist` method of the `BaseBLC` class, which is defined in the code below.

```
public class BaseBLC : PXGraph<BaseBLC>
{
    public virtual void Persist()
    {
        ...
    }
}
```

You can override the method within a BLC extension as follows.

```
public class BaseBLC_Extension : PXGraphExtension<BaseBLC>
{
    public delegate void PersistDelegate();
    [PXOverride]
    public void Persist(PersistDelegate baseMethod)
    {
        ...
        baseMethod();
        ...
    }
}
```

The graph extension should include the declaration of the base method delegate, the `PXOverride` attribute, and the overridden method, which invokes the base method delegate, as the code above shows.

Because the `PXOverride` attribute on the declaration of the method is included in the graph extension, it forces the system to override the original method in the graph instance. Otherwise, the overridden method will never be invoked.

Run-Time Compilation

If you have created an extension for a business logic controller or a data access class in a customization project, during the project publication, the Acumatica Customization Platform stores the code of the class extension in a `.cs` file within the `App_RuntimeCode` folder of the website.

At run time, the platform compiles the code in a separate library that is dynamically merged with the original classes. Run-time compilation is used by default for the code of the *DAC* and *Code* project items. (See [Types of Items in a Customization Project](#) for details.)

Publication of a customization updates the files in the `App_RuntimeCode` folder of the website. Unlike when the `Bin` folder is updated, update of files in the `App_RuntimeCode` folder doesn't cause the application domain to restart. (See [Performing the Publication Process](#) for details.)

Extension Library

An extension library is a Microsoft Visual Studio project that contains customization code and can be individually developed and tested. An extension library `.dll` file must be located in the `Bin` folder of the website. At run time during the website initialization, all the `.dll` files of the folder are loaded into the server memory for use by the Acumatica ERP application. Therefore, all the code extensions included in a library are accessible from the application.

During the first initialization of a base class, the Acumatica Customization Platform automatically discovers an extension for the class in the memory and applies the customization by replacing the

base class with the merged result of the base class and the extension discovered (see the following diagram).

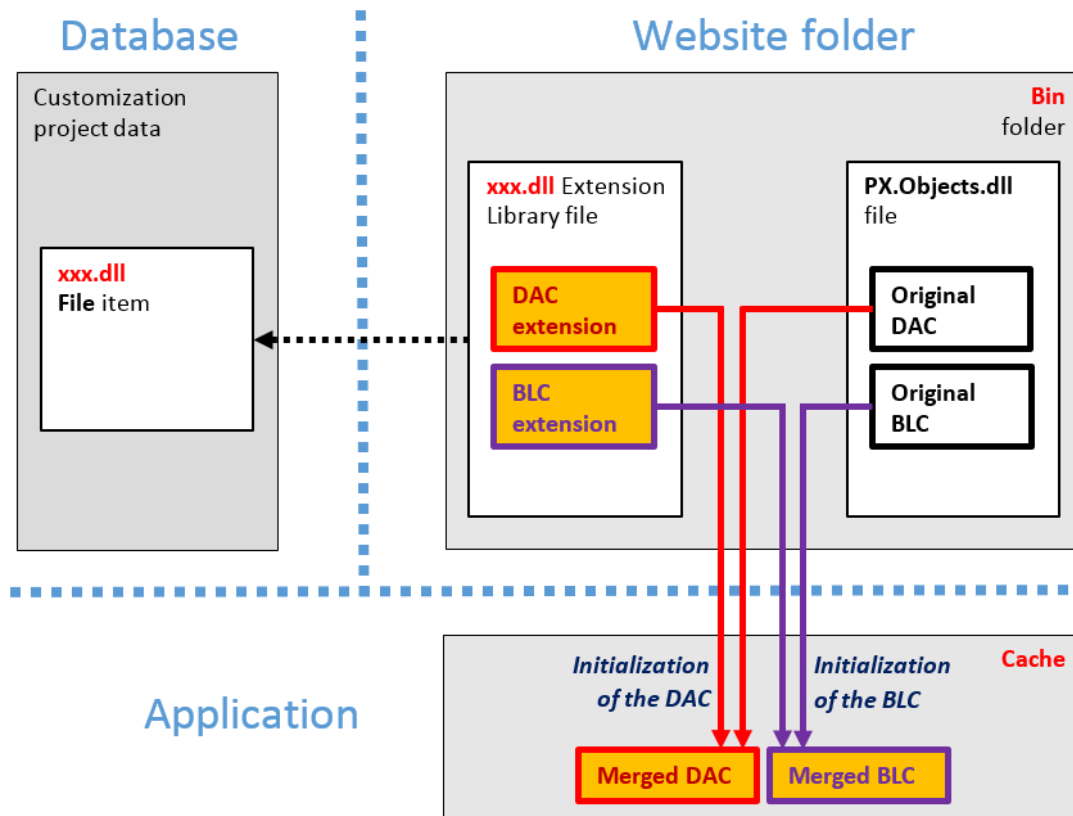


Figure: Actual approach to the use of an extension library for a customization

If you need to deploy the customization code of an extension library to another system, you have to add the library to a customization project as a *File* item to include it in a customization package.

The use of extension libraries that are precompiled provides a measure of protection for your source code and intellectual property.

Extension Library (DLL) Versus Code in a Customization Project

While developing source code for a customization task, you can either keep the code in the customization project as *DAC* and *Code* items or move the code to an extension library and include the library in the project as a *File* item.

DAC and *Code* items are saved in the database and, if the customization project is published, in the files located in the `App_RuntimeCode` folder of the website. Acumatica Customization Platform compiles all the code located in this folder at run time.

We recommend that you keep the source code in the customization project only if the customization is elementary, such as if the code contains an event handler for a control on a form. When you work with a code item, it is easy to view the code of the applied customization directly in the [Code Editor](#) without Microsoft Visual Studio. However if you are solving a complex customization task, it is generally better to develop the solution code in an extension library using Visual Studio.

You can use Visual Studio to develop a *Code* item as well as code in an extension library. But you have no *IntelliSense* feature in Visual Studio for the source code of a *Code* item if the code either uses custom fields or invokes members of class extensions defined in other *Code* and *DAC* items.

You can later customize deployed extensions for the end user by means of code extensions. Therefore, if the customization is compiled to an extension library, more than one customization can be applied to a single Acumatica ERP form.

The following table shows the differences in the use of the code in *DAC* and *Code* items and in an extension library.

	Code in DAC and Code items	Code in an Extension Library
Best for:	Quick start of a customization	Development of a customization project when more than one developer is involved
Primary storage:	Database	File system
Location within the website folder:	App_RuntimeCode	Bin
Intellectual property protection:	No—the source code is open in the deployment package	Yes—the source code is not provided in the deployment package
Run-time compilation without the application domain being restarted:	Yes	No
Editor:	Code Editor , Visual Studio	Visual Studio
<i>IntelliSense</i> feature in Visual Studio:	No	Yes
Debugging:	Yes	Yes
Integration with a version control system:	Yes	Yes
Additional:	Can be moved to an extension library when needed	

To make a decision about how to work with the code, you should answer the following questions:

- What amount of the code will be in the customization project?
- Is there a need for replicability of the customization?
- How many developers will take part in coding?
- Do you need to open the source code on the production environment?

We recommend that you use an extension library if any of the following conditions is met:

- You intend to have more than five class extensions for business logic controllers.
- The customization will be deployed on more than one system.
- The customization code will be developed by a team that needs to use a version control system.
- You have a reason to protect the intellectual property of the source code of the solution.

Changes in the Database Schema

The Acumatica Customization Platform permits the following changes to the database in the scope of a customization project:

- [Creation of custom tables](#)
- [Creation of custom columns in existing tables](#)
- [Creation of views, indexes, and other database objects](#)

Creation of Custom Tables

We recommend that you create a custom table in the database of your development environment by using a database administration tool, such as SQL Server Management Studio, and then import the table schema from the database to a customization project by using the [Customization Project Editor](#). The project keeps the schema in XML format. While publishing the customization project, the platform

executes a special procedure to create the table by the schema, while meeting all the requirements of Acumatica ERP.

Creation of Custom Columns in Existing Tables

To create a custom database-bound field, you add a column to the database table and declare the field in the extension of the base (original) DAC.

The new column is appended to the original table by altering the table schema. When you create the database-bound field by using the [Data Class Editor](#), the platform generates the DAC extension code for the new field and adds to the customization project the XML definition of the new column to be created in the database. To be able to create a UI control for the new field to display on a form, you have to publish the project to make the system create the column in the database table and compile the customization code. After the publication, you can add the control for the new field to the form by using the [Layout Editor](#).

Creation of Views, Indexes, and Other Database Objects

The platform permits you to add to a customization project an SQL script, written in Transact-SQL (MS SQL) dialect, to be executed during the publication of the project. However we recommend that you avoid doing this for the following reasons:

- Because Acumatica ERP supports multi-tenancy, it is difficult to create an SQL script that correctly creates a database object.
- It is difficult to properly specify and use the company mask in custom database objects.
- If you include in a customization project an SQL script written for MS SQL, avoid applying the customization to a website on MySQL Server, because an SQL script created for MS SQL Server might not work properly on MySQL Server.



Warning: A possible result of a custom SQL script is the loss of the integrity and the consistency of data.

Custom Processes During Publication of a Customization

The platform provides you the ability to execute custom processes within the instance of Acumatica ERP at the following times during the publication of a customization project:

- After website files are updated but before the website is restarted
- After the customization has been published and the website is restarted

To implement such processes, you can create a class derived from the `CustomizationPlugin` class, further referred to as the *customization plug-in*, and override one or both of the following methods, which can be invoked at the end of the publication process:

- The `OnPublished()` method is executed right after website files are updated but before the website are restarted. This method is invoked only if run-time compilation is enabled. In a cluster environment, the method is invoked on each cluster node. By using this method, you can update any files within the website except the customization code files located in the `/Bin` folder. For example, you can set up your own UI styles and skins or log-in images.
- The `UpdateDatabase()` method is executed after the customization is published and the website is restarted. In this method, for example, you can already manipulate data in the database by using the business logic implemented in the customization. For example, you can develop source code to import data by using a scenario included in the customization.

In a customization project, you can include multiple classes derived from the `CustomizationPlugin` class.



: The platform does not provide you the ability to set the order of execution of customization plug-ins in a published customization.

For details about how to create a class derived from the `CustomizationPlugin` class, see [To Add a Customization Plug-In to a Project](#).

Performing Customization

As a rule, the process of resolving a customization task can be divided into four stages:

- *Exploration stage*: During this stage, you explore the original code of your Acumatica ERP application instance. This stage is executed in the development environment, which must be the same as the production environment.
- *Development stage*: This stage includes the steps to develop a customization project; the stage is also executed locally in the development environment, which must include all customizations that are included in the production environment.
- *Final testing stage*: In this stage, you test the newly developed customization project in the staging environment, which can be a local copy of the customized production environment. The staging environment must include a copy of the production database.
- *Deployment stage*: This stage must be executed in the production environment.



:

The version number of Acumatica ERP used in the development, staging, and production environments should be the same. If they are not, before deploying and publishing a project on the production instance, you should update the application instances to the same version. If the production application instance has been updated (or will be updated soon) to a particular version, you should update the development and staging instances to the target version and repeat all testing steps. Then, if the tests are successful, you should again create the deployment package to upload it to the production environment.

Also, keep in mind that to carry out the final testing (with the production database) and deployment stages, you must be registered as an internal user with the *Customizer* role assigned.

You develop and maintain customization projects by using the *Customization Tools* of Acumatica Customization Platform. The platform provides the mechanisms to develop and publish customization projects. After the customization project is ready, you prepare the deployment package to distribute the customization project to the staging or production environment (see *Deployment of Customization* for details).

In This Part

- [To Assign the Customizer Role to a User Account](#)
- [To Detect Whether a Customization Project Is Applied to the Application](#)
- [Exploring the Source Code](#)
- [To Develop a Customization Project](#)
- [To Perform Final Testing of a Customization](#)
- [To Deploy a Customization Project](#)

To Assign the Customizer Role to a User Account

A user with the *Administrator* role has full access rights in the system and can assign the *Customizer* role to the needed users by using the *Users* (SM.20.10.10) form.



: The default *admin* user has the *Administrator* role.

If you have the *Administrator* role, do the following to assign the *Customizer* role to a user account:

1. Navigate to **Configuration > User Security > Manage > Users**.
2. In the **Login** box, select the user account to which the *Customizer* role should be assigned.

3. Make sure the user account is not a guest account (in the Summary area) and is not assigned to a guest role (on the **Roles** tab).
4. On the **Roles** tab, select the check box for the *Customizer* role, and click **Save** on the form toolbar.

The screenshot below shows the *Johnson* user account assigned to the *Customizer* role. The account is not a guest account and is not assigned to any guest roles.

The screenshot shows the Acumatica User Security interface. The top navigation bar includes 'ORGANIZATION', 'FINANCE', 'DISTRIBUTION', and 'CONFIGURATION'. The 'User Security' section is active, showing 'New York - Users'. The 'Roles' tab is selected for the user 'Johnson'. The 'Guest Account' checkbox is unchecked. A table lists various roles, with the 'Customizer' role highlighted and its checkbox checked.

Selected	Role Name	Role Description
<input type="checkbox"/>	Administrator	System Administrator
<input type="checkbox"/>	Anonymous	Anonymous
<input type="checkbox"/>	Consultant	Consultant
<input checked="" type="checkbox"/>	Customizer	System Customizer
<input checked="" type="checkbox"/>	Employee	Employee
<input type="checkbox"/>	Entry	Entry Clerk
<input type="checkbox"/>	Field-Level Audit	Role that can access Field-Level Audit
<input type="checkbox"/>	Financial	Financial Controller
<input type="checkbox"/>	Guest	External Guest Role
<input checked="" type="checkbox"/>	Internal User	Allows the user to change personal settings, a...
<input checked="" type="checkbox"/>	MAIN Users	MAIN Users
<input type="checkbox"/>	Management	General Management
<input checked="" type="checkbox"/>	OfficeAdministrator	Office Administrator
<input type="checkbox"/>	Portal Admin	Access to portal configuration
<input type="checkbox"/>	Portal User	Portal user
<input type="checkbox"/>	ReportDesigner	Report Designer
<input type="checkbox"/>	Sales	Sales Manager
<input type="checkbox"/>	Wiki Admin	Wiki Administrator to set other users access ri...
<input type="checkbox"/>	Wiki Author	Role provides access rights for creating Wiki a...

Figure: Assigning the Customizer role to a user account

After you save your changes, the user gets full access for customization of the system as soon as the user refreshes a form in the web browser.

User Access Rights for Customization

A user account must be granted the *Customizer* role to have the appropriate access needed for customization of Acumatica ERP.

The default *admin* user account is granted this role. Therefore, if you are a developer who is going to work with a customization project, you can install an application instance of Acumatica ERP on the local computer and use the *admin* user account to start doing the customization. For testing and deployment of the customization, you should also assign the *Customizer* role to the appropriate user accounts on the test instance of the application and on the production instance. On the production instance, only the users who manage the deployment of customization packages should be granted the *Customizer* role.

The users who will be granted the *Customizer* role must be Acumatica ERP internal users. You cannot assign the *Customizer* role to a user if either of the following conditions is true (or if both are true):

- The user has a guest account—that is, the user account has the **Guest Account** check box selected on the [Users](#) (SM.20.10.10) form.
- The user is assigned to a guest role. A guest role has the **Guest Role** check box selected on the [User Roles](#) (SM.20.10.05) form. The default guest roles are *Anonymous*, *Guest*, and *Portal User*.

A user account that is assigned to the *Customizer* role has access to the following objects of the system:

- The [Customization Menu](#) and [Element Inspector](#), which give this user the ability to inspect the element properties on every form of Acumatica ERP and to add items to a customization project.
- The [Customization Projects](#) (SM.20.45.05) form, which is used to manage and publish multiple customization projects. (See also [Customization Projects Form](#) .)
- The [Customization Project Editor](#), which is used to manage and develop the content of a selected customization project.

To Detect Whether a Customization Project Is Applied to the Application

You can see whether a customization project is applied to the application on the Welcome screen, with more detailed information available on the [Customization Projects](#) (SM.20.45.05) form. To detect whether an Acumatica ERP instance is customized, perform the following actions:

1. Launch the application in the browser.
2. In the bottom of the Welcome screen, check for the presence of the **Customized** string.

If the string exists, it is followed by the names of the customization projects that are currently published, as shown in the following screenshot.

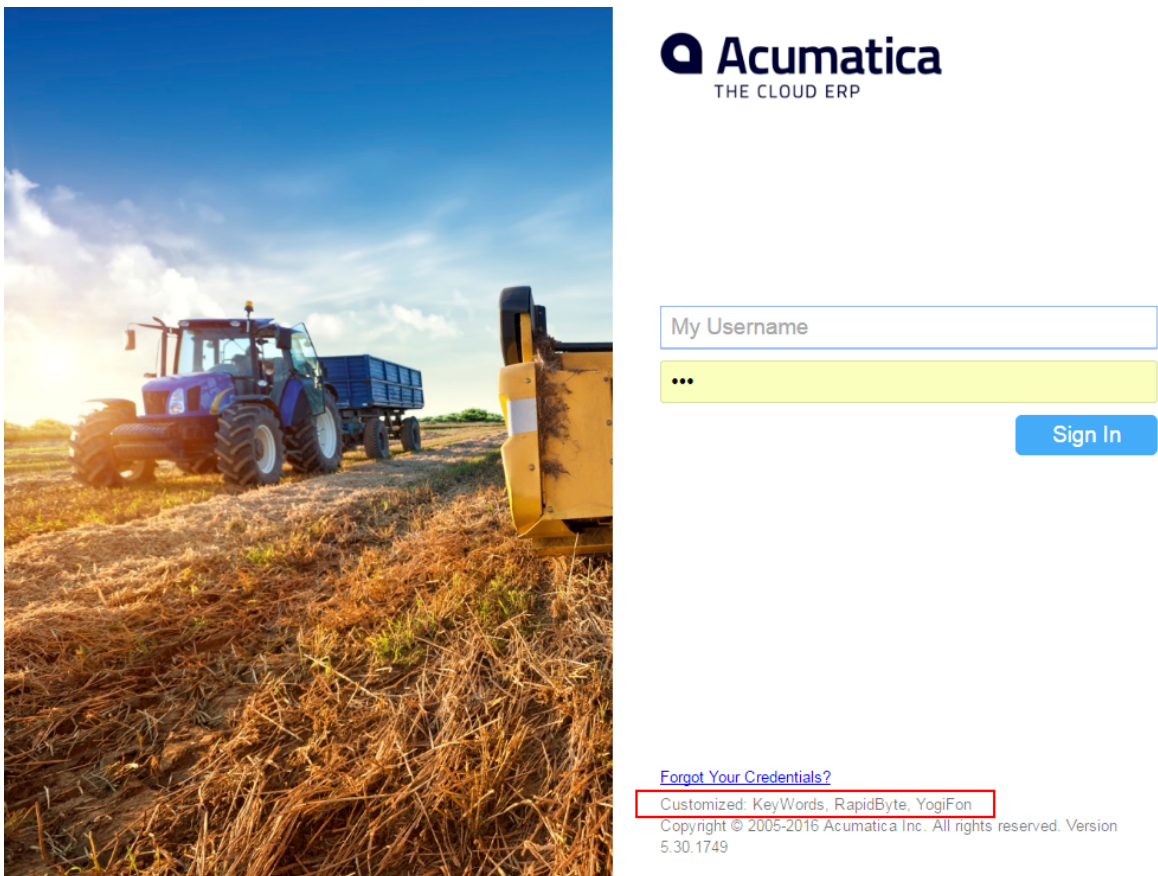


Figure: Viewing the customization projects that are currently published

3. Enter your credentials and click **Sign In**.
4. Navigate to **System > Customization > Manage > Customization Projects**.
5. On the form, view the list of the customization projects that are accessible in the application for your company (see the following screenshot).

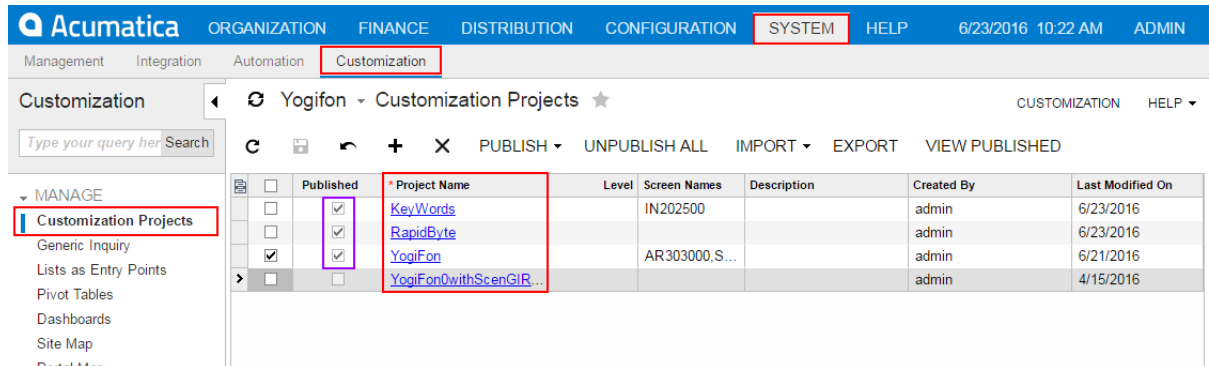


Figure: Viewing the list of customization projects



: If a customization project that you know to be published is absent in the list, the project may be published for another company. See [Customization of a Multi-Company Site](#) for details.

6. In the **Published** column of the list, notice which customization projects have the check box selected. These are the projects that are currently published.
7. Use the [Customization Project Editor](#) to explore the content of each published customization project. To open a project in the editor, click the name of the project.

You can also view the names of the customization projects that are currently published in the **About Acumatica** dialog box, as the following screenshot shows.

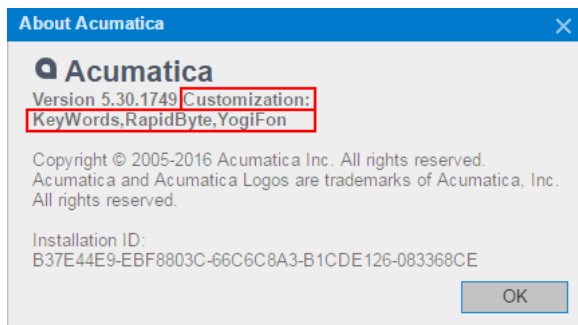


Figure: Viewing the list of the customization projects that are currently published

To open the dialog box, click **Help > About** on a form of Acumatica ERP.

Exploring the Source Code

Before you start to customize an Acumatica ERP application instance, you should analyze your business requirements to identify the changes in business processes that must be done and to identify the application objects for the customization. We recommend that you learn about the structure of the involved webpages by exploring the `.aspx` pages, the code of the business logic controllers (BLCs), and the data access classes (DACs) that are used within the BLCs.

You can use the [Source Code](#) (SM.20.45.70) form (see also [Source Code Browser](#)) to explore the original source code of the application for the following purposes:

- To learn about the structure of the webpages involved in the customization
- To understand which data views are bound with the customized form areas

- To analyze the code that provides the business logic for a form to be customized
- To find and analyze the data views used on a customized webpage
- To understand which data access classes the form being customized is based on
- To learn the attributes of the DAC fields and the relations between DACs

This section contains instructions on how to do the following:

- [To Explore the C# Code of a BLC](#)
- [To Explore the C# Code of a DAC](#)
- [To Explore the ASPX Code of a Page](#)
- [To Find a Customization of the ASPX Code](#)
- [To Find Source Code by a Fragment](#)

To Explore the C# Code of a BLC

If you need to customize the business logic for a form of Acumatica ERP, you have to explore the original source code of the business logic controller (BLC) that provides the business logic for the form. The goal of exploring the code is to discover the data views, methods, and event handlers of the BLC.

To do this, perform the following actions:

1. On a selected form, click **Customization > Inspect Element**, as item a of the screenshot below shows, to activate the *Element Inspector*.
2. Click a control or area of the form to open the *Element Properties Dialog Box* for the control or area. See item b on the screenshot.
3. In the dialog box, click **Actions > View Business Logic Source** (item c).

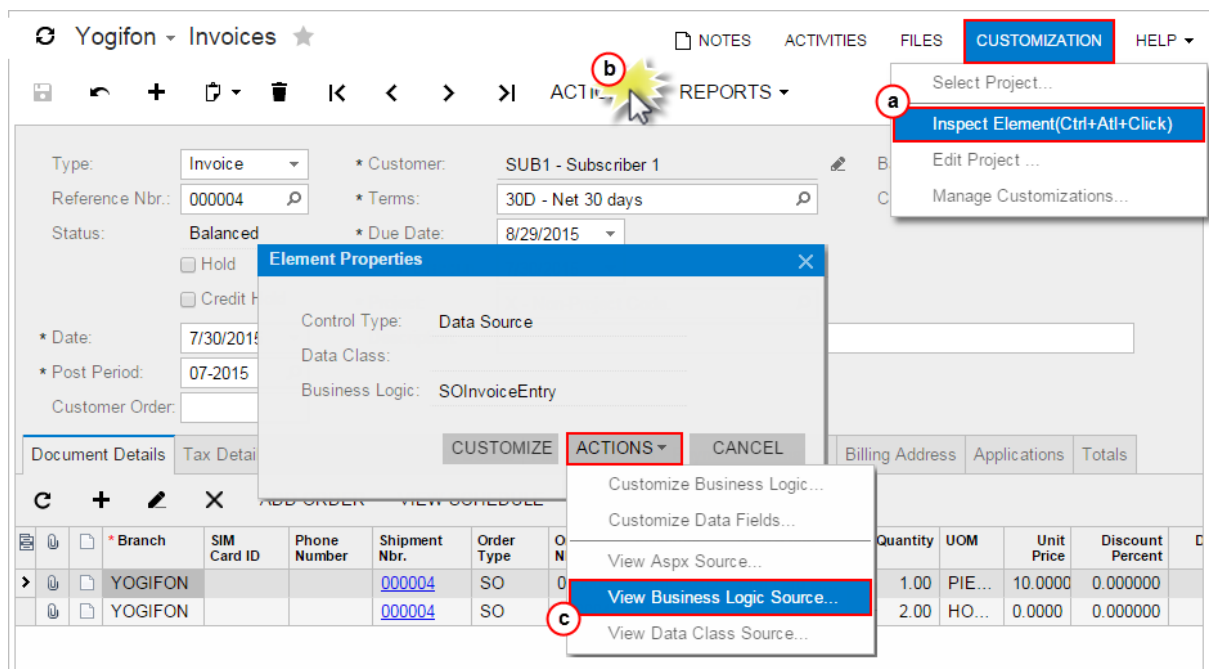


Figure: Selecting the grid container for the customization

4. In the *Source Code* (SM.20.45.70) form (see also *Source Code Browser*), which opens for the BLC, view the source code in the work area, and use the navigation pane to find a method or event handler by its name and open it.

Also, you can open the original BLC code in the Source Code browser in the following ways:

- From the *Code Editor*, by clicking **View Source** on the page toolbar

- From the *Layout Editor*, by clicking **View Source** on the toolbar of the **Events** tab

To Explore the C# Code of a DAC

If you need to customize the attributes of a data field for an existing control or create a new field for a custom control on a form, you may need to explore the original source code of the appropriate DAC.

To do this, perform the following actions:

1. On the selected form, click **Customization > Inspect Element**, as item a of the screenshot below shows, to activate the *Element Inspector*.
2. Click a control or area of the form to open the *Element Properties Dialog Box* for the control or area. See item b on the screenshot.
3. In the dialog box, click **Actions > View Data Class Source** (item c).

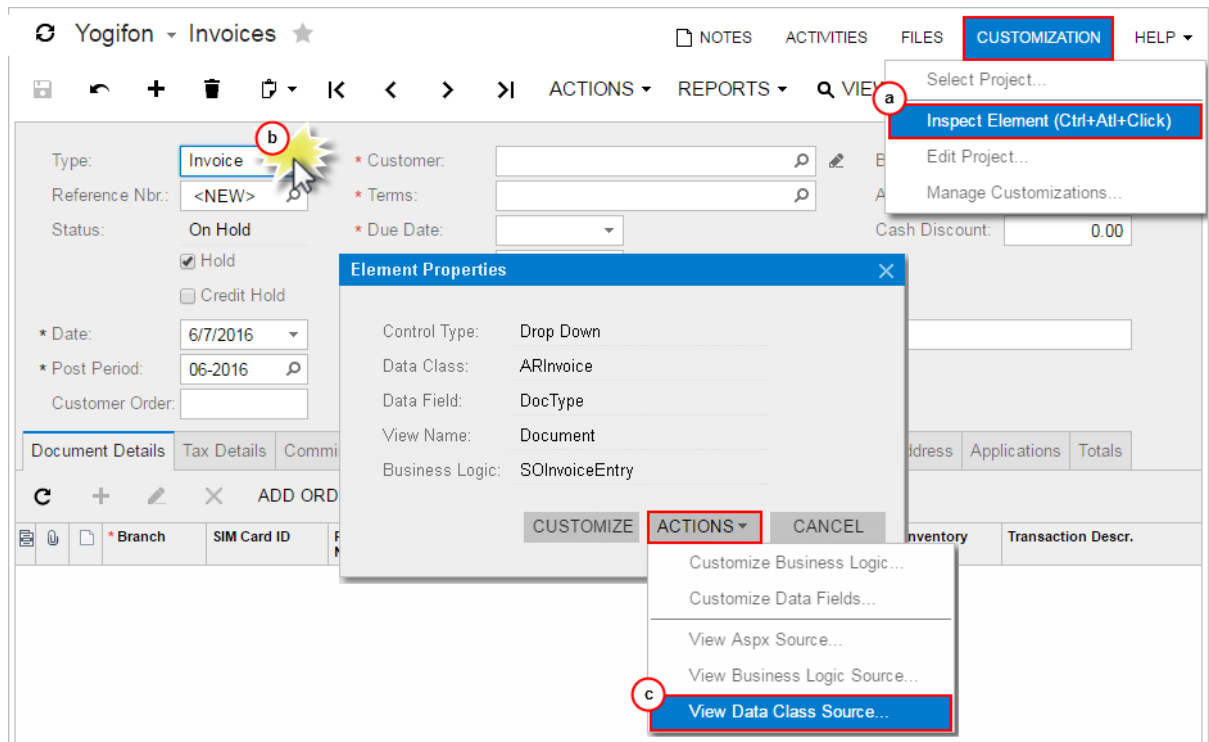


Figure: Selecting the grid container for the customization

4. On the *Source Code* (SM.20.45.05) form (see also *Source Code Browser*), which opens for the DAC, view the data field declarations in the work area.

Also you can open the original DAC code in the Source Code browser in the following ways:

- From the *Data Class Editor*, by clicking **View Source** on the page toolbar
- From the *Layout Editor*, by clicking **View Source** on the toolbar of the **Attributes** tab

To Explore the ASPX Code of a Page

If you need to look closely at the data views that provide data for control containers on a form or to see the corresponding webpage structure—that is, the layout of the containers and the types and properties of the controls—you may need to explore the original ASPX code of the webpage.

To explore this code, perform the following actions:

1. On the selected form, click **Customization > Inspect Element**, as item a in the screenshot below shows, to activate the *Element Inspector*.
2. Click a control or area of the form to open the *Element Properties Dialog Box* for the control or area. See item b on the screenshot.

3. In the dialog box, click **Actions > View ASPX Source** (item c).

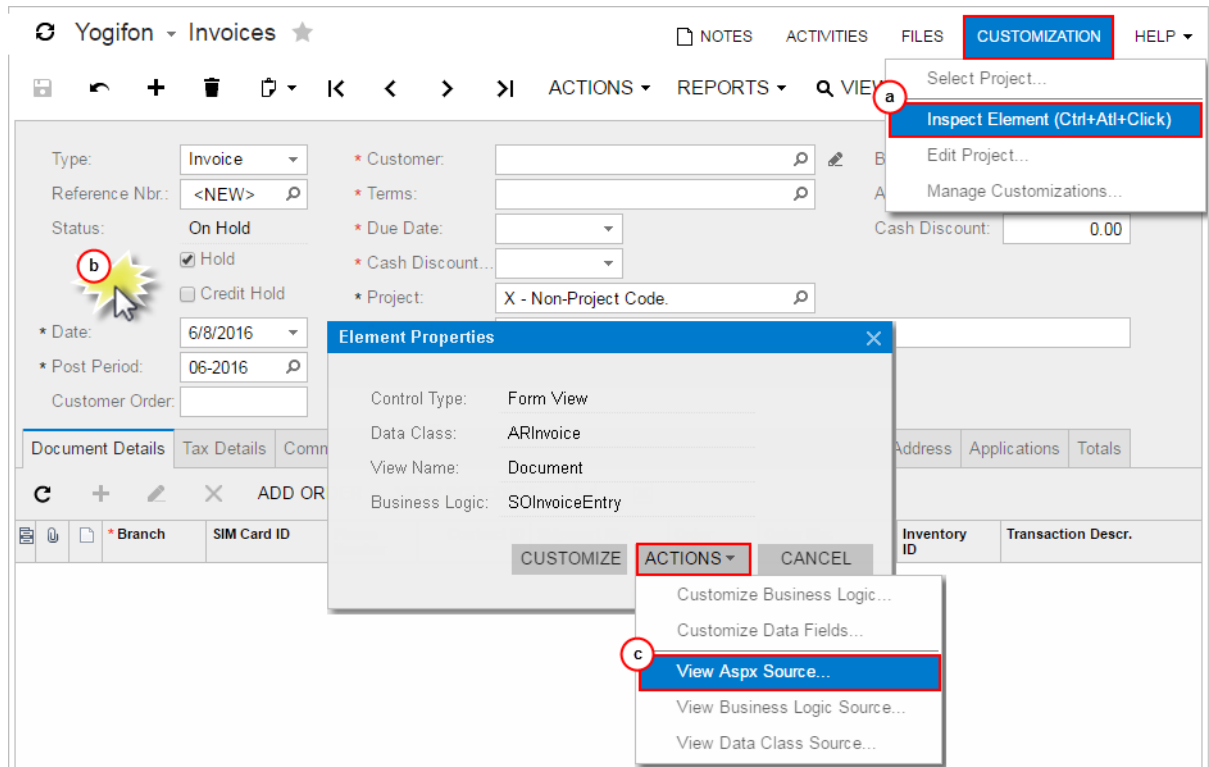


Figure: Selecting the grid container for the customization

4. On the [Source Code](#) (SM.20.45.70) form (see also [Source Code Browser](#)), which opens for the form, view the original ASPX code of the page in the work area.
5. If you need to look closely at the data views that provide data for control containers on the current form, perform a browser search to find the `DataMember` string. The `DataMember` property is used to bind a control container of a form to a data view defined in the business logic controller (BLC) of the form. The property value is the name of the data view.



: Each `DataMember` property value can correspond to any data view name of the BLC. Any container (for example, `PXTab`, `PXGridLevel`, or `PXFormView`) must be bound to a data view declared within a BLC. Any data view except for the main data view can be used by an unlimited number of containers. The main data view must be bound to a single container.

To Find a Customization of the ASPX Code

If the ASPX code for a form is customized, to explore changes in the code, you use the [Layout Editor](#) of the [Customization Project Editor](#), which you access by using the [Customization Projects](#) (SM.20.45.05) form.



: The Source Code browser can display only the original ASPX code of a webpage.

To detect whether a form is currently customized, do the following:

1. Open the form in the browser.
2. Click **Help** on the title bar of the form, as the following screenshot shows.

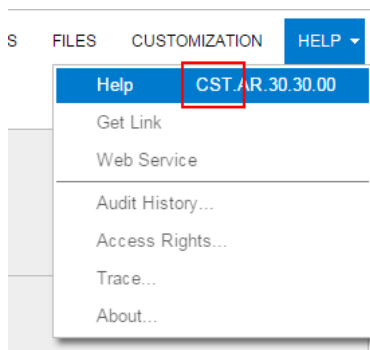


Figure: Checking whether a form has been customized

If the form has been customized, the form identifier has the *CST.* prefix.

Once you know the form has been customized, to find the customization of the ASPX code of the form, perform the following actions:

1. Determine the published customization projects that contain changes for the form as follows:
 - a. Navigate to **System > Customization > Manage > Customization Projects**
 - b. On the form, view the list of the customization projects (see the following screenshot)

	Publish	Project Name	Level	Screen Names	Description	Created By	Last Modified On
<input type="checkbox"/>	<input type="checkbox"/>	EcommerceSe...		IN202500		admin	10/27/2016
<input type="checkbox"/>	<input type="checkbox"/>	HideSSN		CR303010		admin	7/18/2014
<input type="checkbox"/>	<input type="checkbox"/>	POstatus		PO301000		admin	7/18/2014
<input type="checkbox"/>	<input type="checkbox"/>	StockImage				admin	8/13/2014
<input type="checkbox"/>	<input type="checkbox"/>	StockItemImages		AR303000,SO301000,SO303000		admin	10/27/2016
<input type="checkbox"/>	<input checked="" type="checkbox"/>	YogiFon		AR303000,AR409001,SO301000,SO303000		admin	11/3/2016

Figure: Viewing the list of customization projects

- c. In the **Screen Names** column, for the published customization projects (those for which the **Published** check box is selected), scan the form IDs to identify the projects that contains changes for this form.
2. To explore a published project that contains changes for this form, perform the following actions:
 - a. Click the name of the project to open it in the *Customization Project Editor*.
 - b. In the navigation pane of the editor, click **Screens** to open the Customized Screens page.
 - c. On the page (see the screenshot below), click the form ID in the **Screen ID** column to open the *Layout Editor* for the form.

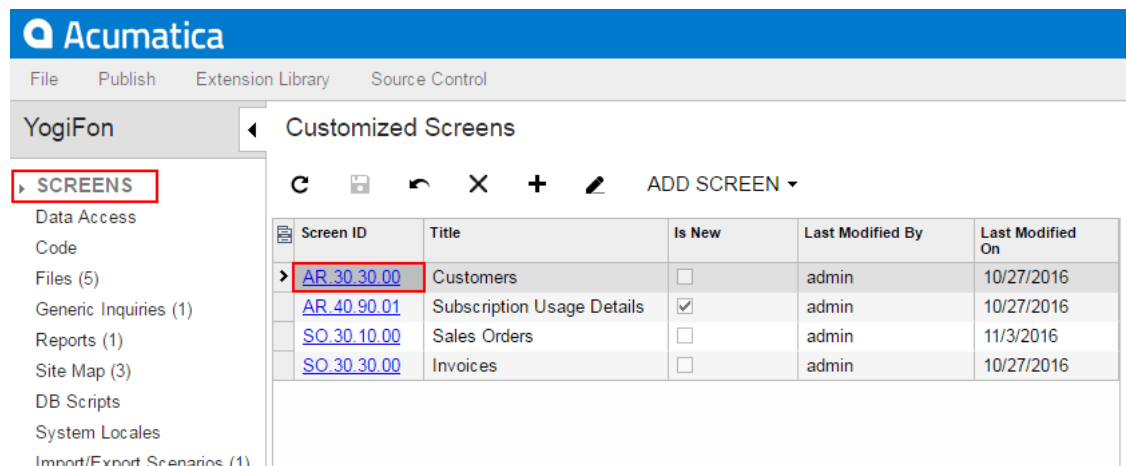


Figure: Viewing the Customized Screens page of the Customization Project Editor

- d. In the Layout Editor, select the **View ASPX** tab item to view the customized ASPX code of the node that is currently selected in the Control Tree
- e. Explore each node of the tree to find the changes, which are highlighted in yellow, as the following screenshot shows.

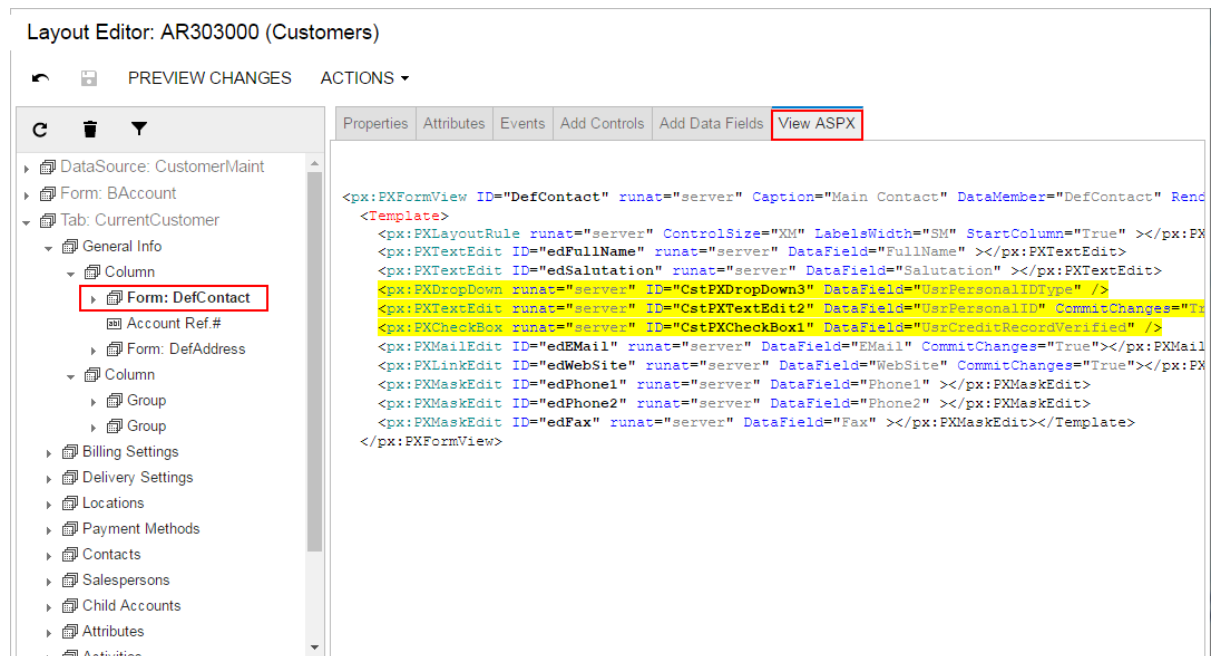


Figure: Viewing the customization of the ASPX code

To Find Source Code by a Fragment

In an instance of Acumatica ERP, a repository with the original C# source code of the application is kept in the \App_Data\CodeRepository folder of the website. You can use the [Source Code](#) (SM.20.45.70) form to find the source code within the repository by a code fragment. (For more information, see also [Source Code Browser](#).)

To do this, perform the following actions:

1. Navigate to **System > Customization > Explore > Source Code**.
2. Click the **File in Files** tab item.
3. In the **Find Text** box, enter (by typing or by copying and pasting) a code fragment.

4. Click **Find** to start the procedure.

The results of the code search are displayed on the form, as the following screenshot shows.

The screenshot shows the Acumatica Customization interface. The 'Source Code' browser is active, displaying search results for the text 'PXGenericInqGrph'. The search results table lists several files, with the first result selected:

Name	Line	Content
PX.Data.Cache.Model.cs	3120	_Graph.GetType() == typeof(PXGraph) _Graph.GetType() == typeof(PXGenericInqGrph) _Graph...
PX.Data.Descriptor.Attributes.SelectorA...	1578	if (sender.Graph.GetType() == typeof(PXGenericInqGrph))
PX.Objects.CM.Descriptor.Attribute.cs	777	if (sender.Graph.GetType() == typeof(PXGenericInqGrph) sender.Graph.GetType() == typeof(PXGrap...
PX.Objects.CM.Descriptor.Attribute.cs	993	if (sender.Graph.GetType() == typeof(PXGenericInqGrph) sender.Graph.GetType() == typeof(PXGrap...
PX.Objects.CM.Descriptor.Attribute.cs	2137	if (sender.Graph.GetType() == typeof(PXGenericInqGrph) sender.Graph.GetType() == typeof(PXGrap...
PX.Objects.CR.Descriptor.Indexer.cs	1295	if (sender.Graph.GetType() != typeof(PXGraph) && sender.Graph.GetType() != typeof(PXGenericInqGr...
PX.Objects.CS.Descriptor.Attribute.cs	2468	if (sender.Locate(e.Row) == null && !(sender.Graph is PXGenericInqGrph))
PX.Objects.CS.Descriptor.Attribute.cs	4724	sender.Graph.GetType() == typeof(PXGenericInqGrph) return;

The selected file's source code is displayed in a text area below the table. The code snippet shows a conditional check for the type of the sender's graph:

```

if (sender.Graph.GetType() == typeof(PXGenericInqGrph))
{
    var values = new Dictionary<string, object>();
    values.Add(_FieldName, sender.GetValue(e.Row, _FieldName));
    if (_DescriptionField != null)
        values.Add(_DescriptionField.Name, sender.GetValue(e.Row,
    row = values;
}
else
{
    row = e.Row;
}
dic[key] = new KeyValuePair<object, bool>(row, false);

```

Figure: Viewing the results of the code search in the Source Code browser

The form displays the results in the following elements:

- A list of the locations of the specified code fragment in the repository files
- A text area with the source code of the location selected in the list

5. Explore the use of the code fragment in the original source code by selecting a location in the list and scrolling the code in the text area.

To Develop a Customization Project

We recommend that you use a separate application instance for each developer working on a single project or group of projects. Multiple developers should not work simultaneously with the same project or projects. You should divide customization scope among the developers on a separate areas so each developer works with a separate application instance. The created customization projects can be sequentially imported into the customer's production environment and applied as if all the divided tasks had been resolved as a single common one.

During the development stage, you develop the customization, which involves implementing your planned changes by using the framework and tools provided by the Acumatica Customization Platform.

Follow this recommended workflow during the development process:

1. Prepare the development environment in accordance with [System Requirements for Acumatica ERP 6.1](#).
2. Install a version of Acumatica ERP that has the same number as the version used in the production environment. (See [Installing Acumatica ERP](#) for details.) All of the following actions must be performed in this environment.
3. Start the application instance.
4. Create a new customization project or select an existing one.

5. Open the project in the [Customization Project Editor](#).
6. Split the customization process into discrete steps, such as adding an UI element or extending business logic, and perform the needed changes step by step. (See [Customizing Elements of the User Interface](#), [Customizing Business Logic](#), and [Customizing the Database Schema](#) for details.)
7. After the completion of every step, validate the changes you have performed.
 Before the validation process begins, the platform automatically saves the content of the project (items and all the external files) to the database of the application instance that is currently being customized. Therefore, we recommend that you download (export) a package that may be used as an archive copy of the current state of the customization project; you may also create an archive copy before saving new changes to the database or before validating the changes. This copy may be useful if the validation process fails. In this case, you can upload the latest archive file to restore the previous state of the customization project.
8. Publish the project.
9. Test the customized application instance.

To Perform Final Testing of a Customization

In this stage, you test the newly developed customization project in the staging environment, which can be a local copy of the production environment. The staging environment must include a copy of the production database.

To apply the customization to the staging environment, you have to do the following:

1. Export the deployment package of the customization project in the development environment. See [To Export a Project](#) for details.
2. Prepare the staging environment in accordance with [System Requirements for Acumatica ERP 6.1](#).
3. In the staging environment, install a version with the same Acumatica ERP version number that is used in the production environment (see [Installing Acumatica ERP](#) for details). All further actions must be performed in the staging environment.
4. Copy the deployment package file to the system where the staging environment was prepared.
5. Start the application instance.
6. Import the project. See [To Import a Project](#) for details.
7. Explore the project content. Ensure that the project items contain appropriate data.
8. Publish the project. See [Publishing Customization Projects](#) for details.
9. Test the customization.

If you have developed a complex project for an application with multiple customization projects published and you can't address some issues that have emerged during this stage, try to perform the original installation with all previous updates, and then upload the saved content of the current project to the staging environment.

To Deploy a Customization Project

In this stage, you deploy the successfully tested customization project to the production environment.

To apply the customization to the production environment, perform the following actions:

1. Export the deployment package of the customization project in the development environment. See [To Export a Project](#) for details.
2. Copy the deployment package file to the file system where the production environment is located.
3. Import the project in the production environment, as described in [To Import a Project](#).

4. Publish the project in the production environment to apply the customization. See [Publishing Customization Projects](#) for details.

Managing Customization Projects

A customization project is a set of changes to the user interface and functionality of Acumatica ERP. A customization project, described more fully in [Customization Project](#), might include the following:

- New custom forms and modifications to existing forms of Acumatica ERP
- Extensions for the business logic
- Custom reports
- Custom application configuration
- Additional files that you need for the customization

In Acumatica ERP, you manage customization projects by means of the [Customization Projects](#) (SM.20.45.05) form, which is shown in the screenshot below. (See also [Customization Projects Form](#) for more information.) On this form, you can add a new customization project, open a customization project for editing in the [Customization Project Editor](#), publish any number of customization projects, cancel the publication of customization projects, export a customization project as the deployment package, import a customization project from an existing deployment package, and delete a customization project.

Published	Project Name	Level	Screen Names	Description	Created By	Last Modified On
<input type="checkbox"/>	AdvCstAEF		PO301000,PO302000		admin	6/23/2016
<input type="checkbox"/>	AdvUI7		IN202500		admin	6/23/2016
<input type="checkbox"/>	CRMAddOn				admin	6/23/2016
<input type="checkbox"/>	FormActionAEF		SO301000		admin	6/23/2016
<input type="checkbox"/>	GridActionsAEF		CS203000		admin	6/23/2016
<input type="checkbox"/>	KeyWords		IN202500		admin	6/23/2016
<input type="checkbox"/>	PurchaseOrderWizard				admin	6/23/2016
<input type="checkbox"/>	RapidByte				admin	6/23/2016
<input checked="" type="checkbox"/>	Yogifon		AR303000,AR409000,SO301000,SO303000		admin	6/23/2016

Figure: Viewing the Customization Projects form



: In MySQL, the maximum size of one packet that can be transmitted to or from a server is 4MB by default. If you use MySQL and want to manage a customization project with the size that is larger than the default maximum value, you have to increase the `max_allowed_packet` variable in the server. The largest possible packet size is 1GB.

In This Part

- [To Create a New Project](#)
- [To Select an Existing Project](#)
- [To Open a Project](#)
- [To Update a Project](#)
- [To Delete a Project](#)
- [To Export a Project](#)
- [To Import a Project](#)

- [To Replace the Content of a Project from a Package](#)
- [To Merge Multiple Projects](#)
- [To Manipulate Customization Projects from the Code](#)

To Create a New Project

Because the Acumatica Customization Platform uses a customization project as a container for the customization items, the platform does not permit you to perform any customization without starting a customization project.

You can create a customization project in the following ways:

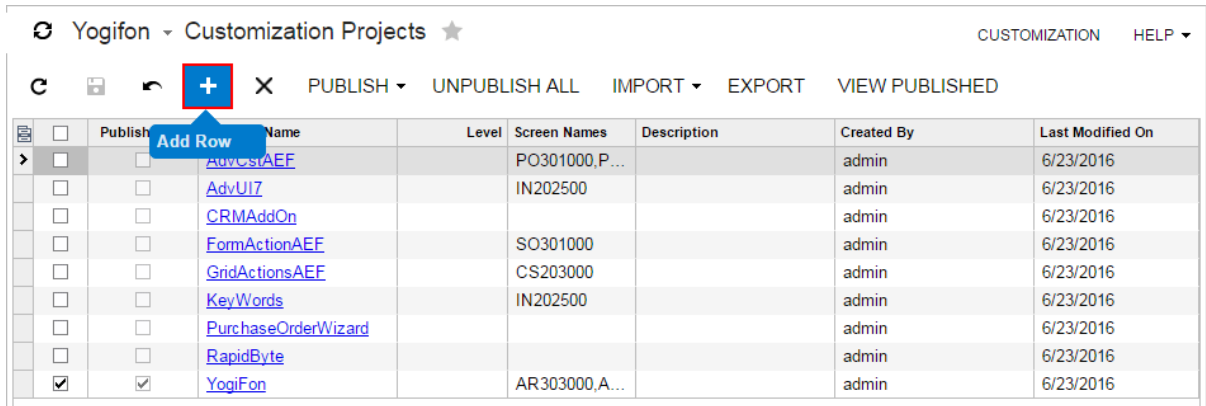
- [On the Customization Projects form](#)
- [From the **Customization** menu](#)
- [From the Element Inspector](#)

There are no differences between these ways because the result is the same: a new customization project with the specified name is added to the Acumatica ERP instance as a record in the database. The platform can then use this project as a container of customization items.

Adding a Customization Project on the Customization Projects Form

To add a customization project on the [Customization Projects](#) (SM.20.45.05) form, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. On the form toolbar, click **Add Row (+)**, as shown in the following screenshot, to add a row to the list of the projects on the form.



	Publish	Name	Level	Screen Names	Description	Created By	Last Modified On
<input type="checkbox"/>	<input type="checkbox"/>	AdvCstAEF		PO301000,P...		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	AdvUI7		IN202500		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	CRMAddOn				admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	FormActionAEF		SO301000		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	GridActionsAEF		CS203000		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	KeyWords		IN202500		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	PurchaseOrderWizard				admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	RapidByte				admin	6/23/2016
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	YogiFon		AR303000,A...		admin	6/23/2016

Figure: Adding a row to the customization project list

3. In the new row, specify a unique name for the project, as the screenshot below shows.
4. Click **Save** on the form toolbar to save the new project in the database.

	Published	* Project Name	Level	Screen Names	Description	Created By	Last Modified On
*	<input type="checkbox"/>	MyProject					
	<input type="checkbox"/>	AdvCstAEF		PO301000,P...		admin	6/23/2016
	<input type="checkbox"/>	AdvUI7		IN202500		admin	6/23/2016
	<input type="checkbox"/>	CRMAddOn				admin	6/23/2016
	<input type="checkbox"/>	FormActionAEF		SO301000		admin	6/23/2016
	<input type="checkbox"/>	GridActionsAEF		CS203000		admin	6/23/2016
	<input type="checkbox"/>	KeyWords		IN202500		admin	6/23/2016
	<input type="checkbox"/>	PurchaseOrderWizard				admin	6/23/2016
	<input type="checkbox"/>	RapidByte				admin	6/23/2016
	<input checked="" type="checkbox"/>	YogiFon		AR303000,A...		admin	6/23/2016

Figure: Entering the new project name

Creating a Customization Project from the Customization Menu

To create a customization project from the *Customization Menu*, perform the following actions:

1. On any form of Acumatica ERP, click **Customization > Select Project**, as the following screenshot shows (item a).
2. In the *Select Customization Project Dialog Box*, which opens, click **New** (item b).
3. In the *New Project Dialog Box*, which opens, enter the name of the new project (item c).
4. In this dialog box, click **OK** (item d) to create the project in the database and close the dialog box.

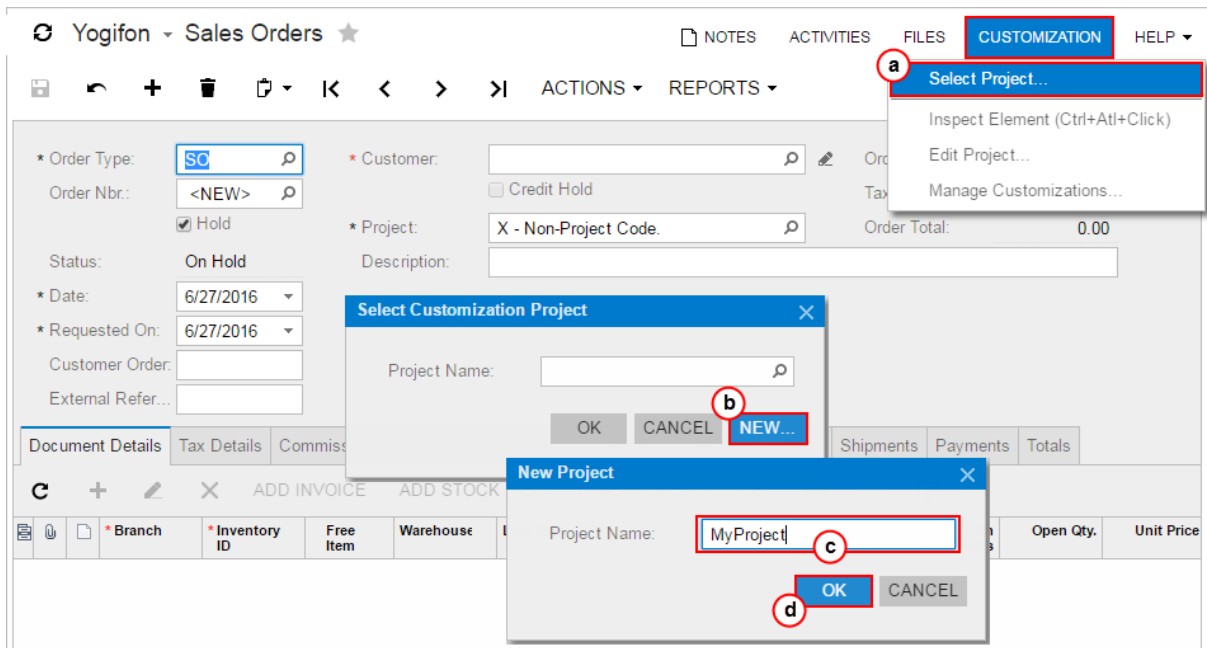


Figure: Creating a customization project from the Customization menu

Creating a Customization Project from the Element Inspector

Suppose that you have not selected a customization project. In the *Element Properties Dialog Box*, if you have tried to start performing customization by clicking **Customize, Actions >**

Customize Business Logic, or **Actions > Customize Data Fields**, the inspector opens the **Select Customization Project** dialog box to force you to select or create a customization project.

To create a customization project from this dialog box, perform the following actions:

1. In the dialog box, click **New**.
2. In the **New Project** dialog box, which opens, enter the name for the new project (item c).
3. In this dialog box, click **OK** to create the project in the database and close the dialog box.

To Select an Existing Project

Because the Acumatica Customization Platform uses a customization project as a container for the customization items, the platform does not permit you to perform any customization without starting a customization project.

You can select an existing customization project in the following ways:

- *By selecting it from the **Customization** menu*
- *By selecting it from the **Element Inspector***

To Select a Customization Project from the Customization Menu

To create a customization project from the *Customization Menu*, perform the following actions:

1. On any form of Acumatica ERP, click **Customization > Select Project**, as the following screenshot shows (item a).
2. In the *Select Customization Project Dialog Box*, which opens, click the **Project Name** selector and select a project name (items b and c).
3. In this dialog box, click **OK** (item d) to select the project and close the dialog box.

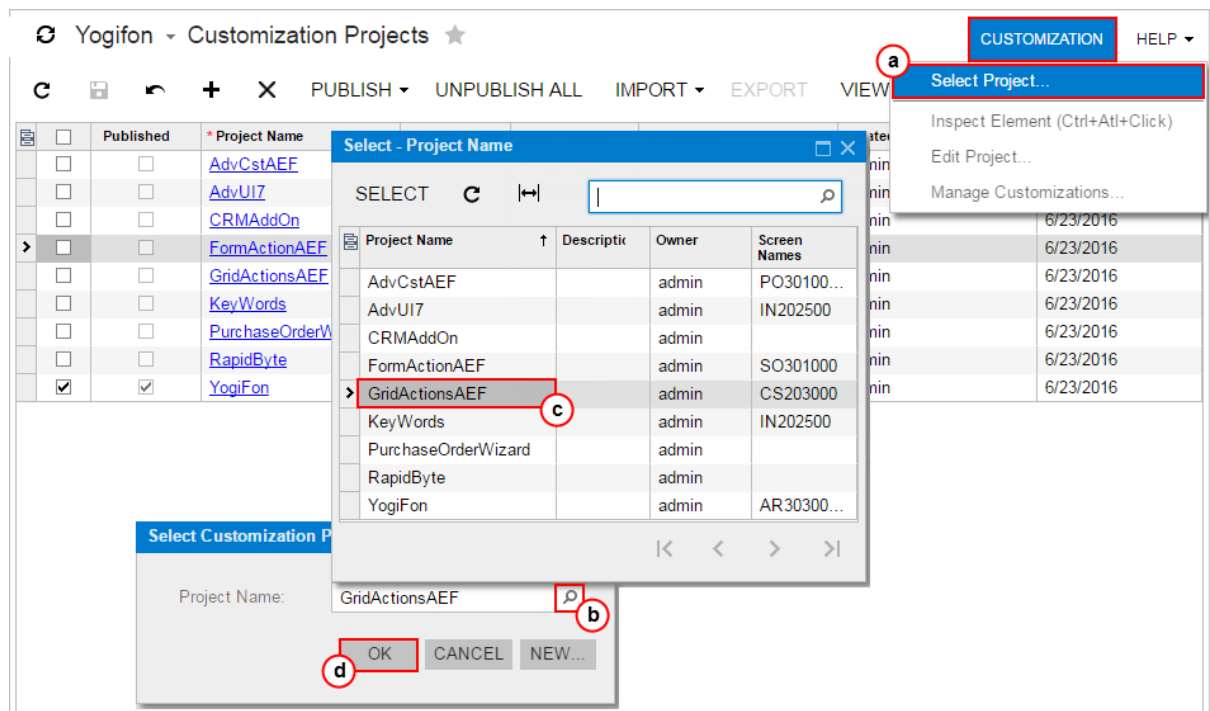


Figure: Selecting a customization project from the Customization menu

To Select a Customization Project from the Element Inspector

Suppose that you have not selected a customization project. If in the [Element Properties Dialog Box](#), you have tried to perform customization by clicking **Customize, Actions > Customize Business Logic**, or **Actions > Customize Data Fields**, the inspector opens the **Select Customization Project** dialog box to force you to select a customization project.

To select a customization project, in the **Select Customization Project** dialog box, perform the following actions:

1. In the **Project Name** selector, select a project name.
2. Click **OK** to select the project and to close the dialog box.

To Open a Project

If you have selected a customization project, you can open the project for editing in the [Customization Project Editor](#) in the following ways:

- On the [Customization Projects Form](#) : Click the project name in the table.
- From the [Customization Menu](#): Click **Customization > Edit Project**.
- From the [Element Properties Dialog Box](#): Click **Customize, Actions > Customize Business Logic**, or **Actions > Customize Data Fields**.

To Update a Project

If you have modified an item of a customization project in the file system by using an integrated development environment (IDE), such as Microsoft Visual Studio, these changes may not be reflected in the customization project yet. You have to update the project in the database before the publication or export of the deployment package of the project.

To update a customization project, perform the following actions:

1. Open the project in the [Customization Project Editor](#). (See [To Open a Project](#) for details.)
2. In navigation pane of the editor, select **Files** to open the Custom Files page.
3. On the page toolbar, click **Detect Modified Files**, as shown in the screenshot below, to review the files included in the project that could have been modified in the file system but haven't been updated in the customization project yet.

If at least one conflict is detected, the **Modified Files Detected** dialog box opens (also shown in the following screenshot). The dialog box displays each changed file with the check boxes in the **Selected** and **Conflict** columns selected. See [Detecting the Project Items Modified in the File System](#) for details.

4. To update the selected files in the project, on the toolbar of the dialog box, click **Update Customization Project**, as the following screenshot shows.

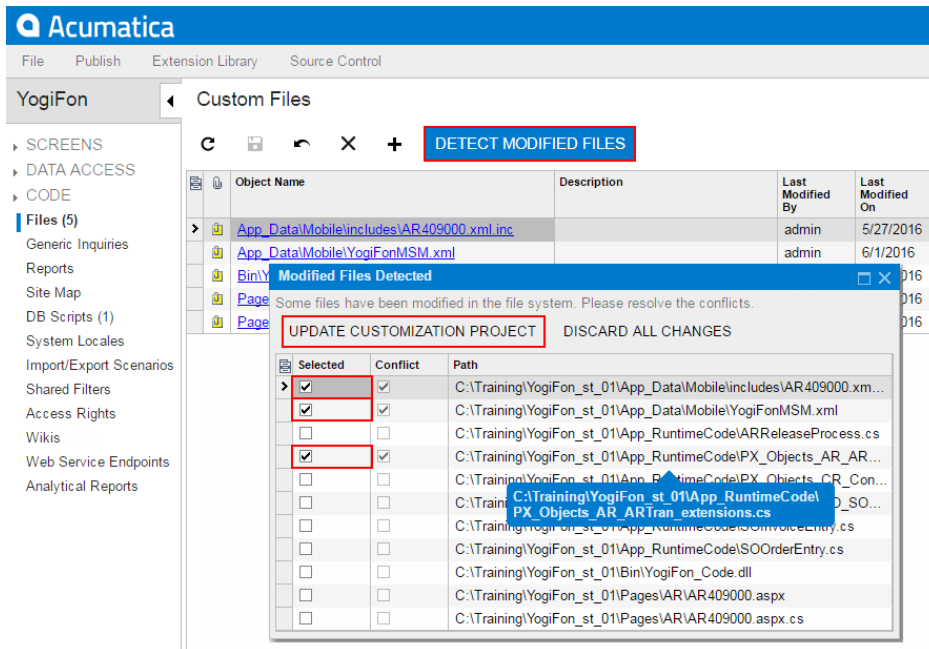


Figure: Updating files in the project

To Delete a Project

If a customization project is not published, you can delete the project from the instance of Acumatica ERP by using the *Customization Projects* (SM.20.45.05) form.

To delete a customization project, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. In the project list, select the row of the project that has to be deleted, as shown in the screenshot below (item a).
3. On the form toolbar, click **Delete Row (X)**, item b in the screenshot, to delete the selected project from the list.
4. On the form toolbar, click **Save** (item c in the screenshot) to save the changes to the database.

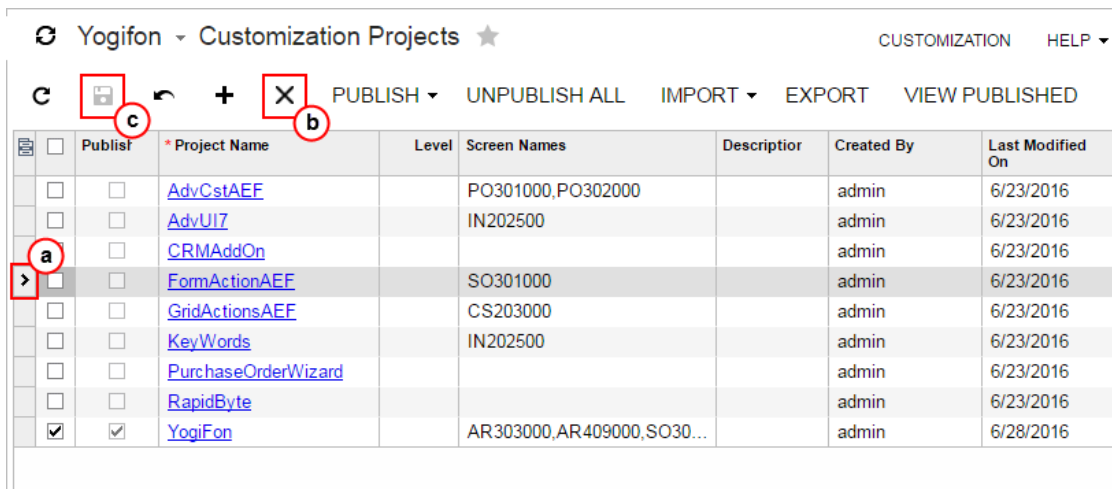


Figure: Deleting a customization project

The platform deletes from the database the project data and the data of the project items. The platform does not delete the files and the objects that were added to the projects from the database, such as site map nodes, reports, user access rights, and integration scenarios.

To Export a Project

You can export (download) a customization project when the project is finished to deploy the customization to the target system. Also, you can download the package to have a backup copy of the customization project you are working on.

Before you download the package, we recommend that you make sure you have included all the needed changes in the customization project. To do this, you take the following actions:

- Make sure that you have added all custom files to the project and uploaded the latest actual version of the files to the project. See [To Update a Project](#) for details.
- Make sure that the database schema is updated in the customization project. See [To Update Custom Tables in the Project](#) for details.
- Make sure you have added the needed site map nodes to the project. See [Site Map](#) for details.
- Publish the project and test the customization before downloading the deployment package, to ensure that you have no issues.

To download the deployment package of a customization project, you should export the project. You can export a customization project in the following ways:

- [By using the Customization Projects form](#)
- [Through the Customization Project Editor](#)

The system creates the deployment package of the project and downloads the zip file of the package on your machine. The file has the same name as the customization project. For more information about a deployment package, see [Deployment of Customization](#).

Exporting a Customization Project by Using the Customization Projects form

To export a customization project by using the [Customization Projects](#) (SM.20.45.05) form, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. In the project list of the form, click the row of the customization project to be exported.
The row is highlighted in the table, as the screenshot below shows.
3. Click **Export** on the form toolbar to export the highlighted project.

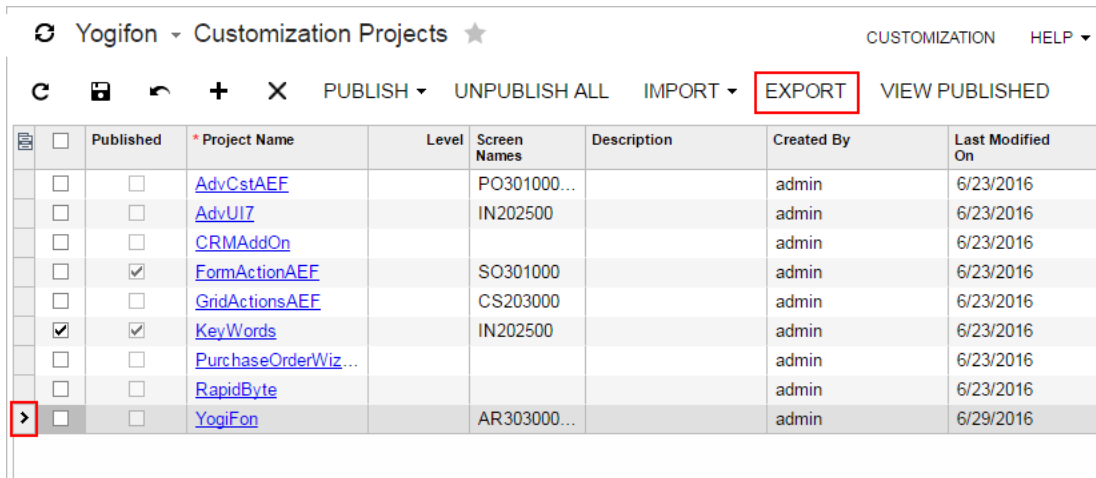


Figure: Exporting a customization project

Exporting the Customization Project Opened in the Customization Project Editor

To export the customization project that is currently opened in the *Customization Project Editor*, click **File > Export Project Package** on the editor menu, as shown in the following screenshot.

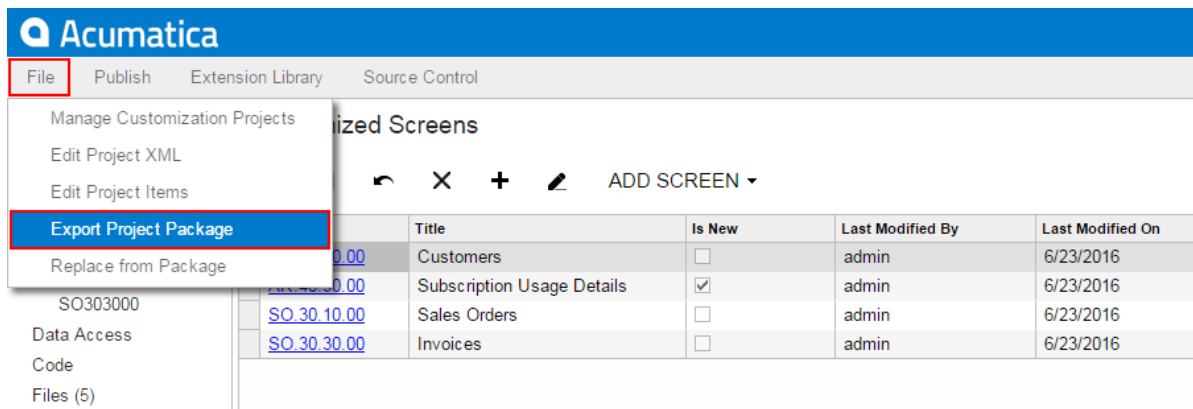


Figure: Exporting the current customization project

Also, you can export the customization project from the *Project XML Editor* of the *Customization Project Editor* by clicking **Download Package** on the page toolbar, as shown in the following screenshot.

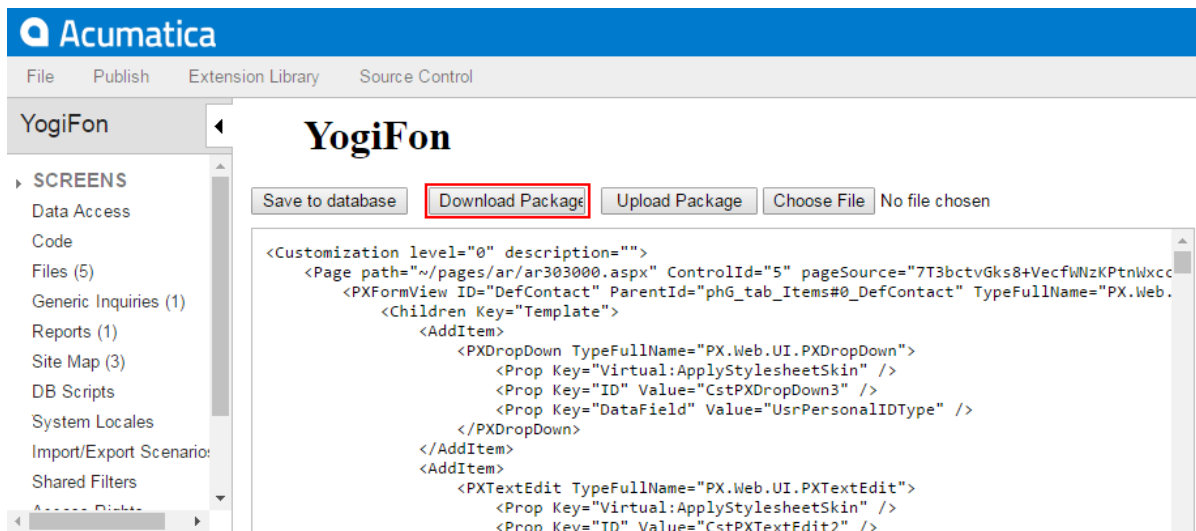


Figure: Exporting the current customization project from the Project XML Editor

To Import a Project

You can import a deployment package to work with the customization project or to publish the final customization on the target website.



: You can import deployment packages of earlier versions of Acumatica ERP to Acumatica ERP 6.1.

To upload the deployment package of a customization project, you should import the project by using the [Customization Projects](#) (SM.20.45.05) form. (See [Customization Projects Form](#) for more information.)

To do this, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. Click **Import** on the form toolbar, as the screenshot below shows.
3. In the **Open Package** dialog box, which opens, click **Choose File**.
4. In the **Open** dialog box, which opens, select the deployment package file to be uploaded.

The name of the selected file is displayed in the **File path** box of the **Open Package** dialog box, as shown in the screenshot.

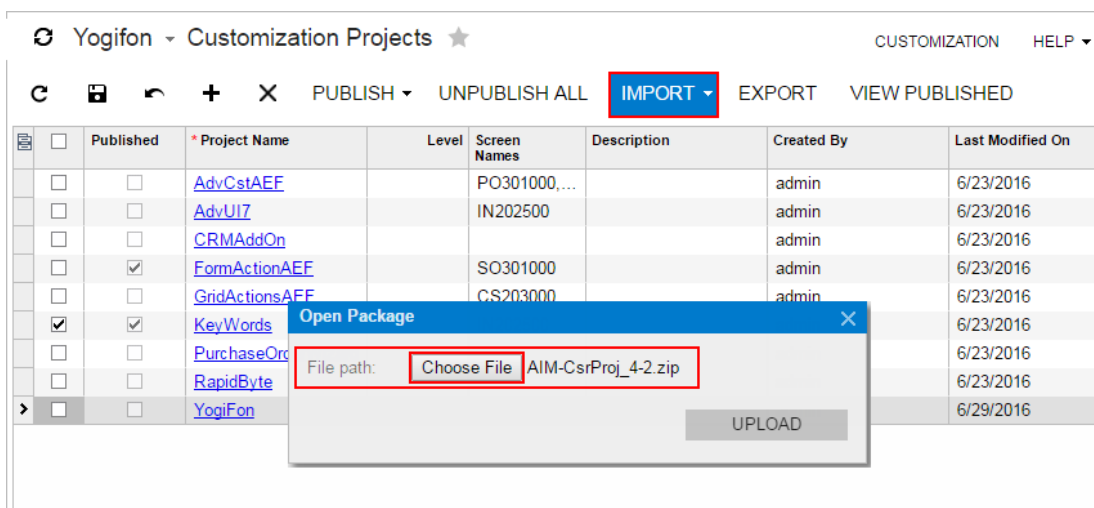


Figure: Importing a customization project

5. In the **Open Package** dialog box, click **Upload**.

The platform uploads the selected package, create the corresponding customization project, and saves the project in the database. As the result, the new customization project appears in the list on the [Customization Projects](#) form; therefore, you can access the project data and manage the project.

To Replace the Content of a Project from a Package

You might need to upload a newer version of the customization project that has been modified outside the system on which you are working, or upload the project from a backup copy.

You can replace the content of a customization project from a deployment package in the following ways:

- [By using the Customization Projects form](#)
- [Through the Customization Project Editor](#)



: The platform does not verify the content of a deployment package before replacing a customization project in the database.

Replacing a Customization Project from a File by Using the Customization Projects Form

To replace the content of a customization project from a file by using the the [Customization Projects](#) (SM.20.45.05) form (see also [Customization Projects Form](#)), perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. In the project list, click the row of the customization project to be updated, as the screenshot below shows.
3. Click **Import > Replace Highlighted Project Content** on the form toolbar, as the screenshot below also shows.
4. In the **Open Package** dialog box, which opens, click **Choose File**.
5. In the **Open** dialog box, which opens, select the deployment package file to be uploaded.

The name of the selected file is displayed in the **File path** text box of the **Open Package** dialog box, as shown in the screenshot.

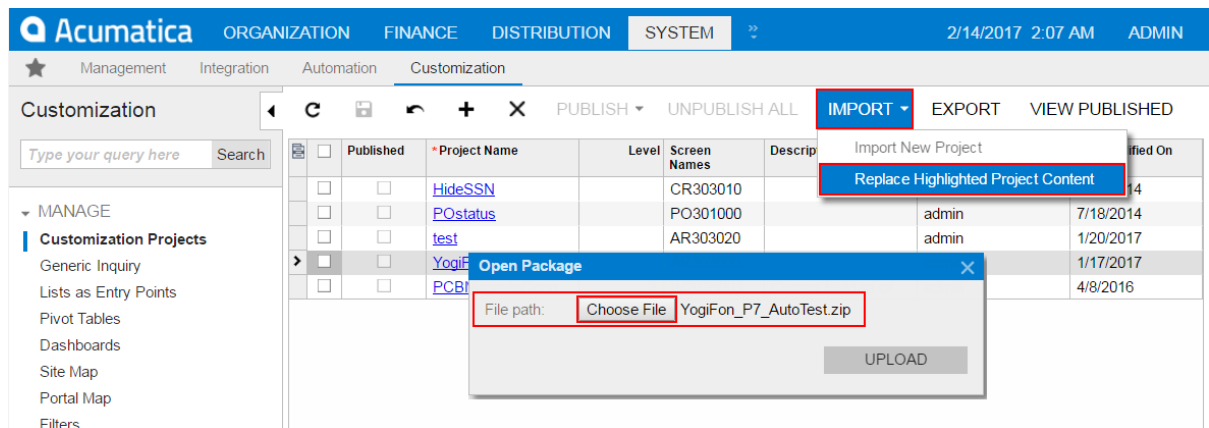


Figure: Replacing the content of a customization project

6. In the **Open Package** dialog box, click **Upload**.

The platform uploads the selected package and replaces in the database the content of the selected project with the content of the deployment package.

Replacing the Customization Project Opened in the Customization Project Editor from a File

To replace the content of a customization project that is currently opened in the [Customization Project Editor](#), perform the following actions:

1. Click **File > Replace from Package** on the editor menu, as shown in the following screenshot.

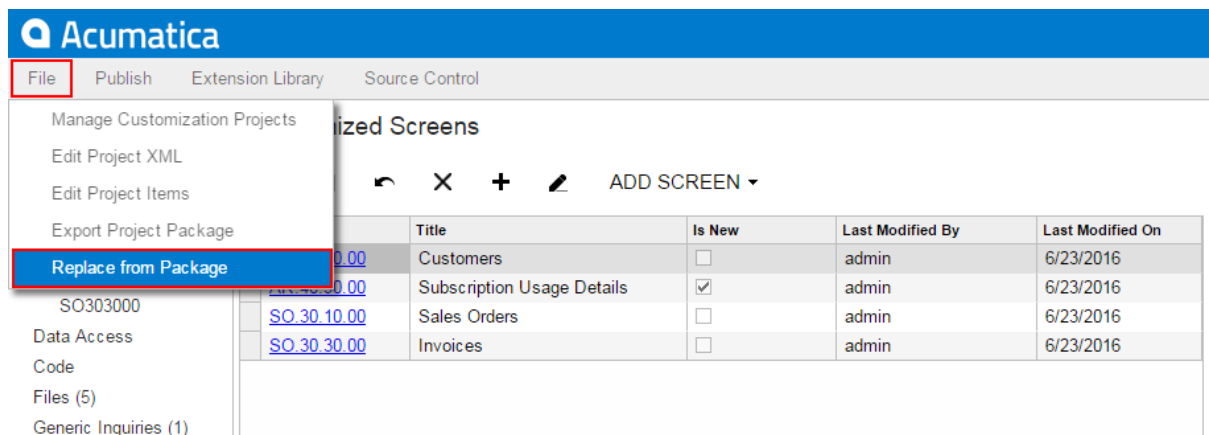


Figure: Replacing the content of the current customization project

2. In the **Open Package** dialog box, which opens, click **Choose File**.
3. In the **Open** dialog box, which opens, select the deployment package file to be uploaded.

The name of the selected file is displayed in the **File path** text box of the **Open Package** dialog box.

4. In the **Open Package** dialog box, click **Upload**.

The platform uploads the selected package and replaces in the database the content of the selected project with the content of the deployment package.

Also, you can replace the content of the customization project by using the [Project XML Editor](#) of the Customization Project Editor. To do this, perform the following actions:

1. In the Customization Project Editor, click **File > Edit Project XML** to open the Project XML Editor for the current customization project.
2. On the toolbar of the Project XML Editor, click **Choose File**.
3. In the **Open** dialog box, which opens, select the deployment package file to be uploaded.
The name of the selected file is displayed in the text box right of the toolbar buttons instead of the *No file chosen* string (which is shown in the screenshot below).
4. On the editor toolbar, click **Upload Package**.

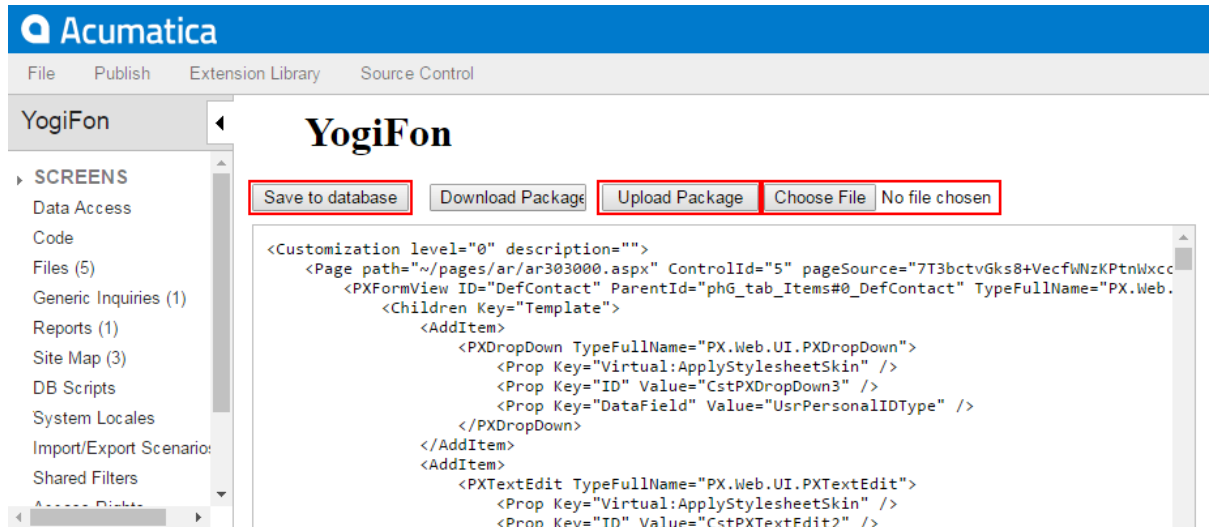


Figure: Replacing the content of a customization project from the Project XML Editor

The platform uploads the selected package and displays its XML code in the Project XML Editor.

5. Explore the content of the uploaded package to ensure that this is the needed one.
6. To replace the content of the project that is currently opened in the Customization Project Editor with the content of the uploaded package, click **Save to Database** on the page toolbar.

By using this approach, you can explore the content of the package before replacing the content of the customization project that is currently opened in the Customization Project Editor.

To Merge Multiple Projects

As a rule, it is better to have multiple customization projects instead of a single one. But if you have two or more customization projects that contain changesets with the same items, and if you are sure that each project is valid and that the merged customization applies to the website properly, we recommend that you merge the projects. You use the [Customization Projects](#) (SM.20.45.05) form as a starting point.

To merge multiple customization projects, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. In the project list, select the check boxes for only the projects to be merged.
3. Click **Publish** on the form toolbar.
4. Click **View Published** on the form toolbar to open the [Published Customization Page](#).
5. Click **Download Package** on the page toolbar to download the `Customization.zip` file, which includes the full content of the merged customization project.

The `Customization.zip` file is the deployment package of the merged customization project. You can use the merged project to publish the final customization project on the target website. You can rename the `.zip` file to assign the needed name to the customization project. To upload the deployment package to a target system, import the `.zip` file, as described in [To Import a Project](#).

To Manipulate Customization Projects from the Code

You can use the web service API for manipulating customization projects from the code. Methods of the web service API can be used, for example, to deploy customization packages to local and remote instances of Acumatica ERP.

For manipulating customization projects, the web service API includes the methods, which are accessible through the SOAP API, described in the following topics:

- [GetPackage\(\) Method](#)
- [PublishPackages\(\) Method](#)
- [UnpublishAllPackages\(\) Method](#)
- [UploadPackage\(\) Method](#)



: A user of an application that invokes any of these methods should be assigned the *Customizer* role in the appropriate instance of Acumatica ERP. See [To Assign the Customizer Role to a User Account](#) for details.

The web service used for manipulating customization projects is available under the URL, which is specified in one of the following ways (see the screenshot below):

- `http://<Computer Name>/<Website Name>/api/servicegate.asmx`, such as `http://MyComputer/YogiFon/api/servicegate.asmx`
- `http://<IP Address>/<Website Name>/api/servicegate.asmx`, such as `http://111.222.3.44/YogiFon/api/servicegate.asmx`

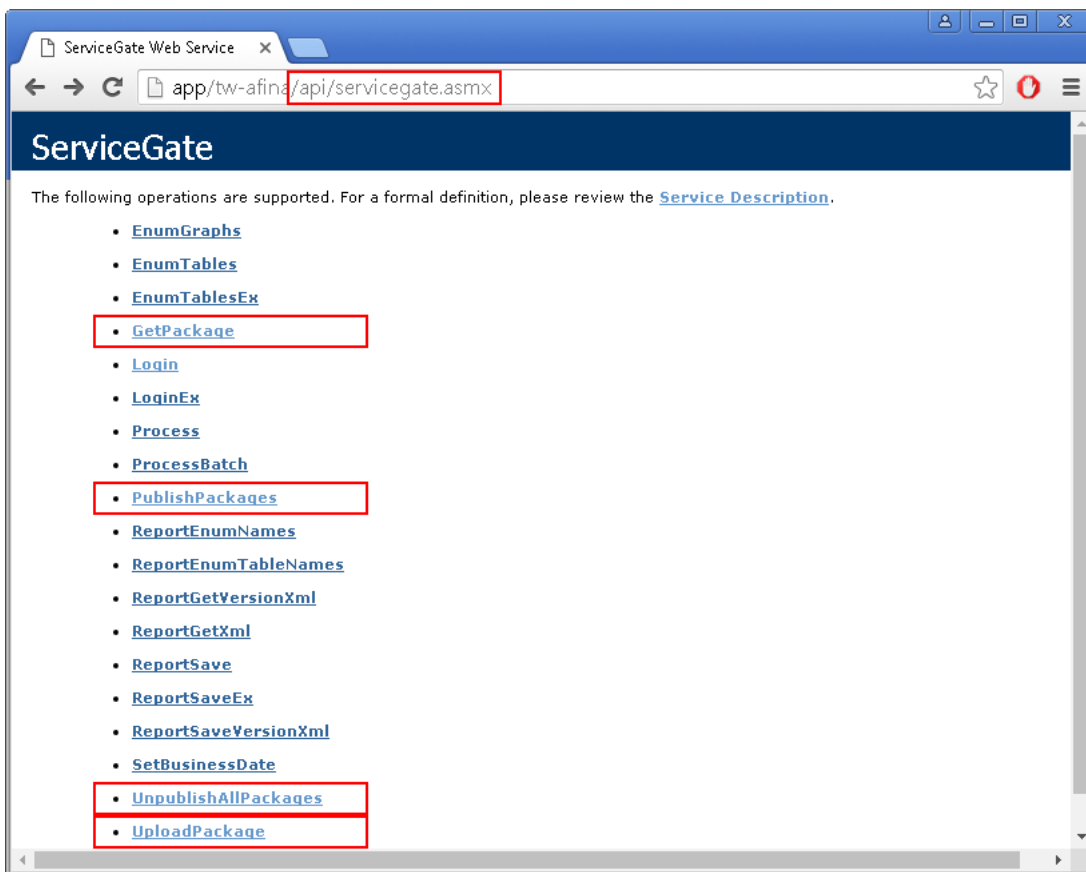


Figure: Accessing the web service through the browser

To use the listed methods in the code, you should create a service gate for the specified URL and log in, as shown in the following code fragment.

```
var webserviceurl = "http://localhost/AcumaticaInstance/api/ServiceGate.asmx";
var username = "admin";
var password = "123";

var client = new ServiceGate.ServiceGate
{
    Url = webserviceurl,
    CookieContainer = new CookieContainer(),
    Timeout = (int) TimeSpan.FromMinutes(5).TotalMilliseconds
};

var loginResult = client.Login(username, password);

if (loginResult.Code == ErrorCode.OK)
{
    // Use the methods to manipulate customization projects
}
```

GetPackage() Method

You use the `GetPackage()` method to get the content of a customization project from the database of an Acumatica ERP instance.

Syntax

```
public byte[] GetPackage(string packageName)
```

Parameters

- `packageName`: The name of the customization project to be loaded from the database.

Return Value

The method returns an array of bytes that corresponds to the content of the deployment package `.zip` file.

Example

The following code logs in to an Acumatica ERP instance, gets the content of the `package1` customization project from the database, and returns the content as a byte array. You can then, for example, save the array to a `.zip` file to download the deployment package in the file system.

```
var webserviceurl = "http://localhost/AcumaticaInstance/api/ServiceGate.asmx";
var username = "admin";
var password = "123";
var packageName = "package1";

var client = new ServiceGate.ServiceGate
{
    Url = webserviceurl,
    CookieContainer = new CookieContainer(),
    Timeout = (int) TimeSpan.FromMinutes(5).TotalMilliseconds
};

var loginResult = client.Login(username, password);

if (loginResult.Code == ErrorCode.OK)
{
    var packageContents = client.GetPackage(packageName);
    //do something with the packageContents package content
}
```

Usage Notes

If there are multiple companies in an instance of Acumatica ERP, this method affects the company to which the service is logged in. To log in to a specific company, add the company name to the user name using the following format: `user@MyCompany`.

PublishPackages() Method

You use the `PublishPackages()` method to publish multiple customization projects that exist in the database of an Acumatica ERP instance.

Syntax

```
public void PublishPackages(string[] packageNames, bool mergeWithExistingPackages)
```

Parameters

- `packageNames`: An array of names of the customization projects to be published.
- `mergeWithExistingPackages`: An indicator of whether the specified customization projects must be merged with the customization projects that are currently published in the same instance of Acumatica ERP. If the value of the parameter is `true` and there are published customization projects in the instance, the platform merges the content of the projects specified in the method with the content of the currently published projects and then applies the merged customization to the instance. If the value of the parameter is `false`, the platform cancels the currently applied customization and publishes only the projects specified in the method.

Example

The following code logs in to an Acumatica ERP instance and publishes the *package1*, *package2*, and *package3* customization projects that exist in the database of the instance.

```
var webserviceurl = "http://localhost/AcumaticaInstance/api/ServiceGate.asmx";
var username = "admin";
var password = "123";
var packageNames = new []{"package1", "package2", "package3"};
var mergeWithExistingPackages = true;

var client = new ServiceGate.ServiceGate
{
    Url = webserviceurl,
    CookieContainer = new CookieContainer(),
    Timeout = (int) TimeSpan.FromMinutes(5).TotalMilliseconds
};

var loginResult = client.Login(username, password);

if (loginResult.Code == ErrorCode.OK)
{
    client.PublishPackages(packageNames, mergeWithExistingPackages);
}
```

Usage Notes

If there are multiple companies in an instance of Acumatica ERP, this method affects the company to which the service is logged in. To log in to a specific company, add the company name to the user name using the following format: *user@MyCompany*.

UnpublishAllPackages() Method

You use the `UnpublishAllPackages()` method to cancel customization of an Acumatica ERP instance.

Syntax

```
public void UnpublishAllPackages()
```

Example

The following code logs in to an Acumatica ERP instance and cancels all customization projects that is currently applied to the instance.

```
var webserviceurl = "http://localhost/AcumaticaInstance/api/ServiceGate.asmx";
var username = "admin";
var password = "123";

var client = new ServiceGate.ServiceGate
{
    Url = webserviceurl,
    CookieContainer = new CookieContainer(),
    Timeout = (int) TimeSpan.FromMinutes(5).TotalMilliseconds
};

var loginResult = client.Login(username, password);

if (loginResult.Code == ErrorCode.OK)
{
    client.UnpublishAllPackages();
}
```

Usage Notes

The method removes all customization changes from the file system, regardless of the companies where the customization projects were published.

UploadPackage() Method

You use the `UploadPackage()` method to import a customization project from the file system. The method saves the content of an uploaded deployment package to the database of an instance of Acumatica ERP.

Syntax

```
public void UploadPackage(string packageName, byte[] packageContents,
                        bool replaceIfPackageExists)
```

Parameters

- `packageName`: The name of the customization project to be saved in the database.
- `packageContents`: An array of bytes that contains the content of the deployment package `.zip` file and will be saved in the database.
- `replaceIfPackageExists`: An indicator of whether the customization platform must replace an existing customization project with the same name in the database. If the value of this parameter is `true` and the database contains a customization project with the same name, the platform replaces the project with the specified content. If the value of the parameter is `false` and the database contains a customization project with the same name, an exception occurs.

Example

The following code logs in to an Acumatica ERP instance, loads the `C:\package1.zip` file content, and saves or updates the `package1` customization project in the database of the instance.

```
var webserviceurl = "http://localhost/AcumaticaInstance/api/ServiceGate.asmx";
var username = "admin";
var password = "123";
var packageName = "package1";
var packageContents = File.ReadAllBytes("C:\package1.zip");
var replaceIfPackageExists = true;

var client = new ServiceGate.ServiceGate
{
    Url = webserviceurl,
    CookieContainer = new CookieContainer(),
    Timeout = (int) TimeSpan.FromMinutes(5).TotalMilliseconds
};

var loginResult = client.Login(username, password);

if (loginResult.Code == ErrorCode.OK)
{
    client.UploadPackage(packageName, packageContents, replaceIfPackageExists);
}
```

Usage Notes

If there are multiple companies in an instance of Acumatica ERP, this method affects the company to which the service is logged in. To log in to a specific company, add the company name to the user name using the following format: `user@MyCompany`.

Publishing Customization Projects

To apply a customization project to an instance of Acumatica ERP, you have to publish the customization project. You can also publish multiple customization projects at once; see [Simultaneous Use of Multiple Customizations](#) for details.

When you publish a customization project, the system applies the changes in the project to the website. After the customization project has been published, users see the modified Acumatica ERP. The changes apply to the website of Acumatica ERP and therefore affect all company tenants in the system (see [Customization of a Multi-Company Site](#) for details).

The Acumatica Customization Platform provides the following ways to manage the publication process:

1. You can develop and include in a customization project the custom code that is executed during the project publication. See [Custom Processes During Publication of a Customization](#) for details.
2. By using additional attributes in DAC extensions, you can specify how the system should apply the original and custom attributes to the field. See [Customization of Field Attributes in DAC Extensions](#) for details.
3. By using SQL script attributes, you can control the execution of batches in SQL scripts. See [Using the SQL Script Attributes](#) for details.

You can cancel the publication of the project and publish the project again, which you might do often during the development and testing of the customization. The application domain does not restart every time you publish the customization project because [Run-Time Compilation](#) is enabled for the website by default.

Detailed instructions are provided in the following topics:

- [To Publish a Single Project](#)
- [To Publish Multiple Projects](#)
- [To Prepare a Project for Publication](#)
- [To Publish the Current Project](#)
- [To Publish the Current Project with a Cleanup Operation](#)
- [To Publish a Customization for a Multi-Company Site](#)
- [Validating Customization Code](#)
- [To View a Published Customization](#)
- [To Unpublish a Customization](#)

To Prepare a Project for Publication

Before you publish a customization project, we recommend that you make sure you have included all the needed changes in the customization project. To do this, you should take the following actions:

- Make sure that you have added all custom files to the project and uploaded the latest version of the files to the project.
- Make sure that the database schema is updated in the customization project. You could have modified custom tables by using a database management tool, such as SQL Management Studio. On the DB Scripts page toolbar, click **Update From Database**. This procedure regenerates the database table schema of the custom tables that have the **Import Table Schema from Database** check box selected. See [SQL Script Editor](#) for details.

- Make sure you have added the needed site map nodes to the project. Open the Site Map page and add any needed site map nodes to the project.
- Make sure that all other objects (generic inquiries, system locales, integration scenarios, shared reusable filters, access rights, wikis, web-service endpoints, and analytical reports) are updated in the project. If an object was changed in the application instance, open the appropriate page of the [Customization Project Editor](#) and click **Reload From Database** on the page toolbar to update the corresponding item in the project. (See [To Update a Project](#) for details.)

To Publish a Single Project

You can publish a single customization project by using the **Publish** action on the [Customization Projects](#) (SM.20.45.05) form.

To do this, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. In the project list, select the check box (in the unlabeled column) for the needed customization project, as the screenshot below shows.
3. Clear any selected check boxes in this column for other customization projects.



: All previously published projects that are not selected will be unpublished.

4. Click **Publish** on the form toolbar to initiate the publication of the selected project.

	Published	* Project Name	Level	Screen Names	Description	Created By	Last Modified On
<input type="checkbox"/>	<input type="checkbox"/>	AdvCstAEF		PO301000,PO302000		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	AdvUI7		IN202500		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	CRMAddOn				admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	FormActionAEF		SO301000		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	GridActionsAEF		CS203000		admin	6/23/2016
<input checked="" type="checkbox"/>	<input type="checkbox"/>	KeyWords		IN202500		admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	PurchaseOrderWizard				admin	6/23/2016
<input type="checkbox"/>	<input type="checkbox"/>	RapidByte				admin	6/23/2016
<input type="checkbox"/>	<input checked="" type="checkbox"/>	YogiFon		AR303000,AR409000,SO30...		admin	6/28/2016

Figure: Publishing a single customization project

To Publish Multiple Projects

To publish multiple customization projects, you perform the following actions on the [Customization Projects](#) (SM.20.45.05) form:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. In the project list, select the check boxes (in the unlabeled column) for the customization projects you want to publish.



: You can save these selections by clicking **Save** on the form toolbar. The form opens with the selections that are previously saved in the database.

3. Click **Publish** on the form toolbar to initiate publication of the selected projects.



: All previously published projects that are not selected will be unpublished.

The platform merges the selected projects into one project and then publishes the project. For more details, see [Performing the Publication Process](#) and [Validating Customization Code](#).

Performing the Publication Process

When you run the publication process, the Acumatica Customization Platform executes the process in the following stages:

1. If you publish more than one customization project, the platform merges the projects into a single customization project.
2. The platform validates the customization project. (See [Validating Customization Code](#) for details.)
3. If the validation completes successfully, the platform applies the changes to the application instance when you click **Publish** in the **Compilation** window.

When the platform merges multiple projects, if different projects include customization for the same application object, the customization from the project with the highest level (an optional number assigned to each project) is added to the merged project. See [Simultaneous Use of Multiple Customizations](#) for details.



: If you manage multiple customization projects and some projects are published, you do not have to unpublish any projects before other projects are published (leaving the check boxes selected for already-published projects in the list). On the [Customization Projects](#) (SM.20.45.05) form, you just need to select any projects you want to publish and click **Publish** on the toolbar. If you clear the check box for a published project, it will be unpublished.

After the publication is complete, you can view the content of the merged customization project by clicking the **View Published** button on the form toolbar of the [Customization Projects](#) form. (See [To View a Published Customization](#) for details.)

After the project is ready, the platform validates and compiles the customization code included in the project. If there are any validation errors, the system displays the error messages in the browser and doesn't compile the code. If the compilation completes successfully, you can click **Publish** in the **Compilation** window to apply the changes contained in the project items to the website.

During the actual publication, the platform applies the changes to the application and database objects and updates the files in the website folder as follows:

- The custom layout is applied to the forms of Acumatica ERP.
- The .cs files with the DAC extension code for the existing data access classes are generated and placed in the file system.
- The .cs files with the BLC extension code for the existing business logic are generated and placed in the file system.
- The .cs files with the custom code (Code items) are generated and placed in the file system.
- The custom files of the project are added to the website folder.
- The custom tables are created in the database, and custom SQL scripts are executed.
- The custom generic inquiries, reports, site map nodes, system locales, integration scenarios, shared reusable filters, access rights, wikis, web-service endpoints, and analytical reports are added to the database.



: We recommend that you back up the database before you publish customization projects, because canceling publication doesn't revert changes made to the database. See [To Unpublish a Customization](#) for details.

After the publication is complete, the application domain always restarts if the project includes assemblies, which are placed in the `/Bin` folder of the website. If you have no assemblies in the project, you can enable run-time compilation and publication will not cause a restart.

If a published customization project contains classes derived from the `CustomizationPlugin` class, the platform launches the implemented custom processes after website files were updated and after the website was restarted. (See [Custom Processes During Publication of a Customization](#) for details.)

After you have published the customization project, the files with the customization code are updated in the file system, and you can work with them in MS Visual Studio. The `.cs` files with code are placed in the `App_RuntimeCode` folder of the website.

Validating Customization Code

While the Acumatica Customization Platform processes publication of a customization project, the platform validates the customization code included in the project. This validation of the code provides not only checking for syntax and semantics but also checking of the compatibility of the code included in the customization project with the original application code.

If you have a customization project that works properly for the current version of Acumatica ERP and have upgraded an application instance to a newer version, the customization project might not work properly or might even prevent the website from starting after the upgrade. This could occur because the code of Acumatica ERP is continuously developed to implement new features or enhance existing functionality. Thus, the code of an updated instance of Acumatica ERP can become inconsistent with the code in a customization project. For example, if the signature of a method that is overridden in the customization code is changed in the original code, a run-time error may occur in the graph extension. As another example, modified or deleted database columns and tables might cause the functionality of a data access class extension to fail.

The platform checks the compatibility of the code included in a customization project with the original application code every time it publishes the project. If there are any compatibility errors, the platform displays the warning and error messages in the **Compilation** window and stops the publication process. See [Messages for Validation Errors](#) and [To Resolve an Issue Discovered During the Validation](#) for detailed information about, respectively, the error messages and the ways to fix the validation errors.

To Publish the Current Project

If you create a customization project in the [Customization Project Editor](#), the best way to publish the customization project that is under development is to use the **Publish Current Project** menu command provided by the editor. This publication does not influence the publication of other customization projects that exist in the application instance.

To publish the customization project that is currently open in the Customization Project Editor, perform one of the following actions:

- Use the Control+Space combination on the keyboard.
- In the editor menu, select **Publish > Publish Current Project**.

When you perform one of these actions, the platform initiates the publication of the project. If another customization project is already published, the platform merges the projects into a single project, and then compiles and validates the merged project. If the validation succeeds, the platform applies the merged customization to the application instance.

You can publish the customization project as described above again after you have made any change to the project.

To Publish the Current Project with a Cleanup Operation

If a customization project contains a database script, during the project publication, the platform executes the script. For optimization purposes, to avoid the execution of database scripts during every publication of the project, the platform saves information about each script that has been executed at least once and has not yet been changed in the database, and omits the repeated execution of such scripts. You can force the platform to clean up all such information about previously executed scripts of a customization project and execute the scripts once more while publishing the project.

To do this, perform the following actions:

1. Open the project in the *Customization Project Editor* (see *To Open a Project* for details).
2. On the menu of the editor, click **Publish > Publish with Cleanup**, as shown in the screenshot below, to clean information about previously executed scripts of the project and initiate the process of publishing the customization project.

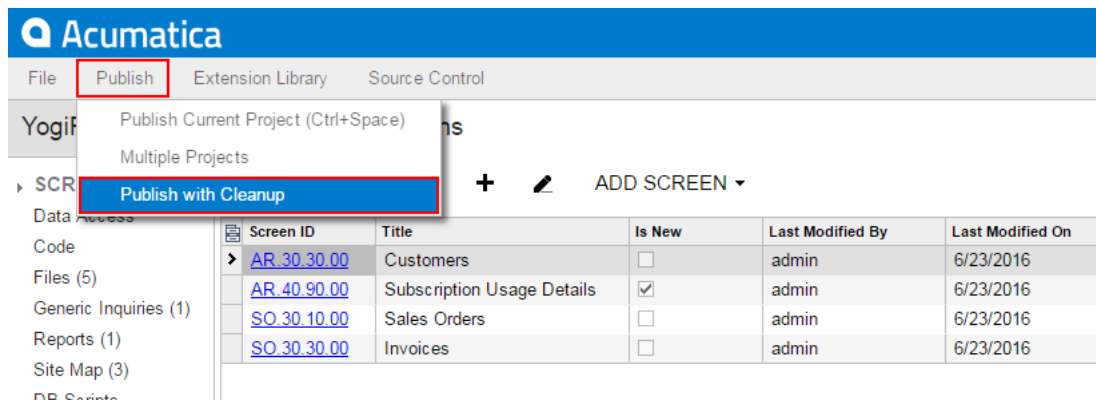


Figure: Publishing a customization project with a cleanup operation

To Publish a Customization for a Multi-Company Site

You can publish a customization project for multiple companies in a multi-company site; see *Customization of a Multi-Company Site* for details. You use the *Customization Projects* (SM.20.45.05) form to publish this customization.

To share customization content stored exclusively in the database for multiple companies, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. Select the check boxes that correspond to the customization projects that you need to publish for multiple companies, as the screenshot below shows (item a).
3. On the toolbar of the form, click **Publish > Publish to Multiple Companies** (item b).
4. In the **Publish to Multiple Companies** dialog box, which opens, do the following:
 - a. Select the companies to which you want to apply the selected customization projects (item c).
 - b. To skip updating the site files, select the **Database Only** check box (item d).
 -  : If you have published all the selected customization projects in the website for a single company at least once, you do not need to update website files. You can apply only the database changes.
 - c. To execute all database scripts of the selected customization projects, select the **Publish with Cleanup** check box (item e).
 -  : When the platform publishes a project that contains a database script, it executes the script and tries to avoid the execution of the script during every publication of the project, for optimization purposes. Therefore, the platform keeps information about each script that is executed at least once and has not yet been changed in the database, and omits the repeated execution of such scripts. If you run the publication with the **Publish with Cleanup** check box selected, the platform cleans up such information about previously executed scripts and executes them once more while publishing the project.
 - d. Click **OK** (item f).

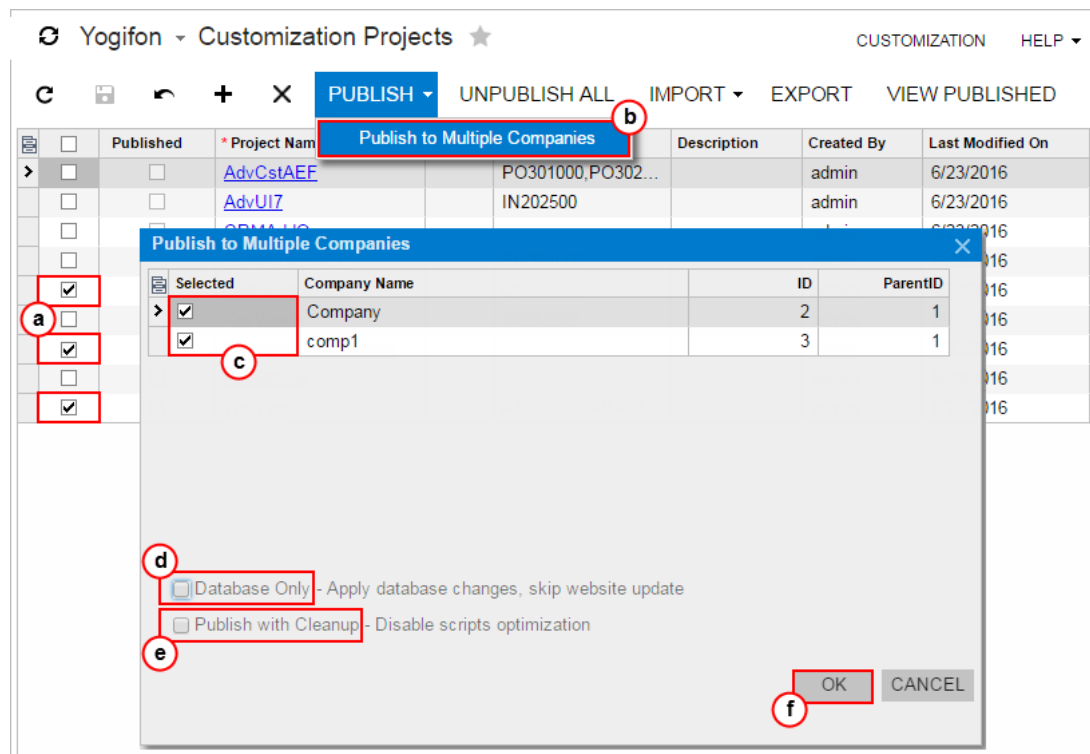


Figure: Publishing Customization for a Multi-Company Site

The platform applies the customization content to each company selected in the dialog box. As a result, the published customization content is saved in the database for each selected company.

To View a Published Customization

You can view the merged content of multiple customization projects that are currently published by using the *Published Customization* page of the *Customization Projects* (SM.20.45.05) form.

When you publish multiple projects at once, the platform merges the projects into a single project and then applies this project to the application instance. (See *To Publish Multiple Projects* for details.) To view the content of the merged project, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. On the form toolbar, click **View Published** on the toolbar, as the following screenshot shows.

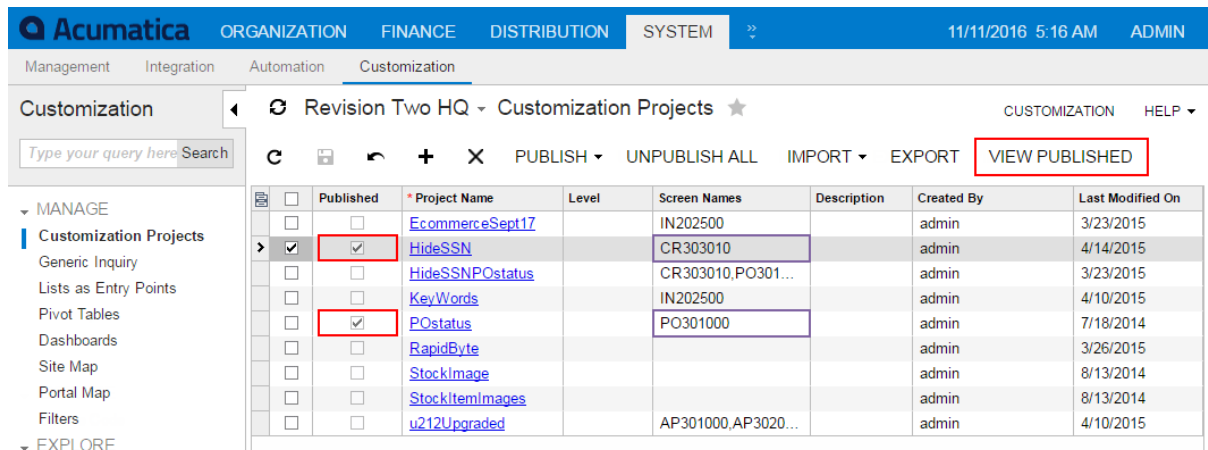


Figure: Opening the Published Customization page

The Published Customization page opens. The screenshot below shows two simultaneously published customization projects, *HideSSN* and *POstatus*. The Published Customization page shows the result of merging these projects.

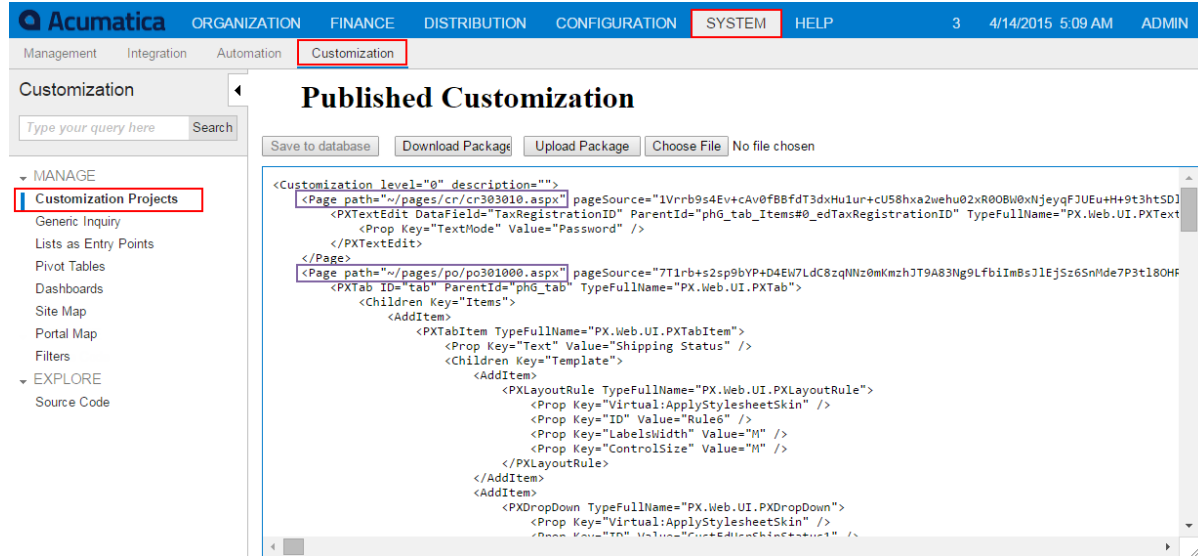


Figure: Viewing the merged XML code of published customization projects

To Unpublish a Customization

When there are multiple customization projects published in an instance of Acumatica ERP, you can use the following options to unpublish the projects:

- [Unpublish all projects](#)
- [Unpublish some projects](#)

In both cases, you use the [Customization Projects](#) (SM.20.45.05) form as a starting point.

To Unpublish All Projects

To remove all customization projects from publication, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. On the form toolbar, click **Unpublish All**.

The platform removes all published customization from the Acumatica ERP instance. See [Unpublishing Customization Projects](#) for details.

To Unpublish Some Projects

To remove some customization projects from publication, perform the following actions:

1. Navigate to **System > Customization > Manage > Customization Projects**.
2. In the project list of the form, be sure the check boxes for projects that you want to remain published are selected, and clear the check boxes for the unnecessary projects.
3. On the form toolbar, click **Publish**.

The platform removes all published customization from the Acumatica ERP instance, merges the content of the customization projects that are currently selected on the form, and applies the merged content to the instance.

Unpublishing Customization Projects

When you unpublish all customization projects, the system reverses the changes introduced by the customization as follows:

- The forms of Acumatica ERP return to their original layout.
- The `.cs` files of the project with customization code are removed from the website folder in the file system.
- The custom files of these projects are removed from the website folder on the file system.

Some changes aren't reversed, as described below:

- Database changes remain in the database after the customization is unpublished. Thus, the generic inquiries, reports, changes to the site map, custom tables, custom database objects, and custom data remain in the database. Changes to the site map remain in the navigation menu of Acumatica ERP. If you need to remove these changes, you must do so manually.
- The `.sln` file of the integrated Microsoft Visual Studio solution and its projects (if any) remain in the file system. However the customization code of the unpublished customization project and the external files added to the customization project are removed from the solution.

For example, if a customization project contains a *Report* item and a *SiteMapNode* item for the report, after you publish and unpublish the project, the report and the site map node remain in the database and remain available in the application, so you need to remove them manually.

There is no difference in the unpublishing process for a single-company site and a multi-company site: The platform deletes the same files in the file system and keeps all the changes in the database.

Managing Items in a Project

You use the *Customization Project Editor* to manage the items in a customization project. The editor includes a page to support for each type of item in a customization project. (See *Types of Items in a Customization Project* for details.) By using the navigation pane of the editor, as the following screenshot shows, you can open these pages (each of which is described in detail in the corresponding part of this guide).

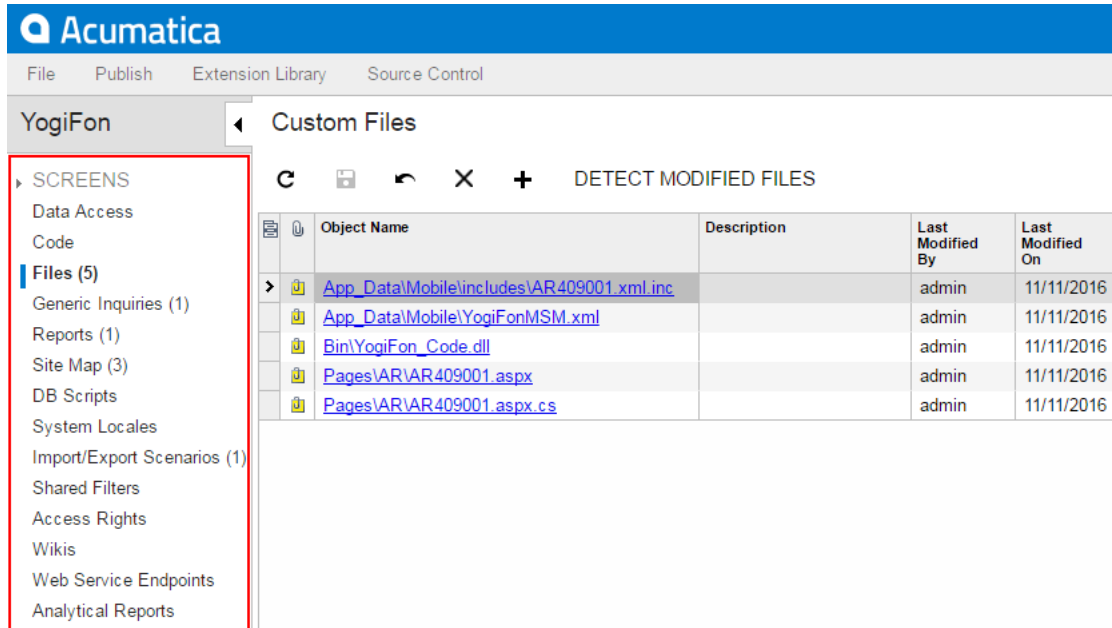


Figure: Viewing the navigation pane of the editor

In This Part

- [Customized Screens](#)
- [Customized Data Classes](#)
- [Code](#)
- [Custom Files](#)
- [Generic Inquiries](#)
- [Custom Reports](#)
- [Site Map](#)
- [Database Scripts](#)
- [System Locales](#)
- [Import and Export Scenarios](#)
- [Shared Filters](#)
- [Access Rights](#)
- [Wikis](#)
- [Web Service Endpoints](#)
- [Analytical Reports](#)

Customized Screens

You use the Customized Screens page of the *Customization Project Editor* to manage *Page* items in the customization project.



The *Page* item for an existing form contains layout change instructions that have to be applied by the platform to the ASPX code of the form during the project publication. For a custom form, the *Page* item keeps the content of the form and the path to the ASPX file of the form. (The path is required to detect changes of the file in the file system in the development environment and to update the file while you publish the project.)

When you start to modify the layout of an existing form, to store the changes, the platform adds a *Page* item for the form to the customization project. (See for details.) When you use the *Layout Editor* to change the form layout, the platform saves each change to this item.

To create a custom form, you invoke **Add Screen > Create New Screen** on the Customized Screens page to create the workable template for the form, and add the template to the customization project as a *Page*, *Code*, *SiteMapNode*, and two *File* items, as described in *To Add a New Custom Form to a Project*. You can then develop the custom form by using both the Layout Editor and Microsoft Visual Studio.

The Customized Screen page contains the list of the *Page* items for existing and custom forms added to the customization project. The following screenshot shows the *Page* items added to the *YogiFon* customization project. In the screenshot, the check mark in the **Is New** column of the table indicates that the form with the name Subscription Usage Details (AR.40.90.00) is a custom one.

Screen ID	Title	Is New	Last Modified By	Last Modified On
AR.30.30.00	Customers	<input type="checkbox"/>	admin	6/23/2016
AR.40.90.00	Subscription Usage Details	<input checked="" type="checkbox"/>	admin	6/23/2016
SO.30.10.00	Sales Orders	<input type="checkbox"/>	admin	6/23/2016
SO.30.30.00	Invoices	<input type="checkbox"/>	admin	6/23/2016

Figure: Viewing the customized and custom forms of Acumatica ERP in the project

On the page, you can perform the operations with the customized screens that are described in the following topics:

- [To Add a Page Item for an Existing Form](#)
- [To Delete a Page Item from a Project](#)
- [To Add a New Custom Form to a Project](#)
- [To Delete a Custom Form from a Project](#)
- [To Delete Items from the Project on the Edit Project Items Page](#)

To Add a Page Item for an Existing Form

You can add a *Page* item for an existing form to a customization project by using both the *Customization Menu* and the *Element Inspector*, or you can add the item in the *Customization Project Editor*.

The following sections provide detailed information:

- [To Add a Page Item by Using the Element Inspector](#)
- [To Add a Page Item on the Customized Screens Page](#)

To Add a Page Item by Using the Element Inspector

To add a *Page* item for an existing form to a customization project by using the [Element Inspector](#), perform the following actions:

1. Open the form in the browser.
2. On the form title bar, click **Customization > Inspect Element** to launch the Element Inspector.
3. On the form, select the UI element (or area) to be customized, to open the [Element Properties Dialog Box](#) for the element (or area).
4. In the dialog box, click **Customize**.
5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one.

Acumatica Customization Platform creates the *Page* item for the form, adds the item to the currently selected customization project, and opens the form in the [Layout Editor](#).

The platform assigns to the new item a name that corresponds to the form ID.

To Add a Page Item on the Customized Screens Page

To add a *Page* item for an existing form to a customization project by using the Customization Project Editor, perform the following actions:

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Click **Screens** in the navigation pane to open the Customized Screens page.
3. On the page toolbar, click **Add Screen > Customize Existing Screen**.
4. In the **Customize Existing Screen** dialog box, which opens, double-click the needed form in the tree, which corresponds to the site map of the application.

The following screenshot demonstrates how you can select the Payments and Applications form (**Finance > Accounts Receivable > Work Area > Enter**) in the **Customize Existing Screen** dialog box.

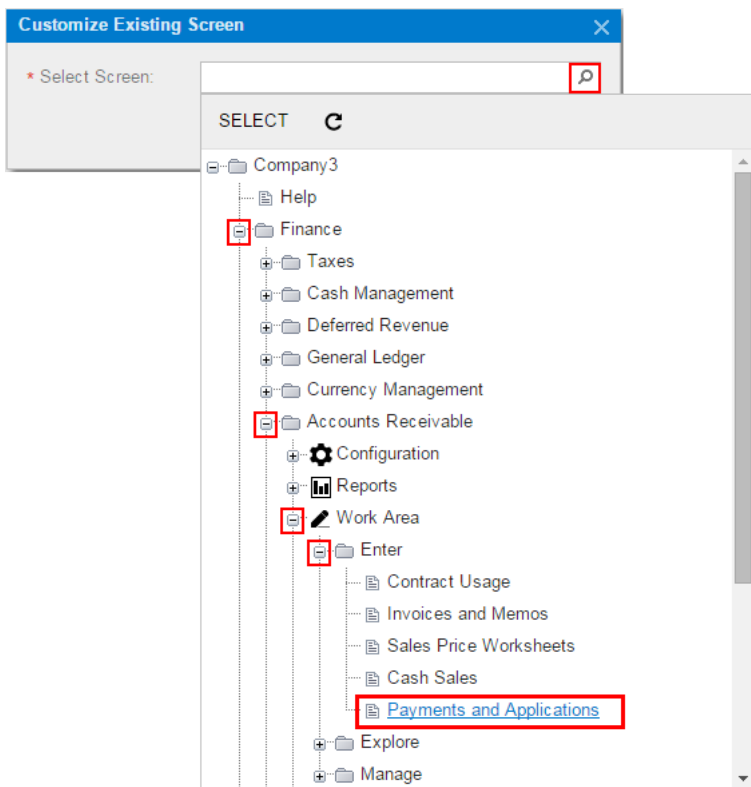


Figure: Selecting the form in the Customize Existing Screen dialog box

As soon as you add the item, the *Layout Editor* opens for the form so that you can start changing the form layout.

To go back to the Customized Screens page of the Customization Project Editor, click **Screens** on the navigation pane. You can see that the added form is saved to the list of project items.

As an alternative to selecting a form in the tree, if you know the screen ID of the form, you can add the appropriate item directly to the table of the Customized Screens page. To do this, perform the following actions (shown in the screenshot below):

1. On the page toolbar, click **Add Row (+)**.
2. In the **Screen ID** column of the new row, type the screen ID of the form.
3. On the page toolbar, click **Save** to save the item to the project.

The screenshot below shows the screen ID of the Journal Transactions form entered in the table: GL.30.10.00. As soon as you specify the screen ID, press Tab on the keyboard to view the name of the form, which appears in the **Title** column; make you sure you are adding the item for the needed form.

Customized Screens					
Screen ID	Title	Is New	Last Modified By	Last Modified On	
* GL.30.10.00		<input type="checkbox"/>			
AR.30.30.00	Customers	<input type="checkbox"/>	admin	7/27/2016	
SO.30.10.00	Sales Orders	<input type="checkbox"/>	admin	7/27/2016	
SO.30.30.00	Invoices	<input type="checkbox"/>	admin	7/27/2016	

Figure: Adding the screen ID of an existing form to the table

To modify the layout of a form, open the Layout Editor for the form by clicking the **Screen ID** of the form in the table or in the navigation pane of the Project Editor.

To Delete a Page Item from a Project

To remove from a customization project a *Page* Item created for an existing or custom form, perform the following actions:

1. Open the project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Screens** in the navigation pane to open the Customized Screens page.
3. In the page table, click the item for the form.
4. On the page toolbar, click **Delete Row** (X).
5. On the page toolbar, click **Save** to save the changes to the customization project.

To Add a New Custom Form to a Project

To add a new custom form to a customization project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Screens** in the navigation pane to open the Customized Screens page.
3. On the page toolbar, click **Add Screen > Create New Screen**.
4. In the **Create New Screen** dialog box, which opens, specify the following parameters to create the files for the new form:
 - **Screen ID:** The unique identifier of the new form
 - **Graph Name:** The name of the new class of the business logic code for the form (also called *graph*)
 - **Graph Namespace:** The namespace to which the new business logic class should be added
 - **Page Title:** The title of the new form
 - **Template:** The form template that provides the default set of containers on the form
 - **Site Map Parent:** The parent node in the site map to which the new form should be added
5. In the dialog box, click **OK**.

For the new form, the system creates the following items and adds them to the customization project:

- The `.aspx` and `.aspx.cs` files, which appear in the **Files** list of the project items.
- The `.cs` file with the business logic code for the form, which appears in the **Code** list of project items.
- The site map node, which appears in the **Site Map** list of project items.
- The *Page* item, with a name that corresponds the new screen ID; this item appears in the **Screens** list of project items. This item contains the link to the new page content, which you can later develop by using the [Layout Editor](#).

To obtain the actual files in the file system, publish the customization project after you have added the form to the project. After that, the following files are available on the file system:

- The `.aspx` and `.aspx.cs` files in the `Pages/<First segment of ScreenID>/` folder of the website
- The `.cs` file with the business logic code in the `App_RuntimeCode` folder of the website

You can develop business logic code for the custom form in Microsoft Visual Studio later.

The screenshot below shows the files of the custom form that has been added to the customization project (through the **Add Screen > Create New Screen** command) with the following parameters:

- **Screen ID:** *KW.30.20.10*
- **Graph Name:** *KeywordsMaint*

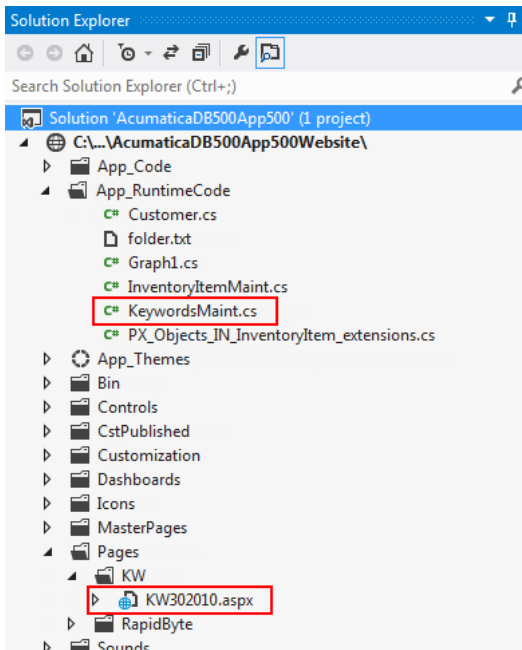


Figure: Viewing the files of the custom form in Visual Studio solution

To Delete a Custom Form from a Project

To remove a custom form from a customization project, you have to delete all the items that have been added to the project for the form.

To do this, perform the following actions:

- Delete from the customization project the *Page* item that was added by the New Screen wizard. (See [To Delete a Page Item from a Project](#) for details.)
- Delete from the project the *Code* item that was added by the New Screen wizard. (See [To Delete a Code Item From a Project](#).)
- Delete from the project the `<FormID>.aspx` and `<FormID>.aspx.cs` *File* items that were added by the New Screen wizard. (For more information, see [To Delete a Custom File From a Project](#).)
- Delete from the project the *SiteMapNode* item that was added by the New Screen wizard. (See [To Delete a Site Map Item from a Project](#) for details.)
- If you added other items for the custom form, such as items for the mobile site map or custom files, delete these items.

To delete multiple items from the customization project successively on a single page, you can use the Edit Project Items page of the Customization Project Editor. (See [To Delete Items from the Project on the Edit Project Items Page](#) for details.)

The system applies the changes to the file system as soon as you publish the customization project.

To Delete Items from the Project on the Edit Project Items Page

You can delete multiple items from the customization project on the Edit Project Items page of the Customization Project Editor. To do this, perform the following actions:

1. On the menu of the editor, click **File > Edit Project Items**.
2. In the table of the Edit Project Items page, which opens, click the item to be deleted, as the following screenshot shows.

The screenshot shows the Acumatica interface. The 'File' menu is open, and 'Edit Project Items' is selected. The main area displays a table of project items. The 'Contact' item is selected, and its XML source code is shown below.

Object Name	Type	Descr	Exc	Createc By	Creat Date	Last Modifie By	Last Modi On
~/pages/so/so30...	Page		<input type="checkbox"/>	admin	10/27/20	admin	11/11/20
~/pages/so/so30...	Page		<input type="checkbox"/>	admin	10/27/20	admin	10/27/20
AR650660.RPX	Report		<input type="checkbox"/>	admin	10/27/20	admin	10/27/20
Subscription Billi...	SiteMapNode		<input type="checkbox"/>	admin	10/27/20	admin	10/27/20
Subscription Us...	SiteMapNode		<input type="checkbox"/>	admin	11/11/20	admin	11/11/20
Active Contracts	SiteMapNode		<input type="checkbox"/>	admin	10/27/20	admin	10/27/20
ARTran	Table		<input type="checkbox"/>	admin	10/27/20	admin	10/27/20
Contact	Table		<input type="checkbox"/>	admin	10/27/20	admin	10/27/20
Contract Usage	XportScenario		<input type="checkbox"/>	admin	10/27/20	admin	10/27/20

```

Source
<Table TableName="Contact">
  <Column TableName="Contact" ColumnName="UsrPersonalIDType" ColumnType="st
  <Column TableName="Contact" ColumnName="UsrPersonalID" ColumnType="string
  <Column TableName="Contact" ColumnName="UsrCreditRecordVerified" ColumnTy
</Table>

```

Figure: Using the Edit Project Items page to delete an item from the customization project

3. Press Delete on the keyboard to delete the selected row from the table.
4. If you need to delete multiple items from the project, repeat Steps 2–3 for each item.
5. On the page toolbar, click **Save** to save the change in the project.

Customized Data Classes

You use the Customized Data Classes page of the *Customization Project Editor* to manage *DAC* items in a customization project.



: A *DAC* item contains data in the XML format used by the platform to create an appropriate extension for the original data access class.

The Customized Data Classes page displays the list of *DAC* items for existing data access classes of Acumatica ERP added to the project. The following screenshot shows the *IN.InventoryItem* data access class that was added to the *KeyWords* customization project.

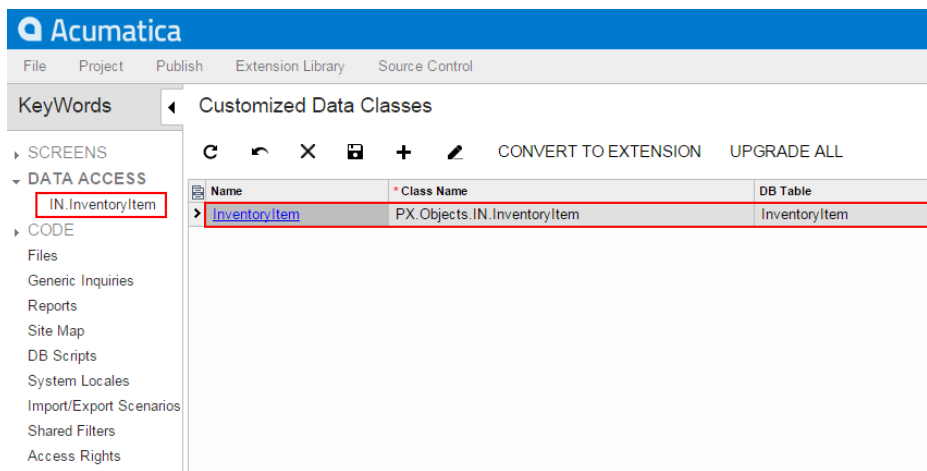


Figure: Viewing a customized data access class in the project

On the page, you can perform several operations with the customized data access classes, as described in the following topics:

- [To Add a DAC Item for an Existing Data Access Class to a Project](#)
- [To Delete a DAC Item from a Project](#)
- [To Convert a DAC Item to a Code Item](#)
- [To Upgrade Technology for Legacy DAC Customization](#)



You cannot create a custom data access class on this page. Instead, you have to use the [Code](#) page of the editor. Custom classes are added to the project as [Code](#) items. See [To Create a Custom Data Access Class](#) for details.

You can use Microsoft Visual Studio to work with a *DAC* item of a customization project that is currently published. During the project publication, the platform creates the `PX_Objects_<DACItemName>_extensions.cs` file with the item content in the `App_RuntimeCode` folder of the website. If you make changes to the code in Visual Studio, you have to update the item in the customization project. See [Detecting the Project Items Modified in the File System](#) for details.

To Add a DAC Item for an Existing Data Access Class to a Project

Before modifying an existing data access class, you have to add a *DAC* item for the class to the customization project. This item is used to store the data of the class extension in XML format. After the item is created, you can modify the class members by using the [Data Class Editor](#). After the customization project is published, the `.cs` file for the item is created in the file system, and you can develop the C# code of the class extension in Microsoft Visual Studio. You can use the Data Class Editor as well as Visual Studio to add custom fields to existing data access classes.

You can add a *DAC* item for an existing data access class to a customization project by using the [Element Inspector](#), or you can create and add the item on the Customized Data Classes page of the Customization Project Editor.

The following sections provide detailed information:

- [To Add a DAC Item by Using the Element Inspector](#)
- [To Add a DAC Item on the Customized Data Classes page](#)

To Add a DAC Item by Using the Element Inspector

1. Open the form in the browser.
2. On the form title bar, click **Customization > Inspect Element** to launch the Element Inspector.
3. On the form, select a UI element for a field of the class to be customized to open the [Element Properties Dialog Box](#) for the element.

The dialog box displays the name of the data access class that contains the selected element in the **Data Class** box, as shown in the screenshot below.

4. In the dialog box, click **Actions > Customize Data Fields**.

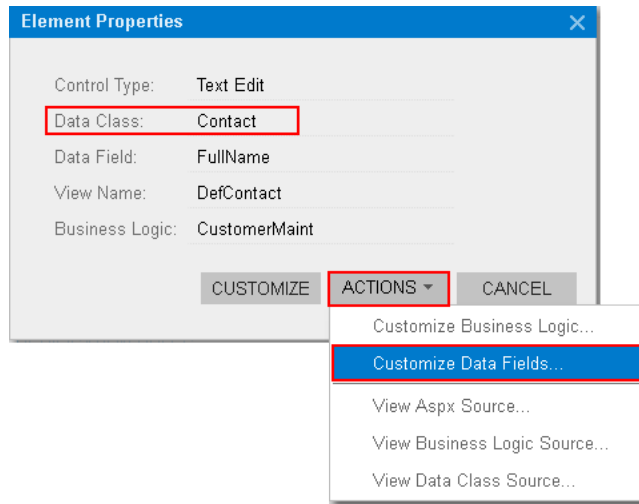


Figure: Using the Element Properties dialog box to start customization of the class

If there is no currently selected customization project, the inspector opens the [Select Customization Project Dialog Box](#) to force you to select an existing customization project or to create a new one.

Acumatica Customization Platform creates the *DAC* item for the class, adds the item to the currently selected customization project, and opens the class in the [Data Class Editor](#).

The platform assigns the new item the name of the data access class.

To Add a *DAC* Item on the Customized Data Classes page

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Click **Data Access** in the navigation pane to open the Customized Data Classes page.
3. On the page toolbar, click **Add New Record (+)**.
4. In the **Select Existing Data Access Class** dialog box, which opens, select the class in the **Class Name** box.

You can type the class name in the **Class Name** box or search for the class by a part of its name, as shown in the screenshot below. As soon as you add the class, the [Data Class Editor](#) opens for it so that you can modify the fields of this class and add custom fields to it.

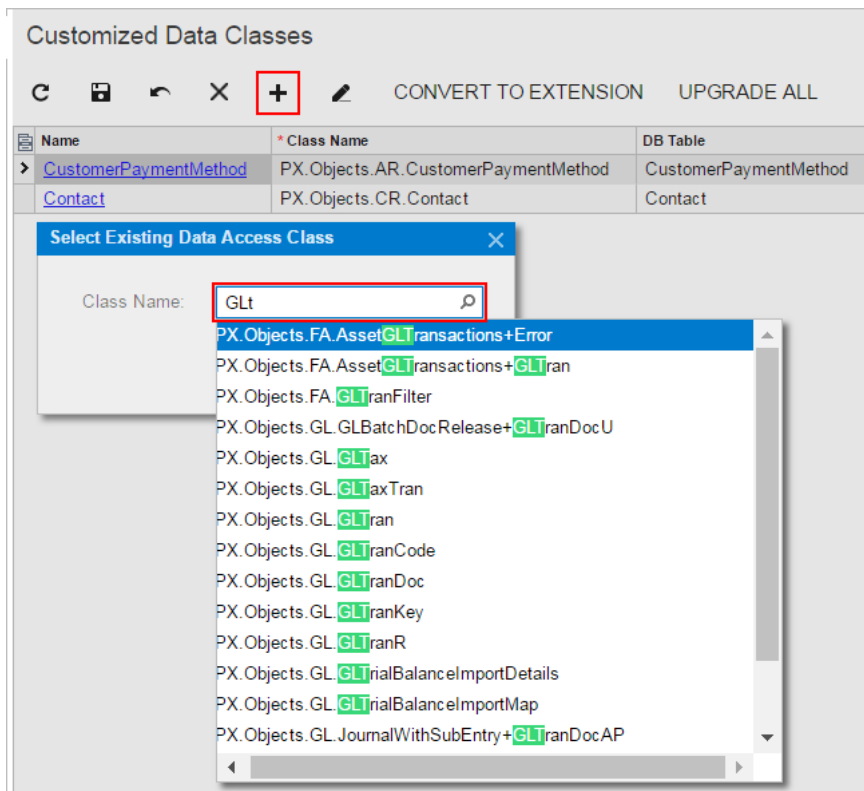


Figure: Adding an existing data access class to the project

5. On the page toolbar, click **Save** to save the item in the customization project.

As soon as you have modified the attributes of an existing field of the class or added a new field to the class and saved the changes in Data Class Editor, the class is added to the customization project and appears in the table of the Customized Data Classes page.

To go back to the Customized Data Classes page of the Customization Project Editor, select **Data Access** on the navigation pane. You can see that the added item is saved to the list of project items.

To Delete a DAC Item from a Project

To remove changes to an existing data access class from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Data Access** in the navigation pane to open the Customized Data Classes page.
3. In the page table, select the item to be deleted.
4. On the page toolbar, click **Delete Row**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you added a bound custom field to an existing data access class, the platform includes the *DAC* and *Table* items for this class in the customization project.



: The *Table* item contains a description of custom columns added to a database table for bound custom fields created in the appropriate data access class.

After you publish the customization project at least once, the database schema is changed. Changes to the database schema aren't deleted when you delete the *DAC* and *Table* items and publish the project. You have to remove the changes manually.

You can delete a *DAC* or *Table* item (or another item) from the customization project on the Edit Project Items page of the Customization Project Editor. (See [To Delete Items from the Project on the Edit Project Items Page](#) for details.)

To Convert a DAC Item to a Code Item

If you have a customized data access class that is added to the project as a *DAC* item, then you can convert the class changes into the class extension code (a *Code* item) to complete the extension development in the Code Editor or in Microsoft Visual Studio. (See [Supported DAC Extension Formats](#) for details.)

To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Data Access** in the navigation pane to open the Customized Data Classes page.
3. In the page table, select the item to be converted, as the screenshot below shows.
4. On the page toolbar, click **Convert to Extension**.



: This action can be applied for only data access classes customized using the technology based on extensions. If you have legacy DAC customization, upgrade it before converting *DAC* items to class extensions. See [To Upgrade Technology for Legacy DAC Customization](#) for details.

The screenshot shows the Acumatica interface with the 'Customized Data Classes' page open. The left navigation pane shows 'DATA ACCESS' selected, with 'CR.Contact' highlighted. The main table lists various data access classes. The 'CONVERT TO EXTENSION' button is highlighted in the toolbar.

Name	* Class Name	DB Table
APIInvoice	PX.Objects.AP.APIInvoice	APRegister
APPayment	PX.Objects.AP.APPayment	APRegister
APRegister	PX.Objects.AP.APRegister	APRegister
APTran	PX.Objects.AP.APTran	APTran
BalancedAPDocument	PX.Objects.AP.BalancedAPDocument	APRegister
APQuickCheck	PX.Objects.AP.Standalone.APQuickCheck	Projection
BAccount	PX.Objects.CR.BAccount	BAccount
Contact	PX.Objects.CR.Contact	Contact

Figure: Converting the *DAC* item to the *Code* item

The platform converts the XML content of the selected item to C# code, deletes the *DAC* item in the customization project, adds the created code to the project as a *Code* item, and opens it in the [Code Editor](#), as shown in the following screenshot.

The screenshot shows the Acumatica Code Editor interface. The left sidebar displays a tree view of the project structure, with 'ContactExtensions' selected under the 'CODE' folder. The main editor area shows the following C# code:

```

1 using PX.Data;
2 using PX.Data.EP;
3 using PX.Objects.AP;
4 using PX.Objects.CR.MassProcess;
5 using PX.Objects.CS;
6 using System;
7 using PX.TM;
8 using PX.SM;
9 using PX.Objects.EP;
10 using System.Collections.Generic;
11 using PX.Objects;
12 using PX.Objects.CR;
13
14 |
15 namespace PX.Objects.CR
16 {
17     [PXNonInstantiatedExtension]
18
19     public class CR_Contact_ExistingColumn: PXCacheExtension<PX.Objects.CR.Contact>
20     {
21
22         #region Fax
23
24         [PXDBString(50)]
25         [PXUIField(DisplayName = "Fax #")]
26         [PhoneValidation()]
27         public string Fax{get;set;}
28
29         #endregion
30
31     }
32 }
33 }
34
35

```

Figure: Viewing the result of the conversion

This operation is irreversible. After you convert the XML data to C# code, you will not be able to work with the item in the Data Class Editor or convert it back to a DAC item. You will be able to edit the code in Code Editor and Visual Studio.



Attention: The **Convert to Extension** action also affects all the inherited classes of the specified DAC if the classes are customized.

The system obtains the name of the *Code* item from the DAC name by appending the *Extensions* suffix to it. After the publication of the customization project, the actual customization code of the class is available in the <DACName>Extensions.cs file in the App_RuntimeCode folder of the website. For example, if you apply the action to the CR.Contact class, as shown in the screenshots above, the operation converts the DAC item to the Code item, automatically giving it the name of ContactExtensions. The action removes the CR.Contact DAC item class from the project and adds the ContactExtensions Code item.

To Upgrade Technology for Legacy DAC Customization

In Acumatica Customization Platform 4.2 and earlier versions, the customization of data access classes was implemented through the direct Microsoft Intermediate Language (MSIL) injection of custom fields into PX.Objects.dll. This resulted in a complex process of publication in which the original library was replaced with the modified version. In Acumatica Customization Platform 6.1, the customization uses the technology based on class extensions and the DAC extension is compiled into a separate library that is dynamically merged with the original DAC by the platform at run time.

If you do not need to change the contents of a customization project, there is no reason to upgrade it. The system will successfully publish the project using the injection of custom fields into PX.Objects.dll in the newer version of the system as well. However if you have to continue the development of the project, we recommend that you upgrade the technology of DAC customization in the project.

The following sections provide detailed information:

- [To Upgrade a Legacy DAC Customization](#)
- [To Upgrade a Library with a Legacy DAC Customization](#)

To Upgrade a Legacy DAC Customization

To upgrade the legacy DAC customization to the technology based on class extensions, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Data Access** in the navigation pane to open the Customized Data Classes page.
3. On the page toolbar, click **Upgrade All**, as shown in the following screenshot.

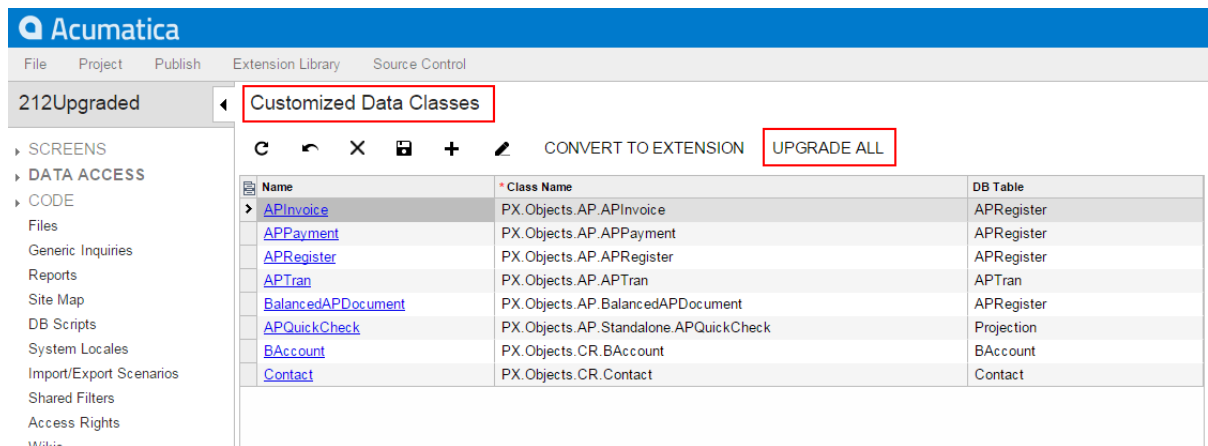


Figure: Upgrading legacy customization

This action launches the upgrade wizard, which processes all the project items of the *DAC* and *Code* types to discover and update the following cases of DAC customization based on the MSIL injection technology:

- A field of a data access class does not contain the `StorageName` attribute, which specifies the storage type of the field. For more information about the storage types, see [Create New Field Dialog Box](#).



: In the Acumatica Customization Platform, you must select the way a custom field will be stored in the database when you add the field to a data access class.

- The code contains a direct reference to a custom field of a row.
- The code contains a direct reference to an abstract class of a DAC field.

See [Using the Upgrade Wizard](#) for details.

After the process is complete, the wizard opens the message box with the list of the items that have been upgraded (see the screenshot below).

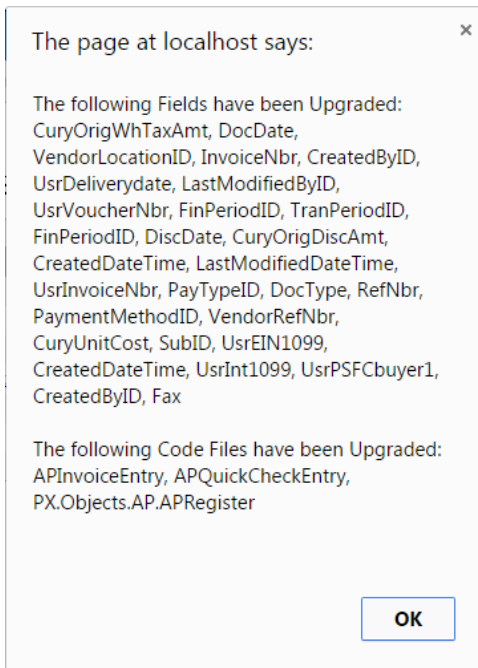


Figure: Viewing the list of upgraded items

If there is no legacy customization in the project, the wizard opens a message box with the relevant information (see the screenshot below).

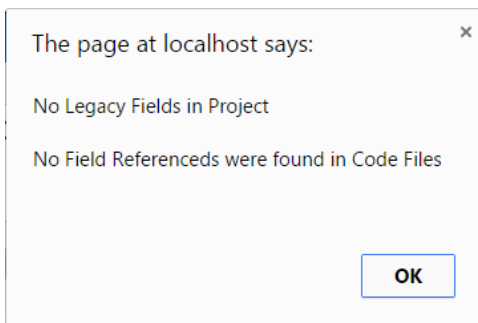


Figure: Viewing the message box that indicates no legacy customization in the project

To Upgrade a Library with a Legacy DAC Customization

If you have legacy customization of data access classes as a library (.dll) and have to modify the customization project, you can add the source code of the library as *Code* items to the customization project and then upgrade it as follows:

1. In the Microsoft Visual Studio project of the library, for each source code file that contains a data access class customization, do the following:
 - a. In Visual Studio (or any text editor), open the file, select **All**, and copy the source code to the clipboard.
 - b. Create a new *Code* item in the customization project.
 - c. Delete the code template from the created item.
 - d. Paste the clipboard content into the item and save the *Code* item to the customization project.
2. Upgrade the customization project.

3. Test the upgraded customization project to ensure that the project is valid and applies to the system after publication.
4. If you need to move the source code back to the library, use the clipboard and the copy-paste approach as well.

Using the Upgrade Wizard

While upgrading the *DAC* items, the wizard does the following:

- Adds the `StorageName` attribute and sets its value to "ExistingColumn" for an existing field.
- Adds the `StorageName` attribute and sets its value to "AddColumn" for a custom field.
- Finds and replaces all references to custom fields in the attributes of other customized fields, such as a `PXSelector`, `PXParent` or `PXFormula`. For example, the wizard inserts the `typeof(DACNameExt.usrFieldName)` reference in the `PXSelector` declaration instead of the `typeof(DACName.usrFileName)` reference.

While upgrading items of the *Code* type, the wizard processes each item to find references to the custom fields and replaces the references by using the following approach:

- The wizard updates the references to the abstract class of the DAC field from `DACName.usrFieldName` to `DACNameExt.usrFieldName`, where the `DACNameExt` is the name of the new extension class.
- The wizard replaces the references to the field `Row.UsrFieldName` with a reference to the field through the DAC extension: `Row.GetExtension<DACNameExt>().UsrFieldName`.

For example, the following code contains the references to the `APRegister.usrVoucherNbr` class and `doc.UsrVoucherNbr` field, which is a field customized based on the MSIL injection technology.

```
sender.RaiseExceptionHandling<APRegister.usrVoucherNbr>(doc,
    doc.UsrVoucherNbr,
    new PXSetPropertyException("..."));
```

After technological upgrade for the customization of DAC classes in the project, the references will look as the following code shows. Now the code refers to the `usrVoucherNbr` class of the `APRegisterExt` extension class that will be generated during publication of the project.

```
sender.RaiseExceptionHandling<APRegisterExt.usrVoucherNbr>(doc,
    doc.GetExtension<APRegisterExt>().UsrVoucherNbr,
    new PXSetPropertyException("..."));
```



Warning: There might be a situation when a legacy customization includes two data access classes with custom fields that have the same names, such as `DACName1.usrTheSameFieldName` and `DACName2.usrTheSameFieldName`. When you upgrade the customization, the wizard replaces the `Row.usrTheSameFieldName` references to each field by using the following pattern: `Row.GetExtension<DACName1Ext or DACName2Ext>().usrTheSameFieldName`. The expression "GetExtension<DACName1Ext or DACName2Ext>" is invalid and causes a compilation error. You are supposed to review these references and insert the correct reference to the needed class, `DACName1Ext` or `DACName2Ext`.

Code

You use the Code page of the [Customization Project Editor](#) to manage *Code* items in the customization project. On the page, you can perform the following operations with items:

- Add a new *Code* item of any of the following subtypes to the project:
 - *New Graph*: A new business logic controller that is derived from the `PXGraph<>` class
 - *New DAC*: A data access class that is derived from the `IBqlTable` class
 - *Graph Extension*: A graph extension that is derived from the `PXGraphExtension<>` class

- *DAC Extension*: A DAC extension that is derived from the `PXCacheExtension<>` class
- *Code File*: Custom C# code
- *Customization Plug-in*: A class that is derived from the `CustomizationPlugin` class
- Delete a *Code* item from the project

The screenshot below shows the *Code* items that have been added to the *YogiFon* project.

Object Name	Description	Last Modified By	Last Modified On
ARReleaseProcess		admin	8/2/2016
CustomerMaint		admin	8/2/2016
SOInvoiceEntry		admin	8/2/2016
SOOrderEntry		admin	8/2/2016

Figure: Viewing the Code items included in the customization project

You can open the [Code Editor](#) for a *Code* item by clicking the object name of the item in the page table.



: You can develop the customization code in the Code Editor. However we recommend that you develop the code in Microsoft Visual Studio (as described in [Integrating the Project Editor with Microsoft Visual Studio](#)) and use the editor for either minor code correction or the insertion of ready portions of code.

To move the code from a *Code* item to the extension library that is bound to the customization project, use the **Move to Extension Lib** action on the Code Editor toolbar.

For detailed information on working with *Code* items, see the following topics:

- [To Create a Custom Business Logic Controller](#)
- [To Create a Custom Data Access Class](#)
- [To Customize an Existing Business Logic Controller](#)
- [To Customize an Existing Data Access Class](#)
- [To Add Custom Code to a Project](#)
- [To Add a Customization Plug-In to a Project](#)
- [To Delete a Code Item From a Project](#)
- [To Move a Code Item to the Extension Library](#)

You can use Visual Studio to work with a *Code* item of a customization project that is currently published. During the publication of the project, the platform creates the `<CodeItemName>.cs` file with the item content in the `App_RuntimeCode` folder of the website. If you make changes to the code in Visual Studio, you have to update the item in the customization project. See [Detecting the Project Items Modified in the File System](#) for details.

To Create a Custom Business Logic Controller

You can add a custom business logic controller to a customization project on the Code page of the Customization Project Editor.

To do this, perform the following actions:

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Select **Code** in the navigation pane to open the Code page.
3. Click **Add New Record** (+) on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *New Graph* in the **File Template** box, as the screenshot below shows.
5. In the **Class Name** box, specify the class name of the business logic controller to be created.
6. Click **OK**.

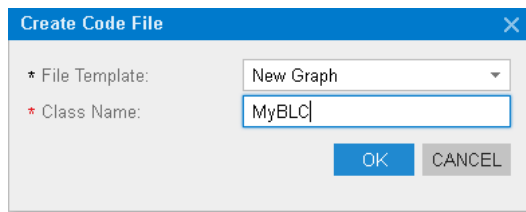


Figure: Adding a Code item for a custom graph to the project

The platform creates the code template of the class derived from the `PXGraph<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the [Code Editor](#).


To Create a Custom Data Access Class

You can add a new data access class (DAC) to a customization project by generating the code from the definition of a database table.

To create a custom data access class for a custom database table and add the created item to a customization project, you have to generate the class template on the Code page of the Customization Project Editor.

To do this, perform the following actions:

1. Create the needed custom table in the database by using a database management tool.
2. Generate the DAC code for the customization project as follows:
 - a. Open the customization project in the Project Editor.
 - b. Click **Code** in the navigation pane to open the Code page.
 - c. Click **Add New Record** (+) on the page toolbar.
 - d. In the **Create Code File** dialog box, which opens, select *New DAC* in the **File Template** box, as the screenshot below shows.
 - e. In the **Class Name** box, specify the class name that corresponds to the name of the table created in the database.

 : If you have just created the table, restart Internet Information Services (IIS) or recycle the application pool to make sure Acumatica ERP is aware of the new table, because it caches the database schema once, when the domain starts.
 - f. Select the **Generate Members from Database** check box.
 - g. Click **OK**.

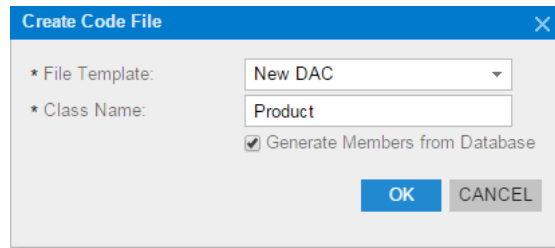


Figure: Adding a new data access class to the customization project

The platform does the following:

- Generates the data access class with members that correspond to the table columns. The class is added to the customization project namespace.
- Adds the class to the customization project as a *Code* item.
- Saves the customization project.
- Opens the created item in the [Code Editor](#).

You can use the Code Editor to modify the generated code. After you publish the customization project, you can work with the custom data access class in MS Visual Studio.

3. In the Code Editor, define the key fields in the DAC. To include a data field in the key, in the type attribute of the field, you have to add the `IsKey` parameter, as the example below shows.

```
[PXDBString(15, IsKey=true)]
```

4. Add the table definition to the customization project by doing the following:
 - a. In the navigation pane of the Project Editor, select **DB Scripts**.
 - b. On the Database Scripts page, which opens, click **Add New Record (+)** on the toolbar.
 - c. In the **Edit SQL Script** dialog box, which opens, select the table name in the **DBObject Name** selector.
 - d. Select the **Import Table Schema from Database** check box, which appears in the dialog box once the platform has found the specified table in the database (see the screenshot below).
 - e. Click **OK**.

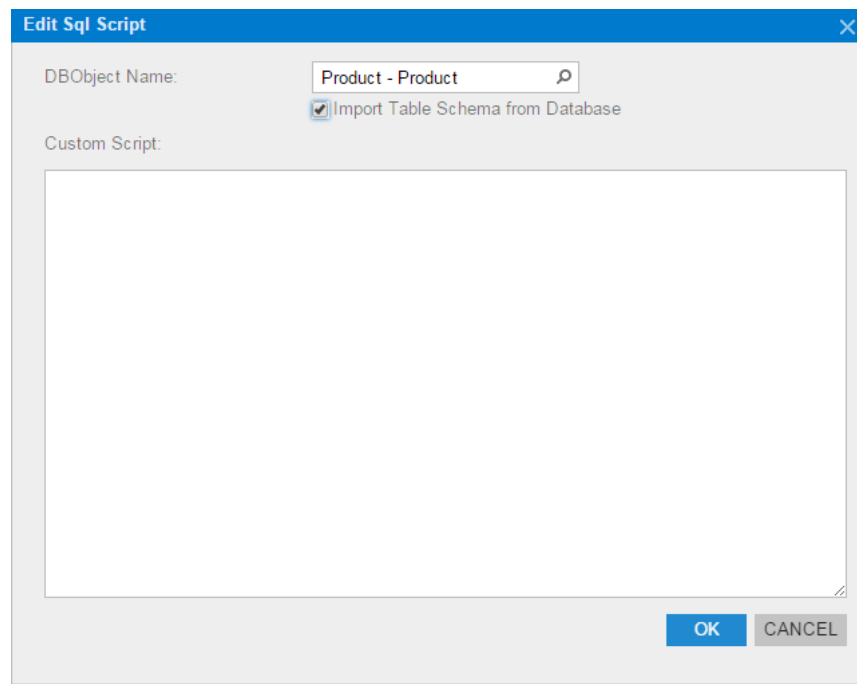


Figure: Adding an SQL script to the customization project

The platform does the following:

- Adds the XML definition of the table to the customization project as an *Sql* item
- Saves the customization project

Every time you publish the customization project, the system checks whether a table with this SQL definition exists in the database. If the table doesn't exist, the system creates the table. If the table exists, the system adjusts the table schema by using the definition, if there is any difference (no data is truncated).

To Customize an Existing Business Logic Controller

You can create the class extension for an existing business logic controller (BLC) and add the *Code* item with the created code to a customization project in several ways, as described in the following sections:

- [To Add a Code Item by Using the Element Inspector](#)
- [To Add a Code Item by Using the Layout Editor](#)
- [To Add a Code Item on the Code page](#)

If you need to extend the code of a BLC that has no webpage associated (such as `ARReleaseProcess`), follow the instructions described in [To Add a Code Item on the Code page](#).

As soon as you add the *Code* item for customization of the business logic to the project, the system generates an extension class for it and opens the code in the [Code Editor](#). You can work with the extension classes in the Code Editor. After you publish the customization project, you can develop the code in MS Visual Studio.

To Add a Code Item by Using the Element Inspector

Typically, you want to modify the business logic that is executed for a certain form of Acumatica ERP.

To add a *Code* item for customization of the business logic for an existing form to a customization project by using the [Element Inspector](#), perform the following actions:

1. Open the form in the browser.

2. On the form title bar, click **Customization > Inspect Element** to launch the Element Inspector.
3. On the form, select any UI element to open the *Element Properties Dialog Box* for the element.

The **Business Logic** box of the dialog box displays the name of the business logic controller that provides business logic for the form, as shown in the screenshot below.

4. In the dialog box, click **Actions > Customize Business Logic**.

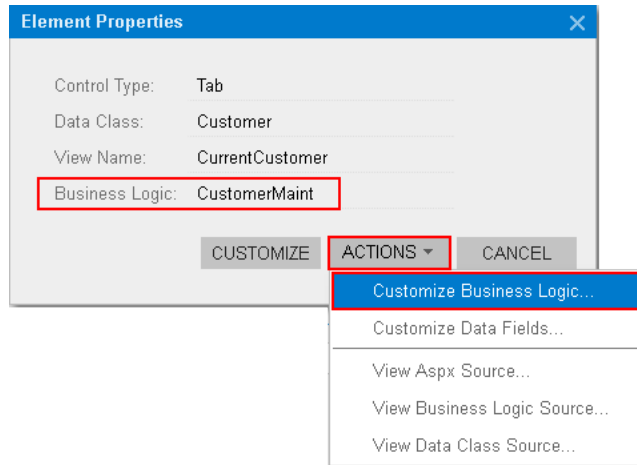


Figure: Using the Element Properties dialog box to customize the business logic for the form

5. If there is no currently selected customization project and the inspector opens the *Select Customization Project Dialog Box*, select an existing customization project or to create a new one.

The platform creates the template of the class that is derived from the `PXGraphExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the *Code Editor*, as shown in the following screenshot.

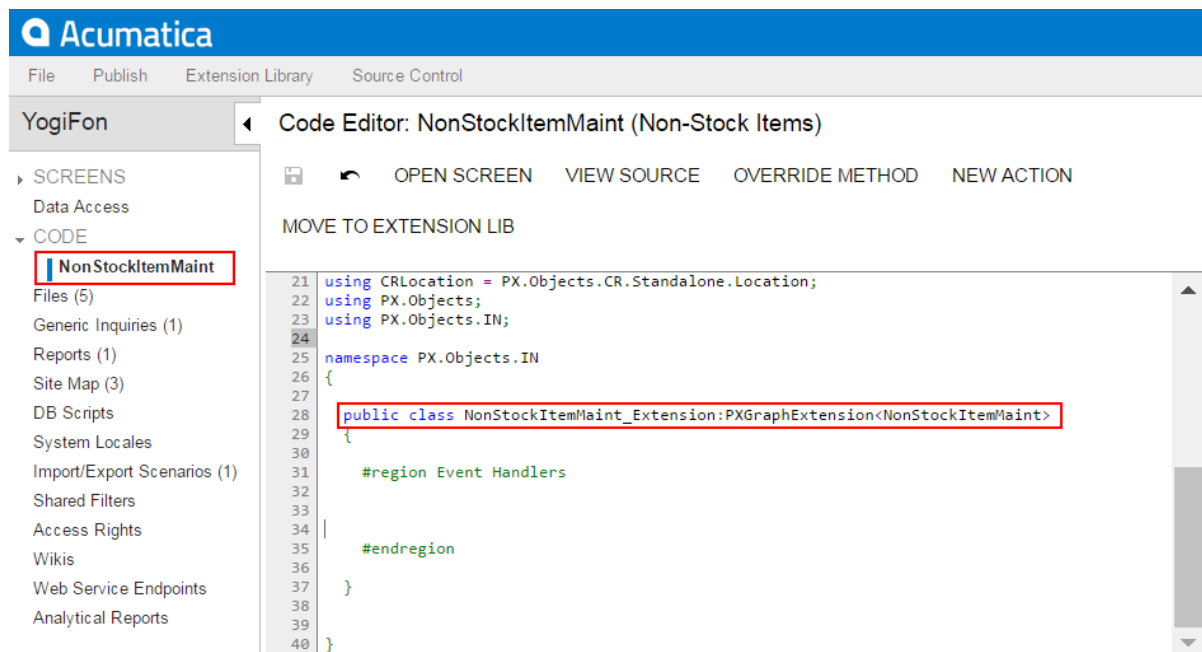


Figure: Viewing the created code template in the Code Editor

To Add a Code Item by Using the Layout Editor

Often, you start a customization of an Acumatica ERP form in the [Layout Editor](#) and you later want to modify the business logic for this form. To customize the business logic of the form, you can add a *Code* item to a customization project from the Layout Editor.

To do this, perform the following action:

1. On the toolbar of the Layout Editor, click **Actions > Customize Business Logic**, as the following screenshot shows.

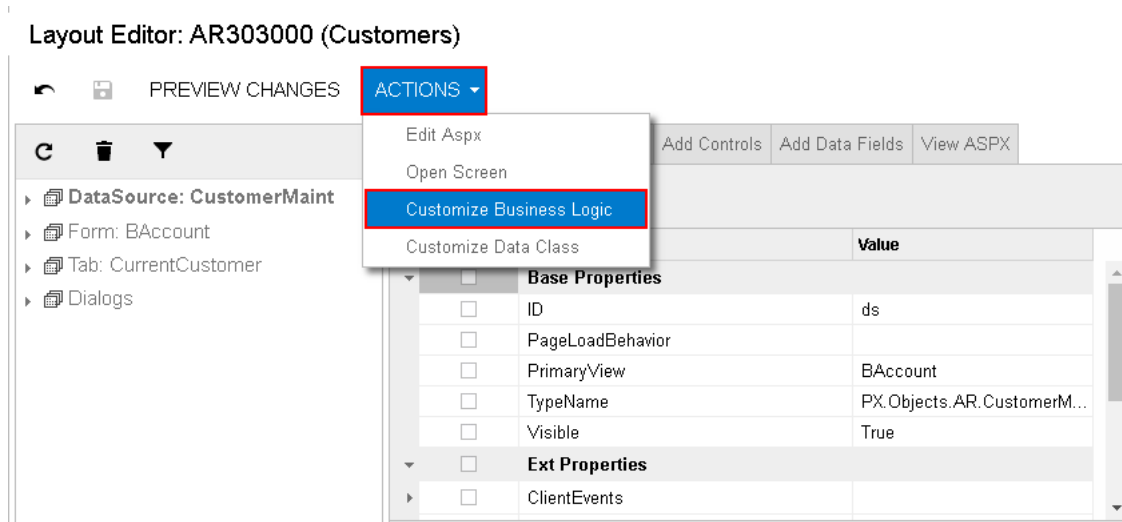


Figure: Starting the customization of the business logic from the Layout Editor

The platform creates the template of the class that is derived from the `PXGraphExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the [Code Editor](#).

To Add a Code Item on the Code page

If you know the name of the business logic controller to be customized, you can create a *Code* item with the graph extension template on the Code page of the Customization Project Editor by using the **Create Code File** dialog box.

To do this, perform the following actions:

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. Click **Add New Record (+)** on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *Graph Extension* in the **File Template** box, as the screenshot below shows.
5. In the **Base Graph** box, select the class name of the business logic controller to be customized.
6. Click **OK**.

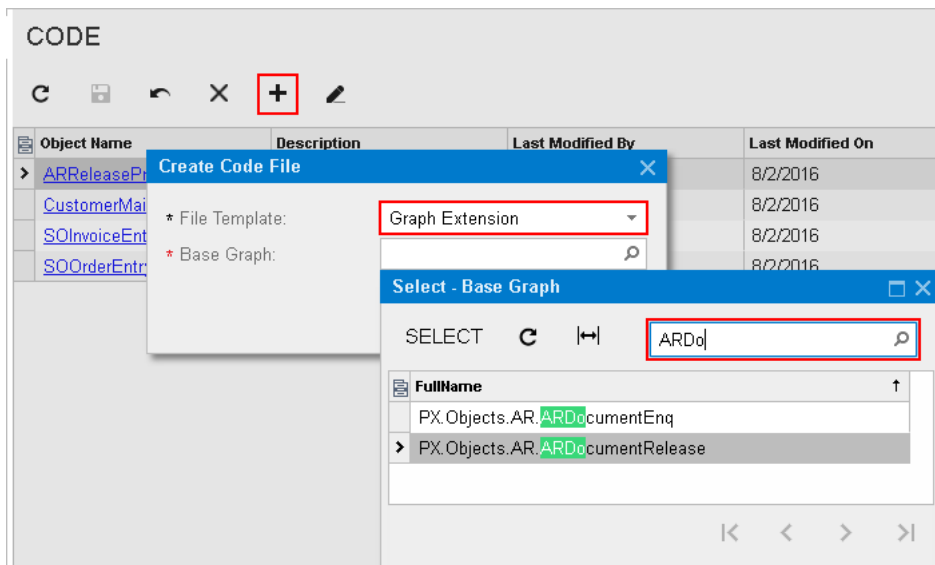


Figure: Adding a Code item with the graph extension to the project

The platform creates the template of the class that is derived from the `PXGraphExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the *Code Editor*.

To Customize an Existing Data Access Class

If you know the name of the data access class to be customized, you can create a *Code* item with the DAC extension template on the Code page of the Customization Project Editor by using the **Create Code File** dialog box.

To do this, perform the following actions:

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. Click **Add New Record** (+) on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *DAC Extension* in the **File Template** box, as the screenshot below shows.
5. In the **Base DAC** box, select the name of the data access class to be customized.
6. Click **OK**.

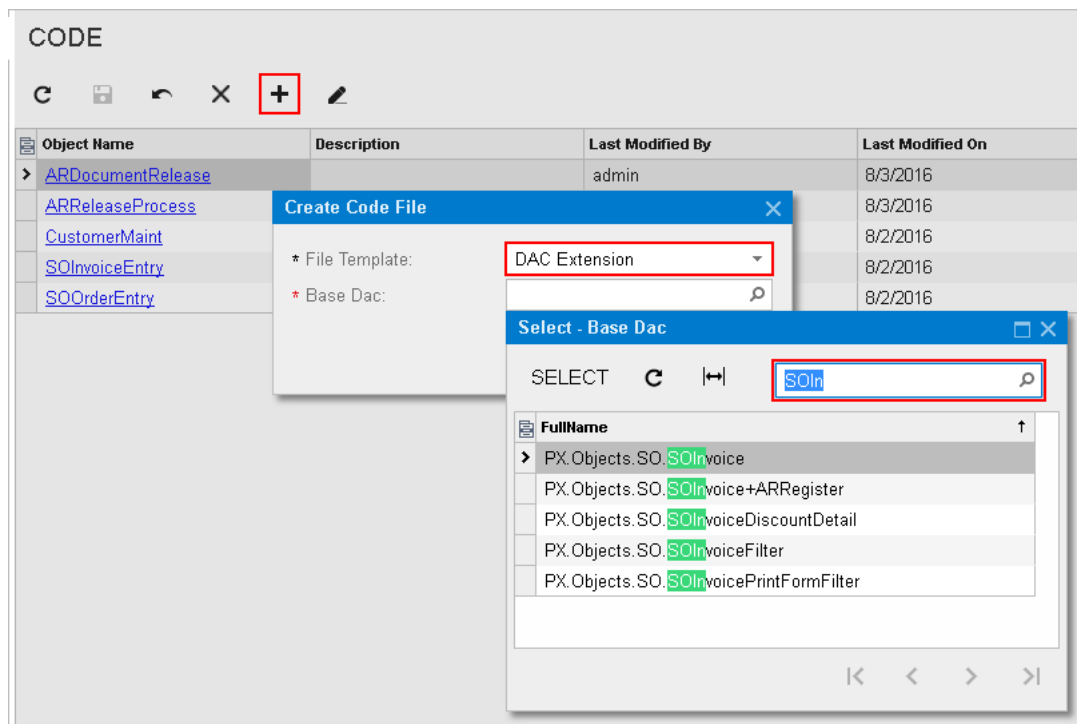


Figure: Adding a Code item with the DAC extension to the project

The platform creates the template of the class that is derived from the `PXCacheExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the *Code Editor*.

To Add Custom Code to a Project

You can add a `.cs` file with some custom code to a customization project. To do this, perform the following actions:

1. Open the customization project in the editor. (See *To Open a Project* for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. Click **Add New Record (+)** on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *Code File* in the **File Template** box, as the screenshot below shows.
5. In the **Class Name** box, specify the name of the new class to be added to the project.
6. Click **OK**.

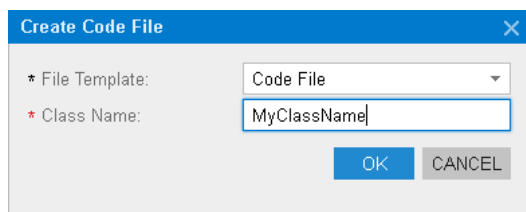


Figure: Adding a Code item for a custom graph to the project

The platform creates the code template of the new class, saves the code as a *Code* item of the project in the database, and opens the item in the *Code Editor*, as the following screenshot shows.

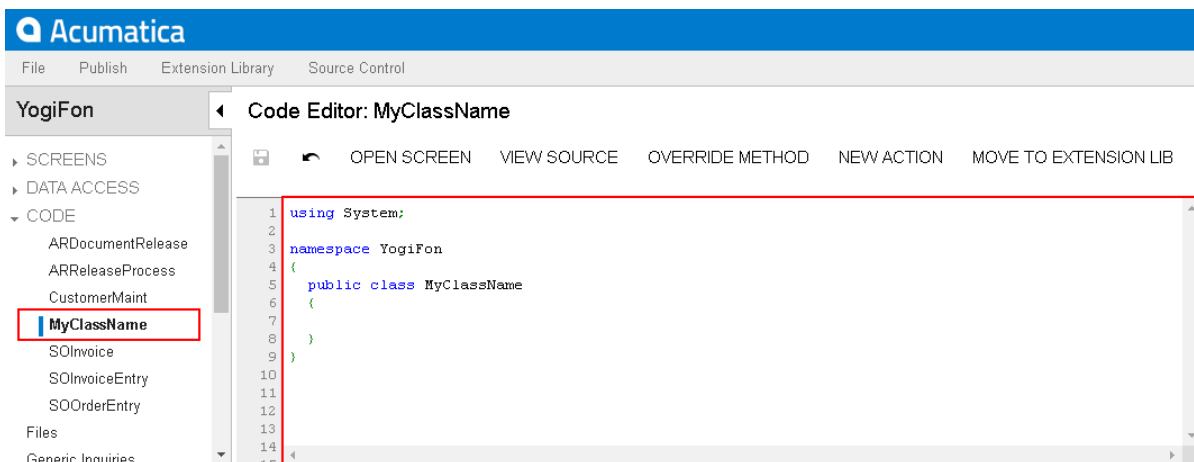


Figure: Viewing the custom code file added to the project

To Add a Customization Plug-In to a Project

As a part of a complex customization, you might need to make changes to the website beyond the customization project. For example, you might need to change the website configuration. In such situations, you can add a customization plug-in to the project with the code to be executed at the end of the publication process.

To add a customization plug-in, perform the following actions:

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. Click **Add New Record (+)** on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *Customization Plug-in* in the **File Template** box, as the screenshot below shows.
5. In the **Class Name** box, enter the name of the plug-in to be added to the project..
6. Click **OK**.

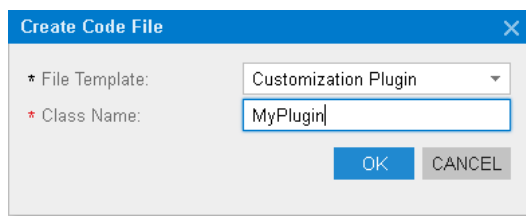


Figure: Adding a customization plug-in to the project

The system generates the plug-in code by template, as shown below.

```
using System;
using PX.Data;
using Customization;

namespace YogiFon
{
    //The customization plug-in is used to execute custom actions after the
    //customization project has been published
    public class MyPlugin: CustomizationPlugin
    {
        //This method is executed right after website files are updated, but before the
        //website is restarted
        //The method is invoked on each cluster node in a cluster environment
        //The method is invoked only if runtime compilation is enabled
        //Do not access custom code published to bin folder; it may not be loaded yet
    }
}
```

```

public override void OnPublished()
{
    this.WriteLog("OnPublished Event");
}

//This method is executed after the customization has been published and the
website is restarted.
public override void UpdateDatabase()
{
    this.WriteLog("UpdateDatabase Event");
}
}
}

```

When a customization project that contains a customization plug-in has been published, the corresponding `.cs` file is created in the `App_RuntimeCode` folder of the website.



: The Acumatica Customization Platform uses the `App_RuntimeCode` folder to keep the CS code of the DAC and *Code* items of all the published customization projects. By default, at run time, the platform compiles the code of this folder in a separate library and dynamically links the library to the Acumatica ERP application. (See [Run-Time Compilation](#) for details.) If you set the `UseRuntimeCompilation` key in the `<appSettings>` section of the `web.config` file (located in the website folder) to `False`, the platform uses the `App_Code/Caches` folder instead of the `App_RuntimeCode` one for the customization code. In this case, the `OnPublished` method of a customization plug-in cannot be executed. Execution of the `UpdateDatabase` method does not depend on the `UseRuntimeCompilation` key value.

To Delete a Code Item From a Project

To remove a *Code* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

You can delete a *Code* item from the customization project on the Edit Project Items page of the Customization Project Editor. (See [To Delete Items from the Project on the Edit Project Items Page](#) for details.)

If you are working in Microsoft Visual Studio, to update the files in the file system, publish the customization project. The `.cs` file of the deleted *Code* item will be removed from the file system.

To Move a Code Item to the Extension Library

You can develop customization code either as *Code* items in a customization project or as source code included in an extension library project in Microsoft Visual Studio. Some part of a customization may exist in the *Code* items of a customization project, while another part can be in an extension library that is included in the customization project as DLL file. (See [Extension Library](#) for details.)

To move the code from a *Code* item to an extension library, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. In the page table, click the name of the item to be moved to open the [Code Editor](#) for the item.
4. On the editor toolbar, click **Move to Extension Lib**.



: Before you launch the operation, be sure that the customization project is bound to an existing extension library. (See [Customization Project Editor](#) for details.)

For more information about the operation, see [Move to Extension Lib Action](#). Also, see [Extension Library \(DLL\) Versus Code in a Customization Project](#) for our recommendations about where you should keep your customization code.

Custom Files

You can add to a customization project any custom file located in the website folder of your instance of Acumatica ERP. When you add a file to a project, the Acumatica Customization Platform stores a copy of the file in the database as a *File* item. The platform then uses the copy of the file from the database for inclusion in the deployment package.



: A *File* item contains the path to a custom file and the GUID of the file content in the file storage of the database. The path is relative to the website folder.

You use the Custom Files page of the [Customization Project Editor](#) to manage *File* items in the customization project. The page displays the list of *File* items included in the project, as shown in the following screenshot.

Object Name	Description	Last Modified By	Last Modified On
App_Data\Mobile\includes\AR409001.xml.inc		admin	11/11/2016
App_Data\Mobile\YogiFonMSM.xml		admin	11/11/2016
Bin\YogiFon_Code.dll		admin	11/11/2016
Pages\AR\AR409001.aspx		admin	11/11/2016
Pages\AR\AR409001.aspx.cs		admin	11/11/2016

Figure: Viewing custom files in the project

When you run the project publication process or export the project, the platform compares each file in the project (in the database) with the original file in the file system and detects the files modified in the file system. If a modified file is found, there is a conflict, and the platform gives you the option to update the files in the project or discard the changes (and use the files from the database). See [Detecting the Project Items Modified in the File System](#) for details.

On the page, you can perform the operations with items described in the following topics:

- [To Add a Custom File to a Project](#)
- [To Update a File Item in a Project](#)
- [To Delete a Custom File From a Project](#)

To Add a Custom File to a Project

To add a custom file to a customization project, do the following:

1. In the file system, place the file in an appropriate folder within the website folder.
For example, if you need to add an extension library file, place it in the `Bin` folder of the website.

2. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
3. Click **Files** in the navigation pane to open the Custom Files page.
4. On the page toolbar, click **Add New record (+)**.
5. In the **Add Files** dialog box, which opens, find the file in the table and select the check box in the **Selected** column for it, as shown in the following screenshot.



: You can select multiple custom files to add them to the project at the same time.

The screenshot shows the Acumatica Custom Files page. The 'Add Files' dialog box is open, displaying a table of files. The file 'Bin\KeyWords.dll' is selected, and the 'SAVE' button is highlighted.

Selected	Path	Modified	Size
<input type="checkbox"/>	App_Data\CheckinFiles.xml	3/25/2015	1973
<input checked="" type="checkbox"/>	App_Data\maintlog.txt	3/26/2015	89332
<input type="checkbox"/>	App_Data\RollbackFiles\Bin\PX.Objects.dll	3/24/2015	16178688
<input type="checkbox"/>	App_Data\UpdateStatus.xml	3/25/2015	109
<input checked="" type="checkbox"/>	App_Data\Website\D190467961.txt	3/10/2015	36
<input type="checkbox"/>	App_RuntimeCode\POOrderEntry.cs	3/19/2015	890
<input type="checkbox"/>	App_RuntimeCode\PX_Objects_PO_POOrder.cs	3/23/2015	2853
<input checked="" type="checkbox"/>	Bin\KeyWords.dll	3/24/2015	5632
<input type="checkbox"/>	Bin\KeyWords.pdb	3/24/2015	11776
<input type="checkbox"/>	Bin\Microsoft.Practices.ServiceLocation.pdb	3/24/2015	24064
<input type="checkbox"/>	Bin\PX.Api.OData.pdb	3/24/2015	185856
<input type="checkbox"/>	Bin\PX.BulkInsert.pdb	3/24/2015	480768
<input type="checkbox"/>	Bin\PX.CCProcessing.pdb	3/24/2015	318976

Figure: Adding a custom file to the project



: For any files other than the ones placed in the `Bin` folder, you can click **Refresh** on the toolbar of the **Add Files** dialog box to make the system update the list of files in the table. If you have changed files in the `Bin` folder of the website, you should refresh the page in the browser by pressing F5 on the keyboard.

6. In the dialog box, click **Save** to save each selected file to the customization project as a *File* item.

If you modify the file added to a customization project in the file system, you have to update the appropriate *File* item in the project.

To Update a File Item in a Project

If you have modified a file of a customization project in the file system and need to use the modified version of the file in the project, you have to update the copy of the file in the database. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Files** in the navigation pane to open the Custom Files page.
3. On the page toolbar, click **Detect Modified Files**, as shown in the screenshot below.

- In the **Modified Files Detected** dialog box, which opens, ensure that the **Conflict** check box is selected for the file.

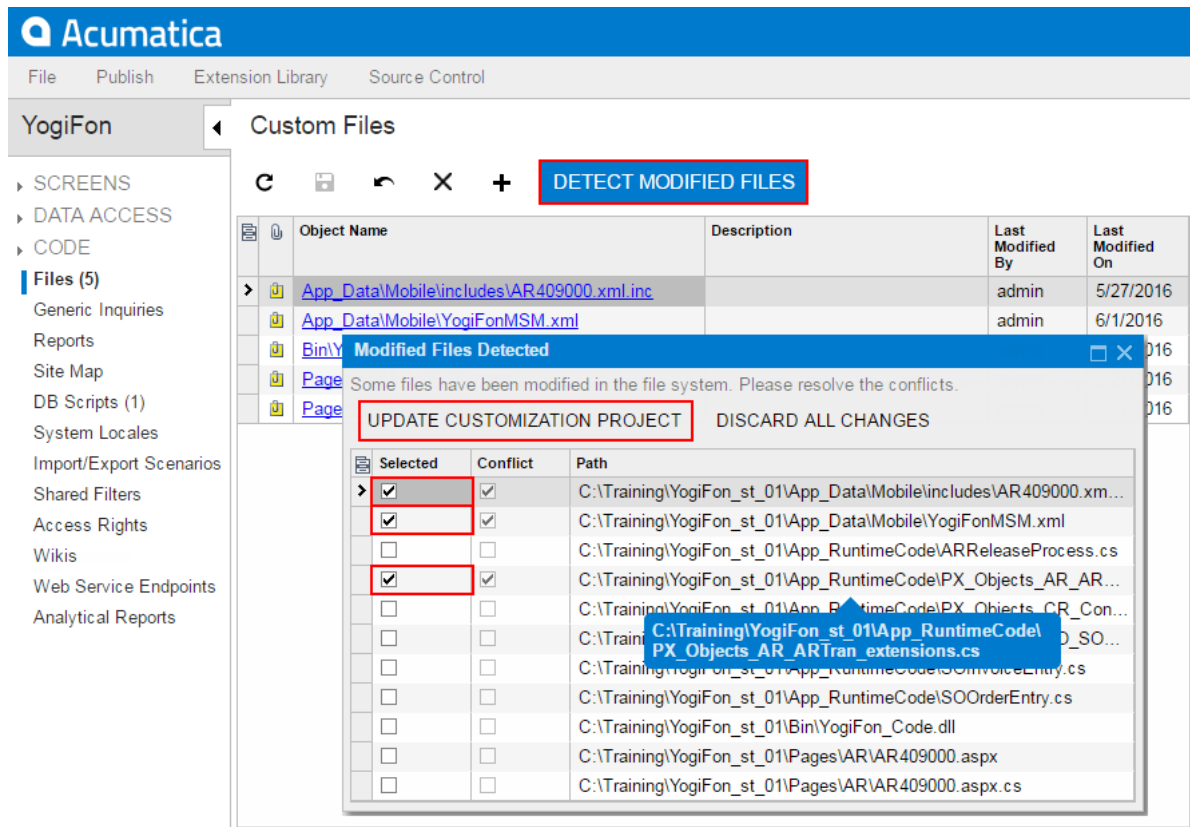


Figure: Updating files in the project

- If multiple files in the project were changed, and you do not want to update some files at the moment, clear the selection of these files in the **Selected** column.
- On the toolbar of the dialog box, click **Update Customization Project** to update the selected files.

If you click **Discard All Changes**, the Acumatica Customization Platform resolves the conflict by overriding the file in the file system using the file copy in the database.

If you make changes to custom files added to a customization project in the file system, the platform does not publish or export the project while a file in the file system differs its copy in the database. You have to resolve all such conflicts before publication or export of the project. See [Detecting the Project Items Modified in the File System](#) for details.

Detecting the Project Items Modified in the File System

In the website folder in the file system, the platform tracks changes that you make to the following files:

- Files with the customization code added to the customization project as `DAC` items
- Files with the customization code added to the project as `Code` items
- Custom files added to the project

If you make changes to these files in the file system, you have to update them in the project before you publish the project or export the deployment package of the project.

When you run the project publication process or export the project, the Acumatica Customization Platform compares each file in the project (in the database) with the original file and detects the files modified in the file system. If a modified file is found, there is a conflict, and the platform opens the **Modified Files Detected** dialog box (see the screenshot below) to give you the option to update the files in the project or discard the changes and use the files from the database.

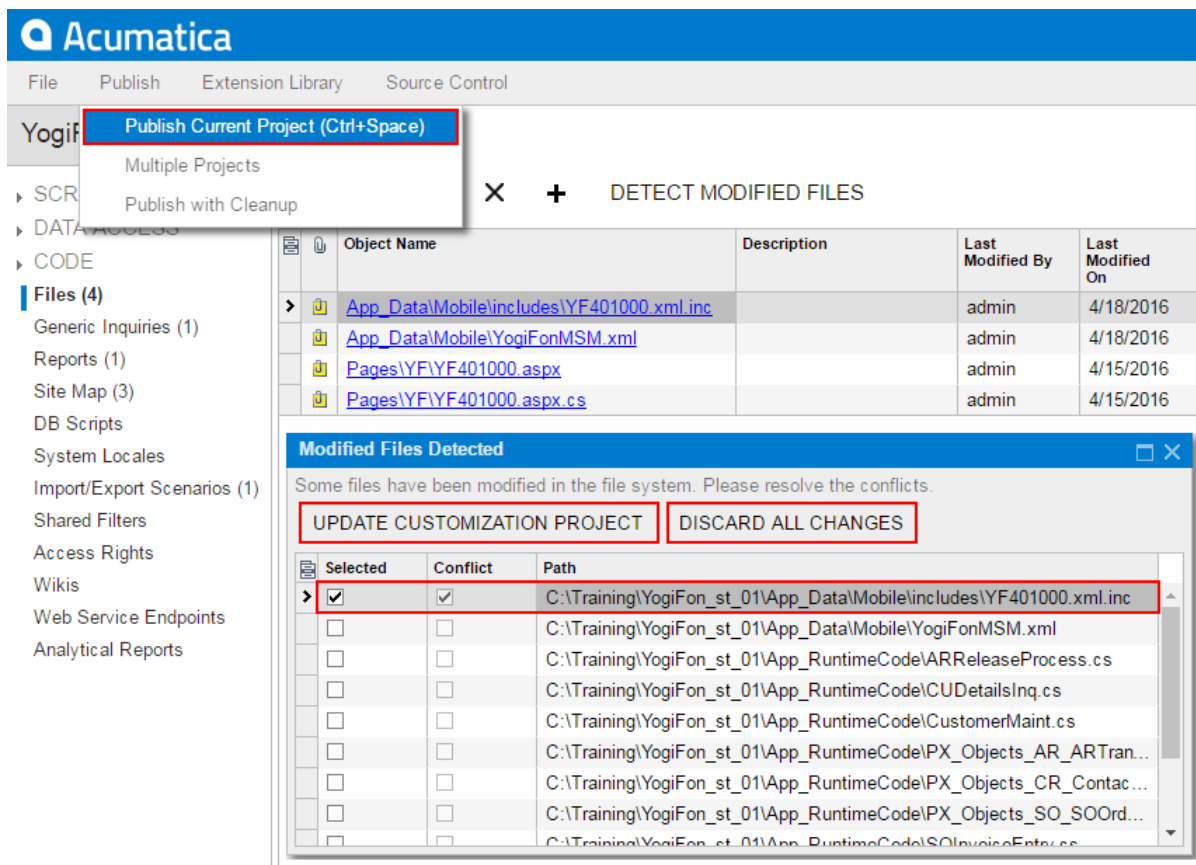


Figure: Viewing the Modified Files Detected dialog box



: If you have used the *File Editor* of the Customization Project Editor to modify a *File* item in a customization project and saved the changes in the database, the changes are not saved in the original file in the file system. Then if you click **Detect Modified Files** on the toolbar of the Files page, the platform does not detect a conflict because the file in the database is newer. The platform automatically updates the original file during the publication of the customization project.

The **Modified Files Detected** dialog box lists all custom and customized files in the website folder. The **Conflict** check box means that the file version in the file system differs from the file version in the customization project. This could happen, for example, if you have modified the customization code in a file by using MS Visual Studio and the change is not yet reflected in the customization project. You have to resolve all conflicts in the project before you publish the project or export the deployment package of the project.

In the dialog box, you can invoke the following actions for conflicting files:

- **Update Customization Project:** Updates the customization project with the file version from the file system.
- **Discard All Changes:** Keeps the file version that exists in the customization project and discard the changes in the file system.

These actions are performed on the selected files—that is, all conflicting files for which you have selected the check box in the **Selected** column.

You can invoke one action for one individual file and another action for another file. For example, you can first select the files that you want to update in the customization project and click **Update Customization Project**. Then you can click **Discard All Changes** to cancel the changes made to all other conflicting files.

Make sure you have updated all appropriate files before removing all remaining conflicts. If you discard changes, after you publish the customization project, the platform updates all selected conflicting files from the database, therefore the files will return to the original state in the file system.

No conflicts will appear in the **Modified Files Detected** dialog box until a file included in the customization project is modified in the file system again.

To Delete a Custom File From a Project

To delete a custom file from a customization project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Files** in the navigation pane to open the Custom Files page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

The *File* item is deleted from the project. The file remains in the file system and you can add it back to the project, if needed.



Warning: If you either publish the customization project after the *File* item is deleted or unpublish all customizations of the website, the platform deletes the original file in the file system.

Generic Inquiries

You use the Generic Inquiries page of the [Customization Project Editor](#) to manage *GenericInquiryScreen* items in the customization project.



: A *GenericInquiryScreen* item contains the data set of a custom or customized generic inquiry form.

The Generic Inquiries page displays the list of the generic inquiries added to the customization project. The following screenshot shows this page displaying one generic inquiry added to the customization project.

Object Name	Description	Last Modified By	Last Modified On
Active Subscribers		admin	6/23/2015

Figure: Viewing the generic inquiry in the project


On the page, you can perform the operations described in the following topics:

- [To Add a Generic Inquiry to a Project](#)
- [To Delete a Generic Inquiry from a Project](#)
- [To Update Generic Inquiry Items in a Project](#)
- [To Redirect to the Generic Inquiry Form](#)

To Add a Generic Inquiry to a Project

You can add to a customization project a custom or customized generic inquiry—the generic inquiry that is saved in the database for the current company. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Generic Inquiries** in the navigation pane to open the Generic Inquiries page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of generic inquiries in the **Add Generic Inquiries** dialog box, which opens, select the check box for each generic inquiry form that you want to include in the project.

 : The **Add Generic Inquiries** dialog box displays all the custom and customized generic inquiries that exist in your instance of Acumatica ERP. You can select multiple generic inquiries to add them to the project simultaneously.
5. In the dialog box, click **Save** to add the selected generic inquiry or inquiries to the customization project.

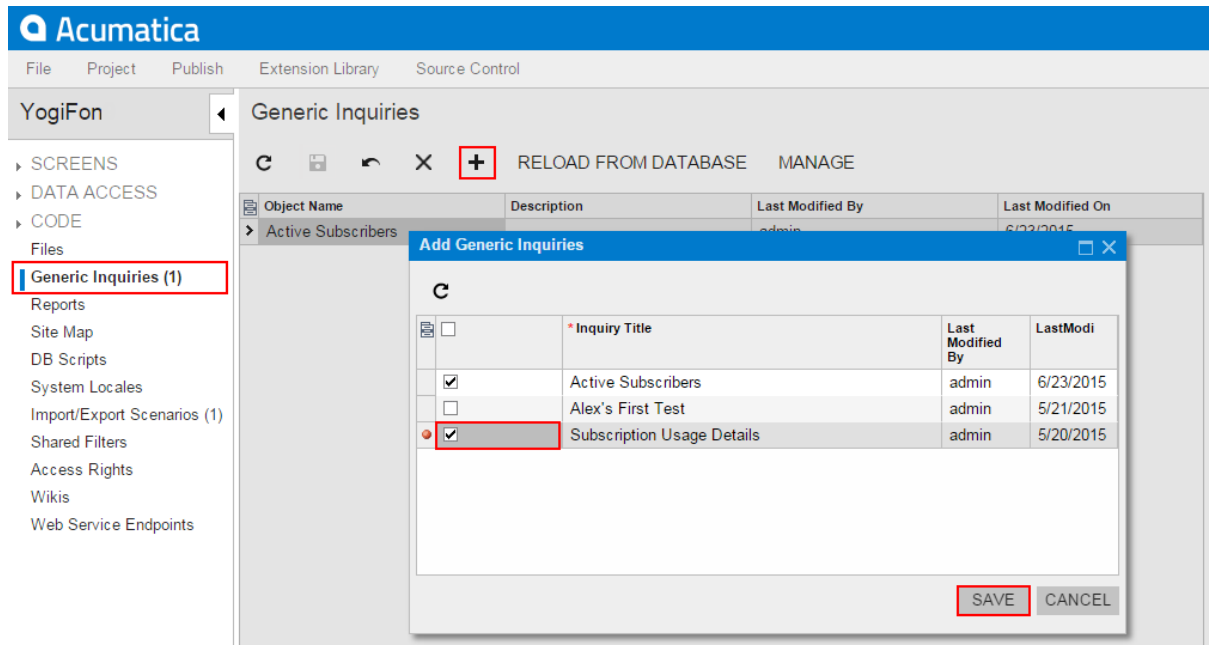


Figure: Adding the generic inquiry to the customization project

The system adds to the project the data for each selected generic inquiry, and you can see each new `GenericInquiryScreen` item in the Project Items table of the [Item XML Editor](#), as shown in the following screenshot.

Object Name	Type	Des	Excluded	Created By	Creation Date	Last Modified By	Last Modified On
PX.Objects.AR.ARTran	DAC		<input type="checkbox"/>	admin	5/20/2015	admin	5/20/2015
PX.Objects.CR.Contact	DAC		<input type="checkbox"/>	admin	5/20/2015	admin	5/20/2015
Active Subscribers	GenericInquiryScreen		<input type="checkbox"/>	admin	6/23/2015	admin	6/23/2015
Subscription Usage Details	GenericInquiryScreen		<input type="checkbox"/>	admin	6/23/2015	admin	6/23/2015
CustomerMaint	Code		<input type="checkbox"/>	admin	5/20/2015	admin	6/2/2015
SOInvoiceEntry	Code		<input type="checkbox"/>	admin	5/20/2015	admin	5/20/2015
~/pages/ar/ar303000.aspx	Page		<input type="checkbox"/>	admin	5/20/2015	admin	6/23/2015
~/pages/so/so303000.aspx	Page		<input type="checkbox"/>	admin	5/20/2015	admin	6/23/2015


```

Source
</GIRelation>
<GIResult>
  <row LineNbr="6" IsActive="1" IsKey="0" Field="tranDate" Caption="Da
  <row LineNbr="7" IsActive="1" IsKey="0" Field="billableQty" Caption=
  <row LineNbr="8" IsActive="1" IsKey="0" Field="uOM" Width="120" IsV
  </GIResult>
</row>
</GITable>
<GIWhere>
  <row LineNbr="1" IsActive="1" DataFieldName="Contract.CONTRTYPE_Attributes" Conditio
  <row LineNbr="2" IsActive="1" OpenBrackets="{ " DataFieldName="Contract.custd
  <row LineNbr="3" IsActive="1" DataFieldName="[Customer]" Condition="NU" IsExpressio
  <row LineNbr="4" IsActive="1" OpenBrackets="{ " DataFieldName="Contract.contr
  <row LineNbr="5" IsActive="1" DataFieldName="[Contract]" Condition="NU" IsExpressio
  <row LineNbr="6" IsActive="1" DataFieldName="PMTran.tranDate" Condition="B " IsExpre
  </GIWhere>
<SiteMap>
  <row Position="616" Title="Subscription Usage Details" Url="~/GenericInquiry/Generic
  </row>
</SiteMap>
</GIDesign>
</data>
</data-set>
</GenericInquiryScreen>

```

Figure: Viewing the new GenericInquiryScreen item included in the project



To give users the ability to navigate to the new inquiry form in Acumatica ERP, you have to add the appropriate site map node to the customization project. See [To Add a Site Map Node to a Project](#) for details.

To Delete a Generic Inquiry from a Project

To remove a *GenericInquiryScreen* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Generic Inquiries** in the navigation pane to open the Generic Inquiries page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *GenericInquiryScreen* item from the project, the generic inquiry remains in the system unless you delete the inquiry by using the [Generic Inquiry \(SM.20.80.00\)](#) form.

If you have added a site map node for a custom inquiry form to the project and removed the inquiry from the project, you should delete the appropriate *SiteMapNode* item. (See [To Delete a Site Map Item from a Project](#) for details.)

To Update Generic Inquiry Items in a Project

If you have used the [Generic Inquiry \(SM.20.80.00\)](#) form to change a generic inquiry included in the customization project, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Generic Inquiries** in the navigation pane to open the Generic Inquiries page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *GenericInquiryScreen* items of the project by using the appropriate data from the database.

To Redirect to the Generic Inquiry Form

When you are working with the *Customization Project Editor*, you may need to open the *Generic Inquiry* (SM.20.80.00) form in the browser. You can use this form, for example, to create a new generic inquiry, to customize an existing one, or to manage existing generic inquiries in Acumatica ERP.

To open the *Generic Inquiry* form from the Customization Project Editor, perform the following actions:

1. Select **Generic Inquiries** in the navigation pane to open the Generic Inquiries page.
2. On the page toolbar, click **Manage**.

As soon as you save a new inquiry on the *Generic Inquiry* form, you can add the inquiry as a *GenericInquiryScreen* item to the project.



: You can limit the list of tables available for constructing generic inquiries on the *Generic Inquiry* form. See [Limiting the List of Tables Available for Generic Inquiries](#) for details.

Limiting the List of Tables Available for Generic Inquiries

You can limit the list of database tables available for constructing generic inquiries on the *Generic Inquiry* (SM.20.80.00) form in the production environment. To do this, in the development environment, use the following approach:

1. Create the `GITables.xml` configuration file to specify the `Allowed` and `Hidden` collections of masks for the full names of the database tables, as described in [GITables.xml File Content](#).
2. Save the configuration file to the `App_Data` folder of the website.
3. Add the file to a customization project as a *File* item. (See [To Add a Custom File to a Project](#) for details.)

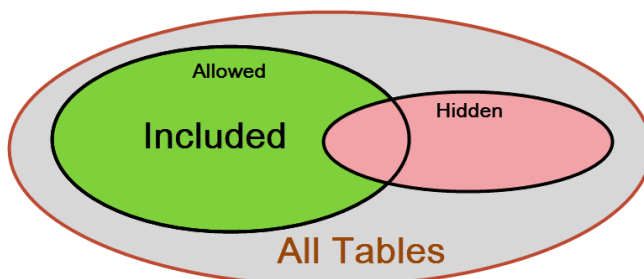
The platform automatically checks whether this file exists in the `App_Data` folder. If the customization project is published in the production environment, the platform applies the file content when a user selects a table for a generic inquiry on the *Generic Inquiry* form.

`GITables.xml` File Content

The configuration file is in XML format and includes the `GITables` element only. This element must include the `Allowed` section and can also contain the `Hidden` section. Each section is a collection of `Table` elements.

The `Allowed` collection specifies the list of tables that are available for use in generic inquiries constructed on the *Generic Inquiry* form. If a table isn't included in the `Allowed` collection, it doesn't appear in the list for selection on the **Table** tab of the form. You can also add the `Hidden` section to the configuration file. This section specifies the tables you want to exclude from the list.

If a table is allowed and not hidden (as illustrated by the green area in the figure below), it is included in the list of the tables available for constructing generic inquiries on the *Generic Inquiry* form. Otherwise, the table is not displayed in this list of tables.



Each `Table` element of the `Allowed` and `Hidden` sections contains the `FullName` attribute, which specifies the table name or the mask for a set of tables.

The attribute value is a string that can contain the following wildcard characters:

- An asterisk (*), which matches any number of characters (or no characters)
- A question mark (?), which matches exactly one character

The example below shows how to exclude the tables by using the `PX.Objects.CR.BAccount` and `PX.*Contact*` masks.

```
<?xml version="1.0" encoding="utf-8"?>
<GITables>
  <Hidden>
    <Table FullName="PX.Objects.CR.BAccount" />
    <Table FullName="PX.*Contact*" />
  </Hidden>
  <Allowed>
    <Table FullName="*" />
  </Allowed>
</GITables>
```

According to the mask with `Contact`, users will not be able to use any tables that contain the word *Contact* in the table name in their inquiries (for example, the `PX.Objects.CR.Contact` table).

To limit the list of database tables by using only the `Allowed` collection, you can empty or remove the `Hidden` section. The following example shows how to include only the tables that matched the `PX.Objects.IN.*` mask.

```
<?xml version="1.0" encoding="utf-8"?>
<GITables>
  <Allowed>
    <Table FullName="PX.Objects.IN.*" />
  </Allowed>
</GITables>
```

Custom Reports

You use the Custom Reports page of the [Customization Project Editor](#) to manage *Report* items in the customization project.



: A *Report* item contains the data set of a custom report created with Acumatica Report Designer.

The Custom Reports page displays the list of the custom reports that have been added to the customization project. The following screenshot shows a custom report that has been constructed in Acumatica Report Designer, saved to the database, and then added to the customization project.

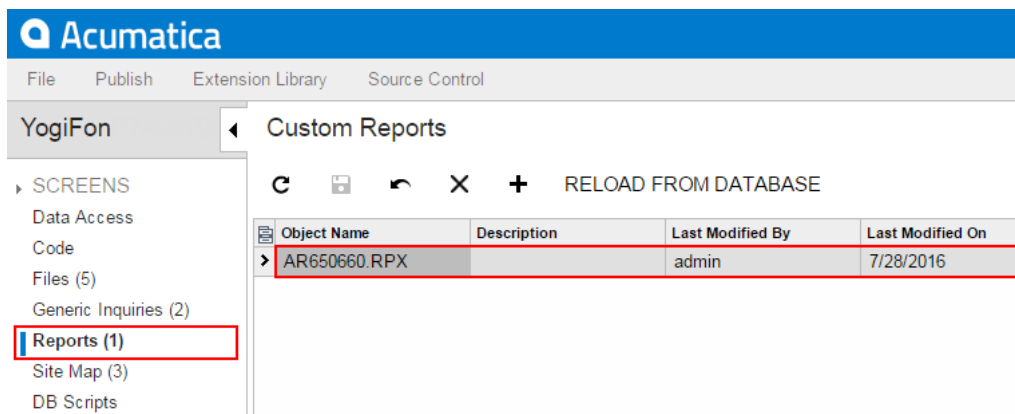


Figure: Viewing the Acumatica Report Designer report in the project

On the page, you can perform the operations described in the following topics:

- [To Add a Custom Report to a Project](#)
- [To Delete a Custom Report from a Project](#)
- [To Update a Custom Report in a Project](#)

To Add a Custom Report to a Project

You can add an Acumatica Report Designer custom report to a customization project. Before adding a report to a project, you have to construct the report in Acumatica Report Designer and save the report to the database. (For more information about reports, see [Report Designer](#) in the Acumatica Framework documentation.)

To add a custom report to a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Reports** in the navigation pane to open the Custom Reports page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In **Name** box of the **Select Report from Database** dialog box, which opens, select the report that you want to include in the project.



: If a custom report is created in Acumatica Report Designer and saved as a file in the file system, you cannot add the report to a customization project as a *Report* item.

5. In the dialog box, click **OK** to add the selected report to the customization project.

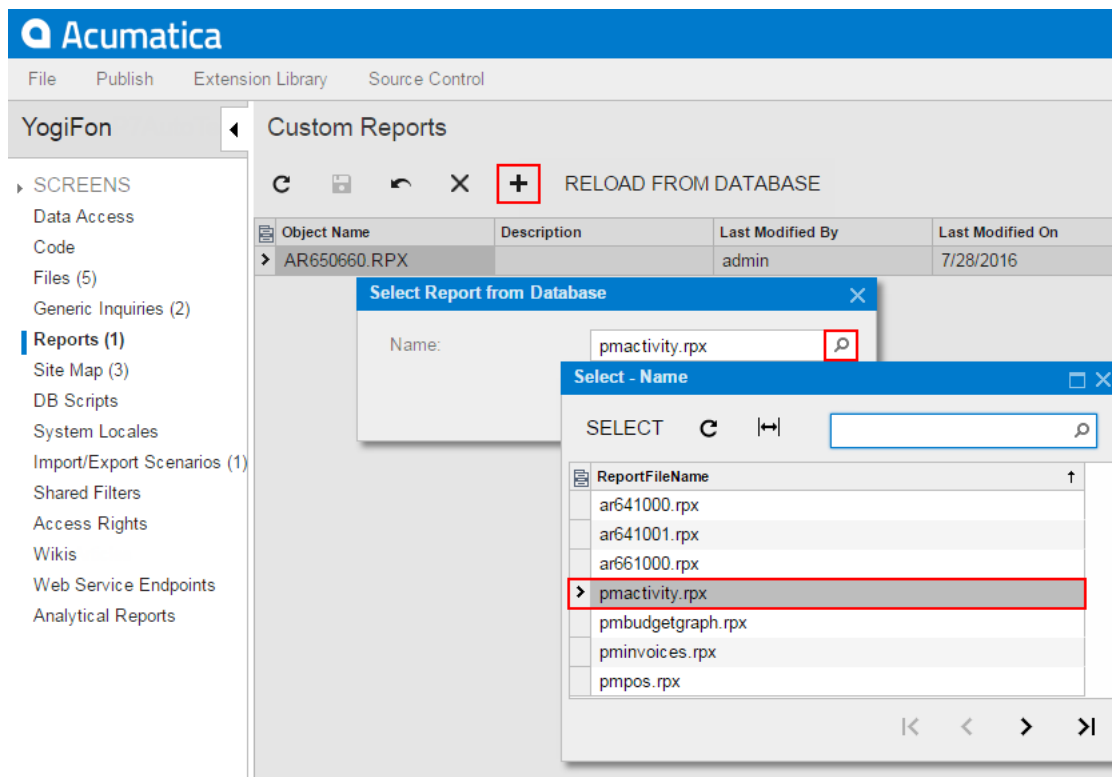


Figure: Adding a custom report to the project



: To give users the ability to navigate to the custom report in Acumatica ERP, you have to add the appropriate site map node to the customization project. See [To Add a Site Map Node to a Project](#) for details.

To Delete a Custom Report from a Project

To remove a custom report from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Reports** in the navigation pane to open the Custom Reports page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row** (X).
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you have added a site map node for the custom report to the project, you also have to delete the appropriate *SiteMapNode* item. (See [To Delete a Site Map Item from a Project](#) for details.)

To Update a Custom Report in a Project

If you have used Acumatica Report Designer to change a custom report included in the customization project, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Reports** in the navigation pane to open the Custom Reports page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *Report* items that are added to the customization project.

Site Map

You use the Site Map page of the *Customization Project Editor* to manage *SiteMapNode* items in the customization project.



: A *SiteMapNode* item contains the data set of a custom site map node for a custom form, generic inquiry, or report included in the customization project.

The Site Map page displays the list of the custom site map nodes that have been added to the customization project. The following screenshot shows the Site Map page for the customization project, which in this case contains two *SiteMapNode* items.

Object Name	Description	Last Modified By	Last Modified On
Subscription Billing Details		admin	9/1/2016
Active Contracts		admin	9/1/2016

Figure: Viewing the site map nodes in the project

On the page, you can perform the operations described in the following topics:

- [To Add a Site Map Node to a Project](#)
- [To Delete a Site Map Item from a Project](#)
- [To Update a Site Map Node in a Project](#)
- [To Redirect to the Site Map Form](#)

To Add a Site Map Node to a Project

Any change to the site map that is saved in the database for the current company can be added to a customization project as a *SiteMapNode* item. Therefore you can add a custom or customized site map node to a customization project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Site Map** in the navigation pane to open the Site Map page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of site map nodes in the **Add Site Map** dialog box, which opens, select the check box for each node that you want to include in the project.



: The **Add Site Map** dialog box displays all the custom site map nodes that have been created in the site map of Acumatica ERP and the nodes that have been modified in the site map. You can select multiple custom site map nodes to add them to the project simultaneously.

5. In the dialog box, click **Save** to add each selected site map node to the customization project.

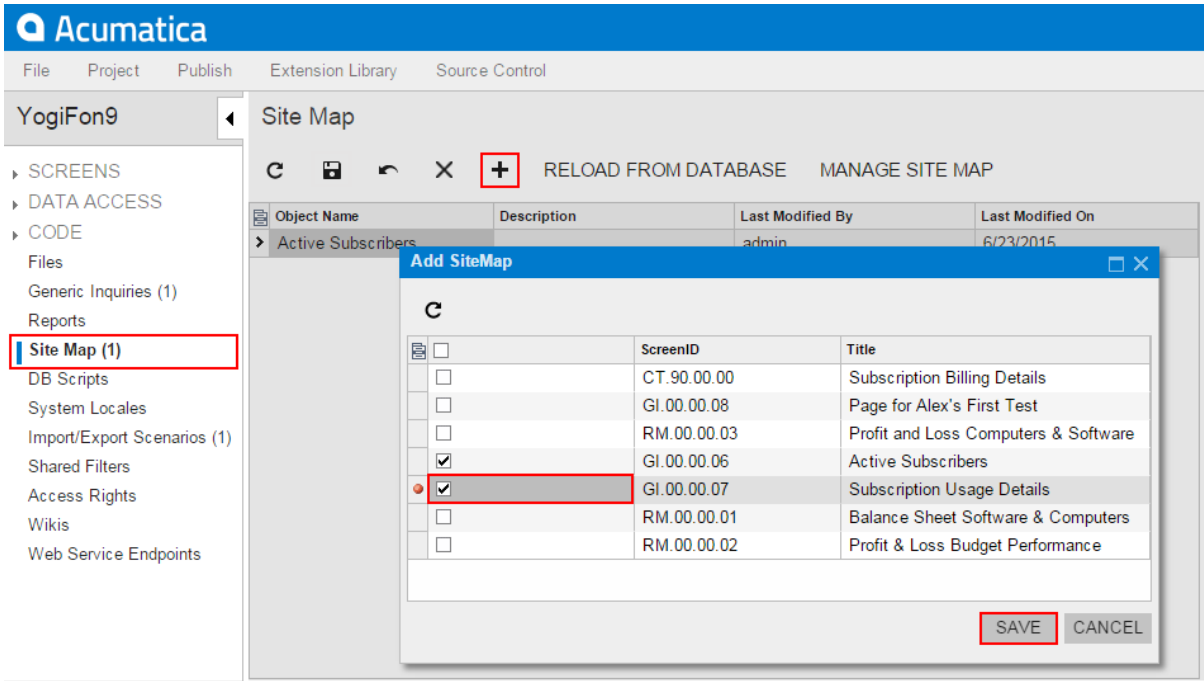


Figure: Adding the site map node to the customization project

The system adds to the project the data from the database for each selected site map node. You can view each new *SiteMapNode* item in the Project Items table of the *Item XML Editor*, as shown in the following screenshot.

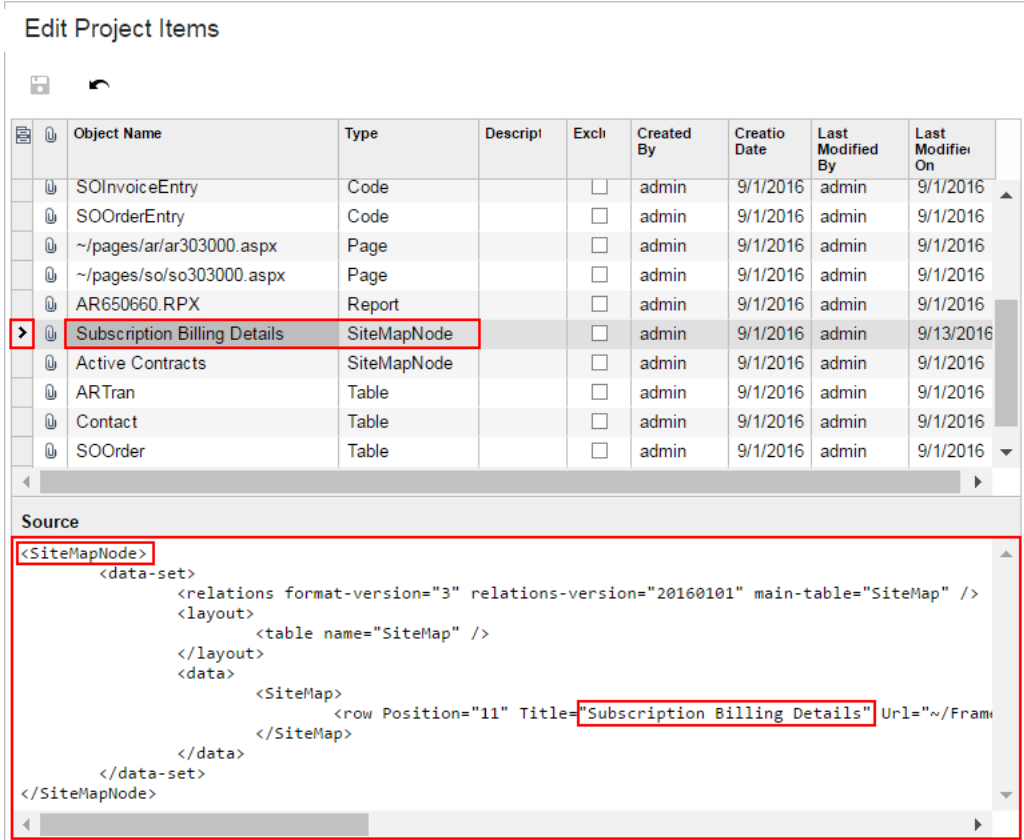


Figure: Viewing the new SiteMapNode item included in the project

To Delete a Site Map Item from a Project

To delete a site map node from a customization project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Site Map** in the navigation pane to open the Site Map page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row** (X).
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *SiteMapNode* item from the project, the node remains in the site map unless you delete the node on the [Site Map](#) (SM.20.05.20) form.

To Update a Site Map Node in a Project

If you have used the [Site Map](#) (SM.20.05.20) form to change a site map node included in the customization project, you should update this node in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Select **Site Map** in the navigation pane to open the Site Map page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *SiteMapNode* items of the project by using the appropriate data from the database.

To Redirect to the Site Map Form

You might need to add to Acumatica ERP a site map node for a custom form, inquiry, or report that you develop for a customization project. To create a custom site map node or modify an existing node, you use the [Site Map](#) (SM.20.05.20) form of Acumatica ERP.

To open the [Site Map](#) form from the [Customization Project Editor](#), perform the following actions:

1. Click **Site Map** in the navigation pane to open the Site Map page of the editor.
2. On the page toolbar, click **Manage Site Map**.

As soon as you save changes to an existing site map node or create a custom node on the [Site Map](#) form, you can add the node as a *SiteMapNode* item to the customization project.

Database Scripts

The Acumatica Customization Platform permits the use of custom SQL scripts for the following changes to the database in the scope of a customization project:

- Creation of custom tables
- Creation of views, indexes, and other database objects
- Insertion of data into tables

(See [Changes in the Database Schema](#) for details.)

You use the Database Scripts page of the [Customization Project Editor](#) to manage *Sql* items in the customization project.



: An *Sql* item contains a custom database table definition or a custom SQL script that has to be executed while the customization project is published.

The Database Scripts page displays the list of the custom SQL scripts that have been added to the customization project. The following screenshot shows the Database Scripts page for the customization project, which in this case contains two *Sql* items.

The screenshot shows the Acumatica interface. The top navigation bar includes 'File', 'Publish', 'Extension Library', and 'Source Control'. The left navigation pane shows a tree view with 'DB Scripts (2)' selected and highlighted with a red box. The main content area displays a table with the following data:

Object Name	Description	Last Modified By	Last Modified On
APAddress		admin	9/13/2016
MyScript		admin	9/13/2016

Figure: Viewing custom SQL scripts in the project

To create a custom table in the database, we recommend that you add the table schema to the customization project, as described in [To Add a Custom Table to a Project](#). To create other database objects or insert data into the tables, you have to compose the corresponding SQL script and add the script to the customization project, as described in [To Add a Custom SQL Script to a Project](#).

On the Database Scripts page, you can perform various operations, as described in the following topics:

- [To Add a Custom Table to a Project](#)
- [To Update Custom Tables in the Project](#)
- [To Add a Custom SQL Script to a Project](#)
- [To Delete an Sql Item From a Project](#)

To Add a Custom Table to a Project

To add a custom table to a customization project, perform the following actions:

1. Create the needed table in the database by using a database administration tool, such as SQL Server Management Studio.



: You have to use a naming convention that provides unique names for your custom tables so that they do not have the names of existing tables of Acumatica ERP.

2. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
3. Click **DB Scripts** in the navigation pane to open the Database Scripts page.
4. On the page toolbar, click **Add New Record (+)**.
5. In the *SQL Script Editor*, which opens, select the custom table in the **DBObject Name** box.
6. In the editor, select the **Import Table Schema from Database** check box, as shown in the screenshot below.
7. Click **OK** to make the Acumatica Customization Platform generate the table schema and add the schema to the customization project.

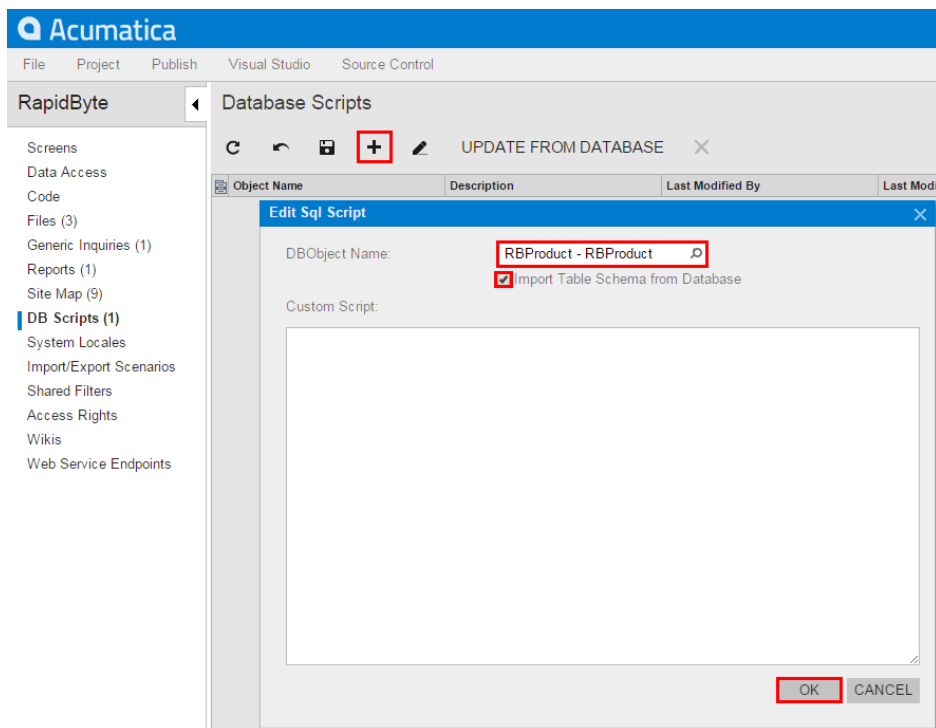


Figure: Adding a custom table to the project

Adding the table schema is the preferred way of adding custom tables to the project. When you publish the customization project, the platform executes each SQL script of the project to update the database. If an *Sql* item contains a custom database table definition, to update the database with the table schema, the Acumatica Customization Platform checks whether a table with this name already exists in the database. If the table exists, the platform generates SQL statements to alter the existing table so that it matches the schema. The platform doesn't drop the existing table and keeps any data in it. This makes it easier to deploy a newer version of the customization project to a system that is already in use. If the table doesn't exist, the platform generates SQL statements to create the table. SQL statements are generated in the SQL dialect of the database management system. Therefore, if you add custom tables to the project by table schema, you keep the customization project independent from the database management system that hosts the database of Acumatica ERP. Below is an example of the table schema of the custom table *Product*.

```
<Sql TableName="RBProduct" TableSchemaXml="#CDATA">
  <CDATA name="TableSchemaXml"><![CDATA[<table name="RBProduct">
  <col name="ProductID" type="Int" identity="true" />
  <col name="ProductCD" type="NVarChar(15)" />
  <col name="ProductName" type="NVarChar(50)" />
  <col name="Active" type="Bit" />
  <col name="StockUnit" type="NVarChar(20)" />
  <col name="UnitPrice" type="Decimal(19,6)" />
  <col name="MinAvailQty" type="Decimal(25,6)" />
  <col name="TStamp" type="Timestamp" />
  <index name="RBProduct_PK" clustered="true" primary="true" unique="true">
    <col name="ProductID" />
  </index>
</table>]]></CDATA>
</Sql>
```

Alternatively, you can add custom tables by adding a custom SQL script that creates the table in the project.

To Update Custom Tables in the Project

After you have added a custom table to the project, you might continue making changes to the table by using a database administration tool, such as SQL Server Management Studio. We recommend that you

update the table schema in the customization project before you export the deployment package of the project or publish the project.

To update the schema of custom tables in the project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **DB Scripts** in the navigation pane to open the Database Scripts page.
3. On the page toolbar, click **Update From Database**.

The platform regenerates the database table schema of all the custom tables added to the project with the **Import Table Schema from Database** check box selected.

To Add a Custom SQL Script to a Project



: Although we provide these instructions, we do not recommend that you add a custom SQL script to a customization project for the following reasons:

- Because Acumatica ERP supports multi-tenancy, it is difficult to create an SQL script that correctly creates a database object.
- It is difficult to properly specify and use the company mask in custom database objects.
- If you include in a customization project an SQL script written for MS SQL, you will need to avoid applying the customization to a website on MySQL Server, because an SQL script created for MS SQL Server will not work properly on MySQL Server.



Warning: A possible result of a custom SQL script is the loss of the integrity and consistency of data.

To add a custom SQL script to a customization project, perform the following actions:

1. Prepare and debug the SQL script with a database administration tool, such as SQL Server Management Studio. (See [Creating a Custom SQL Script](#) for details.)
2. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
3. Click **DB Scripts** in the navigation pane to open the Database Scripts page.
4. On the page toolbar, click **Add New Record (+)**.
5. In the **DBObject Name** box of the [SQL Script Editor](#), which opens, specify the name of the script to be used as the object name of the customization item.
6. In the editor, enter the SQL script into the **Custom Script** text area.
7. In the editor, click **OK** to add the script to the customization project.

If a custom script causes an error, the error message will appear during the publication process when the system executes the custom scripts.



: We really don't want you to do this, but if you plan to add a custom SQL script to a customization project, we recommend that you first test the custom SQL script on MS SQL and MySQL.

Creating a Custom SQL Script

To create database objects other than custom tables or to insert data into tables, you can add custom SQL scripts to a customization project. The Acumatica Customization Platform executes custom SQL scripts added to a project when you publish the project.



: When you publish a customization project that contains a database script, the Acumatica Customization Platform executes the script and tries to avoid executing the script at every publication of the project for optimization purposes. Therefore, the platform keeps information about each script that has been executed at least once and has not since been changed in the database, and omits the repeated execution of these scripts. If you run the **Publish with Cleanup** operation, the platform cleans all the information about previously executed scripts of the customization project and executes this scripts once more while publishing the project. See [Customization Project Editor](#) for details.

While you write scripts, keep the following conditions in mind:

- You have to use a naming convention that provides unique names for your custom objects so that they are not the same as the names of existing database objects.
- A script can be executed multiple times. Therefore, while writing the script, you have to check if an object already exists before you create a new one; otherwise, an error will occur when the script runs on the database that already contains the object.

You can prepare the script directly in the dialect of the target database or prepare the script for "on-the-fly" interpretation by Microsoft SQL Server or MySQL, depending on the target database. So you have the following options:

- **Insert the SQL script prepared for the target database, Microsoft SQL Server or MySQL:** During the publication of the project, the Acumatica Customization Platform executes the script as is. You can use all the functionality of the SQL dialect of the target database, but this script makes the customization project dependent on the database management system that hosts the database of Acumatica ERP.

The screenshot below shows a MySQL script to be added to the customization project for Acumatica ERP on MySQL.

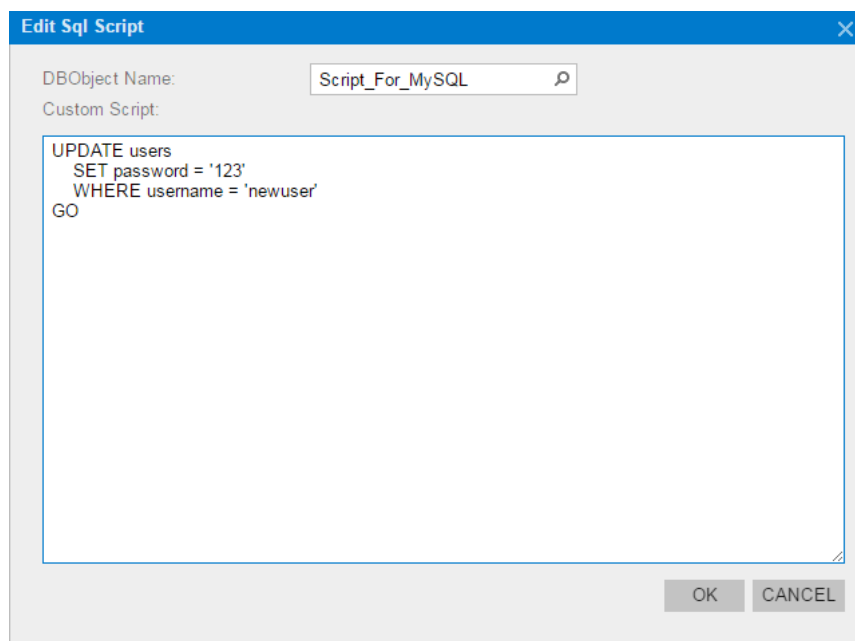


Figure: Viewing a custom script in the dialect of the target database (MySQL)

You can use SQL Script attributes to skip the script batch execution on a database management system. See [Using the SQL Script Attributes](#) for details.

- **Insert the SQL script with support for interpretation for the target database:** Prepare the script in the Microsoft SQL Server dialect and insert the script with the `-- [VmSql]` attribute. During the publication of the project, the system analyzes the script, generates the corresponding SQL statements in the dialect of the target database, and executes the statements. The customization project that includes only these scripts remains independent from the database management system that hosts the database of Acumatica ERP. However the interpreter supports a relatively limited set of SQL language elements. See [Writing Custom SQL Scripts for Interpretation](#) for details.

The screenshot below shows a Microsoft SQL Server script to be interpreted into the SQL dialect of the target database. The script will be analyzed, interpreted, and executed on either Microsoft SQL Server or MySQL.

In the script for interpretation, you have to place the `-- [VmSql]` attribute before each batch of the SQL script to be interpreted.

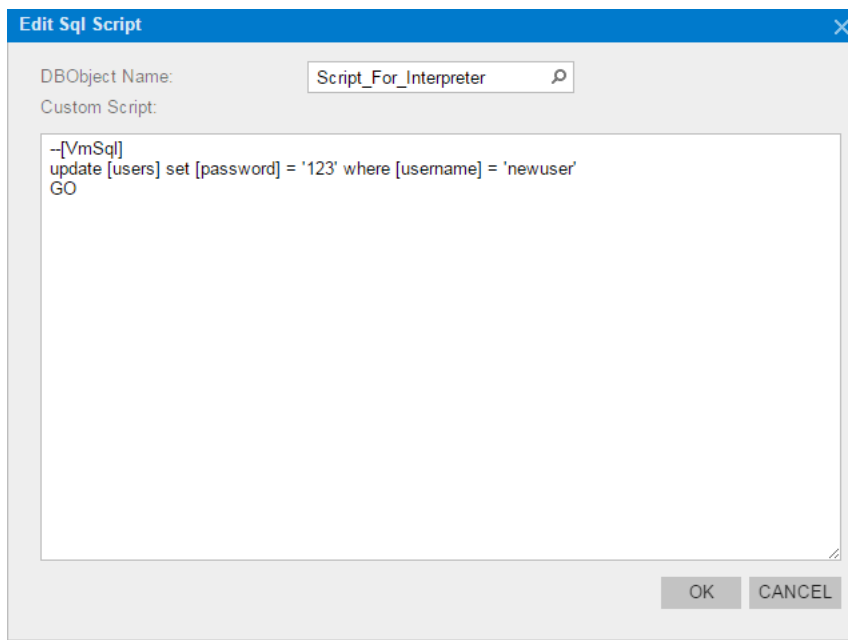


Figure: Viewing a custom script with support for interpretation

Using the SQL Script Attributes

You can decorate a batch of an SQL script with various attributes to control the batch execution.



: An SQL batch is a portion of an SQL script located between two GO statements.

Attributes are based on the line preceding the batch in the following format.

```
--[Attribute(Parameter1 = Value1, Parameter2 = Value2)]
```

Also, it is possible to specify that an attribute is effective only when the script is executed on a particular platform (MySQL or MSSQL). To achieve this, add a short database platform name followed by a colon to the beginning of the attribute name, as follows.

```
--[mysql: Attribute(Parameter1 = Value1, Parameter2 = Value2)]
--[mssql: Attribute(Parameter1 = Value1, Parameter2 = Value2)]
```

The currently used platform names are *azure*, *mssql*, and *mysql*.

We recommend that you use the following attributes for batches in SQL scripts.

Attribute	Description	Paramet	Examples
Native	The script batch will be executed against the database without any changes or attempts to parse and interpret it. You can use SQL clauses specific to the target platform.	-	--[mssql: Native]
Skip	A script batch can be skipped when the upgrade is being executed on a specific platform.	-	--[mysql: Skip]
SmartExecute	The whole batch will be executed once for every company, taking into consideration inter-company visibility mechanisms (CompanyMask). Write your code	-	--[SmartExecute]

Attribute	Description	Paramet	Examples
	in T-SQL, and it will be passed to the interpreter.		
VmSql	The decorated script batch will be interpreted as T-SQL, and a corresponding script for the current database engine will be generated and executed on the database. The number of correctly translated clauses is a limited subset of T-SQL.	-	-- [VmSql]

You can create an SQL script that is executed properly on the MySQL database platform and MSSQL database platform, as shown in the example below. This script contains two batches and demonstrates how to use SQL script attributes to control execution of the batches.

```
--[mysql: Skip]
--[mssql: Native]
IF OBJECT_ID('dbo.View1', 'V') IS NOT NULL DROP VIEW [View1];
CREATE VIEW [View1] AS (SELECT TOP 10 * FROM [AccountClass] )
GO

--[mysql: Native]
--[mssql: Skip]
DROP VIEW IF EXISTS `View1`;
CREATE VIEW `View1` AS (SELECT * FROM `AccountClass` LIMIT 10 )
GO
```

Writing Custom SQL Scripts for Interpretation

In custom scripts for interpretation, you can use the data types and SQL statements that are listed below.

Data Types

The following data types of Microsoft SQL Server are supported for interpretation:

- bit
- char, nchar, varchar, and nvarchar
- smallint, int, and bigint
- date, datetime, and datetime2
- uniqueidentifier
- decimal and double

The following data types of MySQL Server are supported for interpretation:

- binary, varbinary, and longblob
- char, varchar, and longtext
- tinyint, smallint, int, and bigint
- timestamp and datetime
- decimal and double

SQL Statements

If you prepare a script for interpretation, you can use the following elements of Transact-SQL:

- CREATE, ALTER, and DROP statements
- SELECT, INSERT, UPDATE, and DELETE statements with WHERE clauses
- Logical operators: NOT, AND, OR, and EXISTS
- Control-of-flow keywords: IF...ELSE and BEGIN...END
- Expressions: CASE, COALESCE, and NULLIF
- String functions: LEN, CONCAT, REPLACE, CHAR, RTRIM, LTRIM, SUBSTRING, UPPER, LOWER, REPLICATE, and DATALENGTH
- Arithmetic operators: +, -, *, and /
- Mathematical functions: CEILING, ROUND, and FLOOR
- Date and time functions: GETDATE, DAYADD, DATEPART, and DATALENGTH
- Aggregate functions: ABS, MIN, MAX, SUM, and COUNT
- Conversion functions: CAST and CONVERT
- System functions: ISNULL and NEWID
- System variables: @@ROWCOUNT, @@IDENTITY, and @@FETCH_STATUS
- Cryptographic functions: HASHBYTES with MD5 only
- Local variables
- Cursors
- Scalar subselect



: The EXISTS operator can be applied to `sys.tables`, `sys.column`, and `sys.indexes` objects. The DATALENGTH function can be applied to a string or binary object and returns the object length in bytes.

VmSQL Variables

In the script for interpretation, you can use the `@@@is_mssql`, `@@@is_azure`, and `@@@is_mysql` variables. The following table contains values of these variables for MS SQL Server, MS Azure SQL Database, and MySQL Server.

SQL Server	@@@is_mssql	@@@is_azure	@@@is_mysql
MS SQL Server	1	0	0
MS Azure SQL Database	1	1	0
MySQL Server	0	0	1

Error Messages

Unsupported data types cause the following error: *Cannot figure out DbType for SqlDataTypeOption.*

Unsupported elements can cause the following errors:

- *Date interval ... not recognized*: The dates are specified in an unknown format in the functions that work with datetime formats.
- *Unknown algorithm in hashbytes ... not implemented*: An unknown algorithm is specified in the HASHBYTES function. Currently, the interpreter supports MD5 only.
- *Function ... not implemented*: The script contains an unknown function that cannot be interpreted.

To Edit a Custom SQL Script

You can edit a custom SQL script once it is added to a customization project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **DB Scripts** in the navigation pane to open the Database Scripts page.
3. In the page table, select the item to be edited, as the screenshot below shows.
4. On the page toolbar, click **Edit** to open the [SQL Script Editor](#) for the selected item.



: You can click the name of an *Sql* item in the **Object Name** column of the page table to open the SQL Script Editor for the item.

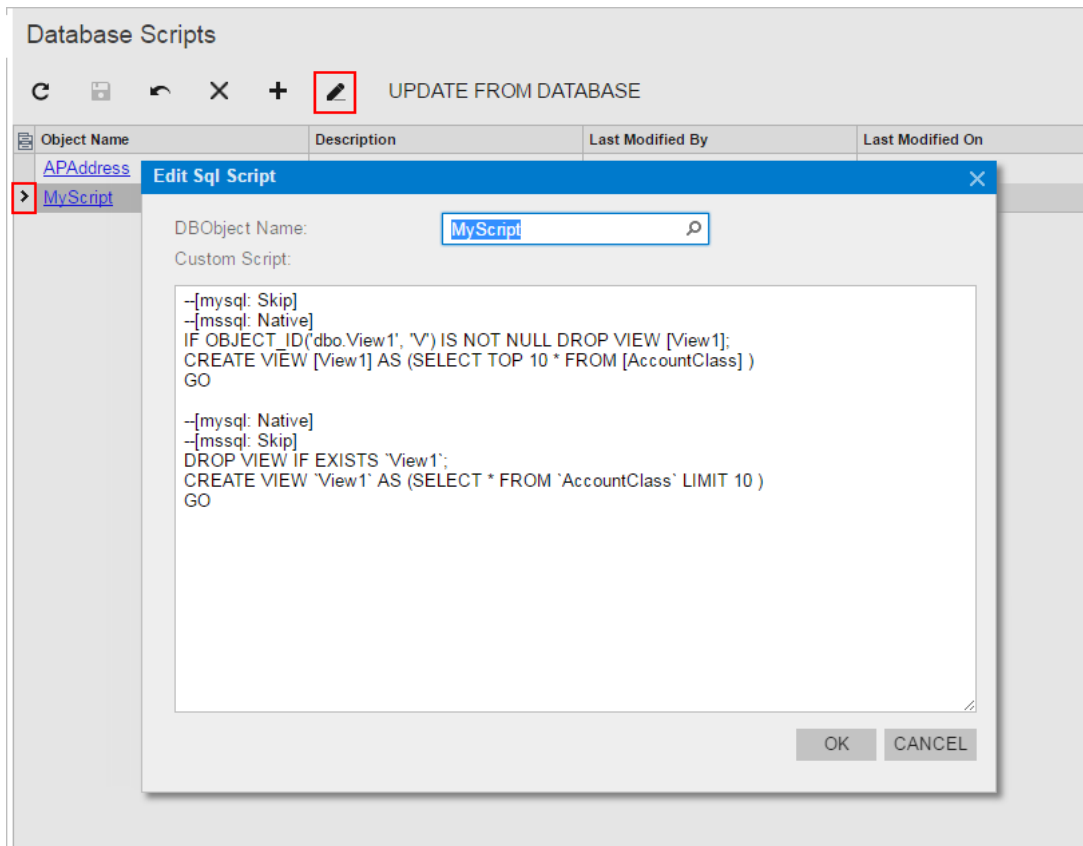


Figure: Opening the SQL Script Editor for the selected item

To Delete an Sql Item From a Project

To delete an *Sql* item from a customization project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **DB Scripts** in the navigation pane to open the Database Scripts page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row** (X).
5. On the page toolbar, click **Save** to save the changes to the customization project.

The item is removed from the customization project. The database objects that could have been created or altered if you have published the customization project remain in the database.

System Locales

You use the System Locales page of the *Customization Project Editor* to manage *Locale* items in the customization project.



: A *Locale* item contains the data set of a system locale, which is a set of parameters that defines the language and other local preferences—such as how to display numbers, dates, and times in the user interface—for a group of users.

The System Locales page displays the list of the system locales that have been added to the customization project. The following screenshot shows the System Locales page for the customization project, which in this case contains two *Locale* items.

Object Name	Description	Last Modified By	Last Modified On
en-US		admin	11/14/2016
fr-CA		admin	11/14/2016

Figure: Viewing system locales in the project

On the page, you can perform various operations, as described in the following topics:

- [To Add a System Locale to a Project](#)
- [To Delete a System Locale from a Project](#)
- [To Update a Custom System Locale in a Project](#)
- [To Redirect to the System Locales Form](#)

To Add a System Locale to a Project

You can add a system locale to a customization project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **System Locales** in the navigation pane to open the System Locales page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of system locales in the **Add Locale** dialog box, which opens, select the check box for each locale that you want to include in the project.



: The **Add Locale** dialog box displays all the system locales that exist in your instance of Acumatica ERP. You can select multiple system locales to add them to the project simultaneously.

5. In the dialog box, click **OK** to add each selected locale to the page table.
6. On the page toolbar, click **Save** to save the changes to the customization project.

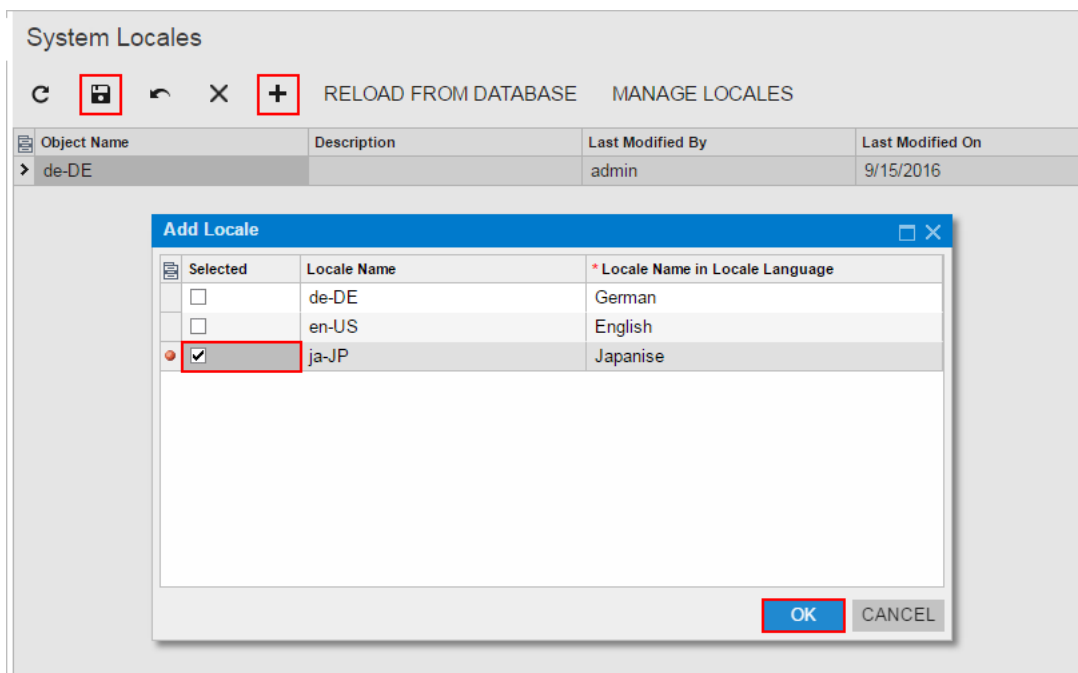


Figure: Adding the system locale to the customization project

The system adds to the project the data from the database for each selected system locale. You can view each new *Locale* item in the Project Items table of the *Item XML Editor*, as shown in the following screenshot.

Edit Project Items

Object Name	Type	Description	Exclude	Created By	Creation Date	Last Modified By	Last Modified On
Item Availability Data	General Inq...		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
ARRReleaseProcess	Code		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
CustomerMaint	Code		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
SOInvoiceEntry	Code		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
SOOrderEntry	Code		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
de-DE	Locale		<input type="checkbox"/>	admin	9/15/2016	admin	9/15/2016
ja-JP	Locale		<input type="checkbox"/>	admin	9/15/2016	admin	9/15/2016
~/pages/ar/ar303000.aspx	Page		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
~/pages/so/so303000.aspx	Page		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
AR650660.RPX	Report		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016

Source

```
<Locale>
  <data-set>
    <relations format-version="3" relations-version="20160101" main-table="Locale">
      <link from="LocalizationTranslation (Locale)" to="Locale (LocaleName)" />
      <link from="LocalizationTranslation (IdValue)" to="LocalizationValue (Id)" type="ToMaster" />
      <link from="LocalizationResource (IdValue)" to="LocalizationValue (Id)" />
      <link from="LocalizationResourceByScreen (IdRes, IdValue)" to="LocalizationResource (Id, IdValue)" />
    </relations>
    <layout>
      <table name="Locale">
        <table name="LocalizationTranslation" uplink="(LocaleName) = (Locale)" />
      </table>
      <table name="LocalizationValue">
        <table name="LocalizationResource" uplink="(Id) = (IdValue)">
          <table name="LocalizationResourceByScreen" uplink="(Id, IdValue) = (IdRes, IdValue)" />
        </table>
      </table>
    </layout>
    <data>
      <Locale>
        <row LocaleName="ja-JP" Description="Japanese" TranslatedName="Japanese" IsActive="true" />
      </Locale>
    </data>
  </data-set>
</Locale>
```

Figure: Viewing the XML code of the *Locale* item included in the project

To Delete a System Locale from a Project

To remove a *Locale* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **System Locales** in the navigation pane to open the System Locales page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row** (X).
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *Locale* item from the project, the system locale remains in the system unless you delete the locale by using the [System Locales](#) (SM.20.05.50) form.

To Update a Custom System Locale in a Project

If you have used the [System Locales](#) (SM.20.05.50) form to change a system locale included in a customization project, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)

2. Click **System Locales** in the navigation pane to open the System Locales page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *Locale* items of the project by using the appropriate data from the database.

To Redirect to the System Locales Form

You might need to add a system locale to Acumatica ERP during customization. To manage system locales in Acumatica ERP, you use the [System Locales](#) (SM.20.05.50) form.

In the [Customization Project Editor](#), to open the [System Locales](#) form, perform the following actions:

1. Click **System Locales** in the navigation pane to open the System Locales page of the editor.
2. On the page toolbar, click **Manage Locales**.

As soon as you add a system locale to the system on the [System Locales](#) form, you can add the locale as a *Locale* item to the customization project. For more information about system locales, see [System Locales](#) in the User Guide.

Import and Export Scenarios

You use the Import and Export Scenarios page of the [Customization Project Editor](#) to manage *XportScenario* items in the customization project.



: An *XportScenario* item contains the data set of a custom export or import scenario used to perform data migration between a legacy application and Acumatica ERP.

The Import and Export Scenarios page displays the list of the integration scenarios that have been added to the customization project. The following screenshot shows the Import and Export Scenarios page for the customization project, which in this case contains three *XportScenario* items.

Object Name	Description	Last Modified By	Last Modified On
ContractUsage		admin	9/19/2016
Export ARM Row Set		admin	9/19/2016
Import ARM Row Set		admin	9/19/2016

Figure: Integration scenarios in the project

On the page, you can perform a variety of operations, including the following:

- [To Add an Integration Scenario to a Project](#)
- [To Delete an Integration Scenario from a Project](#)
- [To Update an Integration Scenario in a Project](#)
- [To Redirect to the Import Scenarios Form](#)

To Add an Integration Scenario to a Project

You can add a custom integration scenario to a customization project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Import/Export Scenarios** in the navigation pane to open the Import and Export Scenarios page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of integration scenarios in the **Add Import or Export Scenario** dialog box, which opens, select the check box for each scenario that you want to include in the project.



The **Add Import or Export Scenario** dialog box displays all the custom integration scenarios that exist in your instance of Acumatica ERP. You can select multiple integration scenarios to add them to the project simultaneously.

5. In the dialog box, click **OK** to add each selected integration scenario to the page table.
6. On the page toolbar, click **Save** to save the changes to the customization project.

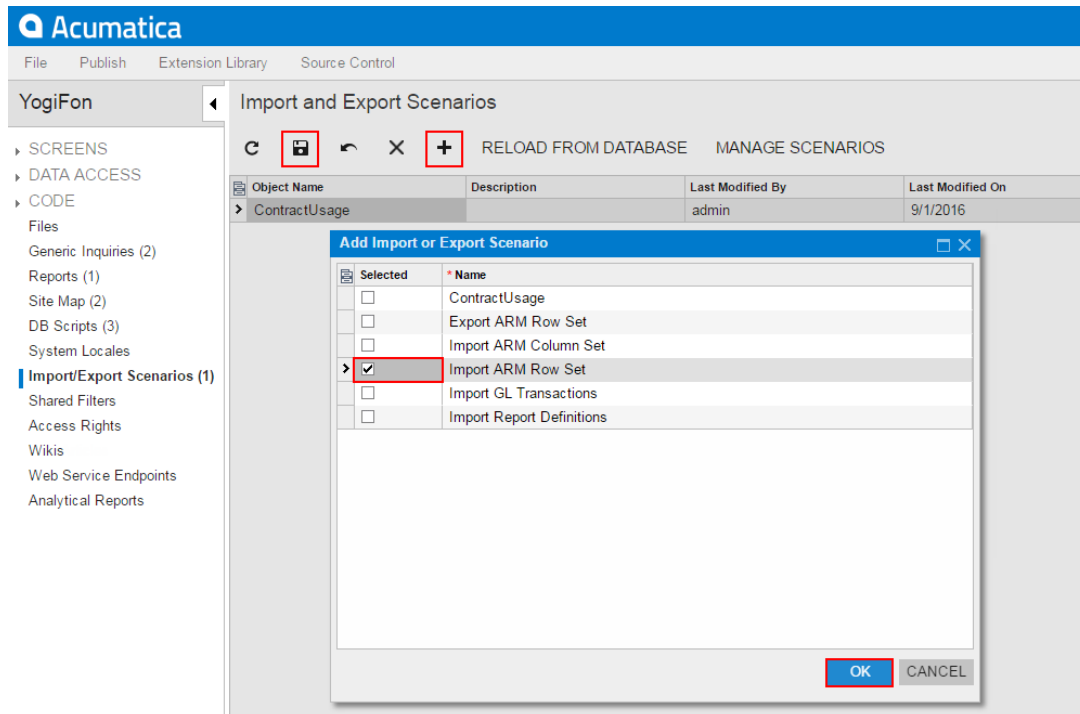


Figure: Adding an integration scenario to the customization project

The system adds to the project the data from the database for each selected integration scenario. You can view each new *XportScenario* item in the Project Items table of the [Item XML Editor](#), as shown in the following screenshot.

Edit Project Items

Object Name	Type	Desc	Exclud	Created By	Creation Date	Last Modified By	Last Modified On
APAddress	Sql		<input type="checkbox"/>	admin	9/13/2016	admin	9/13/2016
MyScript	Sql		<input type="checkbox"/>	admin	9/13/2016	admin	9/14/2016
Script_For_Interpreter	Sql		<input type="checkbox"/>	admin	9/16/2016	admin	9/16/2016
ARTran	Table		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
Contact	Table		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
SOOrder	Table		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2016
HelpRoot_Customize	WikiArticle		<input type="checkbox"/>	admin	9/19/2016	admin	9/19/2016
ContractUsage	XportScenario		<input type="checkbox"/>	admin	9/1/2016	admin	9/19/2016
>	Import ARM Row Set	XportScenario	<input type="checkbox"/>	admin	9/19/2016	admin	9/19/2016

Source

```
<XportScenario>
  <data-set>
    <relations format-version="3" relations-version="20160101" main-table="SYMapping">
      <link from="SYMapping (ProviderID, NoteID)" to="SYProvider (ProviderID, NoteID)" t
      <link from="SYMapping (NoteID)" to="Note (NoteID)" type="ToMaster" />
      <link from="SYProvider (NoteID)" to="Note (NoteID)" type="ToMaster" />
      <link from="SYProviderField (ProviderID)" to="SYProvider (ProviderID)" />
      <link from="SYProviderField (NoteID)" to="Note (NoteID)" type="Note" />
      <link from="SYProviderObject (ProviderID)" to="SYProvider (ProviderID)" />
      <link from="SYProviderObject (NoteID)" to="Note (NoteID)" type="Note" />
      <link from="SYProviderParameter (ProviderID)" to="SYProvider (ProviderID)" />
      <link from="SYProviderParameter (NoteID)" to="Note (NoteID)" type="Note" />
      <link from="SYMappingCondition (MappingID)" to="SYMapping (MappingID)" />
      <link from="SYMappingCondition (NoteID)" to="Note (NoteID)" type="Note" />
      <link from="SYMappingField (MappingID)" to="SYMapping (MappingID)" />
      <link from="SYMappingField (NoteID)" to="Note (NoteID)" type="Note" />
      <link from="SYImportCondition (MappingID)" to="SYMapping (MappingID)" />
      <link from="SYImportCondition (NoteID)" to="Note (NoteID)" type="Note" />
    </relations>
    <layout>
      <table name="SYMapping">
```

Figure: Viewing the XML code of the *XportScenario* item included in the project

An *XportScenario* item contains all the data required for the integration scenario. Therefore, the item includes the data of the data provider.

To Delete an Integration Scenario from a Project

To remove an *XportScenario* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Import/Export Scenarios** in the navigation pane to open the Import and Export Scenarios page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete an *XportScenario* item from the project, the integration scenario remains in the system unless you delete the scenario by using the [Import Scenarios](#) (SM.20.60.25) form.

To Update an Integration Scenario in a Project

If you have used the [Import Scenarios](#) (SM.20.60.25) form to change an integration scenario included in a customization project, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)

2. Click **Import/Export Scenarios** in the navigation pane to open the Import and Export Scenarios page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *XportScenario* items of the project by using the appropriate data from the database.

To Redirect to the Import Scenarios Form

You might need to add an import or export scenario to Acumatica ERP during customization. To manage import scenarios in Acumatica ERP, you use the [Import Scenarios](#) (SM.20.60.25) form.

In the [Customization Project Editor](#), to open this form, perform the following actions:

1. Select **Import/Export Scenarios** in the navigation pane to open the Import and Export Scenarios page.
2. On the page toolbar, click **Manage Scenarios**.

To manage export scenarios in Acumatica ERP, you use the [Export Scenarios](#) (SM.20.70.25) form.

As soon as you add an integration scenario to the system on the [Import Scenarios](#) or [Export Scenarios](#) form, you can add the scenario as an *XportScenario* item to the customization project. For more information about integration scenarios, see the [Integration](#) part of the User Guide.

Shared Filters

Users can create reusable filters, which are available on processing and inquiry forms in Acumatica ERP, to filter the data in the table part of the form; these filters can be reused anytime after creation. Reusable filters can be shared among all users of the system.

You use the Shared Filters page of the [Customization Project Editor](#) to manage *SharedFilter* items in the customization project.



: A *SharedFilter* item contains the data set of a custom reusable shared filter created on a processing or inquiry form of Acumatica ERP.

The Shared Filters page displays the list of the custom reusable shared filters that have been added to the customization project. The following screenshot shows the Shared Filters page for the customization project, which in this case contains two *SharedFilter* items.

The screenshot displays the Acumatica Shared Filters page. The page title is "Shared Filters". The toolbar includes icons for refresh, save, undo, close, and a plus sign, along with the text "RELOAD FROM DATABASE" and "MANAGE FILTERS". The table below shows the following data:

Object Name	Description	Last Modified By	Last Modified On
MySharedFilter&GL505510		admin	9/19/2016
Test&GI000001		admin	9/19/2016

Figure: Viewing the custom reusable shared filters in the project

On the page, you can perform several operations, as described in the following topics:

- [To Add a Shared Filter to a Project](#)
- [To Delete a Shared Filter from a Project](#)
- [To Update a Shared Filter in a Project](#)
- [To Redirect to the Filters Form](#)

To Add a Shared Filter to a Project

You can add a custom reusable shared filter to a customization project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Shared Filters** in the navigation pane to open the Shared Filters page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of integration scenarios in the **Add Shared Filter** dialog box, which opens, select the check box for each filter that you want to include in the project.



: The **Add Shared Filter** dialog box displays all the custom shared filters that exist in your instance of Acumatica ERP. You can select multiple shared filters to add them to the project simultaneously.

5. In the dialog box, click **OK** to add the selected filter or filters to the page table.
6. On the page toolbar, click **Save** to save the changes to the customization project.

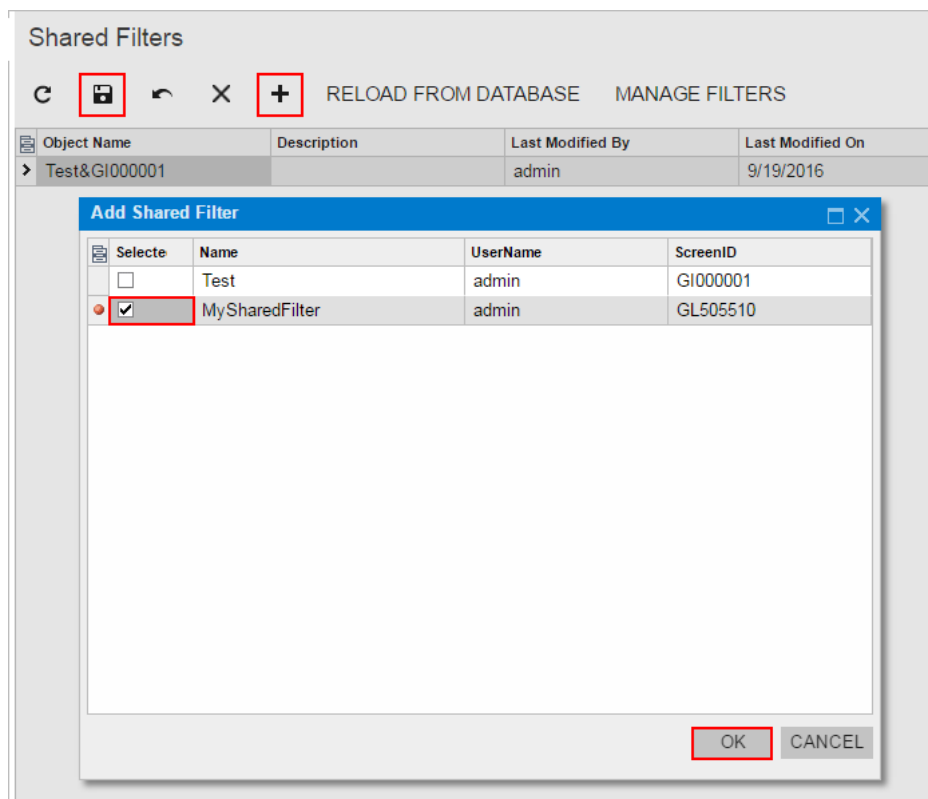


Figure: Adding the shared filter to the customization project

The system adds to the project the data from the database for each selected shared filter. You can view each new *SharedFilter* item in the Project Items table of the [Item XML Editor](#), as shown in the following screenshot.

Edit Project Items

Object Name	Type	Descri	Exclud	Created By	Creation Date	Last Modified
SOOrderEntry	Code		<input type="checkbox"/>	admin	9/1/2016	adm
de-DE	Locale		<input type="checkbox"/>	admin	9/15/2016	adm
en-US	Locale		<input type="checkbox"/>	admin	9/19/2016	adm
ja-JP	Locale		<input type="checkbox"/>	admin	9/15/2016	adm
~/pages/ar/ar303000.aspx	Page		<input type="checkbox"/>	admin	9/1/2016	adm
~/pages/so/so303000.aspx	Page		<input type="checkbox"/>	admin	9/1/2016	adm
AR650660.RPX	Report		<input type="checkbox"/>	admin	9/19/2016	adm
Profit & Loss Budget Performance	ReportDefinition		<input type="checkbox"/>	admin	9/19/2016	adm
RM000001	ScreenWithRights		<input type="checkbox"/>	admin	9/19/2016	adm
MySharedFilter&GL505510	SharedFilter		<input type="checkbox"/>	admin	9/19/2016	adm
Test&GI000001	SharedFilter		<input type="checkbox"/>	admin	9/19/2016	adm
Subscription Billing Details	SiteMapNode		<input type="checkbox"/>	admin	9/1/2016	adm
Finance	SiteMapNode		<input type="checkbox"/>	admin	9/19/2016	adm
Active Contracts	SiteMapNode		<input type="checkbox"/>	admin	9/1/2016	adm

Source

```
<SharedFilter>
  <data-set>
    <relations format-version="3" relations-version="20160101" main-table="FilterH
      <link from="FilterRow (FilterID)" to="FilterHeader (FilterID)" />
    </relations>
    <layout>
      <table name="FilterHeader">
        <table name="FilterRow" uplink="(FilterID) = (FilterID)" />
      </table>
    </layout>
    <data>
      <FilterHeader>
        <row FilterID="417" UserName="admin" ScreenID="GL505510" ViewN
```

Figure: Viewing the XML code of the *SharedFilter* item included in the project

To Delete a Shared Filter from a Project

To remove a *SharedFilter* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Shared Filters** in the navigation pane to open the Shared Filters page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *SharedFilter* item from the project, the custom reusable shared filter remains in the system unless you delete the filter by using the [Filters](#) (CS.20.90.10) form.

To Update a Shared Filter in a Project

If you have used the [Filters](#) (CS.20.90.10) form to change a shared filter included in a customization project, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Shared Filters** in the navigation pane to open the Shared Filters page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *SharedFilter* items of the project by using the appropriate data from the database.

To Redirect to the Filters Form

You might need to add or change a reusable shared filter in Acumatica ERP during a customization. To manage shared filters in Acumatica ERP, you use the *Filters* (CS.20.90.10) form.

In the *Customization Project Editor*, to open this form, perform the following actions:

1. Click **Shared Filters** in the navigation pane to open the Shared Filters page.
2. On the page toolbar, click **Manage Filters**.

As soon as you add a shared filter to the system on the *Filters* form, you can add the filter as a *SharedFilter* item to the customization project. For more information about reusable filters, see *Reusable Filters* in the Interface Guide.

Access Rights

In Acumatica ERP, you can control access to system objects at broad and granular levels, down to the control of form elements, such as buttons, text boxes, and check boxes. Users are assigned to roles, and you give these roles the appropriate levels of access rights to system objects—suites, modules, forms, and form elements.

You use the Access Rights page of the *Customization Project Editor* to manage *ScreenWithRights* items in the customization project.



: A *ScreenWithRights* item contains the data set of custom access rights of roles to a form, down to the control of form elements.

The Access Rights page displays the list of the custom access rights of roles that have been added to the customization project. The following screenshot shows the Access Rights page for the customization project, which in this case contains two *ScreenWithRights* items.

Object Name	Description	Last Modified By	Last Modified On
RM000001		admin	9/19/2016
RM000003		admin	9/20/2016

Figure: Viewing the custom access rights of roles in the project

On the page, you can perform several operations, as described in the following topics:

- [To Add Access Rights to a Project](#)
- [To Delete Access Rights from a Project](#)
- [To Update Access Rights in a Project](#)

- [To Redirect to the Access Rights by Screen Form](#)

To Add Access Rights to a Project

When you create or change the access rights by screen, role, or user in an instance of Acumatica ERP, these changes are saved in the database for the current company.

You can add to a customization project the access rights of roles by screen that are saved in the database for the current company. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Access Rights** in the navigation pane to open the Access Rights page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the **Add Access Rights for Screen** dialog box, which opens, in the list of custom access rights of roles by screen, select the check box for the access rights that you want to include in the project.



: The **Add Access Rights for Screen** dialog box displays all the custom access rights of roles that exist in your instance of Acumatica ERP. You can select multiple access rights of roles to add them to the project simultaneously.

5. In the dialog box, click **OK** to add the selected access rights of roles to the page table.
6. On the page toolbar, click **Save** to save the changes to the customization project.

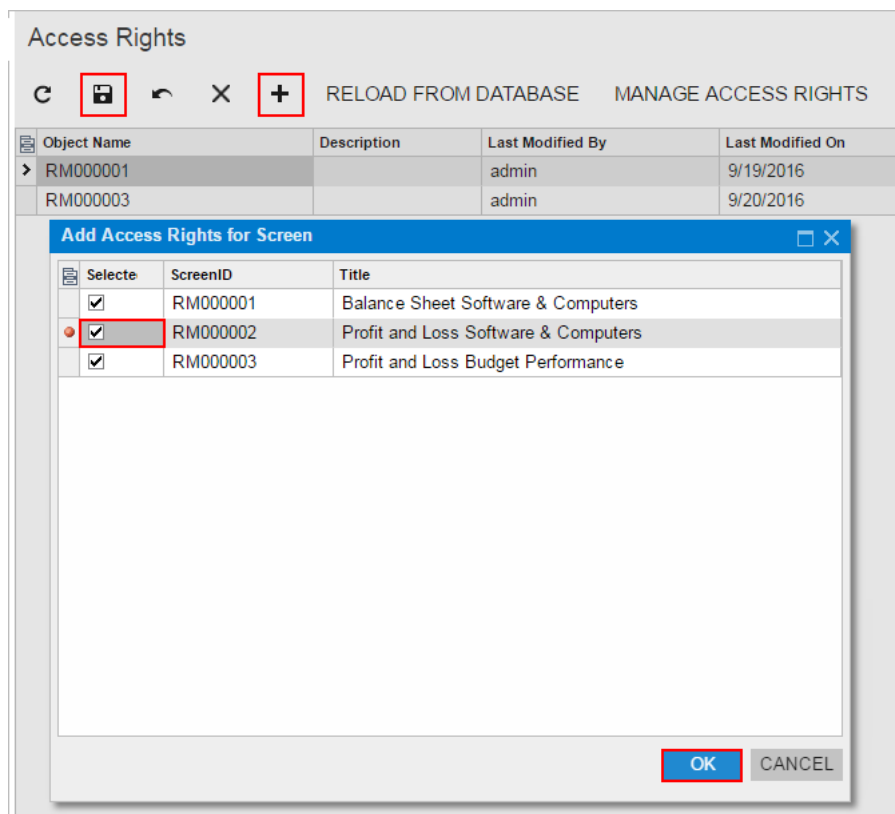


Figure: Adding the access rights to the customization project

The system adds to the project the data from the database for the selected access rights of roles. You can view each new *ScreenWithRights* item in the Project Items table of the [Item XML Editor](#), as shown in the following screenshot.

Edit Project Items

Object Name	Type	Descri	Exc	Created By	Creati Date	Last Modifie By
AR650660.RPX	Report		<input type="checkbox"/>	admin	9/19/201	ad
Profit & Loss Budget Performance	ReportDefinition		<input type="checkbox"/>	admin	9/19/201	ad
RM000001	ScreenWithRights		<input type="checkbox"/>	admin	9/19/201	ad
RM000002	ScreenWithRights		<input type="checkbox"/>	admin	9/20/201	ad
RM000003	ScreenWithRights		<input type="checkbox"/>	admin	9/20/201	ad
MySharedFilter&GL505510	SharedFilter		<input type="checkbox"/>	admin	9/19/201	ad
Test&GI000001	SharedFilter		<input type="checkbox"/>	admin	9/19/201	ad
Subscription Billing Details	SiteMapNode		<input type="checkbox"/>	admin	9/1/2016	ad

Source

```
<ScreenWithRights>
  <data-set>
    <relations format-version="3" relations-version="20160101" main-table="
      <link from="RolesInCache (ScreenID)" to="SiteMap (ScreenID)" />
      <link from="RolesInGraph (ScreenID)" to="SiteMap (ScreenID)" />
      <link from="RolesInMember (ScreenID)" to="SiteMap (ScreenID)" />
      <link from="Roles (Rolename, ApplicationName)" to="RolesInCache
      <link from="Roles (Rolename, ApplicationName)" to="RolesInGraph
      <link from="Roles (Rolename, ApplicationName)" to="RolesInMembe
    </relations>
    <layout>
      <table name="SiteMap">
        <table name="RolesInCache" uplink="(ScreenID) = (Screen
        <table name="RolesInGraph" uplink="(ScreenID) = (Screen
        <table name="RolesInMember" uplink="(ScreenID) = (Scree
      </table>
    </layout>
  </data-set>
</ScreenWithRights>
```

Figure: Viewing the XML code of the *ScreenWithRights* item included in the project

A *ScreenWithRights* item contains all the data required for the access rights of roles to the screen. Therefore, the item includes the data of all the roles applied to the screen.

To Delete Access Rights from a Project

To remove a *ScreenWithRights* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Access Rights** in the navigation pane to open the Access Rights page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row** (X).
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *ScreenWithRights* item from the project, the customization of the access rights of roles remain in the system unless you delete the changes of access rights from the database.

To Update Access Rights in a Project

If you have used Acumatica ERP forms to change access rights included in a customization project, you have to update the appropriate items in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Access Rights** in the navigation pane to open the Access Rights page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *ScreenWithRights* items of the project by using the appropriate data from the database.

To Redirect to the Access Rights by Screen Form

You might need to change the access rights of roles in Acumatica ERP during customization. To manage access rights of roles by screen in Acumatica ERP, you use the [Access Rights by Screen](#) (SM.20.10.20) form.

To open this form from the [Customization Project Editor](#), perform the following actions:

1. Click **Access Rights** in the navigation pane to open the Access Rights page.
2. On the page toolbar, click **Manage Access Rights**.

As soon as you change the access rights of roles in Acumatica ERP, the system saves the changes in the database for your company, and you can add the access rights as a *ScreenWithRights* item to the customization project. For more information about the access rights of roles, see [Managing User Access Rights](#) in the User Guide.

Wikis

You use the Wikis page of the [Customization Project Editor](#) to manage *WikiArticle* items in the customization project.



: A *WikiArticle* item contains the data set of a custom wiki and all the articles created within this wiki.

The Wikis page displays the list of the custom wikis that have been added to the customization project. The following screenshot shows the Wikis page for the customization project, which in this case contains two *WikiArticle* items.

The screenshot shows the Acumatica interface with the 'Wikis' page open. The left navigation pane is expanded to show 'Wikis (2)'. The main content area displays a table with the following data:

Object Name	Description	Last Modified By	Last Modified On
HelpRoot_Customize		admin	9/19/2016
HelpRoot_KB		admin	9/21/2016

Figure: Viewing custom wikis in the project

On the page, you can perform several operations, as described in the following topics:

- [To Add a Custom Wiki to a Project](#)
- [To Delete a Custom Wiki from a Project](#)
- [To Update a Custom Wiki in a Project](#)
- [To Redirect to the Wiki Form](#)

To Add a Custom Wiki to a Project

In Acumatica ERP, you can create a new wiki or modify the properties of an existing one. For example, you can change the access rights to wiki folders and edit the list of categories available for the wiki. Any change to wikis is saved for the appropriate wiki in the database for the current company.

You can add to a customization project the wiki that are saved in the database for the current company. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Wikis** in the navigation pane to open the Wikis page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of custom wikis in the **Add Wiki Page** dialog box, which opens, select the check box for each wiki that you want to include in the project.



: The **Add Wiki Page** dialog box displays all the custom wikis that exist in your instance of Acumatica ERP. You can select multiple wikis to add them to the project simultaneously.

5. In the dialog box, click **OK** to add the selected wiki to the page table.
6. On the page toolbar, click **Save** to save the changes to the customization project.

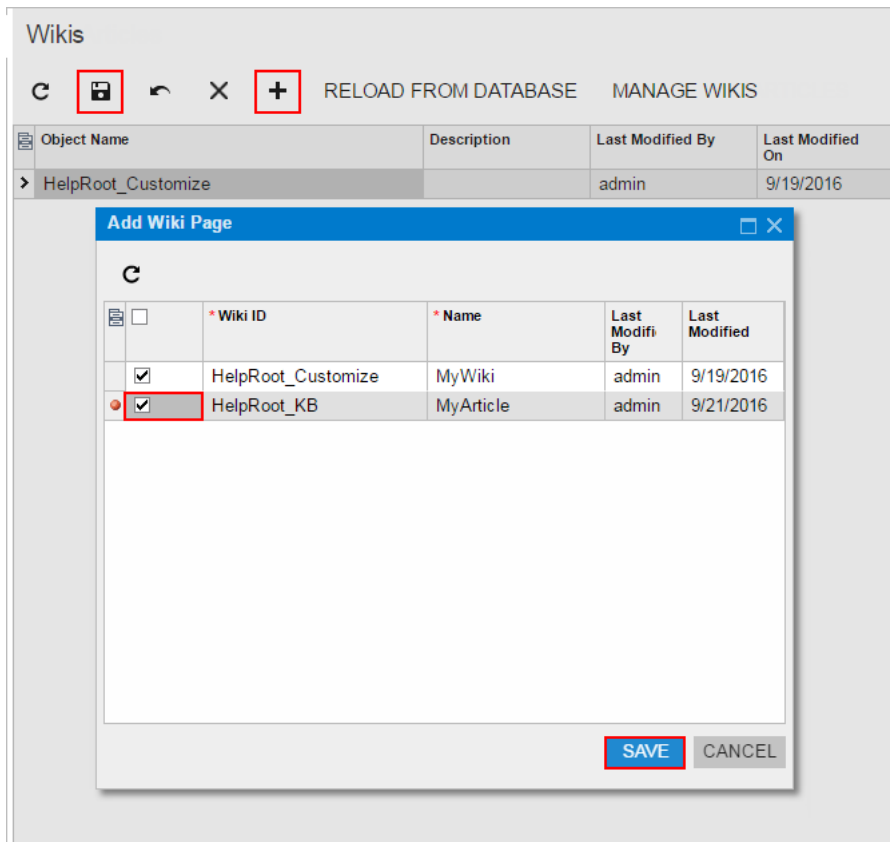


Figure: Adding the custom wiki to the customization project

The system adds to the project each selected wiki. You can view each new *WikiArticle* item in the Project Items table of the [Item XML Editor](#), as shown in the following screenshot.

Edit Project Items

Object Name	Type	Descripti	Excl	Created By	Creatic Date	Last Modified By	Last Modifie On
Active Contracts	SiteMapNode		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2
APAddress	Sql		<input type="checkbox"/>	admin	9/13/2016	admin	9/13/
MyScript	Sql		<input type="checkbox"/>	admin	9/13/2016	admin	9/14/
Script_For_Interpreter	Sql		<input type="checkbox"/>	admin	9/16/2016	admin	9/16/
ARTran	Table		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2
Contact	Table		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2
SOOrder	Table		<input type="checkbox"/>	admin	9/1/2016	admin	9/1/2
HelpRoot_Customize	WikiArticle		<input type="checkbox"/>	admin	9/19/2016	admin	9/19/
HelpRoot_KB	WikiArticle		<input type="checkbox"/>	admin	9/21/2016	admin	9/21/
ContractUsage	XportScenario		<input type="checkbox"/>	admin	9/1/2016	admin	9/19/

Source

```
<WikiArticle>
  <data-set>
    <relations format-version="3" relations-version="20160101" main-table="WikiArticle">
      <link from="WikiAccessRights (PageID)" to="WikiPage (PageID)" />
      <link from="WikiPageLanguage (PageID)" to="WikiPage (PageID)" />
      <link from="WikiRevision (PageID, Language, PageRevisionID)" to="WikiPage (PageID)" />
      <link from="WikiRevision (UID)" to="Dual (UID)" type="NewUID" />
      <link from="WikiPage (PageID)" to="WikiDescriptor (PageID)" type="NewUID" />
      <link from="WikiDescriptor (PageID)" to="WikiDescriptorExt (PageID)" type="NewUID" />
      <link from="WikiCss (CssID)" to="WikiDescriptor (CssID)" />
      <link from="WikiFileInPage (PageID, Language, PageRevisionID)" to="UploadFile (FileID)" type="NewUID" />
      <link from="UploadFileRevision (FileID, FileRevisionID)" to="UploadFile (FileID)" type="NewUID" />
    </relations>
  </data-set>
</WikiArticle>
```

Figure: Viewing the XML code of the *WikiArticle* item included in the project

A *WikiArticle* item contains all the data required to recreate the corresponding wiki in any instance of Acumatica ERP.

To Delete a Custom Wiki from a Project

To remove a *WikiArticle* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Wikis** in the navigation pane to open the Wikis page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *WikiArticle* item from the project, the custom wiki remains in the system unless you delete the wiki by using the [Wiki \(SM.20.20.05\)](#) form.

To Update a Custom Wiki in a Project

If you have changed a wiki included in a customization project by using Acumatica ERP forms, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Wikis** in the navigation pane to open the Wikis page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *WikiArticle* items of the project by using the appropriate data from the database.

To Redirect to the Wiki Form

You might need to add a new wiki or edit an existing one in Acumatica ERP during customization. To manage wikis in Acumatica ERP, you use the [Wiki](#) (SM.20.20.05) form.

To open this form from the [Customization Project Editor](#), perform the following actions:

1. Click **Wikis** in the navigation pane to open the Wikis page.
2. On the page toolbar, click **Manage Wikis**.

As soon as you create or change a wiki in the system, the system saves this wiki in the database for the current company, you can add the wiki as a *WikiArticle* item to the customization project. For more information about wikis, see [Managing Wikis](#) in the User Guide.

Web Service Endpoints

Acumatica ERP provides web services for integration with external systems. Through the web services of Acumatica ERP, external systems can get data records from Acumatica ERP and process these records; new or updated records can also be saved to Acumatica ERP. You can configure contract-based web service endpoints in an instance of Acumatica ERP and include the new configuration in a customization project as an *EntityEndpoint* item.

You use the Web Service Endpoints page of the [Customization Project Editor](#) to manage *EntityEndpoint* items in the customization project.



: A *EntityEndpoint* item contains the data set of a custom contract-based web service endpoint.

The Web Service Endpoints page displays the list of the custom contract-based web service endpoints that have been added to the customization project. The following screenshot shows the Web Service Endpoints page for the customization project, which in this case contains two *EntityEndpoint* items.

The screenshot displays the Acumatica interface for managing web service endpoints. The main area shows a table with the following data:

Object Name	Description	Last Modified By	Last Modified On
6.00.001&MyTestEndpoint		admin	11/14/2016
6.00.002&AP_EP		admin	11/14/2016

Figure: Viewing custom contract-based web service endpoints in the project

On the page, you can perform several operations, as described in the following topics:

- [To Add a Custom Web Service Endpoint to a Project](#)
- [To Delete a Custom Web Service Endpoint from a Project](#)
- [To Update a Custom Web Service Endpoint in a Project](#)

- [To Redirect to the Web Service Endpoints Form](#)

To Add a Custom Web Service Endpoint to a Project

To add a custom contract-based web service endpoint to a customization project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Web Service Endpoints** in the navigation pane to open the Web Service Endpoints page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of custom contract-based web service endpoints in the **Add Entity Endpoint** dialog box, which opens, select the check box for each endpoint that you want to include in the project.



The **Add Entity Endpoint** dialog box displays all the custom contract-based web service endpoints that exist in your instance of Acumatica ERP. You can select multiple endpoints to add them to the project simultaneously.

5. In the dialog box, click **OK** to add each selected endpoint to the page table.
6. On the page toolbar, click **Save** to save the changes to the customization project.

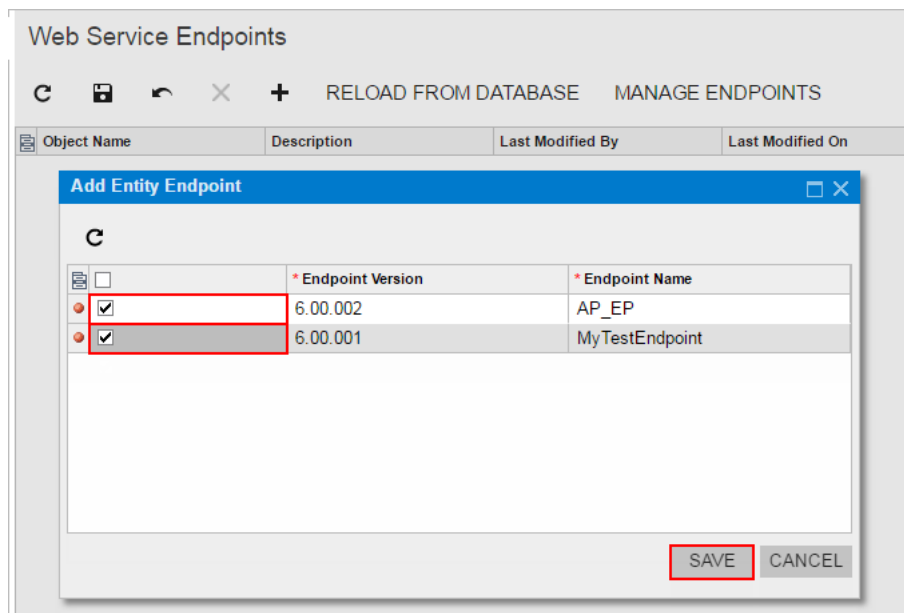


Figure: Adding the custom contract-based web service endpoint to the customization project

The system adds to the project the data from the database for each selected web service endpoint. You can view each new *EntityEndpoint* item in the Project Items table of the [Item XML Editor](#), as shown in the following screenshot.

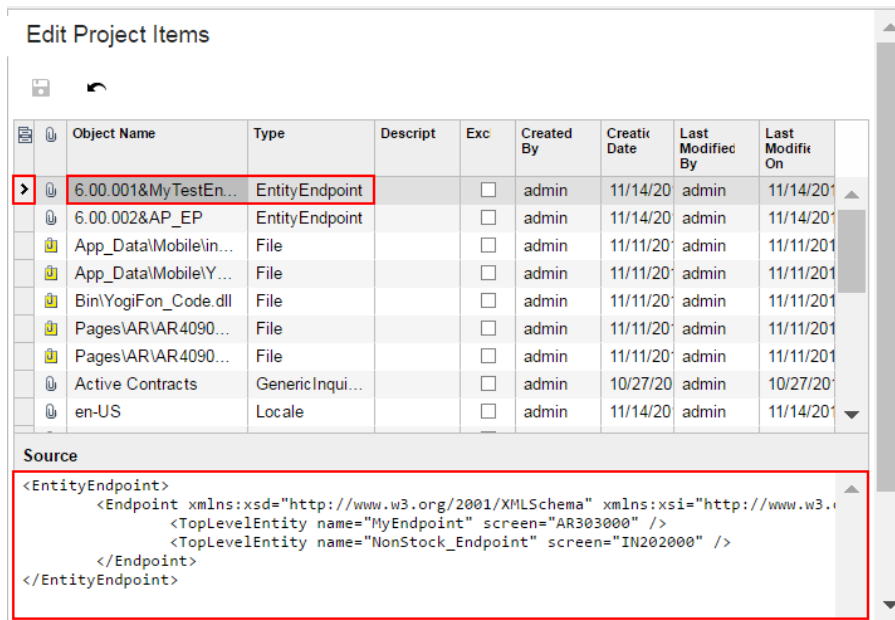


Figure: Viewing the XML code of the *EntityEndpoint* item included in the project

To Delete a Custom Web Service Endpoint from a Project

To remove a *EntityEndpoint* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Web Service Endpoints** in the navigation pane to open the Web Service Endpoints page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *EntityEndpoint* item from the project, the custom contract-based web service endpoint remains in the system unless you delete the endpoint by using the [Web Service Endpoints \(SM.20.70.60\)](#) form.

To Update a Custom Web Service Endpoint in a Project

If you have used the [Web Service Endpoints \(SM.20.70.60\)](#) form to change a custom contract-based web service endpoint included in a customization project, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Web Service Endpoints** in the navigation pane to open the Web Service Endpoints page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *EntityEndpoint* items of the project by using the appropriate data from the database.

To Redirect to the Web Service Endpoints Form

You might need to configure contract-based web service endpoints in Acumatica ERP during customization. To manage contract-based web service endpoints in Acumatica ERP, you use the [Web Service Endpoints \(SM.20.70.60\)](#) form.

To open this form from the [Customization Project Editor](#), perform the following actions:

1. Click **Web Service Endpoints** in the navigation pane to open the Web Service Endpoints page of the editor.
2. On the page toolbar, click **Manage Endpoints**.

As soon as you configure a web service endpoint on the *Web Service Endpoints* form, the system saves the endpoint data in the database for the current company, and you can add the endpoint as a *EntityEndpoint* item to the customization project. For more information about the contract-based web services API, see *Configuring the Contract-Based SOAP and REST API* in the User Guide.

Analytical Reports

Analytical reports are used to display the consolidated and summarized data in a view defined by the report designer at the report design stage. The analytical reports are used to provide the summarized and aggregated values in a variety of views: the data in the analytical report can be displayed in the monthly, yearly, and quarterly views, and the data can provide information from the point of view of departments, selected account classes, and other dimensions.

In Acumatica ERP, you can modify defined analytical reports, create a new analytical report, and delete existing reports. You can add the analytical reports that have been created or modified to the customization project as *ReportDefinition* items.

You use the Analytical Reports page of the *Customization Project Editor* to manage *ReportDefinition* items in the customization project.



: A *ReportDefinition* item contains the data set of a custom analytical report, including the data of a predefined sets of rows, columns, and units.

The Analytical Reports page displays the list of the custom analytical reports that have been added to the customization project. The following screenshot shows the Analytical Reports page for the customization project, which in this case contains two *ReportDefinition* items.

The screenshot shows the Acumatica interface for managing analytical reports. The left navigation pane lists various project items, with 'Analytical Reports (2)' highlighted. The main area displays a table of report definitions.

Object Name	Description	Last Modified By	Last Modified On
P&L - Quarterly		admin	9/21/2016
Profit & Loss Budget Performance		admin	9/19/2016

Figure: Viewing analytical reports in the project

On the page, you can perform a variety of operations, as described in the following topics:

- [To Add a Custom Analytical Report to a Project](#)
- [To Delete a Custom Analytical Report from a Project](#)
- [To Update a Custom Analytical Report in a Project](#)
- [To Redirect to the Report Definitions Form](#)

To Add a Custom Analytical Report to a Project

You can add a custom analytical report to a customization project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Analytical Reports** in the navigation pane to open the Analytical Reports page.
3. On the page toolbar, click **Add New Record (+)**, as shown in the screenshot below.
4. In the list of custom analytical reports in the **Add Report Definition** dialog box, which opens, select the check box for each report that you want to include in the project.



The **Add Report Definition** dialog box displays all the custom analytical reports that exist in your instance of Acumatica ERP. You can select multiple custom analytical reports to add them to the project simultaneously.

5. In the dialog box, click **OK** to add each selected analytical report to the page table.
6. On the page toolbar, click **Save** to save the changes to the customization project.

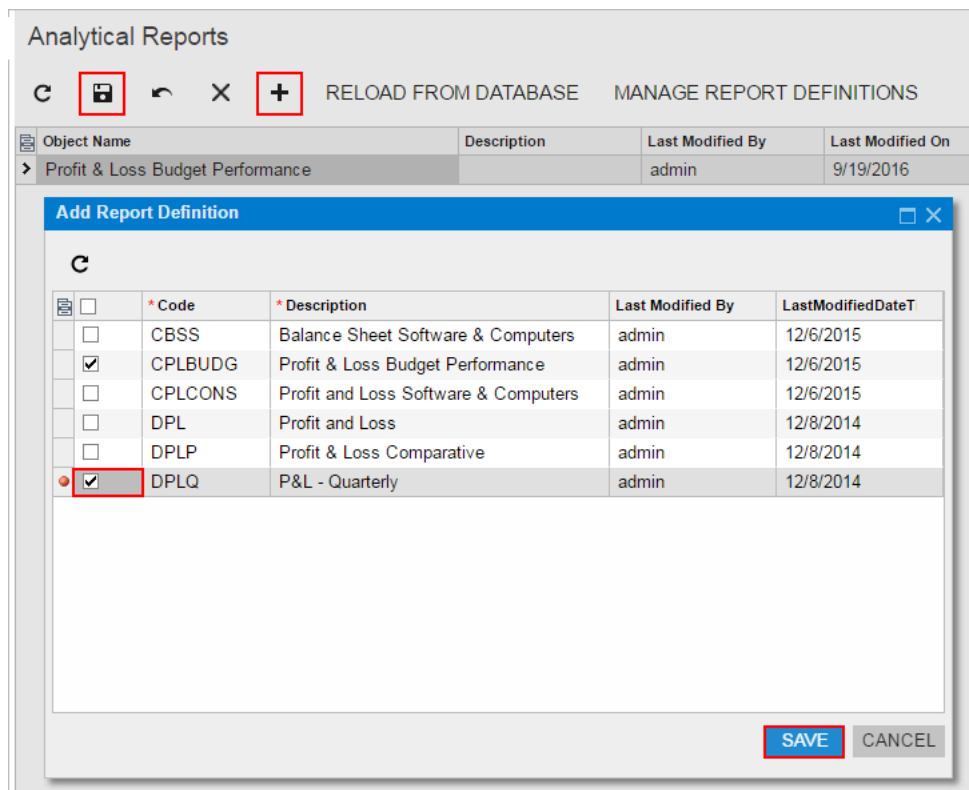


Figure: Adding the custom analytical report to the customization project

The system adds to the project the data from the database for each selected custom analytical report. You can view each new item in the Project Items table of the [Item XML Editor](#), as shown in the following screenshot.

The screenshot shows the 'Edit Project Items' interface. A table lists various project items, with 'P&L - Quarterly' selected. Below the table, the XML code for the selected item is displayed in the 'Source' pane.

Object Name	Type	Description	Exclud	Created By	Creation Date	Last Modified E
ja-JP	Locale		<input type="checkbox"/>	admin	9/15/2016	admin
~/pages/ar/ar303000.aspx	Page		<input type="checkbox"/>	admin	9/1/2016	admin
~/pages/so/so303000.aspx	Page		<input type="checkbox"/>	admin	9/1/2016	admin
AR650660.RPX	Report		<input type="checkbox"/>	admin	9/19/2016	admin
P&L - Quarterly	ReportDefinition		<input type="checkbox"/>	admin	9/21/2016	admin
Profit & Loss Budget Perf...	ReportDefinition		<input type="checkbox"/>	admin	9/19/2016	admin
RM000001	ScreenWithRights		<input type="checkbox"/>	admin	9/19/2016	admin
RM000002	ScreenWithRights		<input type="checkbox"/>	admin	9/20/2016	admin
RM000003	ScreenWithRights		<input type="checkbox"/>	admin	9/20/2016	admin

```

<ReportDefinition>
  <data-set>
    <relations format-version="3" relations-version="20160101" main-table="RMReport"
      <link from="RMColumn (ColumnSetCode)" to="RMColumnSet (ColumnSetCode)"
      <link from="RMColumn (NoteID)" to="Note (NoteID)" type="Note" />
      <link from="RMColumn (NoteID)" to="RMColumnKvExt (RecordID)" type="RowK
      <link from="RMColumnHeader (ColumnSetCode, ColumnCode)" to="RMColumn (C
      <link from="RMColumnHeader (NoteID)" to="Note (NoteID)" type="Note" />
      <link from="RMColumnHeader (NoteID)" to="RMColumnHeaderKvExt (RecordID)
      <link from="RMColumn (DataSourceID)" to="RMDataSource (DataSourceID)" t
      <link from="RMRow (DataSourceID)" to="RMDataSource (DataSourceID)" type
      <link from="RMUnit (DataSourceID)" to="RMDataSource (DataSourceID)" typ
      <link from="RMReport (DataSourceID)" to="RMDataSource (DataSourceID)" t
  
```

Figure: Viewing the XML code of the *ReportDefinition* item included in the project

A *ReportDefinition* item contains all the data required to recreate the corresponding analytical report in any instance of Acumatica ERP.

To Delete a Custom Analytical Report from a Project

To remove a *ReportDefinition* item from a project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Analytical Reports** in the navigation pane to open the Analytical Reports page.
3. In the page table, click the item to be deleted.
4. On the page toolbar, click **Delete Row (X)**.
5. On the page toolbar, click **Save** to save the changes to the customization project.

If you delete a *ReportDefinition* item from the project, the analytical report remains in the system unless you delete the report by using the [Report Definitions \(CS.20.60.00\)](#) form.

To Update a Custom Analytical Report in a Project

If you have used the [Report Definitions \(CS.20.60.00\)](#) form to change an analytical report included in a customization project, you have to update the appropriate item in the project. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Click **Analytical Reports** in the navigation pane to open the Analytical Reports page.
3. On the page toolbar, click **Reload From Database**.

The platform updates all the *ReportDefinition* items of the project by using the appropriate data from the database.

To Redirect to the Report Definitions Form

You might need to add an analytical report to Acumatica ERP during customization. To manage analytical reports in Acumatica ERP, you use the [Report Definitions](#) (CS.20.60.00) form.

To open this form from the [Customization Project Editor](#), perform the following actions:

1. Click **Analytical Reports** in the navigation pane to open the Analytical Reports page.
2. On the page toolbar, click **Manage Report Definitions**.


As soon as you add a new analytical report to the system or change an existing analytical report on the [Report Definitions](#) form, the report is saved in the database for the current company, and you can add the report to the customization project as a *ReportDefinition* item. For more information about analytical reports, see [Managing Analytical Reports](#) in the User Guide.

Customizing Elements of the User Interface

You can customize the user interface by creating a new form or by changing the content and layout of a form that already exists in Acumatica ERP.

This part is intended to describe how to use the visual [Layout Editor](#) to develop the ASPX code of a custom form and modify the ASPX code of an existing form.

The following table lists most of the types of ASPX objects supported in the Acumatica Customization Platform.

Object	Description
PXDataSource	A data source control that connects to a graph instance on the Acumatica ERP server, retrieves data from the graph instance, and sends data to the graph instance; it also provides the data processing for the control containers that are defined in the ASPX page. In Acumatica ERP, a page must contain a single <code>PXDataSource</code> control.
PXFormView	A data-bound UI container control that renders a single record from its associated data source.
PXGrid	A data-bound UI container control that renders a table with multiple records from its associated data source. The object can be displayed in form view mode for a single record and provides paging buttons that can be used to navigate between records. To define form view mode, the <code>PXGrid</code> control must include the <code>RowTemplate</code> element, which can contain controls for the record fields and layout rules for these controls.
PXGridColumn	In a <code>PXGrid</code> container, an object that defines the data field and properties of a column in the grid.
RowTemplate	In a <code>PXGrid</code> container, an object that defines a record field and layout for each control to be rendered in form view mode for the grid.
PXTab	A data-bound UI container control that renders tabs defined by child <code>PXTabItem</code> containers.
PXTabItem	In a <code>PXTab</code> container, a container control that renders a single record from the data source specified for the parent <code>PXTab</code> container.
PXSmartPanel	An UI container control that renders a dialog box.
PXLayoutRule	In a container with controls for a single data record, a component that defines a layout rule used to organize the controls within a row or column.
PXPanel	In a container with controls for a single data record, a container with a caption used to group controls. In a dialog box, it is often used as a container to display a horizontal row of buttons with right alignment.
PXGroupBox	In a container with controls for a single data record, a container with a caption used to group controls. It is designed to be used as a radio button container to render a drop-down field as a set of radio buttons. It contains scripts with logic to support a nested radio button for each value of a drop-down field.
PXRadioButton	In a <code>PXGroupBox</code> container, a radio button that is used for a single constant value of a drop-down field.  : In Acumatica ERP, a radio button can work properly only in a <code>PXGroupBox</code> container that is used for a drop-down data field.
PXLabel	In a container with controls for a single data record, an element to display text.
PXButton	In a container with controls for a single data record, an element to display a button control. In a dialog box, it is usually included in a <code>PXPanel</code> container.

Object	Description
PXJavaScript	In a container with controls for a single data record, a control to keep JavaScript code.
PXTextEdit	In a container with controls for a single data record, a text box to display and edit the value of a <code>string</code> field.
PXNumberEdit	In a container with controls for a single data record, a box to display and edit the value of a <code>decimal</code> or <code>int</code> field.
PXMaskEdit	In a container with controls for a single data record, a text box to display and edit the value of a <code>string</code> field that has the format specified in the data access class.
PXDateTimeEdit	In a container with controls for a single data record, a box to display and select the value of a <code>datetime</code> field.
PXCheckBox	In a container with controls for a single data record, a check box to display and select the value of a <code>bool</code> field.
PXDropDown	In a container with controls for a single data record, a combo box to display, edit, and select the value of a field with a list attribute, such as <code>PXStringList</code> , <code>PXIntList</code> , or <code>PXDecimalList</code> .
PXSelector	In a container with controls for a single data record, a lookup control to display, search, and select the value of a field with the <code>PXSelector</code> attribute.
PXSegmentMask	In a container with controls for a single data record, a lookup control with a specified segmented key value that identifies a data record and consists of one segment or multiple segments, with a list of possible values defined for each segment.
PXTreeSelector	In a container with controls for a single data record, a lookup control to select a value for a field with a <code>PXTreeSelector</code> attribute from a tree control.

When you use the Layout Editor to add a UI element to another UI element on a form, you should understand the rules that are used for nesting objects in ASPX code for Acumatica ERP forms. The following diagram shows which ASPX objects can be included in other ASPX objects.

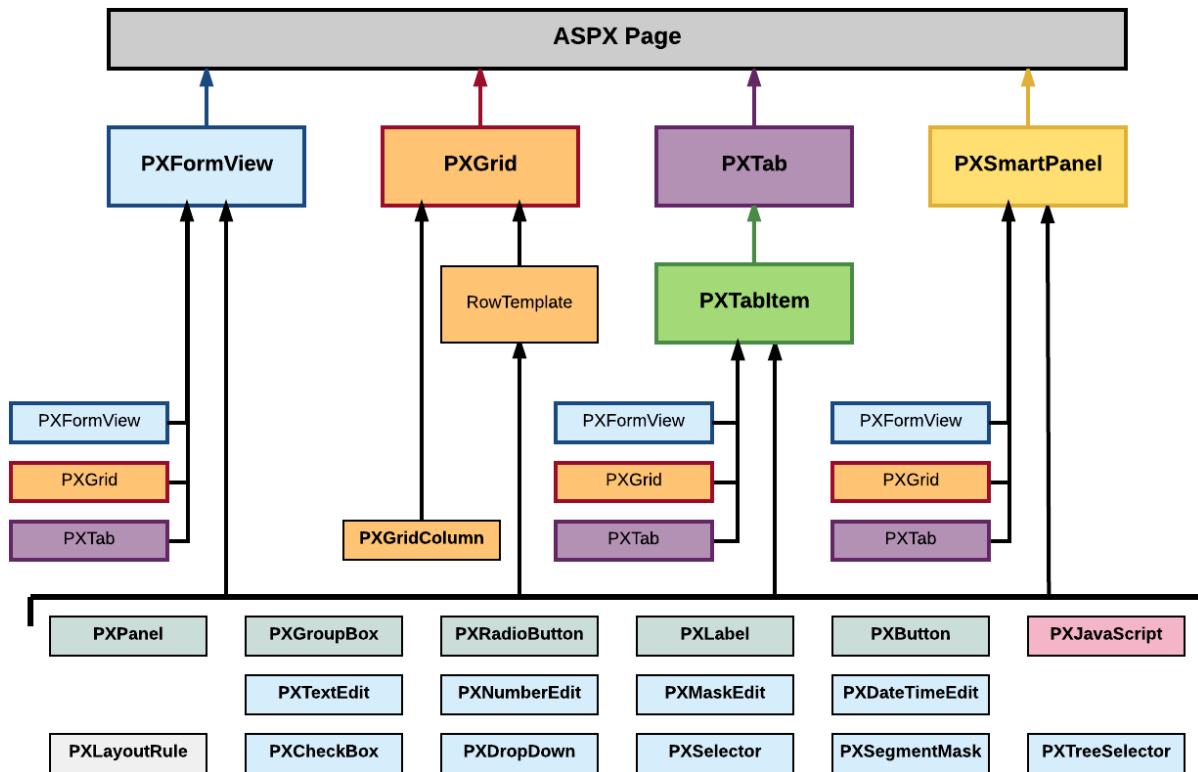


Figure: Nesting rules for elements of an ASPX page

In the diagram, if an arrow goes from object A to object B, it means that multiple instances of object A can be included in a single object B. For example, `PXTab` can contain multiple `PXTabItem` objects.



- The `PXGrid` object can contain a single `RowTemplate` object.
- The `PXSmartPanel` object is used to describe the content of a dialog box.
- A control for a data field (also referred as *box*)—such as `PXTextEdit`, `PXSelector`, and `PXCheckBox`—can be included in ASPX objects of the following types:
 - `PXFormView`
 - `RowTemplate`
 - `PXTabItem`
 - `PXSmartPanel`

In This Part

- *Custom Form*
- *Existing Form*
- *Form Container (PXFormView)*
- *Grid Container (PXGrid)*
- *Tab Container (PXTab)*
- *Tab Item Container (PXTabItem)*
- *Dialog Box (PXSmartPanel)*
- *Box (Control for a Data Field)*
- *Layout Rule (PXLayoutRule)*

- [Panel \(PXPanel\)](#)
- [Group Box \(PXGroupBox\)](#)
- [Label \(PXLabel\)](#)
- [Radio Button \(PXRadioButton\)](#)
- [Button \(PXButton\)](#)
- [Java Script \(PXJavaScript\)](#)
- [Toolbars, Action Buttons, and Menus](#)
- [Other Control Types](#)

Custom Form

By using the Acumatica Customization Platform, you can develop a custom form from scratch and add it to a customization project. To create a new form, you have to create the following types of code:

- ASPX page that contains a description of the UI elements of the form
- CS code that provides the business logic for the form

For each form that works with data from the database, the instance of Acumatica ERP must contain at least the following objects (see the diagram below):

- An ASPX page: The page must contain at least the data source control and a container with controls for data fields.
- A business logic controller (BLC, also referred to as *graph*): The graph must be specified in the `TypeName` property of the data source control of the page. The graph must contain at least one data view, which is specified in the `PrimaryView` property of the data source control and in the `DataMember` property of the container. The graph instance is created on each round trip and initializes the creation of the data view instance based on a BQL statement. The data view provides data manipulation and data flows between the data source control of the ASPX page, the cache object of the graph, and the corresponding table of the database. The BQL statement contains a reference to at least one data access class that is required to map the database table to data records in the cache object.
- A data access class (DAC): On each round trip, the DAC instance is created in the cache object when the data view processes any operation with the corresponding data.
- A table in the database: The table is mapped to the data access class that defines the data record type in the cache object of the graph instance.

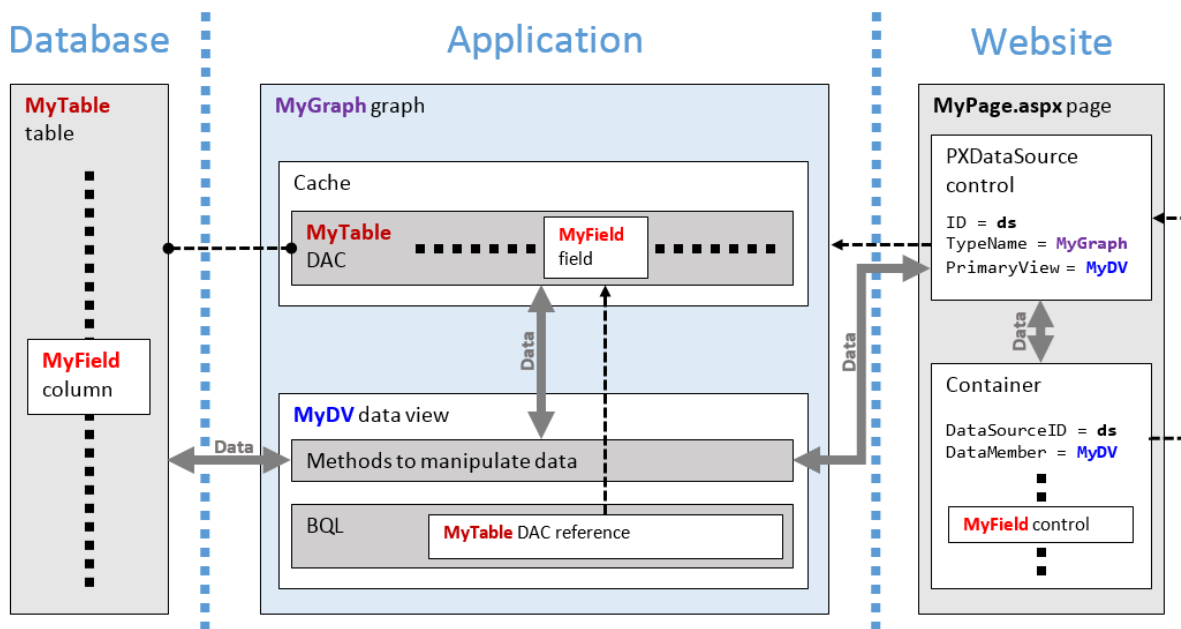


Figure: Objects required for a form that works with data from a database table

For a custom form to be enabled in an instance of Acumatica ERP, the site map of the instance must contain information about the form.

You use the [Customization Project Editor](#) to perform operations with custom forms, as described in the following topics:

- [To Develop a Custom Form](#)
- [To Create a Custom Form Template](#)
- [To Delete a Custom Form from a Project](#)

To Develop a Custom Form

To create and develop a custom form within a customization project, you can use the following approach:

1. Plan the functionality, content, and user interface of the new form.
2. If the custom form requires data from a new table, create the table in the database by using a database administration tool, such as SQL Server Management Studio. (See [To Add a Custom Table to a Project](#) for details.)
3. Create a workable form template by using the New Screen wizard, as described in [To Create a Custom Form Template](#).

After you have created a custom form template and published the customization project, you can develop the form in the same way as you customize an existing form of Acumatica ERP. (See [Existing Form](#) for details.) Because the code templates are added to the `App_RuntimeCode` folder of the Acumatica ERP website, you can develop the code in Microsoft Visual Studio.

4. **C#:** Create data access classes that contain the data field declarations required for the form controls. (See [To Create a Custom Data Access Class](#) for details.)
5. **C#:** In the graph template, define BQL statements in data views to manage data fields declared in data access classes. (See [To Add a New Member](#) for details.)
6. **ASPX:** If needed, add nested containers to the main containers of the form template. (See [To Add a Nested Container](#) for details.)
7. **C#:** For each nested container, in the graph template, define the appropriate data views.
8. **ASPX:** For each added container, specify the name of an appropriate data view in the `DataMember` property. (See [To Set a Container Property](#) for details.)

9. **ASPX:** If required, for each added container, specify other properties. (See [To Set a Container Property](#) for details.)
10. **ASPX:** Add controls for data fields to each container. (See [To Add a Box for a Data Field](#) for details.)
11. **ASPX:** Specify properties for controls. (See [To Set a Box Property](#) for details.)
12. **C#:** Develop business logic for the form in the graph (See [Customizing Business Logic](#) for details.)
13. **ASPX:** If required, add dialog boxes, as described in [To Add a Dialog Box](#).

We recommend that you use the [Layout Editor](#) to create the content of an ASPX page and Visual Studio to develop the business logic for a page. (See [Integrating the Project Editor with Microsoft Visual Studio](#) for details.)

To Create a Custom Form Template

To create a workable template for a custom form by using the New Screen wizard and include the template in a customization project, perform the following actions:

1. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
2. Select the **Screens** node in the navigation pane of the editor to open the Customized Screens page.
3. On the page toolbar, click **Add Screen > Create New Screen**, as item a in the screenshot below shows.
4. In the **Create New Screen** dialog box, which opens, specify values for input controls as follows (item b in the screenshot):
 - a. **Screen ID:** Enter an ID of a custom form in the `XX.**.**.**` format, which consists of the following parts:
 - a. Two-letter code of the module in Acumatica ERP

 : You can use a single unique two-letter code for all custom forms in a solution. Please check the `Pages` folder of the website in the production environment to ensure that the code you want to use is not already used in the instance of Acumatica ERP.
 - b. Two-digit code indicating the form type:
 - a. 10: Setup form
 - b. 20: Maintenance form
 - c. 30: Data entry form
 - d. 40: Inquiry form
 - e. 50: Processing form
 - f. 60: Report form
 - c. Two-digit code indicating the form sequential number
 - d. Two-digit code indicating the subform sequential number
 - b. **Graph Name:** Enter a unique name for the new graph.
 - c. **Graph Namespace:** By default, the New Screen wizard specifies the customization project name as the namespace ID. If you want to change the default ID, enter the ID of an existing or a new namespace for the new graph.
 - d. **Page Title:** Enter the title to be used as the form title in Acumatica ERP.
 - e. **Template:** Select one of the following ASPX page templates for the custom form.

Template	Description
<i>Form (FormView)</i>	A record-editing page with one <code>PXFormView</code> container
<i>Grid (GridView)</i>	A record-editing page with one <code>PXGrid</code> container
<i>Tab (TabView)</i>	A record-editing page with one <code>PXTab</code> container
<i>FormTab</i>	A record-editing page with <code>PXFormView</code> and <code>PXTab</code> containers
<i>FormGrid (FormDetail)</i>	A master-detail editing page with <code>PXFormView</code> and <code>PXGrid</code> containers
<i>TabGrid (TabDetail)</i>	A master-detail page with <code>PXTab</code> and <code>PXGrid</code> containers

- f. **Site Map Parent:** Select a parent site map node to place the form in this location in the site map of Acumatica ERP.
5. Click **OK** (item c) to create the new form.

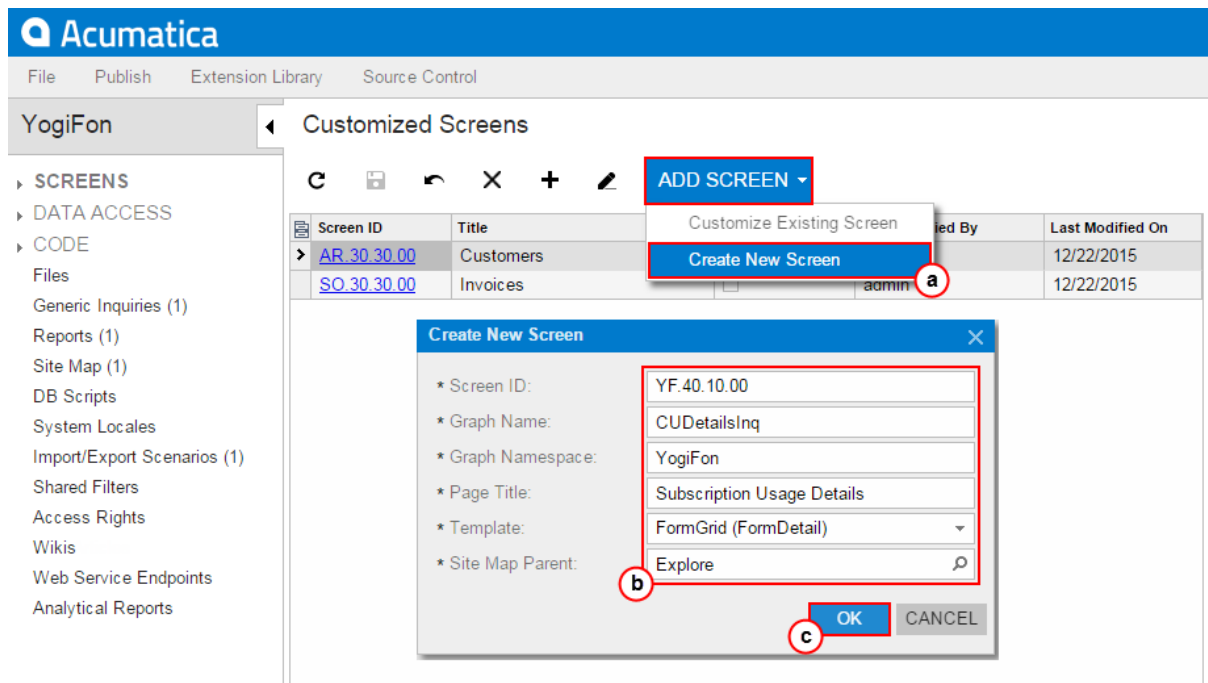


Figure: Using the New Screen wizard to create the new form

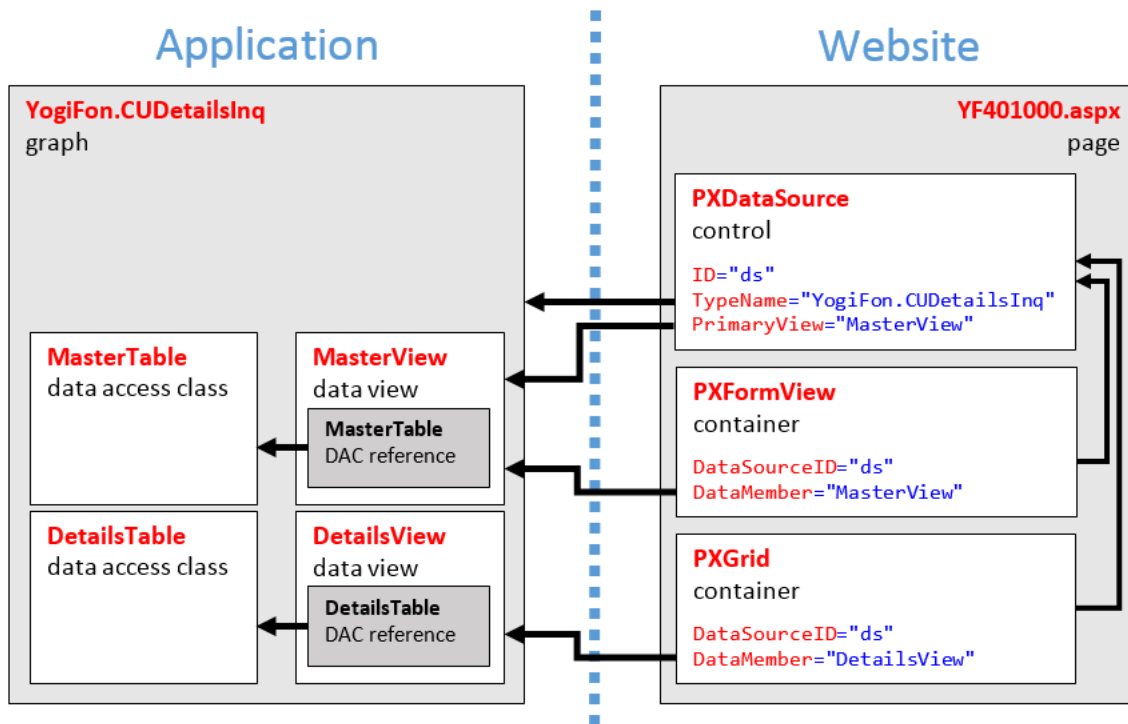
The wizard creates the form template and adds the following items to the customization project.

Item	Description
<i>File</i>	Contains the <code>Pages\XX\XX*****.aspx</code> file with the ASPX code template that has been selected for the new form. The name of the file corresponds to the value that is entered in the Screen ID box. The file is located in the <code>Pages\XX</code> folder of the Acumatica ERP website, where <code>XX</code> is the two-letter code of the screen ID.
<i>File</i>	Contains the <code>Pages\XX\XX*****.aspx.cs</code> file with the C# code for the ASPX page.
<i>Code</i>	Contains the code template of the business logic controller (BLC, also referred to as <i>graph</i>) for the new form. This item is saved in the database. When you publish the project, the

Item	Description
	platform creates a copy of the code in the file with the same name in the <code>App_RuntimeCode</code> folder of the Acumatica ERP website. You can develop the code in Microsoft Visual Studio.
Page	Contains the link to the new page content, which you can later develop by using the Layout Editor .
SiteMapNode	Contains the site map object of the new form.

For example, if you enter in the **Create New Screen** dialog box the values that are displayed in the screenshot above, the wizard creates code templates for both the `CUDetailsInq` business logic controller and the `YF401000` page, which contain all the components required for the form template to work properly (see the following diagram).

Figure: Analyzing the content of the new form template



You can publish the customization project to ensure that it is valid and the custom form can be opened in the browser. At the moment, the form does not contain a control for a field, as the following screenshot shows.

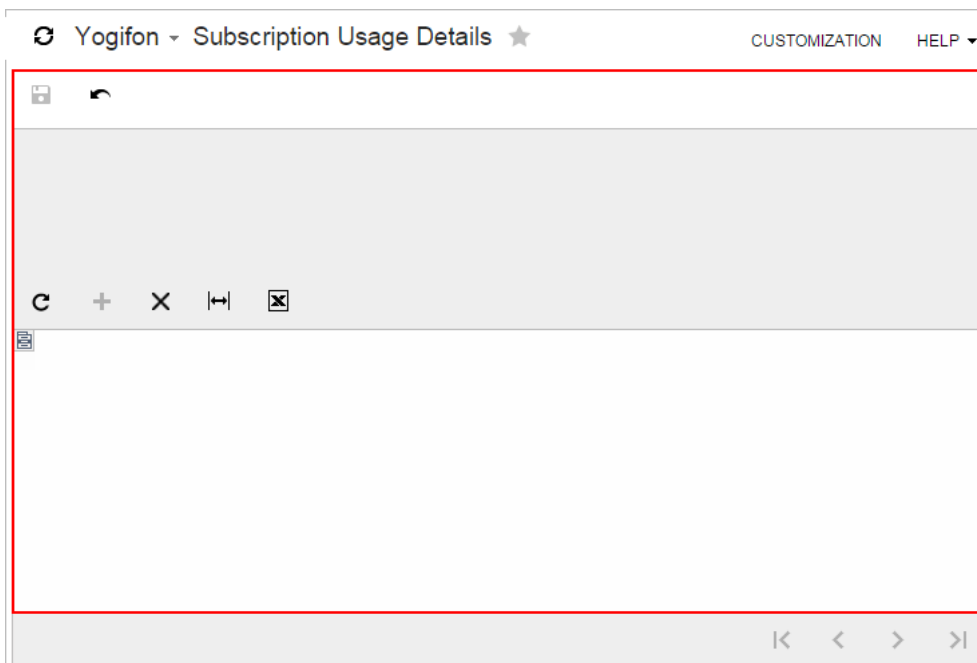


Figure: Viewing the new form

To Delete a Custom Form from a Project

To remove a custom form from a customization project, you have to delete all the items that have been added to the project for the form.

To do this, perform the following actions:

- Delete from the customization project the *Page* item that was added by the New Screen wizard. (See [To Delete a Page Item from a Project](#) for details.)
- Delete from the project the *Code* item that was added by the New Screen wizard. (See [To Delete a Code Item From a Project.](#))
- Delete from the project the `<FormID>.aspx` and `<FormID>.aspx.cs` *File* items that were added by the New Screen wizard. (For more information, see [To Delete a Custom File From a Project.](#))
- Delete from the project the *SiteMapNode* item that was added by the New Screen wizard. (See [To Delete a Site Map Item from a Project](#) for details.)
- If you added other items for the custom form, such as items for the mobile site map or custom files, delete these items.

To delete multiple items from the customization project successively on a single page, you can use the Edit Project Items page of the Customization Project Editor. (See [To Delete Items from the Project on the Edit Project Items Page](#) for details.)

The system applies the changes to the file system as soon as you publish the customization project.

Existing Form

You can customize the user interface of an existing form of Acumatica ERP by using the [Layout Editor](#). The editor is a visual tool that you can use to perform the following changes in the ASPX code for a form:

- Add or delete a container.
- Add or delete a button or radio button. (See [Button \(PXButton\)](#) and [Radio Button \(PXRadioButton\)](#) for details.)
- Add or delete a layout rule. (See [Layout Rule \(PXLayoutRule\)](#) for details.)

- Add or delete a Java script. (See [Java Script \(PXJavaScript\)](#) for details.)
- Create or delete a control for a data field. (See [Box \(Control for a Data Field\)](#) for details.)
- Change the properties of an element in the ASPX code. (See [To Set a Container Property](#), [To Set a Box Property](#) and [To Set a Layout Rule Property](#) for details.)

In Acumatica ERP, an ASPX page must contain a single `PXDataSource` control. You can add the following types of containers immediately to the level of an ASPX page, where the `PXDataSource` control is defined:

- [Form Container \(PXFormView\)](#)
- [Grid Container \(PXGrid\)](#)
- [Tab Container \(PXTab\)](#)
- [Dialog Box \(PXSmartPanel\)](#)

The order of containers in the ASPX page defines the order of appropriate areas with controls on the form.

Detailed instructions on the customization of an existing form are provided in the following topics:

- [To Start a Customization of a Form](#)
- [To Delete a Customization of a Form](#)
- [To Add a Form Container](#)
- [To Add a Tab Container](#)
- [To Add a Grid Container](#)
- [To Add a Dialog Box](#)
- [To Delete a Container](#)

The changeset of a form is stored in the database as a *Page* item of a customization project. During the publication of the project, the Acumatica Customization Platform applies the changeset to the form to create a customized version of the `.aspx` file with the same name in the `pages_xx` subfolder of the `CstPublished` folder of the website.

For example, if you customize the [Customers](#) (AR.30.30.00) form and publish the project, the platform creates the `ar303000.aspx` and `ar303000.aspx.cs` files in the `\CstPublished\pages_ar` folder. If you delete these files, Acumatica ERP uses the original files from the `\Pages\AR` folder to display this form. However if you again publish the customization project, the platform recreates these files, and Acumatica ERP uses the customized version of the form.

We recommend that you use the [Layout Editor](#) to customize the content of an ASPX page, the [Data Class Editor](#) to modify data access classes, and MS Visual Studio to extend the business logic for a page. (See [Integrating the Project Editor with Microsoft Visual Studio](#) for details.)

To Start a Customization of a Form

To start the customization of a form, you have to open the form in the [Layout Editor](#). To do this, you commonly perform the following operations:

1. Open the form in the browser.
2. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).
3. On the form, click the UI element (or area) to be customized; this opens the [Element Properties Dialog Box](#) for the element (or area).
4. In the dialog box, click **Customize**.
5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one. (See [To Select an Existing Project](#) and [To Create a New Project](#) for details.)

If the customization project does not contain a changeset for the form, the [Customization Project Editor](#) adds a *Page* item for the form to the project, to keep the changeset to the ASPX code of the form in the database. The form is opened in the [Layout Editor](#), and you can start customization of the form.

When you click **Save** on the editor toolbar, the editor updates the *Page* item in the database.

To Delete a Customization of a Form

If a customization project contains changes to multiple objects of Acumatica ERP, and you need to remove customization for only a single form, perform the following operations:

1. Remove from the project the *Page* item for the form. (See [To Delete a Page Item from a Project](#) for details.)
2. Publish the project, as described in [To Publish the Current Project](#).

To Add a Form Container

You can add a new `PXFormView` container to a form of Acumatica ERP. To do this, perform the following actions:

1. Open the form in the [Layout Editor](#), as described in [To Add a Page Item for an Existing Form](#).
2. In the editor, click the **Add Controls** tab item.
3. From the **Main Containers** group, drag the **Form** container to the required place in the Control Tree, as shown in the following screenshot.

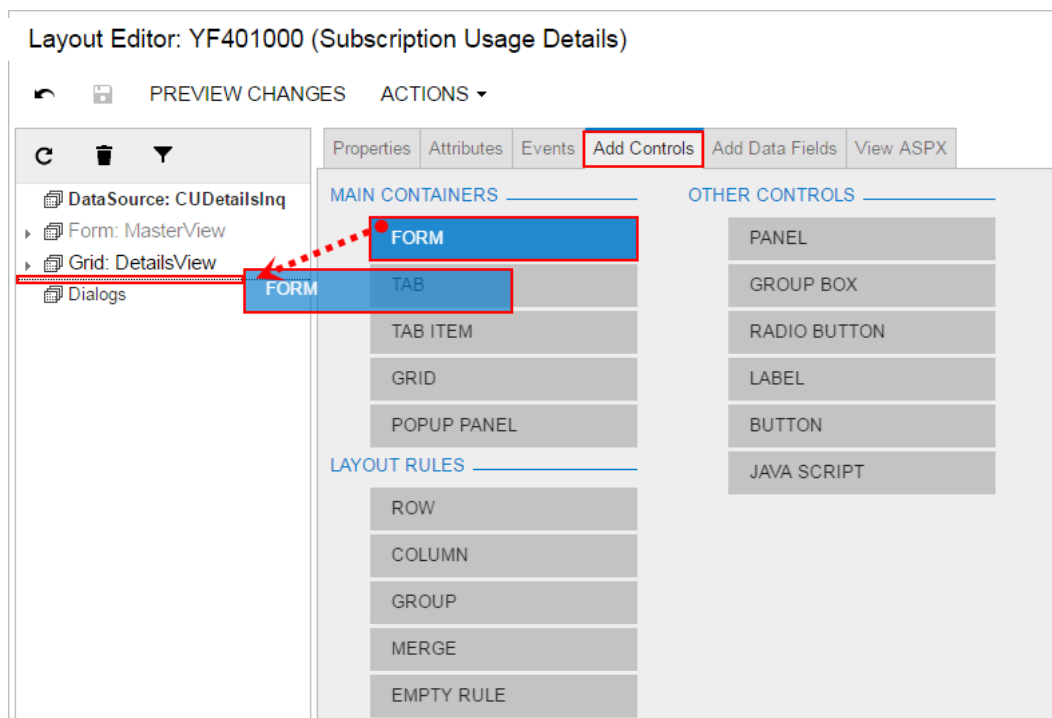


Figure: Adding a form container to a form



: The area of a form container is visible on a customized form only if it contains at least one visible control.

4. In the Control Tree, select the form container that has been added, and specify the item properties, as described in [To Set a Container Property](#).
5. Click **Save** on the toolbar of the Layout Editor to save your changes to the customization project.

For more information about `PXFormView`, see [Form Container \(PXFormView\)](#).

To Add a Grid Container

You can add a new `PXGrid` container to an existing form of Acumatica ERP. To do this, perform the following actions:

1. Open the form in the *Layout Editor*, as described in *To Add a Page Item for an Existing Form*.
2. In the editor, click the **Add Controls** tab item.
3. From the **Main Containers** group, drag the **Grid** container to the required place in the Control Tree, as shown in the following screenshot.

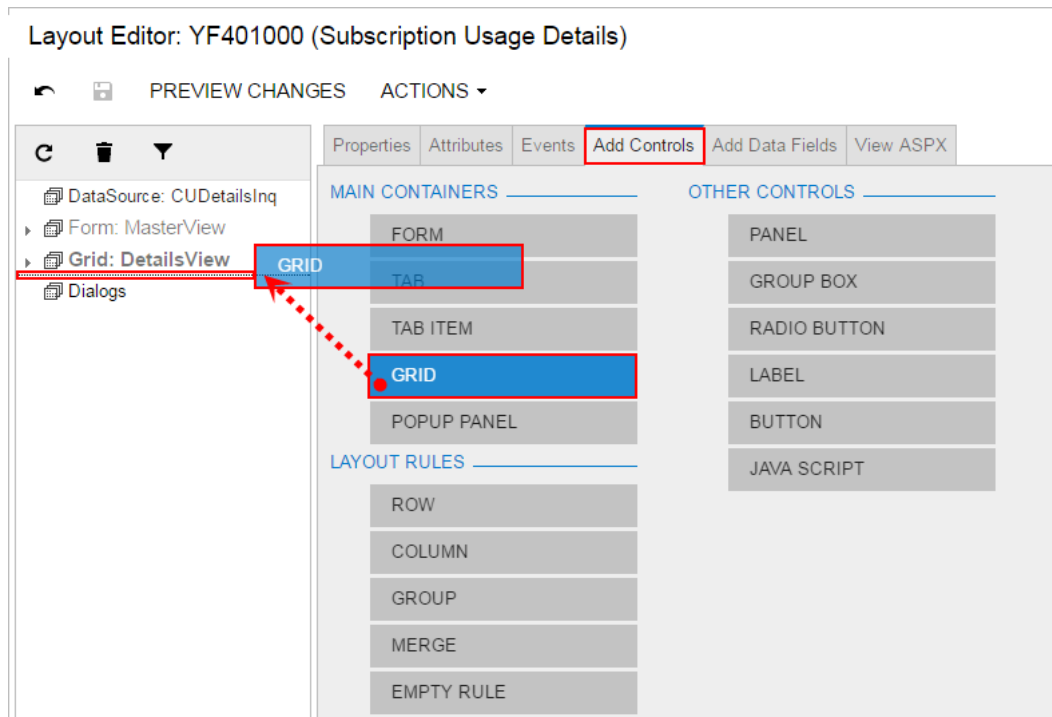


Figure: Adding a grid container to the tab



: A grid is visible on a customized form only if it contains at least one column for a field.

4. In the Control Tree, select the grid that has been added, and specify the item properties, as described in *To Set a Container Property*.
5. Click **Save** on the toolbar of the Layout Editor to save your changes to the customization project.

For more information about `PXTab`, see *Tab Container (PXTab)*.

To Add a Tab Container

You can add a new `PXTab` container to an existing form of Acumatica ERP. To do this, perform the following actions:

1. Open the form in the *Layout Editor*, as described in *To Add a Page Item for an Existing Form*.
2. In the editor, click the **Add Controls** tab item.
3. From the **Main Containers** group, drag the **Tab** container to the required place in the Control Tree, as shown in the following screenshot.

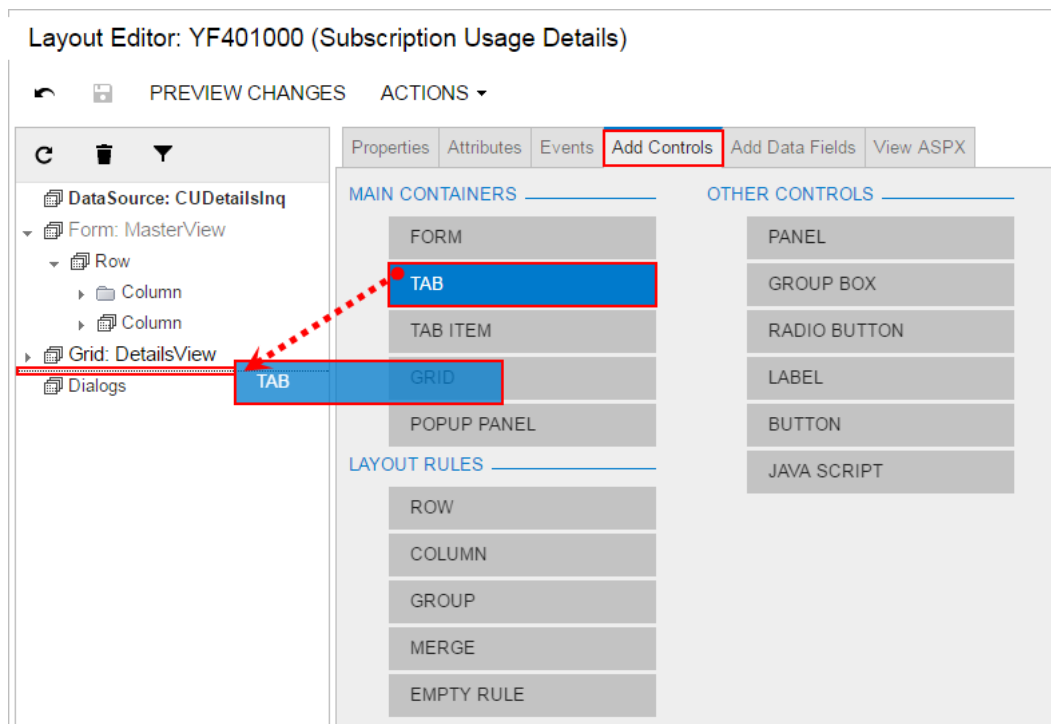


Figure: Adding a tab container to the tab

The `PXTab` container cannot exist without nested `PXTabItem` containers. Therefore, when you add a `PXTab` container, the Layout Editor creates a nested `PXTabItem` container.



: A tab container is visible on a customized form only if there is at least one control for a field in a nested `PXTabItem` container.

4. In the Control Tree, select the tab that has been added, and specify the item properties, as described in [To Set a Container Property](#).
5. Click **Save** on the toolbar of the Layout Editor to save your changes to the customization project.

For more information about `PXTab`, see [Tab Container \(PXTab\)](#).

To Add a New Tab Item to a Tab

You can add a new `PXTabItem` container to a tab on an existing form of Acumatica ERP. To do this, perform the following actions:

1. Open the form in the [Layout Editor](#), as described in [To Add a Page Item for an Existing Form](#).
2. In the editor, click the **Add Controls** tab item.
3. From the **Main Containers** group of the tab item, drag the **Tab Item** container above the customized tab, as shown in the following screenshot.

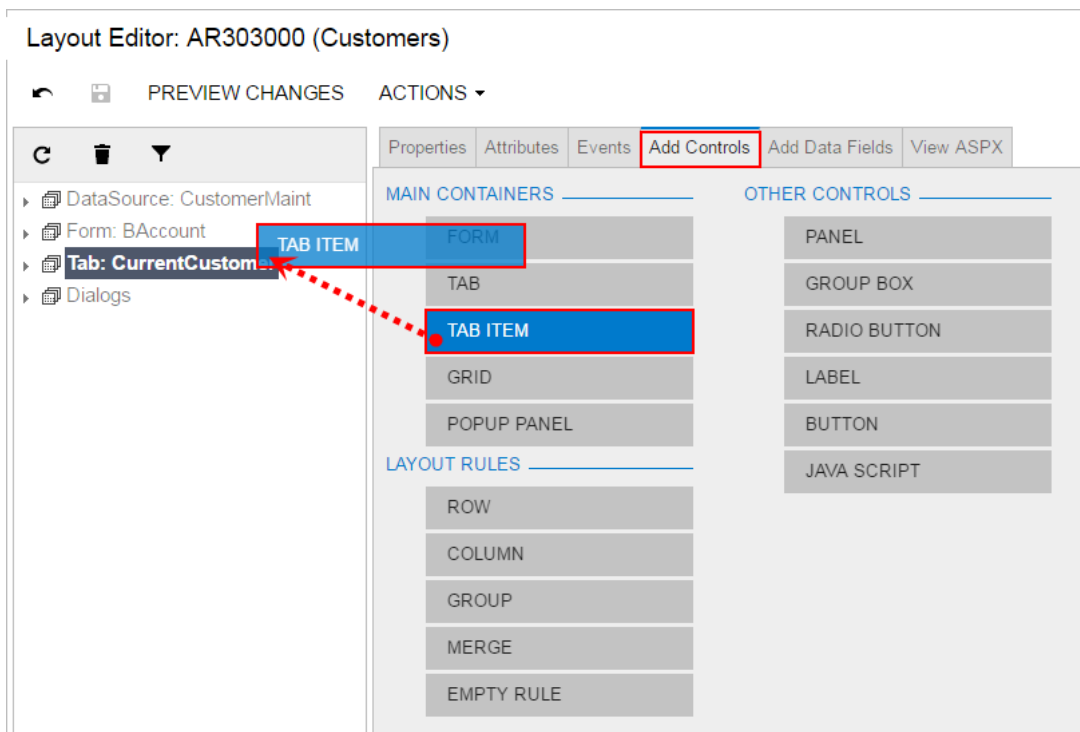


Figure: Adding a tab item to the tab

If the tab contains multiple tab items and you need to add a tab item to a certain position in the tab, you should expand the tab node in the Control Tree to see the existing tab items and drag the **Tab Item** container to the required position, as the following screenshot shows.

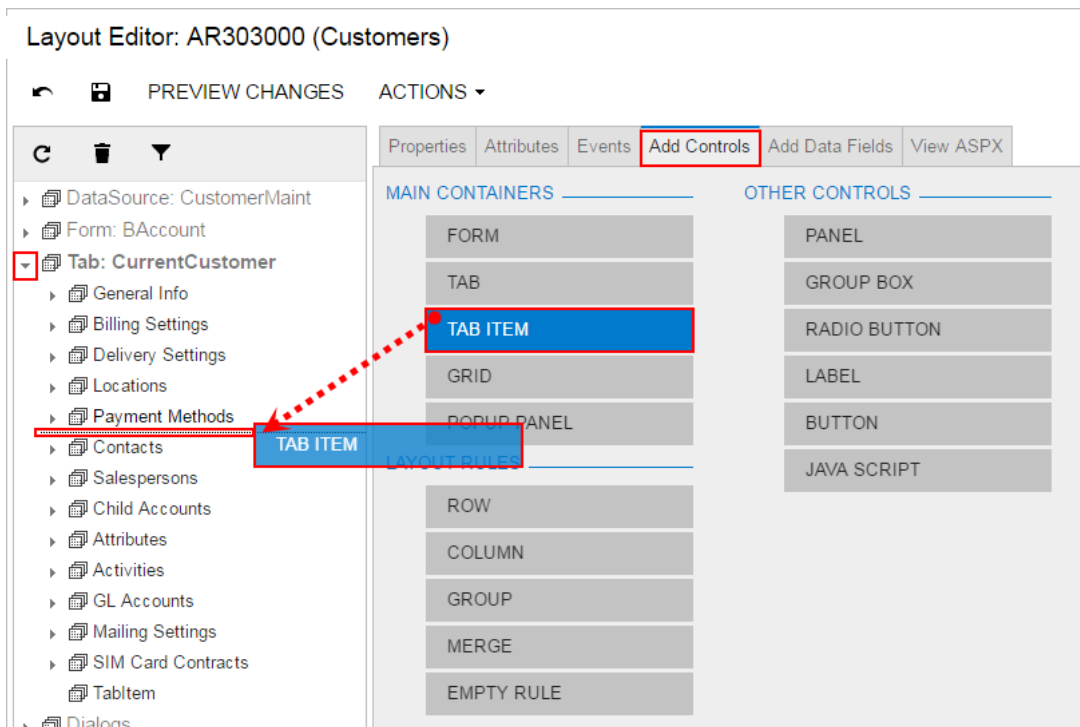


Figure: Adding a tab item to a certain position in the tab



: A tab item is visible if it contains at least one control for a field.

4. In the Control Tree, select the tab item that has been added, and specify the item properties, as described in [To Set a Container Property](#).
5. Click **Save** on the toolbar of the Layout Editor to save your changes to the customization project.

To Add a Dialog Box

You can add a custom dialog box to an Acumatica ERP form. If the dialog box must contain controls for data fields, you should generally perform the following operations:

- Add a new `PXSmartPanel` container to the ASPX page, as described in this topic.
- If needed, define a custom data access class with the declaration of the data fields to be used for controls in the dialog box. See [To Create a Custom Data Access Class](#) for details.
- If needed, in the extension for the graph that is specified in the `TypeName` property of the `PXDataSource` control of the ASPX page, add business logic for the dialog box. (See [To Customize an Existing Business Logic Controller](#) for details.) For example, in the graph extension, you can do the following:
 - Declare a new data view that provides data for the dialog box controls, as described in [To Add a New Member](#)
 - Add an action (with a button on the form toolbar) to open the dialog box
 - Add other business logic for the dialog box
- If the smart panel container must include a box for a data field, add a data-bound container, such as `PXFormView`. Then bind the new container to the data view declared in the graph or graph extension (see [Using the DataMember Property](#) for details) whose BQL statement refers to the data access class with the field declaration.
- If the smart panel must contain a row of buttons, add a nested `PXPanel` container with the `SkinID` property set to `Buttons` (see [Using the SkinID Property](#) for details), and add the buttons to the nested panel, as described in [To Add Another Supported Control](#).

To add a new `PXSmartPanel` container to an ASPX page, perform the following actions:

1. Open the form in the [Layout Editor](#), as described in [To Add a Page Item for an Existing Form](#).
2. In the editor, click the **Add Controls** tab item.
3. From the **Main Containers** group of the tab item, drag the **POPUP PANEL** container into the **Dialogs** node in the Control Tree, as shown in the following screenshot.

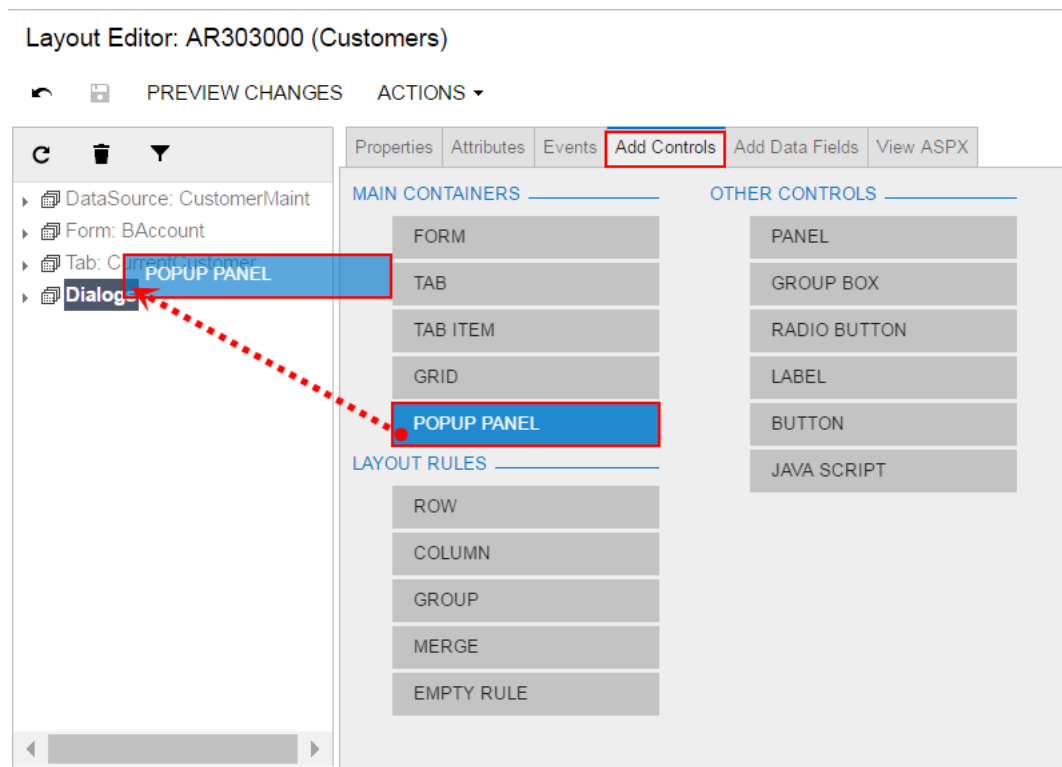


Figure: Adding a form container to a form



: A dialog box can be displayed if it contains at least one visible control.

4. In the Control Tree, select the new container that has been added.
5. Specify the item properties, as described in [To Set a Container Property](#).
6. Click **Save** on the toolbar of the Layout Editor to save your changes to the customization project.

To Delete a Container

You can delete a container from an Acumatica ERP form. To do this, perform the following actions:

1. Open the form in the [Layout Editor](#), as described in [To Add a Page Item for an Existing Form](#).
2. In the Control Tree of the editor, select the container to be deleted.
3. On the toolbar of the Control Tree, click **Delete**, as the following screenshot shows.

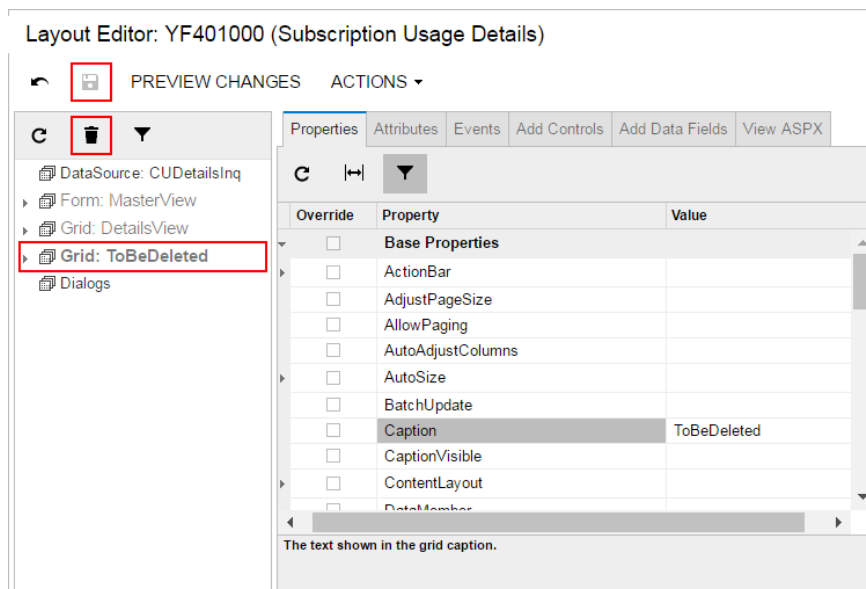


Figure: Deleting a container



Warning: We recommend that you not remove the `DataSource` control.

4. Click **Save** on the toolbar of the Layout Editor to save your changes to the customization project.

Form Container (PXFormView)

`PXFormView` is a data-bound UI container control that renders a single record from its associated data source.

An ASPX page can contain `PXFormView` as a main container. A form container, as the diagram below shows, can be also included in the following types of containers:

- `PXFormView`
- `PXTabItem`
- `PXSmartPanel`

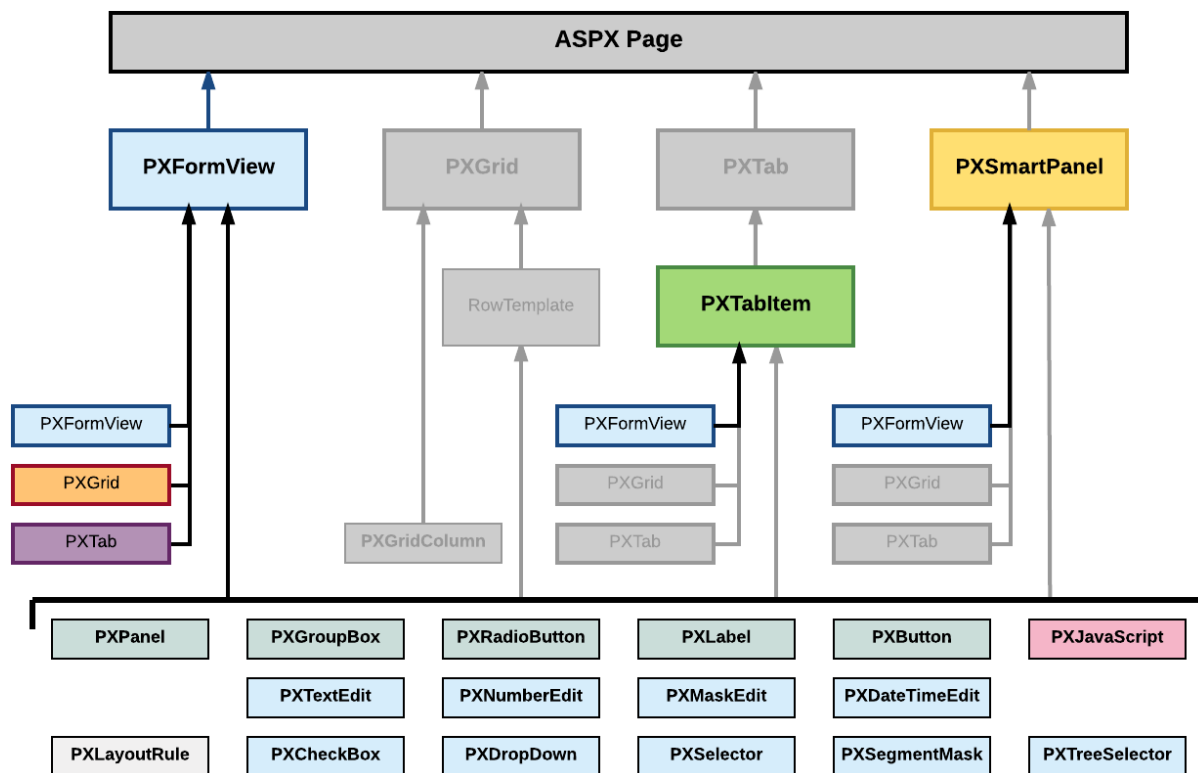


Figure: Nesting rules for a PXFormView container in an ASPX page

A form container can include multiple ASPX objects of the following types:

- A data-bound UI container control: PXFormView, PXGrid, and PXTab
- A layout rule: PXLayoutRule
- A box for a data field: PXTextEdit, PXNumberEdit, PXMaskEdit, PXDateTimeEdit, PXCheckBox, PXDropDown, PXSelector, PXSegmentMask, and PXTreeSelector
- Another control: PXPanel, PXGroupBox, PXRadioButton, PXLabel, PXButton, and PXJavaScript

A box for a data field can be added to a form container if the container is bound to a data view declared within the graph that provides business logic for the ASPX page. If you want to bind a form container to a data view, you must specify the properties as follows for the appropriate PXFormView object:

- The `DataSourceID` property value must be equal to the value of the `ID` property of the `PXDataSource` control.
- The `DataMember` property must contain the name of the data view that is declared in the graph and provides data for the controls of the form container.

To create a new form container in an ASPX page, follow the instructions described in [To Add a Form Container](#).

To delete a form container from an ASPX page, follow the instructions described in [To Delete a Container](#).

For detailed information on working with the content of a form container, see the following topics:

- [To Open a Container in the Layout Editor](#)
- [To Set a Container Property](#)
- [To Add a Nested Container](#)
- [To Add a Box for a Data Field](#)
- [To Add a Layout Rule](#)
- [To Add Another Supported Control](#)

- [To Reorder Child UI Elements](#)
- [To Delete a Child UI Element](#)

To Open a Container in the Layout Editor

You perform the customization of a container by using the [Layout Editor](#). To open a container of an Acumatica ERP form in the editor, perform the following actions:

1. Open the form in the browser.
2. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).
3. On the form, click the area of the container to be customized, which opens the [Element Properties Dialog Box](#) for the container.
4. In the dialog box, click **Customize**.
5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one. (See [To Create a New Project](#) and [To Select an Existing Project](#) for details.)

If the customization project does not contain a changeset for the form, the [Customization Project Editor](#) adds to the project a *Page* item for the form to keep the changeset to the ASPX code of the form in the database. The container is opened in the [Layout Editor](#), and you can start the customization of the container.

When you click **Save** on the editor toolbar, the editor updates the *Page* item in the database.

To Set a Container Property

To include in a customization project changes to the properties of a container, you have to modify the properties by using the [Layout Editor](#). To start setting the properties of a container, perform the following actions:

1. Open the container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. Ensure that the container node is selected in the Control Tree of the editor. Click the arrow left of the node to expand the node if needed.
3. Click the **Properties** tab item to open the list of properties for the container (see the screenshot below).
4. Specify values for the required properties.
5. Click **Save** to save your changes to the customization project.

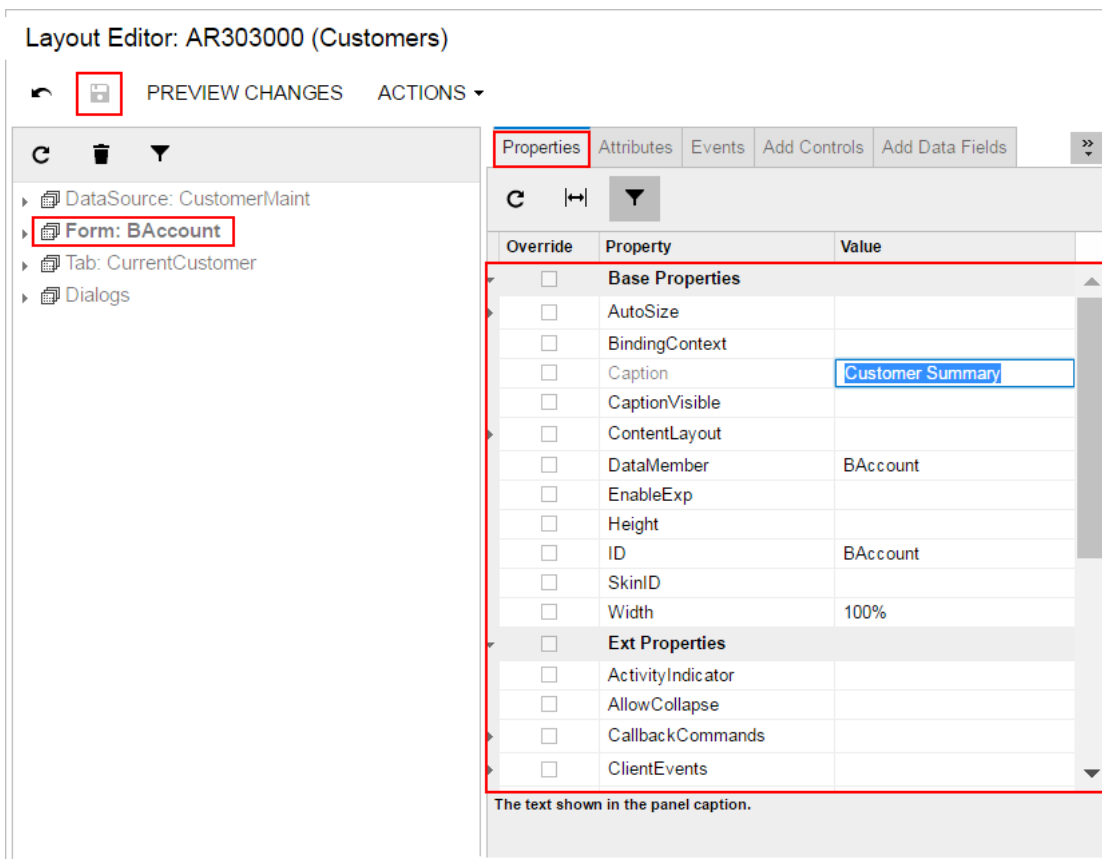


Figure: Setting a property of a container

For detailed information about the `DataMember` property, see [Using the DataMember Property](#).

In Acumatica ERP, for the `PXFormView`, `PXGrid`, and `PXPanel` containers, there are predefined values for the `SkinID` property. See [Using the SkinID Property](#) for details.

If you plan to use a container in the mobile site map, specify the `Caption` property of the container, as described in [Using the Caption Property in the Mobile Site Map](#).

Using the DataMember Property

If you need to find out which data view provides data for a control container on a form, perform a search to find the `DataMember` string in the appropriate ASPX code. The `DataMember` property is used to bind a control container of a form to a data view defined in the business logic controller (BLC, also referred as *graph*) of the form. The property value is the name of the data view. Each `DataMember` property value can correspond to any data view name of the BLC. Any data view except for the main data view can be used by an unlimited number of containers. The main data view must be bound to a single container.

A container can contain a box for a data field if it is bound to a data view declared within a BLC for the following reasons:

- A data field is declared in a data access class (DAC).
- An instance of the DAC record can exist in the cache of a BLC that contains the declaration of a data view with the DAC reference in the BQL statement.
- Each time a data record is selected in the container, the container creates a callback to the `PXDataSource` control that is specified in the `DataSourceID` property of the container. The data source control creates a remote procedure call to the application server to execute the `Display` operation on the data view that is specified as the `DataMember` for the container. The data view checks the existence of the record in the cache and, if the check fails, executes the BQL request and stores the obtained record in the cache.

- The data view provides all data exchange operations with the database, cache, and `PXDataSource` control.

In Acumatica Framework, `DataMember` is used to specify a data view for the following container types:

- `PXFormView`
- `PXGrid`
- `PXTab`
- `PXTreeView`



The `PXTreeView` container is not supported by tools of the Acumatica Customization Platform.

By default, a nested container inherits the `DataMember` property from the parent container. If a nested container is `PXFormView`, `PXGrid`, or `PXTab`, it can be bound to another data view.

If the `DataMember` property is available for other ASPX objects, it has a special purpose. For example, you can specify the `DataMember` property for a `PXSelector` lookup control to define the appropriate data view for the grid of the lookup window.

Using the Caption Property in the Mobile Site Map

If you plan to use a container in the mobile site map, we recommend that you specify a unique name for the `Caption` property of the container. Then, in the mobile site map, you can refer to the container by the specified caption in the `Name` attribute of the `<sm:Container>` tag, as the following code shows.

```
<sm:Container Name="ValueOfTheCaptionProperty">
...
</sm:Container>
```

Otherwise, in the WSDL schema, the screen-based API web service assigns to the container the name of the first child element. If you use this name in the mobile site map, an error may occur after the container content is reordered because the container name might be changed in the WSDL schema.

Using the SkinID Property

In the code of Acumatica ERP, predefined skins are used to assign a style and a set of toolbar buttons to a container. The `SkinID` property of a container specifies which of these skins the system should apply to the container. A skin is specific to a particular container; you cannot share a skin setting between containers of different types. If you do not set the `SkinID` property, the container uses the default skin if one is defined.

The following table lists and describes the predefined skins that are recommended to use for the `PXFormView`, `PXGrid`, and `PXPanel` containers.

UI Element	SkinID	Description	Example
<code>PXFormView</code>	<i>Transparent</i>	Is used to display a simple form container that has no caption and cannot be collapsed.	The form container on the Financial Details tab item of the <i>Bills and Adjustments</i> form (AP.30.10.00)
<code>PXGrid</code>	<i>Attributes</i>	Is used to display a simple grid without a toolbar. The grid contains a predefined set of rows, which can be edited.	The Attributes grid on the <i>Attributes</i> tab item of the <i>#unique_265</i> form (IN.20.20.00)
	<i>Details</i>	Is used to render a detail grid in a master-detail data entry page. The grid has a toolbar that hosts the default actions, such as Refresh , Add , Remove , Fit to Screen , and Export to Excel , and can display custom actions. The grid has no caption and paging is allowed.	The grid on the 1099 Settings tab item of the <i>Accounts Payable Preferences</i> form (AP.10.10.00)

UI Element	SkinID	Description	Example
	<i>Inquire</i>	Is used to display data without adding or removing rows. The grid has a toolbar that contains the Refresh , Fit to Screen , and Export to Excel default actions and can contain custom actions. The grid has no caption and paging is allowed.	The grid on the Attributes tab item of the <i>Customers</i> form (AR.30.30.00)
	<i>Primary</i>	Is used to display an editable primary grid that does not contain its own toolbar. To work with the grid, the user applies the action buttons of the form toolbar. The grid has no caption and paging is allowed.	The grid on the <i>Entry Types</i> form (CA.20.30.00)
	<i>PrimaryInquire</i>	Is used to display a primary grid without the availability to edit data. The grid does not contain its own toolbar. To work with the grid, the user applies the action buttons of the form toolbar, which does not contain the Add , Delete , and Switch Between Grid and Form buttons. The grid has no caption and paging and filtering are allowed.	The grid on the <i>Release AP Documents</i> form (AP.50.10.00)
	<i>ShortList</i>	Is used to display a small grid with a few records inside a form view. The grid has a toolbar that contains the Refresh , Add , and Remove default actions.	The Sales Categories grid on the Attributes tab item of the <i>#unique_265</i> form (IN.20.20.00)
PXPanel	<i>Buttons</i>	Is used in dialog boxes to display a horizontal row of buttons with the right alignment.	The group of buttons in the Add PO Receipt dialog box, which opens if you click the Add PO Receipt button in the toolbar of the Document Details tab item of the <i>Bills and Adjustments</i> form (AP.30.10.00)
	<i>Transparent</i>	Is used to group controls in a form container. The panel has no caption.	The group of controls on the Template Setting tab item of the <i>Order Types</i> form (SO.20.10.00)

Using the SyncPosition Property

If a form contains a grid and the form toolbar includes an action to process a single record that is highlighted in the grid, the action delegate method must have a reference to the highlighted record in the cache.

To use the `Current` property of a `PXCache` object to access the record highlighted in a grid, the `Current` property must be synchronized with record highlighting in the grid. To force the system to provide this synchronization, you have to set the `SyncPosition` property of the `PXGrid` container to `True`.



: If you need to make an action button on the toolbar unavailable, when a grid is empty, you should set the `DependOnGrid` property of the appropriate `PXDSCallbackCommand` object in the `PXDataSource` control to the value that is specified in the `ID` property of the `PXGrid` element.

To Add a Nested Container

A `PXGrid` container cannot include a nested container. A `PXTab` container can include only `PXTabItem` containers. You can add any container as a nested container to a `PXFormView`, `PXTabItem`, or `PXSmartPanel` container, as the following diagram shows.

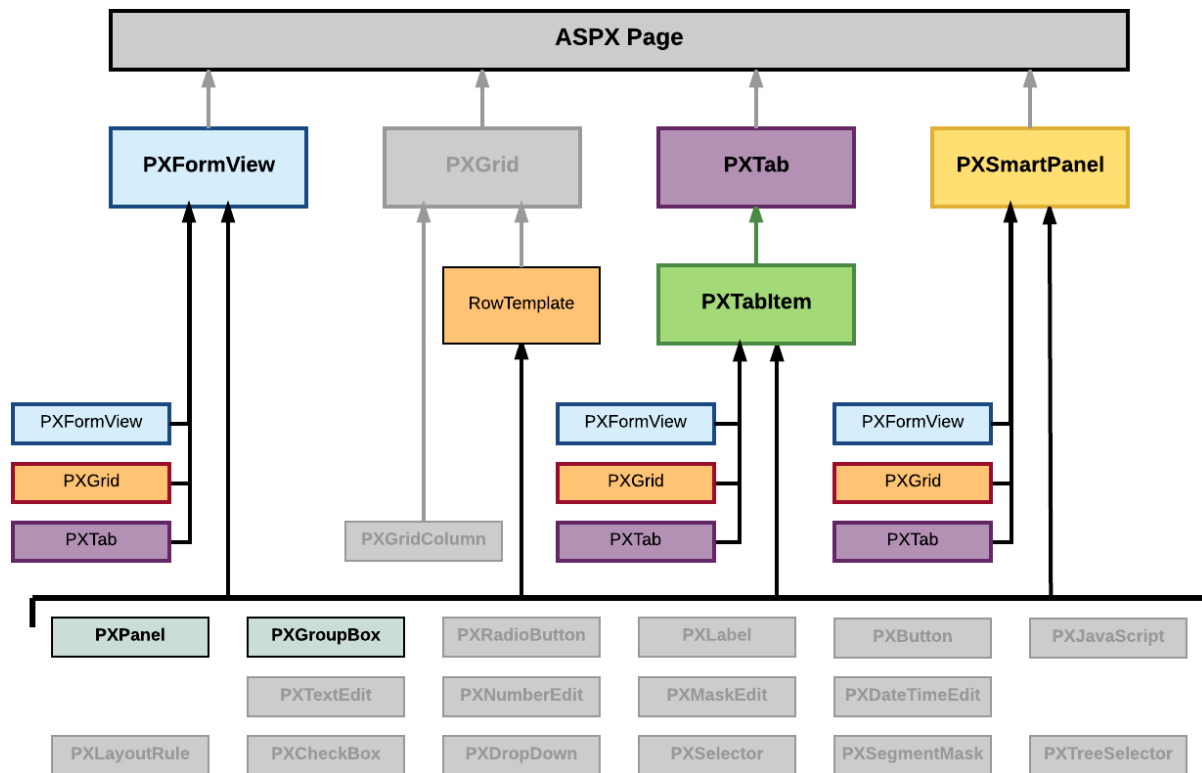


Figure: Nesting rules for containers on an ASPX page

You can include the `PXPanel` and `PXGroupBox` container controls in the `PXFormView`, `RowTemplate`, `PXTabItem`, and `PXSmartPanel` containers.

To add a nested container to a parent container, perform the following actions:

1. Open the parent container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. Ensure that the container node is selected in the Control Tree of the editor. Click the arrow left of the node to expand the node if needed.
3. Click the **Add Controls** tab item (see the screenshot below).
4. From the **Main Containers** or **Other Controls** group, drag the required type of the nested container into the parent container in the Control Tree, as shown in the following screenshot.

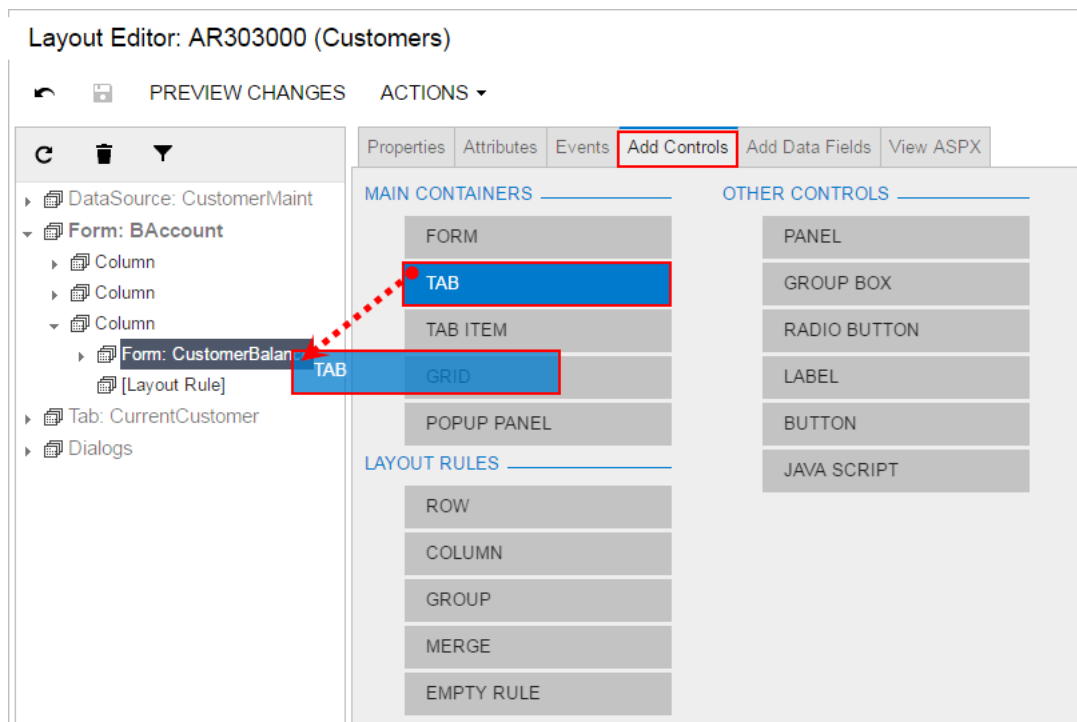


Figure: Adding a nested container to a parent container



: A nested container is visible on the customized form only if it contains at least one visible control.

5. If required, specify properties for the new container, as described in [To Set a Container Property](#).
6. Click **Save** to save changes in the customization project.

To Add a Box for a Data Field

The Acumatica Customization Platform supports the following types of boxes for data fields:

- PXTextEdit
- PXNumberEdit
- PXMaskEdit
- PXDateTimeEdit
- PXCheckBox
- PXDropDown
- PXSelector
- PXSegmentMask
- PXTreeSelector

(See the table in [Customizing Elements of the User Interface](#) for field type descriptions.)

You can add a box for a data field without restrictions immediately to a `PXFormView` or `PXTabItem` element that has the `DataMember` property defined. Also, you can add a box for a data field to a `PXPanel` or `PXGroupBox` container that has the `DataMember` property inherited from the parent container.

Also, you can add a box for a data field to a `RowTemplate` element of a `PXGrid` container that is bound to a data view whose BQL statement refers to the data access class that contains the field declaration.

To add a box for a data field to a container that is bound to a data view, perform the following actions:

1. Open the parent container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. Ensure that the container node is selected in the Control Tree of the editor. Click the arrow left of the node to expand the node if needed.
3. Click the **Add Data Fields** tab item (see the screenshot below).
4. If you need to create a control for a data field that is not accessible through the data view specified for the container in the `DataMember` property but is accessible through another data view of the same graph, and if the **Data View** box is available, select the needed data view in this box. (See [Using Multiple Data Views for Boxes in a Container](#) for details.)
5. On the tab item, click the **All**, **Visible**, or **Custom** filter for the fields provided by the data view to view the appropriate field list.



: You can create a custom field immediately on the **Add Data Fields** tab item by clicking the **New Field** action and then using the [Create New Field](#) dialog box.

6. Find the required data field in the list, and if the field is not used (the check box in the **Used** column is cleared for the field), select the check box for the field in the first (unlabeled) column, as the following screenshot shows.

Layout Editor: AR303000 (Customers)

PREVIEW CHANGES ACTIONS ▾

Properties Attributes Events Add Controls **Add Data Fields** View ASPX

Data View: Lead/Contact(DefContact)

CREATE CONTROLS NEW FIELD ALL **VISIBLE** CUSTOM

	Used	Field Name	Control
<input type="checkbox"/>	<input type="checkbox"/>	DefAddressID (Address)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	DisplayName (Display Name)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	DuplicateFound (Duplicate Found)	CheckBox
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Email (Email)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	ExtRefNbr (Ext Ref Nbr)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Fax	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	FaxType (Fax Type)	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	FirstName (First Name)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	FullName (Company Name)	TextEdit
<input checked="" type="checkbox"/>	<input type="checkbox"/>	Gender	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	IsActive (Active)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	IsAddressSameAsMain (Same As In Account)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	IsConvertible (Can Be Converted)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID (Last Modified By)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_description (Last Modified By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_Modifier_username (Last Modified By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedDateTime (Last Modified Date)	DateTimeEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastName (Last Name)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	MajorStatus	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	MaritalStatus (Marital Status)	ComboBox

Figure: Selecting a data field for which a box to be created



: You can select multiple data fields to create multiple boxes simultaneously.

7. On the list toolbar, click **Create Controls**.
The platform creates a box for the selected data field and adds a node for the box to the Control Tree.
8. Click **Save** to save the changes in the customization project.

You can change the location of a control in a container. See [To Reorder Child UI Elements](#) for details.

For more information about boxes, see [Box \(Control for a Data Field\)](#).

Using Multiple Data Views for Boxes in a Container

The Acumatica Customization Platform supports the use of multiple data views for controls in the same container of an ASPX page.

For example, you can create a container and set the `DataMember` property to the name of the data view that provides most of the fields for boxes in the container. If you also want to create a control for a data field that cannot be accessible through that data view but accessible through another data view of the same graph specified in the `TypeName` property of the `PXDataSource` control, you can specify the required data view immediately in the `DataField` property, as follows.

```
<px:PXField ... DataField="DataViewName.FieldName" />
```

The following code snippet shows how to use the `MyDataView`, `AnotherDataView`, and `OnceMoreDataView` data views declared in the same graph or in extensions for the graph to define boxes for data fields in a `PXFormView` container on an Acumatica ERP form.

```
<px:PXFormView ... DataMember="MyDataView" ...>
  ...
  <px:PXNumberEdit ... DataField="MyField_01" />
  <px:PXSegmentMask ... DataField="MyField_02" />
  <px:PXDateTimeEdit ... DataField="AnotherDataView.FieldName" />
  <px:PXTextEdit ... DataField="MyField_05" />
  <px:PXSelector ... DataField="OnceMoreDataView.OtherFieldName" />
  ...
</px:PXFormView>
```

On the **Add Data Fields** tab item of the Layout Editor, if you change the predefined data view in the **Data View** box and create a control for a data field from the field list for the selected data view, the editor concatenates the data view name with the field name in the `DataField` property of the created control.

To Add a Layout Rule

In a container with multiple controls, the `PXLayoutRule` component is used to provide the following UI customization capabilities:

- Placing controls in multiple rows to uniformly distribute them on the form or tab area of a form
- Placing controls in multiple columns in a row
- Spanning controls across multiple columns in a row
- Merging controls into one row to align them horizontally
- Adjusting the widths of controls and labels in a column
- Hiding the labels of controls
- Grouping controls for users' convenience

The [Layout Editor](#) supports the following types of the `PXLayoutRule` component (with the respective predefined properties noted):

- **Row:** A layout rule with the `StartRow` property, which is set to `True`
- **Column:** A layout rule with the `StartColumn` property, which is set to `True`
- **Group:** A layout rule with the `StartGroup` property, which is set to `True`
- **Merge:** A layout rule with the `Merge` property, which is set to `True`
- **Empty Rule:** A layout rule without predefined properties

To add a layout rule to a container, perform the following actions:

1. Open the container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. Ensure that the container node is selected in the Control Tree of the editor. Click the arrow left of the node to expand the node if needed.
3. Click the **Add Controls** tab item (see the screenshot below).
4. From the **Layout Rules** group, drag the required type of the rule to the needed location in the Control Tree within the container, as shown in the following screenshot.

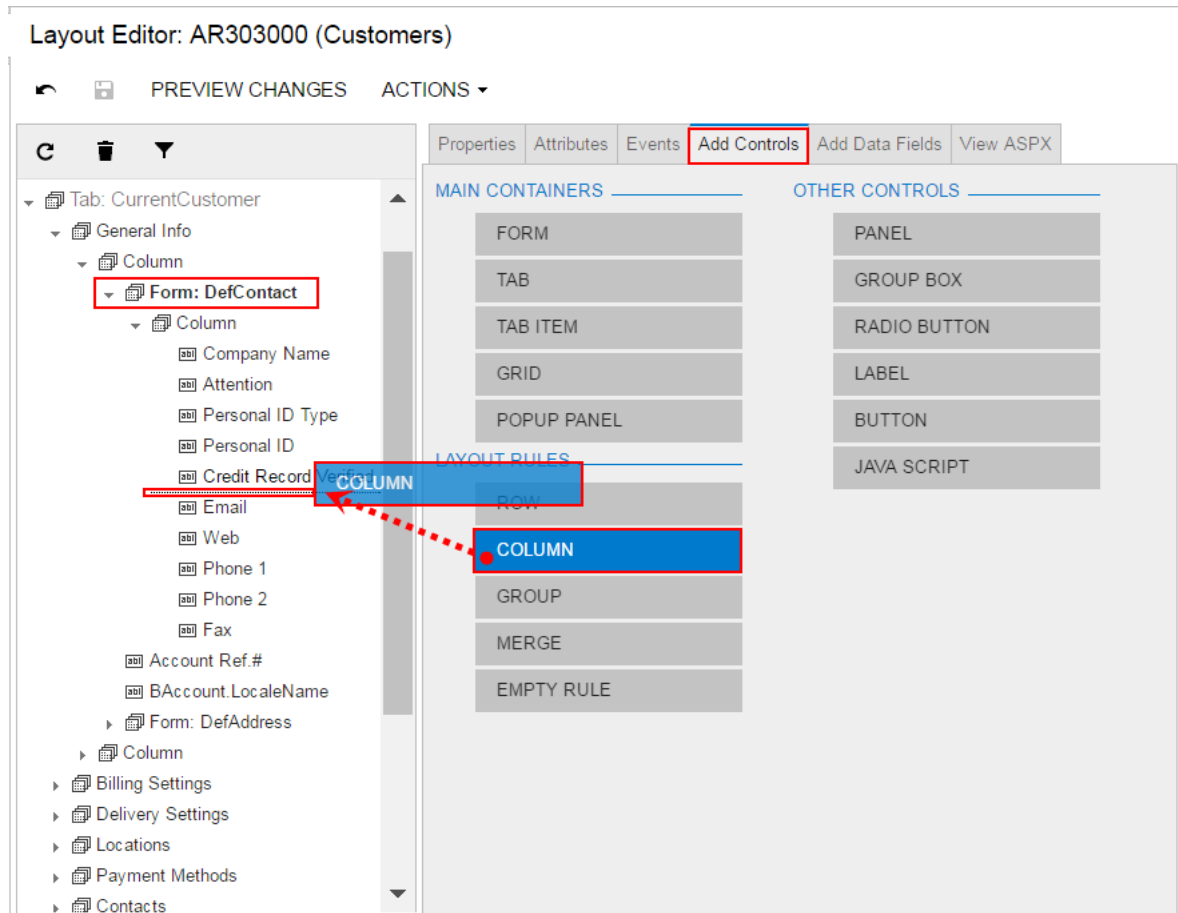


Figure: Adding a layout rule to a container



: A layout rule is visible on the customized form only if it contains at least one visible control.

5. If required, specify properties for the new layout rule.



: In any rule, you can configure any properties that you need. Some properties affect only the next control under the `PXLayoutRule` component, while other properties affect all controls under the rule until the next rule is encountered. Some properties require a corresponding ending rule. See [To Set a Layout Rule Property](#) for details.

6. Click **Save** to save your changes in the customization project.

If you add a layout rule beneath another layout rule, you can override the properties of the `PXLayoutRule` component, which apply to the underlying controls. See [Layout Rule \(PXLayoutRule\)](#) for more information about using layout rules.

To Add Another Supported Control

You use the *Layout Editor* to add to a container a control of any of the following types, which are supported in the Acumatica Customization Platform (listed under **Other Controls** on the **Add Controls** tab item):

- *Panel (PXPanel)*
- *Group Box (PXGroupBox)*
- *Label (PXLabel)*
- *Radio Button (PXRadioButton)*
- *Button (PXButton)*
- *Java Script (PXJavaScript)*

We recommend that you not include a `PXRadioButton` control in a container that neither is bound to a data view nor inherits the `DataMember` property from the parent container.

You can nest a control of the listed types in a `PXPanel` or `PXGroupBox` container. However the `PXGroupBox` control type is especially designed to be used as a radio button container to render a drop-down field as a set of radio buttons. It contains scripts with the logic to support a nested radio button for each value of a drop-down field. So we recommend that you use `PXGroupBox` exclusively to include radio buttons.

To add a control of one of the listed types to a container, perform the following actions:

1. Open the container in the Layout Editor, as described in *To Open a Container in the Layout Editor*.
2. Ensure that the container node is selected in the Control Tree of the editor. Click the arrow left of the node to expand the node if needed.
3. Click the **Add Controls** tab item (see the screenshot below).
4. From the **Other Controls** group, drag the required control type to the needed location in the Control Tree within the container, as shown in the following screenshot.

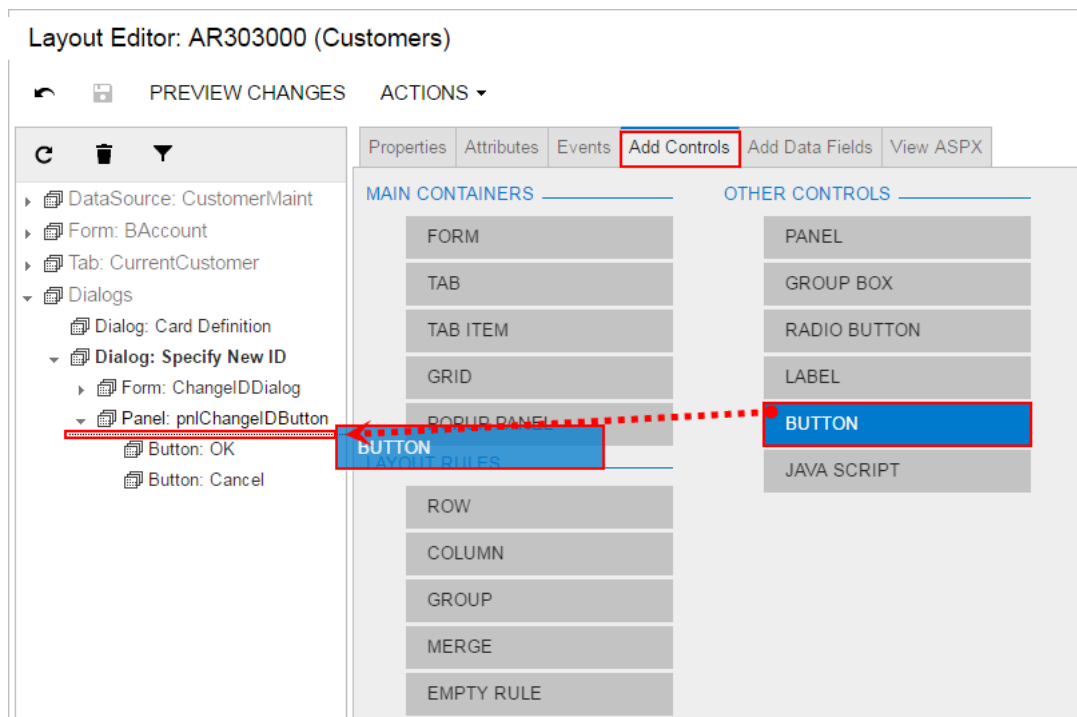


Figure: Adding a layout rule to a container

5. If required, specify properties for the new control.

6. Click **Save** to save your changes in the customization project.

To Reorder Child UI Elements

You can reorder UI elements in a container manually by dragging them in the Control Tree of the *Layout Editor*. However before you start to move elements, note the following simple rules:

- You can move a child element anywhere within its parent container control.
- To move an element within a container, you have to drag the element to the required place.
- Any element moved within a container is automatically aligned according to the nearest `PXLayoutRule` component placed above it. (See *Layout Rule (PXLayoutRule)* for details.)

To move an element within a container, perform the following actions:

1. Open the container in the Layout Editor, as described in *To Open a Container in the Layout Editor*.
2. Ensure that the container node is selected and expanded in the Control Tree of the editor. Click the arrow left of the node to expand the node if needed.
3. In the Control Tree, drag the element to the required position, as the following screenshot shows.

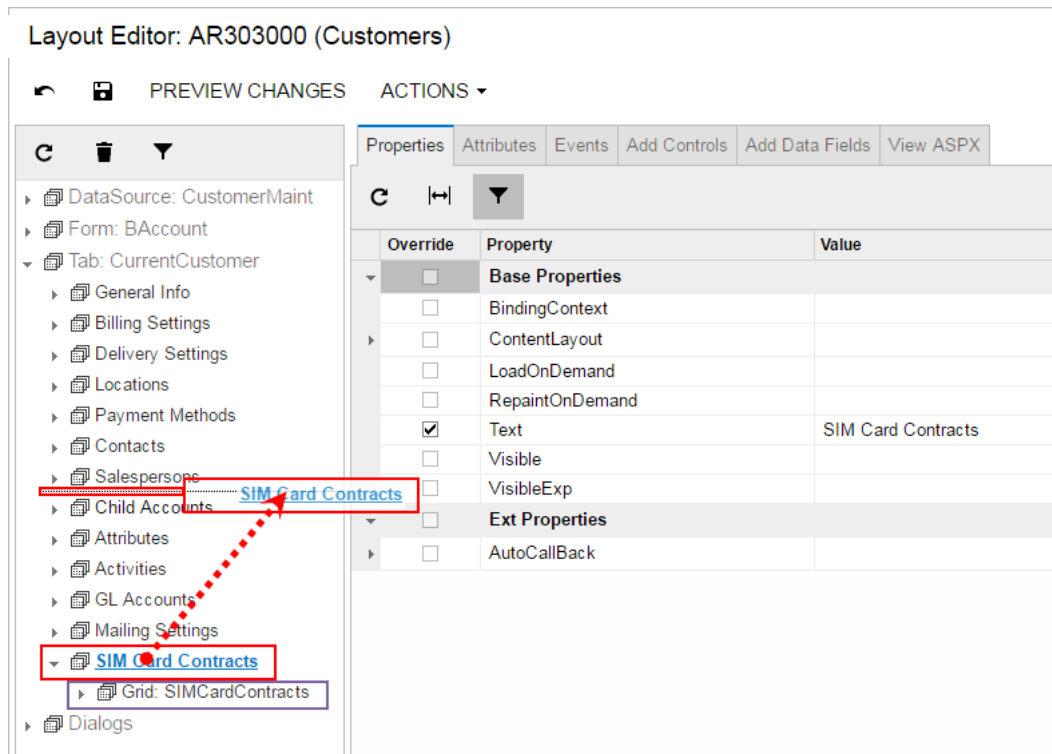


Figure: Moving a control within a container

The Layout Editor moves the element with all its child elements to the new position in the container, as the following screenshot shows.

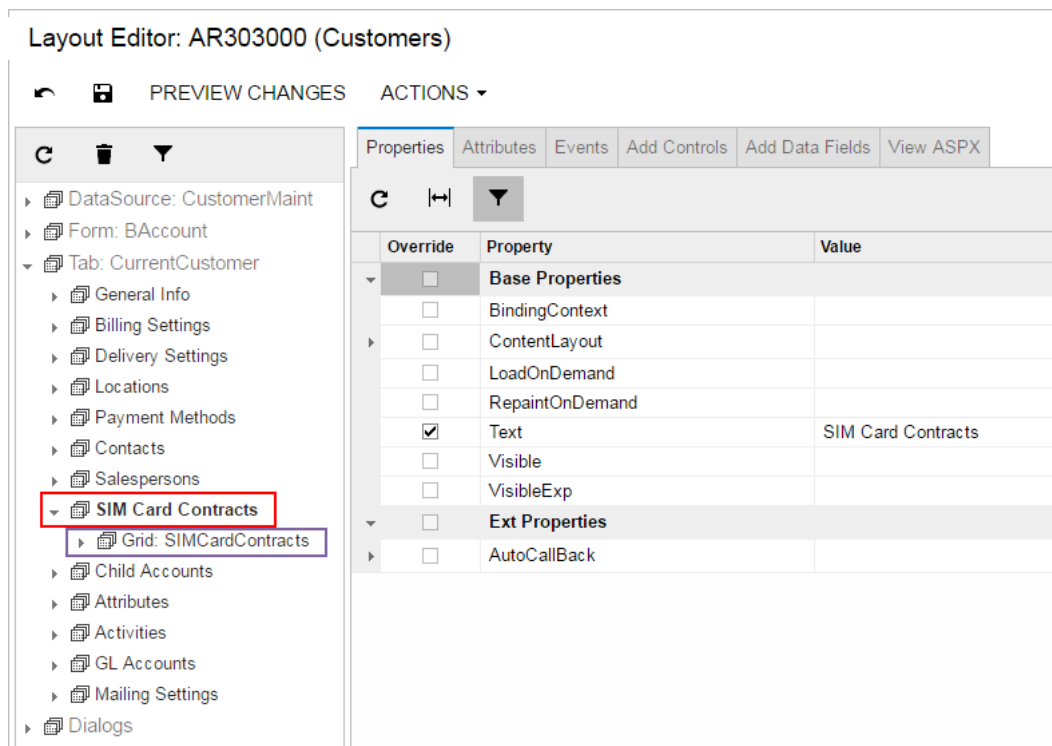


Figure: Noting the placement of the element and its child elements

4. Click **Save** to save your changes to the customization project.

To Delete a Child UI Element

You can delete a child UI element from a container. To do this, perform the following actions:

1. Open the container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. In the Control Tree of the editor, click the arrow left of the container node to expand the node.
3. Select the UI element to be deleted.
4. On the toolbar of the Control Tree, click **Delete**, as the following screenshot shows.

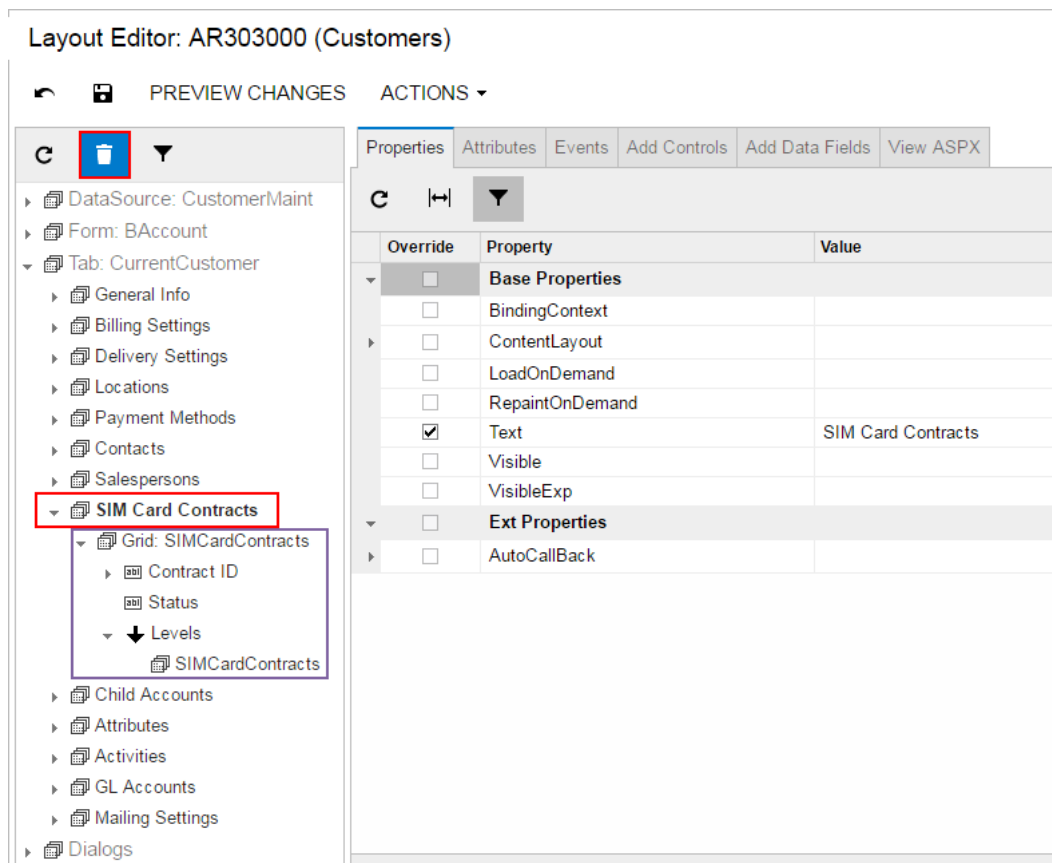


Figure: Deleting an UI element

The platform deletes all child elements of the deleted object.



Warning: If you delete a `PXLayoutRule` component, all the controls under this layout rule are deleted too.

5. Click **Save** on the toolbar of the Layout Editor to save your changes to the customization project.

Grid Container (PXGrid)

`PXGrid` is a data-bound UI container control that renders a table with multiple records from its associated data source. For a single record, the object can be displayed in form view mode, which provides navigation buttons to move between records. For form view mode to be defined, `PXGrid` must include the `RowTemplate` element, which can contain controls for the record fields and layout rules for these controls.

An ASPX page can contain `PXGrid` as a main container that is included immediately in the page. A grid container, as the diagram below shows, can also be included in the following types of containers:

- `PXFormView`
- `PXTabItem`
- `PXSmartPanel`

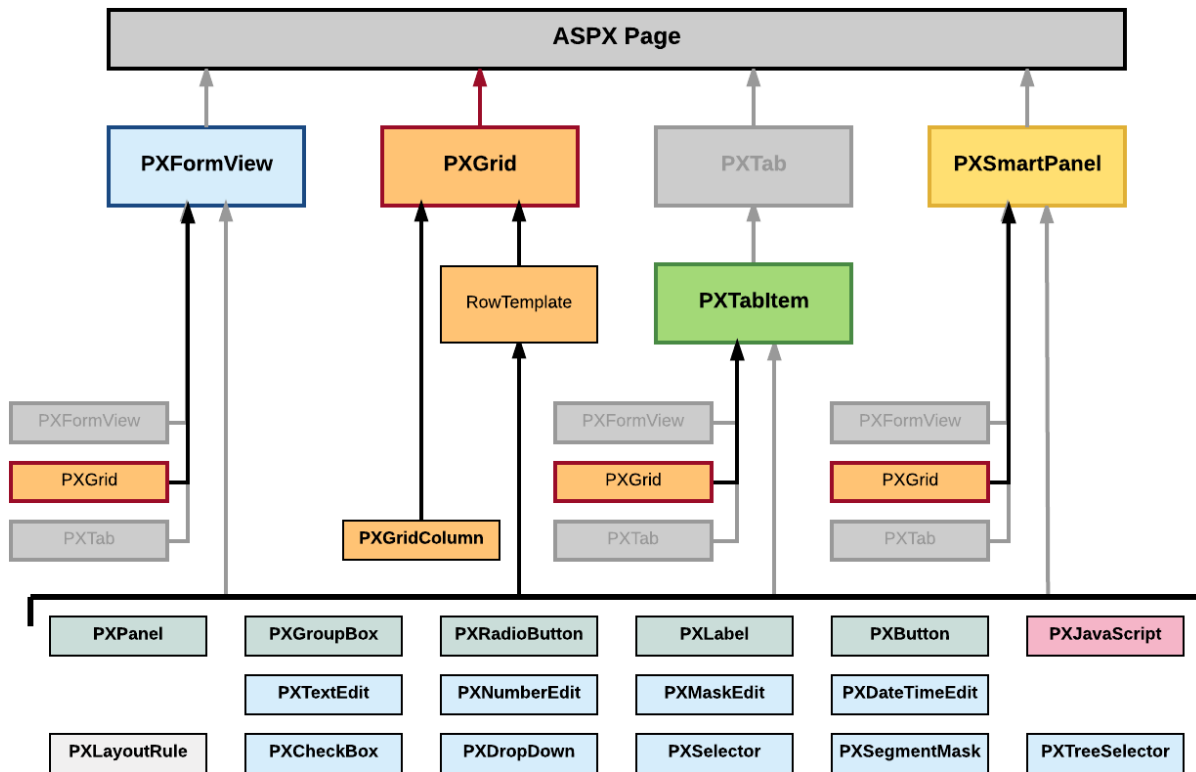


Figure: Nesting rules for a PXGrid container in an ASPX page

A grid container can include multiple `PXGridColumn` objects and a single `RowTemplate` element. In the ASPX code, grid columns are included in the `Columns` element, whereas controls for the form view of the grid belong to the `RowTemplate` element, as the following code snippet shows.

```
<px:PXGrid ID="grid" ... DataSourceID="ds" ...>
  ...
  <Columns>
    ...
    <px:PXGridColumn DataField="Qty" TextAlign="Right" Width="81px"
AutoCallBack="True" />
    ...
  </Columns>
  <RowTemplate>
    ...
    <px:PXNumberEdit ID="edQty" runat="server" DataField="Qty" />
    ...
  </RowTemplate>
  ...
</px:PXGrid>
```

In this code, you can see descriptions of both a grid column and a box for the same `Qty` data field.

To create a new grid in an ASPX page, follow the instructions described in [To Add a Grid Container](#).

To delete a grid from an ASPX page, follow the instructions described in [To Delete a Container](#).

For detailed information on working with the content of a grid container, see the following topics in this section:

- [To Add a Column for a Data Field](#)
- [To Add a Control to the Form View of a Grid](#)

The following topics may also be useful as you work with a grid container:

- [To Open a Container in the Layout Editor](#)
- [To Set a Container Property](#)

- [To Reorder Child UI Elements](#)
- [To Delete a Child UI Element](#)

To Add a Column for a Data Field

In a grid, you can create a column for a data field if the grid container is bound to a data view declared within the graph that provides business logic for the ASPX page. To bind a grid container to a data view, you must specify the properties as follows for the appropriate `PXGrid` object:

- The `DataSourceID` property value must be equal to the value of the `ID` property of the `PXDataSource` control.
- The `DataMember` property must contain the name of the data view that is declared in the graph and provides data for the grid.

To add a column to a grid that is bound to a data view, perform the following actions:

1. Open the grid container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. In the Control Tree of the editor, ensure that the grid node is selected. Click the arrow left of the node to expand the node if needed.
3. In the editor, click the **Add Data Fields** tab item (see the screenshot below).
4. On the tab item, select the **All, Visible** (as shown below), or **Custom** filter for the data fields provided by the data view, to open the appropriate field list.



: You can create a custom field immediately on the **Add Data Fields** tab item by using the [Create New Field](#) dialog box.

5. Find the required data field in the list, and if the field is not used (that is, if the check box in the **Used** column is cleared for the field), select the check box in the unlabeled first column for the field, as the following screenshot shows.

Layout Editor: AR303000 (Customers)

PREVIEW CHANGES ACTIONS ▾

Properties Attributes Events Add Controls **Add Data Fields** View ASPX

Data View: Location with Address(Locations)

CREATE CONTROLS NEW FIELD ALL **VISIBLE** CUSTOM

<input type="checkbox"/>	Used	Field Name	Control
<input type="checkbox"/>	<input type="checkbox"/>	IsRemitAddressSameAsMain (Same as Main)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	IsRemitContactSameAsMain (Same as Main)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID (Last Modified By)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_description (Last Modified By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_Modifier_username (Last Modified By)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	LocationCD (Location ID)	SegmentMask
<input type="checkbox"/>	<input type="checkbox"/>	LocationCreatedByID (Created By)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	LocationCreatedByID_Creator_username (Created By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LocationCreatedByID_description (Created By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LocationLastModifiedByID (Last Modified By)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	LocationLastModifiedByID_description (Last Modified By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LocationLastModifiedByID_Modifier_username (Last Modified By)	TextEdit
<input checked="" type="checkbox"/>	<input type="checkbox"/>	Loc Type (Location Type)	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	PostalCode (Postal Code)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	State	Selector
<input type="checkbox"/>	<input type="checkbox"/>	State_description (State Name)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	State_State_name (State Name)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	TaxRegistrationID (Tax Registration ID)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	VAPAccountID (AP Account)	SegmentMask
<input type="checkbox"/>	<input type="checkbox"/>	VAPAccountID_Account_description (Description)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	VAPAccountID_description (Description)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	VAPSubID (AP Sub.)	SegmentMask

Figure: Selecting a data field for which a column to be created



- You can select multiple data fields to create multiple columns simultaneously.
- In the **Field Name** column of the list, a data field of a joined data access class (DAC) has a name that consists of the DAC name, two underscore characters, and the field name, such as *Product__Active*. If you create a grid column for this field, the new `PXGridColumn` element refers to the field as shown in the following ASPX code.

```
<px:PXGridColumn DataField="Product__Active" ... />
```

6. On the list toolbar, click **Create Controls**.

The platform creates a column for the selected data field, appends this column to the end of the grid column list, and adds a node for the column to the appropriate position in the Control Tree.



- If you need to locate the new column in a position after some existing column, you can select this existing column before you click **Create Controls**. Then the editor inserts new grid columns after the column that is selected in the Control Tree.

At any time, you can change the position of a column in a grid. See [To Reorder Child UI Elements](#) for details.

7. If needed, specify the following properties of the new column:

- Type**—to define a specific type of data in a column (see [Using the Type Property of PXGridColumn](#) for details)
- CommitChanges**—to enable callbacks on the column field (see [Using the CommitChanges Property](#) for details)

- `DisplayMode`—to define the mode of displaying a value in the column (see [Using the DisplayMode Property of PXGridColumn](#) for details)



: If you need to provide hyperlinks to redirect the user from a column cell to another Acumatica ERP form, follow the recommendations described in [Providing Hyperlinks for a Grid Column](#).

8. Click **Save** to save your changes to the customization project.

Using the Type Property of PXGridColumn

The Acumatica Customization Platform supports the following values for the `Type` property of a column in a grid.

Value	Description
<i>NotSet</i>	The default value. An indicator that the field value is displayed in the column as a plain string that is formed based on the field data format.
<i>CheckBox</i>	An indicator that the field value is displayed in the column as a check box, which is selected if the field value is <i>True</i> .
<i>HyperLink</i>	An indicator that the field value is displayed in the column as a hyperlink.
<i>DropDownList</i>	An indicator that the column cell is rendered as a drop-down list, which contains all the values specified for the referred data field.
<i>Icon</i>	An indicator that the field value contains an image URL and is displayed in the column as the referred image.

For example, the following code fragment defines the grid columns on the Automation Schedules (SM.20.50.30) form.

```

...
<px:PXGridColumn AllowUpdate="False" DataField="LastRunStatus" Width="40px"
  Type="Icon" TextAlign="Center" />
<px:PXGridColumn DataField="ScreenID" DisplayFormat="CC.CC.CC.CC" Label="Screen ID"
  LinkCommand="AUScheduleExt_View" />
<px:PXGridColumn DataField="Description" Label="Description" Width="200px" />
<px:PXGridColumn AllowNull="False" DataField="IsActive" Label="Active"
  TextAlign="Center" Type="CheckBox" Width="60px" />
...

```

In the code, the `Type` property for the `LastRunStatus` data field (the **Status** column on the screenshot below) is set to `Icon`. Because the field value contains the image URL, the column cell displays the referred image.

For the `IsActive` data field (the **Active** column), the `Type` property is set to `CheckBox`. As you can see in the screenshot, the column cells are rendered as check boxes.

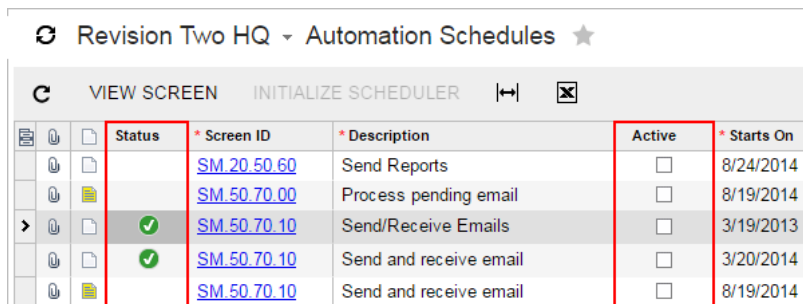


Figure: Viewing different types of columns on the Automation Schedules form

Using the DisplayMode Property of PXGridColumn

The Acumatica Customization Platform supports the following values for the `DisplayMode` property of a column in a grid.

Value	Description
<i>Value</i>	Default value. An indicator that the column cell contains the value of the field.
<i>Text</i>	If there is a description defined for the field, an indicator that the column cell contains the description of the field.
<i>Hint</i>	If there is a description defined for the field, an indicator that the column cell contains the value-description pair of the field.



: The `Type` property priority is higher than the `DisplayMode` property priority. If the `Type` property is set, for example, to `CheckBox`, the `DisplayMode` property is ignored.

Providing Hyperlinks for a Grid Column

To provide hyperlinks for an existing column in a grid, you should perform the following actions:

1. Add the `PXSelector` attribute for the field used to create the column.
2. For the field, create a selector control in the `RowTemplate` element of the `PXGridLevel` container of the ASPX page, as described in [To Add a Control to the Form View of a Grid](#).
3. Set the `AllowEdit` property of the selector control to `True`.



: A selector control contains the value of the key field of a data record from a particular table. Therefore, the control can be used to redirect a user to the form designed to edit the data record.

After you have completed these actions:

- The column fields contain hyperlinks to redirect the user to the data entry form for the records defined by the fields.
- If the form view for the grid is available, the following interface elements appear on the form view:
 - The text box created for the field
 - The **Edit** button, which provides the required redirection to the data entry form for the record defined by the value in the text box

To Add a Control to the Form View of a Grid

If the `AllowFormEdit` property of a grid is set to `True`, the user can switch the grid to form view mode to display the grid columns as controls on the form. This mode gives you the capability to edit a single record selected in the grid.

Therefore, to add a box for a data field to the form view of a grid, you have to add the box to the `RowTemplate` element in the ASPX code. To do this in the [Layout Editor](#), perform the following actions:

1. Open the grid container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. In the Control Tree of the editor, click the arrow left of the node to expand the node.
3. Click the arrow left of the **Levels** node to expand the node, and then expand the node that appears, which has the same name as the grid does.




: In the Control Tree, the Layout Editor assigns to a grid node the `Grid:<DataMember>` name, where the `<DataMember>` is the value of the `DataMember` property of the grid container. To the node that corresponds to the `RowTemplate` element of the grid, the editor assigns the same `<DataMember>` name.

4. If the expanded node contains other expandable nodes, such as `Columns` nodes, expand them to see which boxes are currently included in the form view of the grid.



: The `RowTemplate` element can contain layout rules, which are used to arrange controls in the form view of the grid. Also, you can use this element to provide specific properties of columns in the grid.

5. If you need to place the new box in a position below some existing box, select the node of this box in the Control Tree before you add a new control. (In the screenshot below, the *Mailing ID* node is currently selected in the tree.)
6. In the editor, click the **Add Data Fields** tab item.
7. On the tab item, click the **All**, **Visible**, or **Custom** filter for the data fields provided by the data view to open the appropriate field list.

 : You can create a custom field immediately on the **Add Data Fields** tab item by using the [Create New Field](#) dialog box.
8. Find the required data field in the list, and if the field is not used (that is, if the check box in the **Used** column is cleared for the field), select the check box for the field in the first column, as the following screenshot shows.

Layout Editor: AR303000 (Customers)

PREVIEW CHANGES ACTIONS


Properties Attributes Events Add Controls **Add Data Fields** View ASPX

Data View: NotificationSource(NotificationSources)

CREATE CONTROLS NEW FIELD ALL **VISIBLE** CUSTOM

<input type="checkbox"/>	Used	Field Name	Control
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Active	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	CreatedByID (Created By)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	CreatedByID_Creator_username (Created By)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	CreatedByID_description (Created By)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	EmailAccountID (Email Account)	Selector
<input checked="" type="checkbox"/>	<input type="checkbox"/>	EmailAccountID_description (Email Address)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	EmailAccountID_EmailAccount_address (Email Address)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Format	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID (Last Modified By)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_description (Last Modified By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_Modifier_username (Last Modified By)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	NBranchID (Branch)	SegmentMask
<input type="checkbox"/>	<input checked="" type="checkbox"/>	NBranchID_Branch_acctName (Branch Name)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	NBranchID_description (Branch Name)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	NotificationID (Notification Template)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	NotificationID_description (Notification)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	NotificationID_Notification_name (Notification)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	OverrideSource (Overridden)	CheckBox
<input type="checkbox"/>	<input checked="" type="checkbox"/>	ReportID (Report)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	ReportID_description (Report Name)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	ReportID_SiteMap_title (Report Name)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	SetupID (Mailing ID)	Selector

Figure: Selecting a data field to create a control on the form view of the grid

 : You can select multiple data fields to create multiple columns simultaneously.

9. On the list toolbar, click **Create Controls**.

The platform creates a box for the selected data field and adds a node for the box to the appropriate position in the Control Tree. At any time, you can change the position of a column in a grid. See [To Reorder Child UI Elements](#) for details.
10. If needed, specify properties for the new control.



: If you need to provide hyperlinks to redirect the user from a column cell to another Acumatica ERP form, follow the recommendations described in [Providing Hyperlinks for a Grid Column](#).

11. Click **Save** to save your changes to the customization project.

Tab Container (PXTab)

PXTab is a data-bound UI container control that renders tabs defined by child PXTabItem containers.

An ASPX page can contain PXTab as a main container. A tab container, as the diagram below shows, can be also included in the following types of containers:

- PXFormView
- PXTabItem
- PXSmartPanel

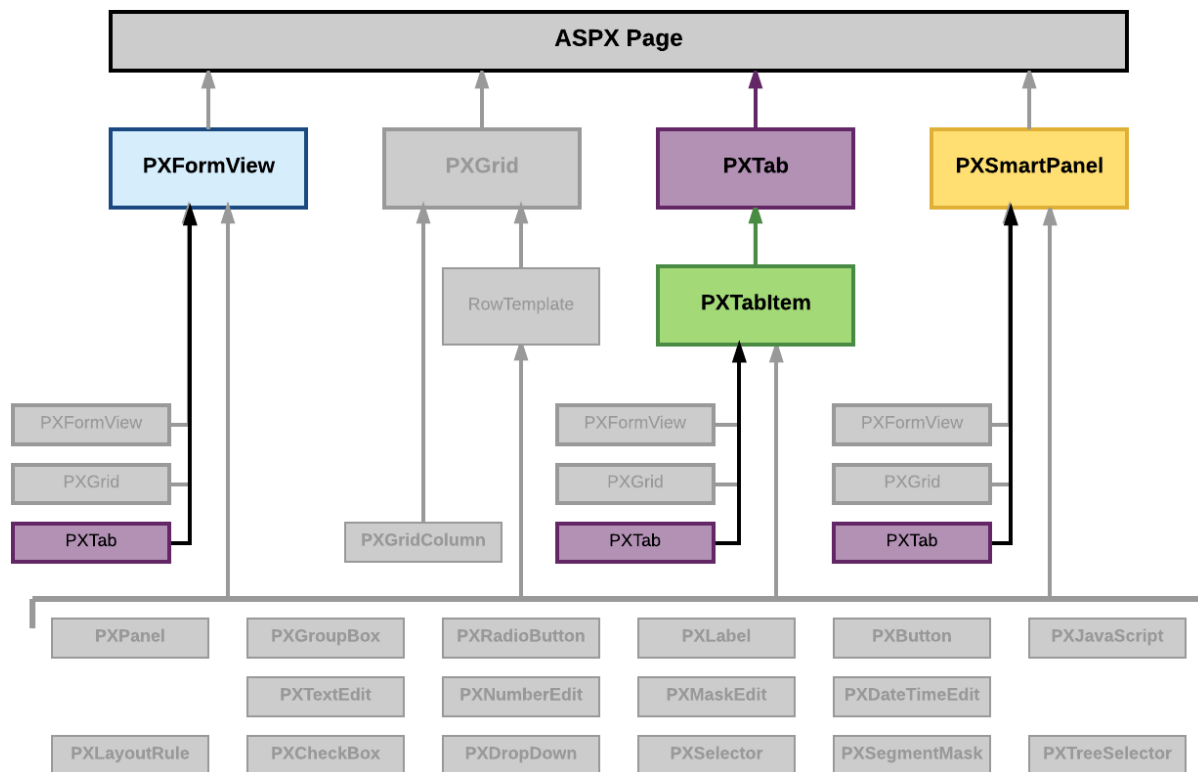


Figure: Nesting rules for a PXTab container in an ASPX page

A tab container can include only PXTabItem container controls.

A box for a data field can be added immediately to a child PXTabItem container if the parent PXTab container is bound to a data view declared within the graph that provides business logic for the ASPX page. To bind a tab container to a data view, you must specify the properties as follows for the appropriate PXTab object:

- The `DataSourceID` property value must be equal to the value of the `ID` property of the `PXDataSource` control.
- The `DataMember` property must contain the name of the data view that is declared in the graph and provides data for the controls of a child container that is not a data-bound UI container.

To create a new tab container in an ASPX page, follow the instructions described in [To Add a Tab Container](#).

To delete a tab container from an ASPX page, follow the instructions described in [To Delete a Container](#).

For detailed information about the `PXTabItem` container control, see [Tab Item Container \(PXTabItem\)](#).

Tab Item Container (PXTabItem)

`PXTabItem` is a container control that can be used to render a single record from the data source specified for the parent `PXTab` container.

A tab item container, as the diagram below shows, can be included only in a `PXTab` container.

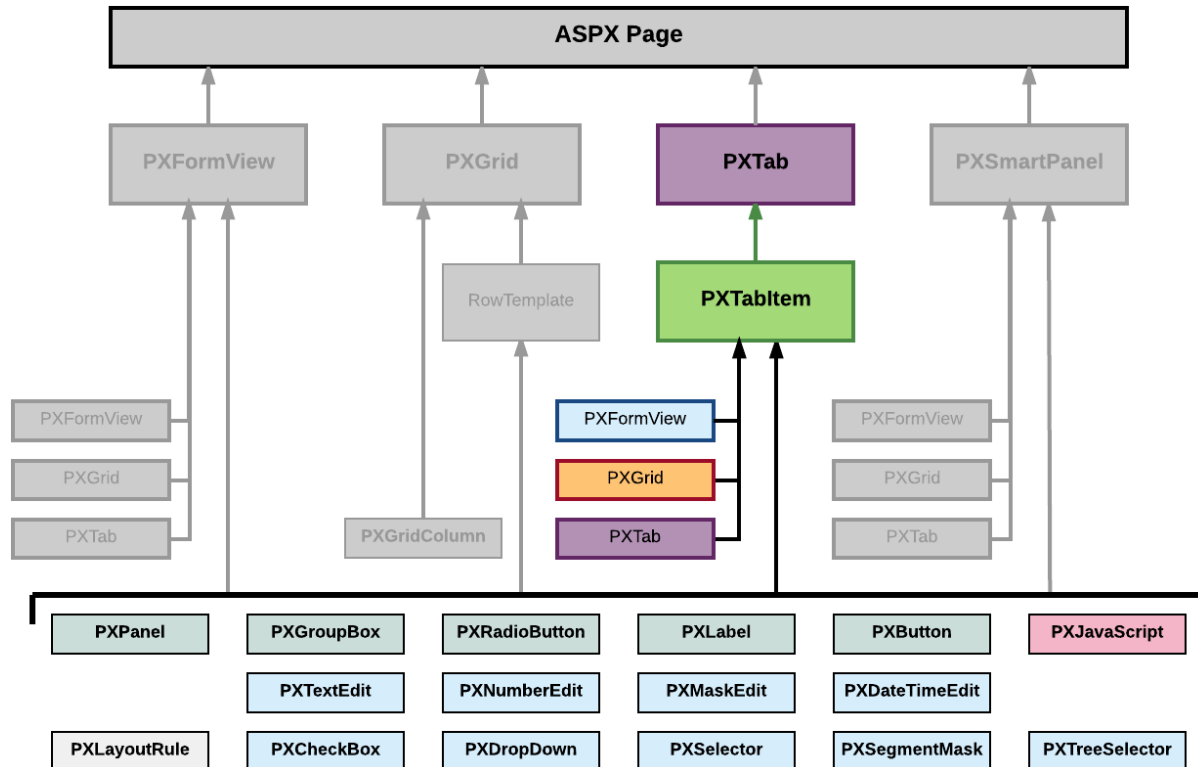


Figure: Nesting rules for a `PXTabItem` container in an ASPX page

A tab item container can include multiple ASPX objects of the following types:

- A data-bound UI container control: `PXFormView`, `PXGrid`, and `PXTab`
- A layout rule: `PXLayoutRule`
- A box for a data field: `PXTextEdit`, `PXNumberEdit`, `PXMaskEdit`, `PXDateTimeEdit`, `PXCheckBox`, `PXDropDown`, `PXSelector`, `PXSegmentMask`, and `PXTreeSelector`
- Another control: `PXPanel`, `PXGroupBox`, `PXRadioButton`, `PXLabel`, `PXButton`, and `PXJavaScript`

A box for a data field can be added to a tab item container if the parent `PXTab` container is bound to a data view declared within the graph that provides business logic for the ASPX page.

To create a new tab item container in a tab container in an ASPX page, follow the instructions described in [To Add a Nested Container](#).

To delete a tab item container from an ASPX page, follow the instructions described in [To Delete a Child UI Element](#).

For detailed information on working with a tab item container, see the [To Conditionally Hide a Tab Item](#) topic in this section. You will find additional information in the following topics:

- [To Open a Container in the Layout Editor](#)
- [To Set a Container Property](#)
- [To Add a Nested Container](#)

- [To Add a Box for a Data Field](#)
- [To Add a Layout Rule](#)
- [To Add Another Supported Control](#)
- [To Reorder Child UI Elements](#)
- [To Delete a Child UI Element](#)

To Conditionally Hide a Tab Item

You can use the `Visible` property of the `PXTabItem` element to set the visibility of the tab item. However if you need to set the dependency of a tab item's visibility from a condition, you should use the `VisibleExp` and `BindingContext` properties of the `PXTabItem` element.

The `VisibleExp` property contains a condition expression that defines a `Boolean` value used to set visibility of the tab item. The expression must consist of two parts and an operator to compare these parts. The expression can contain the values of controls that belong to the container specified in the `BindingContext` property.

The `BindingContext` property defines the ID of the container of the controls whose values can be used in the expression of the `VisibleExp` property.

For example, on a form with form and tab containers, if you need to set the visibility of a tab item to depend on a check box of the form container, you can define the `VisibleExp` and `BindingContext` properties of the `PXTabItem` element, as illustrated in the following ASPX code snippet.

```
<px:PXFormView ID="form" ...>
...
  px:PXCheckBox ... ID="myControlID" ... />
...
</px:PXFormView>
...
<px:PXTab ...>
...
  <px:PXTabItem ... BindingContext="form" ...
    VisibleExp="DataControls[&quot;myControlID&quot;].Value == true">
...
</px:PXTab>
```

In the code above, the expression uses the `DataControls` .NET property of the form object as a dictionary to find the needed control by the specified ID.

Dialog Box (PXSmartPanel)

`PXSmartPanel` is a UI container control that renders a dialog box.

A `PXSmartPanel` container does not have the `DataMember` property; therefore, it cannot contain a UI element for a data field. To add a box for a data field to a dialog box, in the appropriate `PXSmartPanel` container, you have to include the data-bound container that can contain the required data field.

However without binding a data view, you can create, for example, a message box with the controls that contain all required data in the ASPX code.

A `PXSmartPanel` container can include multiple ASPX objects of the following types (see the diagram below):

- A data-bound UI container control: `PXFormView`, `PXGrid`, and `PXTab`
- A layout rule: `PXLayoutRule`
- Another control: `PXPanel`, `PXGroupBox`, `PXLabel`, `PXButton`, and `PXJavaScript`

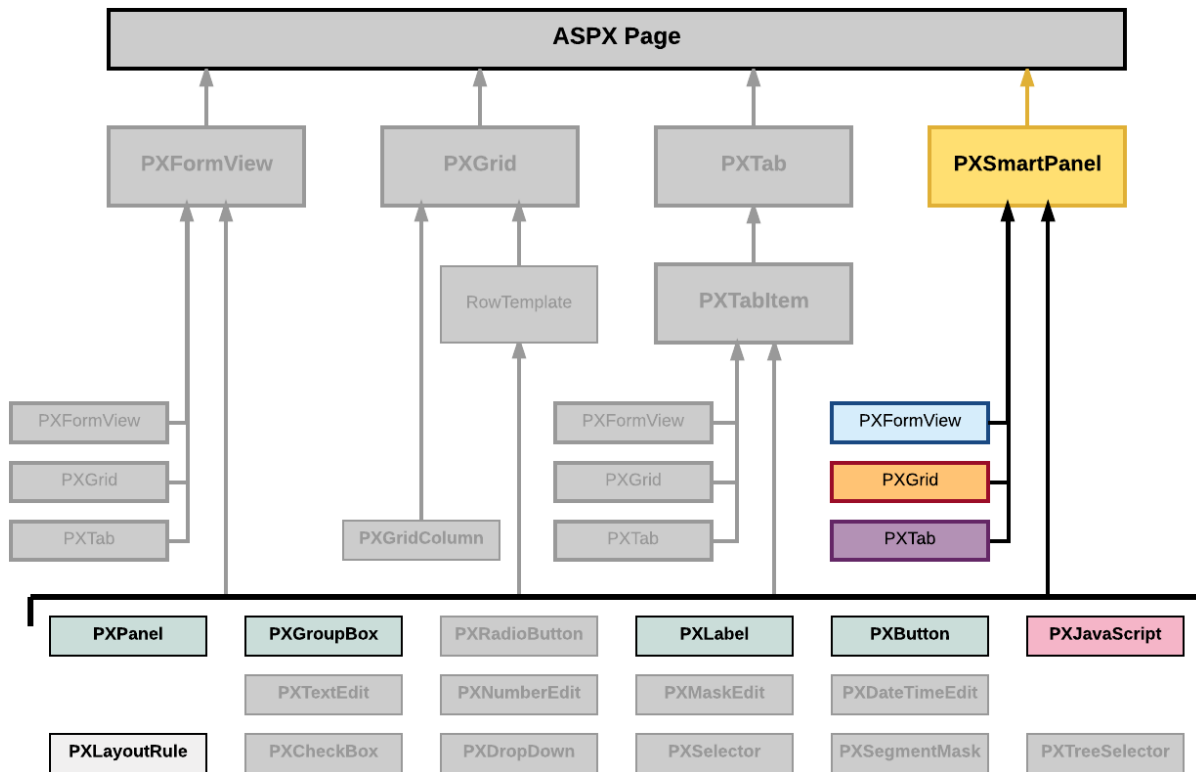


Figure: Nesting rules for a PXSmartPanel container in an ASPX page



: A box for a data field cannot be added immediately to a dialog box because this type of container cannot be bound to a data view.

To create a new dialog box in a form, follow the instructions described in [To Add a Dialog Box](#).

To delete a dialog box from a form, follow the instructions described in [To Delete a Container](#).

In Acumatica ERP, a dialog box usually contains a container for data fields and a `PXPanel` container with `PXButton` elements to get a response from the user. See [Panel \(PXPanel\)](#) and [To Use a Button in a Dialog Box](#) for details.

For detailed information on working with the content of a dialog box, see the [To Open a Smart Panel in the Layout Editor](#) topic in this section. You will find additional information in the following topics:

- [To Set a Container Property](#)
- [To Add a Nested Container](#)
- [To Add Another Supported Control](#)
- [To Reorder Child UI Elements](#)
- [To Delete a Child UI Element](#)

To Open a Smart Panel in the Layout Editor

If you need to activate the [Element Inspector](#) for a pop-up panel, a dialog box, or another UI element that opens in modal mode and makes the [Customization Menu](#) unavailable for selection, you can press Control-Alt.

To open a smart panel in the [Layout Editor](#), perform the following actions:

1. Open the form in the browser.
2. On the form, open the needed dialog box by using the appropriate action.
3. On the keyboard, press the Control-Alt combination.



: The Element Inspector is activated while you keep the Control-Alt combination pressed on the keyboard.

4. Click anywhere inside the dialog box to open the [Element Properties](#) dialog box.
5. In the dialog box, click **Customize**.
6. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one (see [To Create a New Project](#) and [To Select an Existing Project](#) for details).

If the customization project does not contain a changeset for the form, the [Customization Project Editor](#) adds to the project a *Page* item for the form to keep your changes in the database. The smart panel element is opened in the [Layout Editor](#), and you can start to customize the dialog box.

When you click **Save** on the editor toolbar, the editor updates the *Page* item in the database.

Box (Control for a Data Field)

You can use a control for a data field, also referred as a *box*, to display and edit the value of the field on a form.

The Acumatica Customization Platform supports the following types of boxes.

Object	Description
PXTextEdit	A text box to display and edit the value of a <code>string</code> field.
PXNumberEdit	A box to display and edit the value of a <code>decimal</code> or <code>int</code> field.
PXMaskEdit	A text box to display and edit the value of a <code>string</code> field that has the format specified in the data access class.
PXDateTimeEdit	A box to display and select the value of a <code>datetime</code> field.
PXCheckBox	A check box to display and select the value of a <code>bool</code> field.
PXDropDown	A combo box to display, edit, and select the value of a field with a list attribute, such as <code>PXStringList</code> , <code>PXIntList</code> , or <code>PXDecimalList</code> .
PXSelector	A lookup control to display, search for, and select the value of a field with the <code>PXSelector</code> attribute.
PXSegmentMask	A lookup control with a specified segmented key value that identifies a data record and consists of one segment or multiple segments, where the list of possible values is defined for each segment.
PXTreeSelector	A lookup control to select a value for a field with a <code>PXTreeSelector</code> attribute from a tree control.

A box, as the diagram below shows, can be immediately added to the following types of containers:

- `PXFormView`
- `RowTemplate`
- `PXTabItem`

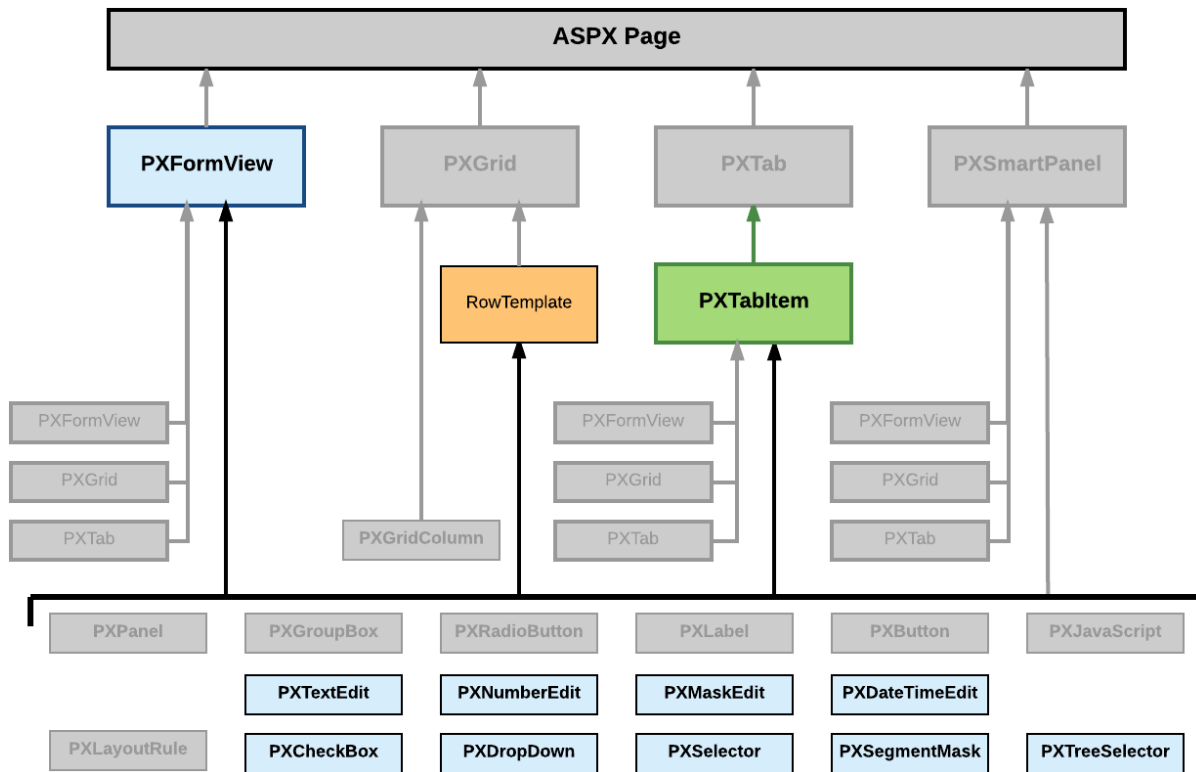


Figure: Nesting rules for a box bound to a data field

To create a box in a container, follow the instructions described in [To Add a Box for a Data Field](#).

To delete a box from a container, follow the instructions described in [To Delete a Child UI Element](#).

For detailed information on customizing a box, see the following topics:

- [To Select a Box in the Layout Editor](#)
- [To Set a Box Property](#)
- [To Change the Type of a Box](#)

To Select a Box in the Layout Editor

To start the customization of a box, you have to select it in the [Layout Editor](#). To do this, perform the following actions.

1. Open the form that contains the box to be customized so that the box is displayed on the screen.
2. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).
3. On the form, click the box to open the [Element Properties Dialog Box](#) for the box.
4. In the dialog box, click **Customize**.
5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one. (See [To Select an Existing Project](#) and [To Create a New Project](#) for details.)

If the customization project does not contain a changeset for the form, the [Customization Project Editor](#) adds to the project a *Page* item for the form, to keep the changeset to ASPX code of the form in the database. The container of the box is opened in the [Layout Editor](#), and the box is selected in the Control Tree of the editor. Therefore you can start customization of the box.

When you click **Save** on the editor toolbar, the editor updates the *Page* item in the database.

To Set a Box Property

To include in a customization project changes to properties of a box, you have to modify the properties by using the *Layout Editor*. To start setting the properties of a box, perform the following actions:

1. Select the box in the Layout Editor, as described in *To Select a Box in the Layout Editor*.
2. Click the **Properties** tab item to open the list of properties for the box.
3. Specify values for the required properties.
4. Click **Save** to save your changes to the customization project.

For detailed information about the `CommitChanges` property, see *Using the CommitChanges Property*.

You can assign a predefined size abbreviation (such as *XXS*, *L* or *XL*) for the `LabelsWidth` and `Size` properties of a box. See *Using Predefined Size Values* for details.

Using the CommitChanges Property

If you need to process the value in a box every time the user changes this value, you need to set the `CommitChanges` property of the box to `True` to enable callbacks for the box.

If callback is enabled for a box in a container on a page, the user has changed the box value, and focus is no longer on the box on the page, the container immediately collects all the modified data and a callback is created to pass the data to the `PXDataSource` control of the page (see the diagram below).

The `PXDataSource` control creates a remote procedure call to the application server to execute the *Update* operation with the modified data on the data view that is specified as the `DataMember` property for the container. The data view executes the *Updating a Data Record* scenario on the data in the cache object of the business logic controller. The cache object raises the events that you can handle to process the modified data.

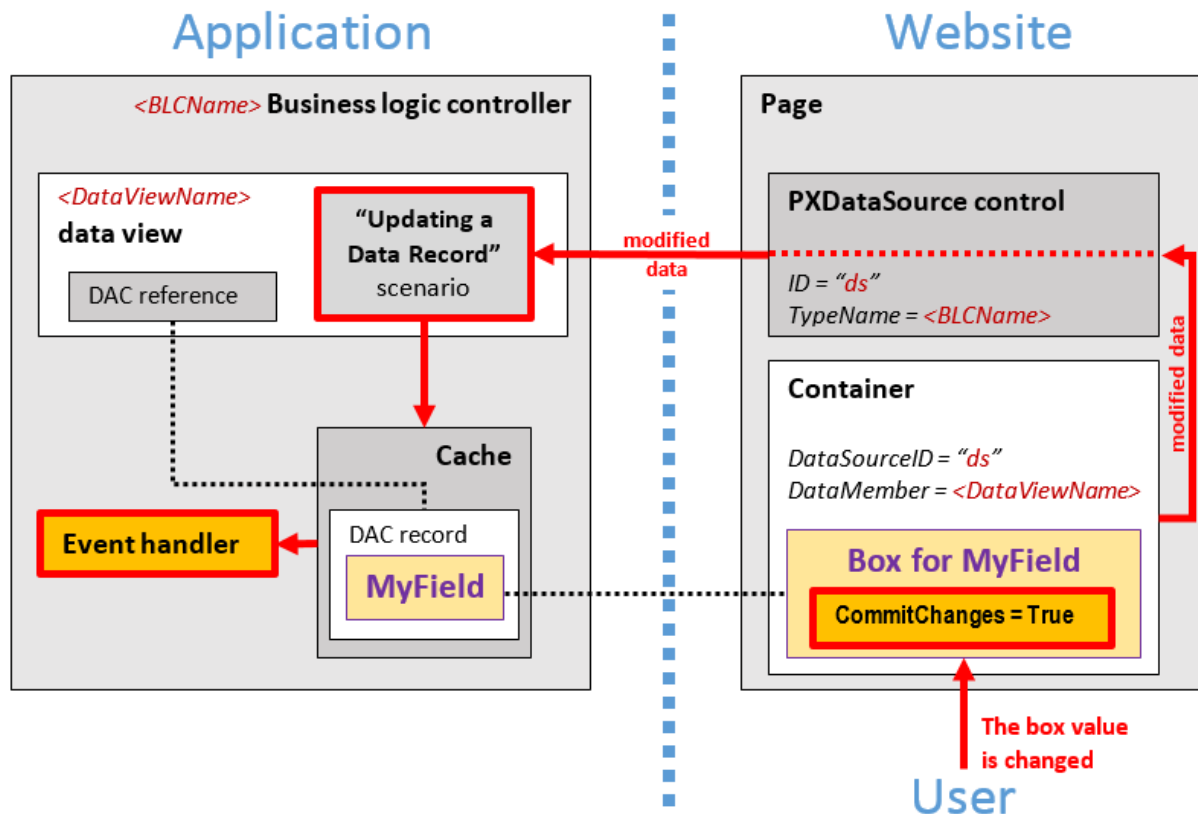


Figure: Usage of the `CommitChanges` property to process a modified data

To Change the Type of a Box

For a data field, you can create a control of any type that is supported in Acumatica ERP. However to avoid issues with rendering controls and processing control values, you have to have an appropriate control for each data field. Therefore, if you have changed the type of a data field, you should recreate all the controls that exist on Acumatica ERP forms for this field.

For example, a text edit box for the *MyFieldName* `PXDBString` field can be defined in the ASPX code as follows.

```
<px:PXTextEdit ... DataField="MyFieldName" />
```

Suppose that you have added the `PXStringList` attribute for the field in the data access class to use the control for the field as a drop-down box. Then you need to redefine the control as follows.

```
<px:PXDropDown ... DataField="MyFieldName" />
```

Because this change must be saved in a customization project, you have to use the [Layout Editor](#) to delete the old definition of the box and to add the new one.

To change the type of a box in a container on a form and to include this modification in a customization project, perform the following actions:

1. Select the box in the Layout Editor, as described in [To Select a Box in the Layout Editor](#).
2. On the toolbar of the Control Tree, click **Delete**.
3. Click the **Add Data Fields** tab item (see the screenshot below).
4. If you need to create a control for a data field that is not accessible through the data view specified for the container in the `DataMember` property, but is accessible through another data view of the same graph, and the **Data View** box gives you the availability to select a data view, select the needed data view in this box. (See [Using Multiple Data Views for Boxes in a Container](#) for details.)
5. On the tab item, click the **All**, **Visible**, or **Custom** filter for the fields provided by the data view selected in the **Data View** box to view the appropriate field list.
6. Find the required data field in the **Field Name** column of the list and select the check box for the field in the unlabeled first column, as the following screenshot shows.

Layout Editor: AR303000 (Customers)

PREVIEW CHANGES ACTIONS ▾

Properties Attributes Events Add Controls **Add Data Fields** View ASPX

Data View: Lead/Contact(DefContact)

CREATE CONTROLS NEW FIELD ALL **VISIBLE** CUSTOM

<input type="checkbox"/>	Used	Field Name	Control
<input type="checkbox"/>	<input type="checkbox"/>	DefAddressID (Address)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	DisplayName (Display Name)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	DuplicateFound (Duplicate Found)	CheckBox
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Email (Email)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	ExtRefNbr (Ext Ref Nbr)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Fax	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	FaxType (Fax Type)	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	FirstName (First Name)	TextEdit
<input type="checkbox"/>	<input checked="" type="checkbox"/>	FullName (Company Name)	TextEdit
<input checked="" type="checkbox"/>	<input type="checkbox"/>	Gender	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	IsActive (Active)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	IsAddressSameAsMain (Same As In Account)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	IsConvertible (Can Be Converted)	CheckBox
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID (Last Modified By)	Selector
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_description (Last Modified By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedByID_Modifier_username (Last Modified By)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastModifiedDateTime (Last Modified Date)	DateTimeEdit
<input type="checkbox"/>	<input type="checkbox"/>	LastName (Last Name)	TextEdit
<input type="checkbox"/>	<input type="checkbox"/>	MajorStatus	ComboBox
<input type="checkbox"/>	<input type="checkbox"/>	MaritalStatus (Marital Status)	ComboBox

Figure: Selecting a data field for which a box is to be created

For each data field in the list, the Layout Editor presets the most appropriate control type. However you can select another control type to be created for a field.



: The control type must match the field type. Otherwise, the control cannot work properly with the field data. If you create a control that does not match the data field, you have to update the data field in the DAC (see [To Customize a Field on the DAC Level](#) for details) or in the graph. (See [To Customize a Field on the Graph Level](#) for details.)

7. If you need to change the type of the control to be created, select the needed type in the **Control** column, as shown in the following screenshot.

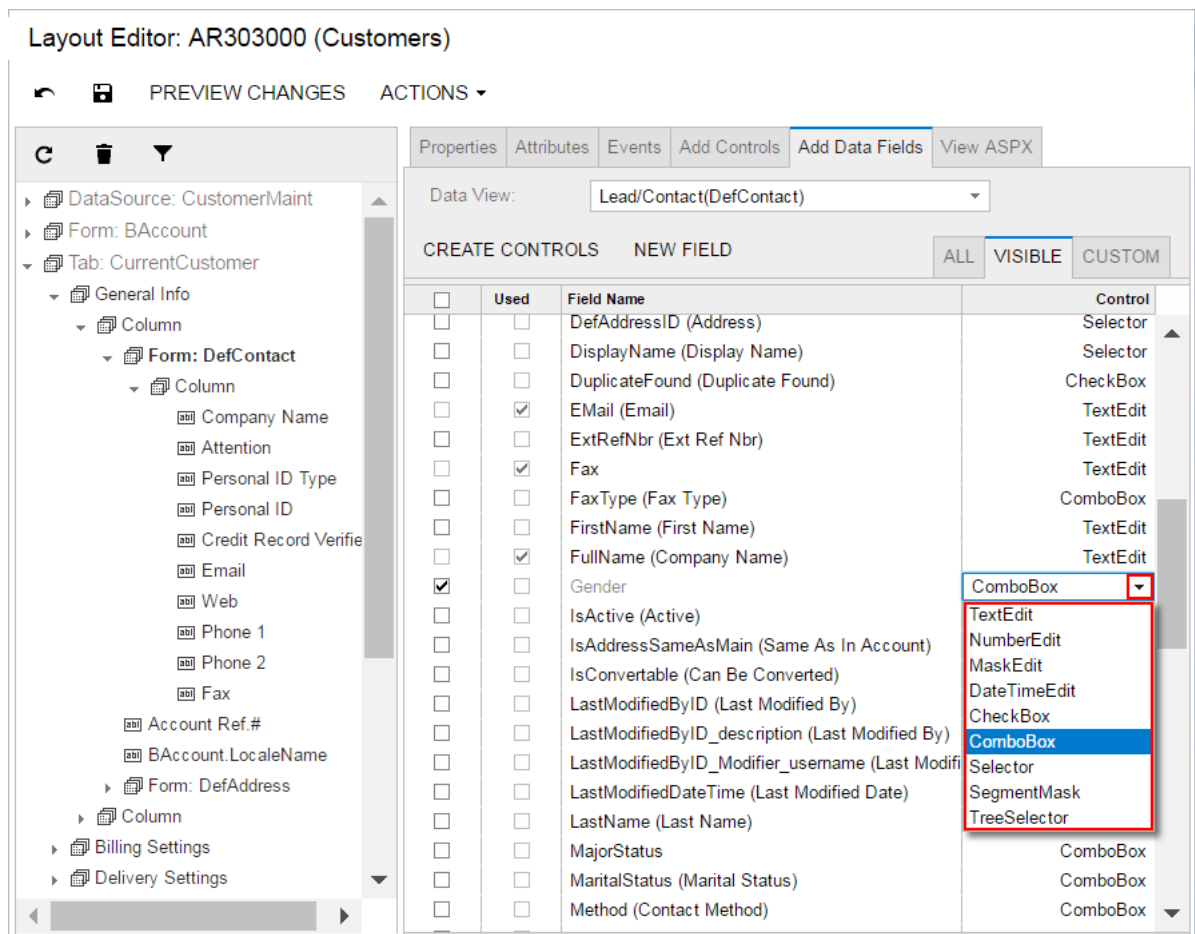


Figure: Selecting a control type for a data field

8. On the list toolbar, click **Create Controls**.

The platform creates a box for the selected data field and adds to the Control Tree a node for the box.

9. Click **Save** to save your changes to the customization project.

Layout Rule (PXLayoutRule)

The layout of a form is organized into a table of controls, with each control located within a certain column and row. The columns and rows are defined by `PXLayoutRule` components added to the layout of the form. The properties of the `PXLayoutRule` component provide the position of the underlying controls and their size and appearance in the UI.

You can use the `PXLayoutRule` component to do the following to manage the layout of UI elements within a container:

- Place controls in multiple rows and columns to uniformly distribute them on the form or tab area of a form (see the diagram below)
- Cause controls to span multiple columns
- Merge controls into one row of the column to align them horizontally
- Adjust the widths of controls and labels
- Hide the labels of controls
- Group controls for users' convenience

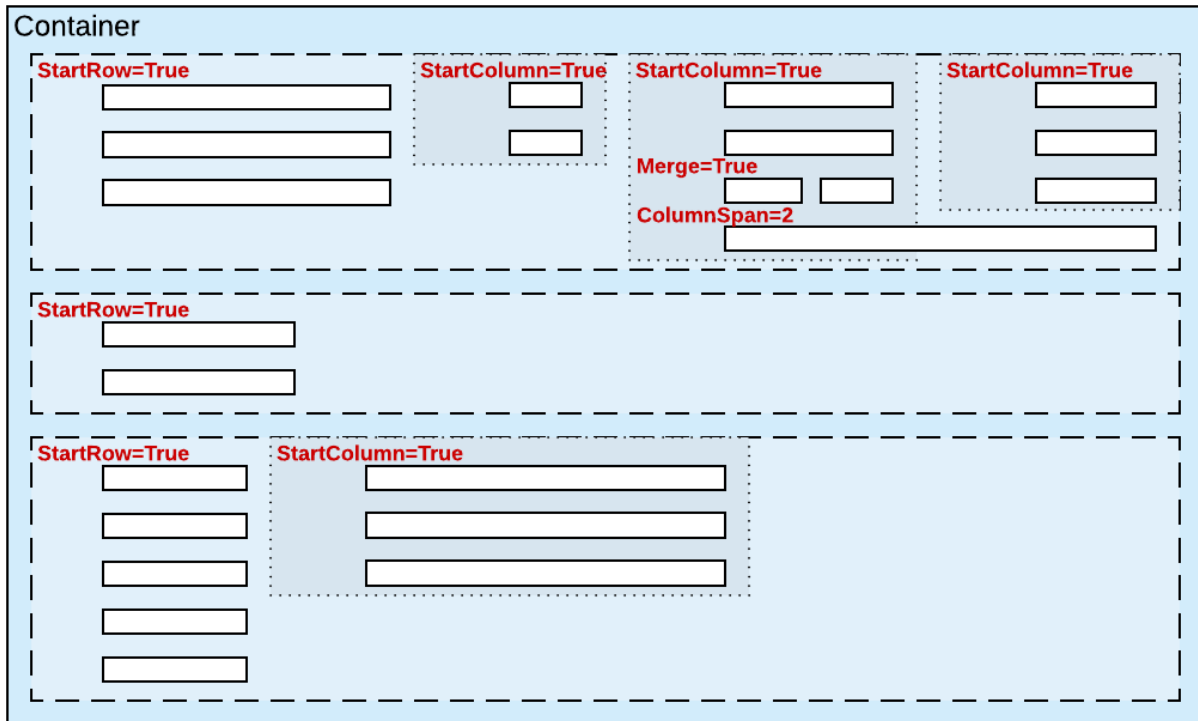


Figure: Layout rules to arrange controls within a container

A layout rule, as the diagram below shows, can be immediately added to the following types of containers:

- `PXFormView`
- `RowTeplate`
- `PXTabItem`
- `PXSmartPanel`

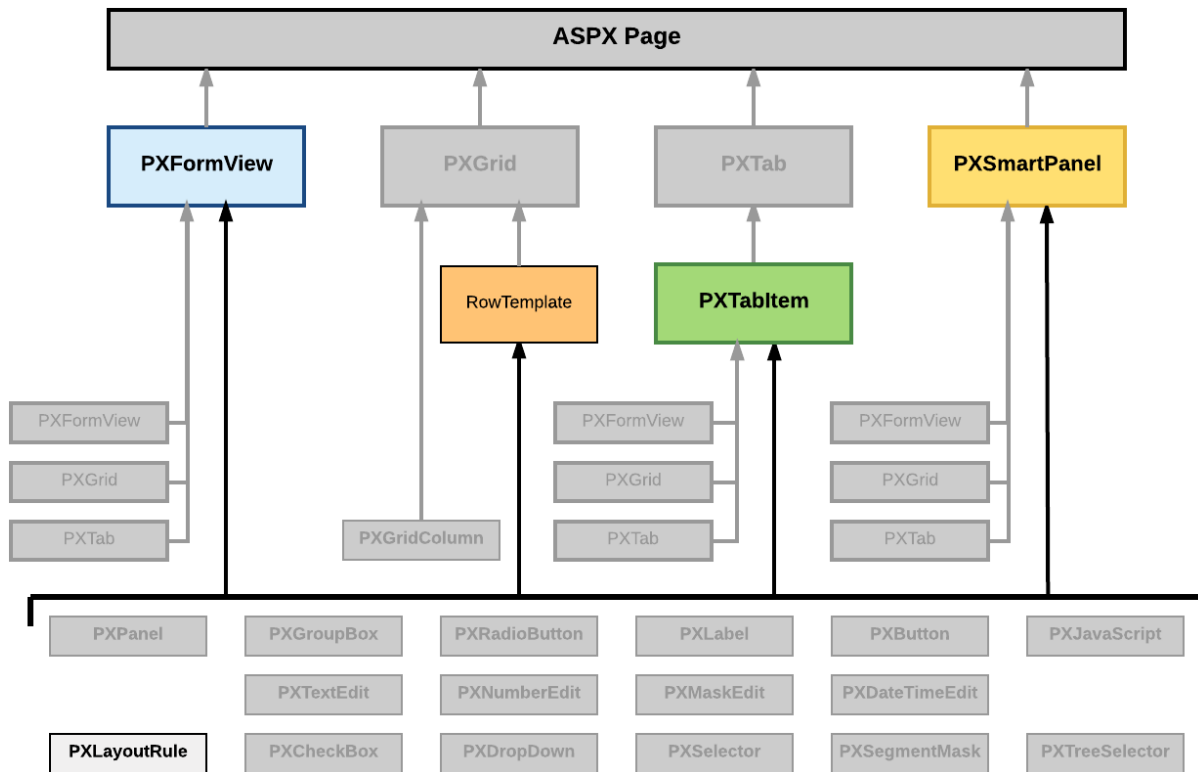


Figure: Nesting rules for a layout rule in an ASPX page

The Acumatica Customization Platform supports the following properties for a `PXLayoutRule` component.

Property	Description
ColumnSpan	Specifies the number of columns spanned by a control placed below the target <code>PXLayoutRule</code> component. This property applies to a single control that is under the layout rule in the ASPX code. (See Using the ColumnSpan Property for details.)
ColumnWidth	Defines the width (in pixels) of a column containing controls. You can set the property to a predefined value (see Using Predefined Size Values for details) or to a value in pixels. The specified width is applied to the column and is not changed when the <code>LabelsWidth</code> property value is specified for the same <code>PXLayoutRule</code> component and when the <code>Size</code> , <code>Width</code> , or <code>LabelWidth</code> property value is specified for a control contained within a column. (See Using the ColumnWidth, ControlSize, and LabelsWidth Properties for details.)
ControlSize	Defines the width for the controls placed within a column. The <code>ControlSize</code> property value is assigned from the predefined list of items. (See Using Predefined Size Values for details.) The specified size is applied to all controls contained within the column if you do not override the size separately for a control by specifying the <code>Size</code> or <code>Width</code> property values. (See Using the ColumnWidth, ControlSize, and LabelsWidth Properties for details.)
GroupCaption	Specifies the caption for the group of controls that is started from the control placed under this rule in the ASPX code. Any rule that has the <code>GroupCaption</code> property value set requires a rule with the <code>EndGroup</code> property value set to <code>True</code> below the last control of the group in the code. All controls between the starting and ending group rules are included in the group. (See Using the GroupCaption, StartGroup, and EndGroup Properties for details.)
EndGroup	Indicates that the control above the rule in the ASPX code is the last control in the group that starts from the <code>PXLayoutRule</code> component with a specified

Property	Description
	GroupCaption property value. (See Using the GroupCaption, StartGroup, and EndGroup Properties for details.)
LabelsWidth	Defines the width (in pixels) of the control labels placed within a column. You select the LabelsWidth property value from the predefined list of items (see Using Predefined Size Values for details) or by typing the width in pixels. The specified size is applied to all control labels contained within the column if you do not override the size separately for a control by specifying the LabelWidth property value. (See Using the ColumnWidth, ControlSize, and LabelsWidth Properties for details.)
Merge	Is used to merge controls into one row and to horizontally align the controls that are placed under this rule in the ASPX code. The first PXLayouRule component added under this rule in the code stops the merging, and the next control is placed alone beneath the merged controls as the leftmost control of the current column. To cancel merging for all controls that follow, you must add the PXLayouRule component without the adjusted property value. (See Using the Merge Property for details.)
StartGroup	Starts the group of controls from the control placed under this rule in the ASPX code. If the GroupCaption property value is not set to True, the group is displayed without a caption. Any rule that has the StartGroup property value set requires a rule with the EndGroup property value set to True below the last control of the group in the code. All controls between the starting and ending group rules in the code are included in the group. (See Using the GroupCaption, StartGroup, and EndGroup Properties for details.)
StartRow	If set to True, starts a new row for the controls following this rule. (See Using the StartRow and StartColumn Properties for details.)
SuppressLabel	Hides all control labels within a column. All control labels placed within the column are hidden if you do not override them separately for a control by specifying a SuppressLabel property value. (See Using the SuppressLabel Property for details.)

By specifying the values of these properties, you can make a variety of changes to the UI. Some properties affect only the next control under the PXLayouRule component in the ASPX code. Other properties affect all controls under the rule until the next rule is encountered in the code. Still other properties require a corresponding ending rule. For instance, the rule with the GroupCaption property value specified requires that a corresponding rule be set with the EndGroup property value. All controls in the code between the GroupCaption and EndGroup rules become part of the group.

To create a layout rule in a container, follow the instructions described in [To Add a Layout Rule](#).

To delete a layout rule from a container, perform the actions described in [To Delete a Child UI Element](#).

For detailed information on working with the content of a grid container, see the following topics:

- [To Select a Layout Rule in the Layout Editor](#)
- [To Set a Layout Rule Property](#)

To Select a Layout Rule in the Layout Editor

To start the customization of a layout rule, you have to select it in the [Layout Editor](#). To do this, perform the following actions.

1. Open the form that contains the container with the layout rule, so that the container is displayed on the screen.
2. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).
3. On the form, click the area of the container to be customized, to open the [Element Properties Dialog Box](#) for the container.

4. In the dialog box, click **Customize**.
5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one. (See [To Select an Existing Project](#) and [To Create a New Project](#) for details.)

If the customization project does not contain a changeset for the form, the [Customization Project Editor](#) adds to the project a *Page* item for the form to keep in the database the changeset to the ASPX code of the form. The container is opened in the [Layout Editor](#) and selected in the Control Tree of the editor.

6. If the required layout rule is not displayed in the Control Tree, expand the subnodes of the container node to display the rule.
7. In the Control Tree, select the layout rule to start the customization.

When you click **Save** on the editor toolbar, the editor updates the *Page* item in the database.

To Set a Layout Rule Property

To include in a customization project changes to properties of a layout rule, you have to modify the properties by using the [Layout Editor](#). To start setting the properties of a layout rule, perform the following actions:

1. Select the layout rule in the Layout Editor, as described in [To Select a Layout Rule in the Layout Editor](#).
2. Click the **Properties** tab item to open the list of properties for the rule. (See the following screenshot.)

Layout Editor: AR303000 (Customers)

PREVIEW CHANGES ACTIONS

Properties Attributes Events Add Controls Add Data Fields View ASPX

DataSource: CustomerMaint
Form: BAccount
Column
Column
Column
Tab: CurrentCustomer
Dialogs

Override	Property	Value
	Base Properties	
<input type="checkbox"/>	ColumnSpan	
<input type="checkbox"/>	ColumnWidth	
<input type="checkbox"/>	ControlSize	M
<input type="checkbox"/>	EndGroup	
<input type="checkbox"/>	GroupCaption	
<input type="checkbox"/>	LabelsWidth	SM
<input type="checkbox"/>	Merge	
<input type="checkbox"/>	StartColumn	True
<input type="checkbox"/>	StartGroup	
<input type="checkbox"/>	StartRow	
<input type="checkbox"/>	SuppressLabel	
	Ext Properties	
<input type="checkbox"/>	ClientIDMode	
<input type="checkbox"/>	ValidateRequestMode	
<input type="checkbox"/>	ViewStateMode	

Figure: Viewing the list of layout rule properties in the Layout Editor

3. Specify values for the required properties.
4. Click **Save** to save changes in the customization project.

The most important properties of the `PXLayoutRule` component are described in the following topics:

- [Using the StartRow and StartColumn Properties](#)
- [Using the ColumnWidth, ControlSize, and LabelsWidth Properties](#)

- [Using Predefined Size Values](#)
- [Using the ColumnSpan Property](#)
- [Using the Merge Property](#)
- [Using the GroupCaption, StartGroup, and EndGroup Properties](#)
- [Using the SuppressLabel Property](#)

Using the StartRow and StartColumn Properties

By default, the system places all the controls of a container into a column within the first row, as shown in the diagram below. To do this, the system initially sets to *True* the `StartRow` property value for the uppermost `PXLayoutRule` component in a container.

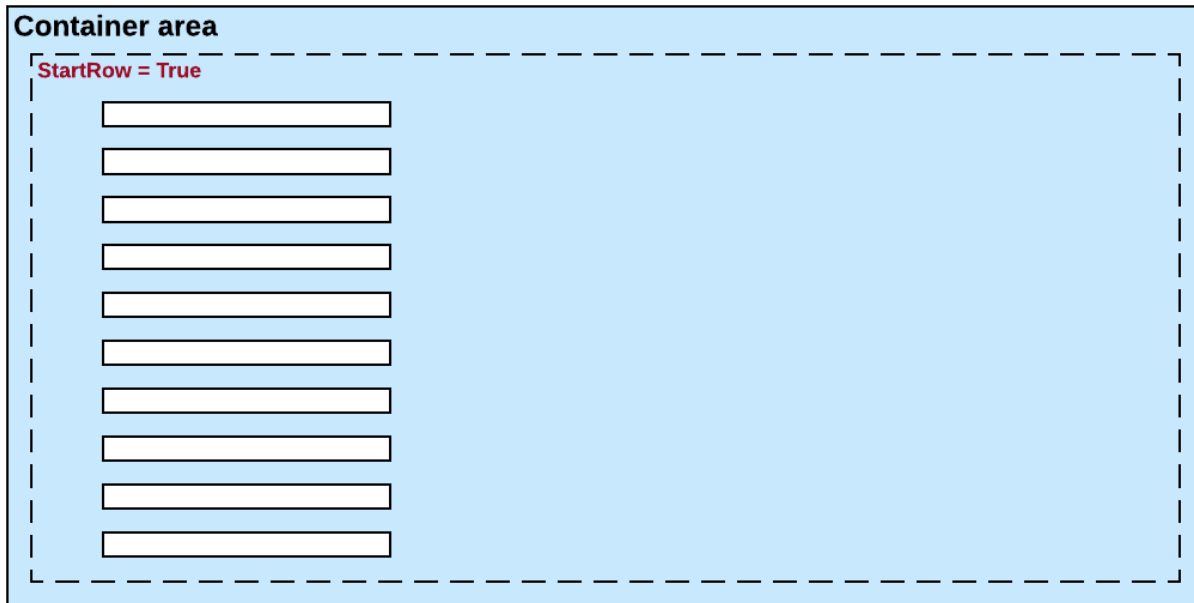


Figure: Viewing the default layout of controls of a container on a form

The controls are placed within a single column until you add a layout rule with the `StartColumn` or `Merge` property value set to *True*.



Important: For the proper layout, the `StartRow` property value must be set to *True* for the uppermost `PXLayoutRule` component of a container.

To best use the area of a form or tab item area, you can place controls in multiple columns within a row by adding the `PXLayoutRule` components with the `StartColumn` property value set to *True*. This property creates a new column of controls within the current row, as the following diagram shows.

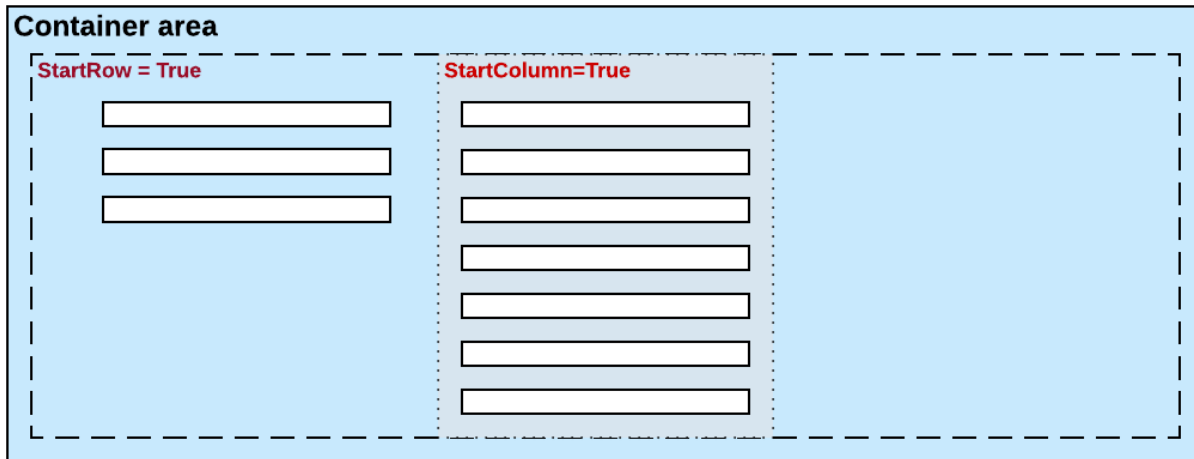


Figure: Creating a new column by adding a single PXLayouRule component

The first control under this rule corresponds to the highest control in the column.

Every new `PXLayouRule` component that has the `StartRow` property value set to `True` initializes a new independent placeholder of controls, which are placed in a single column by default. To place controls in multiple columns within the new row, you should include in the placeholder a new layout rule with the `StartColumn` property value set to `True`, as shown in the following diagram.

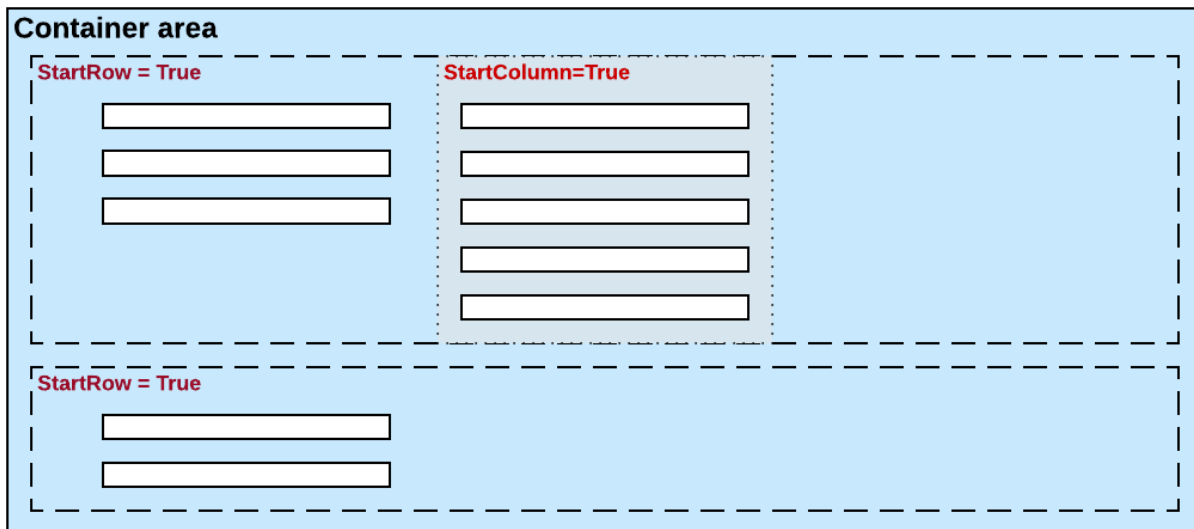


Figure: Creating a new row by adding a single PXLayouRule component

Because the values of the `ColumnWidth`, `ControlSize`, and `LabelsWidth` properties are never inherited from the previously declared `PXLayouRule` component, you might need to define these properties exclusively for every new row and column.

Using the `ColumnWidth`, `ControlSize`, and `LabelsWidth` Properties

You can use the `PXLayouRule` components to define the sizes for every control (that is, its input area) and its label within a column, group, or merged set of controls. Every `PXLayouRule` component that has the `StartRow` or `StartColumn` property value set to `True` must have one of the following sets of properties defined:

- `LabelsWidth` and `ControlSize`
- `LabelsWidth` and `ColumnWidth`

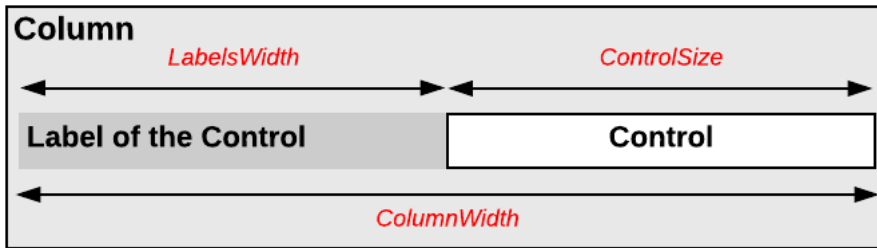


Figure: Using layout rule properties to define control sizes



: You should not set both `ColumnWidth` and `ControlSize` property values for the same `PXLayoutRule` component; in this case, the system will use the value of the `ControlSize` property.

Please note the following points about setting the sizes of controls and their labels:

1. The values of the `ColumnWidth`, `ControlSize`, and `LabelsWidth` properties must be defined exclusively for every `PXLayoutRule` component; they are never inherited from the previously declared one.
2. You can change the size of a single control or its label by defining values for the `Size`, `Width`, and `LabelsWidth` properties of the control. Property values that are set for a control have a higher priority than the properties of the `PXLayoutRule` component.
3. You can assign a predefined size abbreviation (such as `XXS`, `L`, or `XL`) for the `ColumnWidth`, `LabelsWidth`, and `ControlSize` properties of a layout rule and the `LabelsWidth` and `Size` properties of a control. (See [Using Predefined Size Values](#) for details.)
4. The `PXDateTimeEdit` and `PXNumberEdit` control types have a predefined `Width` property value, which you cannot change by setting the `ColumnWidth` or `ControlSize` property values for the appropriate `PXLayoutRule` component. To change the width of this control, set a value for the `Size` or `Width` property of the control.

Using Predefined Size Values

You can use the following predefined values for the `ColumnWidth`, `LabelsWidth`, and `ControlSize` properties of the `PXLayoutRule` component and the `LabelsWidth` and `Size` properties of a control.

Predefined Value	ColumnWidth	LabelsWidth and ControlSize of a Layout Rule; LabelsWidth and Size Properties of a Control
<code>XXS</code>	100px	40px
<code>XS</code>	150px	70px
<code>S</code>	200px	100px
<code>SM</code>	-	150px
<code>M</code>	250px	200px
<code>XM</code>	300px	250px
<code>L</code>	350px	300px
<code>XL</code>	400px	350px
<code>XXL</code>	450px	400px

Note the following points about setting the predefined sizes of controls and their labels:

- For any property for which there are predefined values, you can specify a value in pixels, such as `55px`. (This format is mandatory if you don't use abbreviations, because the property value can be defined only in pixels.)

- There is no predefined value for the `Width` property of a control. Therefore, you can specify a value for this property by typing any value in pixels, such as `55px`. Before specifying the `Width` property value for a control, you must define the `Size` property value for the control as `Empty`.



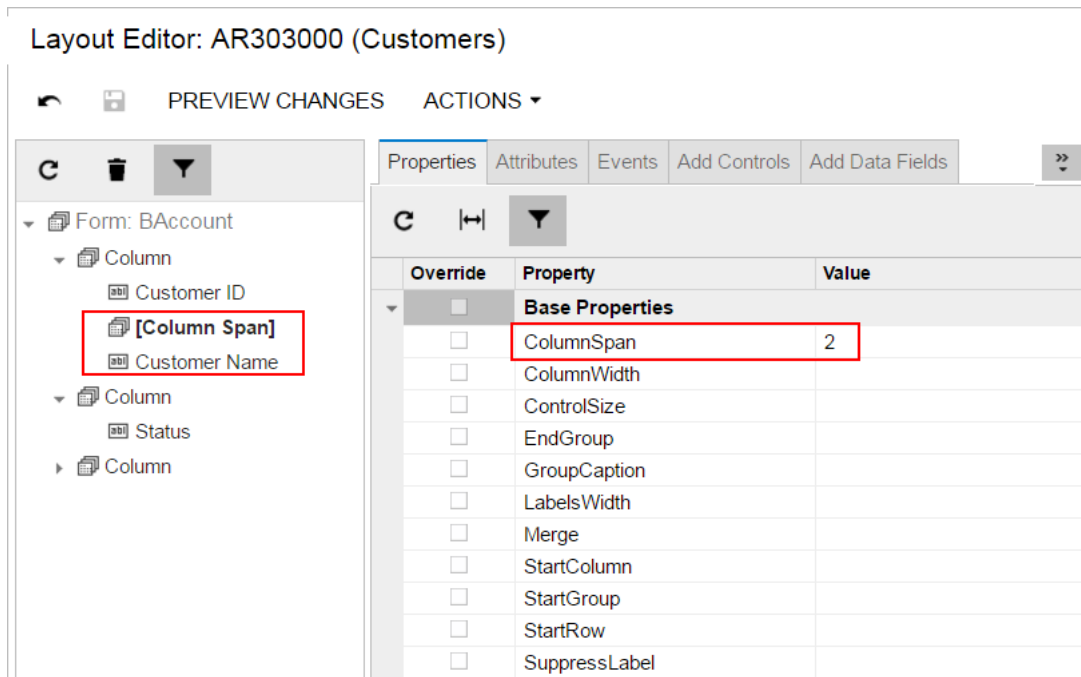
: The `Width` property is declared in ASP.NET. The `Size` property is declared in Acumatica Framework so you can use the predefined values.

Using the `ColumnSpan` Property

You specify the `ColumnSpan` property value for a `PXLayoutRule` component by manually typing the number of columns spanned by the first control placed below the rule.

For example, the form container on the [Customers](#) (AR.30.30.00) form has three columns of boxes, and there is a layout rule with the `ColumnSpan` property set to 2 in the first column, as the following screenshot shows.

Figure: Viewing the `ColumnSpan` property in the Layout Editor



This property forces the system to make the box spans two columns, as shown in the following screenshot.

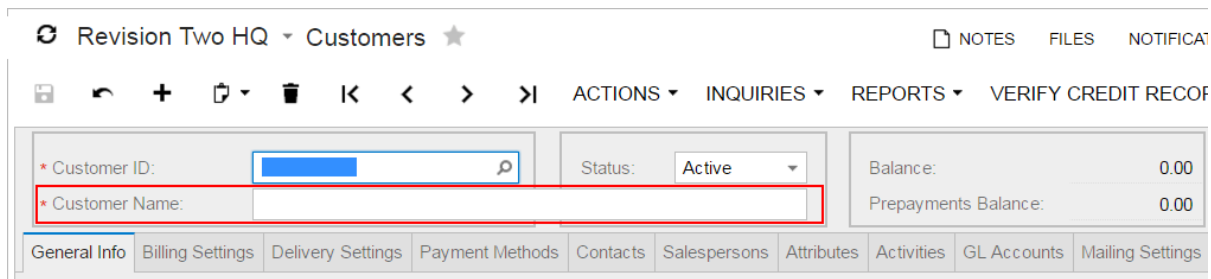


Figure: Viewing the box that spans two columns on the form

A `PXLayoutRule` component with the `ColumnSpan` property value specified is handled as follows:

- The `LabelsWidth` property value is always inherited from the previously declared `PXLayoutRule` component that has the `StartRow` or `StartColumn` property value set to `True`.

- If a value for the `ColumnWidth` or `ControlSize` property is specified for the component, the value is ignored.

Using the Merge Property

Merging means placing controls so that they are horizontally aligned. Horizontal alignment is performed for the controls that are placed between a layout rule with the `Merge` property set to `True` and any other subsequent layout rule. Therefore, to cancel merging for all of the following controls, you have to add a `PXLayoutRule` component, with or without the `Merge` property specified.

For example, the **Billing Settings** tab item on the *Customers* (AR.30.30.00) form has three pairs of merged check boxes in the **Print and Email Settings** group, as the following screenshot shows.

Figure: Using the Merge property for two boxes

The screenshot shows the Layout Editor interface for the AR303000 (Customers) form. The left pane displays the tree structure of the form, with the **Billing Settings** tab selected. Under **Billing Settings**, there are three groups: **Print Invoices**, **Send Invoices by Email**, and **Print and Email Settings**. The **Print and Email Settings** group is highlighted with a red box, and its **Merge** property is set to `True`. The right pane shows the Properties window for the selected group, with the **Merge** property set to `True`.

Override	Property	Value
Base Properties		
<input type="checkbox"/>	ColumnSpan	
<input type="checkbox"/>	ColumnWidth	
<input type="checkbox"/>	ControlSize	M
<input type="checkbox"/>	EndGroup	
<input type="checkbox"/>	GroupCaption	
<input type="checkbox"/>	LabelsWidth	M
<input type="checkbox"/>	Merge	True
<input type="checkbox"/>	StartColumn	
<input type="checkbox"/>	StartGroup	
<input type="checkbox"/>	StartRow	
<input type="checkbox"/>	SuppressLabel	
Ext Properties		
<input type="checkbox"/>	ClientIDMode	
<input type="checkbox"/>	ValidateRequestMode	
<input type="checkbox"/>	ViewStateMode	

This property forces the system to render the boxes in one column, as shown in the following screenshot.

Revision Two HQ ▾ Customers ★

NOTES FILES NOTIFICATIONS

ACTIONS ▾ INQUIRIES ▾ REPORTS ▾ VERIFY CREDIT RECORD

* Customer ID: Status: Active Balance: 0.00
 * Customer Name: Prepayments Balance: 0.00

General Info **Billing Settings** Delivery Settings Payment Methods Contacts Salespersons Attributes Activities GL Accounts Mailing Settings SIM C

BILL-TO CONTACT Same as Main
 Company Name:
 Attention:
 Phone 1:
 Phone 2:
 Fax:
 Email:
 Web:

BILL-TO ADDRESS Same as Main
 Address Line 1:

PARENT INFO
 Parent Account:
 Consolidate Balance
 Consolidate Statements
 Share Credit Policy

PRINT AND EMAIL SETTINGS
 Send Invoices by Email Print Invoices
 Send Dunning Letters by Email Print Dunning Letters
 Send Statements by Email Print Statements

Statement Type:
 Multi-Currency Statements

DEFAULT PAYMENT METHOD

Figure: Viewing the boxes merged in a single column on the form

A `PXLayoutRule` component with the `Merge` property value set to `True` is handled as follows:

- If the `ColumnWidth` property value is set for the same `PXLayoutRule` component, the value is ignored.
- The default values for the `ControlSize` and `LabelsWidth` properties are inherited from the previously declared `PXLayoutRule` component with the `StartRow` or `StartColumn` property value set to `True`. You can override these property values if necessary by specifying the `ControlSize` and `LabelsWidth` property values from the predefined list of options. (See [Using Predefined Size Values](#) for details.)

Using the `GroupCaption`, `StartGroup`, and `EndGroup` Properties

You can organize controls in a container within groups to make users' work more logical. To group multiple controls within a column, generally you have to add two `PXLayoutRule` components that have the following properties set to define the first and the last controls in the group, respectively:

- `GroupCaption` and `EndGroup`—to create a group with the caption specified in the `GroupCaption` property
- `StartGroup` and `EndGroup`—to create a group without a caption



: You can specify both the `GroupCaption` property and the `StartGroup` property for the `PXLayoutRule` component that starts a group.

For example, by specifying the `GroupCaption` property value for the corresponding `PXLayoutRule` components placed above a control, you start the group of controls and set up the header for the group. You should also add a `PXLayoutRule` component with the `EndGroup` property value set to `True` below (in the code) the last control that is included in the group.

The system works as follows for all `PXLayoutRule` components with the `GroupCaption` or `StartGroup` property value specified:

- If the `GroupCaption`, `StartGroup`, or `EndGroup` property is set for a `PXLayoutRule` component, the system ignores the `ColumnWidth` property value specified for the same component.
- The default values for the `ControlSize` and `LabelsWidth` properties are inherited from the previously declared `PXLayoutRule` component with the `StartRow` or `StartColumn` property value set to `True`. You can override these property values if necessary by specifying the `ControlSize`

and `LabelsWidth` property values in the layout rule that starts a group. (See [Using Predefined Size Values](#) for details.)

You end a group by using a `PXLayoutRule` component with a `GroupCaption`, `StartGroup`, or `EndGroup` property specified. Therefore, if there is another group that starts immediately below a group, you can omit the layout rule that ends the upper group, as shown in the third column of the row displayed in the example in following diagram.

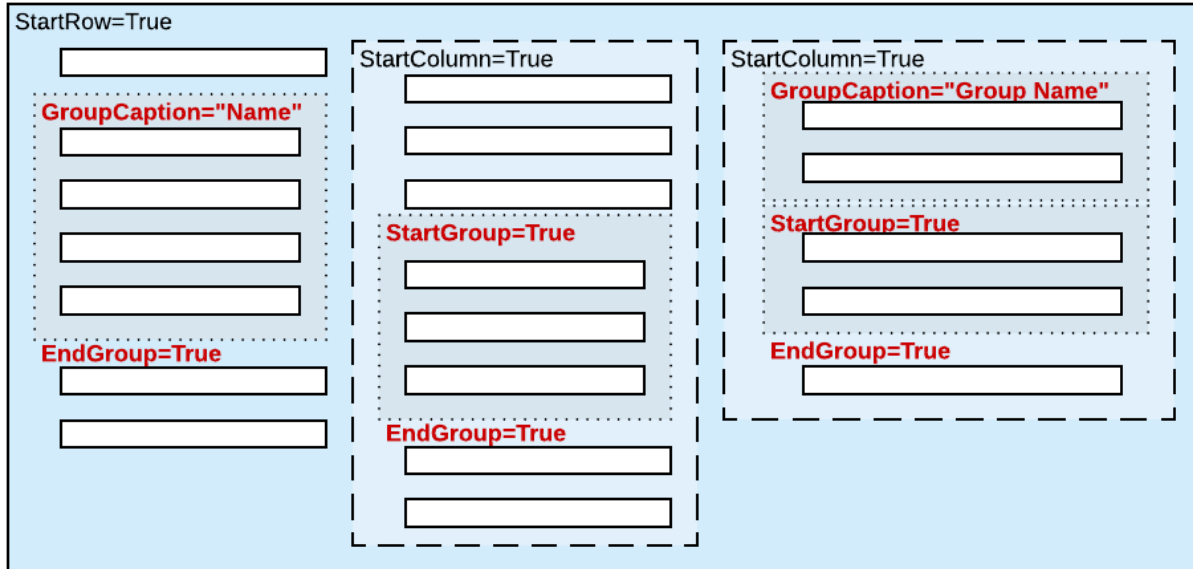


Figure: Possible use of layout rules with grouping properties

Using the SuppressLabel Property

Every control for a data field contains both a label and the input area of the control. The label is displayed left of the input area, except with check boxes. The label of a check box is displayed right of the input area of the check box. When you add a check box onto a form, it is automatically aligned with other input controls within the appropriate column. As a result, the area left of a check box is empty.

To hide labels of the controls placed within a column, you should set the `SuppressLabel` property value of the `PXLayoutRule` component of the column to `True`. Then within the column, all check boxes are placed without any space to the left of the input control.



: If needed, you can align a check box to the left of the column by setting to `True` the `AlignLeft` property of the control. Also, you can set the `SuppressLabel` property value to `True` for any other control to hide its label.

The `SuppressLabel` property affects all controls of the group that are placed under the `PXLayoutRule` component with the `True` value of this property. The `SuppressLabel` property value must be defined for every `PXLayoutRule` component for the controls placed beneath the component and included in the same column; this property is never inherited from the previously declared property.



: The `SuppressLabel` property value is never applied to `PXLayoutRule` components with the `ColumnSpan` property value specified.

For example, the **Parent Info** group on the **Billing Settings** tab item on the [Customers](#) (AR.30.30.00) form is displayed, as shown in the following screenshot.

Figure: Viewing a group of controls with labels

If you set the `SuppressLabel` property of the group layout rule to `True`, the labels of all controls of the group are hidden, as shown in the following screenshot.

Figure: Viewing the same group of controls after applying the `SuppressLabel` property

Panel (PXPanel)

In a container with controls for a single data record, you can use a `PXPanel` element as a container with a caption to group controls. However we recommend that you use the `PXLayoutRule` component for this purpose. (See [Layout Rule \(PXLayoutRule\)](#) for details.)

In Acumatica ERP, a panel is used as a container to display a horizontal row of buttons with right alignment in a dialog box (see [Dialog Box \(PXSmartPanel\)](#) for details), as the following screenshot shows.

Figure: Using a panel as a button container in a dialog box

If you open the dialog box displayed in the screenshot above in the Layout Editor, you can see that the panel is located below a form container and contains two buttons.

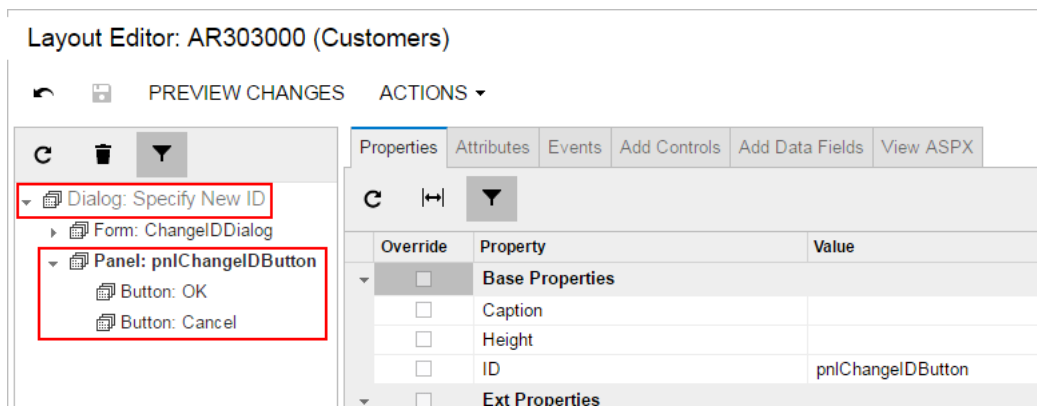


Figure: Viewing the panel content in the Layout Editor

You can add the `PXPanel` element to a dialog box and buttons to this element, as described in [To Add Another Supported Control](#) and [To Use a Button in a Dialog Box](#). In a panel, to arrange buttons in a horizontal row with right alignment, you can specify the `SkinID` property of the `PXPanel` element, as described in [Using the SkinID Property](#).

Group Box (PXGroupBox)

In a container with controls for a single data record, you can use a `PXGroupBox` element as a container with a caption to group controls. However we recommend that you use the `PXLayoutRule` component for this purpose. (See [Layout Rule \(PXLayoutRule\)](#) for details.)

In Acumatica ERP, the `PXGroupBox` element is designed to be exclusively used as a radio button container to render a drop-down data field as a set of radio buttons. It contains scripts with the logic to support a nested radio button for each value of a drop-down field.

A group box for a drop-down data field can be immediately added to the following types of containers:

- `PXFormView`
- `RowTemplate`
- `PXTabItem`

To add a `PXGroupBox` element to a container, follow the instructions described in [To Add Another Supported Control](#).

To delete a group box from a container, follow the instructions described in [To Delete a Child UI Element](#).

For detailed information on customizing a group box, see the following topics:

- [To Open a Group Box in the Layout Editor](#)
- [To Create a Group Box for a Drop-Down Field](#)
- [To Set a Group Box Property](#)

To Open a Group Box in the Layout Editor

To start the customization of a group box on a form, you have to open it in the [Layout Editor](#). To do this, perform the following actions:

1. Open the form to display the appropriate set of radio buttons in the browser.



: In Acumatica ERP, a radio button can work properly only in a `PXGroupBox` container that is used for a drop-down data field.

2. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).



: If you need to activate the *Element Inspector* for a pop-up panel, a dialog box, or another UI element that opens in modal mode and makes the *Customization Menu* unavailable for selection, you can press Control-Alt.

3. On the form, click the area with radio buttons to open the *Element Properties* dialog box for the group box.
4. In the dialog box, click **Customize**.
5. If there is no currently selected customization project and the inspector opens the *Select Customization Project Dialog Box*, select an existing customization project or create a new one. (See *To Create a New Project* and *To Select an Existing Project* for details.)

If the customization project does not contain a changeset for the form, the *Customization Project Editor* adds to the project a *Page* item for the form, to keep the changeset to the ASPX code of the form in the database. The group box is opened in the *Layout Editor*, and you can start the customization of the box and the nested radio buttons.

When you click **Save** on the editor toolbar, the editor updates the *Page* item in the database.

To Create a Group Box for a Drop-Down Field

In Acumatica ERP, the `PXDropDown` box type is generally used to display on a form a field with a list attribute, such as `PXStringList`, `PXIntList`, or `PXDecimalList`. However if you want to display such a field as a set of radio buttons, where one radio button is used to display and select each single constant value of the field, you can create a `PXGroupBox` element and include `PXRadioButton` elements in it.

To create a group box for a drop-down data field in a container, perform the following actions:

1. Open the container in the Layout Editor, as described in *To Open a Container in the Layout Editor*.
2. In the container, add a group box, as described in *To Add Another Supported Control*.
3. For the new group box, specify the `DataField` property to bind the box to the data field. See *To Set a Group Box Property* and *Using the DataField Property* for details.
4. If required, specify the `Caption` and `RenderStyle` properties of the group box, as the following screenshot shows.

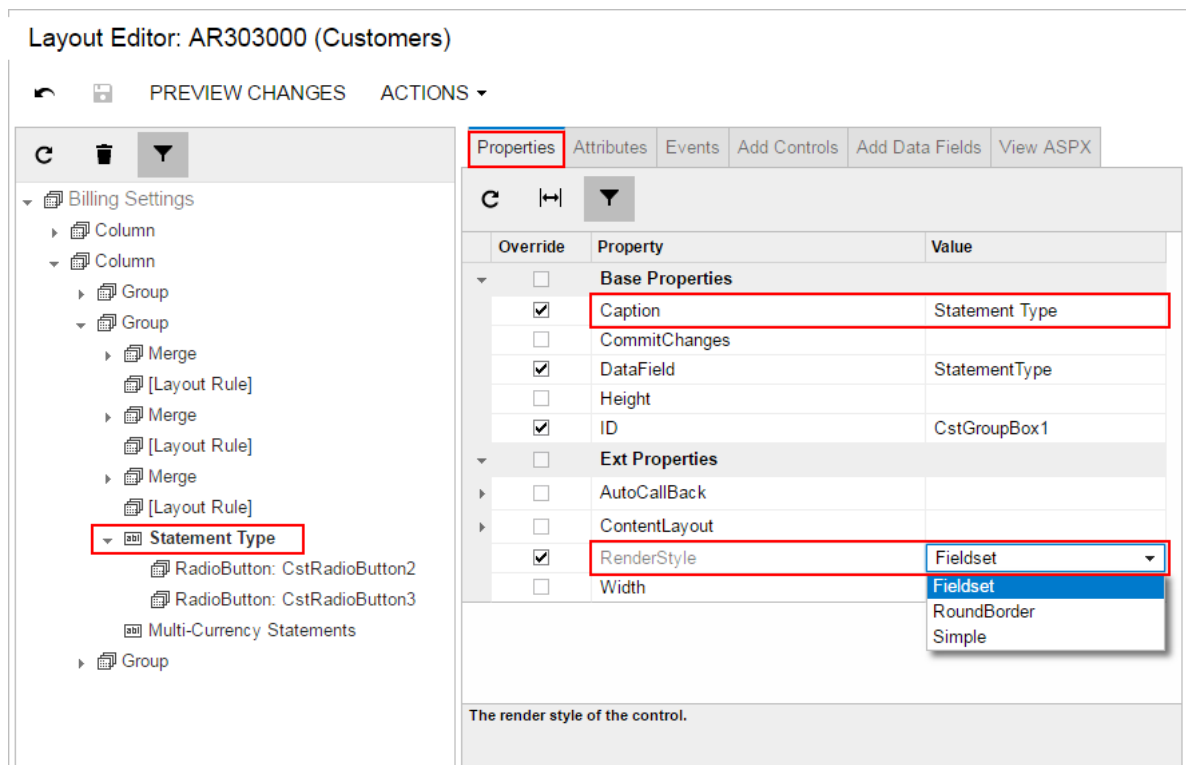


Figure: Setting the properties of a group box

(See [Using the Caption Property](#) and [Using the RenderStyle Property](#) for details.)

5. For each value defined in the field list, do the following:
 - a. To the group box node of the Control Tree, add a radio button, as described in [To Add Another Supported Control](#).
 - b. In the C# code, find the value (usually it is an one-symbol value) in the list.
 - c. On the **Properties** tab item, enter this value in the `Value` property.
6. Click **Save** to save changes in the customization project.

For example, as the result of the described actions for the `StatementType` data field with the `O` and `B` values defined in the list, you might get the following ASPX code.

```
<px:PXGroupBox runat="server" ID="CstGroupBox1" DataField="StatementType"
  Caption="Statement Type" RenderStyle="Fieldset">
  <Template>
    <px:PXRadioButton runat="server" ID="CstRadioButton2" Value="O" />
    <px:PXRadioButton runat="server" ID="CstRadioButton3" Value="B" />
  </Template>
</px:PXGroupBox>
```

To Set a Group Box Property

To include changes to the properties of a group box in a customization project, you have to modify the properties by using the [Layout Editor](#). To start setting the properties of a group box, perform the following actions:

1. Select the group box in the Layout Editor, as described in [To Open a Group Box in the Layout Editor](#).
2. Click the **Properties** tab item to open the list of properties for the group box.

3. Specify values for the required properties.
4. Click **Save** to save your changes to the customization project.

To bind a group box to a data field, you have to specify the `DataField` property, as described in [Using the DataField Property](#).

To set a caption for a group box, you use the `Caption` property. (See [Using the Caption Property](#) for details.)

To define the style of the group box on the form, you have to select a value of the `RenderStyle` property, see [Using the RenderStyle Property](#) for details.

Using the DataField Property

You use a group box to display a data field with a list attribute as a set of radio buttons, where one radio button is used to display and select each single constant value of the field. To bind a group box to a data field, you have to specify the name of the data field in the `DataField` property of the `PXGroupBox` element in the ASPX code, as follows.

```
<px:PXGroupBox ... DataField="<Field Name>" ...>
```



Attention: The group box must contain a radio button for each value defined in the list of the field.

In the `DataField` property of the `PXGroupBox` element, you can specify the name of a data field that is accessible through another data view of the same graph. See [Using Multiple Data Views for Boxes in a Container](#) for details.

Using the Caption Property

You can define a caption for a group box by using the `Caption` property of the `PXGroupBox` element in the ASPX code as follows.

```
<px:PXGroupBox ... Caption="Example of Group Box Caption" ...>
```

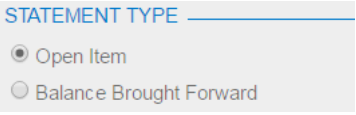
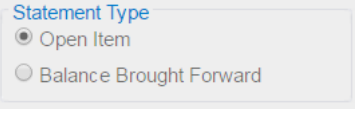
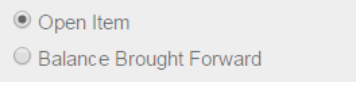
If the `RenderStyle` property of a `PXGroupBox` element is set to *Simple*, the `Caption` property is ignored. See [Using the RenderStyle Property](#) for details.

Using the RenderStyle Property

To define the style of a group box on the form, you have to select a value of the `RenderStyle` property of the `PXGroupBox` element in the ASPX code, as follows.

```
<px:PXGroupBox ... RenderStyle="StyleName" ...>
```

The Acumatica Customization Platform supports the following `RenderStyle` values for the `PXGroupBox` element.

Name	Description	Example
Fieldset	Indicates that the group of radio buttons can be displayed with a caption in the same style as in a grouping layout rule.	
RoundBorder	The default value. Indicates that the group of radio buttons can be displayed with a caption in a rounded border.	
Simple	Indicates that the group of radio buttons can be displayed without a caption and border.	

Label (PXLabel)

In a container with controls for a single data record, you can use a `PXLabel` element to display text. For example, you should use a label to render on a form static text that is stored in the ASPX code.

A `PXLabel` element can be immediately added to the following types of containers:

- `PXFormView`
- `RowTemplate`
- `PXTabItem`
- `PXSmartPanel`

You can view the use of the `PXLabel` elements, for example, in the **Aging Settings** group on the Statement Cycles (AR.20.28.00) form, shown in the following screenshot.

The screenshot shows the Acumatica user interface for the Accounts Receivable Statement Cycles form. The left sidebar has a search bar and a list of menu items, with 'Statement Cycles' highlighted. The main content area is divided into sections: 'GENERAL SETTINGS', 'AGING SETTINGS', and 'FINANCE CHARGE SETTINGS'. In the 'AGING SETTINGS' section, there is a table with three columns: 'Aging Period', 'Days Due', and 'Message Description'. The 'Aging Period' column contains values 1, 2, 3, and 4. The 'Days Due' column contains values 0, 0, 0, and 'Over Days'. The 'Message Description' column contains asterisks and input fields. In the 'FINANCE CHARGE SETTINGS' section, there is a checked checkbox for 'Apply Overdue Charges' and an 'Overdue Charge ID' field.

Figure: Viewing the labels on the Statement Cycles form

The use of a label can be required when a field contains a quantity, but you need to add a unit type for the value. For example, if you need to display a field as **Duration** [value] **Days**, where *Duration* is the value of the `DisplayName` parameter of the `PXUIField` attribute of the field, you have to add a `PXLabel` element that contains the *Days* string constant. To arrange the label in the same column as the field, you need to use the `Merge` layout rule, as described in [To Set a Layout Rule Property](#) and [Using the Merge Property](#).

You can add the `PXLabel` element to a container, as described in [To Add Another Supported Control](#).

To delete a label from an ASPX page, follow the instructions described in [To Delete a Child UI Element](#).

Radio Button (PXRadioButton)

In Acumatica ERP, a radio button can work properly only in a `PXGroupBox` container. (See [Group Box \(PXGroupBox\)](#) for details.) A `PXGroupBox` element contains scripts to provide the appropriate logic to use a nested radio button for a single constant value of a bound drop-down data field.

To use a `PXRadioButton` element on a form, follow the instructions described in [To Create a Group Box for a Drop-Down Field](#).

To bind a radio button to a value from the list of a drop-down data field, follow the instructions described in [To Bind a Radio Button to a Value in the List of a Data Field](#).

To Bind a Radio Button to a Value in the List of a Data Field

In the parent group box, to bind a radio button to a value in the list of a drop-down data field that is bound to this group box, perform the following actions:

1. Open the group box in the Layout Editor, as described in [To Open a Group Box in the Layout Editor](#).
2. In the Control Tree, select the node of the radio button.
3. Click the **Properties** tab item to open the list of properties for the radio button.
4. Set the `Value` property to an appropriate value (usually it is a one-symbol value) of the list defined in the C# code, as the following screenshot shows.

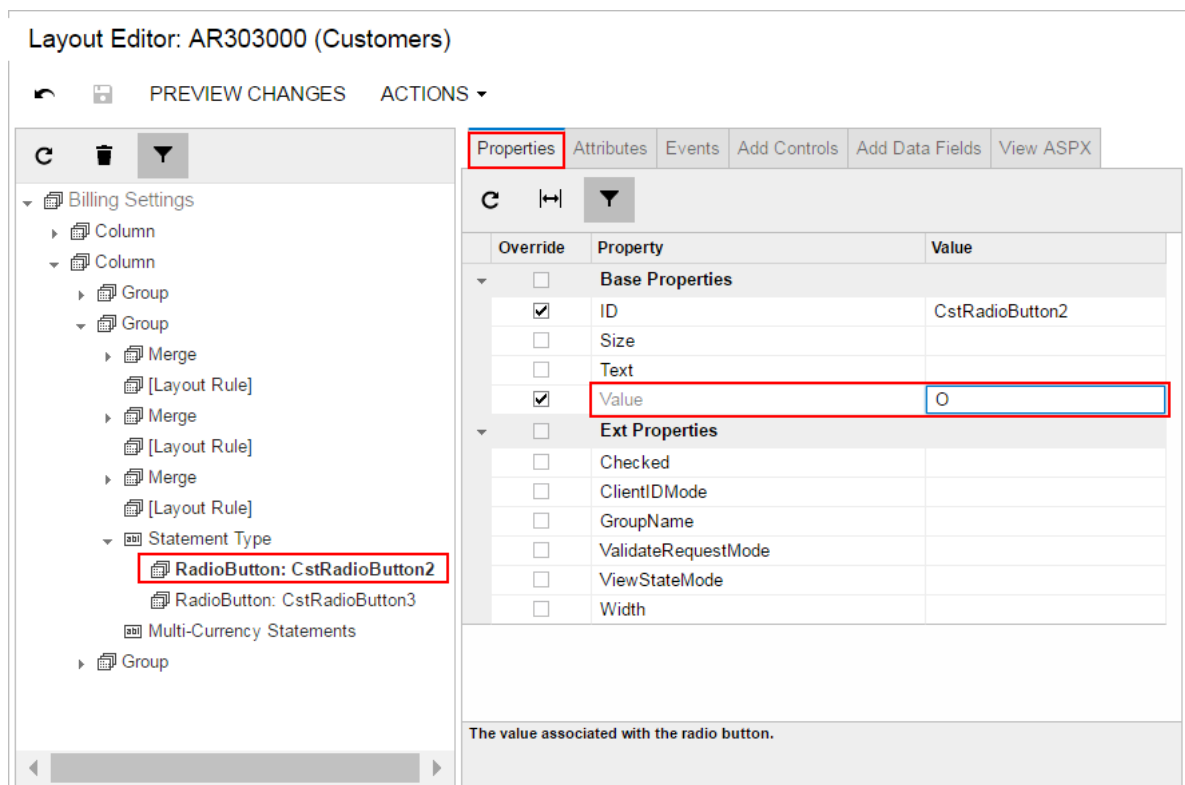


Figure: Setting the Value property of a radio button

5. Click **Save** to save your changes to the customization project.

Button (PXButton)

In Acumatica ERP, the `PXButton` element is used for the following purposes:

- In a dialog box—to display a button that closes the dialog box and returns a predefined `DialogResult` value
- In a container with controls for a single data record (including a dialog box) of a form—to display a button that invokes a method defined in the graph, which provides the business logic for the form

The most common use case of the `PXButton` element is in a dialog box.

A button, as the diagram below shows, can be immediately added to the following types of containers:

- `PXFormView`
- `RowTemplate`
- `PXTabItem`
- `PXSmartPanel`
- `PXPanel`

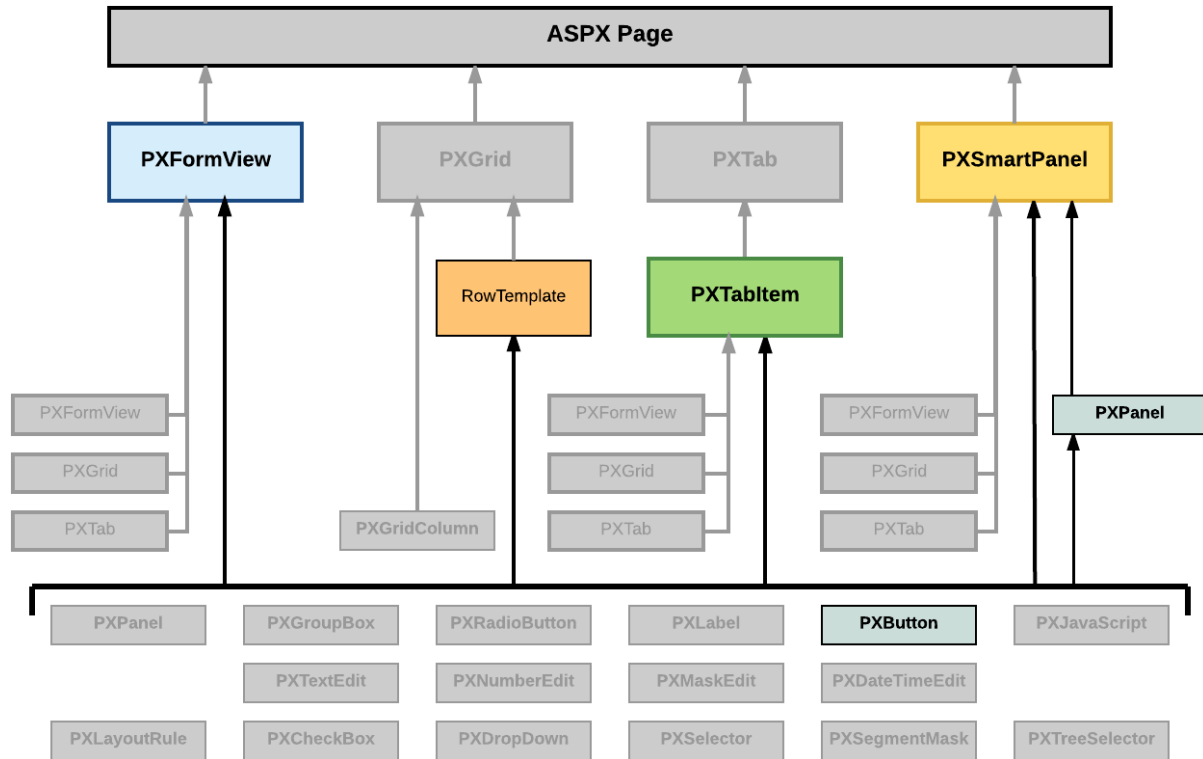


Figure: Nesting rules for a button in an ASPX page

To create a button in a container, follow the instructions described in [To Add Another Supported Control](#).

To delete a button from a container, follow the instructions described in [To Delete a Child UI Element](#).

For detailed information about using a button, see the following topics:

- [To Use a Button in a Dialog Box](#)
- [To Use a Button to Invoke a Method](#)

To Use a Button in a Dialog Box

In a dialog box, you can create buttons of the following types:

- A standard button that invokes no method but closes the dialog box and returns a specified predefined value, which can be processed by the method that has opened the dialog box



: To arrange buttons in a horizontal row with right alignment in a dialog box, we recommend that you use the `PXPanel` container with the `SkinID` property set to `Buttons`.

- A button that invokes a method of the graph that provides business logic for the form

To add a standard button to a `PXPanel` container of a dialog box, perform the following actions:

1. Add the button to a `PXPanel` container, as described in [To Add Another Supported Control](#).
2. In the Control Tree of the Layout Editor, select the node of the new button.

3. Click the **Properties** tab item to open the list of properties for the button.
4. Specify values for the following properties:
 - a. **DialogResult**: Specify the value that is returned when the button closes the dialog box. Set the property to one of the following values defined in the `WebDialogResult` enumerator.

Value	Description
<i>OK</i>	The user clicked a button with the <code>DialogResult</code> property set to <i>OK</i> .
<i>Cancel</i>	The user clicked a button with the <code>DialogResult</code> property set to <i>Cancel</i> .
<i>Abort</i>	The user clicked a button with the <code>DialogResult</code> property set to <i>Abort</i> .
<i>Retry</i>	The user clicked a button with the <code>DialogResult</code> property set to <i>Retry</i> .
<i>Ignore</i>	The user clicked a button with the <code>DialogResult</code> property set to <i>Ignore</i> .
<i>Yes</i>	The user clicked a button with the <code>DialogResult</code> property set to <i>Yes</i> .
<i>No</i>	The user clicked a button with the <code>DialogResult</code> property set to <i>No</i> .

When the user clicks the added button, the dialog box is closed, and the specified value is returned to the method that has opened the dialog box.

- b. **Text**: Specify the text to be displayed on the button.



: In the `Text` property of a standard button, you can define any text, regardless of the value of the `DialogResult` property.

5. Click **Save** to save changes in the customization project.

The following ASPX code snippet creates a panel with two standard buttons arranged a horizontal row with right alignment.

```
<px:PXPanel ... SkinID="Buttons">
  <px:PXButton ... DialogResult="OK" Text="Next" />
  <px:PXButton ... DialogResult="Cancel" Text="Return" />
</px:PXPanel>
```

If you need to include in the same dialog box a button that invokes a method of the graph, you can add the button to the same `PXPanel` container and set the button properties as described in [To Use a Button to Invoke a Method](#).

To Use a Button to Invoke a Method

You can use the `PXButton` element on a form to invoke a method of the graph that provides business logic for the form.

To include in a container a button that invokes a method of the graph, perform the following actions:

1. Open the container in the Layout Editor, as described in [To Open a Container in the Layout Editor](#).
2. Add the button to the container, as described in [To Add Another Supported Control](#).
3. In the Control Tree of the Layout Editor, select the node of the new button.
4. Click the **Properties** tab item to open the list of properties for the button.
5. Specify values for the following properties:
 - a. **CommandSourceID**: Set the property to the value specified in the `ID` property of the `PXDataSource` control.

- b. **CommandName**: In the property, specify the name of the method to be invoked when the user clicks the button on the form.



: The method must be defined in the graph that is specified in the `TypeName` property of the `PXDataSource` control.

- c. **Text**: Specify the text to be displayed on the button.

6. Click **Save to save your changes to the customization project.**

The following ASPX code creates a button that invokes the `MethodName` method referenced in the `PXDataSource` control with the `ID` property set to `ds`.

```
<px:PXButton ... CommandName="MethodName" CommandSourceID="ds" Text="ButtonText" />
```

The `MethodName` method must be referenced in a `PXDSCallbackCommand` element of the `PXDataSource` control as follows.

```
<px:PXDataSource ID="ds" ... TypeName="PX.Objects.<GraphName>" ...>
  <CallbackCommands>
    ...
    <px:PXDSCallbackCommand Name="MethodName" Visible="False" ... />
    ...
  </CallbackCommands>
</px:PXDataSource>
```

To add a `PXDSCallbackCommand` element for the button to the `PXDataSource` control, perform the following actions:

1. In the Control Tree of the Layout Editor, select the node of the `PXDataSource` control.
2. Click the arrow left of the node to expand it.
3. Click the **Add Controls** tab item.
4. From the **Other Controls** group, drag the **Button** item to the needed location in the Control Tree within the `PXDataSource` control, as shown in the following screenshot.

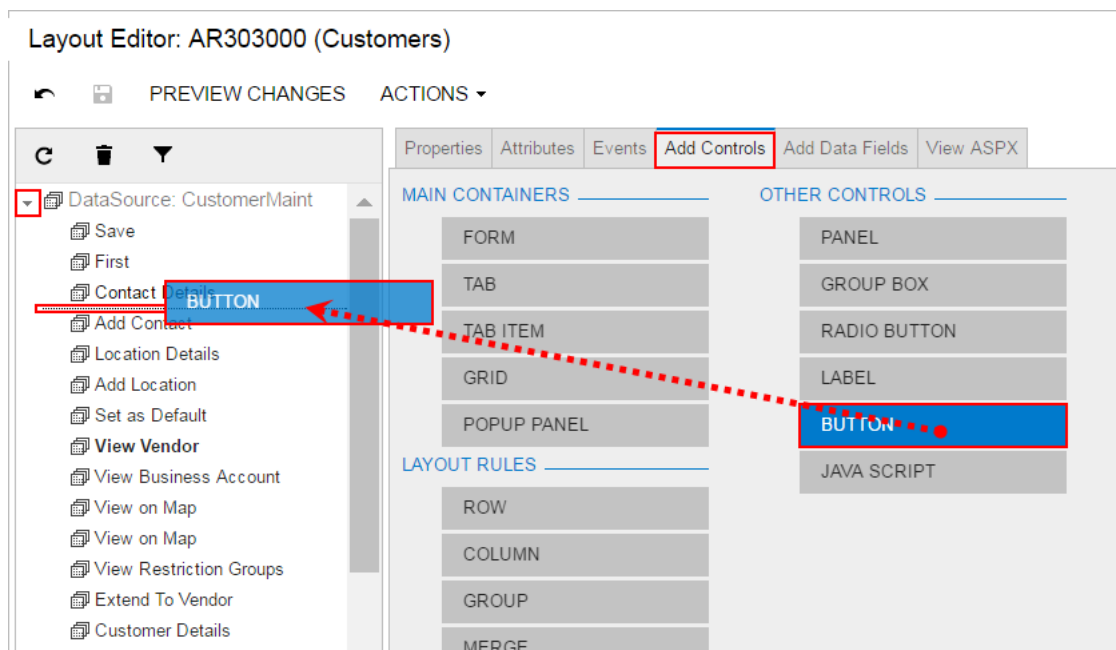


Figure: Adding a button command to the `PXDataSource` control

5. In the Control Tree of the Layout Editor, select the new `DSCallbackCommand` node.

6. Click the **Properties** tab item to open the list of properties for the node.
7. Specify values for the following properties of the callback command for the button (see the screenshot below):
 - a. **Name:** In the property, specify the name of the method to be invoked when the user clicks the button on the form.
 - b. **Visible:** If you do not want to add the button to the form toolbar, set the property to *False*. The default value is *True*.
 - c. **DependOnGrid:** If you need to make an action button unavailable on the toolbar when the grid is empty, set the property to the value that is specified in the `ID` property of the `PXGrid` element.
 - d. **PopupVisible:** If the form is opened as a pop-up window and you want to display the button on the toolbar of the pop-up window, set the property to *True*. The default value is *False*.
 - e. **CommitChanges:** If you need to update data in the cache before the application server executes the action, set the property to *True*. Otherwise, the action processes the data currently stored in the cache. The default value is *False*.

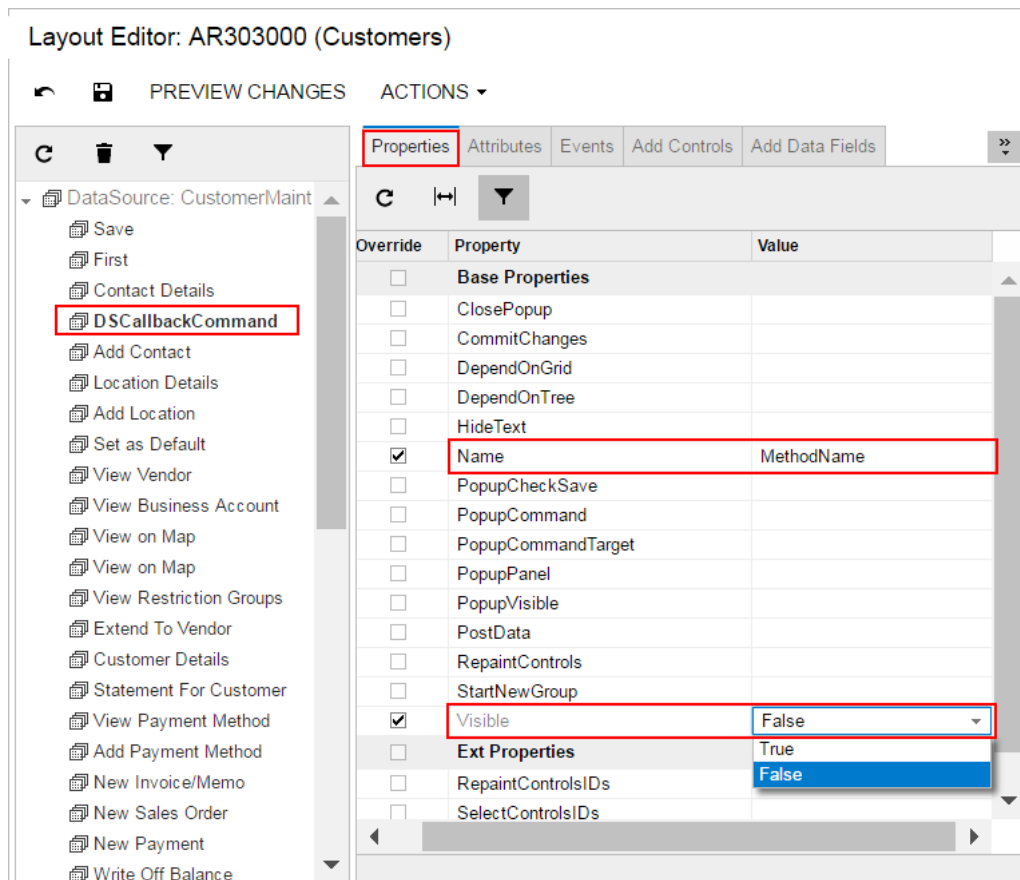


Figure: Setting properties for a callback command

8. Click **Save** to save your changes to the customization project.

Java Script (PXJavaScript)

The `PXJavaScript` element is designed to keep JavaScript code in an ASPX page.

A `PXJavaScript` control can be immediately added to the following types of containers:

- `PXFormView`

- RowTemplate
- PXTabItem
- PXSmartPanel
- PXPanel

To create a JavaScript control in a container, follow the instructions described in [To Add Another Supported Control](#).

To delete a JavaScript control from a container, follow the instructions described in [To Delete a Child UI Element](#).

For a detailed example of using a JavaScript control, see [How do I play a sound in response to a custom action or button in Acumatica?](#)

Toolbars, Action Buttons, and Menus

On an Acumatica ERP form, toolbars, action buttons, and menus are the types of UI elements that are created and processed by the graph that is specified in the `TypeName` property of the `PXDataSource` control and provides business logic for the form. Therefore, these UI elements cannot be created or customized by using the Layout Editor. See [Customizing Business Logic](#) for details.

The standard page toolbar buttons include data manipulation (**Insert**, **Delete**, **Save**, and **Cancel**), navigation (**Next**, **Previous**, **First**, and **Last**), and clipboard (**Copy** and **Paste**). Every toolbar button corresponds to an action declared in the graph. You can use explicit or implicit declaration of actions to add standard buttons to the toolbar. To add standard toolbar buttons to the form and add selector controls, in the graph class declaration, specify the main DAC as the second type parameter of `PXGraph`, as the following code shows.

```
public class GraphName : PXGraph<GraphName, MainDACName>
{
    public PXSelect<MainDACName> PrimaryDataViewName;
}
```

To add a custom action to a form toolbar, you use [Code Editor](#) to create the action declaration in the graph extension. See [To Add an Action](#) for details.

To create a menu on a form toolbar, in the graph or graph extension, you have to declare one action as a menu holder on the toolbar, and then declare all required actions and add these actions to the menu holder by using the `AddMenuAction` method of the `PXAction` class.

Other Control Types

The Acumatica Customization Platform does not support some control types, such as `PXSplitContainer` and `PXTreeView`, that are supported by the Acumatica Framework. However the Acumatica Customization Platform gives you the ability to add a control of an unsupported type to an Acumatica ERP form and save the change to a customization project.

To do this, perform the following actions:

1. Open the form in the [Layout Editor](#), as described in [To Start a Customization of a Form](#).
2. On the editor toolbar, click **Actions > Edit ASPX** to open the ASPX code of the form in the [ASPX Editor](#).
3. Modify the code in the editor.
4. Click **Generate Customization Script** to save to the customization project the changes to the *Page* item for the form.

After you have added a new control to the customization project, you can select the control node in the Control Tree of the Layout Editor and specify the control properties, as described in [To Set a Box Property](#).

Customizing 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 cache 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 implements methods that are executed as event handlers. Business logic can be divided into common logic relevant to different parts of the application and the logic specific to an application form. 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.

You use [Customization Tools](#) of the Acumatica Customization Platform to customize business logic for forms of Acumatica ERP.

The customization of existing business logic is based on the extension technology, which is described in greater detail in [DAC Extensions](#) and [Graph Extensions](#).

This part contains detailed instructions on how to customize a data access class, data field, graph, data view and an action of Acumatica ERP and how to include the changeset in a customization project.

In This Part

- [Data Access Class](#)
- [Data Field](#)
- [Graph](#)
- [Data View](#)
- [Action](#)

Data Access Class

A data access class (DAC) is used to represent a database table in the code of Acumatica ERP. If a DAC is bound to the database, it must have the same class name as the database table. A data access class is inherited from the `IBqlTable` class and contains data field declarations.

You can use [Customization Project Editor](#) to do the following:

- Create a new DAC
- Add a custom field to an existing DAC
- Customize the declaration of a data field of an existing DAC

For detailed information on customizing data access classes, see the following topics:

- [To Start the Customization of a Data Access Class](#)
- [To Add a Custom Data Field](#)
- [To Create a New DAC](#)
- [To Create a DAC Extension](#)

See [Data Field](#) for detailed instructions on the customization of a data field.

To Start the Customization of a Data Access Class



: Before you start a DAC customization, we recommend that you check the possibility to use the appropriate business entity attributes, which can be defined on the Attributes form (CS.20.50.00), instead of custom fields. See [Usage Entity Attributes Instead of Custom Fields](#) for details.

You might need to customize an existing data access class in the following cases:

- On a form, you want to create a custom data field to be used for a custom control.
- In Acumatica ERP, you want to change the business logic for a data field in the DAC.



Important: If multiple Acumatica ERP forms contain controls for the same field, changing an attribute of the field in the DAC modifies the behavior of these controls for the field on all forms.

If you know the name of the data access class to be customized, you can first add a *DAC* item for the class on the Customized Data Classes page of the *Customization Project Editor* (see *To Add a DAC Item for an Existing Data Access Class to a Project* for details) and then click the item to open it in the *Data Class Editor*.

You usually start the customization of a data access class on a form that contains boxes used by users to work with data of an appropriate business entity, such as a sales order invoice.

To start the customization of a DAC from a form, perform the following actions:

1. Open the form in the browser.
2. On the form title bar, click **Customization > Inspect Element** to launch the *Element Inspector*.



: If you need to activate the *Element Inspector* for a pop-up panel, a dialog box, or another UI element that opens in modal mode and makes the *Customization Menu* unavailable for selection, you can press Control-Alt.

3. On the form, click the area with the boxes that are used for the data of the business entity, to open the *Element Properties* dialog box for the area.
4. In the **Data Class** box of the dialog box, view the name of the DAC for the element you clicked. (See the following screenshot.) If you are not sure that this DAC is the needed one, click **Cancel** and repeat Step 3.
5. Click **Actions > Customize Data Fields**, as the following screenshot shows.

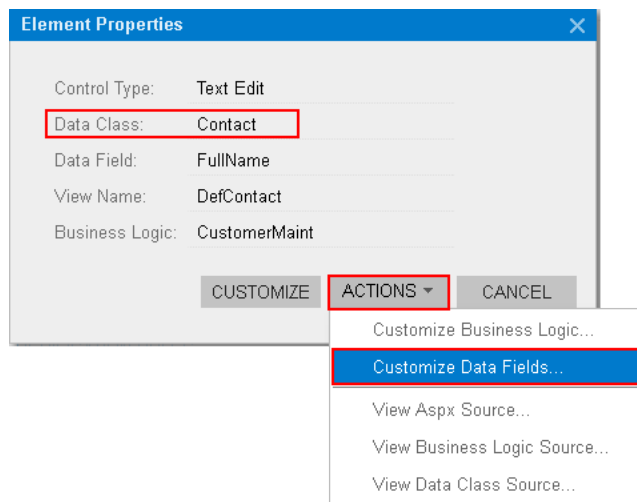


Figure: Using the Element Properties dialog box to start the customization of the class

6. If there is no currently selected customization project and the inspector opens the *Select Customization Project Dialog Box*, select an existing customization project or create a new one (see *To Create a New Project* and *To Select an Existing Project* for details).

If the customization project does not contain a *DAC* item for the data access class, the *Customization Project Editor* adds the item to the project to keep the changes in the database. The DAC is opened in the *Data Class Editor*, and you can start the customization of the class.

When you click **Save** on the editor toolbar, the editor updates the *DAC* item in the database.

Usage Entity Attributes Instead of Custom Fields

Under some circumstances, you will find it best to use entity attributes defined on the Attributes form (CS205000; **Configuration > Common Settings > Common Settings**), while under other circumstances, custom fields will help you better meet the requirements of your customization task.

You might use attributes under the following circumstances:

- You don't need a complex layout for attribute fields because they aren't used for manual input, so you can use the easiest way of creating additional fields.
- You'd prefer an approach that helps you to localize customizations that you make to the system to a limited number of spaces, which in turn helps you to avoid having scattered changes throughout the system.

The use of custom fields provides more functionality than attributes do. Using attributes as columns in a grid can present difficulties. For example, in a grid, you can display columns for either all attributes or none of them. Also, you cannot rename a column for an attribute in a grid.

You might use custom fields under the following circumstances:

- These fields are for manual input and you need a nice-looking layout for them.
- You also have some business logic of validation based on a custom field, so you might need the customization code for these fields.

To Add a Custom Data Field

You can add a custom data field to an existing data access class by using the [Layout Editor](#) or [Data Class Editor](#), as described in the following sections:

- [By Using the Data Class Editor](#)
- [By Using the Layout Editor](#)

Both of the editors opens the [New Field](#) dialog box, which you use to enter the parameters of the new field and to launch the New Field wizard, which includes an XML declaration of the new field in the *DAC* item for the modified data access class. If the *DAC* item for the modified data access class is absent in the customization project, the wizard creates the appropriate item. Also, the wizard creates a *Table* item with a description of the custom column added to the table for the custom field.

You can select the way a custom field will be stored in the database when you add the field to a data access class. You can add a custom field as one of the following:

- Without mapping to the database
- As a new column to the database table by altering the table schema
- As a name-value pair to a dedicated table without altering the table schema

The detailed information is described in the following sections:

- [To Add a Custom Field Without Mapping to the Database](#)
- [To Add a Custom Field as a New Column](#)
- [To Add a Custom Field as a Name-Value Pair](#)
- [To Delete a Custom Field](#)

To be able to create a control for the new field to be displayed on a form, you have to publish the project to make the system create the column in the database table and compile the customization code. After the publication, you can add the control for the new field to a form by using the Layout Editor. See [To Add a Box for a Data Field](#) or [To Add a Column for a Data Field](#) for details.

By Using the Data Class Editor

You generally use the [Data Class Editor](#) to add a custom field to a data access class (DAC) and store the changes in the customization project as a *DAC* item. To do this, perform the following actions:

1. Start customization of the data access class, as described in [To Start the Customization of a Data Access Class](#).
2. On the toolbar of the Data Class page, click **Add Field > Create New Field**.
3. In the **New Field** dialog box, which opens, specify the custom field parameters that are described in [Create New Field Dialog Box](#).
4. Click **OK** to save changes in the customization project.

By Using the Layout Editor

If you customize a form by using the [Layout Editor](#), you may need to create a custom data field to be used to create a new control on the form. By using the editor, you can create a new field in the main DAC of the data view that provides data for the object selected in the Control Tree. To do this, perform the following actions:

1. In the Control Tree of the Layout Editor, select the node of a container to which you intend to add a new control, or a subnode within the container node.
2. Click the **Add Data Fields** tab item.
3. On the table toolbar of the tab item, click **New Field**.
4. In the **New Field** dialog box, which opens, specify the custom field parameters, which are described in [Create New Field Dialog Box](#).
5. Click **OK** to save your changes to the customization project.

To Add a Custom Field Without Mapping to the Database

To add a custom field without mapping to the database, perform the following actions:

1. Open **Create New Field** dialog box for the data access class [By Using the Data Class Editor](#) or [By Using the Layout Editor](#).
2. In the dialog box, specify the custom field parameters and select *NonPersistedField* in **Storage Type**.
3. Click **OK** to add the field to the data access class.

The New Field wizard includes an XML declaration of the new field in the *DAC* item for the modified data access class. The new field declaration contains the data type attribute that specifies an unbound field, such as `[PXString]`.

To Add a Custom Field as a New Column

To add a custom field as a new column that will be appended to the original table of Acumatica ERP, perform the following actions:

1. Open the **Create New Field** dialog box for the data access class [by using the Data Class Editor](#) or [by using the Layout Editor](#).
2. In the dialog box, specify the custom field parameters and select *DBTableColumn* in the **Storage Type** box to make the system append the column to the original table in the database.
3. Click **OK** to add the field to the data access class.

The New Field wizard includes an XML declaration of the new field in the *DAC* item for the modified data access class. If the *DAC* item for the modified data access class is absent in the customization project, the wizard creates the appropriate item. Also, the wizard creates a *Table* item with a description of the custom column to be created in the database table for the custom field.

To Add a Custom Field as a Name-Value Pair

To add a custom field a name-value pairs stored in a dedicated table of Acumatica ERP, perform the following actions:

1. Open **Create New Field** dialog box for the data access class *By Using the Data Class Editor* or *By Using the Layout Editor*.
2. In the dialog box, specify the custom field parameters and select *NameValuePair* in **Storage Type**.
3. Click **OK** to add the field to the data access class.

The New Field wizard includes an XML declaration of the new field in the *DAC* item for the modified data access class. If the *DAC* item for the modified data access class is absent in the customization project, the wizard creates the appropriate item. The wizard doesn't generate any definitions to change the database schema.

To Delete a Custom Field

To delete a custom field or custom attributes on a field from the data access class, open the data access class in the Data Class Editor and delete the custom field from the field list. If you work in Microsoft Visual Studio, publish the customization project to update the file with the customized data access class in the file system. After the project has been published, the customization will be removed from the data access class.


To Create a New DAC

You can add a new data access class (DAC) to a customization project by generating the code from the definition of a database table.

To create a custom data access class for a custom database table and add the created item to a customization project, you have to generate the class template on the Code page of the Customization Project Editor.

To do this, perform the following actions:

1. Create the needed custom table in the database by using a database management tool.
2. Generate the DAC code for the customization project as follows:
 - a. Open the customization project in the Project Editor.
 - b. Click **Code** in the navigation pane to open the Code page.
 - c. Click **Add New Record (+)** on the page toolbar.
 - d. In the **Create Code File** dialog box, which opens, select *New DAC* in the **File Template** box, as the screenshot below shows.
 - e. In the **Class Name** box, specify the class name that corresponds to the name of the table created in the database.

 : If you have just created the table, restart Internet Information Services (IIS) or recycle the application pool to make sure Acumatica ERP is aware of the new table, because it caches the database schema once, when the domain starts.
 - f. Select the **Generate Members from Database** check box.
 - g. Click **OK**.

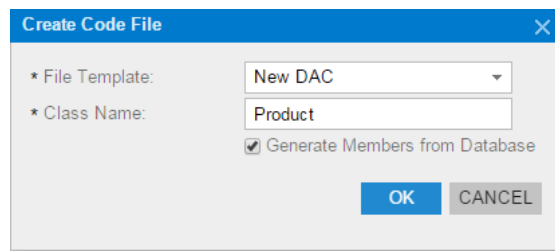


Figure: Adding a new data access class to the customization project

The platform does the following:

- Generates the data access class with members that correspond to the table columns. The class is added to the customization project namespace.
- Adds the class to the customization project as a *Code* item.
- Saves the customization project.
- Opens the created item in the [Code Editor](#).

You can use the Code Editor to modify the generated code. After you publish the customization project, you can work with the custom data access class in MS Visual Studio.

3. In the Code Editor, define the key fields in the DAC. To include a data field in the key, in the type attribute of the field, you have to add the `IsKey` parameter, as the example below shows.

```
[PXDBString(15, IsKey=true)]
```

4. Add the table definition to the customization project by doing the following:
 - a. In the navigation pane of the Project Editor, select **DB Scripts**.
 - b. On the Database Scripts page, which opens, click **Add New Record (+)** on the toolbar.
 - c. In the **Edit SQL Script** dialog box, which opens, select the table name in the **DBObject Name** selector.
 - d. Select the **Import Table Schema from Database** check box, which appears in the dialog box once the platform has found the specified table in the database (see the screenshot below).
 - e. Click **OK**.

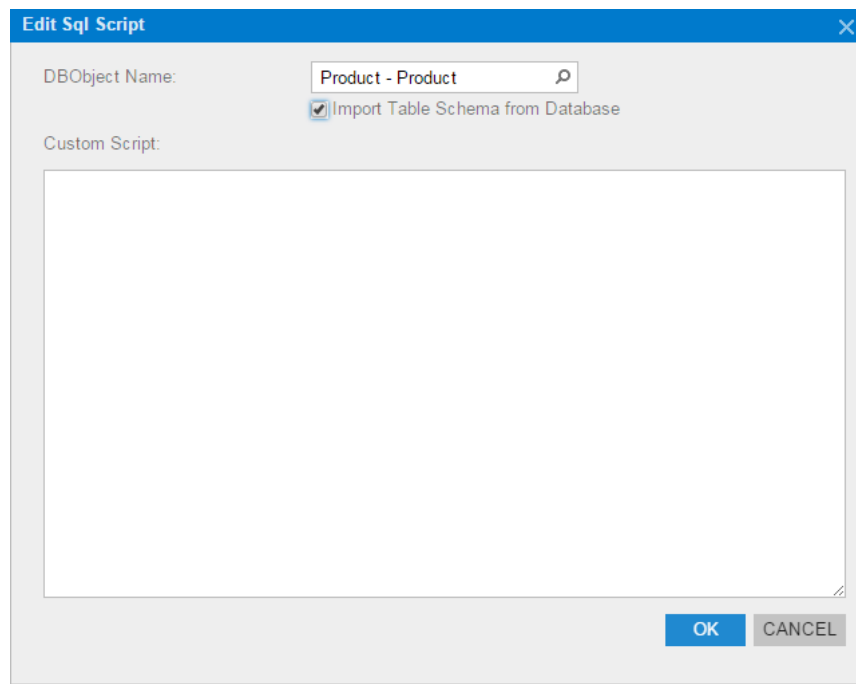


Figure: Adding an SQL script to the customization project

The platform does the following:

- Adds the XML definition of the table to the customization project as an *Sql* item
- Saves the customization project

Every time you publish the customization project, the system checks whether a table with this SQL definition exists in the database. If the table doesn't exist, the system creates the table. If the table exists, the system adjusts the table schema by using the definition, if there is any difference (no data is truncated).

To Create a DAC Extension

If you know the name of the data access class to be customized, you can create a *Code* item with the DAC extension template on the Code page of the Customization Project Editor by using the **Create Code File** dialog box.

To do this, perform the following actions:

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. Click **Add New Record (+)** on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *DAC Extension* in the **File Template** box, as the screenshot below shows.
5. In the **Base DAC** box, select the name of the data access class to be customized.
6. Click **OK**.

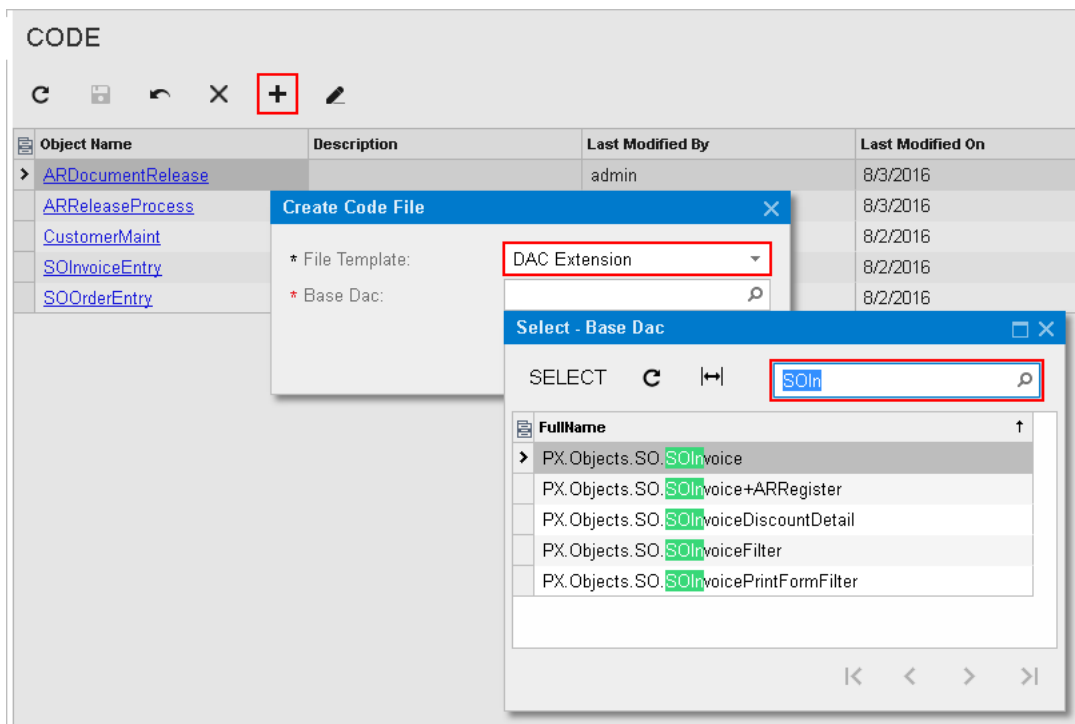


Figure: Adding a Code item with the DAC extension to the project

The platform creates the template of the class that is derived from the `PXCacheExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the *Code Editor*.

Data Field

In a data access class (DAC), a data field declaration consists of the following elements:

- A public abstract class that implements the `PX.Data.IBqlField` interface. This abstract class is used as a *type* to reference the field in the BQL statements specified in the data views and attributes added to data field declarations.
- Attributes that provide the common business logic for the field. On a field, you use attributes to define the data field specification, which includes multiple parameters, such as the data type, default value, and field caption in the UI. The type attribute, such as `PXDBString` and `PXDBDecimal`, is the only mandatory attribute of a DAC field.
- A public virtual property of a nullable data type that corresponds to the data type of this field. This property keeps the value of the data field. The attributes are added to the property and not to the abstract class of the data field declaration.

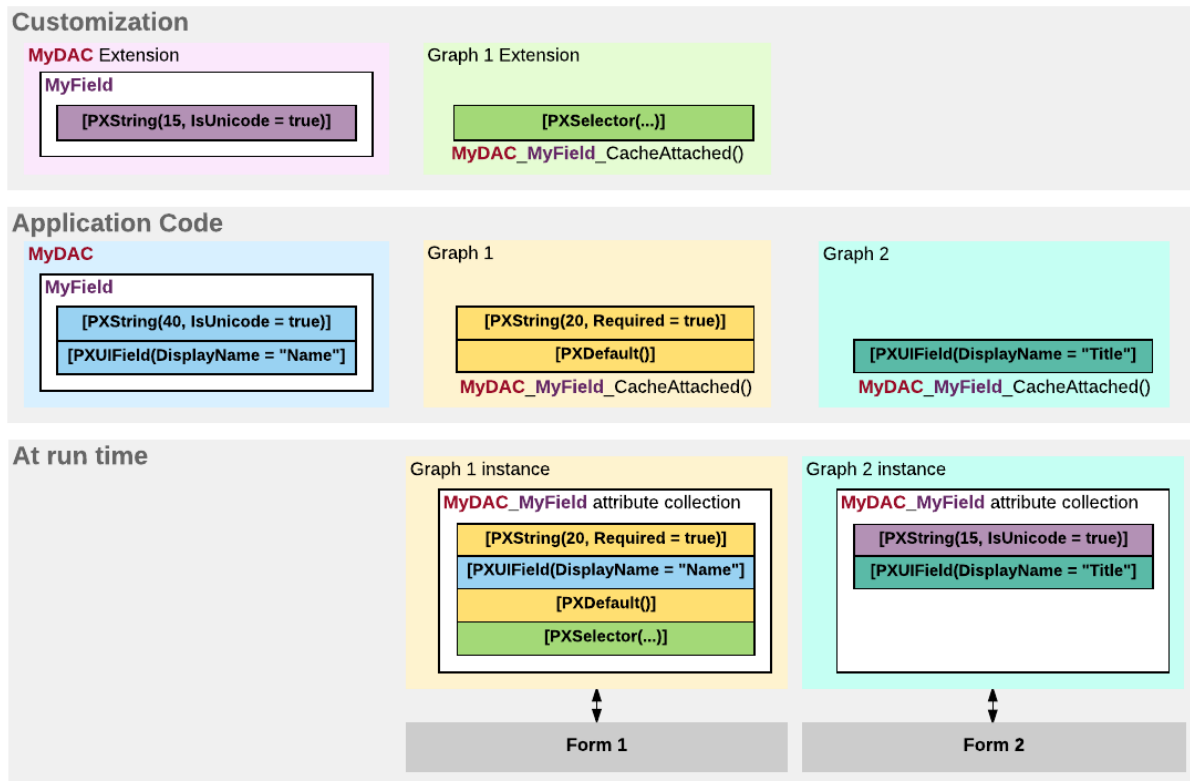
For a data field, you can customize only field attributes. You do this in one of the following ways:

- In the DAC (on the DAC level)
- In a graph (on the graph level)

On the DAC level, you can customize the attributes of a field by adding, deleting, replacing, or modifying an attribute in the DAC extension by using the *Data Class Editor*.

On the graph level, you can define a new set of attributes for a field and apply changes by means of the `DACName_FieldPropertyName_CacheAttached()` event handler in the graph extension.

The following diagram shows an example that demonstrates how the attributes of a field declared in a data access class can be changed on the DAC and graph levels for two forms.



: In the example, as recommended in [Customization of Field Attributes in DAC Extensions](#), the `CacheAttached()` event handler is always used to merge the custom attributes of the field with the original ones instead of replacing them on the graph level.

Figure: Example of the customization of attributes for a data field on the DAC and graph levels

On the diagram, you can see that the attribute collection for the same data field in the cache object can be different for graphs that provide business logic for different forms.

For detailed information on customizing a data field, see the following topics:

- [To Customize a Field on the DAC Level](#)
- [To Customize a Field on the Graph Level](#)
- [To Set a Default Value](#)
- [To Change the Label of a Field](#)
- [To Make a Field Mandatory](#)
- [To Customize the Table of a Selector Field](#)
- [To Add an Event Handler for a Field](#)
- [To Provide Multi-Language Support for a Field](#)


To Customize a Field on the DAC Level

If an attribute is added to a field declaration within a data access class (DAC), the common business logic of this attribute is applied to each data record of this type for each graph that uses this class. Therefore, if multiple Acumatica ERP forms contain controls for the same field, a change to an attribute of the field in the DAC modifies the behavior of these controls for the field on all forms.

You can use the [Data Class Editor](#) to customize the attributes of an existing data field in the DAC and to store the changes in the customization project as a *DAC* item. Because the goal of the customization of a field on the DAC level is to change the behavior or appearance of controls for the field on all forms, you start the customization of a data field from any form that contains a control for the field.

To customize a data field from a form that contains a control for the field, perform the following actions:

1. Open the form in the browser and make the control visible on the form.
2. On the form title bar, click **Customization > Inspect Element** to launch the *Element Inspector*.



: If you need to activate the *Element Inspector* for a pop-up panel, a dialog box, or another UI element that opens in modal mode and makes the *Customization Menu* unavailable for selection, you can press Control-Alt.
3. On the form, click the control to open the *Element Properties* dialog box for the control.
4. In the dialog box, click **Actions > Customize Data Fields**.
5. If there is no currently selected customization project and the inspector opens the *Select Customization Project Dialog Box*, select an existing customization project or create a new one. (See *To Select an Existing Project* and *To Create a New Project* for details.)

If the customization project does not contain a *DAC* item for the data access class, the *Customization Project Editor* adds the item to the project to keep the changes in the database. The *DAC* is opened in the *Data Class Editor*, and the needed field is added to list of customized fields of the *DAC*.
6. In the list of customized fields of the *DAC*, click the name of the field to be customized.

Now you can start the customization of the field on the *DAC* level.

The *Data Class* editor shows the custom and original attributes of the field in the work area, and you can use the following objects:

- The **Customize Attributes** text area: Use this area to define the custom attributes of the field.
- A drop-down list that is visible for only existing customized fields of the data access class: In this list, select one of the following ways of applying the custom attributes to the field:
 - *Keep Original*: The original attributes remain on the field until you select another option in this box.
 - *Replace Original*: The original attributes of the field are replaced with the custom attributes specified in the **Customize Attributes** text area.
 - *Append to Original*: The custom attributes are added to the end of the list of the original attributes of the data field. If you use this option, make sure you do not duplicate the attributes of the field.
- The **Selector Columns** toolbar action, which is available for only the fields that are selectors: Use this action to open the *Customize Selector Columns* dialog box, in which you can modify the columns in the selector table.
- The **Edit Attributes** toolbar action, which is unavailable for a custom field: Invoke this action to change the values of parameters for the existing original attributes of the selected field.

When you click **Save** on the editor toolbar, the editor updates the *DAC* item in the database.

To Start the Customization of a Field from the Layout Editor

If you customize a form by using the *Layout Editor*, you may need to change the attributes a data field to change the appearance or behavior of an appropriate control on the form. If the control node is selected in the Control Tree, you can open the *Data Class* editor for the field immediately from the Layout Editor. To do this, perform the following actions:

1. Click the **Attributes** tab item.
2. On the tab item, click **Customize Attributes**.

If the *DAC* item for the data access class that contains the field declaration is absent in the customization project, the Acumatica Customization Platform creates an appropriate item and saves it in the database. Then the platform opens the item in the *Data Class* editor, and you can customize the attributes of the field.

To Customize a Field on the Graph Level

If an attribute is added to a field declaration within a graph, the common business logic of this attribute is applied to each data record of this type on the graph level for only this graph. Therefore, if multiple Acumatica ERP forms contain controls for the same field, a change of an attribute of the field within the graph modifies the behavior of a single control for the field on only the form for which the graph provides business logic.

Because the goal of customizing a field on the graph level is to change the behavior or appearance of a specific control for the field on a specific form, you start the customization of a data field from the form.

To customize a field on the graph level, perform the following actions:

1. Open the form in the browser and make the control visible on the form.
2. On the form title bar, click **Customization > Inspect Element** to launch the *Element Inspector*.



: If you need to activate the *Element Inspector* for a pop-up panel, a dialog box, or another UI element that opens in modal mode and makes the *Customization Menu* unavailable for selection, you can press Control-Alt.

3. On the form, click the control to open the *Element Properties* dialog box for the control.
4. In the dialog box, click **Actions > Customize**.
5. If there is no currently selected customization project and the inspector opens the *Select Customization Project Dialog Box*, select an existing customization project or create a new one. (See *To Select an Existing Project* or *To Create a New Project* for details.)

If the customization project does not contain a changeset for the form, the *Customization Project Editor* adds to the project a *Page* item for the form, to keep the changeset to the ASPX code of the form in the database. The container of the control is opened in the *Layout Editor*, and the control is selected in the Control Tree of the editor.

6. Click the **Attributes** tab item, which is used to review and customize the attributes of the DAC field that is bound to the control currently selected in the Control Tree.
7. Click **Override On Screen Level** to create in the graph extension the code template, which includes the original attributes of the field and the `DACName_FieldPropertyName_CacheAttached()` event handler, which replaces the attributes within the graph.

If the customization project does not contain an extension for the graph, the *Customization Project Editor* adds to the project a *Code* item for the graph extension, to keep the code in the database. Then the Acumatica Customization Platform opens the *Code* item in the Code Editor, and you can start the customization of the field on the graph level.

In the graph extension, you can define the field attributes as you need to. However you can use one of the following approaches to customize the attributes of a field on the graph level:

- Define a new set of custom attributes to replace the original attributes of the field (as in the created template).
- Use the `PXCustomizeBaseAttribute`, `PXMergeAttributes`, `PXRemoveBaseAttribute`, and `PXCustomizeSelectorColumns` attributes to customize the original attributes of a field instead of replacing it with custom ones.

We recommend that you use the second approach, which is described in *Customization of Field Attributes in DAC Extensions*.

When you click **Save** on the Code editor toolbar, the editor updates the *Code* item in the database.



: You can develop the customization code in the Code Editor. However we recommend that you develop the code in Microsoft Visual Studio (as described in *Integrating the Project Editor with Microsoft Visual Studio*) and use the editor for either minor code corrections or the insertion of completed portions of code.

To Set a Default Value

When a user creates a new record of a business entity through a form of Acumatica ERP, you might need to set a default value for a data field. You use the `PXDefault` attribute for a data field to set a

constant as the default value, or you 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.



: If the default value is taken from a field that can be auto-generated by the database (such as an ID), you should use the `PXDBDefault` attribute instead of the `PXDefault` attribute.

You can set a default value for an original or custom text field on the DAC and graph levels. The following sections provide detailed information:

- [To Set a Default Value for an Original Field on the DAC Level](#)
- [To Set a Default Value for an Original Field on the Graph Level](#)
- [To Set a Default Value for a Custom Field on the DAC Level](#)

To Set a Default Value for an Original Field on the DAC Level

To set a default value for a field used on multiple forms, you should customize the original attributes of the field in the data access class extension. To do this, perform the following actions:

1. Open the field in the Data Class Editor, as described in [To Customize a Field on the DAC Level](#).
2. In the **Customize Attributes** box, select *Append to Original*.
3. In the edit area below the box, add the `PXDefault` attribute and define a default value for the attribute.
4. Click **Save** on the editor toolbar to save your changes to the customization project.

To Set a Default Value for an Original Field on the Graph Level

To set a default value for a field used on a single form, you should customize the original attributes of the field in the graph extension. To do this, perform the following actions:

1. Create the code template that includes the original attributes of the field and the `DACName_FieldPropertyName_CacheAttached()` event handler, which replaces the attributes within the graph, as described in [To Customize a Field on the Graph Level](#).
2. By using the Code editor, replace the original attributes in the template, as shown in the following code snippet.

```
[PXMergeAttributes(Method = MergeMethod.Append)]
[PXDefault(false, PersistingCheck = PXPersistingCheck.Nothing)]
protected void DACName_FieldPropertyName_CacheAttached(PXCache cache)
{
}
```

3. Click **Save** on the editor toolbar to save your changes to the customization project.

To Set a Default Value for a Custom Field on the DAC Level

If you need to set a default value for a custom data field on the DAC level, you add the `PXDefault` attribute and define a default value for the attribute in the edit area of the Data Class editor, as shown in the following screenshot.

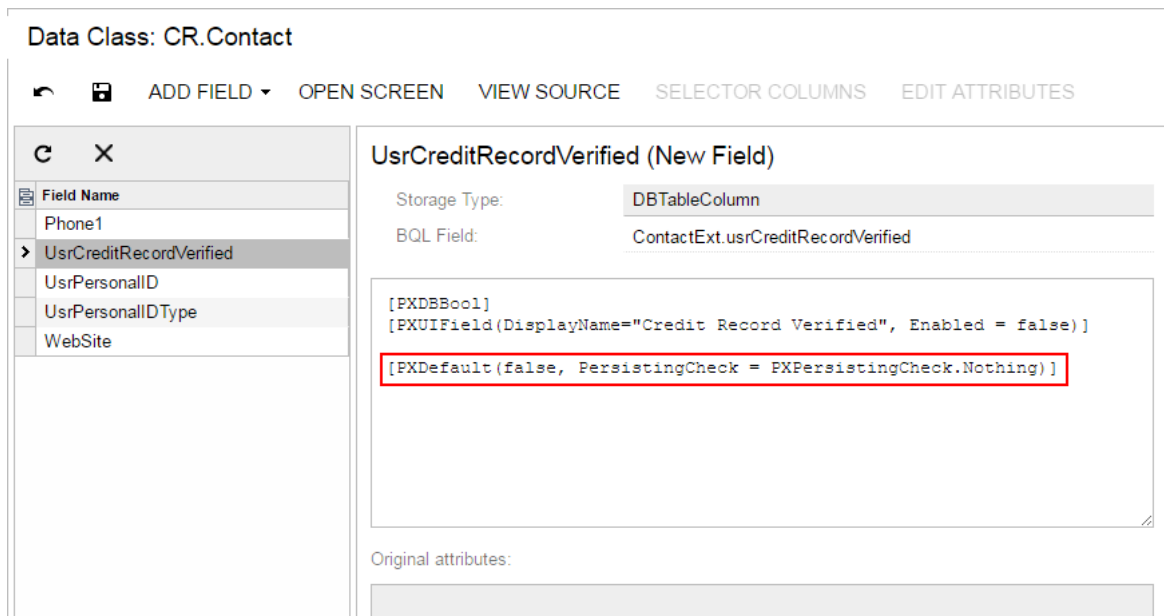


Figure: Setting a default value for a custom field

To Change the Label of a Field

You can change the label for an original data field on the DAC and graph levels. The following sections provide detailed information:

- [To Change the Label of a Field on the DAC Level](#)
- [To Change the Label of a Field on the Graph Level](#)

To Change the Label of a Field on the DAC Level

To change the label of a control for a field used on multiple forms, you should customize the `PXUIField` attribute of the field in the data access class extension. To do this, perform the following actions:

1. Open the field in the Data Class Editor, as described in [To Customize a Field on the DAC Level](#).
2. In the **Customize Attributes** box, select *Append to Original*.
3. In the editor, click **Edit Attributes** on the page toolbar.
4. In the **Attribute** list of the **Customize Attributes** dialog box, which opens, click the `PXUIField` attribute to select it.
5. In the parameter list, specify a new label for the `DisplayName` parameter, as the screenshot below shows.
6. Click **OK** to exit the dialog box.
7. Click **Save** on the editor toolbar to save your changes to the customization project.

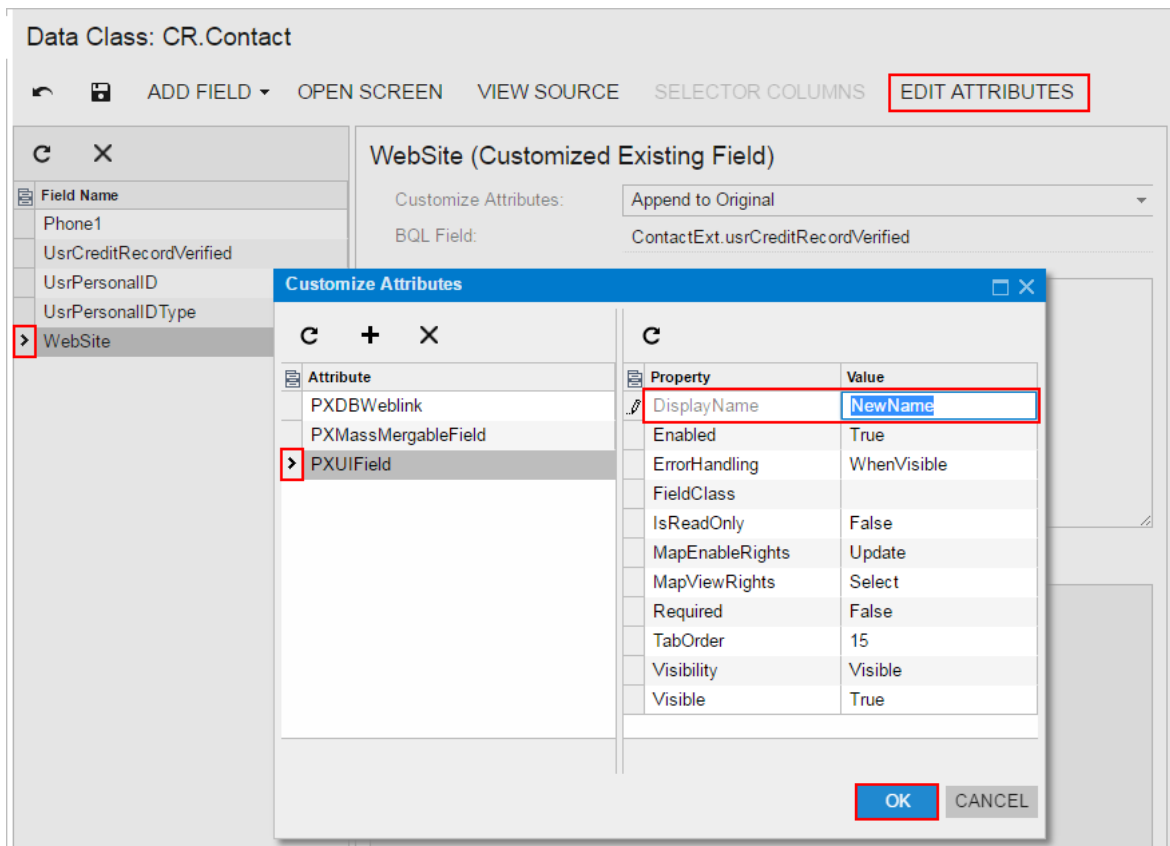


Figure: Changing the display label of a field

As a result, the editor adds the following attribute for the field to the DAC extension.

```
[PXCustomizeBaseAttribute(typeof(PXUIFieldAttribute), "DisplayName", "NewName")]
```

To Change the Label of a Field on the Graph Level

To change the label of a field for a single form, you should customize the `PXUIField` attribute of the field in the graph extension. To do this, perform the following actions:

1. Create the code template that includes the original attributes of the field and the `DACName_FieldPropertyName_CacheAttached()` event handler, which replaces the attributes within the graph, as described in [To Customize a Field on the Graph Level](#).
2. By using the Code editor, replace the original attributes in the template, as shown in the following code snippet.

```
[PXCustomizeBaseAttribute(typeof(PXUIFieldAttribute), "DisplayName",
"NewName")]
protected void DACName_FieldPropertyName_CacheAttached(PXCache cache)
{
}
```

3. Click **Save** on the editor toolbar to save your changes to the customization project.

To Make a Field Mandatory

If a field is mandatory for input, the user has to specify a value for the field before saving the record that contains the field. To make a field mandatory for input, you use the `PXDefault` attribute without parameters. A mandatory field is marked by an asterisk (*) on the form to cue the reader that a value

must be specified for it. To mark a control for a mandatory field with an asterisk, you set the `Required` parameter of the `PXUIField` attribute to `true`.

You can make an original or custom field mandatory and mark a control for the field with an asterisk on the DAC and graph levels. The following sections provide detailed information:

- [To Make an Original Field Mandatory on the DAC Level](#)
- [To Make a Custom Field Mandatory on the DAC Level](#)
- [To Make a Field Mandatory on the Graph Level](#)

To Make an Original Field Mandatory on the DAC Level

To make an original field mandatory and mark the control for the field with an asterisk on multiple forms with an asterisk, you should customize the original attributes of the field in the data access class extension. To do this, perform the following actions:

1. Open the field in the Data Class Editor, as described in [To Customize a Field on the DAC Level](#).
2. In the **Customize Attributes** box, select *Append to Original*.
3. In the edit area below the box, add the `PXDefault` attribute without parameters.
4. In the editor, click **Edit Attributes** on the page toolbar.
5. In the **Attribute** list of the **Customize Attributes** dialog box, which opens, click the `PXUIField` attribute to select it.
6. In the parameter list, set the `Required` parameter to `True`.
7. Click **OK** to exit the dialog box.

The edit area of the editor must contain the following code:

```
[PXDefault]
[PXCustomizeBaseAttribute(typeof(PXUIFieldAttribute), "Required", true)]
```

8. Click **Save** on the editor toolbar to save your changes to the customization project.

To Make a Custom Field Mandatory on the DAC Level

If you need to set a default value for a custom data field, you add the `PXDefault` attribute without parameters and set the `Required` parameter of the `PXUIField` attribute to `True`, as in the following code snippet.

```
[PXDefault]
[PXUIField(..., Required = true)]
```

To Make a Field Mandatory on the Graph Level

To make a field mandatory and mark the field with an asterisk for the field control used on a single form, you should customize the field attributes in the graph extension. To do this, perform the following actions:

1. Create the code template that includes the field attributes and the `DACName_FieldPropertyName_CacheAttached()` event handler, which replaces the attributes within the graph, as described in [To Customize a Field on the Graph Level](#).
2. By using the Code editor, replace the original attributes in the template, as shown in the following code snippet.

```
[PXDefault]
[PXCustomizeBaseAttribute(typeof(PXUIFieldAttribute), "Required", true)]
protected void DACName_FieldPropertyName_CacheAttached(PXCache cache)
```

```
{
}
```

3. Click **Save** on the editor toolbar to save your changes to the customization project.

To Customize the Table of a Selector Field

For a selector field, you use the *Data Class Editor* to add, delete, and sort the columns of the selector table and store the changes in the customization project as a *DAC* item.

You can make an original or custom field mandatory and mark a control for the field with an asterisk on the data access class (DAC) and graph levels. The following sections provide detailed information:

- [To Customize the Table of an Original Selector Field on the DAC Level](#)
- [To Customize the Table of an Original Selector Field on the Graph Level](#)

To Customize the Table of an Original Selector Field on the DAC Level

To customize the table of an original selector field for all controls for the field, you should customize the `PXSelector` attribute of the field in the data access class extension. To do this, perform the following actions:

1. Open the field in the Data Class Editor, as described in [To Customize a Field on the DAC Level](#).
2. In the editor, click **Selector Columns** on the page toolbar.
3. In the **Customize Selector Columns** dialog box, which opens, make the required changes.

You can use this dialog box to add new columns to the selector table and to reorder columns in the table. (See [Customize Selector Columns Dialog Box](#) for details.)

4. Click **OK** to add the `PXCustomizeSelectorColumns` attribute for the field in the edit area of the Data Class editor based on your changes, as shown in the following screenshot.

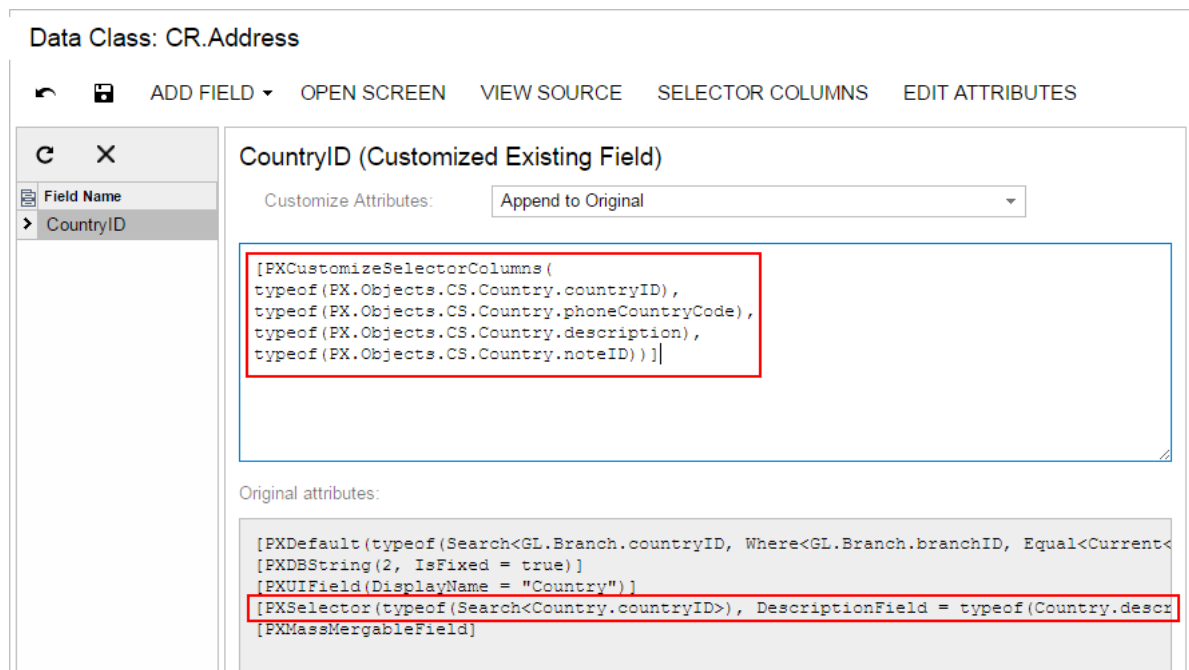


Figure: Viewing the `PXCustomizeSelectorColumns` attribute in the Data Class editor

5. Click **Save** on the editor toolbar to save your changes to the customization project.

To Customize the Table of an Original Selector Field on the Graph Level

To customize the selector table for the selector control on a single form, you should customize the `PXSelector` attribute of the appropriate field in the graph extension. To do this, perform the following actions:

1. Create the code template that includes the field attributes and the `DACName_FieldPropertyName_CacheAttached()` event handler that replaces the attributes within the graph, as described in [To Customize a Field on the Graph Level](#).
2. By using the Code editor, replace the original attributes in the template, as shown in the following code snippet.

```
[PXCustomizeSelectorColumns(<NEW CONTENT OF THE PXSELECTOR ATTRIBUTE>)]
protected void DACName_FieldPropertyName_CacheAttached(PXCache cache)
{
}
```

3. Click **Save** on the editor toolbar to save your changes to the customization project.

To Add an Event Handler for a Field

You can add an event handler for an original or custom field on the graph and DAC levels. The following sections provide detailed information:

- [To Add an Event Handler for a Field on the Graph Level](#)
- [To Add an Event Handler for a Field on the DAC Level](#)

To Add an Event Handler for a Field on the Graph Level

To add an event handler for a field control used on a single form, perform the following actions:

1. Open the form in the browser and make the control visible on the form.
2. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).

 : If you need to activate the [Element Inspector](#) for a pop-up panel, a dialog box, or another UI element that opens in modal mode and makes the [Customization Menu](#) unavailable for selection, you can press Control-Alt.
3. On the form, click the control for which you are adding an event handler to open the [Element Properties](#) dialog box for the control.
4. In the dialog box, click **Actions > Customize**.
5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one. (See [To Select an Existing Project](#) or [To Create a New Project](#) for details.)

If the customization project does not contain a changeset for the form, the [Customization Project Editor](#) adds a *Page* item for the form to the project to keep the changeset to ASPX code of the form in the database. The container of the control is opened in the [Layout Editor](#), and the control is selected in the Control Tree of the editor.

6. If you need to provide additional business logic to process a control value immediately after the value is changed, follow the instructions described in [To Add an Event Handler](#). If you need to override the existing business logic for the field, follow the instructions described in [To Override an Event Handler](#).

To Add an Event Handler for a Field on the DAC Level

Typically, you develop an event handler to provide the business logic for a field on the graph level. However if you need to add the same business logic for a field used on multiple forms, you add the event handler on the DAC level as follow:

1. Create a custom attribute, as described in Lesson 8 of the [T200 Acumatica Framework Fundamentals](#) course.
2. Add the needed event handler to the custom attribute.
3. Add the custom attribute to the field, as described in [To Customize a Field on the DAC Level](#).

To Provide Multi-Language Support for a Field

Acumatica ERP 5.3 supports locale-specific settings and the translation of the strings used on the application interface. Starting with Version 6, Acumatica ERP also provides the functionality to translate user input to multiple languages and store translations in the database. (See [Locales and Languages](#) for details.)

For example, if there are multiple active system locales in an instance of Acumatica ERP, and a text field on a form is declared as a multi-language field, the field control displays the value for the current locale. If the database does not contain a value for the current locale, the control displays the value for the default system locale.

To declare a text field as a multi-language one, in the data access class, you should use the `PXDBLocalizableString` attribute instead of `PXDBString` or `PXDBText`. The `PXDBText` attribute has to be replaced with the `PXDBLocalizableString` one without any length specified.



:

- Please do not confuse the `PXDBLocalizableString` attribute with `PXDBLocalizedString`. The `PXDBLocalizedString` attribute is deprecated and should not be used in the customization code.
- By default, starting with Acumatica ERP 6, the system supports multilingual user input for some boxes listed in [Boxes that Have Multi-Language Support](#). If the `PXDBString` attribute for a box from the list was customized in the earlier versions, after you upgrade to Acumatica ERP 6, the box will not have multi-language support. To resolve the issue, we recommend that you replace the `PXDBString` attribute with `PXDBLocalizableString` in the customization code.

You can provide multi-language support for an original or custom text field. The following sections provide detailed information:

- [To Provide Multi-Language Support for an Original Field on the DAC Level](#)
- [To Provide Multi-Language Support for an Original Field on the Graph Level](#)
- [To Provide Multi-Language Support for a Custom Field](#)

To Provide Multi-Language Support for an Original Field on the DAC Level

To provide multi-language support for a field used on multiple forms, you should customize the original attributes of the field in the data access class extension. To do this, perform the following actions:

1. Open the field in the Data Class Editor, as described in [To Customize a Field on the DAC Level](#).
2. In the editor, click **Edit Attributes** on the page toolbar.
3. In the **Customize Attributes** dialog box, which opens, select the `PXDBString` (or `PXDBText`) attribute, and click **Delete** (X) on the toolbar.
4. Click **OK** to save your changes and close the dialog box.
5. In the **Customize Attributes** box of the editor, select *Append to Original*, as shown in the screenshot below.
6. In the work area below the box, add the `PXDBLocalizableString` attribute, which has the same parameters as the deleted `PXDBString` (or `PXDBText`) attribute.

7. Click **Save** on the editor toolbar to save your changes to the customization project.

For example, after you perform these actions for the **Description** field on the Sales Orders form (SO301000; **Distribution > Sales Orders > Work Area > Enter**), the **Customize Attributes** area of the Data Class editor contains the following code (as shown in the screenshot below).

```
[PXDBLocalizableString(255, IsUnicode = true)]
[PXRemoveBaseAttribute(typeof(PXDBStringAttribute))]
```

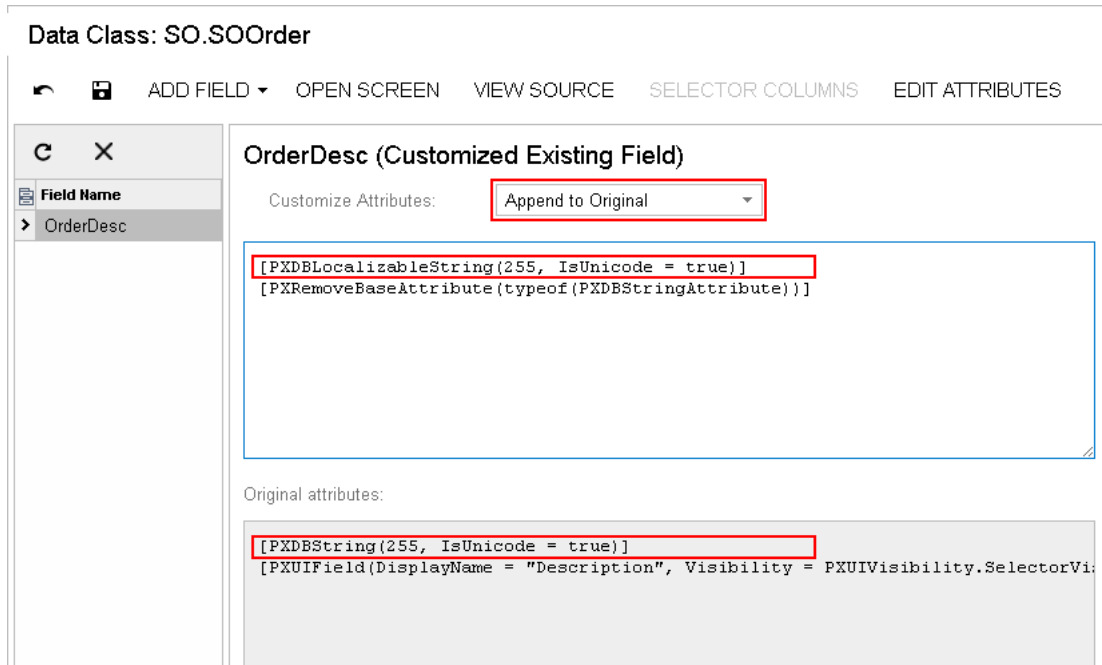


Figure: Providing multi-language support for the Description field of the Sales Orders form

To Provide Multi-Language Support for an Original Field on the Graph Level

To provide multi-language support for a field used on a single form, you should customize the original attributes of the field in the graph extension. To do this, perform the following actions:

1. Create the code template, which includes the original attributes of the field and the `DACName_FieldPropertyName_CacheAttached()` event handler, which replaces the attributes within the graph, as described in [To Customize a Field on the Graph Level](#).
2. By using the Code editor, replace the original attributes in the template, as shown in the following code snippet.

```
[PXMergeAttributes(Method = MergeMethod.Merge)]
[PXDBLocalizableString(255, IsUnicode = true)]
protected void DACName_FieldPropertyName_CacheAttached(PXCache cache)
{
}
```

3. Click **Save** on the editor toolbar to save the changes to the customization project.

To Provide Multi-Language Support for a Custom Field

If you need to create a custom text box with multi-language support, first you create a custom field with the `PXDBString` attribute. Then in the DAC, you replace this attribute name with `PXDBLocalizableString`.

Graph

A business logic controller (BLC, also referred as *graph*) is used to provide the business logic for an Acumatica ERP form. A graph instance is created when a data request occurs and discarded after the request is processed.

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 a rich syntax for building the 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.

Each graph contains the `Caches` collection of cache objects and the `Views` collection of data view objects. The framework handles these objects automatically; you don't have to initialize and control them. A `PXView` data view object contains two main parts:

- The BQL command, which is defined by the type of the data view
- The optional delegate, which constructs the data set that is returned instead of the BQL command's execution result

`PXView` objects, like graphs, are initialized and destructed on each round trip.



: The order in which data views are defined in a graph is important, because it defines the order of saving data to the database. (This order, however doesn't define the order in which data views are executed.) The data view that you specify in the `PrimaryView` property should always be defined first in the graph.

If a BQL statement of a data view refers to multiple data access classes (DACs), the data view creates the `PXCache` object for each DAC to keep appropriate data records, as shown in the following diagram.

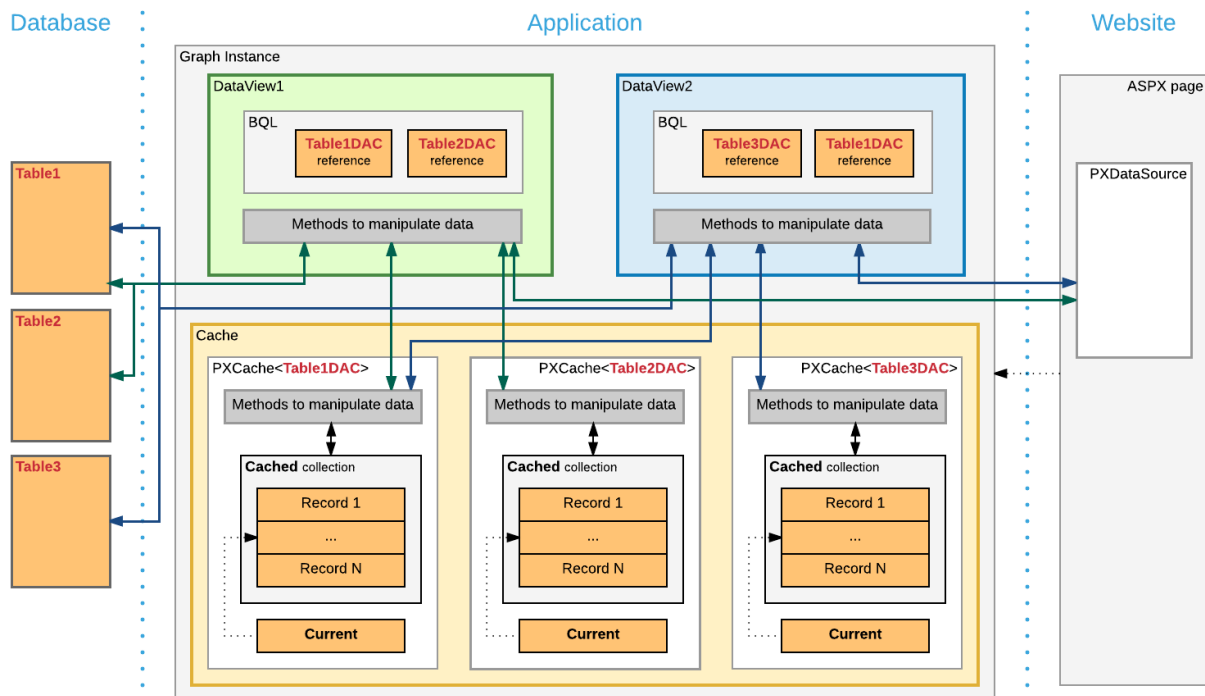


Figure: Example of two data views working with the records of three DACs

If multiple data views contain a reference to the same DAC, for this DAC, the graph cache contains a single `PXCache` object that is used by the data views.

On a webpage, you bind each container to a data view that provides data for the container. To bind a container and a data view, you specify the data view name in the `DataMember` property of the container

in the ASPX code. When a webpage requests data, the system invokes the `ExecuteSelect()` method of the graph with the data view name passed as an argument to execute every data view bound to the containers of the webpage. Note that data views that aren't bound to a container are not executed by a request from the UI.

When a data record is modified on the page, the framework invokes the `ExecuteInsert()`, `ExecuteUpdate()`, or `ExecuteDelete()` method of the graph, passing the name of the data view as an argument. The graph gets the data view by its name and invokes the corresponding method of the data view.



: You shouldn't use the `ExecuteSelect()`, `ExecuteInsert()`, `ExecuteUpdate()`, and `ExecuteDelete()` methods for purposes other than debugging.

You use the [Customization Tools](#) of the Acumatica Customization Platform to create a custom graph and to create new members and override existing ones in an existing graph.

For detailed information on creating a custom graph and customizing an existing graph, see the following topics:

- [To Start the Customization of a Graph](#)
- [To Create a Custom Graph](#)
- [To Add a New Member](#)
- [To Add an Action](#)
- [To Add an Event Handler](#)
- [To Override an Event Handler](#)
- [To Override a Virtual Method](#)

To Start the Customization of a Graph

You can create the class extension for an existing business logic controller (BLC) and add the `Code` item with the created code to a customization project in several ways, as described in the following sections:

- [To Add a Code Item by Using the Element Inspector](#)
- [To Add a Code Item by Using the Layout Editor](#)
- [To Add a Code Item on the Code page](#)

If you need to extend the code of a BLC that has no webpage associated (such as `ARReleaseProcess`), follow the instructions described in [To Add a Code Item on the Code page](#).

As soon as you add the `Code` item for customization of the business logic to the project, the system generates an extension class for it and opens the code in the [Code Editor](#). You can work with the extension classes in the Code Editor. After you publish the customization project, you can develop the code in MS Visual Studio.

To Add a Code Item by Using the Element Inspector

Typically, you want to modify the business logic that is executed for a certain form of Acumatica ERP.

To add a `Code` item for customization of the business logic for an existing form to a customization project by using the [Element Inspector](#), perform the following actions:

1. Open the form in the browser.
2. On the form title bar, click **Customization > Inspect Element** to launch the Element Inspector.
3. On the form, select any UI element to open the [Element Properties Dialog Box](#) for the element.

The **Business Logic** box of the dialog box displays the name of the business logic controller that provides business logic for the form, as shown in the screenshot below.

4. In the dialog box, click **Actions > Customize Business Logic**.

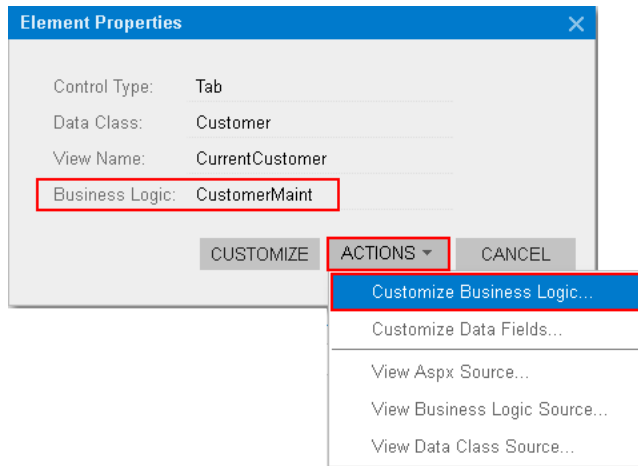


Figure: Using the Element Properties dialog box to customize the business logic for the form

5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or to create a new one.

The platform creates the template of the class that is derived from the `PXGraphExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the [Code Editor](#), as shown in the following screenshot.

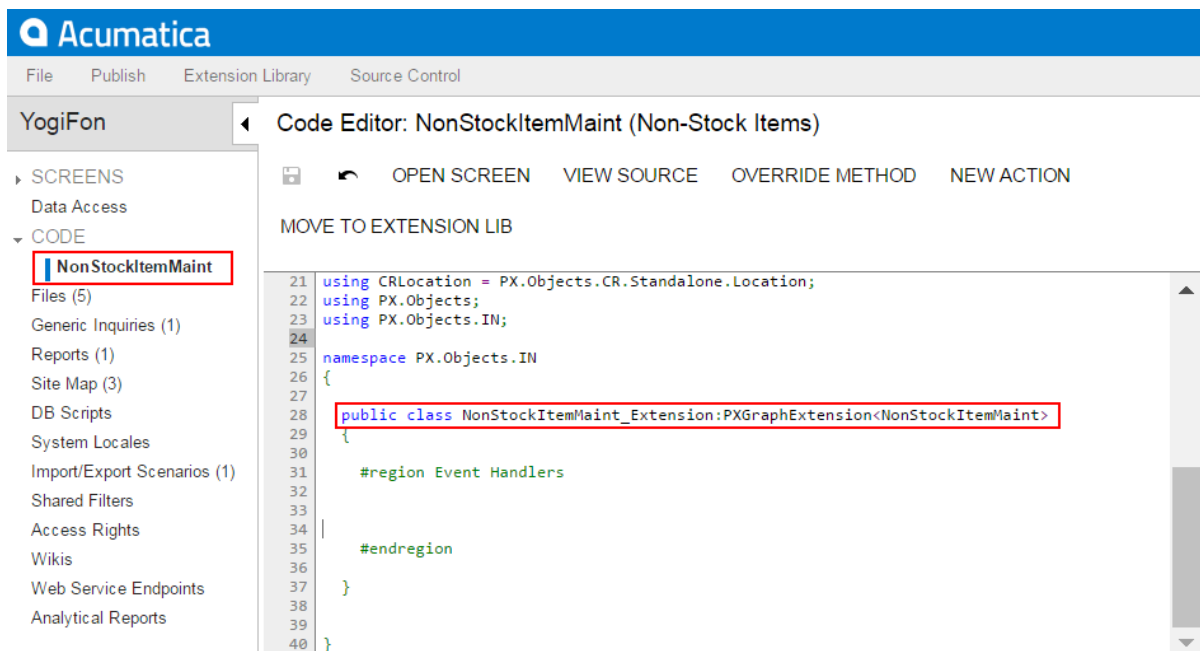


Figure: Viewing the created code template in the Code Editor

To Add a *Code* Item by Using the Layout Editor

Often, you start a customization of an Acumatica ERP form in the [Layout Editor](#) and you later want to modify the business logic for this form. To customize the business logic of the form, you can add a *Code* item to a customization project from the Layout Editor.

To do this, perform the following action:

1. On the toolbar of the Layout Editor, click **Actions > Customize Business Logic**, as the following screenshot shows.

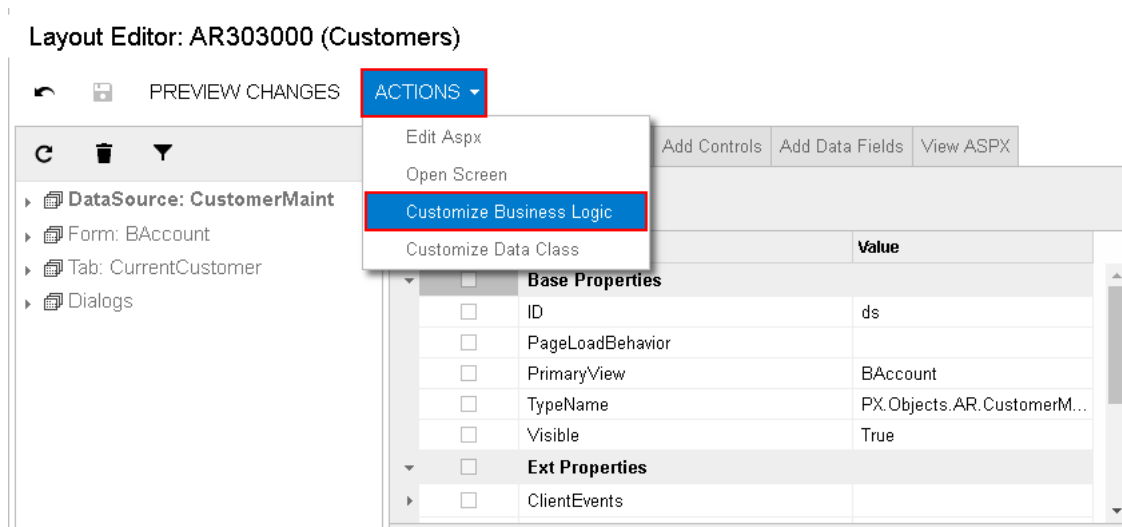


Figure: Starting the customization of the business logic from the Layout Editor

The platform creates the template of the class that is derived from the `PXGraphExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the [Code Editor](#).

To Add a *Code* Item on the Code page

If you know the name of the business logic controller to be customized, you can create a *Code* item with the graph extension template on the Code page of the Customization Project Editor by using the **Create Code File** dialog box.

To do this, perform the following actions:

1. Open the customization project in the editor. (See [To Open a Project](#) for details.)
2. Click **Code** in the navigation pane to open the Code page.
3. Click **Add New Record (+)** on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *Graph Extension* in the **File Template** box, as the screenshot below shows.
5. In the **Base Graph** box, select the class name of the business logic controller to be customized.
6. Click **OK**.

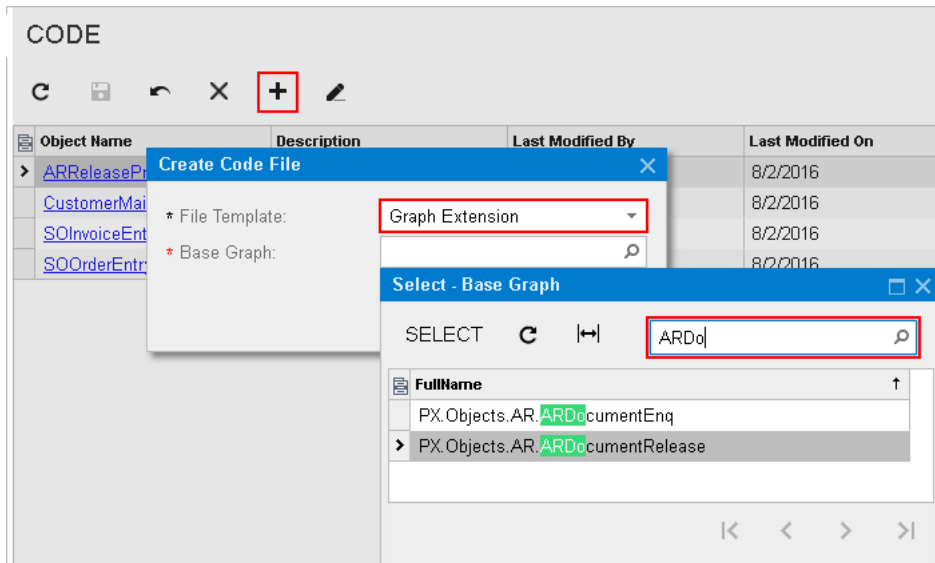


Figure: Adding a Code item with the graph extension to the project

The platform creates the template of the class that is derived from the `PXGraphExtension<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the *Code Editor*.

To Create a Custom Graph

You can add a custom business logic controller to a customization project on the Code page of the Customization Project Editor.

To do this, perform the following actions:

1. Open the customization project in the editor. (See *To Open a Project* for details.)
2. Select **Code** in the navigation pane to open the Code page.
3. Click **Add New Record** (+) on the page toolbar.
4. In the **Create Code File** dialog box, which opens, select *New Graph* in the **File Template** box, as the screenshot below shows.
5. In the **Class Name** box, specify the class name of the business logic controller to be created.
6. Click **OK**.

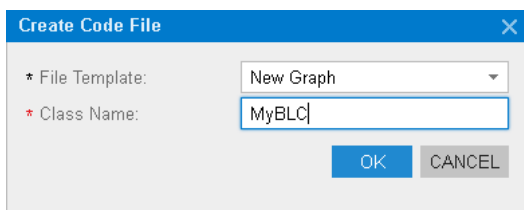


Figure: Adding a Code item for a custom graph to the project

The platform creates the code template of the class derived from the `PXGraph<>` class, saves the code as a *Code* item of the project in the database, and opens the item in the *Code Editor*.


To Add a New Member

You can add a new member (such as a variable, data view, or method) to a custom graph or to an extension for a graph in Acumatica ERP.

Initially, the code of a custom graph or an extension for an existing graph is saved in the appropriate *Code* item of the customization project. If the project has not been published, you can use only the [Code Editor](#) to add a new member to the graph code.

After the customization project has been published, the code is also saved in the corresponding C# file in the `App_RuntimeCode` folder of the website. (See [Changes in the Application Code \(C#\)](#) for details.) You can open the file in Microsoft Visual Studio and use Visual Studio to add a new member to the graph code. To do this, perform the following actions:

1. In Visual Studio, click **File > Open > Web Site**, and select your Acumatica ERP development environment.
2. Open the `App_RuntimeCode\<CodeItemName>.cs` file.
3. In the code, add the needed class member.
4. Save the file.
5. In the Customization Project Editor, select the **Files** node in the navigation pane.
6. On the Custom Files page, which opens, click **Detect Modified Files** on the page toolbar.



: The customization platform compares the content of each file added to the site with the content of the associated item of the customization project in the database. If a difference is detected, the platform opens the **Modified Files Detected** dialog box so you can resolve the detected conflicts. You can either update the customization project in the database or discard the file changes in the file system to resolve the conflicts.
7. In the **Modified Files Detected** dialog box, which opens, ensure that the modified code file is selected.
8. Click **Update Customization Project** on the dialog box toolbar to update the *Code* item that is saved in the database.

If the graph code is included in an extension library whose binary file is added to the customization project as a *File* item, you can use Visual Studio to add a new member to the graph code in the library solution. (See [Extension Library](#) for details.) If you make changes to the extension library file in the file system, you have to update the appropriate *File* item in the customization project. See [To Update a File Item in a Project](#) for details.

To Add an Action

To add a new action to the toolbar of an original form of Acumatica ERP, you have to add to the appropriate graph extension the action declaration, toolbar button declaration, and action delegate method. (See [To Start the Customization of a Graph](#) for details.)

To add a new action to the toolbar of a custom form, you have to add the action code to the appropriate custom graph. (See [To Create a Custom Form Template](#) and [To Create a Custom Graph](#) for details.)

You can use the [Code Editor](#) to add an action template to the graph extension or custom graph that is saved in a customization project as a *Code* item and currently opened in the editor. To do this, perform the following actions:

1. Click the **New Action** button of the Code Editor to open the **Create Action** dialog box.
2. In the dialog box, specify the name of the action delegate method in the **Action Name** box and the name of the action button in the **Display Name** box, as shown in the following screenshot.

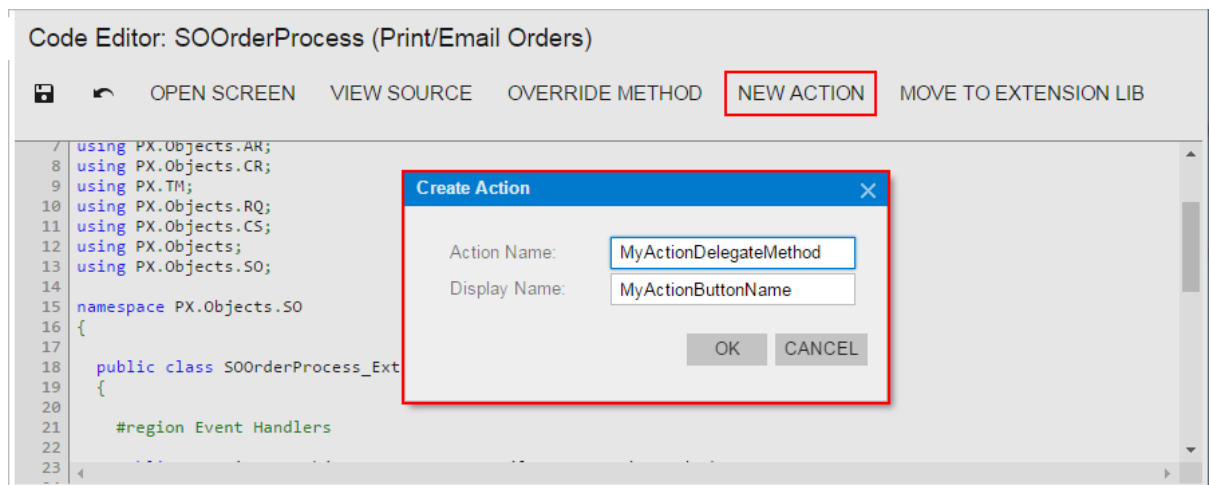


Figure: Opening the New Action dialog box

3. Click **OK** to create the action template and add it to the item.

The system adds to the code a template of the action declaration that includes the following class members:

- The declaration of the action delegate method
- The declaration of the button attributes to add the button to the form toolbar with the specified name
- The template of the action delegate method

The following example shows the template code for an action.

```

public PXAction<DACName> myActionDelegateMethod;

[PXButton(CommitChanges = true)]
[PXUIField(DisplayName = "MyActionButtonName")]
protected void MyActionDelegateMethod()
{
    // the body of the action delegate method
}

```

If the code of a graph extension or custom graph is moved to an extension library whose binary file is added to the customization project as a *File* item, you should develop the action code from scratch by using Visual Studio. If you make changes to the extension library file in the file system, you have to update the appropriate *File* item in the customization project. See [To Update a File Item in a Project](#) for details.

To Add an Event Handler

If you customize a form, you may need to provide additional business logic for processing a control value immediately after the value is changed.

To implement business logic for data changed by the user on an original form, you should add event handlers for data fields (or data records) to the extension of the graph that provides business logic for the form.

To process data modification on a custom form, you should add event handlers for data fields (or data records) to the appropriate custom graph.

You can use the [Layout Editor](#) to add to the appropriate customization code a workable template of an event handler for a data field (or data record) that can be modified on a form. To do this, perform the following actions:

1. Open the form in the browser and make the control for the field visible on the form.

2. On the form title bar, click **Customization > Inspect Element** to launch the *Element Inspector*.



: If you need to activate the *Element Inspector* for a pop-up panel, dialog box, or another UI element that opens in modal mode and makes the *Customization Menu* unavailable for selection, you can press Control-Alt.

3. On the form, click the control to open the *Element Properties* dialog box for the control.
4. In the dialog box, click **Actions > Customize**.
5. If there is no currently selected customization project and the inspector opens the *Select Customization Project Dialog Box*, select an existing customization project or create a new one. (See *To Create a New Project* and *To Select an Existing Project* for details.)
If the customization project does not contain a changeset for the form, the *Customization Project Editor* adds to the project a *Page* item for the form, to keep the changeset to the ASPX code of the form in the database. The container of the control is opened in the *Layout Editor*, and the control is selected in the Control Tree of the editor.
6. Select the node of the control in the Control Tree.
7. Click the **Properties** tab item to open the list of properties for the control.
8. In the list, set the `CommitChange` property to *True*. (See *Using the CommitChanges Property* for detailed information about the property.)
9. Click **Save** to save your changes in the customization project.
10. Click the **Events** tab item to open the list of available event types for the control.
11. In the list, click the type of the event that you want to process for the field to highlight the type, as shown in the screenshot below.
12. On the list toolbar, click **Add Handler > Keep Base Method**.

Layout Editor: AR303000 (Customers)

PREVIEW CHANGES ACTIONS ▾

Properties Attributes **Events** Add Controls Add Data Fields

Data Class: PX.Objects.CR.Contact
Field Name: UsrPersonalID
Business Logic: PX.Objects.AR.CustomerMaint::DefContact

ADD HANDLER ▾ VIEW SOURCE

Event	In Source	Customized
Com		<input type="checkbox"/>
Row		<input type="checkbox"/>
RowSelected	<input type="checkbox"/>	<input type="checkbox"/>
FieldSelecting	<input type="checkbox"/>	<input type="checkbox"/>
RowInserting	<input type="checkbox"/>	<input type="checkbox"/>
RowInserted	<input checked="" type="checkbox"/>	<input type="checkbox"/>
RowUpdating	<input type="checkbox"/>	<input type="checkbox"/>
RowUpdated	<input type="checkbox"/>	<input type="checkbox"/>
RowDeleting	<input type="checkbox"/>	<input type="checkbox"/>
RowDeleted	<input type="checkbox"/>	<input type="checkbox"/>
FieldDefaulting	<input type="checkbox"/>	<input type="checkbox"/>
FieldUpdating	<input type="checkbox"/>	<input type="checkbox"/>
FieldVerifying	<input type="checkbox"/>	<input type="checkbox"/>
ExceptionHandling	<input type="checkbox"/>	<input type="checkbox"/>

Figure: Adding an event handler for a data field

If there is no extension for the graph that provides business logic for the form in the customization project, the Acumatica Customization Platform creates a template for the extension and saves it as a *Code* item in the database. Then the platform adds to the graph extension a template for the event

handler highlighted in the list and opens the *Code* item in the *Code Editor*, as shown in the following screenshot.



```

Code Editor: CustomerMaint (Customers)
OPEN SCREEN VIEW SOURCE OVERRIDE METHOD NEW ACTION MOVE TO EXTENSION LIB
14 using PX.Objects.AR;
15 using PX.Objects.CT;
16
17 namespace PX.Objects.AR
18 {
19
20     public class CustomerMaint_Extension:PXGraphExtension<CustomerMaint>
21     {
22
23         #region Event Handlers
24
25         protected void Contact_UsrPersonalID_FieldDefaulting(PXCache cache, PXFieldDefaultingEventArgs e)
26         {
27
28             var row = (Contact)e.Row;
29
30         }
31
32
33

```

Figure: Viewing the event handler template

In the template, the event handler has two parameters, as is defined in the base graph. As a result, the event handler is added to the appropriate event handler collection.

When you publish the customization project, the platform saves the graph extension as a C# source code file in the *App_RuntimeCode* folder of the website. You can later develop the event handler in Microsoft Visual Studio. (See *Integrating the Project Editor with Microsoft Visual Studio* for details.)

To Override an Event Handler

If you customize a form, you may need to override the business logic that processes a control value immediately after the value is changed.

To implement business logic for data changed by the user on an original form, you should add event handlers for data fields (or data records) to the extension of the graph that provides business logic for the form.

To process data modification on a custom form, you should add event handlers for data fields (or data records) to the appropriate custom graph.

You can use the *Layout Editor* to add to the appropriate customization code a workable template of an event handler that overrides the base event handler for a data field (or data record). To do this, perform the following actions:

1. Open the form in the browser and make the control for the field visible on the form.
2. On the form title bar, click **Customization > Inspect Element** to launch the *Element Inspector*.



: If you need to activate the *Element Inspector* for a pop-up panel, dialog box, or other UI element that opens in modal mode and makes the *Customization Menu* unavailable for selection, you can press Control-Alt.

3. On the form, click the control to open the *Element Properties* dialog box for the control.
4. In the dialog box, click **Actions > Customize**.
5. If there is no currently selected customization project and the inspector opens the *Select Customization Project Dialog Box*, elect an existing customization project or create a new one. (See *To Create a New Project* and *To Select an Existing Project* for details.)

If the customization project does not contain a changeset for the form, the *Customization Project Editor* adds a *Page* item for the form to the project, to keep the changeset to the ASPX code of the form in the database. The container of the control is opened in the *Layout Editor*, and the control is selected in the Control Tree of the editor.

6. Select the node of the control in the Control Tree.
7. Click the **Properties** tab item to open the list of properties for the control.
8. In the list, ensure that the `CommitChange` property is set to `True`. (See [Using the CommitChanges Property](#) for detailed information about the property.)
9. Click **Save** to save your changes in the customization project.
10. Click the **Events** tab item to open the list of available event types for the control.
11. In the list, click the type of the event that you want to process for the field to highlight the type, as shown in the screenshot below.
12. On the list toolbar, click **Add Handler > Override Base Method**.

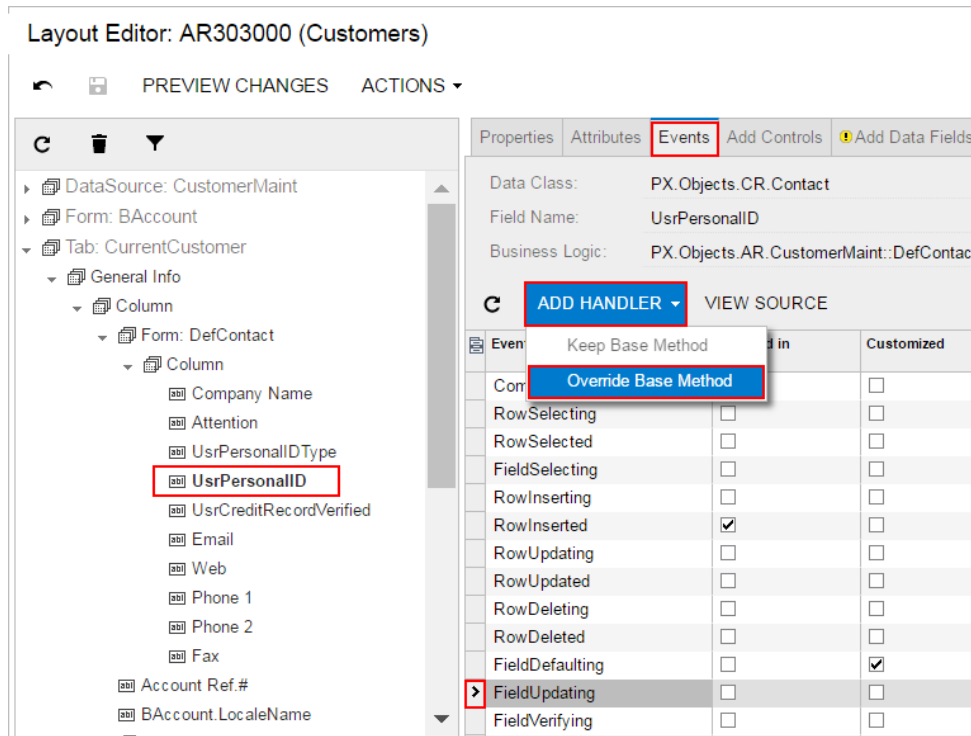


Figure: Adding an event handler for a data field

If there is no extension for the graph that provides business logic for the form in the customization project, the Acumatica Customization Platform creates a template for the extension and saves it as a *Code* item in the database. Then the platform adds to the graph extension a template for the event handler highlighted in the list and opens the *Code* item in the [Code Editor](#), as shown in the following screenshot.

Code Editor: CustomerMaint (Customers)

OPEN SCREEN VIEW SOURCE OVERRIDE METHOD NEW ACTION MOVE TO EXTENSION LIB

```

14 using PX.Objects.AR;
15 using PX.Objects.CT;
16
17 namespace PX.Objects.AR
18 {
19
20     public class CustomerMaint_Extension:PXGraphExtension<CustomerMaint>
21     {
22
23         #region Event Handlers
24
25         protected void Contact_UsrPersonalID_FieldUpdating(PXCache cache, PXFieldUpdatingEventArgs e, PXFieldUpdating InvokeBaseHandler)
26         {
27             if(InvokeBaseHandler != null)
28                 InvokeBaseHandler(cache, e);
29             var row = (Contact)e.Row;
30         }
31
32
33

```

Figure: Viewing the event handler template

In the template, the event handler has an additional parameter to replace the base event handler collection of the graph.

When you publish the customization project, the platform saves the graph extension as a C# source code file in the `App_RuntimeCode` folder of the website. You can later develop the event handler in Microsoft Visual Studio. (See [Integrating the Project Editor with Microsoft Visual Studio](#) for details.)

To Override a Virtual Method

In a graph extension, you can override the virtual methods defined within the base graph. As with the event handlers, you have two options:

- You can define the override method with exactly the same signature—that is, the return value, the name of the method, and all method parameters—as the overridden base virtual method has. As a result, the method is added to the queue of all override methods. When the system invokes the base method, all methods in the queue are executed sequentially, from the first one to the last one. The lower the level the BLC extension has, the earlier the system invokes the override method.
- You can define the override method with an additional parameter, which represents the delegate for one of the following:
 - The override method with an additional parameter from the extension of the previous level, if such a method exists
 - The base virtual method, if no override methods with additional parameters declared within lower-level extensions exist

In both cases, you should attach the `PXOverride` attribute to the override method declared within the BLC extension, as described in the following sections below:

- [Override Method That Is Added to the Override Method Queue](#)
- [Override Method That Replaces the Original Method](#)

Override Method That Is Added to the Override Method Queue

By declaring an override method with exactly the same signature as the overridden base virtual method has, you extend the base method execution. The base BLC method is replaced at run time with the queue of methods, which starts with the base BLC method. When the system invokes the base method, all methods in the queue are executed sequentially, from the first one to the last one. The lower the level the BLC extension has, the earlier the system invokes the override method. If the system has invoked the base method, you have no option to prevent the override method queue from execution. To prevent the base method executions, see the [Override Method That Replaces the Original Method](#) section below.

To add an override method that is added to the override method queue, perform the following actions:

1. Create the graph extension, as described in [To Start the Customization of a Graph](#).
2. In the Code Editor, click **View Source** to view the code of the base graph in the [Source Code Browser](#).



: In an instance of Acumatica ERP, a repository with the original C# source code of the application is kept in the \App_Data\CodeRepository folder of the website.

3. In the browser, select and copy the code of the method.
4. In the Code Editor, paste the method code in the graph extension.
5. In the graph extension, insert the `PXOverride` attribute immediately before the method signature.
6. Clear the method body.
7. Develop the code of the method.
8. Click **Save** in Code Editor to save your changes.

Override Method That Replaces the Original Method

The override method with an additional parameter replaces the base BLC virtual method. When the virtual method is invoked, the system invokes the override method with an additional parameter of the highest-level BLC extension. The system passes a link to the override method with an additional parameter from the extension of the previous level, if such a method exists, or to the base virtual method.

You use a delegate as an additional parameter to encapsulate the method with exactly the same signature as the base virtual method. If the base virtual method contains a list of parameters, then you should not use the same list of parameters when you declare the override method with an additional parameter. To declare a `MyMethod` override method with an additional parameter within a BLC extension, you can use the following example.

```
public class BaseBLCExt : PXGraphExtension<BaseBLC>
{
    public delegate ReturnType MyMethodDelegate([List of Parameters]);
    [PXOverride]
    public ReturnType MyMethod([List of Parameters,] MyMethodDelegate baseMethod)
    {
        return baseMethod(adapter);
    }
}
```

You can decide whether to call the method pointed to by the delegate. By invoking the base method, you also start the execution of the override method queue.

To add an override method that replaces the base BLC virtual method, perform the following actions:

1. Create the graph extension, as described in [To Start the Customization of a Graph](#).
2. In the Code Editor, click **Override Method**, as shown in the screenshot below.
3. In the **Selected** column of the **Select Methods to Override** dialog box, which opens, select the method to be overridden.
4. Click **Save** to add the code template for the override method to the graph extension.

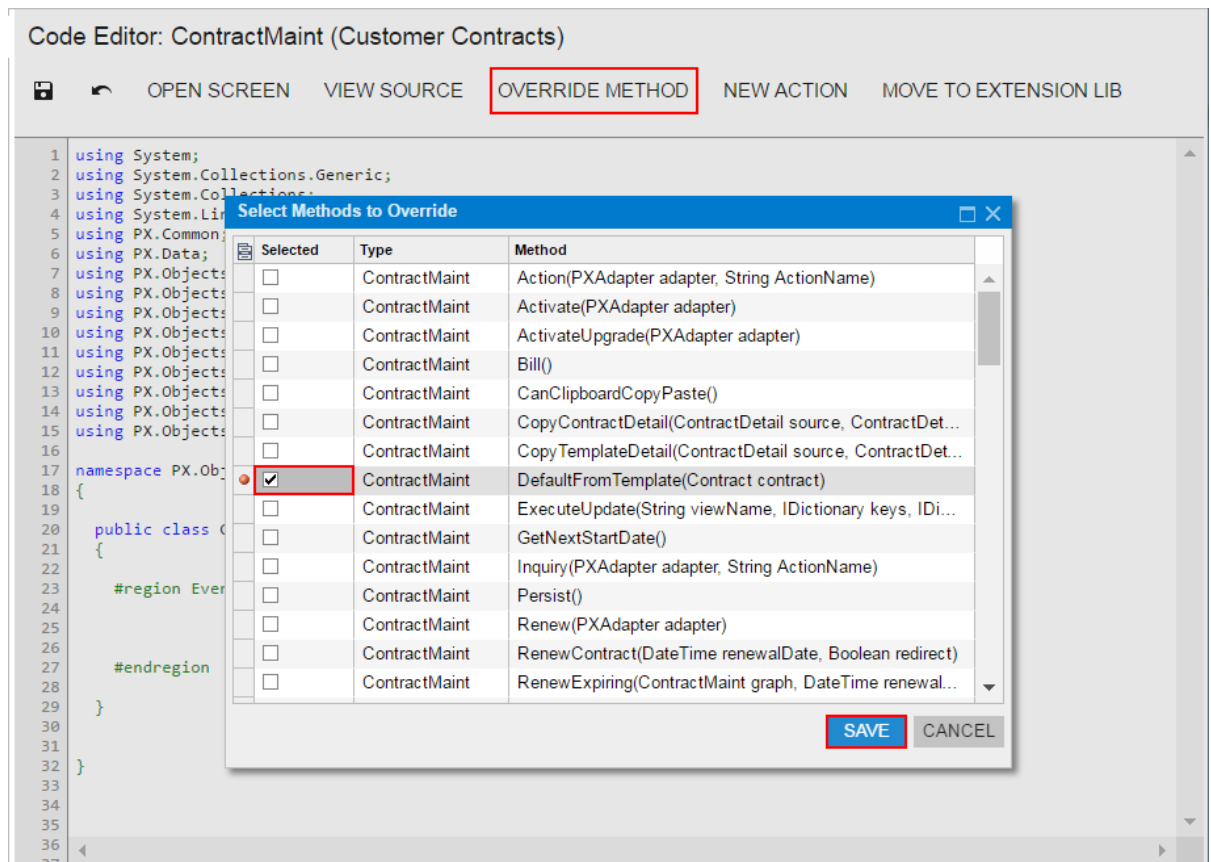


Figure: Selecting a method to be overridden

Once the override method template has been created (see the following screenshot), you can implement the needed code within the template.

```

Code Editor: ContractMaint (Customer Contracts)
OPEN SCREEN VIEW SOURCE OVERRIDE METHOD NEW ACTION MOVE TO EXTENSION LIB

1 using System;
2 using System.Collections.Generic;
3 using System.Collections;
4 using System.Linq;
5 using PX.Common;
6 using PX.Data;
7 using PX.Objects.EP;
8 using PX.Objects.IN;
9 using PX.Objects.CR;
10 using PX.Objects.AR;
11 using PX.Objects.GL;
12 using PX.Objects.CS;
13 using PX.Objects.CM;
14 using PX.Objects;
15 using PX.Objects.CT;
16
17 namespace PX.Objects.CT
18 {
19
20     public class ContractMaint_Extension:PXGraphExtension<ContractMaint>
21     {
22
23         #region Event Handlers
24         public delegate void DefaultFromTemplateDelegate(Contract contract);
25         [PXOverride]
26         public void DefaultFromTemplate(Contract contract, DefaultFromTemplateDelegate baseMethod)
27         {
28             baseMethod(contract);
29         }
30
31
32
33
34         #endregion
35
36
37

```

Figure: Viewing the override method template in the Code Editor

When you publish the customization project, the platform saves the graph extension as a C# source code file in the `App_RuntimeCode` folder of the website. You can later develop the event handler in Microsoft Visual Studio. (See [Integrating the Project Editor with Microsoft Visual Studio](#) for details.)

Data View

In a business logic controller (BLC, also referred to as *graph*), a *data view* is a `PXView` object that is used to access and manipulate data. In an ASPX page, to obtain data for controls, each container has to be bound to a data view of the BLC that is bound to the `PXDataSource` control of the page.

A data view object contains two main parts:

- The BQL command, which is defined by the type of the data view
- The optional delegate, which constructs the data set that is returned instead of the result of the execution of the BQL command

In a graph extension, you can include the following member types:

- Declaration of a custom data view
- Redefinition for an existing data view
- Declaration of the delegate for an existing data view
- An override method for an existing data view delegate

To declare a custom data view in a graph extension, you can follow the instructions described in [To Add a New Member](#).

For detailed information on customizing a data view, see the following topics:

- [To Override a Data View](#)
- [To Add a Data View Delegate](#)
- [To Override a Data View Delegate](#)

To Override a Data View

To modify a data view, you have to redefine the data view in the graph extension class. The data view redefined within a BLC extension completely replaces the base data view within the *Views* collection of a graph instance, including all attributes attached to the data view declared within the base graph. You can either attach the same set of attributes to the data view or completely redeclare the attributes. For details, see [Graph Extensions](#). The data view must have exactly the same identifier, which is referred to in the appropriate container in the ASPX page.

To redefine a data view in the graph extension, perform the following actions:

1. Create the graph extension, as described in [To Start the Customization of a Graph](#), if required.
2. In the Code Editor, click **View Source** to view the code of the base graph in the [Source Code Browser](#).



: In an instance of Acumatica ERP, a repository with the original C# source code of the application is kept in the `\App_Data\CodeRepository` folder of the website.

3. In the browser, select and copy the data view declaration.
4. In the Code Editor, paste the data view declaration in the graph extension.
5. In the graph extension, redefine the data view declaration as required.
6. Click **Save** in the Code Editor to save your changes.

To Add a Data View Delegate

By default, when a data view object is requested by the UI or you invoke the `Select()` method on the object, the system executes the query specified in the data view declaration. However you can define a dynamic query, which is an optional graph method (called the data view delegate) that is executed when the data view is requested. If no dynamic query is defined in the graph, Acumatica Framework executes the BQL statement from the data view declaration.

You can use the data view delegate in the following cases:

- If you construct the query dynamically at run time by adding `Where<>` and `Join<>` clauses depending on some condition, typically a filter
- If the query retrieves data fields that cannot be calculated declaratively by attributes—for instance, if you retrieve values aggregated by calculated data fields
- If the result set has data records that aren't retrieved from the database and are composed dynamically in code

In a graph extension, to define the delegate for a data view, you should redeclare the data view and add a method that has the same name as the data view but uses a different case of the first letter (for example, if the data view name is *MyDataView*, the name of the delegate must be *myDataView*). The delegate returns an `IEnumerable` object, as shown in the code below.

```
// The delegate for the MyDataView data view
protected virtual IEnumerable myDataView()
{
    // The code to be used to construct the query dynamically at run time, execute
    // the query, and return the list of retrieved records
}
```



: When you declare or alter a data view delegate within a graph extension, the new delegate is attached to the corresponding data view. To query a data view declared within the base graph or lower-level extension from the data view delegate, you should redeclare the data view within the graph extension. You do not need to redeclare a generic `PXSelect<Table>` data member in the graph extension when it will not be used from the data view delegate. For details, see [Graph Extensions](#).

You can add a data view delegate to a custom graph or to an extension for an existing graph, which is saved in the appropriate *Code* item of the customization project. See [To Create a Custom Graph](#) and [To Start the Customization of a Graph](#) for details.

If the project has not been published, you can use only the *Code Editor* to add a new member to the graph code.

After the customization project has been published, the code is also saved in the corresponding C# file in the `App_RuntimeCode` folder of the website. (See [Changes in the Application Code \(C#\)](#) for details.) You can open the file in Microsoft Visual Studio and use Visual Studio to add a data view delegate to the graph code. To do this, perform the following actions:

1. In Visual Studio, click **File > Open > Web Site**, and select your Acumatica ERP development environment.
2. Open the `App_RuntimeCode\<CodeItemName>.cs` file.
3. In the code, develop the delegate for the data view.



: An example of implementation of a data view delegate for a custom graph is described in Step 11.7 of the [T300 Acumatica Customization Platform](#) course. See [Declaring or Altering a BLC Data View Delegate](#) for an example of a data view delegate for a graph extension.

4. Debug the code by using Visual Studio, as described in [To Debug the Customization Code](#).
5. Save the file.
6. In the Customization Project Editor, select the **Files** node in the navigation pane.
7. On the Custom Files page, which opens, click **Detect Modified Files** on the page toolbar.



: The customization platform compares the content of each file added to the site with the content of the associated item of the customization project in the database. If a difference is detected, the platform opens the **Modified Files Detected** dialog box so you can resolve the detected conflicts. You can either update the customization project in the database or discard the file changes in the file system to resolve the conflicts.

8. In the **Modified Files Detected** dialog box, which opens, ensure that the modified code file is selected.
9. Click **Update Customization Project** on the dialog box toolbar to update the *Code* item that is saved in the database.

To Override a Data View Delegate

You can modify the data view delegate—that is, the method that is invoked when the records are retrieved through the data view. To do this, perform the following actions:

1. Create the graph extension, as described in [To Start the Customization of a Graph](#), if required.
2. In the Code Editor, click **View Source** to view the code of the base graph in the [Source Code Browser](#).



: In an instance of Acumatica ERP, a repository with the original C# source code of the application is kept in the `\App_Data\CodeRepository` folder of the website.

3. In the browser, select and copy the data view declaration and the data view delegate.



: The override data view delegate must have exactly the same signature—the return value, the name of the method, and any method parameters—as the base data view delegate.

4. In the Code Editor, paste the code in the graph extension.



: When you declare or alter a data view delegate within a graph extension, the new delegate is attached to the corresponding data view. To query a data view declared within the base graph or a lower-level extension from the data view delegate, you should redeclare the data view within the graph extension. You do not need to redeclare a generic `PXSelect<Table>` data member in the graph extension when it will not be used from the data view delegate. For details, see [Graph Extensions](#).

5. In the graph extension, rewrite the data view delegate code as you need to.

6. Click **Save** in the Code Editor to save your changes.

Action

An *action* is a graph member of the `PXAction` type. An action always has the delegate defined. Every action is represented by the `PXAction` object and placed in the *Actions* collection of the appropriate graph. To construct an instance of the `PXAction` class, you use a graph member of the `PXAction` type and a delegate from the highest-level extension discovered.

To modify the business logic of an action that is defined within a graph, you should override the action delegate.

To rename, disable, or hide an action button, you should override the action attributes.

The redefined action delegate must have exactly the same signature—that is, the return value, the name of the method, and any method parameters—as the base action delegate has. You always have to redefine the action delegate to alter either its delegate or the attributes attached to the action. To use the action declared within the base graph or a lower-level extension from the action delegate, you should redefine the generic `PXAction<TNode>` data member within the graph extension. You do not need to redefine the data member when it is not meant to be used from the action delegate. For details, see [Graph Extensions](#). When you redefine an action delegate, you have to repeat the attributes attached to the action. Every action delegate must have `PXButtonAttribute` (or the derived attribute) and `PXUIFieldAttribute` attached.

For detailed information on customizing an action, see the following topics:

- [To Start the Customization of an Action](#)
- [To Override an Action Delegate Method](#)
- [To Rename an Action Button](#)
- [To Disable or Enable an Action](#)
- [To Hide or Show an Action](#)

To Start the Customization of an Action

If you need to customize an action represented by a button on a form, perform the following actions:

1. To find the declaration of the action, do the following:
 - a. Open the form in the browser, and make the button visible on the form (if it isn't when the form is brought up).
 - b. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).
 - c. On the form, click the button to open the [Element Properties](#) dialog box for the button.



: If the action command is a part of an action menu, open the menu to display the command and use the keyboard shortcut Ctrl+Alt+Click to inspect the action item in the menu, as shown in the following screenshot.

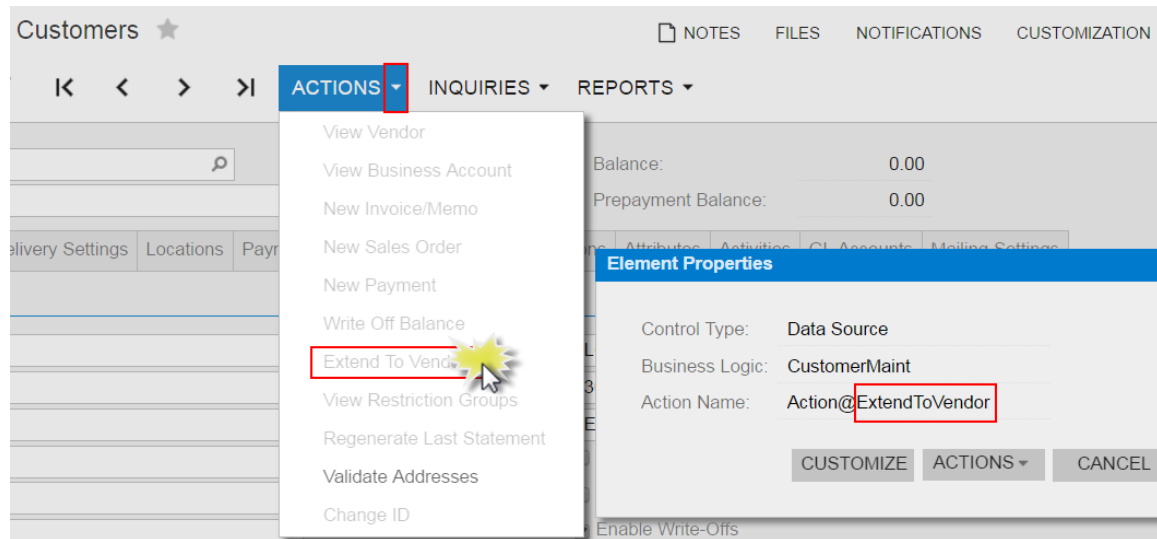


Figure: Displaying the action name in the Element Properties dialog box

- d. In the dialog box, click **Actions > View Business Logic Source** to open the source code of the graph whose name is displayed in the Business Logic box of the Element Properties dialog box.



: In an instance of Acumatica ERP, the repository with the original C# source code of the application is kept in the \App_Data\CodeRepository folder of the website.

- e. In the **Methods** list of the *Source Code* browser, which opens for the graph, search for and click the action name to display the action delegate method in the work area of the browser, as the following screenshot shows.



Figure: Viewing the action delegate method in the Source Code browser

- f. If the search fails, try to find the action declaration in the base class of the graph.



: If the button has a unique name, you can also find the action declaration in the Source Code Browser, as described in [To Find Source Code by a Fragment](#), by using the button name as a code fragment.

2. Explore the action declaration in the source code of the original graph.
3. Select and copy the action declaration.
4. Create an extension for the graph, as described in [To Start the Customization of a Graph](#), if needed.
5. In the graph extension, paste the action declaration and develop the needed code to change the behavior and appearance of the action.

To Override an Action Delegate Method

With the technology based on extension models, an action from base graph is always completely replaced by the identically named action declared within a graph extension.

To override an action delegate method in a graph extension, you should declare both the graph member of the `PXAction` type and the delegate. You should attach a new set of attributes to the action delegate declared within the graph extension. Also, you need to invoke the `Press()` method on the base graph action. Because you have redeclared the member of `PXAction`, you have prevented the action delegate execution from infinite loops.



: If you have a customization that replaces an original action declaration statically, after you upgrade Acumatica ERP to a new version, new functionality of the action may become unavailable.

Do the following to override an action delegate method:

1. Explore the original action declaration and copy the declaration to the graph extension, as described in [To Start the Customization of an Action](#).



Attention: We recommend that you not remove or change any attributes of the action.

2. In the action declaration, replace the action delegate with the following code template.

```
public virtual IEnumerable myAction(PXAdapter adapter)
{
    return Base.MyAction.Press(adapter);
}
#endregion
```

3. In the code template, replace `myAction` and `MyAction` with the appropriate names.
4. In the template, redefine the action delegate arguments and return type based on the signature of the base action delegate.
5. Implement the needed code in the override action delegate.
6. Click **Save** in the Code Editor to save your changes.

To Rename an Action Button

To rename an action button, you should redefine the `DisplayName` parameter of the `PXUIField` attribute for the button.

You can change attributes of an action button in one of the following ways:

- [Dynamically at run time, in the `Initialize\(\)` method of the graph extension](#)
- [Statically, by overriding the action attributes in the graph extension](#)

To Rename an Action Button Dynamically at Run Time

1. Explore the original action declaration, as described in [To Start the Customization of an Action](#) but without copying the declaration to the graph extension.
2. In the graph extension, add the following code.

```
#region Extended initialization

public override void Initialize()
{
    base.Initialize();
    Base.MyAction.SetCaption("NEW NAME");
}
#endregion
```

3. In the added code, replace `MyAction` with the action name and specify the needed button name.
4. Click **Save** in Code Editor to save the changes.

To Rename an Action Button Statically

To override action attributes in a graph extension statically, you should declare both the graph member of the `PXAction` type and the delegate. You should attach a new set of attributes to the action delegate, declared within the graph extension. Also, you need to invoke the `Press()` method on the base graph action. Because you have redeclared the member of `PXAction`, you have prevented the action delegate execution from infinite loops.



: If you have a customization that replaces an original action declaration statically, after you upgrade Acumatica ERP to a new version, the new functionality of the action may become unavailable.

To rename an action button statically, perform the following actions:

1. Explore the original action declaration, and copy the declaration to the graph extension, as described in [To Start the Customization of an Action](#).
2. In the action declaration, specify the required name for the action button in the `PXUIField` attribute, as the following code snippet shows.

```
...
[PXUIField(DisplayName = "NEW NAME", ...)]
...
```



Attention: We recommend that you not remove or change other attributes of the action.

3. Replace the action delegate with the following code template.

```
public virtual IEnumerable myAction(PXAdapter adapter)
{
    return Base.MyAction.Press(adapter);
}
#endregion
```

4. In the code template, replace `myAction` and `MyAction` with the appropriate names.
5. In the template, redefine the action delegate arguments and return type based on the signature of the base action delegate.
6. Click **Save** in the Code Editor to save your changes.

To Disable or Enable an Action

To disable or enable an action button, you can redefine the `Enabled` parameter of the `PXUIField` attribute for the button. However we recommend that you change the attribute dynamically at run time in the `Initialize()` method of the graph extension.



: If you have a customization that replaces an original action declaration statically, after you upgrade Acumatica ERP to a new version, any new functionality of the action may become unavailable.

To disable or enable an action button, do the following:

1. Explore the original action declaration, as described in [To Start the Customization of an Action](#) but without copying the declaration to the graph extension.
2. In the graph extension, add the following code.

```
#region Extended initialization

public override void Initialize()
{
    base.Initialize();
    Base.MyAction.SetEnabled(false);
}

#endregion
```



: To enable the action button, use `true` instead of `false` for the `SetEnabled` method call.

3. In the added code, replace `MyAction` with the action name.
4. Click **Save** in the Code Editor to save your changes.

To Hide or Show an Action

To hide or show an action button, you should redefine the `Visible` parameter of the `PXUIField` attribute for the button.

You can change attributes of an action button by using one of the following approaches:

- [Dynamically at run time, in the `Initialize\(\)` method of the graph extension](#)
- [Statically, by overriding the action attributes in the graph extension](#)
- [Statically, by changing visibility of the button in the ASPX code](#)

To Hide or Show an Action Button at Run Time

1. Explore the original action declaration, as described in [To Start the Customization of an Action](#) but without copying the declaration to the graph extension.
2. In the graph extension, add the following code.

```
#region Extended initialization

public override void Initialize()
{
    base.Initialize();
    Base.MyAction.SetVisible(false);
}

#endregion
```



: To show the action button, use `true` instead of `false` for the `SetVisible` method call.

3. In the added code, replace `MyAction` with the action name.
4. Click **Save** in the Code Editor to save your changes.

To Hide or Show an Action Button Statically

To override action attributes in a graph extension statically, you should declare both the graph member of the `PXAction` type and the delegate. You should attach a new set of attributes to the action delegate declared within the graph extension. Also, you need to invoke the `Press()` method on the base graph action. Because you have redeclared the member of `PXAction`, you prevent the action delegate execution from infinite loops.



: If you have a customization that replaces an original action declaration statically, after you upgrade Acumatica ERP to a new version, any new functionality of the action may become unavailable.

To hide or show the action button, perform the following actions:

1. Explore the original action declaration and copy the declaration to the graph extension, as described in [To Start the Customization of an Action](#).
2. In the action declaration, set the `Visible` parameter of the `PXUIField` attribute to `false`, as the following code snippet shows.



: To show the action button, set the `Visible` parameter to `true`.

```
...
[PXUIField(..., Visible = false)]
...
```



Attention: We recommend that you not remove or change other attributes of the action.

3. Replace the action delegate with the following code template.

```
public virtual IEnumerable myAction(PXAdapter adapter)
{
    return Base.MyAction.Press(adapter);
}
```

4. In the code template, replace `myAction` and `MyAction` with the appropriate names.
5. In the template, redefine the action delegate arguments and return type based on the signature of the base action delegate.
6. Click **Save** in the Code Editor to save your changes.

To Hide or Show an Action Button in the ASPX Code

If a callback command for the button is declared in the `PXDataSource` control, you can hide the button by customizing the ASPX code. To do this, perform the following actions:

1. Open the form in the browser and make the button visible on the form.
2. On the form title bar, click **Customization > Inspect Element** to launch the [Element Inspector](#).
3. On the form, click the button to open the [Element Properties](#) dialog box for the button.




: If the button is a part of an action menu, inspect the menu button instead of the button.

4. In the dialog box, click **Customize**.
5. If there is no currently selected customization project and the inspector opens the [Select Customization Project Dialog Box](#), select an existing customization project or create a new one.

If there is no a *Page* item for the form in the customization project, the Acumatica Customization Platform creates such item, adds the item to the project, and opens the form in the [Layout Editor](#).



: For users' convenience, when the Layout Editor is opened, the Control Tree displays only the node of the object selected with the Element Inspector. You can click **Show/Hide All Controls** () on the tree toolbar to view all the controls of the form in the tree.

6. In the Control Tree of the editor, click the arrow left of the `PXDataSource` node to expand it.
7. Select the `PXDSCallbackCommand` element with the button name.
8. Click the **Properties** tab item to open the list of properties for the callback.
9. Set the `Visible` property of the element to `False` to hide the button.



: To show the button, set `Visible` property to `True`.

10. Click **Save** to save your changes to the customization project.

Customizing the Database Schema

You can use the platform to customize the database of an Acumatica ERP instance. The database customization can include changes in both the data and the schema of the database.

To change the database schema, you can add to a customization project the following scripts:

- A script to create a custom table
- A script to create a custom column in an existing table
- A script to create an extension table for an existing table
- A custom script


When you publish a customization project with a database script, which changes the database schema, the script is executed. Changes to the database schema aren't deleted when you unpublish the project or delete the script and publish the project once more. You have to remove the changes to the database schema manually.

For detailed information on creating custom database scripts and adding the scripts to a customization project, see the following topics:

- [To Create a Custom Table](#)
- [To Create a Custom Column in an Existing Table](#)
- [To Create an Extension Table](#)
- [To Add a Custom SQL Script to a Customization Project](#)

To Create a Custom Table

To add a custom table to a customization project, perform the following actions:

1. Create the needed table in the database by using a database administration tool, such as SQL Server Management Studio.
 -  : You have to use a naming convention that provides unique names for your custom tables so that they do not have the names of existing tables of Acumatica ERP.
2. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
3. Click **DB Scripts** in the navigation pane to open the Database Scripts page.
4. On the page toolbar, click **Add New Record (+)**.
5. In the *SQL Script Editor*, which opens, select the custom table in the **DBObject Name** box.
6. In the editor, select the **Import Table Schema from Database** check box, as shown in the screenshot below.
7. Click **OK** to make the Acumatica Customization Platform generate the table schema and add the schema to the customization project.

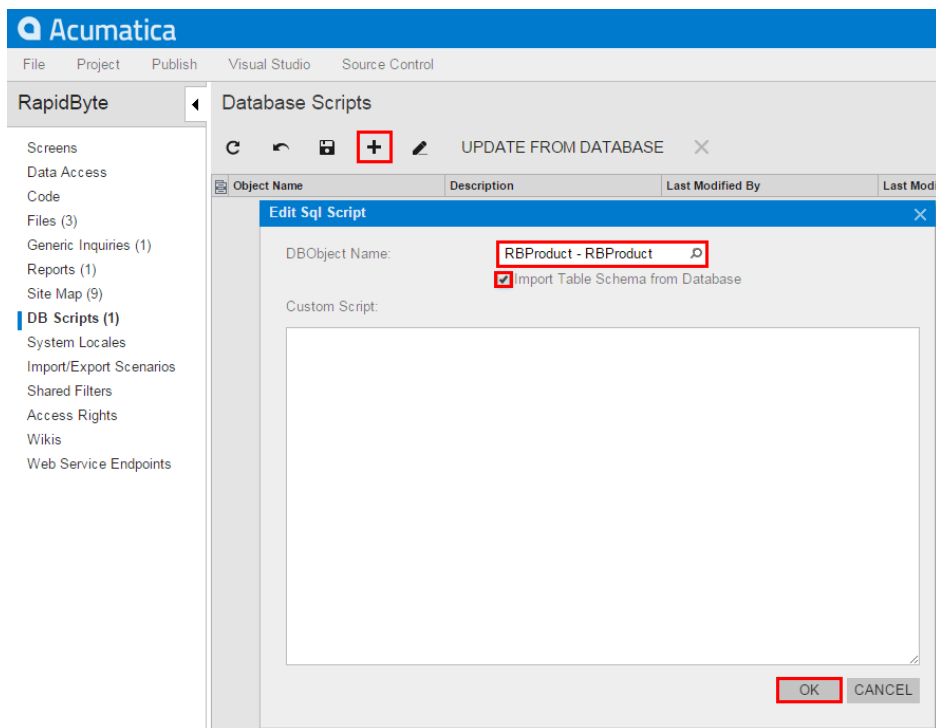


Figure: Adding a custom table to the project

Adding the table schema is the preferred way of adding custom tables to the project. When you publish the customization project, the platform executes each SQL script of the project to update the database. If an *Sql* item contains a custom database table definition, to update the database with the table schema, the Acumatica Customization Platform checks whether a table with this name already exists in the database. If the table exists, the platform generates SQL statements to alter the existing table so that it matches the schema. The platform doesn't drop the existing table and keeps any data in it. This makes it easier to deploy a newer version of the customization project to a system that is already in use. If the table doesn't exist, the platform generates SQL statements to create the table. SQL statements are generated in the SQL dialect of the database management system. Therefore, if you add custom tables to the project by table schema, you keep the customization project independent from the database management system that hosts the database of Acumatica ERP. Below is an example of the table schema of the custom table *Product*.

```
<Sql TableName="RBProduct" TableSchemaXml="#CDATA">
  <CDATA name="TableSchemaXml"><![CDATA[<table name="RBProduct">
  <col name="ProductID" type="Int" identity="true" />
  <col name="ProductCD" type="NVarChar(15)" />
  <col name="ProductName" type="NVarChar(50)" />
  <col name="Active" type="Bit" />
  <col name="StockUnit" type="NVarChar(20)" />
  <col name="UnitPrice" type="Decimal(19,6)" />
  <col name="MinAvailQty" type="Decimal(25,6)" />
  <col name="TStamp" type="Timestamp" />
  <index name="RBProduct_PK" clustered="true" primary="true" unique="true">
    <col name="ProductID" />
  </index>
</table>]]></CDATA>
</Sql>
```

Alternatively, you can add custom tables by adding a custom SQL script that creates the table in the project.

To Create a Custom Column in an Existing Table

Because you can need a new column in an existing database table only for a bound data field, the Acumatica Customization Platform automatically creates a custom column when you create a custom bound data field by using the New Field wizard.

To add a custom field as a new column that will be appended to the original table of Acumatica ERP, perform the following actions:

1. Open the **Create New Field** dialog box for the data access class *by using the Data Class Editor* or *by using the Layout Editor*.
2. In the dialog box, specify the custom field parameters and select *DBTableColumn* in the **Storage Type** box to make the system append the column to the original table in the database.
3. Click **OK** to add the field to the data access class.

The New Field wizard includes an XML declaration of the new field in the *DAC* item for the modified data access class. If the *DAC* item for the modified data access class is absent in the customization project, the wizard creates the appropriate item. Also, the wizard creates a *Table* item with a description of the custom column to be created in the database table for the custom field.

After you publish the customization project at least once, the database schema is changed. Changes to the database schema aren't deleted when you unpublish the project or delete the *DAC* and *Table* items and publish the project once more. You have to remove the changes manually.

To Create an Extension Table

The Acumatica Customization Platform provides the following options to create a custom column in a table of the database:

- Add the column to the original table
- Create a separate table that is an extension of the original table, and add the column to the extension table

In some cases, you may need to store new field values in a separate table or use this table independently from the original database table. You can use the Acumatica Customization Platform to create a separate extension database table. This table must include all main key fields from the original database table and your new fields. You should create a DAC extension that holds the new fields and is mapped to the extension table. The platform will automatically synchronize the extension table with the original database table by expanding every create, retrieve, update, and delete operation on the base (original) DAC to each discovered DAC extension that is mapped to an extension table.

To declare a DAC extension that is mapped to an extension table, perform the following actions:

1. Create an extension table in the database according to the requirements described in *Requirements for an Extension Table Schema*.
2. Declare a DAC extension mapped to the created table, as described in *DAC Extension Mapped to an Extension Table*.

By using the `PXTable` attribute, you specify that the DAC extension is mapped to the extension table with the same name.

Requirements for an Extension Table Schema

You should ensure that the following requirements are met when you create an extension table:

- The extension table should have the same set of main key columns as the original database table has.
- The extension table must include the following columns if they are declared within the original database table:
 - `[CompanyID][int] NOT NULL`

- [DeletedDatabaseRecord][bit] NOT NULL
- For an extension table that can be used separately, you should also declare the following audit columns:
 - [tstamp][timestamp] NULL
 - [CreatedByID][uniqueidentifier] NOT NULL
 - [CreatedByScreenID][char] (8) NOT NULL
 - [CreatedDateTime][datetime] NOT NULL
 - [LastModifiedByID][uniqueidentifier] NOT NULL
 - [LastModifiedByScreenID][char] (8) NOT NULL
 - [LastModifiedDateTime][datetime] NOT NULL

The example below shows the declaration of the `InventoryItemTableExtension` extension table. Notice that this table will not be used independently from the original database table.

```
CREATE TABLE [dbo].[InventoryItemTableExtension]
(
    [CompanyID] [int] NOT NULL,
    [InventoryID] [int] NOT NULL,
    [DeletedDatabaseRecord] [bit] NOT NULL,
    [ExtTableDescr] [nvarchar] (256) NULL
    CONSTRAINT [InventoryItemTableExtension_PK] PRIMARY KEY CLUSTERED
    (
        [CompanyID] ASC,
        [InventoryID] ASC
    )
    WITH (PAD_INDEX = OFF,
        STATISTICS_NORECOMPUTE = OFF,
        IGNORE_DUP_KEY = OFF,
        ALLOW_ROW_LOCKS = ON,
        ALLOW_PAGE_LOCKS = ON)
    ON [PRIMARY]
) ON [PRIMARY]
ALTER TABLE [dbo].[InventoryItemTableExtension] ADD DEFAULT ((0))
    FOR [DeletedDatabaseRecord]
GO
```

DAC Extension Mapped to an Extension Table

You can define a DAC extension mapped to an extension table by using either the default *Inner Join* way or the optional *Left Outer Join* way. To specify which way is used, you need to set the value for the `IsOptional` parameter of the `PXTable` attribute. By default, the system sets this parameter value to *false*. You need to specify the `IsOptional` parameter value only if you need to set its value to *true*.

```
[PXTable(IsOptional = value)]
public class TableExtension : PXCACHEExtension<BaseDAC>
{ ... }
```

If the DAC contains surrogate and natural keys, then the `PXTable` attribute attached to the DAC extension should reference the surrogate key as well as other database key fields (but not the natural key). If the DAC doesn't have surrogate and natural keys, no key fields should be specified in the `PXTable` attribute. See the following code example of the declaration of the `PXTable` attribute with key references.

```
[PXTable(typeof(BaseDAC.surrogateKey),
    typeof(BaseDAC.otherDBKeyField),
    IsOptional = value)]
public class TableExtension : PXCACHEExtension<BaseDAC>
```



```
{ ... }
```



: The natural key is a user-friendly value that is not used as a key in the database. The surrogate key, which is the internal value corresponding to the natural key, is not shown to the user and is assigned by the system. When you use a natural key, the DAC field that serves as a surrogate key is bound to the database key column, but is not marked as key within its attributes.

In the sample code shown below, the `Location` database table contains both the surrogate `LocationID` key and the natural `LocationCD` key. The `Location` database table main key contains the `BAccountID` and `LocationID` columns. Because `LocationCD` is a natural key, we need to specify the corresponding surrogate key, `LocationID`, as well as the other database key field, `BAccountID`, in the `PXTable` attribute.

```
[PXTable(typeof(Location.locationID),
         typeof(Location.bAccountID))]
public class LocationTableExtension : PXCacheExtension<Location>
{ ... }
```

The Left Outer Join Way

The following example shows the declaration of a DAC extension mapped to an extension table with the *Left Outer Join* way. Notice that the `IsOptional` parameter of the `PXTable` attribute is set to `true`.

```
[PXTable(IsOptional = true)]
class LeftJoinTableExtension : PXCacheExtension<BaseDAC>
{
}
```

The *Left Outer Join* way covers the common steps required to add an extension tables to a customization project. For details on populating an extension table with records, see [SQL Script Editor](#).

The *Left Outer Join* way:

- Can be used when the original and extension database tables are not necessarily synchronized.
- Causes an extension table record to be created when the appropriate original table record is created or updated.
- Never excludes an original database table record from the result set.
- Calculates the default field values if no extension table record is found.
- Can be used as a standalone DAC.

The following example shows the declaration of the `InventoryItemTableExtension` DAC extension mapped to the extension table with the *Left Outer Join* way.

```
[PXTable(typeof(InventoryItem.inventoryID), IsOptional = true)]
public class InventoryItemTableExtension : PXCacheExtension<InventoryItem>
{
    #region ExtTableDescr
    public abstract class extTableDescr : PX.Data.IBqlField
    {
    }
    [PXDBString(255)]
    [PXDefault("Additional description")]
    [PXUIField(DisplayName = "Ext_Table Description")]
    public string ExtTableDescr { get; set; }
    #endregion
}
```

Suppose that you have added the corresponding **Ext_Table Description** control to the header area of the Stock Items (IN.20.25.00) form. If you open the form, by clicking navigation buttons on the form

toolbar, you can ensure that all the stock items are visible, while the **Ext_Table Description** control has the default *Additional description* value set, as the screenshot below illustrates.

The screenshot shows the Acumatica interface for the 'New York - Stock Items' form. The 'Ext_Table Description' field is highlighted with a red box and contains the text 'Additional description'. The 'Inventory ID' field contains '301KITSTD3'. The 'Item Status' is 'Active' and the 'Description' is 'Std kit #3'. The 'Ext_Table Description' field is also highlighted with a red box. The 'General Settings' tab is selected, and the 'ITEM DEFAULTS' section shows 'Item Class' as 'MISC - Miscellaneous' and 'Type' as 'Finished Good'. The 'UNIT OF MEASURE' section shows 'Base Unit' as 'PC'. The 'PHYSICAL INVENTORY' section shows 'PI Cycle' and 'ABC Code' fields.

Figure: Exploring the behavior of the customized Stock Items form

If you update a data record (by changing the value of any control, including **Ext_Table Description**), a new database record is added to the extension table, as the following two screenshots illustrate.

The screenshot shows the Acumatica interface for the 'New York - Stock Items' form. The 'Ext_Table Description' field is highlighted with a red box and contains the text 'After adding and saving any extension comment, a new record is added to the extension table'. The 'Inventory ID' field contains '301KITSTD3 - Std kit #3'. The 'Item Status' is 'Active' and the 'Description' is 'Std kit #3'. The 'Ext_Table Description' field is also highlighted with a red box. The 'General Settings' tab is selected, and the 'ITEM DEFAULTS' section shows 'Item Class' as 'MISC - Miscellaneous' and 'Type' as 'Finished Good'. The 'UNIT OF MEASURE' section shows 'Base Unit' as 'PC'. The 'PHYSICAL INVENTORY' section shows 'PI Cycle' and 'ABC Code' fields.

Figure: Entering and saving the description in the new control

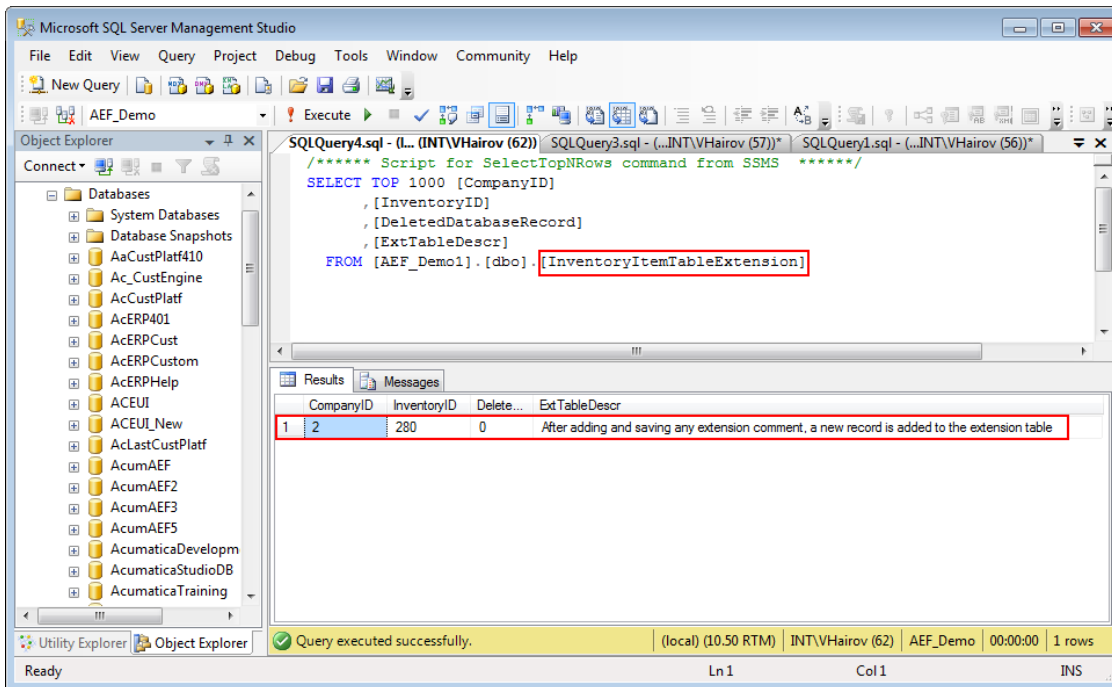


Figure: Analyzing the record added to the new database table

When you use the *Left Outer Join* way, data records within the original and extension tables are not necessarily synchronized (as the figure below illustrates); data record synchronization works as follows:

- If an appropriate data record does not exist within an extension table when the system queries the original table, the system automatically generates and assigns default values to every field of the DAC extension that is mapped to the extension table; otherwise, DAC extension field values are read from the database.
- When a new record is inserted into the original table, the system automatically inserts a new record into the extension table.
- If a data record does not exist within an extension table when the system updates the original table, the system automatically inserts a data record into the extension table. Otherwise, if there are no modified fields of the DAC extension that is mapped to the extension table, the system does not update the extension table data record.
- When the system deletes the data record in the original table, it automatically deletes the appropriate data record from the extension table, if such a record exists.



:

To use an extension table independently from the original database table, you should declare a data view by using a DAC extension that is mapped to an extension table as the main DAC, as shown below.

```
public class BaseBLCExt : PXGraphExtension<BaseBLC>
{
    public PXSelect<TableExtension> Objects;
}
```

In the example of the data view declaration above, extension table data records have no reference to the original database table records. You can work with these data records just as you would work with any other DAC instance.

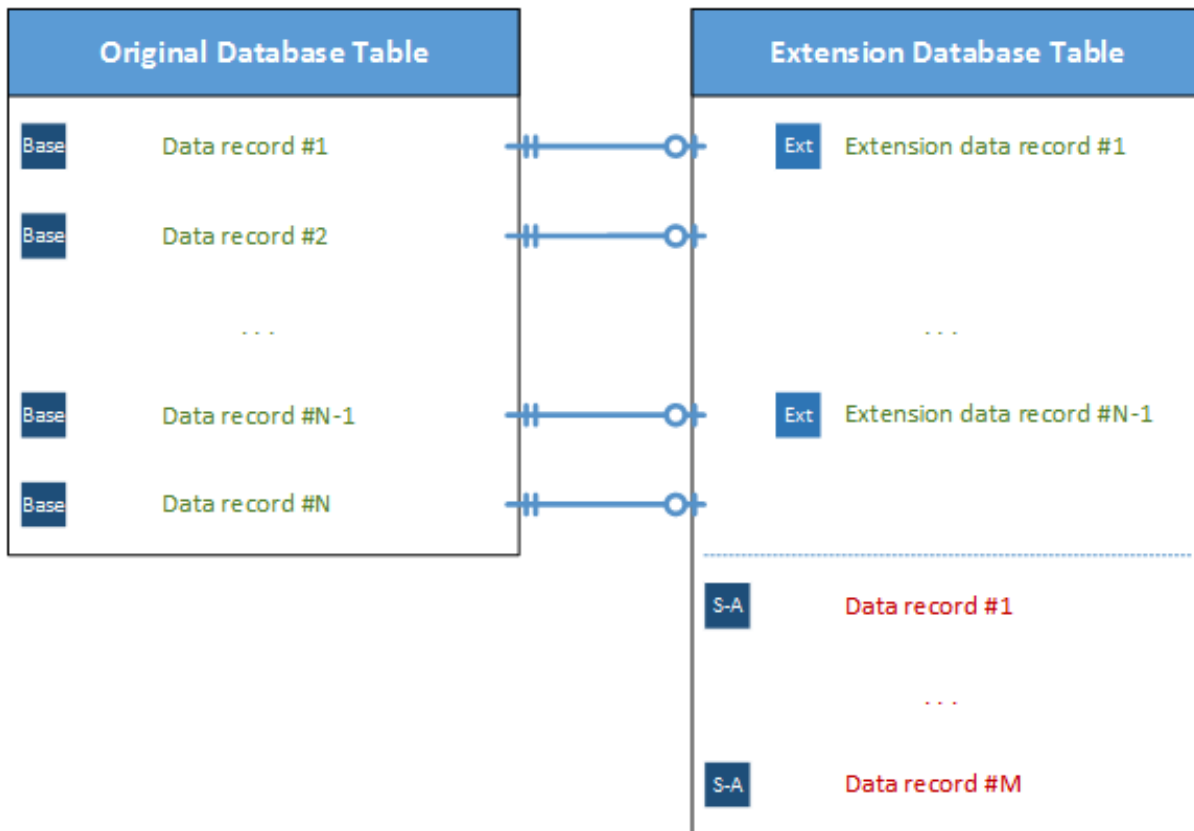


Figure: Exploring data record synchronization when the Left Outer Join way is used

The Inner Join Way

The *Inner Join* is the default way.

The following example shows the declaration of a DAC extension mapped to an extension table with the default *Inner Join* way.

```
[PXTable]
class InnerJoinTableExtension : PXCacheExtension<BaseDAC>
{
}
```

The *Inner Join* way:

- Can be used only when the original table and extension table are always synchronized.
- Causes an extension table record to be automatically created only when the appropriate original database table record is created.
- Requires the main key column values to be copied from the original table to each extension table.
- Excludes an original database table record from the result set when no corresponding extension table record is found.
- Can be used as a standalone DAC.

The sample code below shows the declaration of the `InventoryItemTableExtension` DAC extension, which is mapped to the extension table by using the default *Inner Join* way.

```
[PXTable(typeof(InventoryItem.inventoryID))]
public class InventoryItemTableExtension : PXCacheExtension<InventoryItem>
```

```

{
  #region ExtTableDescr
  public abstract class extTableDescr : PX.Data.IBqlField
  {
  }
  [PXDBString(255)]
  [PXDefault("Additional description")]
  [PXUIField(DisplayName = "Ext_Table Description")]
  public string ExtTableDescr { get; set; }
  #endregion
}

```

If you again open the Stock Items (IN.20.25.00) form, by clicking navigation buttons on the form toolbar of the Stock Items form, you will see that only one stock item is visible, and it has the modified **Ext_Table Description**.

To have access to the other database table records, you need to populate the extension table with the appropriate records. You can do this by using the following script.

```

INSERT INTO [dbo].[InventoryItemTableExtension]
SELECT
    CompanyID,
    InventoryID,
    0,
    N'Additional description'
FROM [dbo].[InventoryItem]
WHERE NOT EXISTS
(
    SELECT * FROM [dbo].[InventoryItemTableExtension] AS t
    WHERE t.CompanyID = [dbo].[InventoryItem].CompanyID
        AND t.InventoryID = [dbo].[InventoryItem].InventoryID
)
GO

```

After you copy data records from the `InventoryItem` database table to the `InventoryItemTableExtension` user extension table, you will notice that all stock items are visible again.

When you use the *Inner Join* way, the data records within the base and extended tables must always be synchronized (see the screenshot below). With this way, data record synchronization works as follows:

- If an appropriate data record does not exist within an extension table when the system queries the original table, the system excludes the original table record from the result set.
- When a new record is inserted into the original table, the system automatically inserts a new record into the extension table.
- When the system updates the data record in the original table, it does not update the extension table if there are no modified fields of the DAC extension that is mapped to the extension table.
- When it deletes the original table data record, the system automatically deletes the appropriate data record from the extension table, if such a record exists.



: To use the extension table independently from the original database table, you should declare a data view by using a DAC extension, which is mapped to an extension table, as the main DAC (for details, see the [The Left Outer Join Way](#) section).

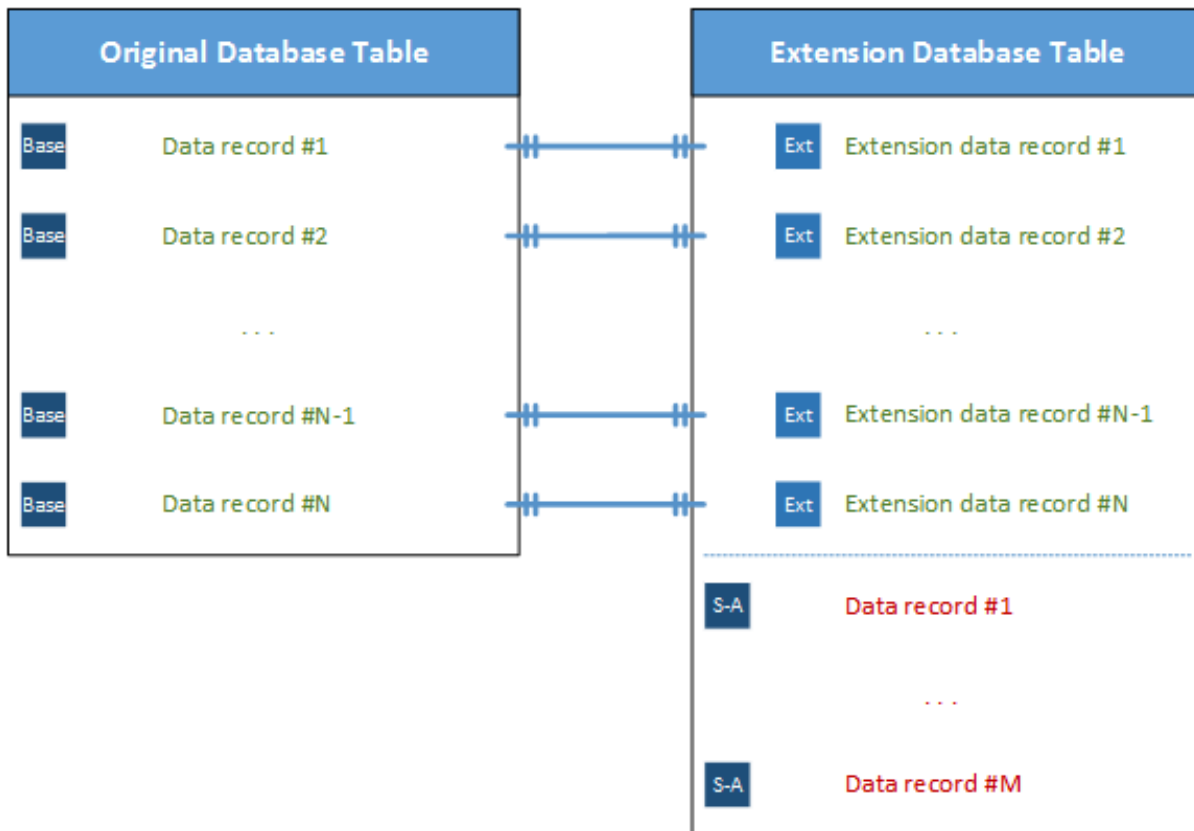


Figure: Exploring data record synchronization when you use the Inner Join way

To Add a Custom SQL Script to a Customization Project



: Although we provide these instructions, we do not recommend that you add a custom SQL script to a customization project for the following reasons:

- Because Acumatica ERP supports multi-tenancy, it is difficult to create an SQL script that correctly creates a database object.
- It is difficult to properly specify and use the company mask in custom database objects.
- If you include in a customization project an SQL script written for MS SQL, you will need to avoid applying the customization to a website on MySQL Server, because an SQL script created for MS SQL Server will not work properly on MySQL Server.



Warning: A possible result of a custom SQL script is the loss of the integrity and consistency of data.

To add a custom SQL script to a customization project, perform the following actions:

1. Prepare and debug the SQL script with a database administration tool, such as SQL Server Management Studio. (See [Creating a Custom SQL Script](#) for details.)
2. Open the customization project in the Customization Project Editor. (See [To Open a Project](#) for details.)
3. Click **DB Scripts** in the navigation pane to open the Database Scripts page.
4. On the page toolbar, click **Add New Record (+)**.
5. In the **DObject Name** box of the [SQL Script Editor](#), which opens, specify the name of the script to be used as the object name of the customization item.
6. In the editor, enter the SQL script into the **Custom Script** text area.

7. In the editor, click **OK** to add the script to the customization project.

If a custom script causes an error, the error message will appear during the publication process when the system executes the custom scripts.



: We really don't want you to do this, but if you plan to add a custom SQL script to a customization project, we recommend that you first test the custom SQL script on MS SQL and MySQL.

Integrating the Project Editor with Microsoft Visual Studio

You do not need to install MS Visual Studio to customize the Acumatica ERP UI or to develop a simple changes of business logic, because the platform includes the [Customization Tools](#), which are designed for these purposes and available in your web browser. However you can use Visual Studio to develop the customization code.

If you plan to use Visual Studio on the server that hosts the website of Acumatica ERP 6.1, the environment where you install and use Acumatica Framework or Acumatica ERP should meet particular requirements, which are described in [System Requirements for Acumatica Framework 6.1](#).

To use Visual Studio while customizing Acumatica ERP, you can select one of the following approaches:

- Create an [Extension Library](#), develop the customization code in Visual Studio, compile the DLL binary file within the `Bin` folder of the website, and add this custom file to the customization project
- Develop and debug the customization code of a customization project on the fly by using Visual Studio

This part of the guide describes the second approach, because the first one is obvious and does not require any integration between the Customization Project Editor and Visual Studio.

To use the second approach, you should perform the following steps each time you want to use Visual Studio to develop and debug the customization code on the fly:

1. In [Data Class Editor](#), create extensions for existing data access classes if required, as described in [To Create a DAC Extension](#).
2. In [Code Editor](#), for business logic controllers, create new classes or class extensions that contain required code templates, as described in [To Create a Custom Graph](#) and [To Start the Customization of a Graph](#).
3. Publish the customization project to obtain the corresponding C# files within the `App_RuntimeCode` website folder. (See [Performing the Publication Process](#) for details.)
4. By using Visual Studio, develop, modify, and debug the customization code in these files on the fly. (See [To Debug the Customization Code](#) for details.)
5. In the Customization Project Editor, update the code in the customization project, as described in [To Synchronize Code Changes with the Customization Project](#).

In This Chapter

- [To Work with a Code Item](#)
- [To Work with Data Access Classes](#)
- [To Debug the Customization Code](#)
- [To Synchronize Code Changes with the Customization Project](#)

To Work with a Code Item

The [Code Editor](#) does not provide facilities you can use in Microsoft Visual Studio to develop and debug any source code in customization projects. To start working with a *Code* item in Visual Studio, perform the following instructions:

1. Publish the customization project, as described in [Publishing Customization Projects](#).

After you have published the customization project, the `<CodeItemName>.cs` file with the item code is placed in the `App_RuntimeCode` folder of the website.

2. Launch Visual Studio.
3. To open the `<CodeItemName>.cs` file in Visual Studio, on the main menu, click **File > Open > File**, and select the file in the **Open File** dialog box, which opens.



: If you need to debug the customization code, you have to open the website instead of the file. See [To Debug the Customization Code](#) for details.

4. After you have finished editing the source code of the item, click **Save** to save your changes in the file.

Because the `App_RuntimeCode` folder contains the customization code that is published, the changes in the file are immediately applied to the application instance.

5. Open or refresh the form that uses the code, to make sure that the business logic has been changed for the form.



: If an error exists in the changed file, the website might be corrupted. You can recover the website, for example, in one of the following ways:

- Debug the code
 - Undo the changes
 - Publish the customization project again to update the files in the `App_RuntimeCode` folder
 - Delete the file from the `App_RuntimeCode` folder
6. To update the `Code` item in the customization project, follow the instructions described in [To Synchronize Code Changes with the Customization Project](#).

To Work with Data Access Classes

You can use Microsoft Visual Studio to work with a data access class (DAC) that has been added to a customization project.

A customization project can contain code for a DAC in the following item types (see [Supported DAC Extension Formats](#) for details):

- As a `DAC` item with XML data for an existing DAC
- As a `Code` item with the code of an existing DAC extension
- As a `Code` item with the code of a custom DAC
- In a `File` item that keeps a `.dll` file of the extension library that contains the binary code of a DAC extension or custom DAC

Once a customization project is published, you can use Visual Studio to develop, modify, and debug the customization code of a `Code` item on the fly. However you cannot use this approach for a `DAC` item.

If you need to work in Visual Studio with the code of the DAC whose changeset is added to a customization project as a `DAC` item in XML format, perform the following actions:

1. Follow the instructions described in [To Convert a DAC Item to a Code Item](#) to convert the item to the `Code` item.
2. Follow the instructions described in [To Work with a Code Item](#) to change the code of the DAC extension.

To Debug the Customization Code

The [Code Editor](#) does not provide facilities you can use to debug a code item of a customization project. We recommend that you use Microsoft Visual Studio to develop and debug any source code in customization projects. To start debugging an existing `Code` item in Visual Studio, perform the following instructions:

1. Publish the customization project, as described in [Publishing Customization Projects](#).


2. In the file system, open in a text editor the `web.config` file located in the website folder.
3. In `<system.web>` tag of the file, locate the `<compilation>` element.
4. Set to `true` the `debug` attribute of the element, as follows.

```
<compilation debug="true" ...>
```



: See [Debug Mode in ASP.NET Applications](#) for details.

5. In a browser, launch your instance of Acumatica ERP to have a running process in the system.
6. Launch Visual Studio.
7. To open the website of your instance of Acumatica ERP in Visual Studio, on the main menu, click **File > Open > Web Site**, and select the website folder in the **Open Web Site** dialog box, which opens.
8. In the Solution Explorer of Visual Studio, expand the `App_RuntimeCode` website folder, and double-click the file of the `Code` item to open it.
9. In the item source code, which opens, set a breakpoint on the line of the code where you need to stop the process for debugging.
10. On the main menu, click **Debug > Attach to Process**, and in the **Attach to Process** dialog box, which opens, select the `w3wp.exe` process in the list of **Available Processes**.

 : In the dialog box, ensure that the **Show processes from all users** check box is selected. If the check box is cleared, the list will not display the `w3wp.exe` process record.
11. In the browser, open the form of Acumatica ERP whose business logic code you want to debug.
12. In the browser, run an operation that invokes the fragment of the code that contains the breakpoint.

The process will be stopped at the breakpoint, and you can debug the code in Visual Studio.

To Synchronize Code Changes with the Customization Project

When you create a `Code` item in a customization project, Acumatica Customization Platform performs the following actions:

- Saves the item content in the database as a `Code` item
- Adds the item ID to the customization project

When you publish the project, the platform creates the `<CodeItemName>.cs` file in the `App_RuntimeCode` website folder of the current instance of Acumatica ERP.

If you use the [Code Editor](#) to modify a `Code` item and click **Save** on the page toolbar, the platform saves the changes only in the database. To update files in the `App_RuntimeCode` folder, you have to publish the project again.

If you have modified the item code in Microsoft Visual Studio, you have to synchronize the changes with the item in the database. To do that, do the following:

- Follow the instructions described in [To Update a File Item in a Project](#) to update files in the customization project.

For more information about the synchronization mechanism provided by the Acumatica Customization Platform, see [Detecting the Project Items Modified in the File System](#).

Integrating the Project Editor with a Version Control System

If a customization task is large or complex or it is implemented by a group of developers, you might need to use a version control system for the customization project. The platform supports the integration of the *Customization Project Editor* with a version control system.

The Customization Project Editor includes the **Source Control** menu, which contains the following commands to integrate the editor with a version control system (see the screenshot below):

- **Save Project to Folder:** Saves the customization project as a set of files to a local folder that is used for integration with a source control system. When you invoke this action, the system opens the **Saves Project to Folder** dialog box so that you can select the name and location of the folder inside a repository.
- **Open Project from Folder:** Loads the customization project from the repository.
- **Setup Source Control:** Opens the **Source Control Setup** dialog box, which you can use to specify a configuration string for connection to a version control system, if required. For example, to control versions, you would set up the configuration string for the Team Foundation Server (TFS), as described in *To Integrate the Customization Project Editor with TFS*, but it is not needed for Git (see *To Integrate the Customization Project Editor with Git* for details).

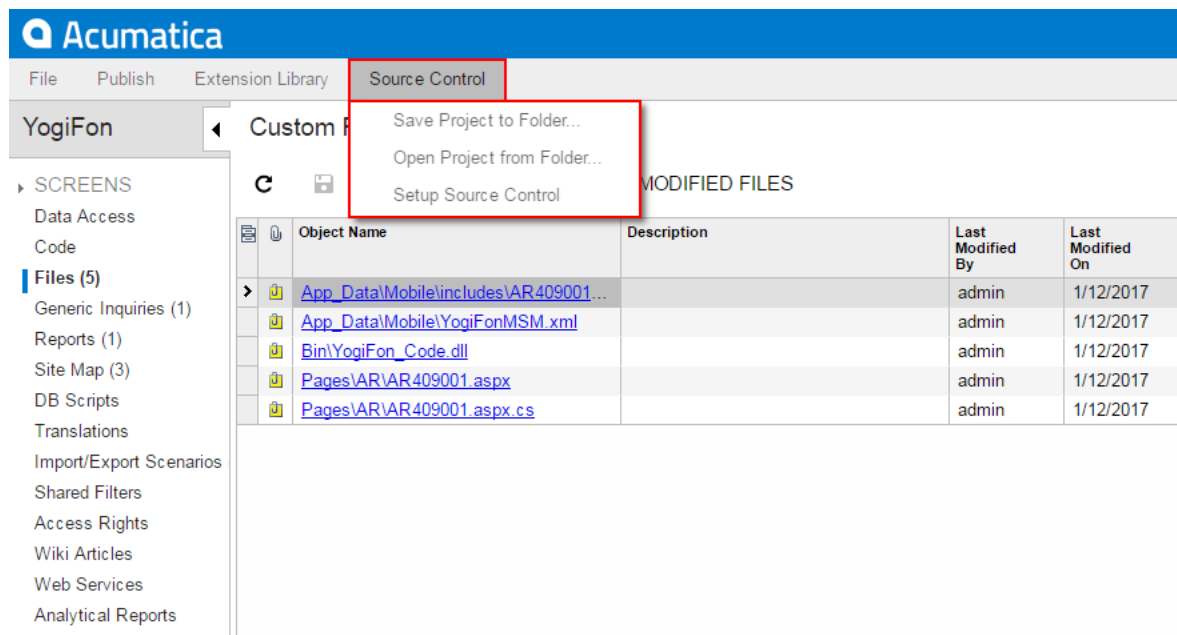


Figure: Viewing the commands for integration with a version control system

For detailed information on integrating the editor with a version control system, see the following topics:

- [To Save a Project to a Local Folder](#)
- [To Update the Content of a Project from a Local Folder](#)
- [To Configure a Connection String](#)
- [To Integrate the Customization Project Editor with TFS](#)
- [To Integrate the Customization Project Editor with Git](#)

To Save a Project to a Local Folder

You can save a customization project as set of files to a local folder that can be used for integration with a source control system. To do this, perform the following actions:

1. Open the project in the *Customization Project Editor*.
2. Click **Source Control > Saves Project to Folder**, as the screenshot below shows.
3. In the **Saves Project to Folder** dialog box, which opens, do the following:
 - a. In the **Parent Folder** selector, select the parent folder.
 - b. In the **Project Name** box, specify the name of the new folder to be used as the project storage.
4. Click **OK**.

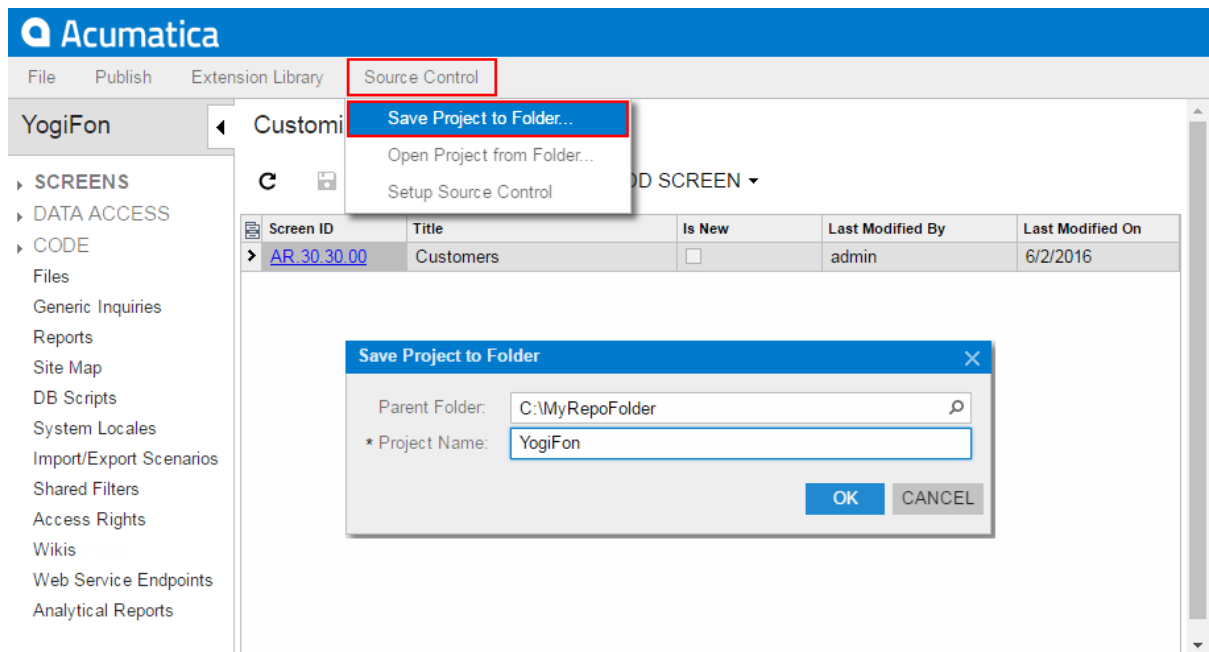


Figure: Saving the project to a local folder

Within the selected parent folder, the platform creates the folder with the project name that you have specified. This folder includes at least the `_project` subfolder that contains an XML file for each item of the project, as the following screenshot shows.

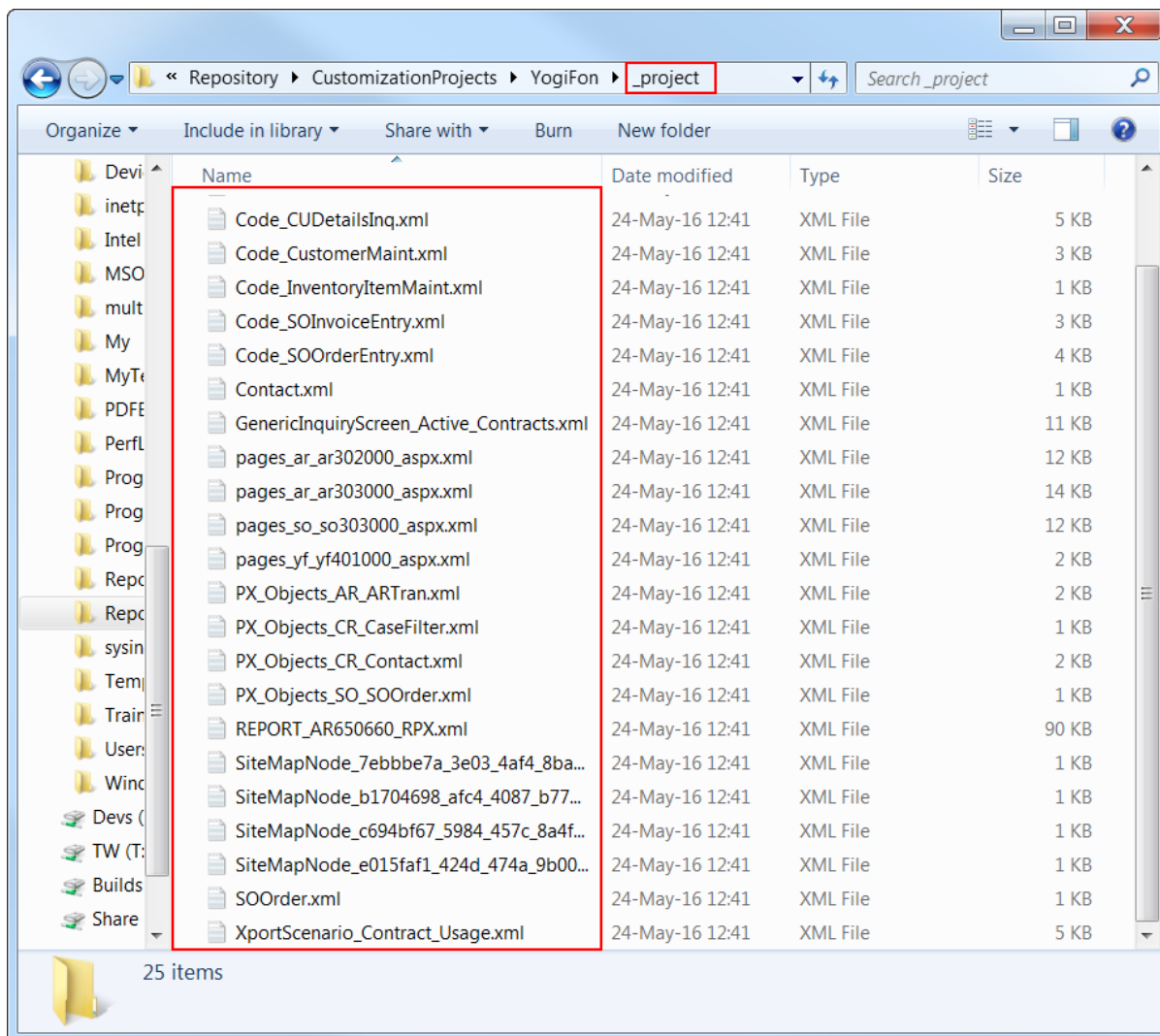


Figure: Viewing the content of the .project folder

If the customization project contains custom files, the platform keeps the paths to these files. Therefore, the project folder includes the corresponding folders for these custom files, as shown in the following screenshot.

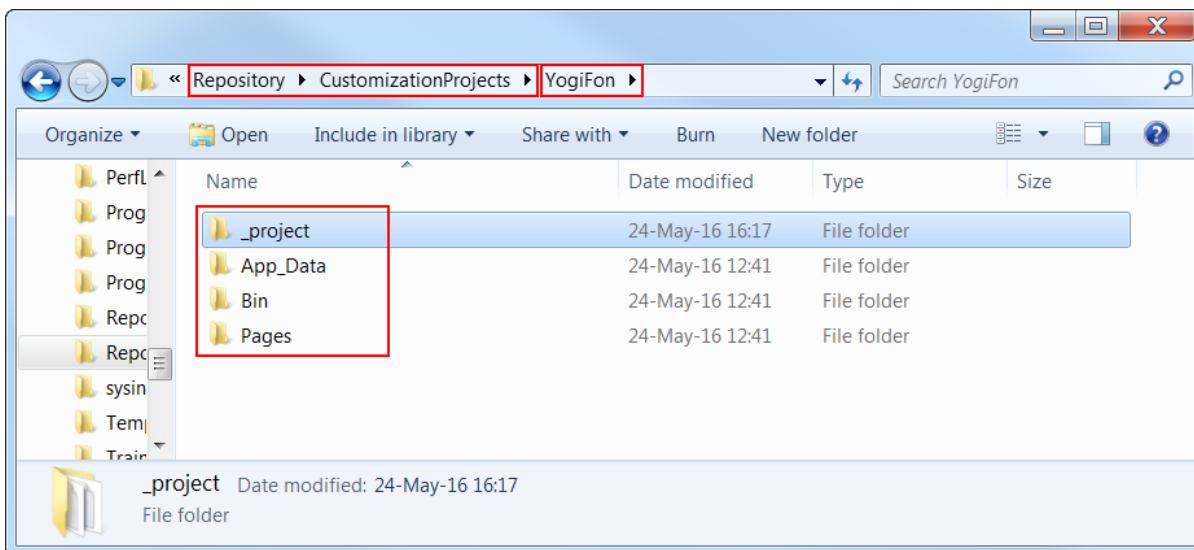


Figure: Viewing the folders inside the project folder

To Update the Content of a Project from a Local Folder

If you have saved a customization project to a local folder by invoking the **Source Control > Saves Project to Folder** command of the *Customization Project Editor*, you can update the content of the project that is currently opened in the editor with the data from the folder.

To do this, perform the following actions:

1. On the editor menu, click **Source Control > Open Project from Folder**, as shown in the screenshot below.
2. In the **Containing Folder** box of the **Open Project from Folder** dialog box, which opens, select the local folder that contains the files of the needed customization project.



Attention: If the folder is empty, the content of the project in the database is cleared.

3. Click **OK**.

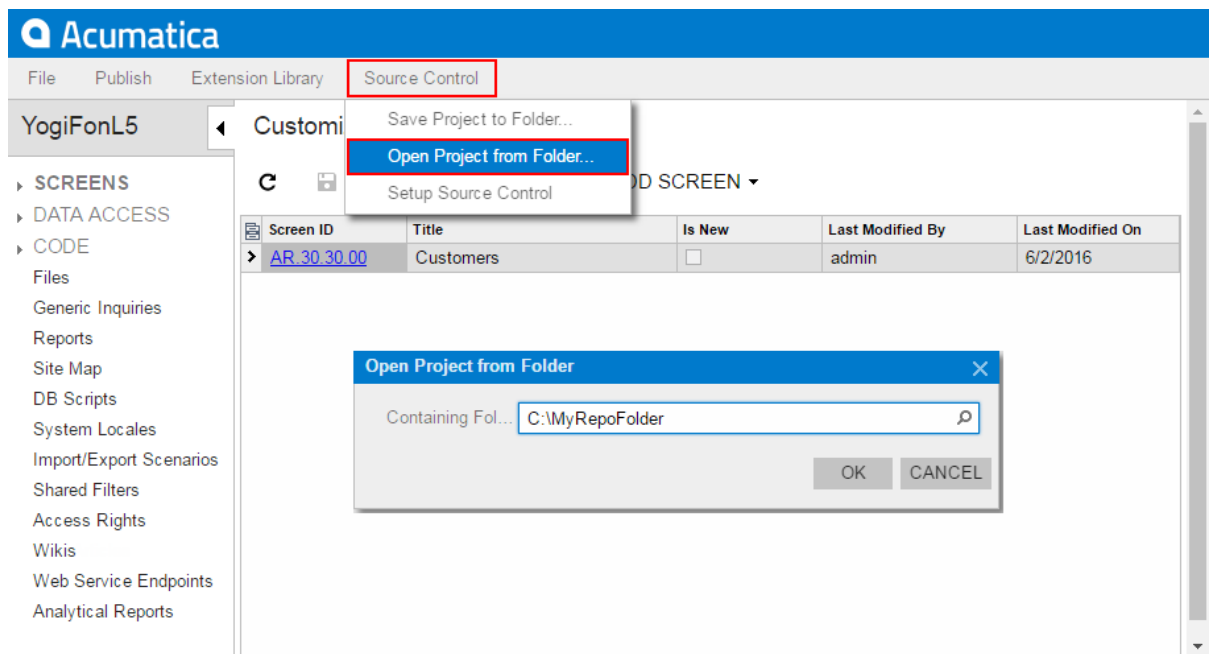


Figure: Loading the customization project from the repository folder

To Configure a Connection String

To use a self-made or third-party plug-in to integrate with a version control system, you can specify the configuration string required to connect the *Customization Project Editor* with the version control system through the plug-in. To do this, perform the following actions:

1. Open the customization project in the Customization Project Editor.
2. Click **Source Control > Setup Source Control**, as the screenshot below shows.
3. In **Config** box of the **Source Control Setup** dialog box, which opens, specify the plug-in to be used and the data required by the plug-in to connect to the version control system.
4. Click **OK**.

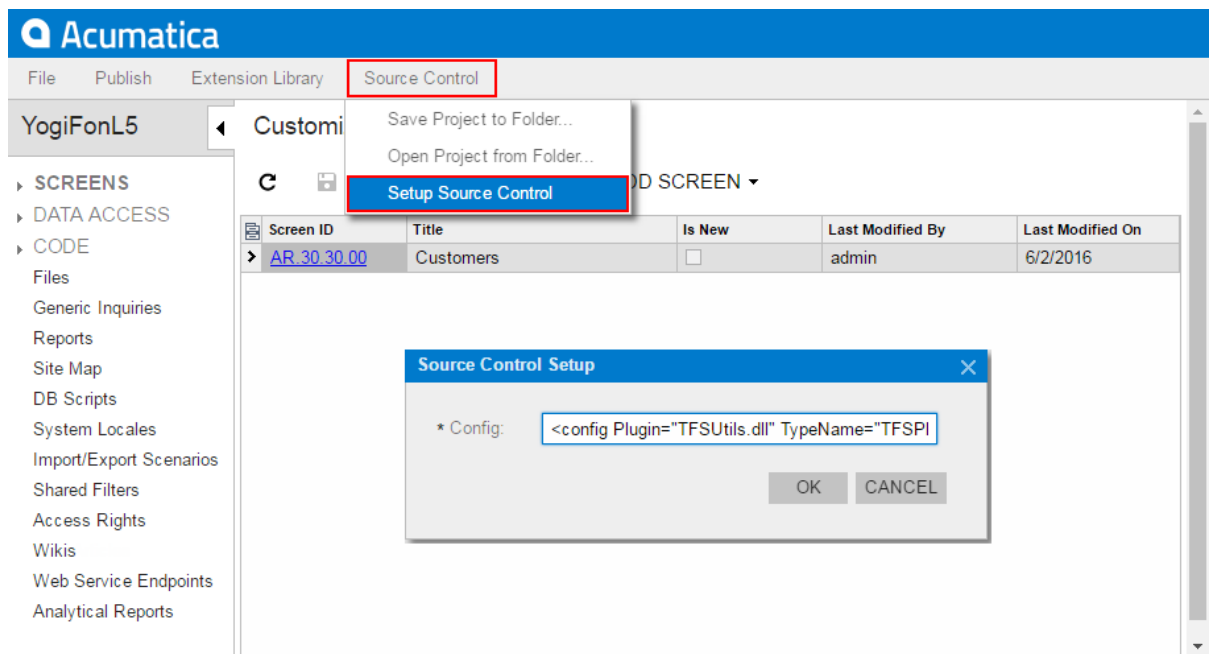


Figure: Setting a configuration string for a version control system

The editor tries to connect with the version control system by using the specified configuration string. If the connection fails, the editor displays an error message with the issue description, as the following screenshot shows.

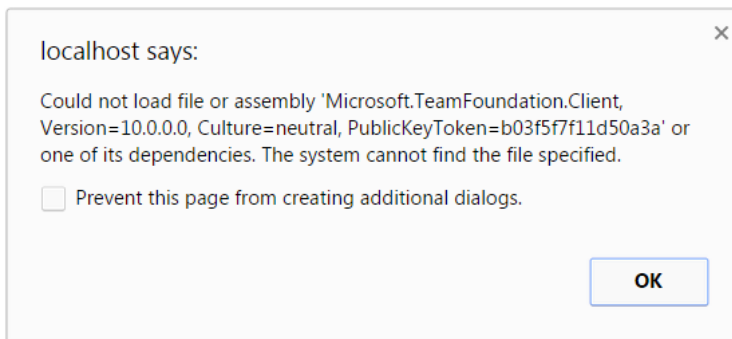


Figure: Viewing the box with a connection error message



: You can develop a plug-in such as `TFSUtils.dll`, which has been created by Acumatica for TFS. If needed, you can decompile this plug-in or ask for the source code to use it as a sample.

To Integrate the Customization Project Editor with TFS

You can use the `TFSUtils` plug-in by Acumatica to integrate the *Customization Project Editor* with Team Foundation Server (TFS). The `TFSUtils.dll` plug-in file is located in the `\Customization\SourceControl` folder of the website, as shown in the following screenshot.

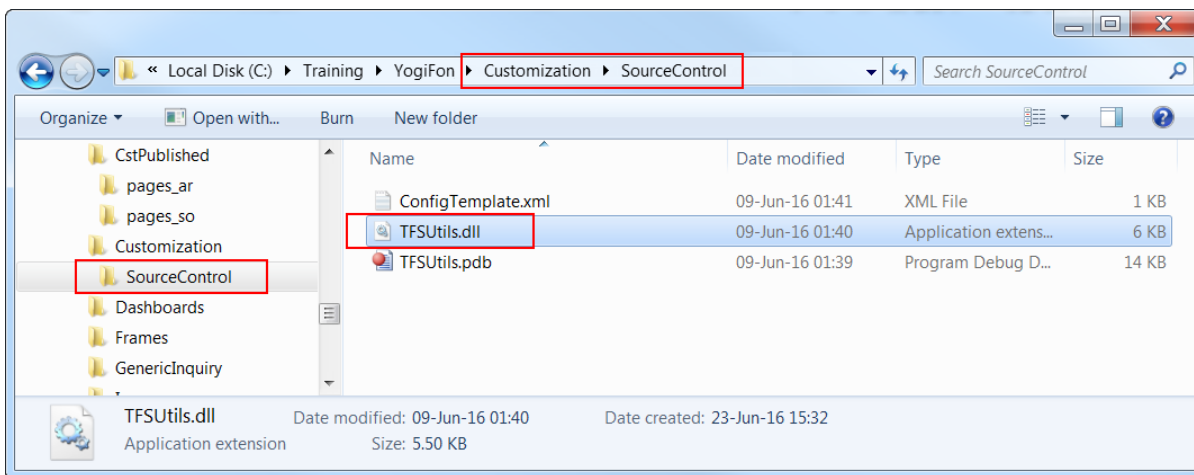


Figure: Viewing the TFSUtils.dll plug-in



: For the configuration string, the platform provides a template that is located in the \Customization\SourceControl\ConfigTemplate.xml file.

To set up the configuration string for TFS, perform the following actions:

1. In the Customization Project Editor, click **Source Control > Setup Source Control**.
2. In the **Source Control Setup** dialog box, which opens, specify the configuration string that contains the following information:
 - **Plugin**: The file name of the plug-in that has to be used by the platform for TFS
 - **TypeName**: The type of the plug-in
 - **workspace**: A set of working folder mappings, which represent the location of your client-side folders on your local disk and the corresponding repository folders
 - **user**: Your user account in Windows
 - **server**: The URL of TFS
 - **psw**: The password for your user account in Windows
3. Click **OK** to force the editor to connect with TFS by using the specified configuration string.

To Integrate the Customization Project Editor with Git

To start tracking changes in a customization project by using Git, perform the following actions:

1. Create a remote Git repository to be used as the central repository for your projects.
2. Copy a version of the repository to your development environment.
3. Save the customization project to a folder within the local copy of the repository.
4. Commit the changes that are made in the local storage to the central repository.

Before starting the project development, a customizer has to do the following each time:

1. Pull the content of the central repository to his or her local storage to update the customization project content.
2. Load the project from the local storage, as described in [To Update the Content of a Project from a Local Folder](#).

After the project is modified, the customizer has to do the following:

1. Update the project in the local storage, as described in [To Save a Project to a Local Folder](#).
2. Commit the changes to the central repository.

Troubleshooting Customization

This part is intended to describe how to solve issues that occur while you are developing or applying a customization. It will be under ongoing development to cover all the approaches and ways required to solve each typical issue that is result of an error in customization or a misunderstanding of the platform features.

In This Part

- [To Discover the Method That Has Thrown an Exception](#)
- [To Write to the Trace Log from the Code](#)
- [To Log All Exceptions to a File](#)
- [To Debug the Customization Code](#)
- [To Validate a BQL Statement](#)
- [To Measure the Execution Time of a BQL Statement](#)
- [To Discover the Cause of Performance Degradation](#)
- [To Force the Platform to Execute Database Scripts](#)
- [To Resolve Issues While Upgrading a Customized Website](#)

To Discover the Method That Has Thrown an Exception

If an exception has occurred at run time and a message box with an error or a warning has been displayed on the form, you can easily discover the method that has thrown the exception. To do this, perform the following actions:

1. Read the message, and close the message box.
2. On a form of Acumatica ERP, click **Help > Trace**, as shown in the following screenshot.

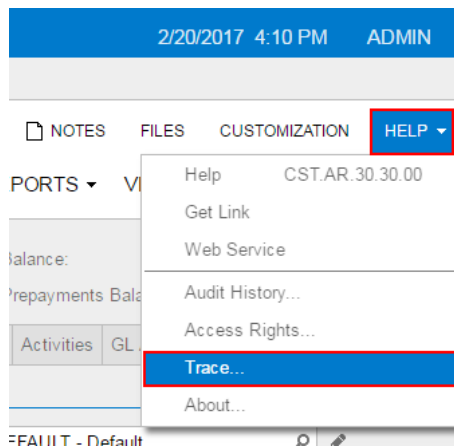


Figure: Opening the Acumatica Trace page

3. On the Acumatica Trace page, which opens to display the trace log, find the appropriate error message, and click the arrow left of the **Details** label for the message to show the detailed information on the error.

In the list of the method calls that were on the stack at the moment of the error, the upper method is the one that threw the exception.



: By default, the trace log is stored on the Acumatica ERP server for the current user session. If the user opens two Acumatica ERP forms simultaneously in two tabs or windows of a browser, the Acumatica ERP server creates a subsession for each opened form and stores the trace logs separately for each subsession. After a session or subsession of the user is completed (due to the user logging off or being timed out), the server clears the appropriate log.

To save the trace logs of all subsessions of all users in a single file on the server, a user with the *Administrator* role has to change the `web.config` file of the Acumatica ERP instance to configure the appropriate trace provider. See the *Trace* section of *Using Logs* for details.

For example, if you open the *Customers* (AR.30.30.00) form of an Acumatica ERP instance in which the *YogiFon* customization project is published (see the *T300 Acumatica Customization Platform* training course for details) and click **Verify Credit Record** on the form toolbar, the system shows an error message, as the following screenshot shows.

The screenshot shows the Acumatica ERP interface for the 'Customers' form. The 'Verify Credit Record' button is highlighted in red. A modal dialog box is displayed in the foreground with the title 'localhost says:' and the message 'The credit verification failed. Personal ID is invalid.' The dialog also includes a checkbox for 'Prevent this page from creating additional dialogs.' and an 'OK' button.

Figure: Viewing a message box with an error

On the Acumatica Trace page, as the following screenshot shows, you can find the error message. After you expand the error details, you can see that the exception has been thrown by the `CustomerMaint_Extension.VerifyCreditRecord()` method.

Acumatica Trace:

[Expand All](#) [Collapse All](#)

Error: **Error #14: Inserting 'GL Batch' record raised at least one error. Please review the errors.** [Send](#)

Raised At: 2/20/2017 7:16:28 PM Screen: GL.30.10.00 Command: Save

▶ Details:

Error: **The credit verification failed. Personal ID is invalid.** [Send](#)

Raised At: 2/20/2017 7:15:23 PM Screen: AR.30.30.00 Command: VerifyCreditRecord

▼ Details:

```

2/20/2017 7:15:23 PM Error:
The credit verification failed. Personal ID is invalid.

at PX.Objects.AR.CustomerMaint_Extension.VerifyCreditRecord()
at PX.Data.PXAction`1.<<DisplayClass3_U.<.ctor>B__U(PXAdapter adapter)
at PX.Data.PXAction`1.a(PXAdapter A_0)
at PX.Data.PXAction`1.d__31.MoveNext()
at PX.Data.PXAction`1.d__31.MoveNext()
at PX.Web.UI.PXBaseDataSource.tryExecutePendingCommand(String viewName, String[] sortColumns, Boolean[] descendings, Object[] searches,
Object[] parameters, PXFilterRow[] filters, DataSourceSelectArguments arguments, Boolean& closeWindowRequired, Int32& adapterStartRow,
Int32& adapterTotalRows)
at PX.Web.UI.PXBaseDataSource.ExecuteSelect(String viewName, DataSourceSelectArguments arguments, PXDSSelectArguments pxarguments)

```

Version: 6.10.0121 Customization: YogiFon

Figure: Viewing the needed method in the list of method calls

To Write to the Trace Log from the Code

When you are debugging the customization code, you may need to know that a method or code fragment has been executed (for example, during an action or event). You can write information to the trace log directly from your code by using one of the following static methods of the `PXTrace` static class:

- `public static void WriteError(string message)`
- `public static void WriteError(Exception e)`
- `public static void WriteWarning(string message)`
- `public static void WriteWarning(Exception e)`
- `public static void WriteInformation(string message)`
- `public static void WriteInformation(Exception e)`
- `public static void WriteEvent(Event item)`

You would generally use these methods temporarily, for debugging purposes. Thus, to avoid changes in the customization project, you can add this method to the customization code that is already applied to the application instance. To do this, perform the following actions:

1. Make sure that the customization project that contains the code you are debugging is published. If the project is unpublished, publish it, as described in [Publishing Customization Projects](#).
2. Launch Microsoft Visual Studio.
3. In Visual Studio, open the file with the code, which is currently saved in the `App_RuntimeCode` website folder.
4. Add a call of one of the methods listed above to the appropriate code fragment.
5. Save your changes to the file.
6. In the browser, open or refresh the form that you are troubleshooting.
7. On the form, perform the sequence of actions that invokes the changed code.
8. Click **Help > Trace** to open the Acumatica Trace page.

- On the page, explore the trace log to make sure that the changed code fragment has been executed.



: When you publish or export the project, the Acumatica Customization Platform compares the code in the App_RuntimeCode website folder with the appropriate code in the database and discovers the files changed in the file system. If a modified file is found, there is a conflict, and the platform opens the **Modified Files Detected** dialog box to give you the option to update the files in the project or discard the changes. See *Detecting the Project Items Modified in the File System* for details.

For example, suppose that you open in Visual Studio the `SOOrderEntry.cs` file of the *YogiFon* customization project, which is published (see the *T300 Acumatica Customization Platform* training course for details) and add the `PXTrace.WriteInformation("SOOrder_CustomerID_FieldUpdated")` method call to the `SOOrder_CustomerID_FieldUpdated()` event handler of the `SOOrderEntry_Extension` class, as shown in the following screenshot. In this case, the specified text string is written to the trace log every time the `CustomerID` field is updated on the Sales Orders (SO.30.10.00) form.

```

SOOrderEntry.cs
ERP_6.00.1681
PX.Objects.SO.SOOrderEntry_Extension
SOOrder_CustomerID_FieldUpdated(PXCache sender, PXF
0 references
55 protected void SOOrder_CustomerID_FieldUpdated(PXCache sender, PXFieldUpdatedEventArgs e)
56 {
57     SOOrder order = e.Row as SOOrder;
58     BAccount customer = PXSelectorAttribute.Select<SOOrder.customerID>(sender, order) as BAccount;
59     PXTrace.WriteInformation("SOOrder_CustomerID_FieldUpdated");
60     if (customer != null)
61     {
62         Contact defContact = PXSelect<Contact,
63             Where<Contact.bAccountID, Equal<Required<Contact.bAccountID>>,
64             And<Contact.contactID, Equal<Required<Contact.contactID>>>>>
65             .Select(Base, customer.BAccountID, customer.DefContactID);
66         if (defContact != null)
67         {
68             ContactExt contactExt = PXCache<Contact>.GetExtension<ContactExt>(defContact);
69             sender.SetValue<SOOrderExt.usrCRVerified>(order, contactExt.UsrCreditRecordVerified);
70         }
71     }
72 }
73

```

Figure: Adding the WriteInformation method call to the code

If you were to refresh the form in the browser, change the customer ID in the **Customer** box, and click **Help > Trace** to open the Acumatica Trace page, the trace log would contain a record with the text specified in the `WriteInformation` method call, as the following screenshot shows.

Acumatica Trace:

[Expand All](#)
[Collapse All](#)

Information:	SOOrder_CustomerID_FieldUpdated	Send
	Raised At: 2/21/2017 7:09:52 PM Screen: SO.30.10.00 Command: Save Details:	
Error:	The credit verification failed. Personal ID is invalid.	Send
	Raised At: 2/21/2017 7:09:08 PM Screen: AR.30.30.00 Command: VerifyCreditRecord Details:	

Version: 6.10.0121 Customization: YogiFon

Figure: Viewing the information record in the trace log



: By default, the trace log is stored on the Acumatica ERP server for the current user session. If the user opens two Acumatica ERP forms simultaneously in two tabs or windows of a browser, the Acumatica ERP server creates a subsession for each opened form and stores the trace logs separately for each subsession.

After a session or subsession of the user is completed (due to the user logging off or being timed out), the server clears the appropriate log.

To save the trace logs of all subsessions of all users in a single file on the server, a user with the *Administrator* role has to change the `web.config` file of the Acumatica ERP instance to configure the appropriate trace provider. See the *Trace* section of *Using Logs* for details.

To Log All Exceptions to a File

Acumatica ERP provides a mechanism you can use for catching and logging all exceptions in the system. (See the *First-Chance Exception Log* section of *Using Logs* for details.) If you have been assigned the *Administrator* role, you can activate this mechanism and specify the name of the log file. To do this, perform the following actions:

1. In the file system, open in a text editor the `web.config` file located in the website folder.
2. Within the `<appSettings>` tag of the file, turn on the `EnableFirstChanceExceptionsLogging` key, as follows.

```
<add key="EnableFirstChanceExceptionsLogging" value="True" />
```

3. If you need to change the log file name, which is `firstchanceexceptions.log` by default, you can specify the needed name in the `FirstChanceExceptionsLogFileName` key, as follows.

```
<add key="FirstChanceExceptionsLogFileName" value="MyLog.log" />
```



: By default, the first-chance exception log file is saved in the `App_Data` folder of the website.

You can open the log file in a text editor to view the content of the first-change exception log, as the following screenshot shows.

```
firstchanceexceptions.log - Notepad
File Edit Format View Help
=====
Feb/22 13:38:24.3731
System.Reflection.ReflectionTypeLoadException: Unable to load one or more of the requested types. Retrieve the LoaderExceptions
at System.Reflection.RuntimeModule.GetTypes(RuntimeModule module)
at System.Reflection.RuntimeAssembly.get_DefinedTypes()

Loader exceptions:
System.IO.FileLoadException: Could not load file or assembly 'System.Web.Mvc, Version=1.0.0.0, Culture=neutral, PublicKeyToken=
File name: 'System.Web.Mvc, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'

WRN: Assembly binding logging is turned OFF.
To enable assembly bind failure logging, set the registry value [HKLM\Software\Microsoft\Fusion!EnableLog] (DWORD) to 1.
Note: There is some performance penalty associated with assembly bind failure logging.
To turn this feature off, remove the registry value [HKLM\Software\Microsoft\Fusion!EnableLog].

Stack trace:
at PX.Data.PXFirstChanceExceptionLogger.a(Object A_0, FirstChanceExceptionEventArgs A_1)
at System.Reflection.RuntimeModule.GetTypes(RuntimeModule module)
at System.Reflection.RuntimeAssembly.get_DefinedTypes()
at Autofac.Util.AssemblyExtensions.GetLoadableTypes(Assembly assembly)
at System.Linq.Enumerable.<SelectManyIterator>d__16.MoveNext()
at System.Linq.Enumerable.WhereEnumerableIterator.MoveNext()
at Autofac.Features.Scanning.ScanningRegistrationExtensions.ScanTypes(IEnumerable`1 types, IComponentRegistry cr, IRegistrat
at Autofac.ContainerBuilder.Build(IComponentRegistry componentRegistry, Boolean excludeDefaultModules)
at Autofac.ContainerBuilder.Build(ContainerBuildOptions options)
at Autofac.ModuleRegistrationExtensions.RegisterAssemblyModules(IModuleRegistrar registrar, Type moduleType, Assembly[] asse
at PX.Data.DependencyInjection.CompositionRoot.CreateContainer()
at ASP.global_asax.Application_Start(Object sender, EventArgs e)
at System.RuntimeMethodHandle.InvokeMethod(Object target, Object[] arguments, Signature sig, Boolean constructor)
at System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(Object obj, Object[] parameters, Object[] arguments)
at System.Reflection.RuntimeMethodInfo.Invoke(Object obj, BindingFlags invokeAttr, Binder binder, Object[] parameters, Cultu
at System.Reflection.MethodBase.Invoke(Object obj, Object[] parameters)
at System.Web.HttpApplication.InvokeMethodWithAssert(MethodInfo method, Int32 paramCount, Object eventSource, EventArgs even
at System.Web.HttpApplication.ProcessSpecialRequest(HttpContext context, MethodInfo method, Int32 paramCount, Object eventSou
```

Figure: Viewing the first-chance exception log

To Debug the Customization Code

The *Code Editor* does not provide facilities you can use to debug a code item of a customization project. We recommend that you use Microsoft Visual Studio to develop and debug any source code in

customization projects. To start debugging an existing *Code* item in Visual Studio, perform the following instructions:

1. Publish the customization project, as described in [Publishing Customization Projects](#).
2. In the file system, open in a text editor the `web.config` file located in the website folder.
3. In `<system.web>` tag of the file, locate the `<compilation>` element.
4. Set to *true* the `debug` attribute of the element, as follows.

```
<compilation debug="true" ...>
```



: See [Debug Mode in ASP.NET Applications](#) for details.

5. In a browser, launch your instance of Acumatica ERP to have a running process in the system.
6. Launch Visual Studio.
7. To open the website of your instance of Acumatica ERP in Visual Studio, on the main menu, click **File > Open > Web Site**, and select the website folder in the **Open Web Site** dialog box, which opens.
8. In the Solution Explorer of Visual Studio, expand the `App_RuntimeCode` website folder, and double-click the file of the *Code* item to open it.
9. In the item source code, which opens, set a breakpoint on the line of the code where you need to stop the process for debugging.
10. On the main menu, click **Debug > Attach to Process**, and in the **Attach to Process** dialog box, which opens, select the `w3wp.exe` process in the list of **Available Processes**.

 : In the dialog box, ensure that the **Show processes from all users** check box is selected. If the check box is cleared, the list will not display the `w3wp.exe` process record.
11. In the browser, open the form of Acumatica ERP whose business logic code you want to debug.
12. In the browser, run an operation that invokes the fragment of the code that contains the breakpoint.

The process will be stopped at the breakpoint, and you can debug the code in Visual Studio.

To Validate a BQL Statement

Suppose that you have a BQL statement in the customization code that does not work properly or you want to make sure that a BQL statement is correctly converted into an SQL query. To validate the BQL statement, you can use the Request Profiler.

To do this, perform the following actions:

1. Open the [Request Profiler](#) form (SM.20.50.70) in the browser.
2. In the **Profiler Options** section, select the **Log SQL Requests** check box, which indicates that the SQL request data should be stored in the database.
3. On the form toolbar, click **Start**, as shown in the following screenshot.

Acumatica ORGANIZATION FINANCE DISTRIBUTION CONFIGURATION **SYSTEM** HELP 3/13/2017 2:28 PM ADMIN

Management Integration Automation Customization

Management Management **Yogifon - Request Profiler** CUSTOMIZATION HELP

Type your query here Search **START** STOP REFRESH RESULTS CLEAR LOG COLLECT MEMORY ACTIVE THREADS

EXPLORE
Companies
MANAGE
Companies
System Locales
Translation Dictionaries
Translation Sets
Rebuild Full-Text Entity Index
PROCESS
Apply Updates
Request Profiler
Collect Translation Sets
CONFIGURE
Update Preferences

PROFILER OPTIONS FILTERS MEMORY USAGE

Log SQL Requests
 Log SQL Requests Stack Trace
 Log Trace Messages
 Log Exceptions

URL:
User Name:
Svr Time Threshold:
SQL Count Threshold:

Managed Memory, M.: 232
Working Set, Mb: 942
GC Collections: 23/13/8

C + X SQL TRACE OPEN URL

Request Start Time (UTC)	User Name	URL	Command Target	Command Name	Client Time	Server Time, ms	SQL Time, ms	Server CPU, ms	SQL Count	Memory Before
--------------------------	-----------	-----	----------------	--------------	-------------	-----------------	--------------	----------------	-----------	---------------

Figure: Launching the SQL Profiler

4. In the browser, open the form that uses the customization code with the BQL statement.
5. On this form, perform the action that executes the BQL statement.
6. Open the *Request Profiler* form (SM.20.50.70).
7. On the form toolbar, click **Refresh Results** to refresh the log table.
8. In the **Command Name** column of the log table, find the URL request that is related to the needed BQL statement.
9. Select the needed URL request, as shown in the screenshot below.
10. On the table toolbar, click **SQL** to open the **SQL Profiler** pop-up panel.

Yogifon - Request Profiler CUSTOMIZATION HELP

START STOP REFRESH RESULTS CLEAR LOG COLLECT MEMORY ACTIVE THREADS

PROFILER OPTIONS FILTERS MEMORY USAGE

Log SQL Requests
 Log SQL Requests Stack Trace
 Log Trace Messages
 Log Exceptions

URL:
User Name:
Svr Time Threshold:
SQL Count Threshold:


Managed Memory, M.: 545
Working Set, Mb: 1006
GC Collections: 25/14/8

C + X **SQL** TRACE OPEN URL

Request Start Time (UTC)	User Name	URL	Command Target	Command Name	Client Time	Server Time, ms	SQL Time, ms	Server CPU, ms	SQL Count	Memory Before
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$phF...	Save	1190	1,076.28	613.92	390.00	45	632,578,144
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	InitRow	254	212.20	34.18	187.20	8	196,227,720
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	DropDown	397	310.90	114.00	187.20	16	225,363,960
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	Refresh	263	201.01	39.00	171.60	14	258,139,968
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	FetchRow	1163	1,056.57	651.17	405.60	57	288,126,112
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	Save	344	239.12	30.69	234.00	8	334,333,784
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	InitRow	534	177.62	1.99	187.20	2	368,854,280
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	FetchRow	387	281.14	40.49	234.00	28	395,784,232
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	Save	328	221.80	12.87	202.80	12	436,446,552
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	RepaintTab	483	136.20	7.22	140.40	2	471,869,080
13 Mar 14:37	admin	~/pages/so/so301000.aspx	ctI00\$ph...	RepaintTab	169	129.53	1.72	124.80	2	493,179,616
13 Mar 14:38	admin	~/pages/so/so301000.aspx	ctI00\$phD...	Save	771	659.12	333.07	312.00	54	515,342,336

K < > I

Figure: Opening the SQL Profiler pop-up panel for the selected URL request

11. On the panel, in the list of SQL requests for the currently selected URL request, find and click the SQL request that corresponds to the BQL statement.
12. On the panel toolbar, click **Switch Between Grid and Form** (), as shown in the following screenshot, to view the SQL statement of the selected request.

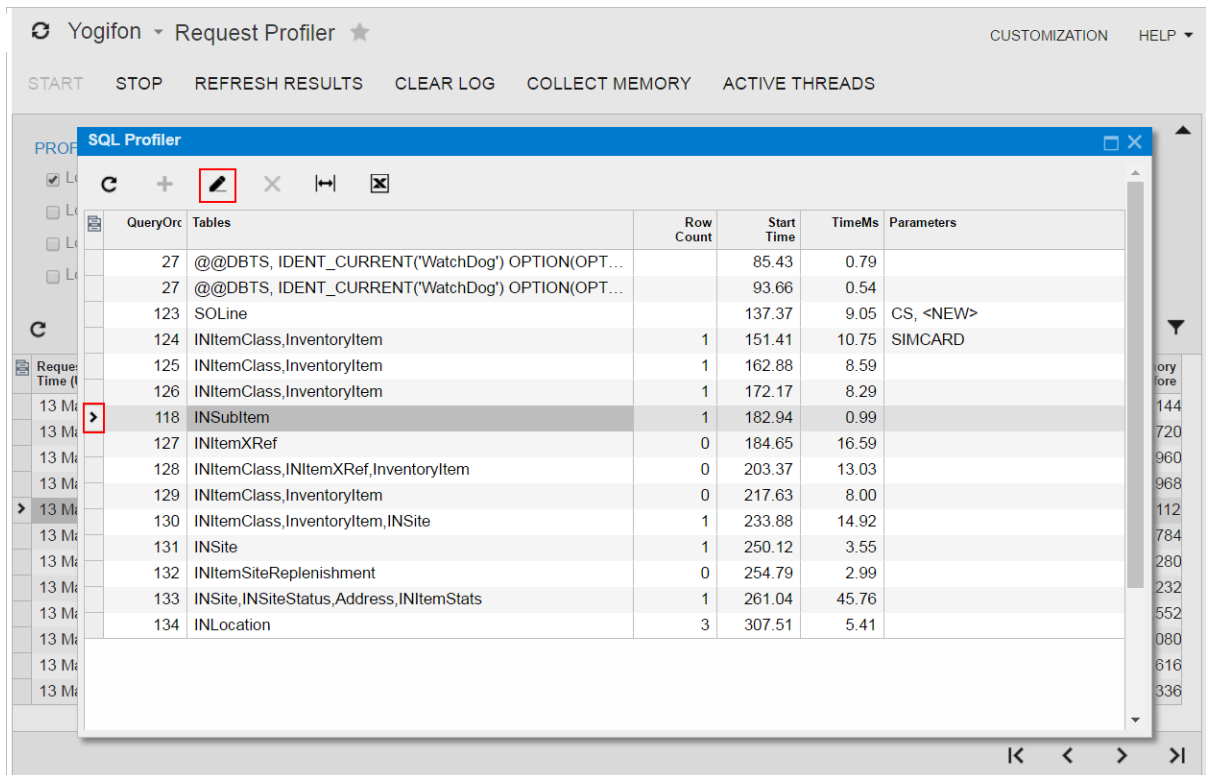



Figure: Switching the panel to display the content of the selected SQL request

 : Because you know the name of the table that you use in the BQL statement, we recommend that you specify the name as a filter in the **Tables** column of the SQL profiler table, as the following screenshot shows.

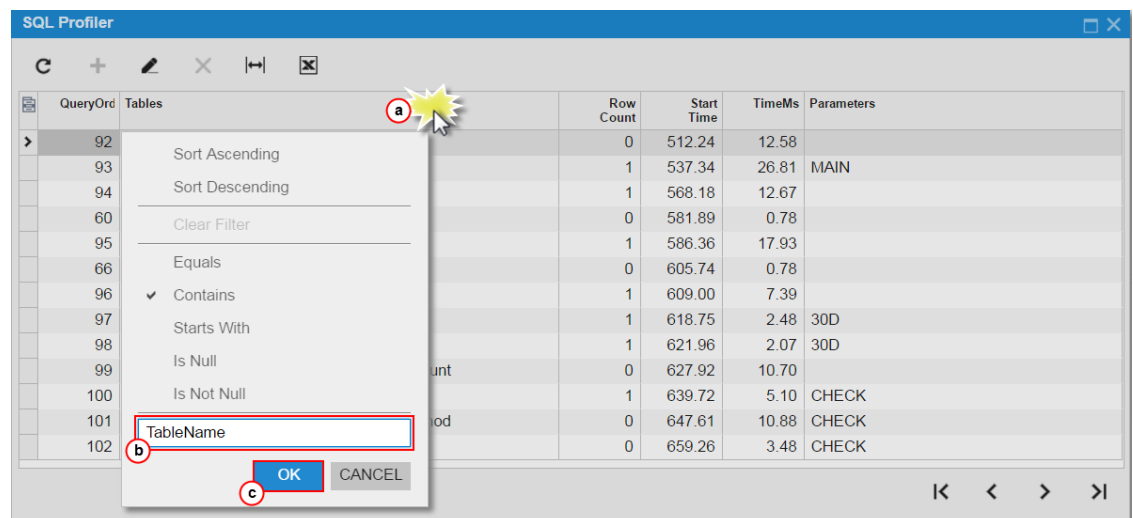


Figure: Setting a filter for the SQL request list

If the BQL statement contains a parameter, you can also use the parameter to filter the list.

If a filter is specified for a column of the table of the **SQL Profiler** pop-up panel, the filter icon is displayed in the column caption, as shown in the following screenshot.

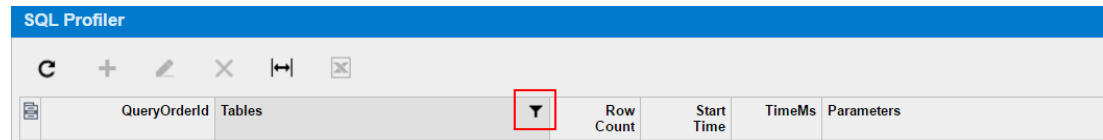


Figure: Viewing the filter icon on the panel

A filter is applied to the list for every URL request until you clear the filter.

13. In the SQL area of the panel, shown in the following screenshot, view the SQL statement to validate the BQL statement used in the customization code.

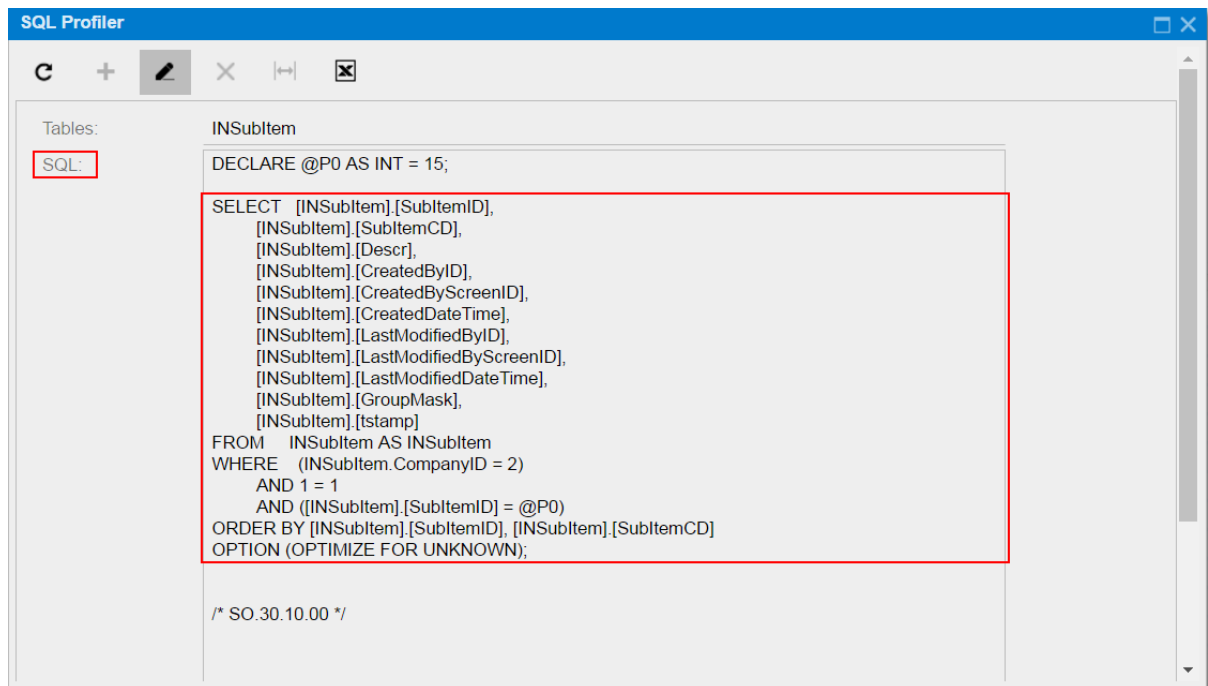


Figure: Viewing the SQL statement

14. On the form toolbar of the *Request Profiler* form (SM.20.50.70), click **Stop** to stop the Request Profiler.



: We recommend that you activate the *Log SQL Requests* mode for only a limited period because it can degrade system performance.

15. On the form toolbar, click **Clear Log** to delete the SQL request data from the database.

To Measure the Execution Time of a BQL Statement

Suppose that in the customization code you have a BQL statement that can affect system performance. To measure the execution time of the BQL statement, you can use the Request Profiler.

To do this, perform the following actions:

1. Open the *Request Profiler* form (SM.20.50.70) in the browser.
2. In the **Profiler Options** area of the form, select the **Log SQL Requests** check box, that indicates that the SQL request data should be stored in the database.
3. On the form toolbar, click **Start**, as shown in the following screenshot.

The screenshot shows the Acumatica Request Profiler interface. The top navigation bar includes 'ORGANIZATION', 'FINANCE', 'DISTRIBUTION', 'CONFIGURATION', 'SYSTEM', and 'HELP'. The 'SYSTEM' menu is highlighted. Below the navigation bar, there are tabs for 'Management', 'Integration', 'Automation', and 'Customization'. The 'Management' tab is active, showing a search bar and a list of navigation items. The 'Request Profiler' item is highlighted in the 'MANAGE' section. The main content area displays the 'Request Profiler' configuration for 'Yogifon'. The toolbar includes buttons for 'START', 'STOP', 'REFRESH RESULTS', 'CLEAR LOG', 'COLLECT MEMORY', and 'ACTIVE THREADS'. The 'START' button is highlighted. The configuration area is divided into three sections: 'PROFILER OPTIONS', 'FILTERS', and 'MEMORY USAGE'. Under 'PROFILER OPTIONS', the 'Log SQL Requests' checkbox is checked and highlighted. Below this are input fields for 'URL', 'User Name', 'Svr Time Threshold', and 'SQL Count Threshold'. The 'MEMORY USAGE' section shows 'Managed Memory, M.: 232', 'Working Set, Mb: 942', and 'GC Collections: 23/13/8'. Below the configuration area is a table with columns: Request Start Time (UTC), User Name, URL, Command Target, Command Name, Client Time, Server Time, ms, SQL Time, ms, Server CPU, ms, SQL Count, and Memory Before. The table is currently empty.

Figure: Launching the SQL Profiler

4. In the browser, open the form that uses the customization code with the BQL statement.
5. On the form, perform the action that executes the BQL statement.
6. Open the [Request Profiler](#) form (SM.20.50.70).
7. On the form toolbar, click **Refresh Results** to refresh the log table.
8. In the **Command Name** column of the log table, find the URL request that is related to the needed BQL statement.
9. Click the needed URL request, as shown in the screenshot below.
10. On table toolbar, click **SQL** to open the **SQL Profiler** pop-up panel.

Yogifon - Request Profiler CUSTOMIZATION HELP ▾

START STOP REFRESH RESULTS CLEAR LOG COLLECT MEMORY ACTIVE THREADS

PROFILER OPTIONS

Log SQL Requests

Log SQL Requests Stack Trace

Log Trace Messages

Log Exceptions

FILTERS

URL:

User Name:

Svr Time Threshold:

SQL Count Threshold:

MEMORY USAGE

Managed Memory, Mb: 359

Working Set, Mb: 908

GC Collections: 44/28/14

SQL TRACE OPEN URL x ▾

Request Start Time (UTC)	User Name	URL	Command Target	Command Name	Client Time	Server Time, ms	SQL Time, ms	Server CPU, ms	SQL Count	Memory Before
14 Mar 13:23	admin	~/?screenid=so303000&do...		HTML		877.67	429.65	546.00	45	168,097,160
14 Mar 13:23		~/?screenid=so303000&do...				880.60	0.00	0.00	0	00
14 Mar 13:23		~/?screenid=so303000&do...				1,082.46	28.78	0.00	6	00
14 Mar 13:23	admin	~/pages/so/so303000.aspx	ctl00\$phDS\$ds	ReloadPage	8027	5,153.32	1,712.46	842.41	136	240,672,000
14 Mar 13:23	admin	~/pages/so/so303000.aspx		HTML		7,255.21	464.38	1,154.41	93	234,168,816
14 Mar 13:24	admin	~/pages/in/in202000.aspx	ctl00\$phDS\$ds	ReloadPage	2215	833.12	224.71	483.60	27	294,172,704
14 Mar 13:24	admin	~/pages/in/in202000.aspx		HTML		1,360.69	22.58	265.20	21	324,765,216
14 Mar 13:25	admin	~/pages/in/in202500.aspx		HTML		1,042.70	233.04	686.40	31	163,568,208
14 Mar 13:25	admin	~/pages/in/in202500.aspx	ctl00\$phDS\$ds	ReloadPage	1909	1,747.40	419.83	234.00	36	222,056,384
14 Mar 13:26	admin	~/pages/in/in202500.aspx	ctl00\$phG\$tab	Save	937	761.67	342.66	421.20	47	337,184,728
14 Mar 13:26	admin	~/pages/so/so301000.aspx	ctl00\$phDS\$ds	ReloadPage	1263	1,029.38	509.48	530.40	68	463,617,960
14 Mar 13:26	admin	~/pages/so/so301000.aspx	ctl00\$phF\$form	Save	1054	934.13	569.49	374.40	45	284,636,480
14 Mar 13:27	admin	~/pages/so/so301000.aspx	ctl00\$phG\$tabSt...	FetchRow	671	559.44	255.73	296.40	55	419,420,312
14 Mar 13:27	admin	~/pages/so/so301000.aspx	ctl00\$phDS\$ds	Save	628	515.92	199.40	312.00	54	364,512,184
14 Mar 13:27	admin	~/pages/so/so301000.aspx	LongRun	Action@PrepareInvo...		2,307.33	1,227.04	982.81	369	440,416,544
14 Mar 13:27	admin	~/pages/so/so303000.aspx	ctl00\$phDS\$ds	Save	778	621.89	255.23	343.20	52	357,328,152
14 Mar 13:28	admin	~/pages/so/so303000.aspx	LongRun	Action@Release		5,798.21	3,537.37	3,135.62	761	433,680,128

⏪ ⏩ ⏴ ⏵

Figure: Opening the SQL Profiler pop-up panel for the selected URL request

- 11.** On the panel, in the table, specify filters for one or both of the **Tables** and **Parameters** columns to shorten the list, as follows:
 - a.** Click the column caption to open a simple filter for the column, as the following screenshot shows.
 - b.** Enter the name of the needed table (for the **Tables** column) or parameter (for the **Parameters** column) that is used in the BQL statement you are referring to.
 - c.** Click **OK** to save the filter.

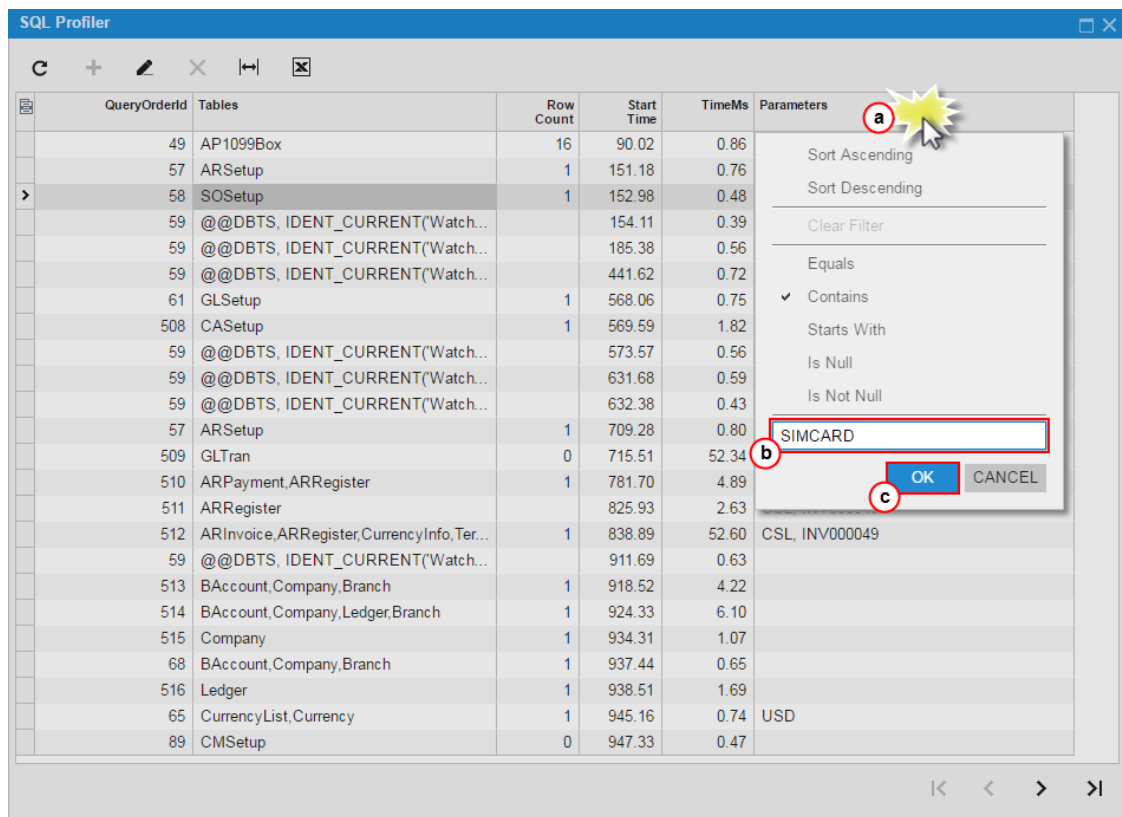


Figure: Setting a filter for the SQL request list



: If a filter is specified for a column of the table of the **SQL Profiler** pop-up panel, the filter icon is displayed in the column caption, as shown in the following screenshot.

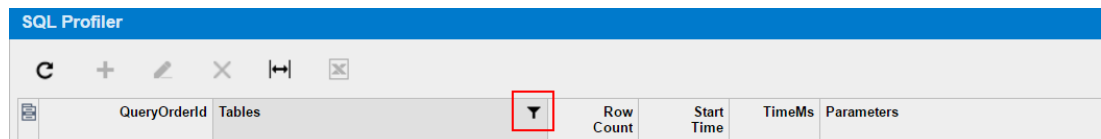


Figure: Viewing the filter icon on the panel

The filter is applied to the list for every URL request until you clear the filter.

- In the filtered list of SQL requests for the currently selected URL request, find the SQL request that corresponds to the BQL statement.

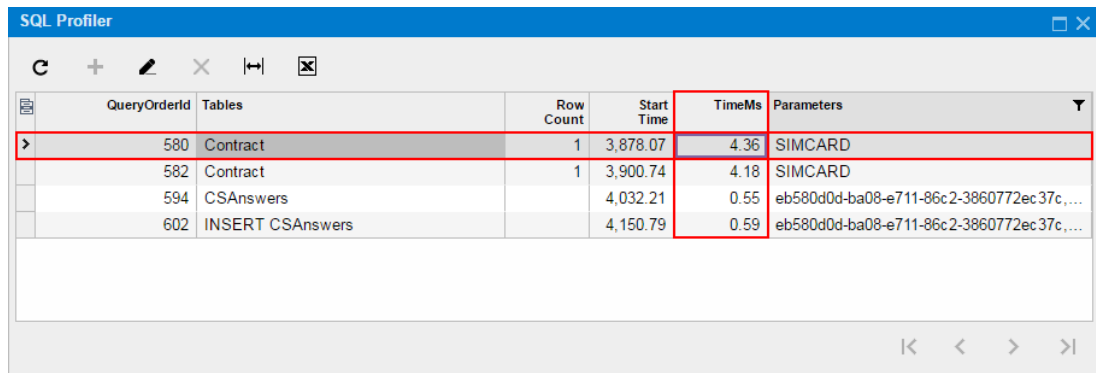


Figure: Viewing the duration of the selected SQL request

The execution time of the SQL request is displayed in the **TimeMs** column.

- On the form toolbar of the [Request Profiler](#) form (SM.20.50.70), click **Stop** to stop the Request Profiler.



: We recommend that you activate the *Log SQL Requests* mode for only a limited period because it can degrade system performance.

- On the form toolbar, click **Clear Log** to delete the SQL request data from the database.

To Discover the Cause of Performance Degradation

Suppose that after publishing a customization project, you notice that the Acumatica ERP performance on a customized form is degraded. To check whether the content of the customization project is the cause of the performance degradation and to find the code fragment that affects system performance, you can use the Request Profiler.

The following sections provide detailed information:

- [To Check Whether a Customization Project Affects System Performance](#)
- [To Find the Code Fragment That Is Affecting System Performance](#)

To Check Whether a Customization Project Affects System Performance



To check whether a customization project degrades the Acumatica ERP performance, perform the following actions:

- Unpublish all customization projects, as described in [To Unpublish a Customization](#).
- Open the [Request Profiler](#) form (SM.20.50.70) in the browser.
- In the **Profiler Options** section, select the **Log SQL Requests** check box, which indicates that the SQL request data should be stored in the database.
- On the form toolbar, click **Start**, as shown in the following screenshot.


The screenshot shows the Acumatica Request Profiler form. The top navigation bar includes 'Acumatica', 'ORGANIZATION', 'FINANCE', 'DISTRIBUTION', 'CONFIGURATION', 'SYSTEM', and 'HELP'. The 'SYSTEM' tab is active. Below the navigation bar, there are tabs for 'Management', 'Integration', 'Automation', and 'Customization'. The main content area is titled 'Yogifon - Request Profiler' and includes a search bar and a toolbar with buttons for 'START', 'STOP', 'REFRESH RESULTS', 'CLEAR LOG', 'COLLECT MEMORY', and 'ACTIVE THREADS'. The 'START' button is highlighted with a red box. Below the toolbar, there are sections for 'PROFILER OPTIONS' and 'FILTERS'. In the 'PROFILER OPTIONS' section, the 'Log SQL Requests' checkbox is checked and highlighted with a red box. Other options include 'Log SQL Requests Stack Trace', 'Log Trace Messages', and 'Log Exceptions'. The 'FILTERS' section includes fields for 'URL', 'User Name', 'Svr Time Threshold', and 'SQL Count Threshold'. To the right, there is a 'MEMORY USAGE' section showing 'Managed Memory, M.: 232', 'Working Set, Mb: 942', and 'GC Collections: 23/13/8'. At the bottom, there is a table with columns: 'Request Start Time (UTC)', 'User Name', 'URL', 'Command Target', 'Command Name', 'Client Time', 'Server Time, ms', 'SQL Time, ms', 'Server CPU, ms', 'SQL Count', and 'Memory Before'.

Figure: Launching the SQL Profiler

- In the browser, open the form that you suspect has experienced performance issues since the customization project was published.
- Reload the page by using the browser toolbar button.
- On the form, sequentially perform multiple actions available on the form toolbar, such as **Refresh** or **Save**. Be sure to execute each action that is customized on the form.

8. On the form toolbar of the [Request Profiler](#) form (SM.20.50.70), click **Refresh Results** to refresh the log table.
9. On the table toolbar, click **Export to Excel** () to save the log data to a Microsoft Excel file.
10. Publish the customization project, as described in [To Publish a Single Project](#).
11. Return to the [Request Profiler](#) form, and on the form toolbar, click **Clear Log** to clear the table.
12. In the browser, open the form that you use for the performance test.
13. Perform the same sequence of actions that you did in Step 6 and Step 7.
14. Return to the [Request Profiler](#) form, and on the form toolbar, click **Refresh Results** to refresh the log table.
15. On the table toolbar, click **Export to Excel** () to save the log data to a Microsoft Excel file.
16. In Microsoft Excel, open the Microsoft Excel files you have saved to compare the duration values for the same actions.

Customization of the business logic for a form can degrade the system performance, however the duration of an action can increase by no more than 10 percent. If the duration of an action on the customized form is more than 10 percent longer than the duration of the same action before customization, then the customization code must be reworked. To do this, follow the instructions described in [To Find the Code Fragment That Is Affecting System Performance](#).
17. On the form toolbar of the [Request Profiler](#) form (SM.20.50.70), click **Stop** to stop the Request Profiler.

 : We recommend that you activate the *Log SQL Requests* mode for only a limited period because it can degrade system performance.
18. On the form toolbar, click **Clear Log** to delete the SQL request data from the database.

To Find the Code Fragment That Is Affecting System Performance

If you have confirmed that the system performance is degraded on a customized form, you can use the Request Profiler to discover the code fragment that is causing the performance degradation. To do this, perform the following actions:

1. Open the [Request Profiler](#) form (SM.20.50.70) in the browser.
2. In the **Profiler Options** section, select the **Log SQL Requests** check box, which indicates that the SQL request data should be stored in the database.
3. Select the **Log SQL Requests Stack Trace** check box, which indicates that the SQL request stack trace data should be stored in the database.
4. On the form toolbar, click **Start**.
5. In the browser, open the customized form that you have confirmed as experiencing performance degradation.
6. On the form, sequentially perform the actions that you identified as executing too slowly.
7. On the form toolbar of the [Request Profiler](#) form (SM.20.50.70), click **Refresh Results** to refresh the log table.
8. In the log table, rearrange the log records in descending order to find the URL requests with the highest values in each of the columns described below.

Column	Description
Server Time, ms	The aggregate duration of the URL request execution on the server (in milliseconds).
SQL Time, ms	The aggregate duration of execution of all the SQL requests in the URL request (in milliseconds).
Server CPU, ms	The aggregate duration of CPU usage during the URL request (in milliseconds).
SQL Count	The count of the SQL requests to the database in the URL request. This count is the difference between the total number of the SQL requests in the URL request and the number of requests to the cache.

9. Select a record that has an unusually large value in any column listed in the previous step, and then click **SQL** on table toolbar, as shown in the following screenshot, to open the **SQL Profiler** pop-up panel.

Yogifon - Request Profiler

START STOP REFRESH RESULTS CLEAR LOG COLLECT MEMORY ACTIVE THREADS

PROFILER OPTIONS: Log SQL Requests, Log SQL Requests Stack Trace, Log Trace Messages, Log Exceptions

FILTERS: URL: _____, User Name: _____, Svr Time Threshold: 500, SQL Count Threshold: _____

MEMORY USAGE: Managed Memory, Mb: 69, Working Set, Mb: 460, GC Collections: 6/3/3

SQL TRACE OPEN URL


Request Start Time (UTC)	User Name	URL	Command Target	Command Name	Client Time	Server Time, ms	SQL Time, ms	Server CPU, ms	SQL Count	Memory Before
14 Mar 13:23	admin	~/pages/so/so303000.aspx		HTML		7,255.21	464.38	1,154.41	93	234,168,816
14 Mar 13:28	admin	~/pages/so/so303000.aspx	LongRun	Action@Release		5,798.21	3,537.37	3,135.62	761	433,680,128
14 Mar 13:23	admin	~/pages/so/so303000.aspx	ctl00\$phDS\$ds	ReloadPage	8027	5,153.32	1,712.46	842.41	136	240,672,000
14 Mar 13:27	admin	~/pages/so/so301000.aspx	LongRun	Action@PrepareIn...		2,307.33	1,227.04	982.81	369	440,416,544
14 Mar 13:25	admin	~/pages/in/in202500.aspx	ctl00\$phDS\$ds	ReloadPage	1909	1,747.40	419.83	234.00	36	222,056,384
14 Mar 13:24	admin	~/pages/in/in202000.aspx		HTML		1,360.69	22.58	265.20	21	324,765,216
14 Mar 13:23	admin	~/?screenid=so303000&docty...				1,082.46	28.78	0.00	6	00
14 Mar 13:25	admin	~/pages/in/in202500.aspx		HTML		1,042.70	233.04	686.40	31	163,568,208
14 Mar 13:26	admin	~/pages/so/so301000.aspx	ctl00\$phDS\$ds	ReloadPage	1263	1,029.38	509.48	530.40	68	463,617,960
14 Mar 13:26	admin	~/pages/so/so301000.aspx	ctl00\$phF\$form	Save	1054	934.13	569.49	374.40	45	284,636,480
14 Mar 13:23	admin	~/?screenid=so303000&docty...				880.60	0.00	0.00	0	00
14 Mar 13:23	admin	~/?screenid=so303000&docty...		HTML		877.67	429.65	546.00	45	168,097,160
14 Mar 13:24	admin	~/pages/in/in202000.aspx	ctl00\$phDS\$ds	ReloadPage	2215	833.12	224.71	483.60	27	294,172,704
14 Mar 13:26	admin	~/pages/in/in202500.aspx	ctl00\$phG\$tab	Save	937	761.67	342.66	421.20	47	337,184,728
14 Mar 13:27	admin	~/pages/so/so303000.aspx	ctl00\$phDS\$ds	Save	778	621.89	255.23	343.20	52	357,328,152
14 Mar 13:27	admin	~/pages/so/so301000.aspx	ctl00\$phG\$tab...	FetchRow	671	559.44	255.73	296.40	55	419,420,312
14 Mar 13:27	admin	~/pages/so/so301000.aspx	ctl00\$phDS\$ds	Save	628	515.92	199.40	312.00	54	364,512,184

Figure: Opening the SQL Profiler pop-up panel for the selected URL request

10. In the panel, use the standard table navigation buttons to explore the list of SQL requests, which are initially ordered by start time (see the following screenshot).

QueryOrderId	Tables	Row Count	Start Time	TimeMs	Parameters
337	AP1099Box	16	33.30	0.91	
338	ARSetup	1	92.23	0.84	
339	SOSetup	1	94.14	0.64	
286	@@DBTS, IDENT_CURRENT('Watch...		95.66	0.66	
286	@@DBTS, IDENT_CURRENT('Watch...		127.46	0.66	
286	@@DBTS, IDENT_CURRENT('Watch...		302.83	1.64	
340	GLSetup	1	415.59	0.90	
341	CASetup	1	417.40	2.53	
286	@@DBTS, IDENT_CURRENT('Watch...		422.17	0.64	
286	@@DBTS, IDENT_CURRENT('Watch...		479.44	1.75	
286	@@DBTS, IDENT_CURRENT('Watch...		481.35	1.01	
338	ARSetup	1	512.30	0.90	
342	GLTran	0	518.43	10.41	AR, CSL, INV000051

Figure: Viewing navigation buttons on the SQL Profiler pop-up panel

11. If needed, in the SQL profiler table, rearrange records by the value in the **TimeMs** column in descending order to find the SQL requests with the longest duration.
12. To view the SQL statement and the stack trace for an SQL request, select the request in the table and then click **Switch Between Grid and Form** (), as shown in the following screenshot.

QueryOrderId	Tables	Row Count	Start Time	TimeMs	Parameters
392	NoteDoc	0	2,442.81	0.67	1c6f185e-5e09-e711-86c2-3860772ec37c
371	Account	1	2,447.63	0.78	
361	FinPeriod		2,450.27	0.83	2017-03-15 00:00:00.000, 2017-03-15 00:00:00.000
403	Batch		2,465.06	3.07	AR, <NEW>
404	Account,PMAccountGroup,Contract,PM...	0	2,517.38	63.23	INV000051, CSL, AR, AR, <NEW>, <NE...
404	Account,PMAccountGroup,Contract,PM...	0	2,588.17	2.13	INV000051, CSL, AR, AR, <NEW>, <NE...
405	NumberingSequence		2,595.19	5.28	BATCH, 2017-03-15 00:00:00.000
406	UPDATE NumberingSequence		2,601.01	3.39	000085, BATCH, 000084
407	INSERT CurrencyInfo		2,606.67	2.17	USD, USD, 2017-03-15 00:00:00.000, M
408	@@IDENTITY OPTION(OPTIMIZE FO...		2,609.18	0.87	
408	@@IDENTITY OPTION(OPTIMIZE FO...		2,610.46	0.55	
409	INSERT Batch		2,613.01	5.44	AR, 000085, 2017-03-15 00:00:00.000, 20...
325	Note	0	2,619.30	0.66	166f185e-5e09-e711-86c2-3860772ec37c

Figure: Switching the panel to display the SQL statement and the stack trace of the selected SQL request

13. In the stack trace, find the method that is causing the performance degradation.
14. Launch Microsoft Visual Studio.
15. In Visual Studio, open the file with the method; the file is currently saved in the `App_RuntimeCode` website folder.
16. Analyze the method body to find the code snippet that is causing the issue.
17. On the form toolbar of the *Request Profiler* form (SM.20.50.70), click **Stop** to stop the Request Profiler.



: We recommend that you activate the *Log SQL Requests* mode for only a limited period because it can degrade system performance.

18. On the form toolbar, click **Clear Log** to delete the SQL request data from the database.

To Force the Platform to Execute Database Scripts

You can discover that the database scripts that are included in a customization project have not been executed by the Acumatica Customization Platform when you publish the project. This issue arises when you have first published the project and then changed the scripts.

When you publish the project, the platform executes all the scripts included in the project. For optimization purposes, to avoid the execution of database scripts during every publication of the project, the platform saves information about each script that has been executed at least once and has not yet been changed in the database, and omits the repeated execution of such scripts.

To force the platform to clean up all such information about previously executed scripts of a customization project and execute the scripts again while publishing the project, follow the instructions described in [To Publish the Current Project with a Cleanup Operation](#).

To Resolve Issues While Upgrading a Customized Website

If you have a customization project that works properly for the current version of Acumatica ERP and you need to upgrade an application instance to a newer version, the customization project might not work properly or might even prevent the website from starting after the upgrade. This happens because the code of Acumatica ERP is continuously developed to implement new features or enhance existing functionality, so the code of an updated instance of Acumatica ERP can become inconsistent with the code in a customization project. For example, if the signature of a method that is overridden in the customization code is changed in the original code, a run-time error may occur in the graph extension. As another example, modified or deleted database columns and tables might cause the functionality of a data access class extension to fail.

To prevent these issues between the customization and the current version of Acumatica ERP, before you upgrade a customized instance of Acumatica ERP to a new version, the Acumatica Customization Platform validates the compatibility of the code included in all currently published customization projects with the application code of the update version. This validation process executes the following checks in the code of the published customization to detect the breaking changes in the code of Acumatica ERP:

- In graph extensions:
 - Checking the signature for each method that is overridden by using the `PXOverride` attribute.
 - Checking the existence of each base graph.
- In data access class extensions, if a field attribute is overridden:
 - Checking the field existence.
 - Checking that the field type is not changed. (If the customization code uses the field value as decimal and the field type is changed, for example, from `PXDBDecimal` to `PXDBString` that is currently binding this field to a database column of the string type, an exception will occur on a request to the database.)
- In binary DLL files:
 - Checking all the referenced methods, properties, fields, return types, and signatures.



: In some specific cases, a binary file can be updated automatically with a new signature. For example, in a referenced graph, the process updates the BQL statement that is changed for a data view. If a changed signature is detected and cannot be fixed automatically, an error is reported, and the validation fails.

This validation protects a customized website from an update that contains breaking changes and might make the website unworkable.

You can upgrade an instance of Acumatica ERP in one of the following ways:

- On the Application Maintenance page of the Acumatica ERP Configuration Wizard. (See [Updating Acumatica ERP by Using the Configuration Wizard](#) for details.)
- On the [Apply Updates](#) form (SM203510; **System > Management > Process**). (See [Updating Acumatica ERP by Using the Web Interface](#) for details.)

When you start upgrading a customized instance of Acumatica ERP to a newer version, the platform launches the validation process, to check the published customization compatibility with the new version code to detect breaking changes. If the validation fails, the platform cancels the upgrade process and shows an appropriate error message. (See [Messages for Validation Errors](#) for details.)

To upgrade a customized instance of Acumatica ERP, follow the instructions presented in the following topics:

- [To Validate the Compatibility of the Published Customization with a New Version Before an Upgrade](#)
- [To Resolve an Issue Discovered During the Validation](#)
- [To Use the Technical Release Notes to Find the Breaking Changes](#)
- [To Use an Ignore List for the Validation Errors](#)

To Validate the Compatibility of the Published Customization with a New Version Before an Upgrade

You can validate the compatibility of the currently published customization with an updated version before you start the upgrade process.

To do this, perform the following actions:

1. Navigate to the [Apply Updates](#) form (SM203510; **System > Management > Process**).
2. On the **Updates** tab of the form, in the table of available updates, click the product version to which you want to upgrade your Acumatica ERP instance, as shown in the screenshot below.
3. On the table toolbar, click **Download Package**.
When the download is complete, the **Ready to Install** check box is automatically selected.
4. In the table toolbar, click **Validate Customization** to start the process of validating the compatibility of the currently published customization code with the code of the selected product version.

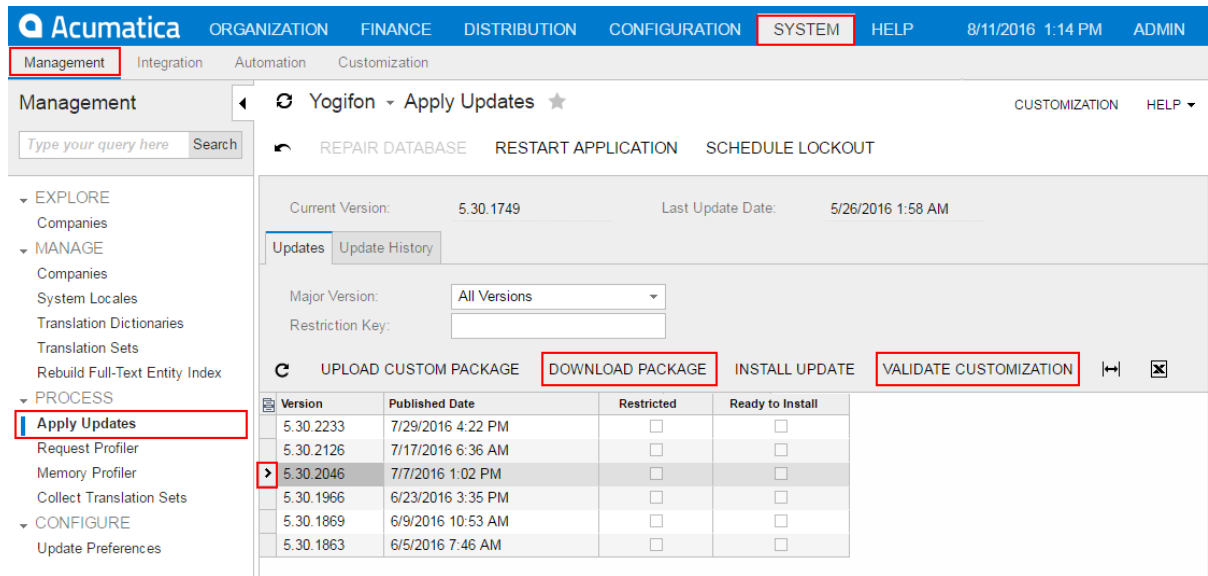


Figure: Validating the product version

If the validation succeeds, you know that you can upgrade your Acumatica ERP instance to the selected version without an issue occurring.

If the validation fails, the **Validation Failed** window opens to display the list of the executed checks and discovered errors, as the following screenshot shows.

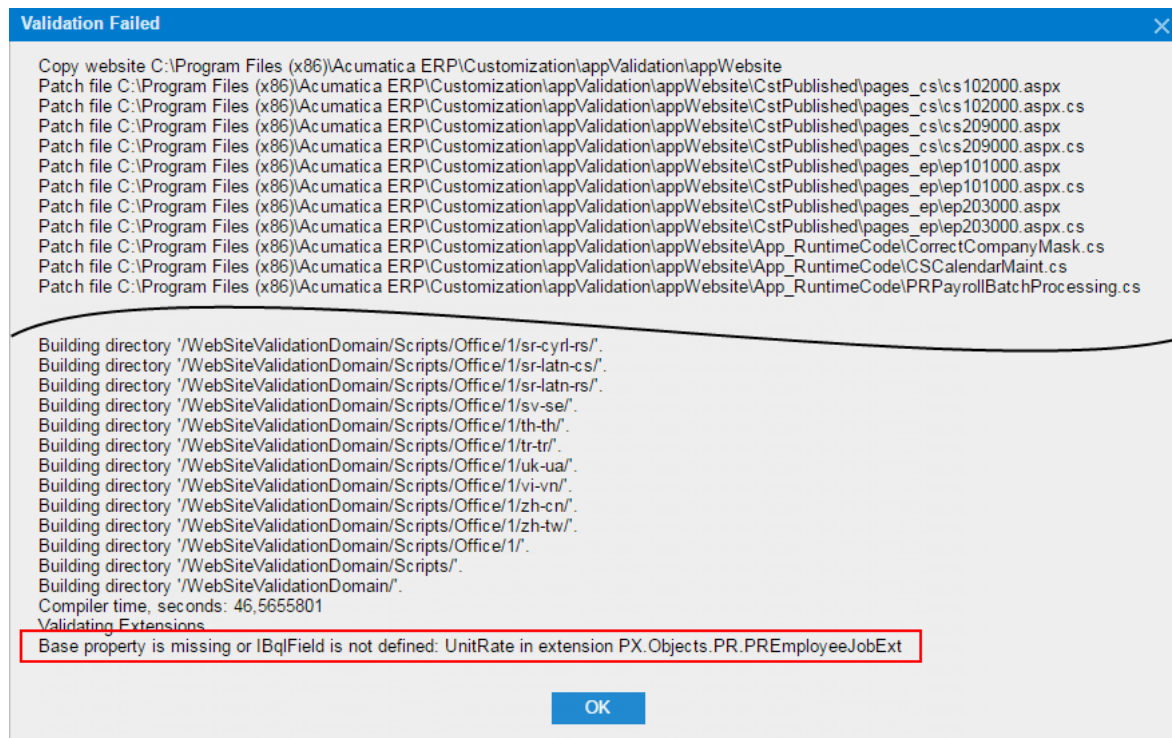



Figure: Viewing an error message in the Validation Failed window

To resolve any discovered issues, we recommend that you use the approach described in [To Resolve an Issue Discovered During the Validation](#).

To Resolve an Issue Discovered During the Validation

If you need to upgrade an Acumatica ERP instance in the production environment, if the validation of the compatibility of the code included in all currently published customization projects with the code of the new product version fails, do the following:

- If the customization was developed by a third-party, please contact their support department directly.
- If you are a developer of this customization:
 1. Learn what the error message means. (See [Messages for Validation Errors](#) for details.)
 2. Find the customization code fragment related to the error, and try to resolve the issue you have discovered yourself by using the error description.
 3. Check the technical release notes for the selected product version to ensure that the reason of the error is a change in the product code, and find the recommendation to solve the issue. (See [To Use the Technical Release Notes to Find the Breaking Changes](#) for details.)
 4. In your development environment:
 - a. Install the needed version of Acumatica ERP and deploy the new application instance. (See [Installing Acumatica ERP Locally](#) for details.)
 - b. Import all customization projects. (See [To Import a Project](#) for details.)
 - c. Validate and fix the customization code in each project.

 : If you are unable to resolve an issue yourself, please contact Acumatica support. You can create a support case for the issue on the [Support](#) page of the [Acumatica Partner Portal](#).

If you are sure that a discovered error is not really an error, you can force the validation process to ignore the error. See [To Use an Ignore List for the Validation Errors](#) for details.
 5. In the production environment:
 - a. Unpublish all customization projects. (See [To Unpublish a Customization](#) for details.)
 - b. Upgrade the Acumatica ERP instance. (See [Updating Acumatica ERP by Using the Web Interface](#) for details.)
 - c. Update the customization projects. (See [To Replace the Content of a Project from a Package](#) for details.)
 - d. Publish the fixed customization projects. (See [Publishing Customization Projects](#) for details.)



: The validation that detects breaking changes is by default turned on in an instance of Acumatica ERP. If you need to temporarily omit this validation, you can turn off the `CheckCustomizationCompatibility` key by including the following string in the `<appSettings>` section of the `web.config` file located in the website folder:

```
<add key="CheckCustomizationCompatibility" value="False" />
```

We do not recommend that you turn off the validation permanently.

Messages for Validation Errors

This topic, which is intended to be used by developers, describes the messages that may occur during the validation of the customization code compatibility with the code of Acumatica ERP.

Failed to resolve method reference

This error occurs if the validation process finds in a custom DLL a reference to a method that is absent with the same signature in the code of Acumatica ERP.

The following example of the error message informs you that the `AM.Objects.dll` file contains a reference to the `System.Void PX.Data.PXLineNbrAttribute::.ctor(System.Type)` method, which has not been declared or has another signature in the `PX.Data` assembly.

```
AM.Objects.dll
Failed to resolve method reference:
System.Void PX.Data.PXLineNbrAttribute::.ctor(System.Type)
declared in
PX.Data, Version=1.0.0.0, Culture=neutral, PublicKeyToken=3b136cac2f602b8e
```

Because the customization code is consistent with the previous version of Acumatica ERP, this error occurs because the method overridden in the customization code has been removed or its signature has been changed in the update. To ensure that this change is implemented in the code of the new version of Acumatica ERP, see the technical release notes for this version, as described in [To Use the Technical Release Notes to Find the Breaking Changes](#).

To fix the error, in the code of the specified extension library, you should refer to an appropriate method declared in the assembly.

Failed to resolve type reference

This error occurs if the validation process finds in a custom DLL a reference to a type whose declaration is absent in the code of Acumatica ERP.

The following example of the error message informs you that the `FETempFix.dll` file contains the reference to the `PX.Data.PXGraphWithActionsBase`2` type that is not declared in the `PX.Data` assembly.

```
FETempFix.dll
Failed to resolve type reference:
PX.Data.PXGraphWithActionsBase`2
declared in
PX.Data, Version=1.0.0.0, Culture=neutral, PublicKeyToken=3b136cac2f602b8e
```

To fix the error, in the code of the specified extension library, you have to refer to an appropriate type declared in the assembly.

Could not resolve

This error occurs if the validation process finds a reference to a custom DLL that cannot be found.

The following example of the error message informs you that the `FullRegen.dll` file contains the `ADODB` reference, which cannot be resolved.

```
FullRegen.dll
Could not resolve:
ADODB, Version=7.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
```

To fix the error, in the code of the specified extension library, you have to refer to a DLL file located in the `Bin` folder of the website.

Declaring Type missing

This error occurs if the validation process finds, in the customization code located in the `App_RuntimeCode` folder of the website, a reference to a type whose declaration is absent in the code of Acumatica ERP.

The following example of the error message informs you that the `PX.Objects.EP.CRBaseActivityMaint`1<PX.Objects.CR.CRTaskMaint>` class contains a reference to the `BaseBAccount` type, which is undeclared.

```
Declaring Type missing:
BaseBAccount
from
PX.Objects.EP.CRBaseActivityMaint`1<PX.Objects.CR.CRTaskMaint>
```

To fix the error, in the corresponding *Code* or *DAC* item of the customization project, you have to refer to the appropriate type.

Referenced Field missing

This error occurs if the validation process finds, in the customization code located in the `App_RuntimeCode` folder of the website, a reference to a field that is undeclared in the data access class (DAC).

The following example of the error message informs you that the `PX.Objects.CS.CSAnswers` data access class does not contain the `_EntityID` field.

```
Referenced Field missing:
_EntityID
from
PX.Objects.CS.CSAnswers
```

To fix the error, in the corresponding *Code* or *DAC* item of the customization project, you have to refer to the appropriate field.

Base property is missing or IBqlField is not defined

This error can occur if the validation process finds a data access class extension that contains a reference of a field with both of the following implementation features:

- The field name does not contain the *Usr* prefix.
- The DAC extension does not contain the `IBqlField` declaration.

For such cases, the validation determines that the DAC extension overrides a field existing in the code of Acumatica ERP. If this field cannot be found in the code of Acumatica ERP, the validation reports about the error.

The following example of the error message informs you that the `Age` field is referenced in the `CQHRIS.ContactExtension` extension of the `CQHRIS.Contact` data access class; however the code of Acumatica ERP does not contain the field declaration.

```
Base property is missing or IBqlField is not defined:
Age
in extension
CQHRIS.ContactExtension
```

If the field is custom, to fix the error, you have to add the `IBqlField` declaration in the DAC extension. If the code refers to a field that was removed from the code of Acumatica ERP, you should alter the DAC extension to make it consistent with the base class.

To Use the Technical Release Notes to Find the Breaking Changes

If you need to know all the changes that have been performed in some version of Acumatica ERP and may affect reports, customizations, and integration built in a prior version, see the list of changes in the Technical Release Notes document for the new version.

The Technical Release Notes documents contain the following lists of breaking changes:

- Removed screens
- Removed tables
- Removed fields
- Modified field types
- Removed data access classes
- Removed graphs
- Removed constants
- Other removed classes
- Modified interfaces and abstract classes
- Other breaking changes
- Other breaking fields

To download a Technical Release Notes document for the needed version of Acumatica ERP, perform the following actions:

1. In the browser, open [Acumatica Partner Portal](#).
2. In the **Product Links** list of the webpage, click **Download Acumatica**.
3. In the **Links** list at right side of the webpage, which opens, click **Release Notes**.
4. On the Release Notes webpage, which opens, select the tab for the needed version of Acumatica ERP, and then click the link for the Technical Release Notes document for this version, as the following screenshot shows.

Figure: Downloading the Technical Release Notes on the Partner Portal

After you have downloaded the document, you can use it to search for the object that is specified in the error message that occurred during the validation process.



: The Technical Release Notes document contains information about the changes in the selected version of Acumatica ERP. If you upgrade an instance of Acumatica ERP, for example, from Version 5.2 to Version 6.0, you should explore the technical release notes for Versions 5.3 and 6.0.

If the object is found, and the document contains information about how to fix the issue, use the information. Otherwise, on the [Support](#) page of the portal, click **New Support Case** to create a support case for the issue.

To Use an Ignore List for the Validation Errors

If validation of the compatibility of the published customization with the new version fails, the platform cancels the upgrade process and shows an appropriate error message. However for some cases when you are sure that a discovered error is not really an error, you can force the validation process to ignore the error, so you can continue the process of upgrading of the customized instance of Acumatica ERP. Also, you may need to ignore, for example, an error in a third-party DLL file.

To force the validation process to ignore an error, perform the following actions:

1. In the `App_Data` folder of the website, create a `.txt` file whose name has the *CstValidationIgnore* prefix, such as `CstValidationIgnore.txt` or `CstValidationIgnore_ProjectName.txt`.



: You can create multiple `CstValidationIgnore*.txt` files with ignore lists for an instance of Acumatica ERP.

2. In the tool used for the upgrade, which may be the Acumatica ERP Configuration Wizard or the [Apply Updates](#) form (SM203510; **System** > **Management** > **Process**), select the error message and copy it to the clipboard.

3. In the file, paste the error message from the clipboard.



: You can add multiple error messages to each of multiple files with ignore lists.

4. Save the file.

5. Determine which the customization project has the code that provokes the error message.

6. Add the file with the ignore list to the project (see [To Add a Custom File to a Project](#) for details) so it will ignore the error when you publish the project in the production environment.



: While it validates a customization project before the project publication, the Acumatica Customization Platform ignores all the errors that are specified in the `CstValidationIgnore*.txt` files with the *CstValidationIgnore* prefix in the file name if these files are located in the `App_Data` folder of the website.

As a result, if you publish the customization project or upgrade an Acumatica ERP instance where the project is published, the validation process omits the error that is specified in the file.

Examples

This part of the guide is intended to describe examples of how you can customize the user interface and business logic of Acumatica ERP forms.

In This Part

- [Examples of User Interface Customization](#)
- [Examples of Functional Customization](#)

Examples of User Interface Customization

UI customization includes changing the look and behavior of forms, tweaking form design, and manually editing the .aspx code. UI customization is always related to the .aspx code of the corresponding forms and represents *aspx changesets* that are added to the existing .aspx code.

For customization to the layout of existing forms, you can use the [Layout Editor](#) that you open from the Customization Project Editor.

For the layout design of new forms, you should use the Layout Editor that you open from Microsoft Visual Studio.

The next topics describe the rules for user interface customization and give examples of typical UI customization tasks:

- [Dragging, Moving, and Deleting UI Controls and Grid Columns](#)
- [Adding Input Controls](#)
- [Adding Advanced Controls](#)
- [Adding Columns to a Grid](#)
- [Modifying Columns in a Selector](#)
- [Adding PXLayoutRule Components](#)

Dragging, Moving, and Deleting UI Controls and Grid Columns

This topic describes in detail how to change the placement of UI controls by dragging them and how to use the facilities of the Layout Editor to delete or move UI controls and grid columns.

Dragging and Dropping UI Controls

Before you start to manually move UI controls by dragging them in the Control Tree of the Layout Editor, note the following simple rules:

- You can move a UI control anywhere within its container control.
- To move a UI control within a container, you have to drag the element to the required place.
- Any UI control moved within a form is automatically aligned according to the nearest **PXLayoutRule** component placed above it. (See [Layout Rule \(PXLayoutRule\)](#) for details.)

On the *Stock Items* form, suppose that you want to move up the **Is a Kit** check box, to make it the first control in the first column of the **General Settings** tab area, as the screenshot below illustrates.

The screenshot shows the Acumatica Inventory form for 'New York - Stock Items'. The 'General Settings' tab is active, and the 'Is a Kit' checkbox is highlighted with a red box and a red arrow. The 'Stock Items' menu item in the left sidebar is also highlighted with a red box.

Figure: Viewing the form before dragging the UI control

To move the check box:

1. Click the **Customization** menu on the form and choose the **Inspect Element** command.
2. Click the **Is a Kit** check box.
3. Click the **Customize** button on the **Element Properties** dialog box.

The customization item is added to the current project if you have selected one by using the **Customization > Select Project** command, or you will be asked to select the customization project to add the item to it.

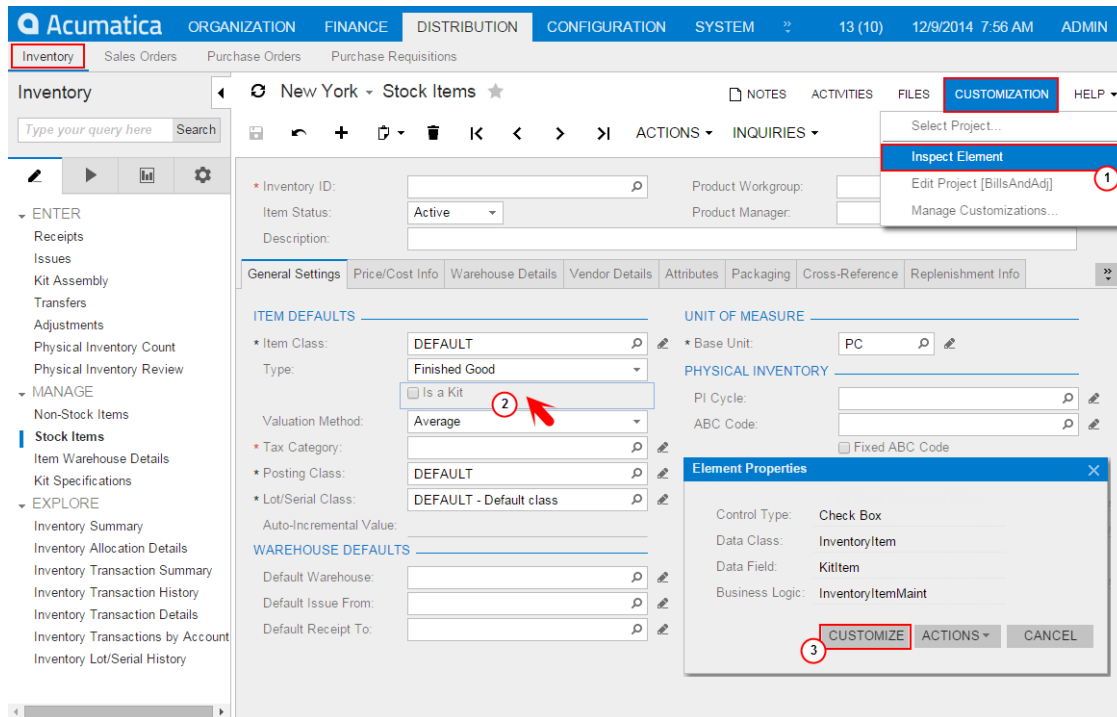


Figure: Selecting the UI control for customization

4. In the Control Tree of the Layout Editor, drag the **Is a Kit** check box to the position above the **Item Class** control.
5. Save the changes to the customization project.

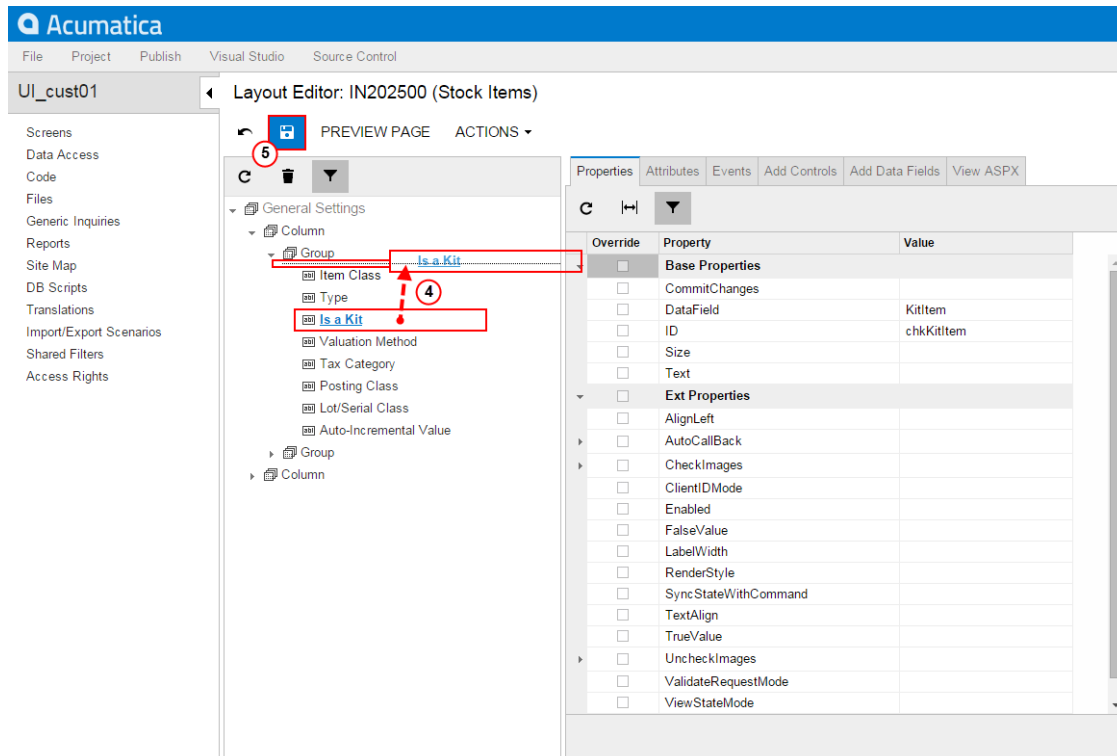


Figure: Moving the UI control to the required position

Once the project is published, the check box appears at the new position (see the following screenshot).

Figure: Noticing the new placement of the check box

Moving and Deleting UI Controls and Grid Columns

You use the Layout Editor to implement advanced customization of a form—for instance, if you need to shift a column within a grid, to delete a UI control or table column from a form area, or to change the position of a UI control on a form while having the capability to change its original container.



: After changing containers, you should perform functional changes concerning the code of appropriate main data access classes (DACs) or data views of business logic controllers (BLCs, also referred to as *graphs*). For details, see [Customizing DAC Attributes](#).

Suppose that you have to customize the **General Settings** and **Warehouse Details** tabs of the *Stock Items* form as follows:

1. Delete the **Auto-Incremental Value** input control. (See the first screenshot below.)
2. Move down the **Tax Category** control so that it is placed under the **Lot/Serial Class** control, as the first screenshot also illustrates.



: The **Lot/Serial Class** field is visible if the **Lot and Serial Tracking** feature is enabled on the *Enable/Disable Features* (CS.10.00.00) form.

3. Move to the left the **Status** column of the details table so that it is placed at the right of the **Default** column, as shown in the second screenshot below

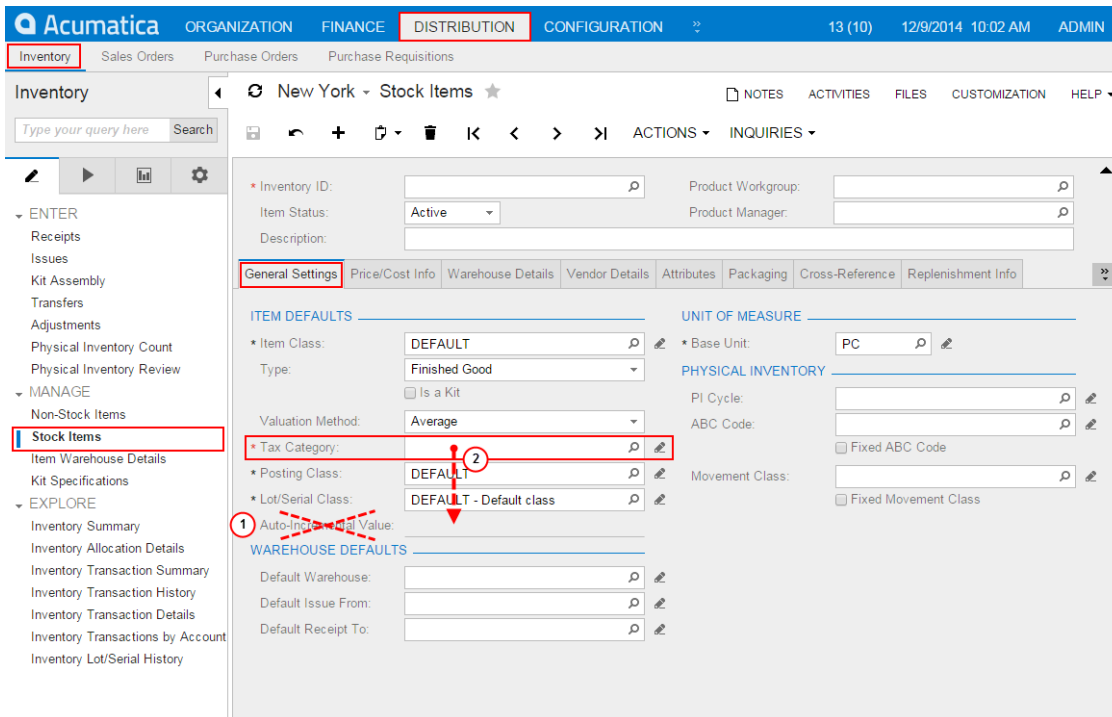


Figure: Viewing the General Settings tab before changes

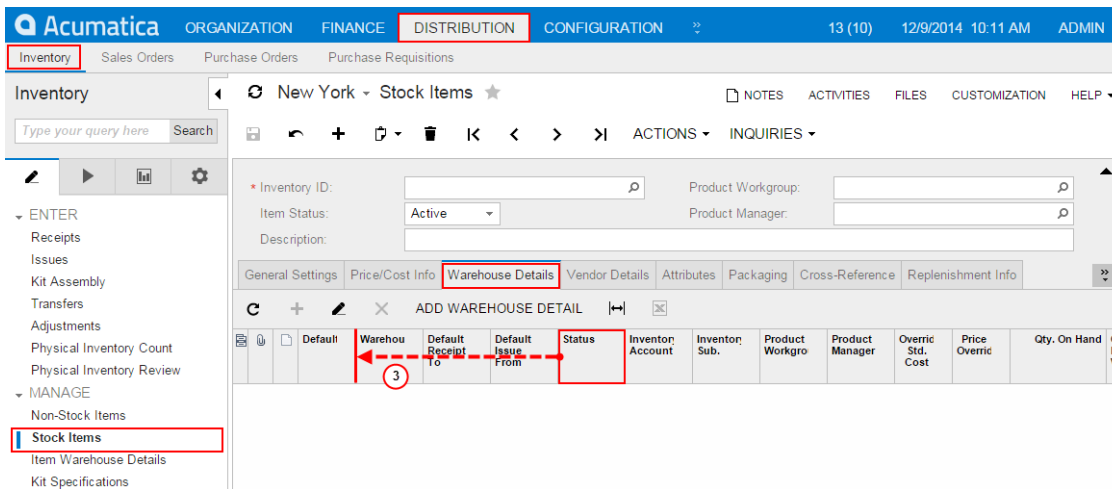


Figure: Viewing the details table of the Warehouse Details tab before changes

To open the Layout Editor, open the *StockItems* form, click the **Customization** menu, choose the **Inspect Element** command, select the **Auto-Incremental Value** control on the form, and click the **Customize** button on the **Element Properties** dialog box.

The Control Tree of the editor displays content of the **General Settings** tab item, to which the **Auto-Incremental Value** control belongs. This control is currently selected and highlighted in bold.



: To view all controls and containers of the customized form on the tree, click the filter button of the tree toolbar.

To delete the selected control, click the **Delete** button of the tree toolbar.

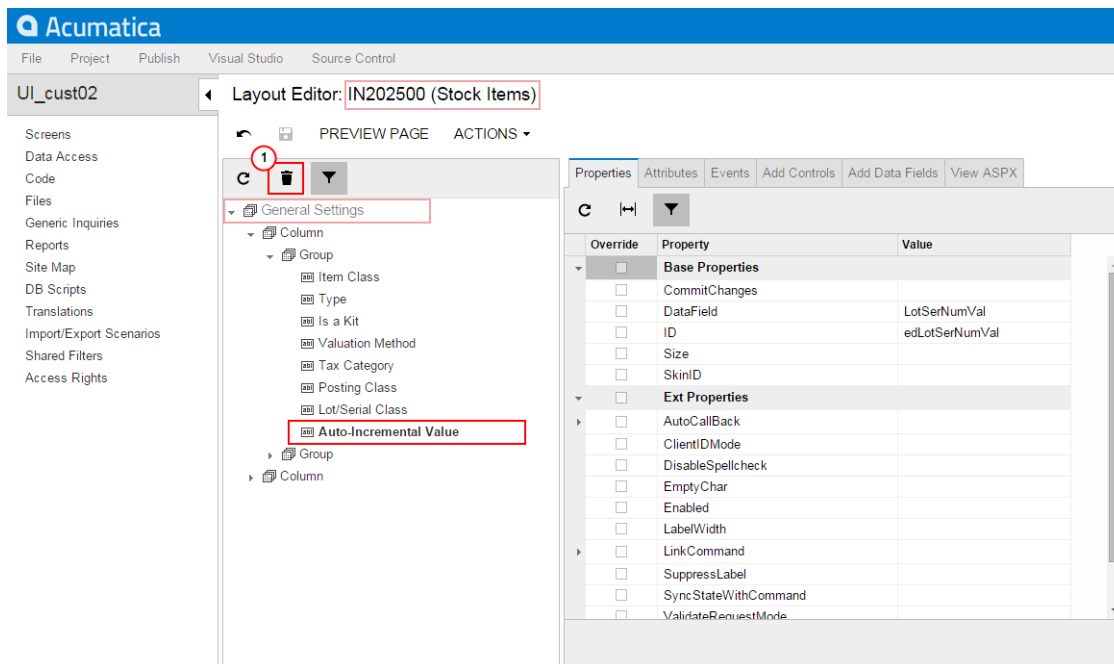


Figure: Removing the selected UI control

Now you should move down the **Tax Category** UI control, to place it under the **Lot/Serial Class** control. Select the **Tax Category** UI control on the tree, drag it and drop to the required position, as the screenshot below illustrates.

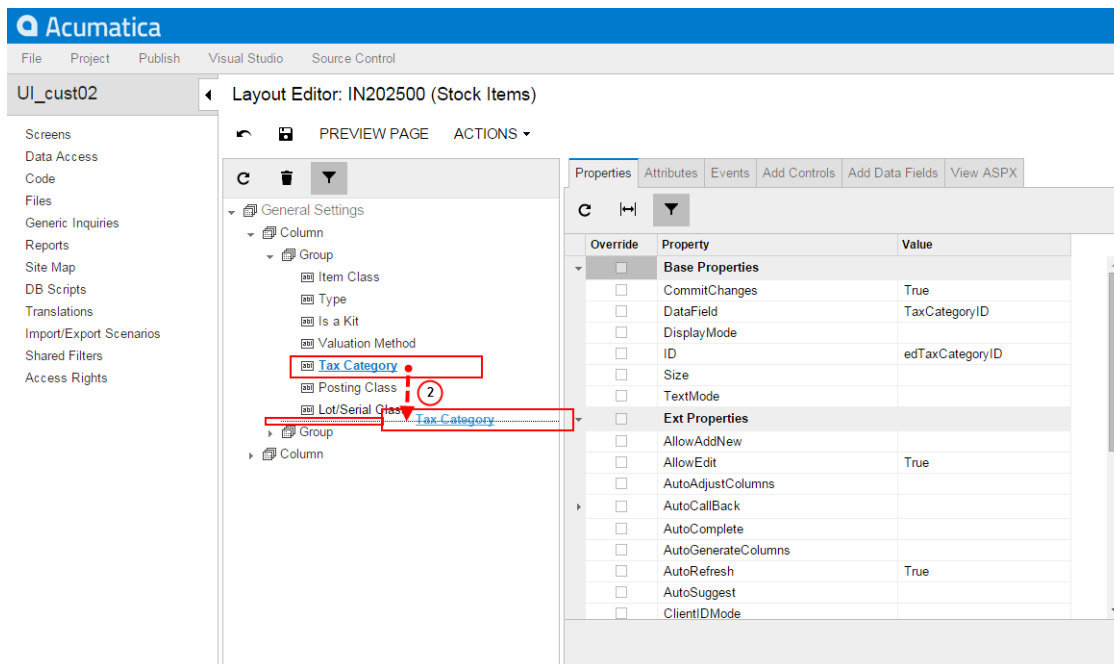


Figure: Moving the control on the Control Tree

Thus, to change the placement of a UI control, you select it on the Control Tree of the Layout Editor, drag it and drop to place this control appropriately in relation to other UI controls.

Click **Save** on the toolbar of the Layout Editor to save changes to the current customization project.

To begin moving the **Status** column, you can:

- Open the *StockItems* form and use the Element Inspector for the **Status** column on the form
- Open the Layout Editor for the form and find the column in the tree of controls in the editor

To use the first approach: on the *StockItems* form, choose the **Warehouse Details** tab, click the **Customization** menu, choose the **Inspect Element** command, select the **Status** column on the grid, and click **Customize** on the **Element Properties** dialog box.

Drag and drop the **Status** control to the required position in the tree, as the screenshot below illustrates.

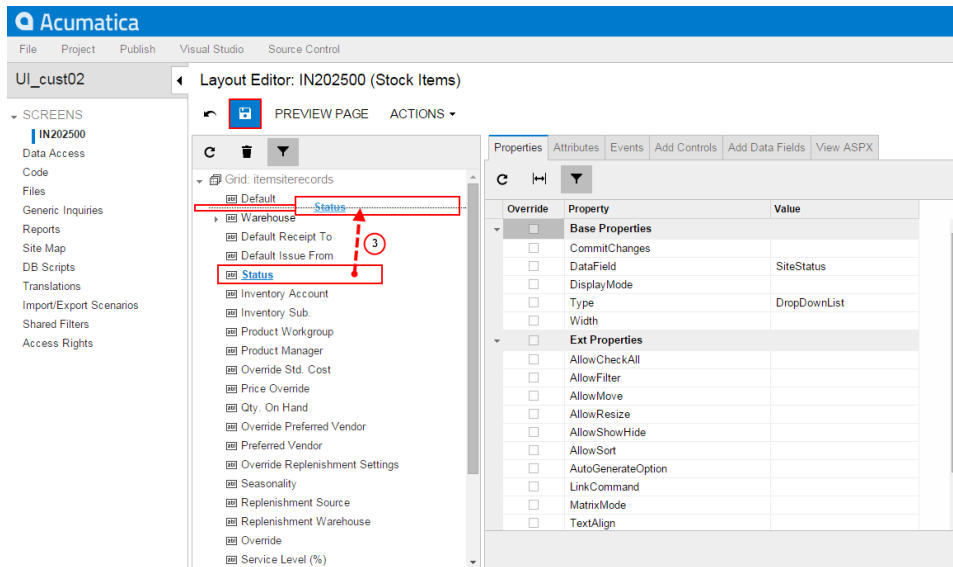


Figure: Moving the grid column to the left

Thus, to change the placement of a grid column, you find the data column in the Control Tree of the Layout Editor and drag it to the desired position.

Click **Save** on the toolbar of the Layout Editor to save changes to the current customization project.

After you publish the project, you will see the modified **General Settings** tab on the *Stock Items* form, as the following screenshot illustrates.

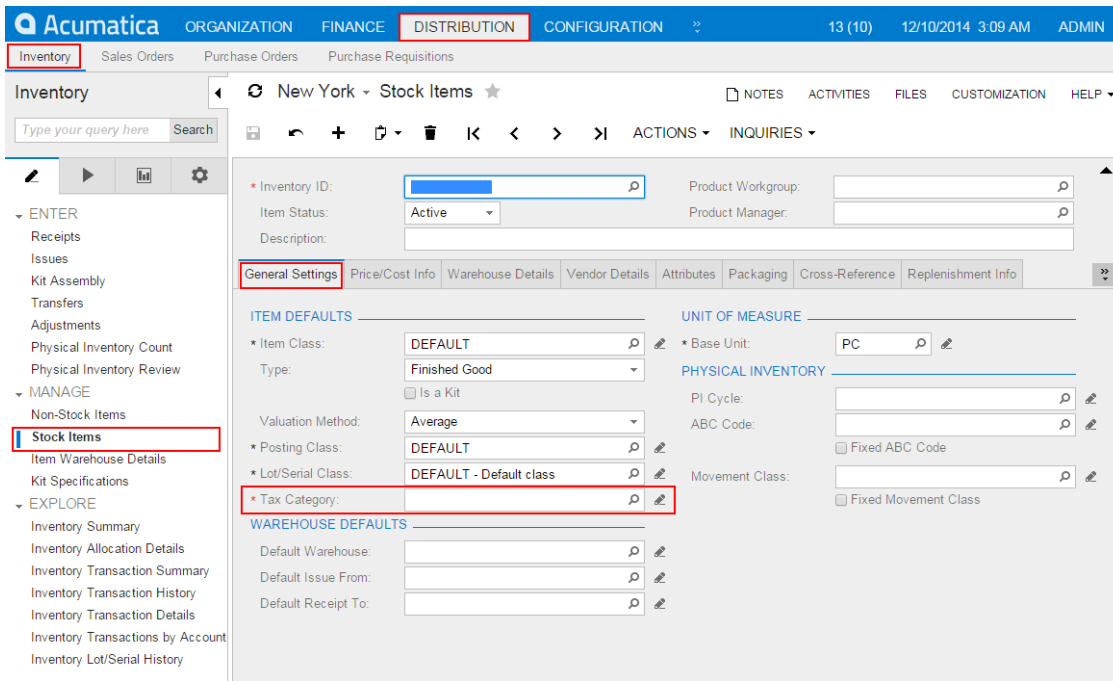


Figure: Viewing the modified General Settings tab of the form

If you open the **Warehouse Details** tab of this form, you will see that the column of the details table has been moved to the left, as shown in the screenshot below.

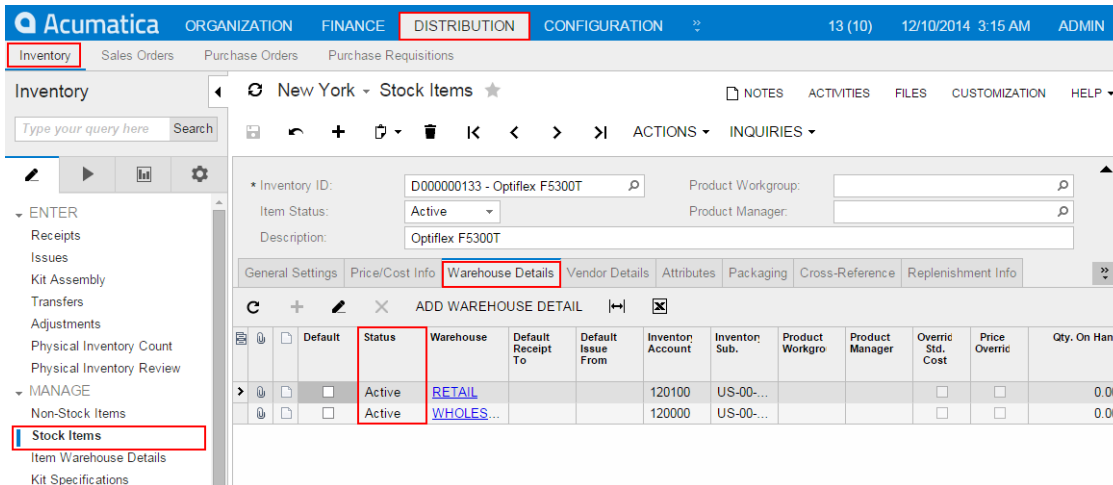


Figure: Viewing the moved column on the Warehouse Details tab

If you need to analyze the added content (changeset) of the customization project, open the project in the Project Editor, choose the **Edit Project Items** command on the **File** menu item and select the **Object Name** field of the customized object (see the screenshot below).

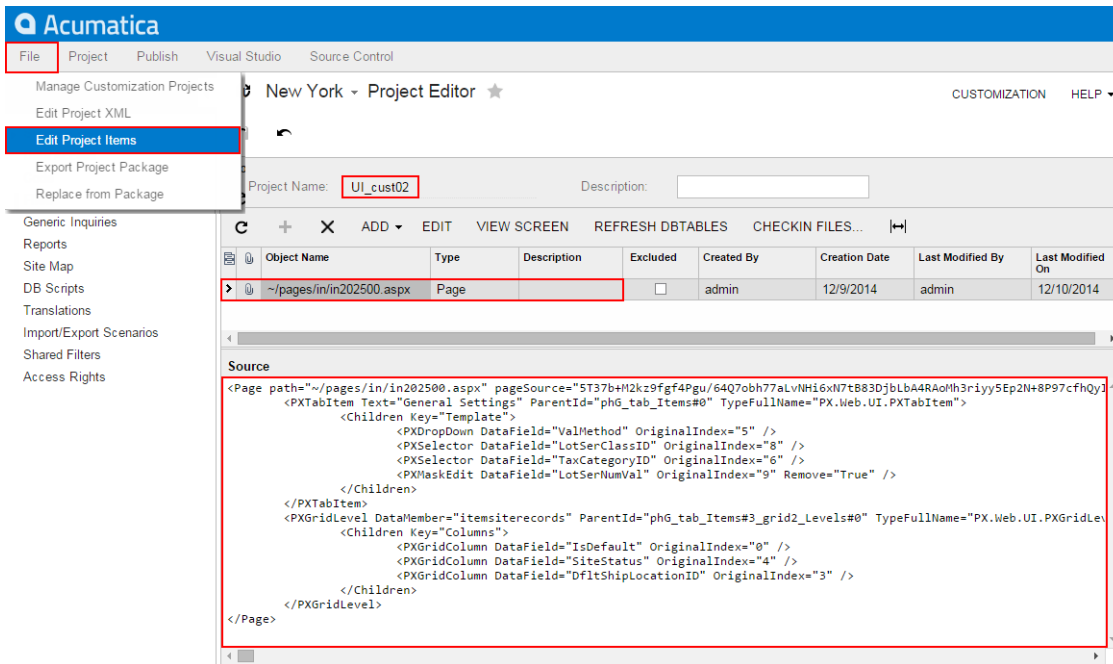


Figure: Analyzing the added content of the customization project

Adding Input Controls

If you need to add a control onto a form, you use the Layout Editor.

The screenshot below illustrates the original *Journal Transactions* form. Suppose that you want to add a text field (for instance, the **CreatedByID** selector of the *GL Batch* DAC) to the second column of the form below the **Auto Reversing** check box.

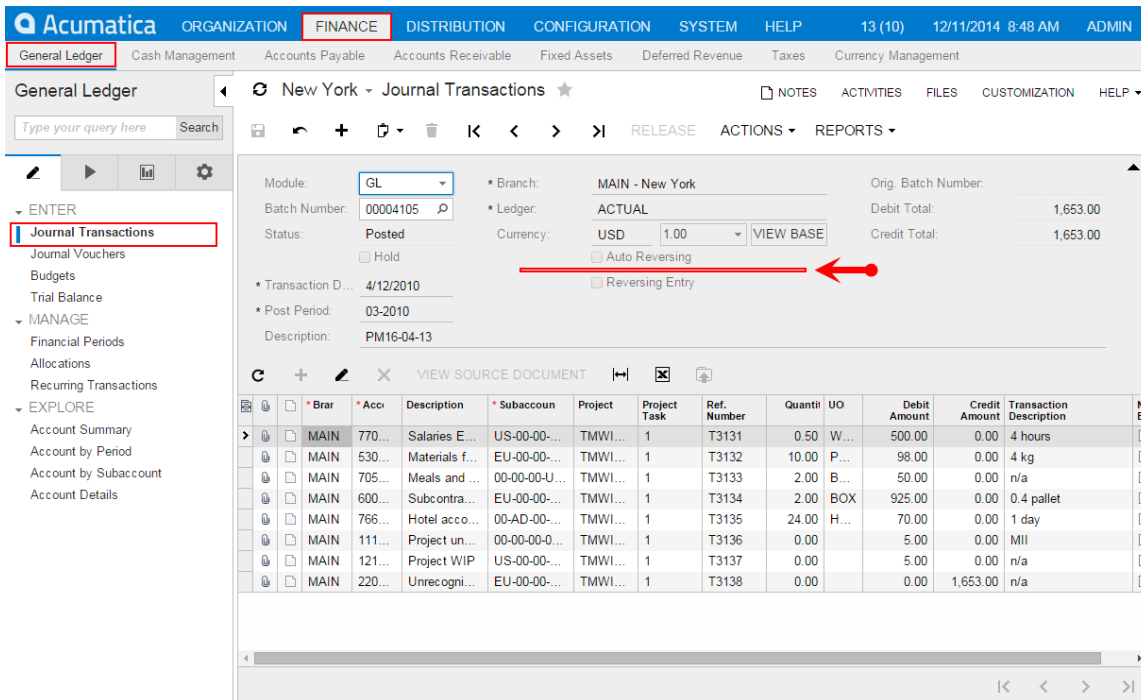


Figure: Viewing the original form

To open the Layout Editor, open the *Journal Transactions* form, click the **Customization** menu, choose the **Inspect Element** command, select the **Auto Reversing** control, and click the **Customize** button on the **Element Properties** dialog box.

The editor is opened with the **Auto Reversing** control selected and highlighted in bold in the tree of controls.

To create a control below the selected item in the tree:

1. Click on the **Add Data Field** tab.
2. Select the **CreatedByID** field in the list on the tab.
3. Click the **Create Controls** button on the tab toolbar.

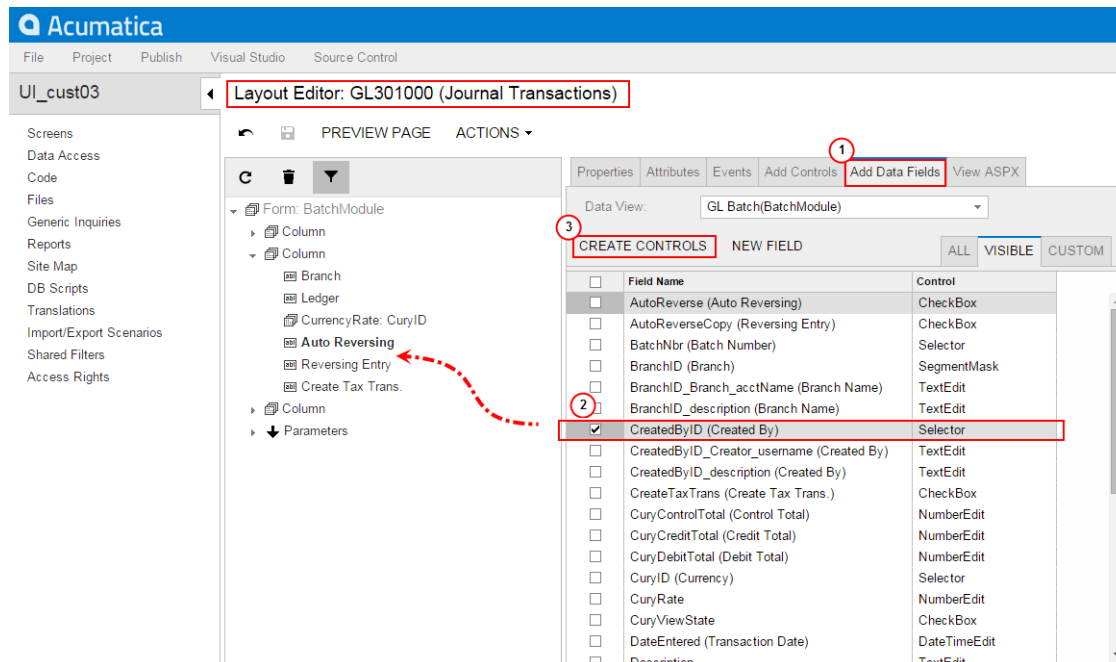


Figure: Creating a control below the selected one in the tree of controls

If you need to change the placement of a UI control, select the control on the Control Tree of the Layout Editor, and drag and drop it to the needed place.

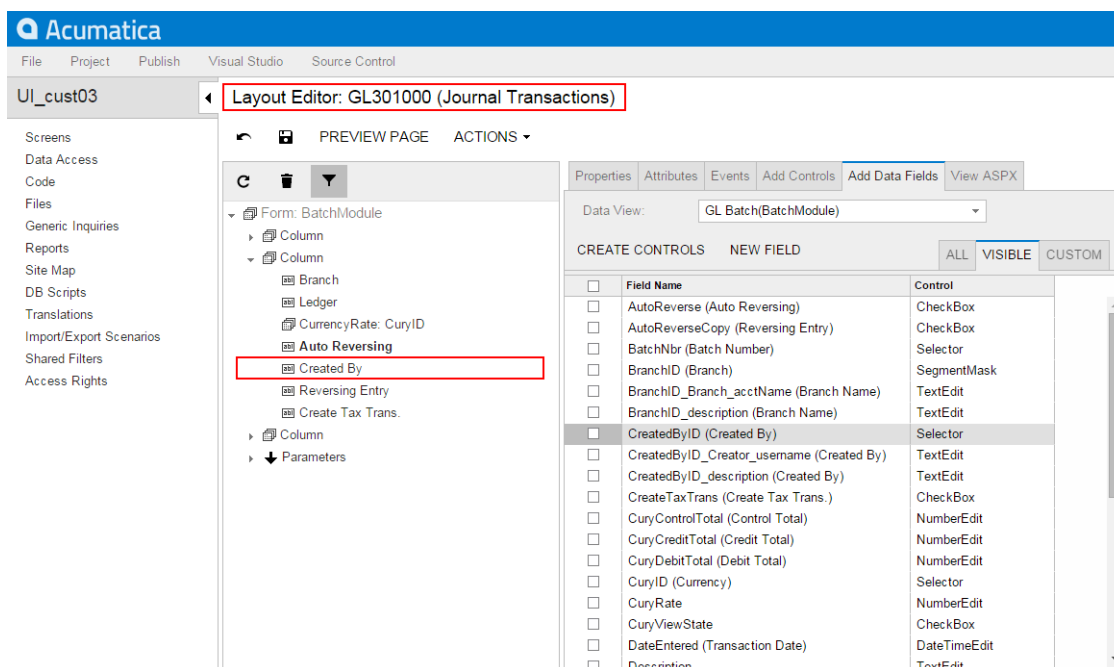


Figure: Viewing the Layout Editor with the created control

Click **Save** on the toolbar of the Layout Editor to save changes to the current customization project. After publishing the customization project, you will see the added field on the *Journal Transactions* form, as the screenshot below illustrates.

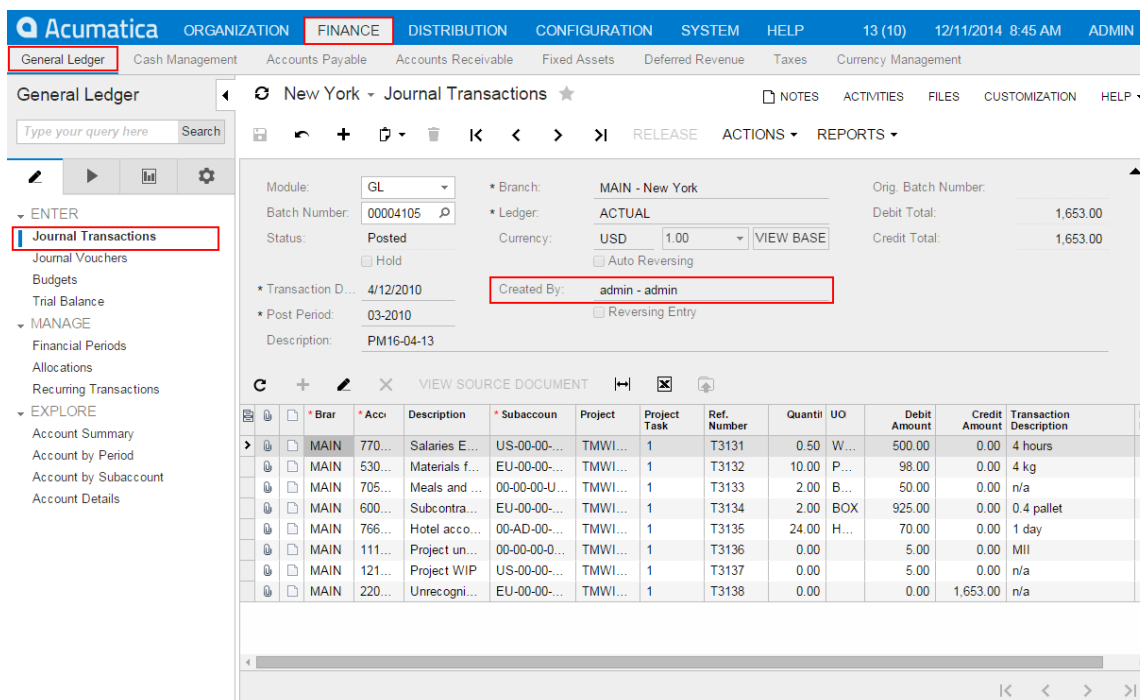


Figure: Viewing the customized form

To view the fragment of the .aspx code that represents the customization result, open the form in the Layout Editor and select the **View ASPX** tab. The customized fragment is highlighted in yellow, as the screenshot below illustrates.

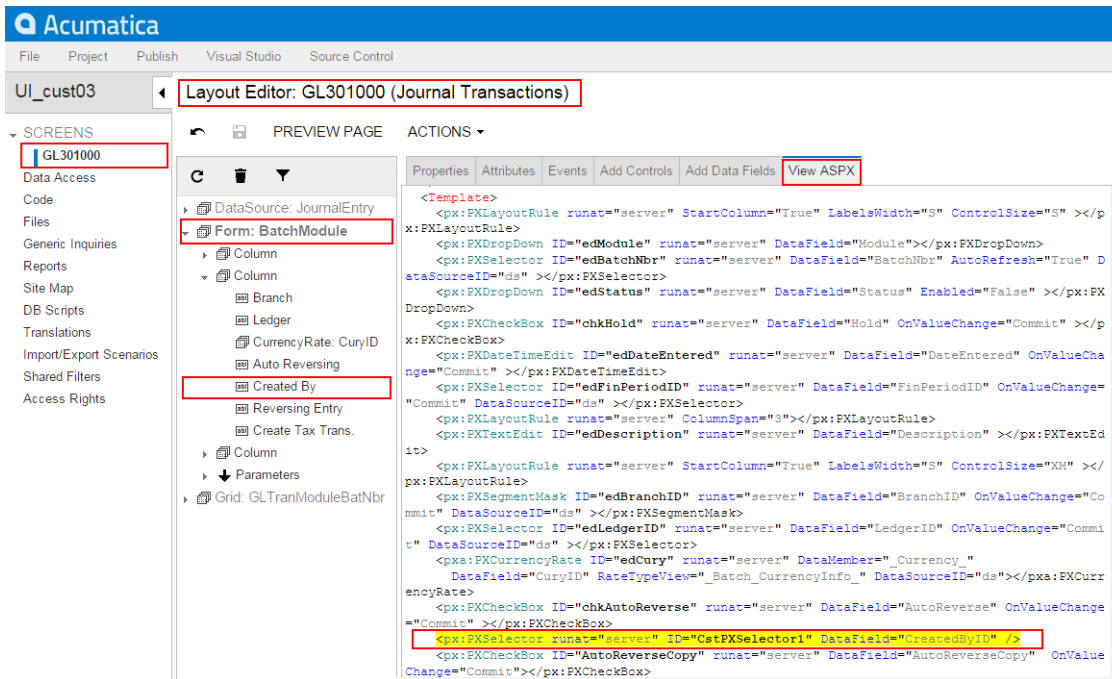


Figure: Analyzing the changeset of the .aspx code

To view the changes to the layout of the form introduced by the new control, open the project in the Project Editor, choose the **Edit Project Items** command on the **File** menu item and select the **ObjectName** field of the customized object (see the screenshot below). You can see the new Page item that contains the .aspx code changeset.

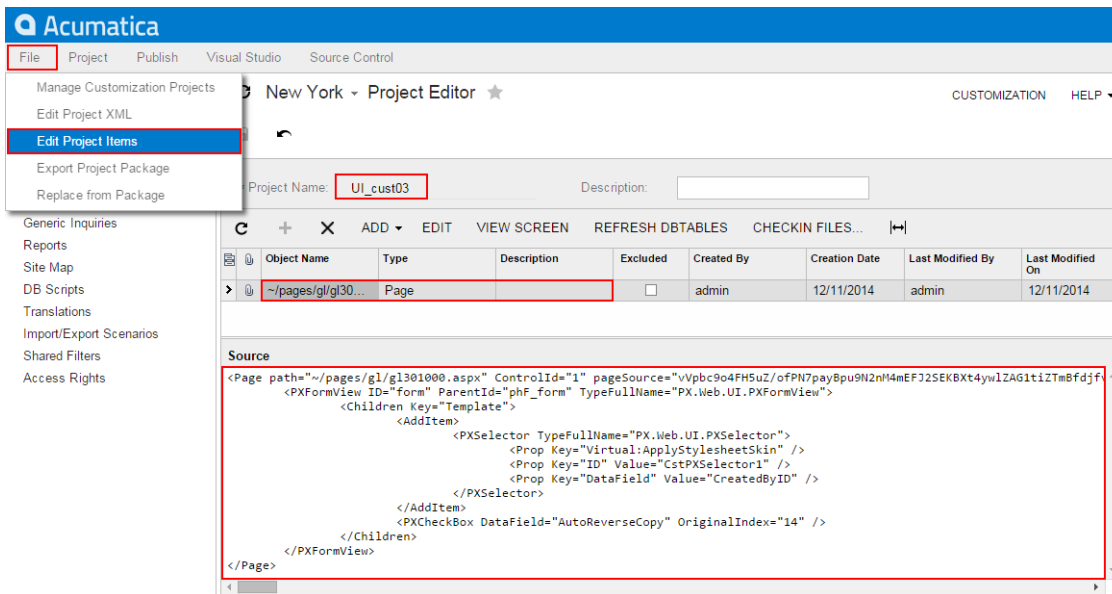


Figure: Analyzing the added content of the customization project



: As a rule, if you add a new data field, you need to perform some additional actions related to the functional customization. For an example of the creation of a new data field, see [Adding Data Fields](#).

Adding Advanced Controls

Suppose that you need to customize the *Stock Items* form and you have divided the customization task into the following steps:

- [Adding a new tab item](#) onto the form.
- [Adding a panel onto the tab item](#).
- [Adding an UI control onto the new panel](#).



: This step is needed to make the added tab and its content visible at run time; otherwise, empty container controls aren't displayed.

To view the original form, navigate to **Distribution > Inventory > Manage > Stock Items** (see the screenshot below).

Figure: Viewing the Stock Items form before changes

Adding a New Tab Item

Suppose that you need to add the *Relative Positioning Layout* tab item to the *Stock Items* form and set its position as the leftmost one.

To do this, perform the following actions:

1. On the *Stock Items* form, click the **Customization** menu, choose **Inspect Element**, select the tab control, and click **Customize** on the **Element Properties** dialog box.
2. In the Layout Editor that appears, click the *Tab: ItemSettings* node in the tree of controls to view all items of the tab.
3. Open the **Add Controls** tab of the Layout Editor.
4. Drag and drop the **Tab Item** (*PXTabItem*) container above the *General Settings* node in the tree, as shown in the screenshot below.

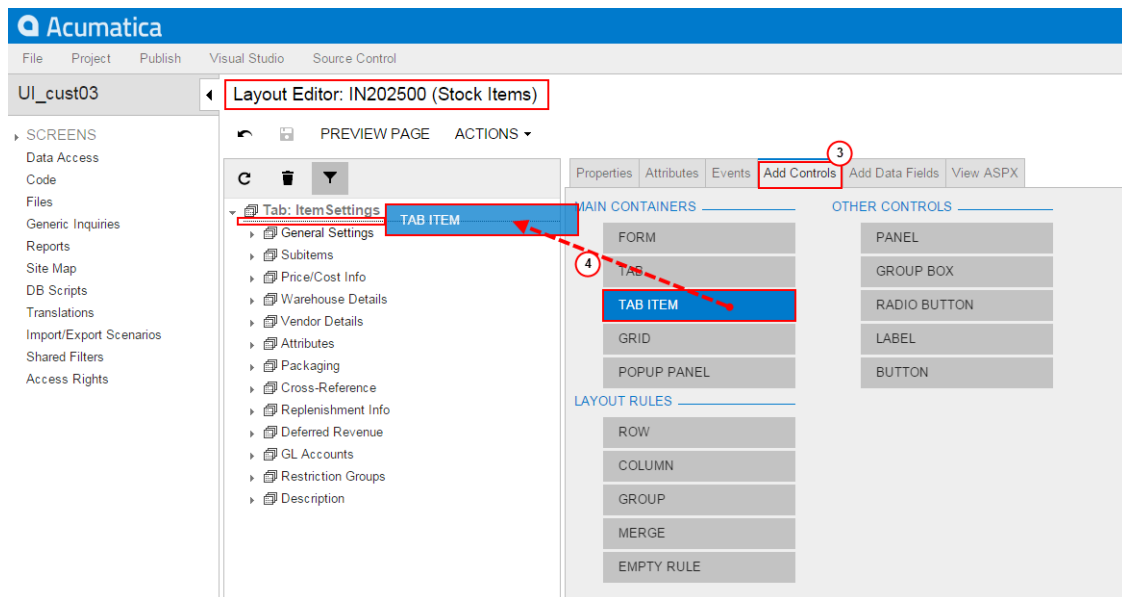


Figure: Adding a tab item to the left position on the tab area of the Stock Items form

5. Select the *TabItem* node that has been added.
6. Select the **Properties** tab of the Layout Editor.
7. Select the **Text** property and enter *Relative Positioning Layout* to specify the name of the new tab, as shown in the screenshot below.
8. Click **Save** on the toolbar of the Layout Editor to save changes to the current customization project.

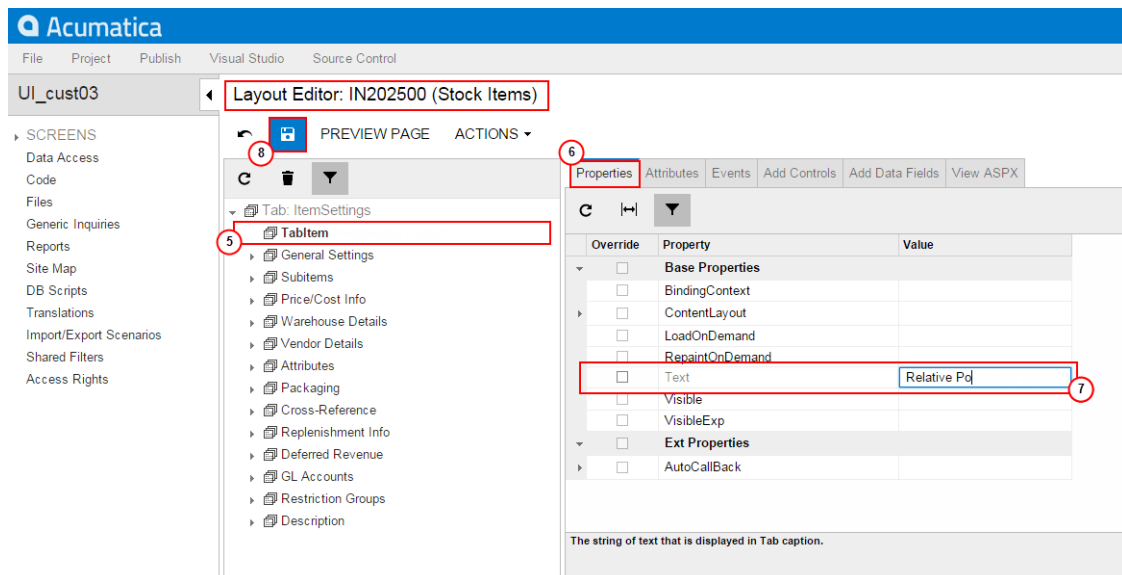


Figure: Defining the name of the added tab item

Thus, to add a new tab item to a tab, you have to expand the tab node in the Control Tree of the Layout Editor, select the **Add Controls** tab in the editor, drag and drop the **Tab Item** container to place this control appropriately in relation to other items in the tab.

A container is visible on the form if the container contains a visible field control. So at the moment you cannot view the *Relative Positioning Layout* tab item on the *Stock Items* form after the customization project is published or on the form preview that can be opened by clicking **Preview Page** in the editor.

Adding a Panel onto the PXTabItem Container Control

Now you have to add a panel (*PXPanel* control) onto the created *PXTabItem* control (that is, onto the new tab item), which is empty at the moment.

Complete the following steps:

1. Open the **Add Controls** tab of the Layout Editor.
2. On the opened tab item, find and drag the **Panel** (*PXPanel*) control.
3. Drop it into the *Relative Positioning Layout* tab item in the tree of controls, as shown in the screenshot below.

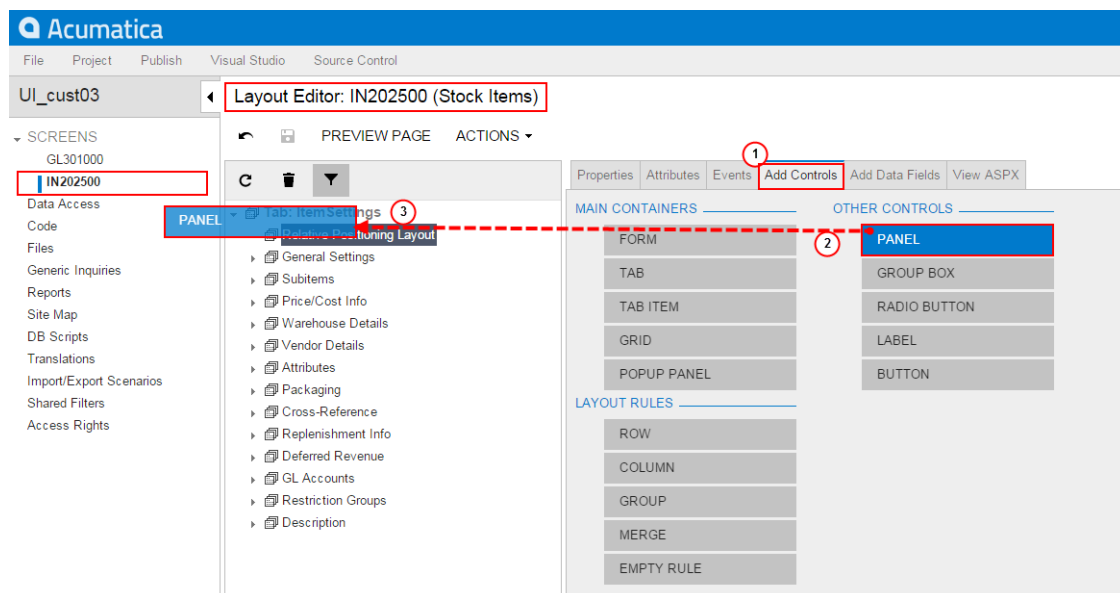


Figure: Adding a panel onto the created *PXTabItem* container control

4. Select the *Panel: CstPanel1* node that has been added.
5. Select the **Properties** tab of the Layout Editor.
6. Select the **Caption** property and enter `Audit` to specify the name of the new panel.

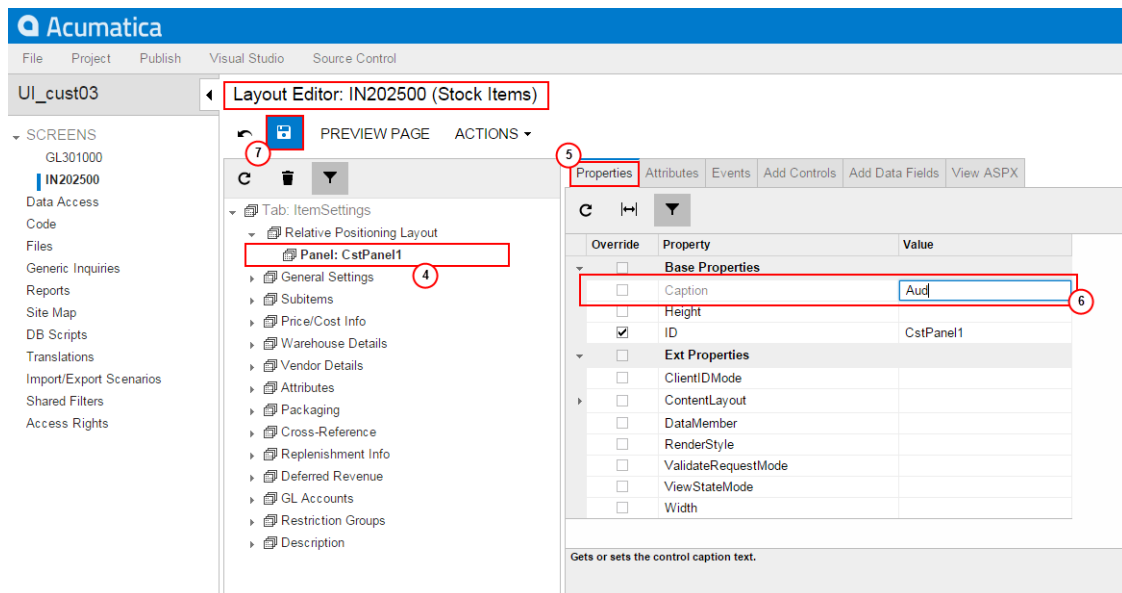


Figure: Defining the name of the added *PXPANEL* control

7. Click **Save** on the toolbar of the Layout Editor to save changes to the current customization project.

Thus, to add a new control to a container, you have to open the Layout Editor, find the container in the Control Tree, select the **Add Controls** tab in the editor, find the required type of the control, and drag and drop the needed control onto the container. Then you can specify the properties of the created control.

Adding a UI Control onto the *PXPANEL* Control

The panel is still not visible on the tab because it doesn't contain any visible controls. Add a control to the panel, as described below:

1. Select the *Panel: Audit* node in the tree of controls.
2. Open the **Add Data Fields** tab of the Layout Editor.
3. Select the **CreatedByID** field in the list that at the moment displays the visible fields of the `InventoryItem` data access class.
4. Click **Create Controls** on the tab toolbar (see the screenshot below).

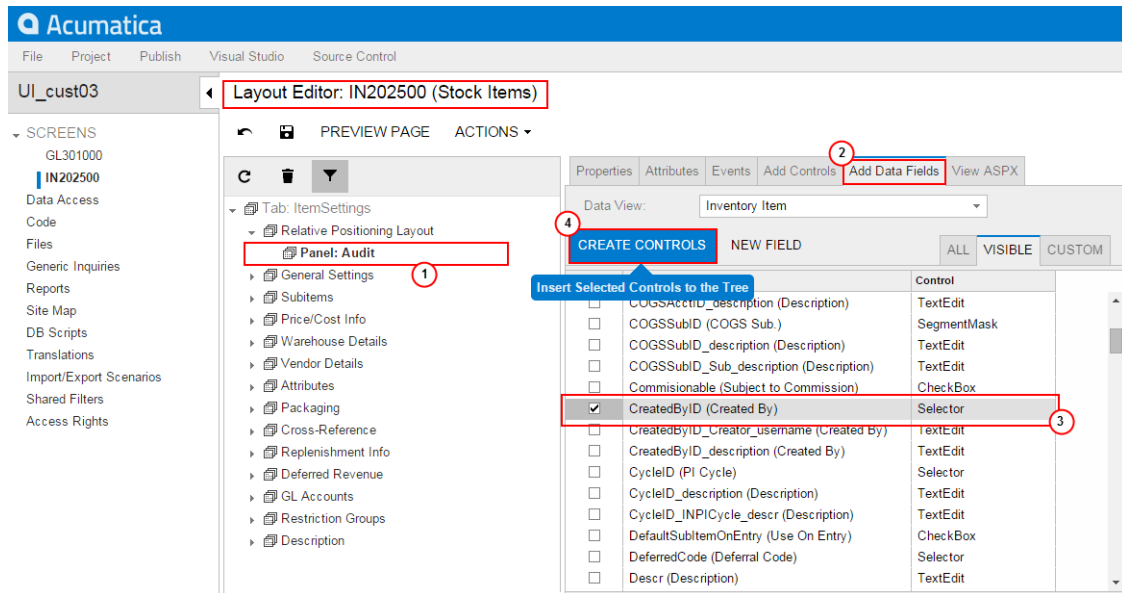


Figure: Adding a UI control to the *PXPanel* container control

5. Click the *Panel: Audit* node to expand and view the created control in the tree of controls.
6. Open the **Add Controls** tab of the Layout Editor.
7. Drag the **Empty Rule** (*PXLayoutRule*) control and drop it above the *Created By* control in the tree, as shown in the screenshot below.

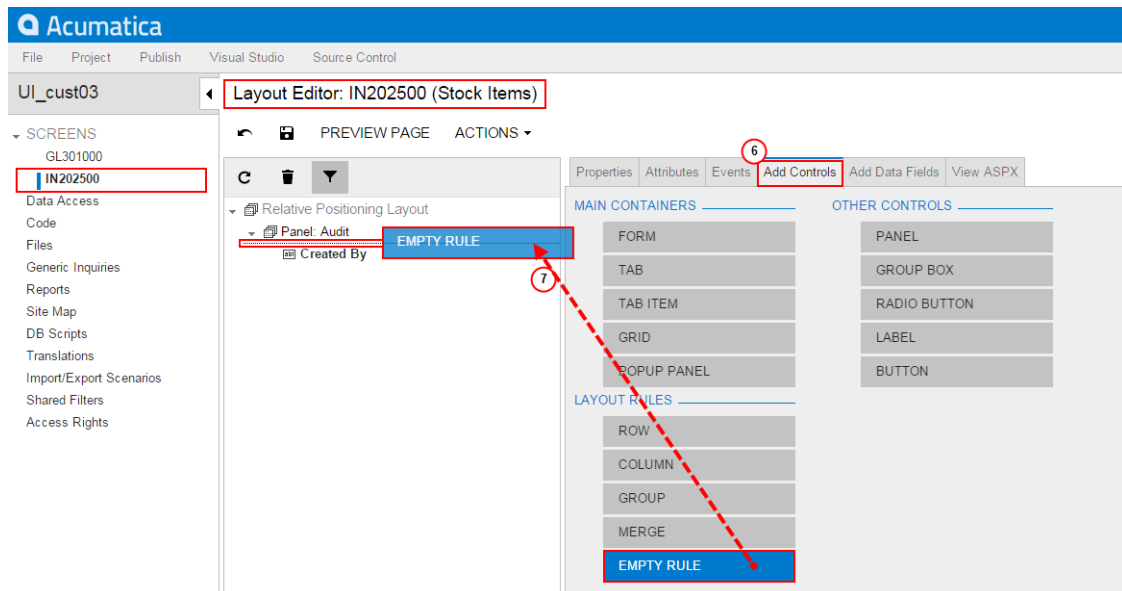


Figure: Adding the *PXLayoutRule* onto the panel

8. In the Control Tree, select the *Layout Rule* node that has been added.
9. Select the **Properties** tab of the Layout Editor.
10. Type *SM* as the **ControlSize** property value and *S* as the **LabelWidth** property value.
11. Set the **StartRow** property to *True*, as shown in the screenshot below.

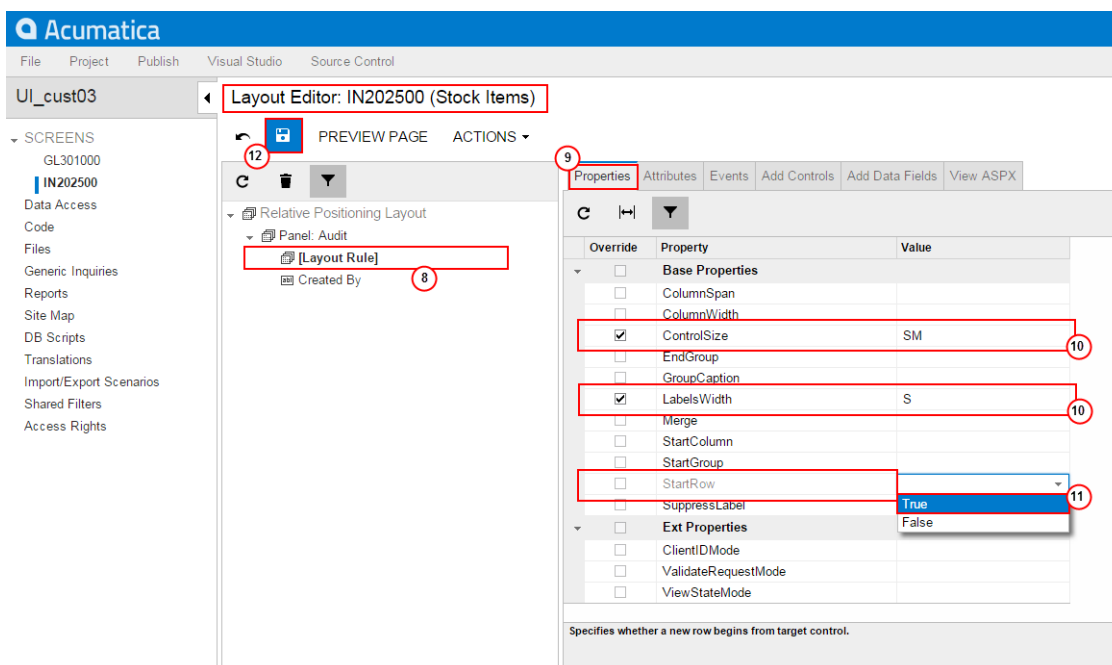


Figure: Setting property values for the added PXLayouRule object

12. Click **Save** on the toolbar of the Layout Editor to save changes to the customization project.

Thus, to apply a layout rule to controls, you have to place the layout rule above the controls. Then you can specify the properties of the created layout rule.

To view the changes, click **Preview Page** on the toolbar or publish the project and review the changes on the *Stock Items* form after the customization has been applied.

The control is displayed on the panel on the Relative Positioning Layout tab (see the screenshot below) that has been added in [Adding a New Tab Item](#).

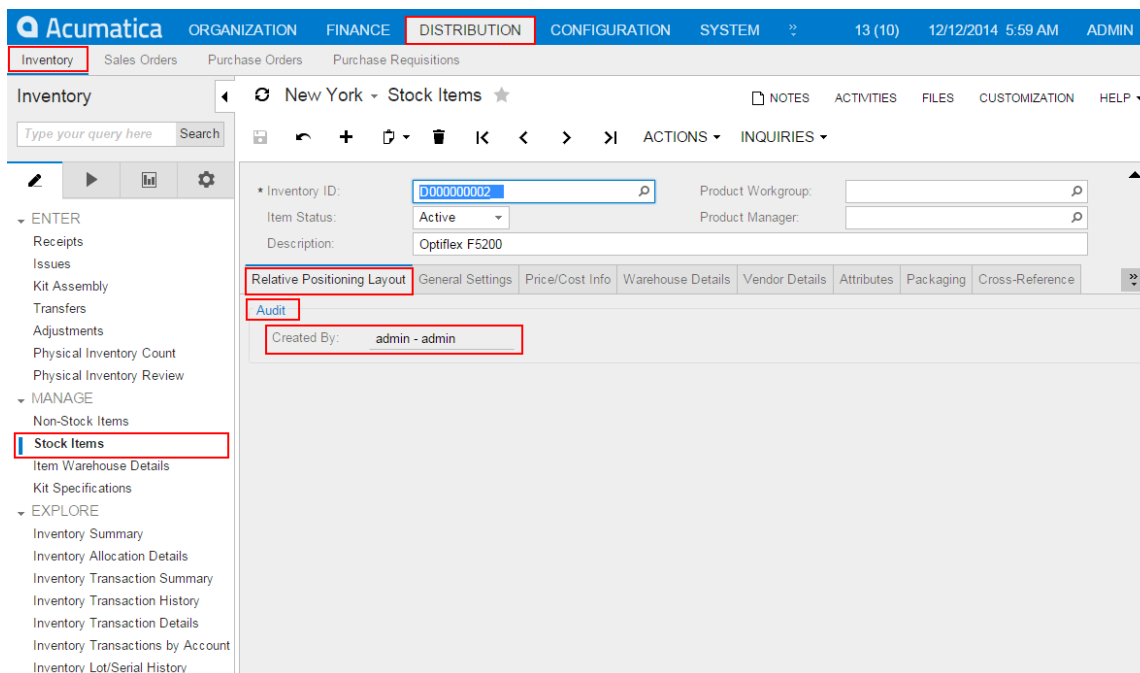


Figure: Viewing the added UI control at run time

To view the fragment of the .aspx code that represents the customization result, open the form in the Layout Editor and select the **View ASPX** tab. The customized fragment is highlighted in yellow, as the screenshot below illustrates.

The screenshot shows the Acumatica Layout Editor interface. The left sidebar displays a tree view of the form structure, with 'IN202500' selected. The main area shows the 'View ASPX' tab, displaying the ASPX code for the 'Relative Positioning Layout'. The code is highlighted in yellow, showing the following structure:

```

<px:PXTab ID="tab" runat="server" Width="100%" Height="606px" DataSourceID="ds" DataMember="ItemSettings" FilesIndicator="False" NoteIndicator="False">
  <AutoSize Enabled="True" Container="Window" MinHeight="150" ></AutoSize>
  <Items>
    <px:PXTabItem Text="Relative Positioning Layout">
      <Template>
        <px:PXPanel runat="server" ID="CstPanel1" Caption="Audit">
          <px:FXLayoutRule runat="server" ID="CstLayoutRule6" ControlSize="SM" LabelsWidth="5" StartRow="True" />
          <px:FXSelector runat="server" ID="CstFXSelector5" DataField="CreatedByID" />
        </px:PXPanel>
      </Template>
    </px:PXTabItem>
    <px:PXTabItem Text="General Settings">
      <Template>
        <px:FXLayoutRule runat="server" StartColumn="True" LabelsWidth="5M" ControlSize="5M" ></px:FXLayoutRule>
        <px:FXLayoutRule runat="server" StartGroup="True" GroupCaption="Item Defaults" ></px:FXLayoutRule>
        <px:FXSelector CommitChanges="True" ID="edItemClassID" runat="server" DataField="ItemClassID" AllowEdit="True" ></px:FXSelector>
        <px:FXDropDown ID="edItemType" runat="server" DataField="ItemType" ></px:FXDropDown>
        <px:FXCheckBox SuppressLabel="True" ID="chkKitItem" runat="server" DataField="KitItem" ></px:FXCheckBox>
        <px:FXDropDown CommitChanges="True" ID="edValMethod" runat="server" DataField="ValMethod" ></px:FXDropDown>
        <px:FXSelector ID="edTaxCategoryID" runat="server" DataField="TaxCategoryID" AllowEdit="True" CommitChanges="True" AutoRefresh="True" ></px:FXSelector>
        <px:FXSelector CommitChanges="True" ID="edPostClassID" runat="server" DataField="PostClassID" ></px:FXSelector>
      </Template>
    </px:PXTabItem>
  </Items>
</px:PXTab>

```

Figure: Viewing the changes to the layout of the form

If you need to analyze the added content (changeset) of the customization project, open the project in the Project Editor, choose the **Edit Project Items** command on the **File** menu item and select the **Object Name** field of the customized object (see the screenshot below). The Page item represents the changes to the .aspx code.

The screenshot shows the Acumatica Project Editor interface. The 'File' menu is open, and 'Edit Project Items' is selected. The 'Object Name' field is highlighted, and the source code for the selected page is displayed in the main area. The source code is highlighted in yellow, showing the following structure:

```

<Page path="~/pages/in/in202500.aspx" ControlID="6" pageSource="5T37b+M2kz9fgf4Pgu/64Q7obh77aLvNH16xN7tB83DjLbA4Ra0h3riyy5Ep2N+8P">
  <PXTab ID="tab" ParentID="pHG_tab" TypeFullName="PX.Web.UI.PXTab">
    <Children Key="Items">
      <AddItem>
        <PXTabItem TypeFullName="PX.Web.UI.PXTabItem">
          <Prop Key="Text" Value="Relative Positioning Layout" />
          <Children Key="Template">
            <AddItem>
              <PXPanel TypeFullName="PX.Web.UI.PXPanel">
                <Prop Key="Virtual:ApplyStyleSheetSkin" />
                <Prop Key="ID" Value="CstPanel1" />
                <Prop Key="Caption" Value="Audit" />
                <Children Key="Controls">
                  <AddItem>
                    <PXLayoutRule TypeFullName="PX.Web.UI.PXLayoutRule">
                      <Prop Key="Virtual:ApplyStyleSheetSkin" />
                      <Prop Key="ID" Value="CstLayoutRule6" />
                      <Prop Key="ControlSize" Value="SM" />
                    </PXLayoutRule>
                  </AddItem>
                </Children>
              </PXPanel>
            </AddItem>
          </Children>
        </PXTabItem>
      </AddItem>
    </Children>
  </PXTab>
</Page>

```

Figure: Analyzing the added content of the customization project

Adding Columns to a Grid

Suppose that you need to add a column to the details table (grid) of the *Journal Transactions* form shown in the screenshot below.

	* Branch	* Account	Description	* Subaccount	Project	Project Task	Ref. Number	Quantity	UOM	Debit Amount
	MAIN	200000	Accounts Paya...	US-00-00-00-000	X		000678	0.00		0.00
	SOUTH	698103	Inter-branch Ex...	US-PU-HS-00-...	X		000678	100.00		11,500.00
	MAIN	198100	Inter-branch ma...	SG-00-00-00-000	X			100.00		11,500.00
	SOUTH	198100	Inter-branch ma...	SG-00-00-00-000	X			100.00		0.00

Figure: Viewing the original form

For instance, suppose that you need to add the column for the *CreateDateTime* data field to the position after the **Ref. Number** column.

To do this, perform the following actions:

1. On the *Journal Transactions* form, click the **Customization** menu, choose **Inspect Element**, select the **Ref. Number** column header, and click **Customize** on the **Element Properties** dialog box.



: The Layout Editor appears with only the grid node displayed in the Control Tree. All other containers are hidden by the filter of the editor. The inspected **Ref. Number** node is already selected and highlighted in bold.

2. In the Layout Editor, open the **Add Data Fields** tab, as shown in the screenshot below.
3. Apply the **All** filter to the list of the fields.
4. Select the *CreateDateTime* field in the list.
5. Click **Create Controls** in the tab toolbar.



: New columns are added to the grid in the position after the item selected in the Control Tree of the Layout Editor. If you need to change the placement of a UI control, select the control on the tree, drag and drop it to the needed place.

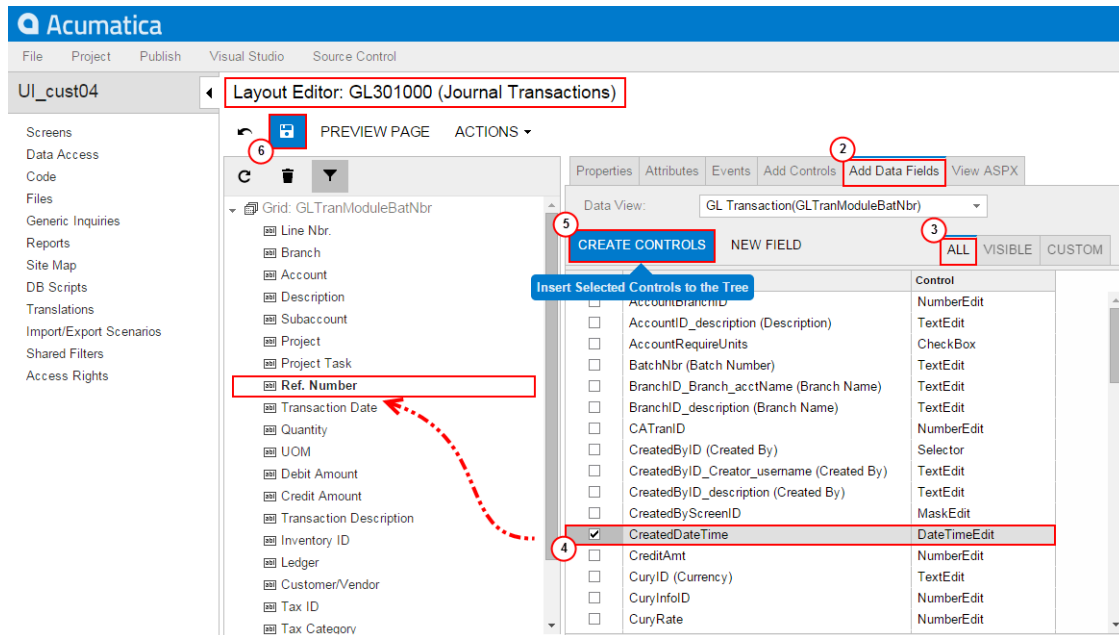


Figure: Adding the column to the grid

6. Click **Save** on the toolbar of the Layout Editor to save changes to the current customization project.

Thus, to add new columns to a grid, you have to open the grid node on the Control Tree of the Layout Editor, select the **Add Data Fields** tab in the editor, select needed fields in the list on the tab, click **Create Controls**, and then drag and drop new columns to desired positions inside the grid. You can add a column for any field that belongs to any data access class (DAC) of any data view of the business logic controller bound to the form. To select a DAC to view its fields in the list, use the **Data View** dropdown list on the **Add Data Fields** tab.

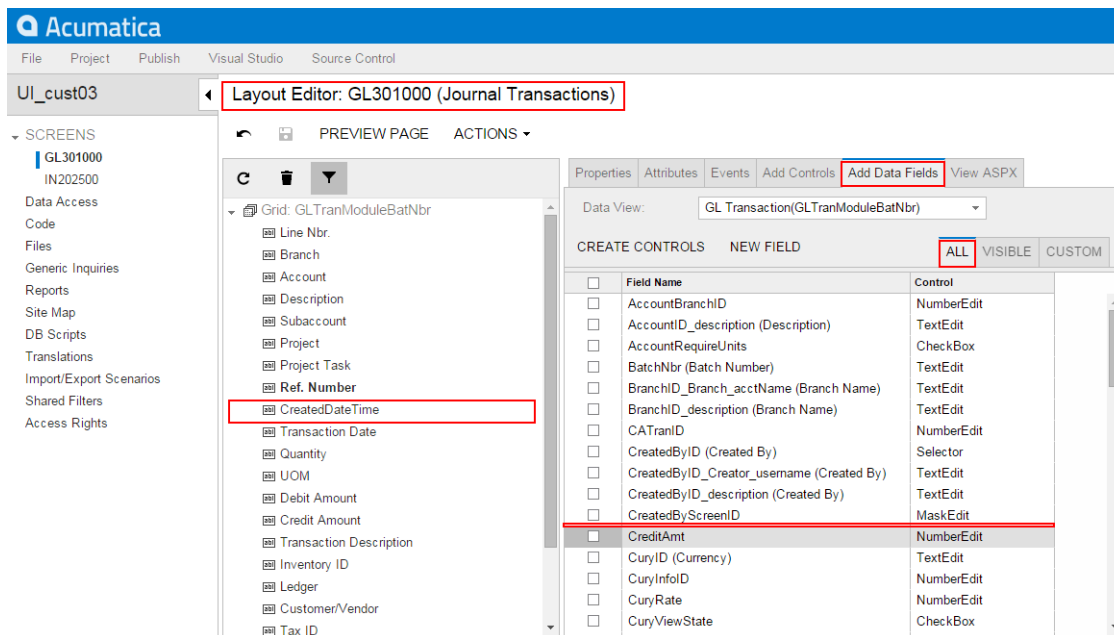


Figure: Viewing the Layout Editor with the created column

To view the changes on the form, click **Preview Page** on the toolbar or publish the project and review the changes on the *Journal Transactions* form after the customization has been applied.

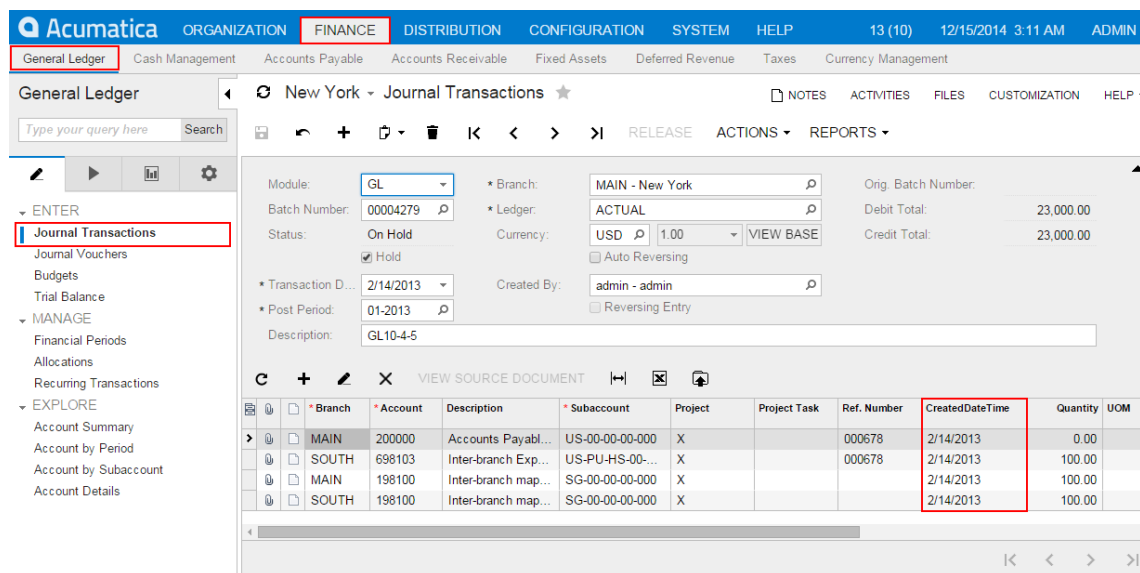


Figure: Viewing the customized form

To view the fragment of the .aspx code that represents the customization result, open the form in the Layout Editor and select the **View ASPX** tab. The customized fragment is highlighted in yellow, as the screenshot below illustrates.

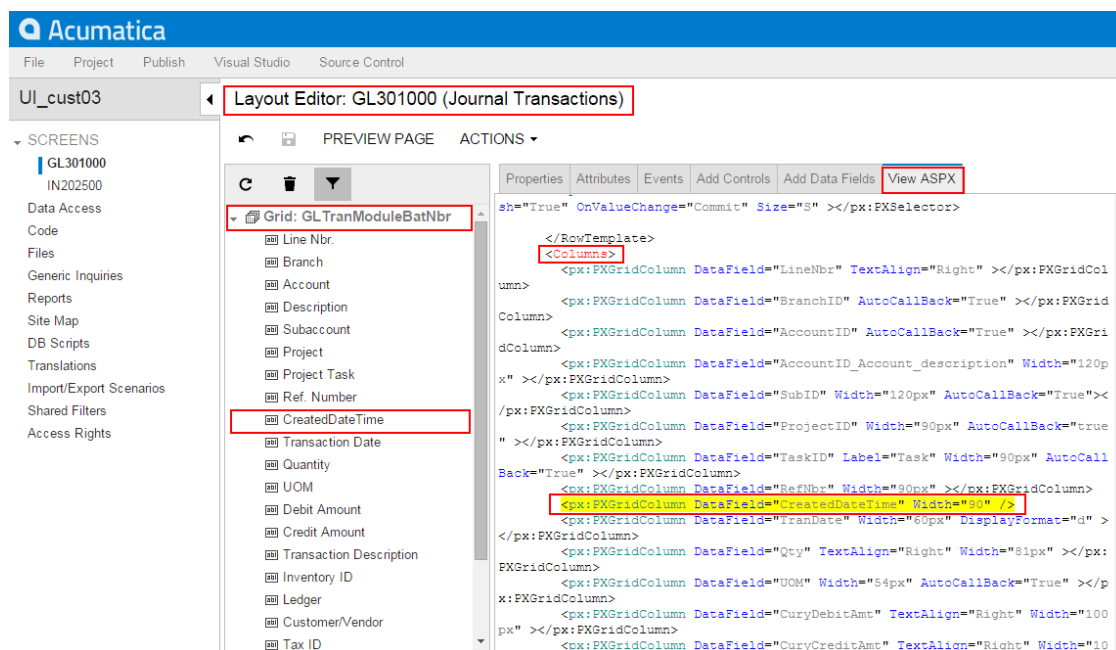


Figure: Analyzing the changeset of the .aspx code

To analyze the changes to the layout of the form introduced by the new column, open the project in the Project Editor, choose the **Edit Project Items** command on the **File** menu item and select the **Object Name** field of the customized object (see the screenshot below). You can see the new **Page** item that contains the .aspx code changeset.

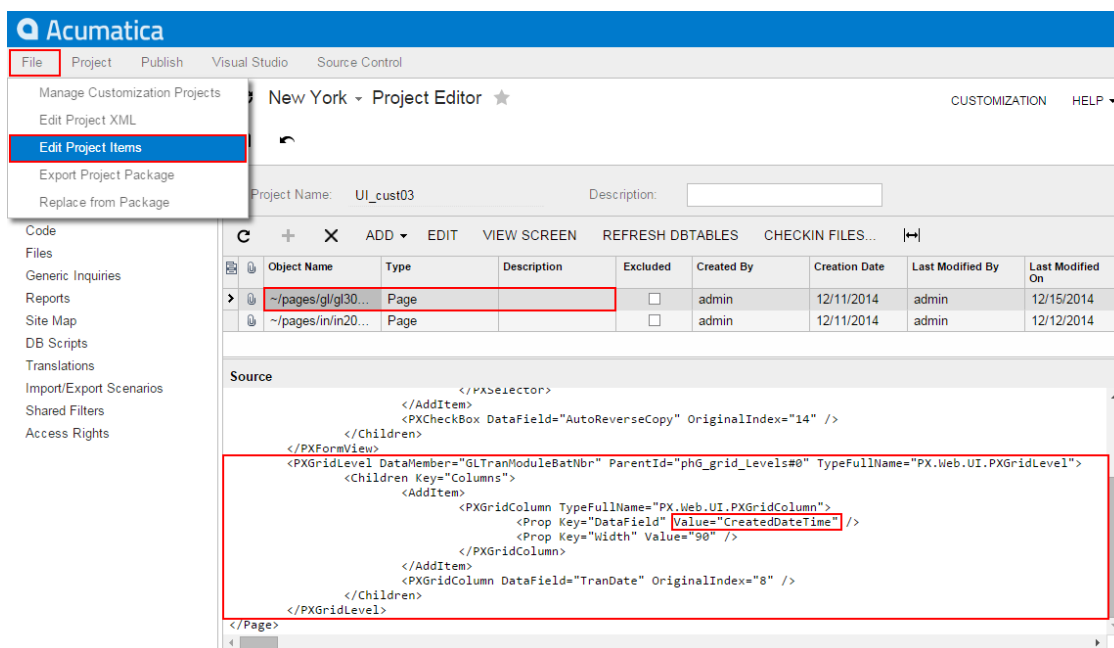


Figure: Analyzing the added content of the customization project

Modifying Columns in a Selector

If you add columns to a selector, you give the user the capability to see more information about each item in the lookup window, so the user can select the appropriate data record.

The screenshot below illustrates the original view of the **Batch Number** lookup window (also referred to as *selector*). The **Batch Number** is a control on the form of the *Journal Transactions* form. The lookup window of this control includes eight columns, as shown in the screenshot below.

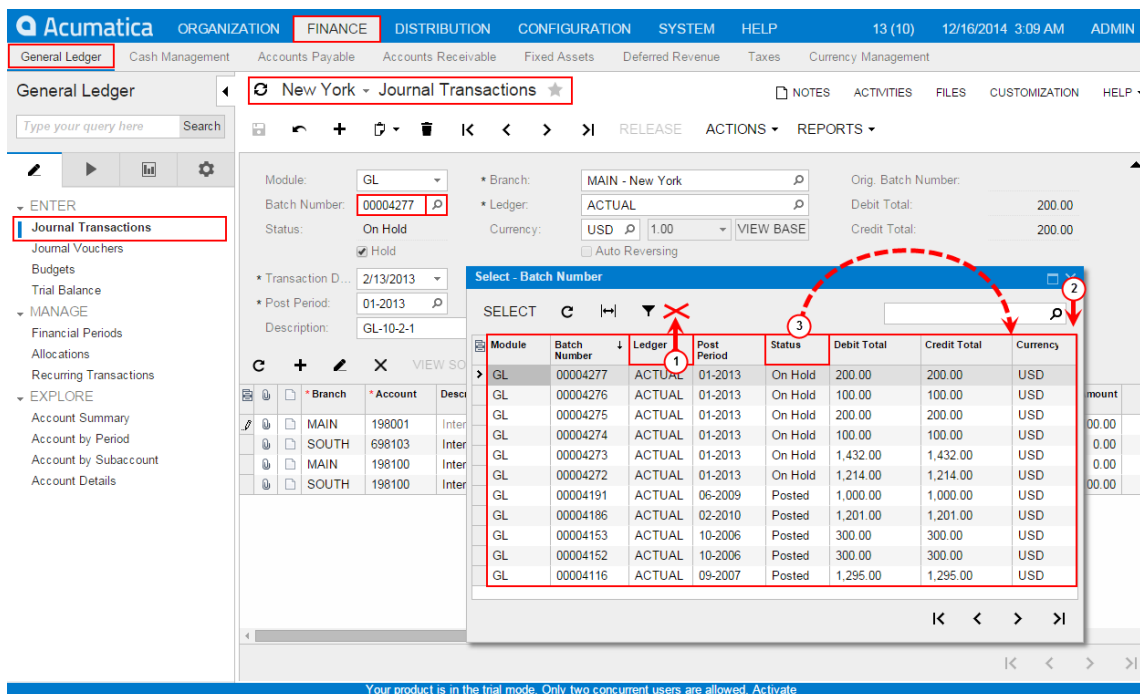


Figure: Viewing the current structure and content of the lookup window

Suppose the UI customization task for the **Batch Number** selector includes the following steps (see the screenshot above):

1. Deleting an existing column, such as the **Ledger** column.
2. Adding a column for the **Orig. Batch Number** field to the place after the **Currency** column.
3. Changing the order of the selector columns, so the **Status** column moves to the position between the **Credit Total** and **Currency** columns.

To do this, perform the following actions:

1. On the *Journal Transactions* form (see the screenshot below), click the **Customization** menu (a), choose **Inspect Element** (b), select the **Batch Number** control (c), click **Actions** on the **Element Properties** dialog box (d) and select the **Customize Data Fields** command (e).

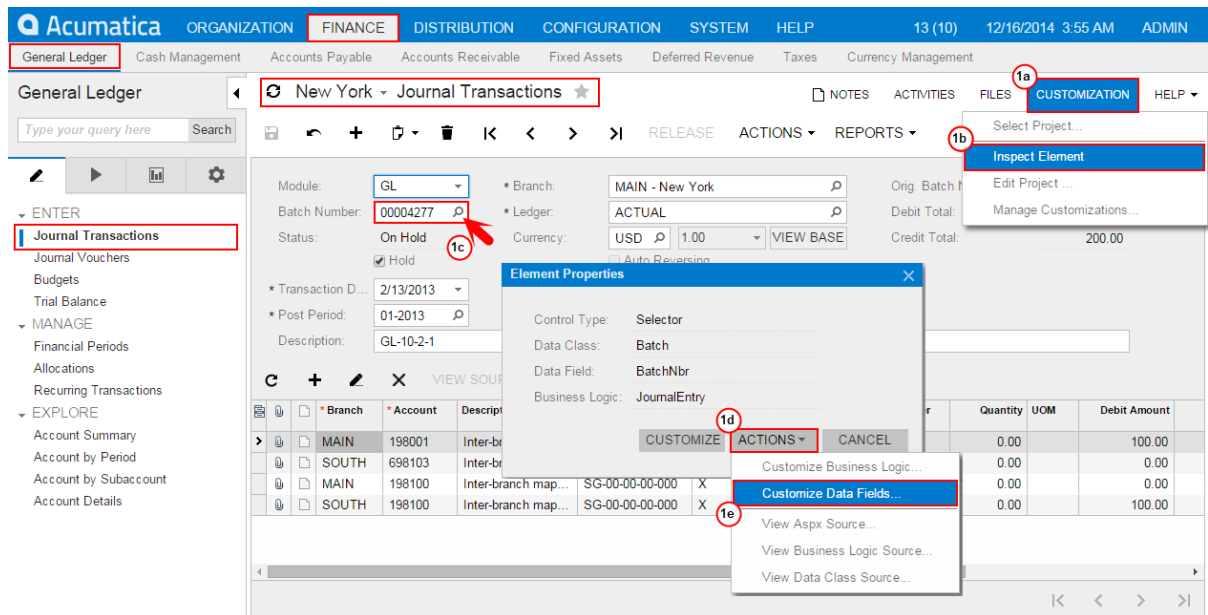


Figure: Selecting the control for the selector customization

The Data Class Editor appears with only the **BatchNbr** field in the list of the customized fields of the class. The field is already selected in the field; therefore the work area of the editor holds information about the field.



: The selected field type is selector, so the **Selector Columns** button is enabled in the editor toolbar.

2. To open the **Customize Selector Column** dialog box, click **Selector Columns**, as shown in the screenshot below.
3. In the table of the **Customize Selector Column** dialog box, select the row with the **Ledger** column.
4. Click **Delete Row** to delete the selected **Ledger** column from the selector.

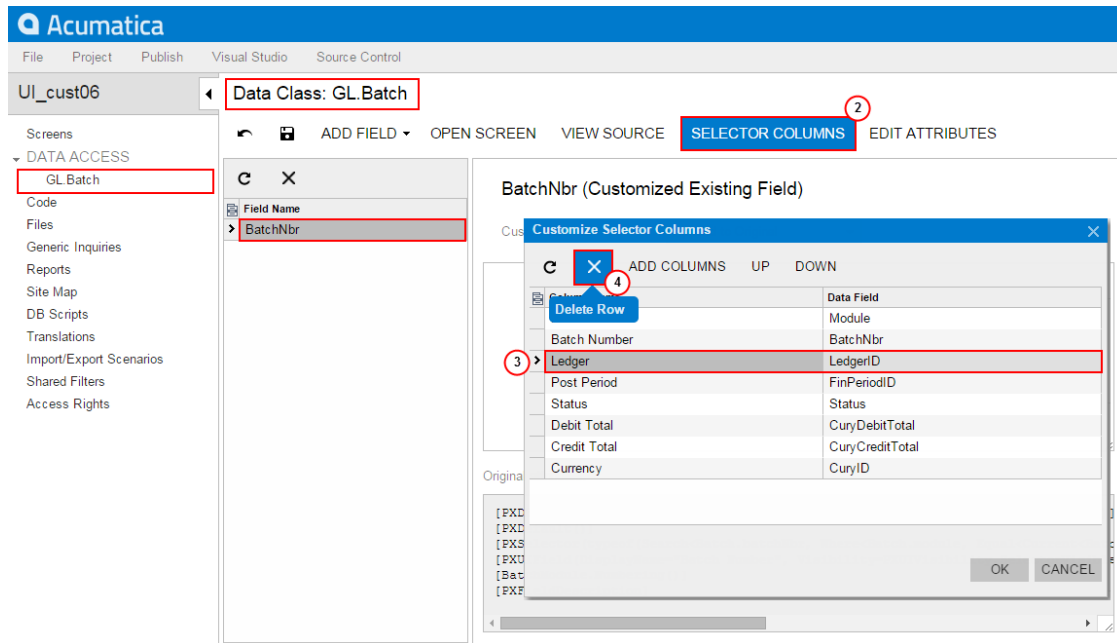


Figure: Deleting the selected column from the selector

5. To open the **Add Columns to Selector** dialog box, click **Add Columns** on the toolbar of the **Customize Selector Column** dialog box.
6. In the **Add Columns to Selector** dialog box, select the **All** tab to view all fields of the customized data access class (DAC) in the table.
7. In the table, select the row with the **Orig. Batch Number** column name.
8. Click **OK** to add the selected field to the table of the **Customize Selector Column** dialog box.

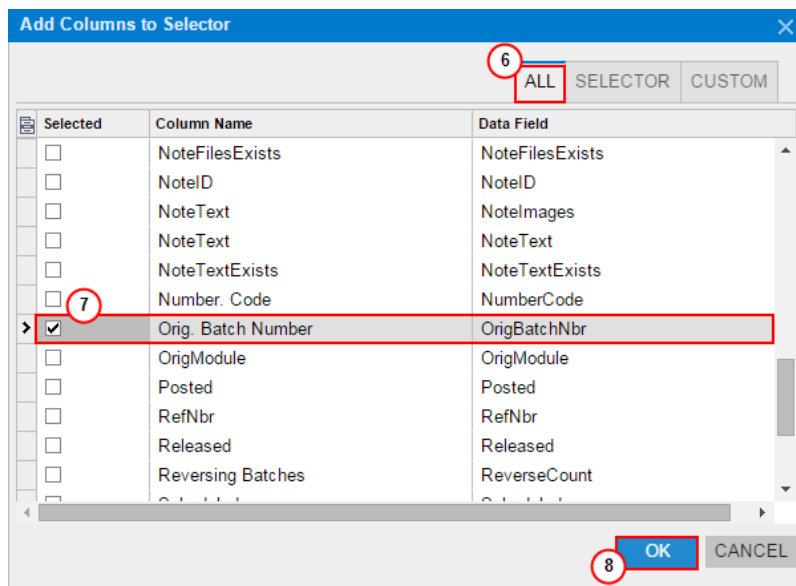


Figure: Adding the column for the field to the table of the Customize Selector Column Dialog



: New columns are added to the position after the last existing column of the selector.

The **Orig. Batch Number** column is created in the table of the **Customize Selector Column** dialog box (see the screenshot below) at the rightmost position. Therefore, you do not need to move this column inside the selector. But you still have to move the **Status** column to position between the **Credit Total** and **Currency** columns.

9. Select the row with the **Status** column.
10. Click **Down** twice to move the selected row to the required position, as shown in the screenshot below.
11. Click **OK** on the **Customize Selector Column** dialog box to apply the changes to the selector.

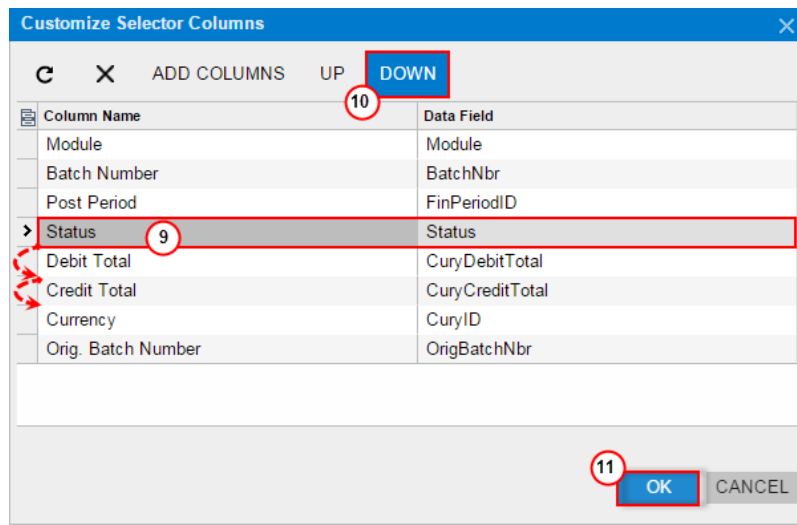


Figure: Moving the column to the needed position

12. Click **Save** on the toolbar of the Data Class Editor to save the changes to the current customization project.

After you click **OK** in the **Customize Selector Columns** dialog box, the system applies the modifications to the selector table. As a result, the `PXCustomizeSelectorColumns` attribute is added to the selector field and you can view the attribute in the **Customize Attributes** text area of the Data Class Editor (see the screenshot below). This attribute defines the new set and order of the columns in the selector.

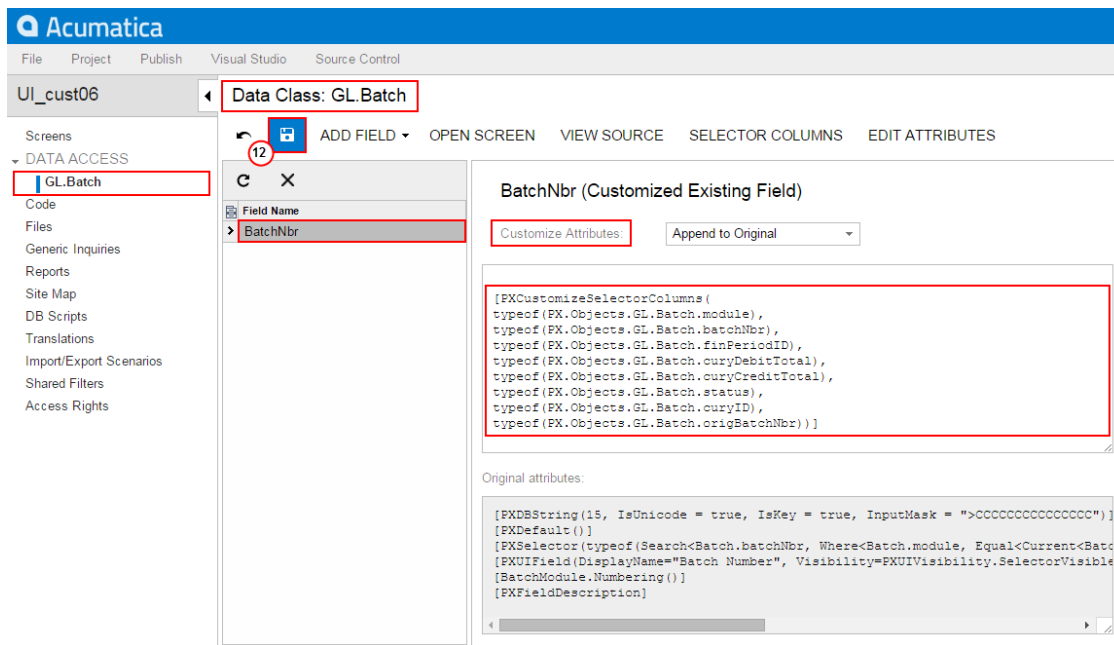


Figure: Viewing the changes in the Customize Attributes text area

After publishing the project, open the *Journal Transactions* form, click the Magnifier icon of the **Batch Number** control, and view the lookup window. Notice the absence of the deleted column, the position of the moved **Status** column, and the presence of the **Orig. Batch Number** column, as shown in the screenshot below.

Module	Batch Number	Post Period	Debit Total	Credit Total	Status	Currency	Orig. Batch Number
GL	00003913	06-2007	480.00	480.00	Posted	USD	
GL	00003909	06-2007	480.00	480.00	Posted	USD	
GL	00002995	01-2010	400.00	400.00	Posted	USD	00002983
GL	00002994	12-2009	400.00	400.00	Posted	USD	00002983
GL	00002993	11-2009	400.00	400.00	Posted	USD	00002983
GL	00002992	10-2009	400.00	400.00	Posted	USD	00002983
GL	00002991	09-2009	400.00	400.00	Posted	USD	00002983
GL	00002990	08-2009	400.00	400.00	Posted	USD	00002983
GL	00002989	07-2009	400.00	400.00	Posted	USD	00002983
GL	00002988	06-2009	400.00	400.00	Posted	USD	00002983
GL	00002987	05-2009	400.00	400.00	Posted	USD	00002983
GL	00002986	04-2009	400.00	400.00	Posted	USD	00002983

Figure: Viewing the modified structure and content of the lookup window

If you need to analyze the added content (changeset) of the customization project, open the project in the Project Editor, choose the **Edit Project Items** command on the **File** menu item, and select the **Object Name** field of the customized object (see the screenshot below). The *DAC* object represents the changes to the data access class code.

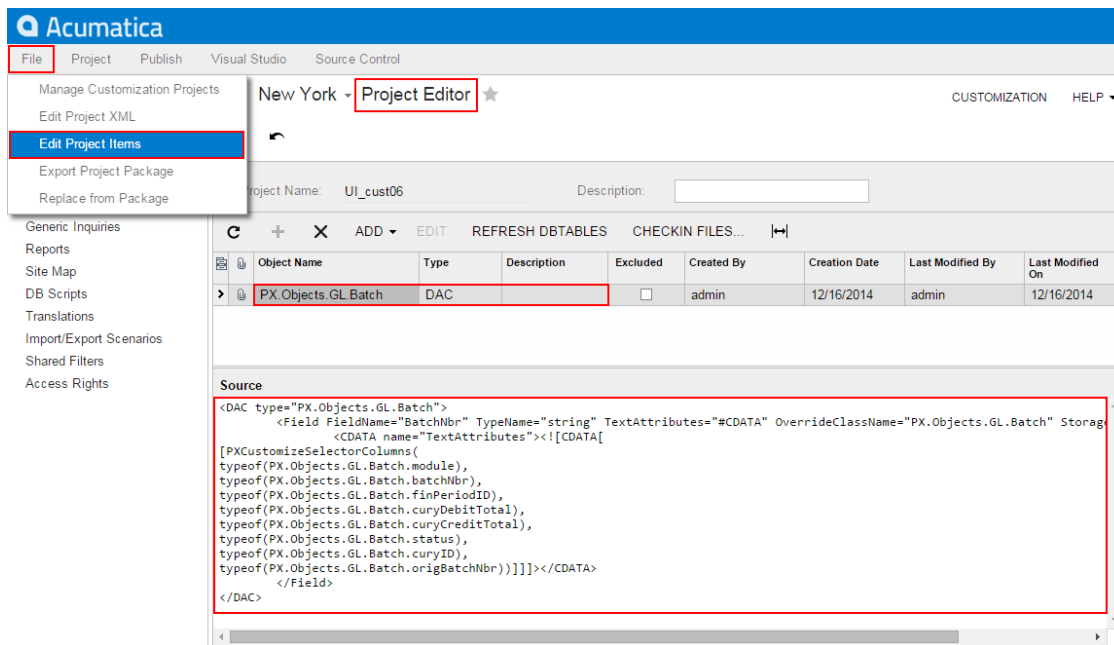


Figure: Analyzing the added content of the customization project

Adding PXLayoutrule Components

This topic describes in detail the use of the `PXLayoutrule` component, which facilitates a relative positioning layout.

The `PXLayoutrule` component, which you access from the Layout Editor, provides the following UI customization capabilities:

- Placing controls in multiple columns to uniformly distribute them on the form or tab area of a form
- Spanning controls across multiple columns
- Merging controls into one row to align them horizontally
- Adjusting the widths of controls and labels
- Hiding the labels of controls
- Grouping controls for users' convenience

This topic describes the most important properties of the `PXLayoutrule` component, which are visible when the **Properties** tab on the Layout Editor is selected (see the screenshot below).

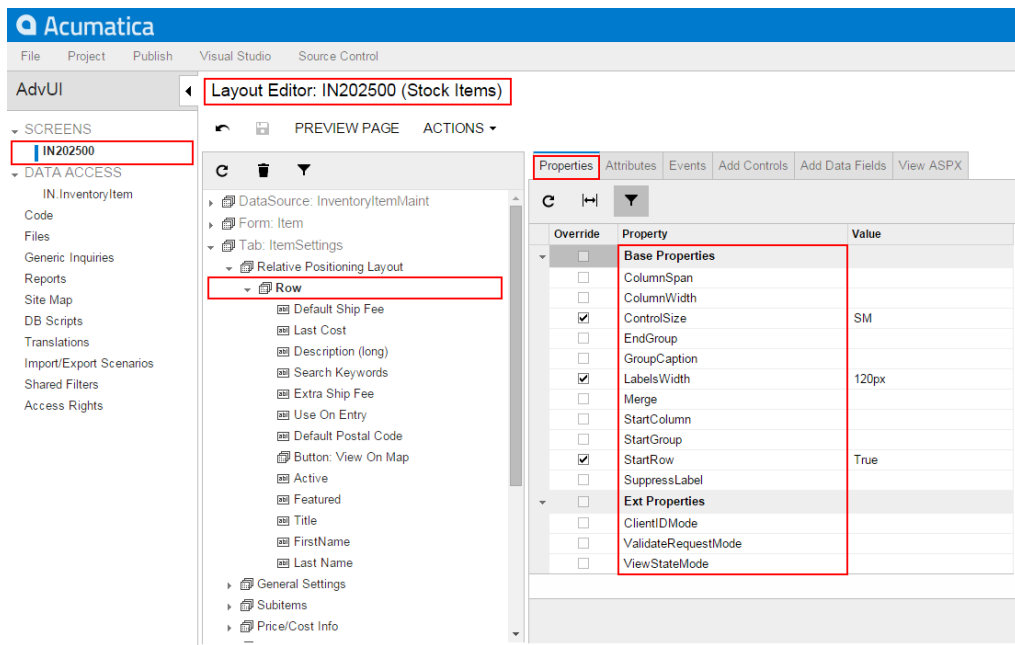


Figure: Viewing the PXLayoutRule component and its most important properties

The nodes of some levels and the subnodes of the lowest level of the Control Tree of the Layout Editor represent the container controls and the simple controls of the selected area of the form. In this example, the **Relative Positioning Layout**, **General Settings**, and other low-level nodes of the `PXTabItem` type correspond to the tabitem controls, while the **FirstName** and **Last Name** subnodes of the `PXTextEdit` type correspond to the `UsrSprFirstName` and `UsrSprLastName` data fields and are placed under the **Relative Positioning Layout** node. The Control Tree also contains two high-level nodes. The highest ones represent placeholders, while the nodes beneath them correspond to the container controls, such as `PXFormView`, `PXTab`, and `PXGrid`. Nodes of the `PXTab` include `PXTabItem` nodes, while `PXTabItem` nodes include the subnode that corresponds to the controls. To see the subnode, you should extract appropriate nodes. The `PXLayoutRule` component can be added only on the subnode level.

Suppose that you have performed the following customization actions:

- Appending the new tab to the *Stock Items* form and naming it **Relative Positioning Layout**. (See [Adding a New Tab Item](#).)
- Adding user fields to both the *InventoryItem* database table and the *InventoryItem* data access class (DAC).
- Adding several controls for new and already-existing *InventoryItem* DAC fields onto the created **Relative Positioning Layout** tab.
- Adding the `PXLayoutRule` component with the **StartRow** property value set to `True` as the uppermost subnode of the **Relative Positioning Layout** tab.

The following screenshot illustrates the initial layout of the **Relative Positioning Layout** tab on the form.

The screenshot shows the Acumatica interface for the 'Stock Items' form. The 'DISTRIBUTION' menu is highlighted. The 'Stock Items' tab is active, and the 'Relative Positioning Layout' sub-tab is selected. The form displays various fields and controls, including:

- Inventory ID: [Text Field]
- Item Status: Active (Dropdown)
- Description: [Text Field]
- Product Workgroup: [Text Field]
- Product Manager: [Text Field]
- Default Ship Fee: 0.00 (Text Field)
- Last Cost: 0.000000 (Text Field)
- Description (long): [Text Field]
- Search Keywords: [Text Field]
- Extra Ship Fee: 0.00 (Text Field)
- Use On Entry: (Checkbox)
- Default Postal Co.: [Text Field]
- VIEW ON MAP (Button)
- Active: (Checkbox)
- Featured: (Checkbox)
- Title: [Text Field]
- FirstName: [Text Field]
- Last Name: [Text Field]

Figure: Viewing the initial layout of the added tab on the Stock Items form



: You must always set the **StartRow** property value to *True* for the uppermost `PXLayoutRule` subnode to provide the proper layout; we assume that you have taken this action already. The **StartRow** property is described in the next section.

The following sections describe the most important properties of the `PXLayoutRule` component in the order that corresponds to the order of actions within the typical complex customization task, which must be resolved step by step. All the customization actions are described with the assumption that you have already opened the Layout Editor, and the first instruction starts when you begin to work with the editor.



: You should not actually complete the actions described below. You should only analyze the instructions and results of performing them.

In this section:

- [StartRow and StartColumn Properties](#)
- [ColumnSpan Property](#)
- [Merge Property](#)
- [GroupCaption and EndGroup Properties](#)
- [SupressLabel Property](#)
- [Nested Container Controls](#)
- [The Results of UI Customization that Uses the PXLayoutRule Component](#)

StartRow and StartColumn Properties

By default, the system places all controls into a column. You need to initially set to *True* the **StartRow** property value for the uppermost `PXLayoutRule` subnode to optimize the layout. Controls are placed within a single column, until you add the `PXLayoutRule` component with the **StartColumn** or **Merge** property value set to *True*.

To best use the area of a placeholder, you can place controls in multiple columns within a row of a form or tab area by setting the **StartColumn** property value of the `PXLayoutRule` subnode to *True*. This

property creates a new column of controls within the current row. The first subnode under this rule corresponds to the highest control in the column.

Every `PXLayoutRule` component that has the **StartRow** property value set to `True` initializes a new independent set of columns. By setting the **StartRow** property value to `True`, you create a new independent set of controls that are placed by default into a column. To place controls in multiple columns within the new row, you should add the `PXLayoutRule` subnode with the **StartColumn** property value set to `True`.

Every `PXLayoutRule` component that has the **StartRow** or **StartColumn** property value set to `True` must have one of the following sets of properties defined:

- **LabelsWidth** and **ControlSize**
- **LabelsWidth** and **ColumnWidth**

You may not set both **ColumnWidth** and **ControlSize** property values for the same `PXLayoutRule` component; in this case, the system will use the value of the **ControlSize** property.

For all `PXLayoutRule` components with the **StartRow** or **StartColumn** property value set to `True`:

- You can assign the **ColumnWidth** and **LabelsWidth** property values from the predefined list of options. (See [Using Predefined Size Values](#) for details.)
- The values of the **ColumnWidth**, **ControlSize** and **LabelsWidth** properties must be defined exclusively for every `PXLayoutRule` component; they are never inherited from the previously declared one.

You may assign the **StartRow** property only for the uppermost subnode, but this is not an optimal layout for a complex form. Therefore, for this task, you should thrice drag and drop the **Row** layout rule (`PXLayoutRule` with the predefined **StartRow** property value to `True`) to the positions above each of the **Extra Ship Fee**, **Active**, and **Title** subnodes, as shown in the screenshot below.

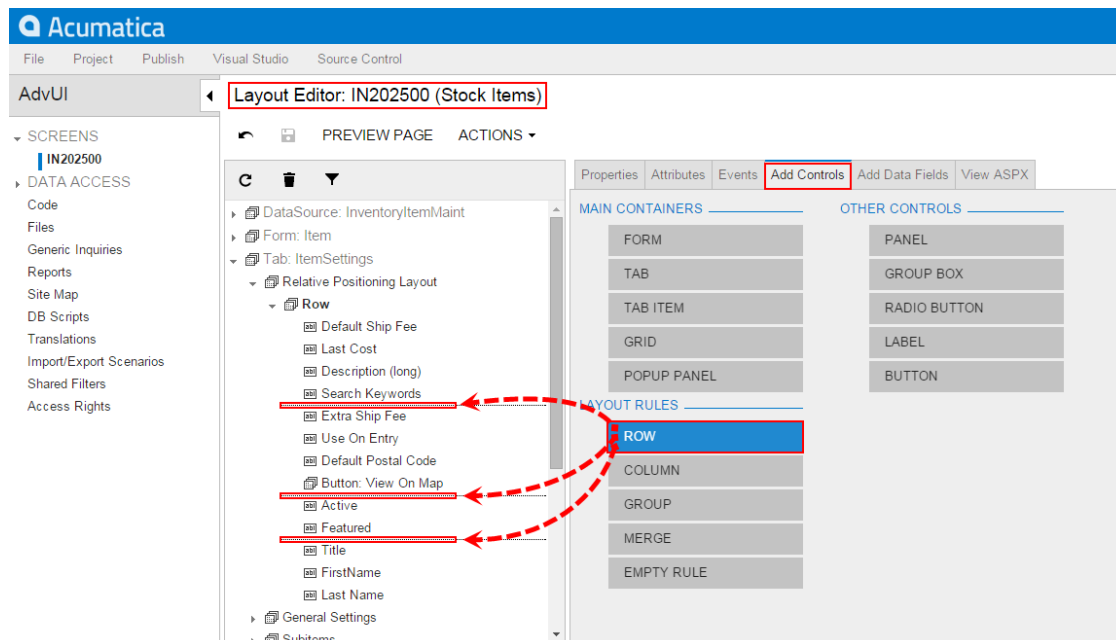


Figure: Adding the Row PXLayoutRule components to the tab item

Because the values of the **ColumnWidth**, **ControlSize** and **LabelsWidth** properties for the rows and columns never inherited from the previously declared `PXLayoutRule` component, you have to define these properties exclusively for every new row, as shown in the screenshot below.

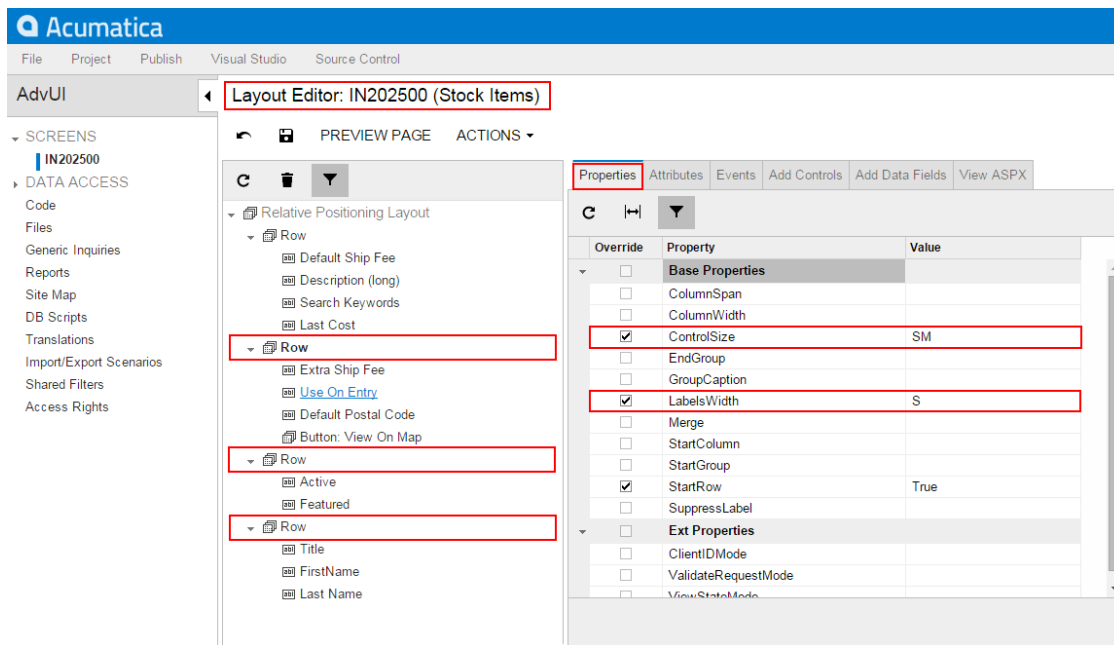


Figure: Specifying the properties for the new PXLayoutRule components

In this example, you should make the **Last Cost** control the highest one in the second column. However you need two controls placed under this control to keep their positions in first column. To resolve this part of the customization task, you should perform the following actions (see also the screenshot below):

1. Drag and drop the **Last Cost** subnode to the last position in the first row.

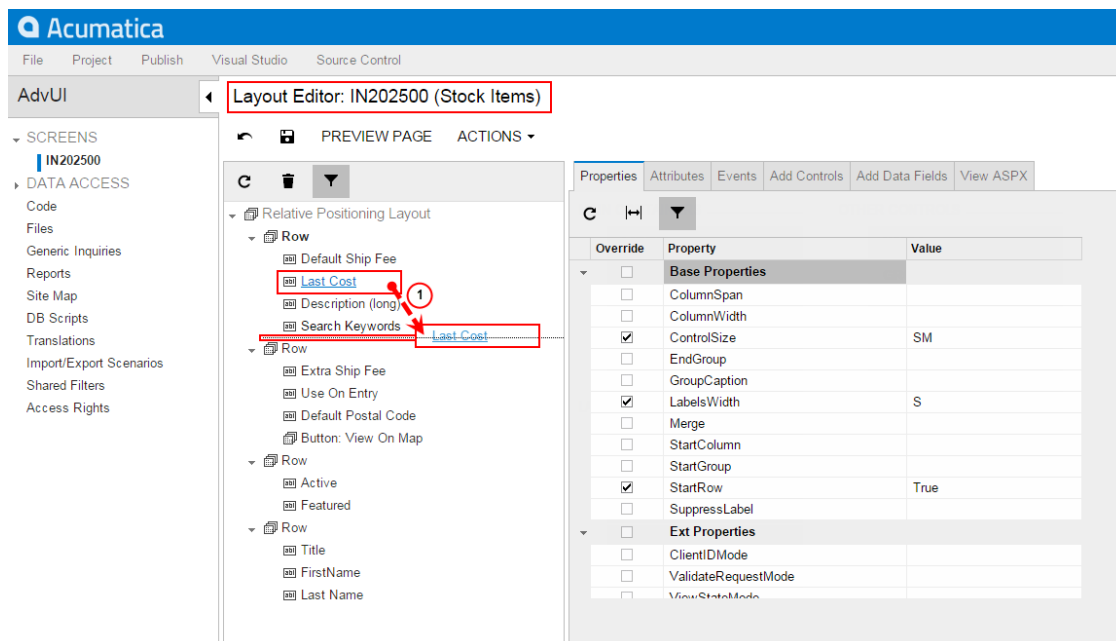


Figure: Moving the subnode in the tree of controls

2. Select the **Add Controls** tab of the Layout Editor.
3. Drag and drop the **Column** layout rule (PXLayoutRule with the predefined **StartColumn** property) above the **Last Cost** node in the tree, as shown in the screenshot below.

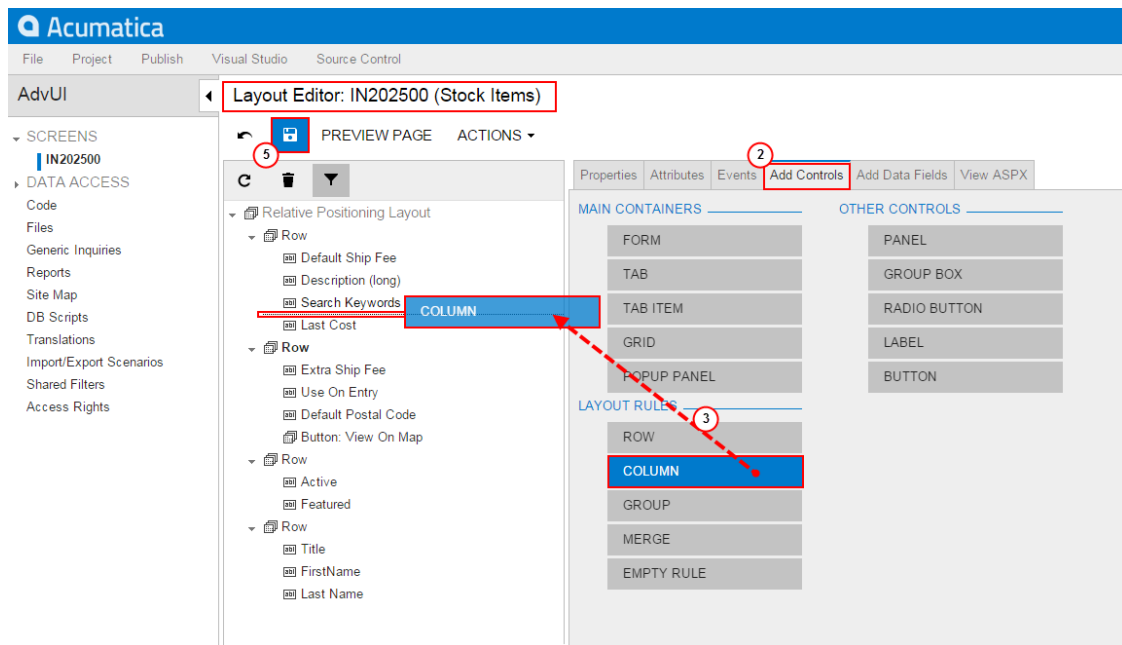


Figure: Adding the Column PXLayouRule component

4. Select the **Properties** tab of the Layout Editor and specify the values for the **ControlSize** (SM) and **LabelWidth** (s) properties for the new column.
5. Click **Save** on the toolbar of the Layout Editor to save changes to the customization project.

You can see the changes of the form by clicking **Preview Page** on the toolbar of the Layout Editor. As the screenshot below illustrates, the **Last Cost** control is now placed on the second column of the **Relative Positioning Layout** tab, while the **Description (long)** and **Search Keywords** input controls still keep their positions in the first column.

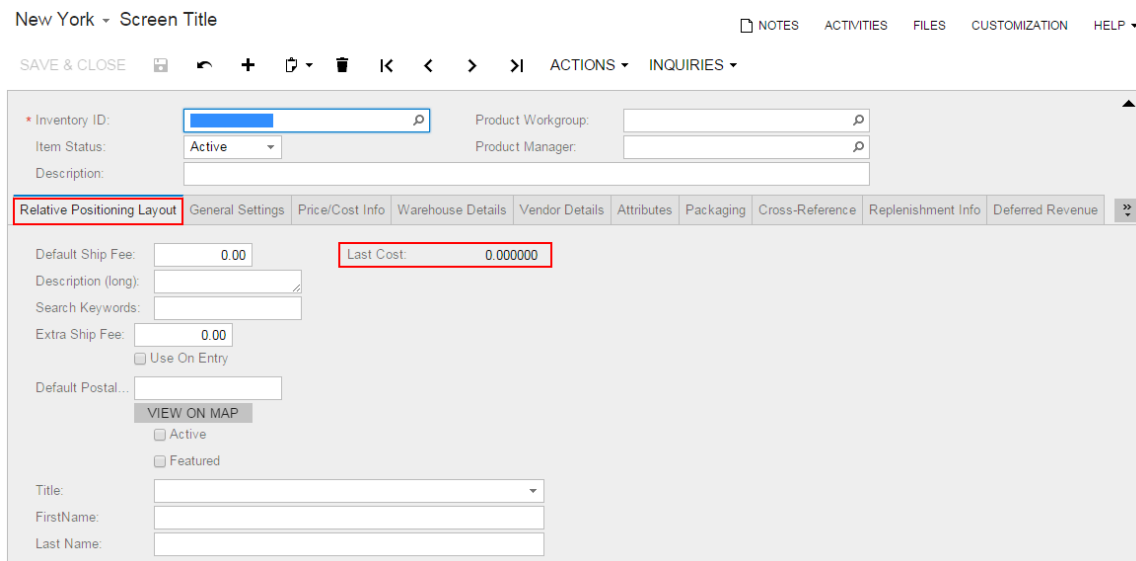


Figure: Viewing the control that was placed on the new column

ColumnSpan Property

Next, you need to widen the **Description (long)** and **Search Keywords** controls so that they span two columns. To do this, you use the **ColumnSpan** property.

You specify the **ColumnSpan** property value for a `PXLayoutRule` subnode by manually typing the number of columns spanned by the control that is the first one placed below the rule. Hence, you should specify this property value twice for each subnode that corresponds to the **Description (long)** and **Search Keywords** controls. Each of these controls (or another type of control) spans the specified number of columns, starting from the column to which it originally belongs.

For all `PXLayoutRule` components with the **ColumnSpan** property value specified:

- The **LabelsWidth** property value is always inherited from the previously declared `PXLayoutRule` component that has the **StartRow** or **StartColumn** property value set to *True*.
- The values for the **ColumnWidth** and **ControlSize** properties are never applied to these `PXLayoutRule` components.

To adjust the **ColumnSpan** property of the controls, you should perform the following actions:

1. Select the **Add Controls** tab of the Layout Editor.
2. Twice drag and drop the **Empty Rule** layout rule (`PXLayoutRule` without any predefined property) — above each of the **Description (long)** and **Search Keywords** nodes in the tree, as shown in the screenshot below.

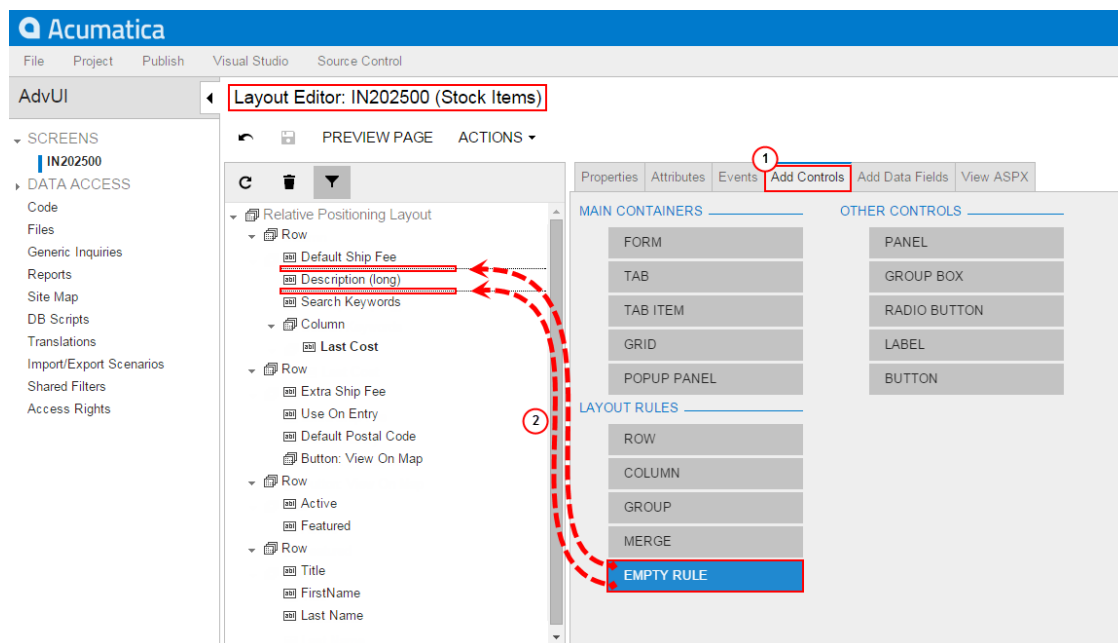


Figure: Adding the Empty Rule `PXLayoutRule` components

3. Select the **Properties** tab of the Layout Editor.
4. Select the **Layout Rule** subnode, which corresponds to the **Description (long)** control.
5. Type 2 as the **ColumnSpan** property value.
6. Select the **Layout Rule** subnode, which corresponds to the **Search Keywords** control.
7. Type 2 as the **ColumnSpan** property value.
8. Click **Save** on the toolbar of the Layout Editor to save changes to the customization project.

As the screenshot below illustrates, the **Description (long)** and **Search Keywords** controls now span two columns.

The screenshot shows a software interface with a 'Relative Positioning Layout' tab selected. The interface contains several input fields and controls. A red box highlights the 'Description (long):' and 'Search Keywords:' fields, which are wider than the other fields. Other fields include 'Default Ship Fee' (0.00), 'Last Cost' (0.000000), 'Extra Ship Fee' (0.00), 'Default Postal...', 'Title', 'FirstName', and 'Last Name'. There are also checkboxes for 'Use On Entry', 'Active', and 'Featured', and a 'VIEW ON MAP' button.

Figure: Viewing the widened controls

Merge Property

Merging means placing controls so that they are horizontally aligned.

You should align controls horizontally by merging the **Extra Ship Fee** input control with the **Use On Entry** check box, and the **Default Postal Code** input control with the **View On Map** button.

To do this, you set the **Merge** property value to *True* for the corresponding `PXLayoutRule` components placed above the **Extra Ship Fee** and **Use On Entry** subnodes.



: Horizontal alignment is performed for the controls that are placed between this and any other subsequent `PXLayoutRule` component; the first `PXLayoutRule` component discovered breaks the merging. To cancel merging for all of the following controls, you must add the `PXLayoutRule` component without the adjusted property value

For all `PXLayoutRule` components that have the **Merge** property value set to *True*:

- The **ColumnWidth** property value is never applied to the controls if the **Merge** property is set to *True* for the same `PSLayoutRule` component.
- The values for the **ControlSize** and **LabelsWidth** properties are inherited by default from the previously declared `PXLayoutRule` component with the **StartRow** or **StartColumn** property values set to *True*. You can override these property values if necessary by specifying the **ControlSize** and **LabelsWidth** property values from the predefined list of options. (See [Using Predefined Size Values](#) for details.)

To merge controls, you should perform the following actions (see also the screenshot below):

1. Select the **Row** node that contains the controls.
2. Select the **Properties** tab of the Layout Editor.
3. Set the **Merge** property value to *True*, as shown in the screenshot below.

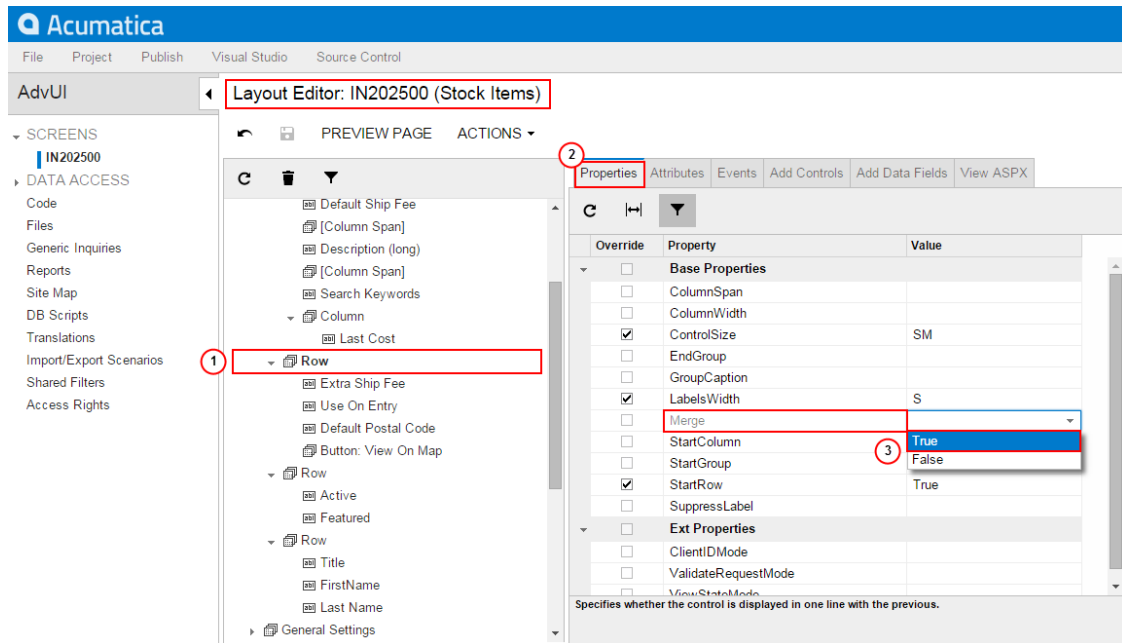


Figure: Setting the Merge property for the PXLayoutRule component

4. Select the **Add Controls** tab of the Layout Editor.
5. Drag and drop the **Merge** layout rule (PXLayoutRule with the predefined **Merge** property) above the **Default Postal Code** nodes in the tree, as shown in the screenshot below.

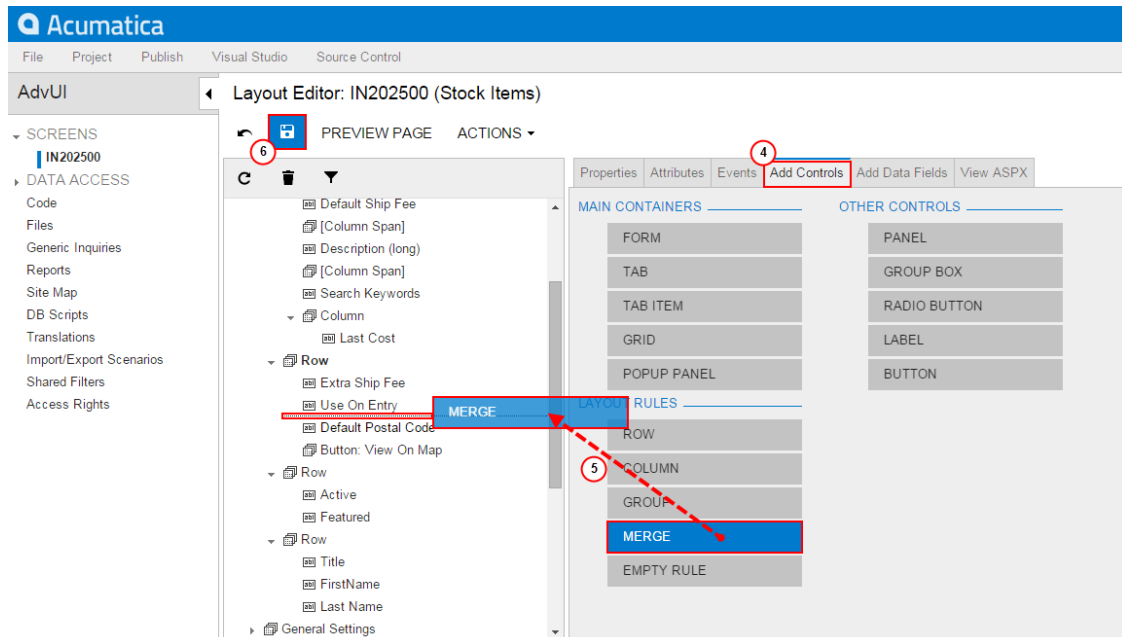


Figure: Adding the Merge PXLayoutRule component

6. Click **Save** on the toolbar of the Layout Editor to save changes to the customization project.



: Because of the previously added PXLayoutRule component (with the **StartRow** property value set to **True**) above the **Active** subnode, you do not need to add a PXLayoutRule component below the last merged controls.

As the screenshot below illustrates, the two pairs of controls are horizontally aligned now.

The screenshot displays a software interface with a tabbed menu at the top. The 'Relative Positioning Layout' tab is selected and highlighted with a red box. Below the tabs, there are several input fields and checkboxes. The 'Extra Ship Fee' field is highlighted with a red box and contains the value '0.00'. To its right is a checkbox labeled 'Use On Entry'. Other visible fields include 'Default Ship F...' with '0.00', 'Last Cost' with '0.000000', 'Description (lo...', 'Search Keywo...', 'Default Postal...' with a 'VIEW ON MAP' button, 'Active' and 'Featured' checkboxes, and text input fields for 'Title', 'FirstName', and 'Last Name'.

Figure: Viewing the horizontally aligned controls

GroupCaption and EndGroup Properties

Now you will create a group with a caption and include the **Active** and **Featured** check boxes in this group. Organizing controls on a form within groups makes users' work more logical.

By specifying the **GroupCaption** property value for the corresponding `PXLayoutRule` components placed above the first subnode, you create the group of controls and set up the header for the group. You should also add a `PXLayoutRule` component with the **EndGroup** property value set to `True` below the last control that is included in the group.

For all `PXLayoutRule` components with the **GroupCaption** property value specified:

- The **ColumnWidth** property value is never applied to the controls if the **GroupCaption** or **EndGroup** property is set to `True` for the same `PSLayoutRule` component.
- The values for the **ControlSize** and **LabelsWidth** properties are inherited by default from the previously declared `PXLayoutRule` component with the **StartRow** or **StartColumn** property values set to `True`. You can override these property values if necessary by specifying the **ControlSize** and **LabelsWidth** property values from the predefined list of options. (See [Using Predefined Size Values](#) for details.)

To create the group of controls, you should perform the following actions:

1. Select the **Row** node that contains the **Active** and **Featured** controls.
2. Select the **Properties** tab of the Layout Editor.
3. Type `Status` as the **GroupCaption** property value and set the **StartGroup** property value to `True`, as shown in the screenshot below.

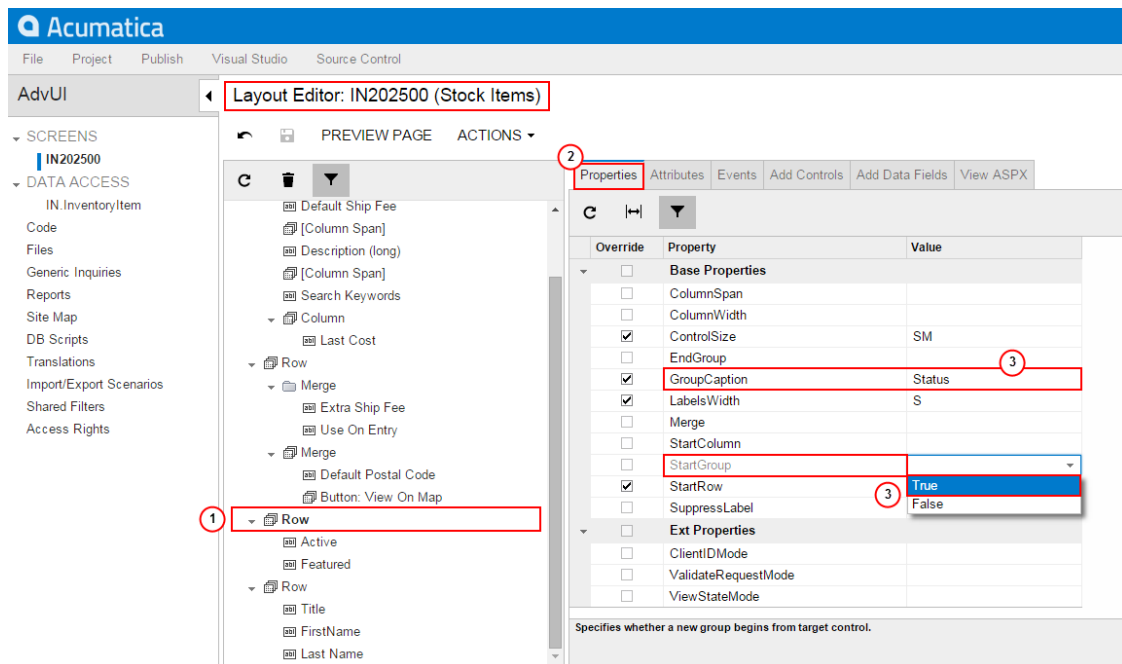


Figure: Setting the StartGroup property for the PXLAYOUTRULE component

4. Select the **Add Controls** tab of the Layout Editor.
5. Drag and drop the **Empty Rule** layout rule under the **Active** node in the tree, as shown in the screenshot below.

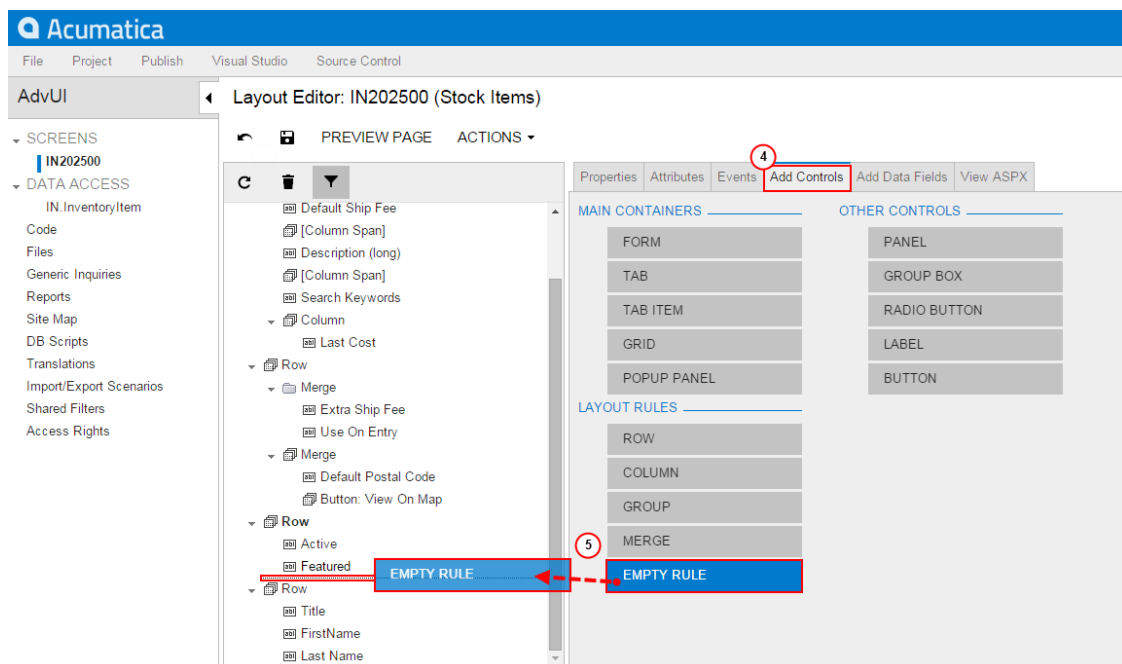


Figure: Adding the Empty Rule PXLAYOUTRULE components

6. Select the last created **Layout Rule** node.
7. Select the **Properties** tab of the Layout Editor.
8. Set the **EndGroup** property value to *True*, as shown in the screenshot below.

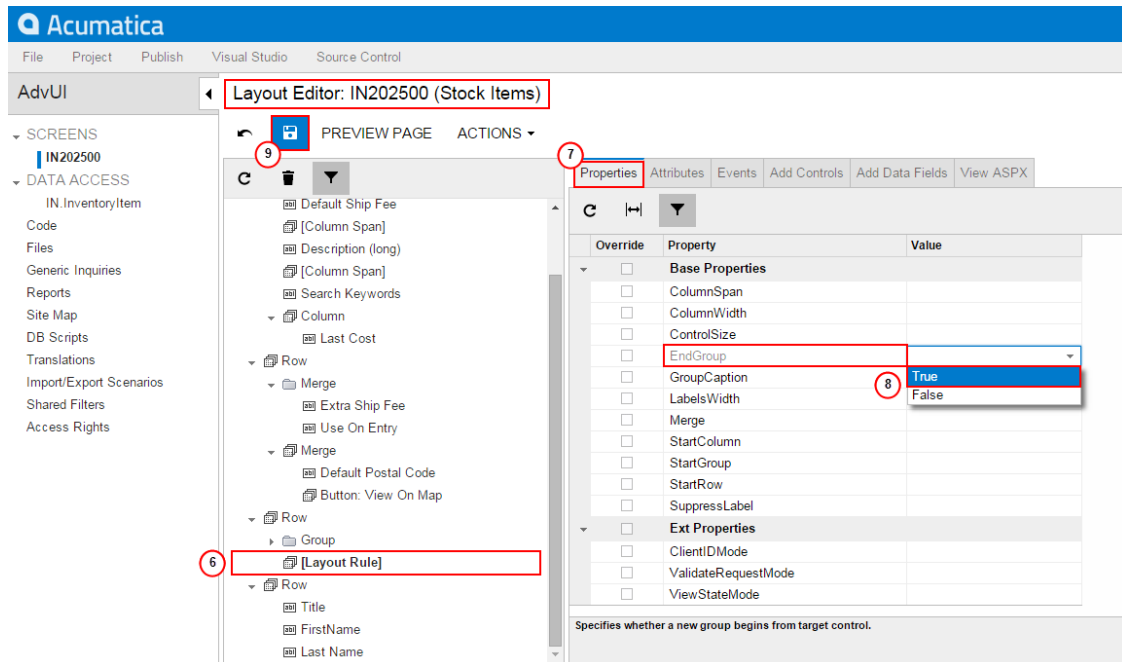


Figure: Adjusting the EndGroup property for the PXLayouRule component

9. Click **Save** on the toolbar of the Layout Editor to save changes to the customization project.

As the screenshot below illustrates, the **Active** and **Featured** check boxes are now organized in the added **Status** group.

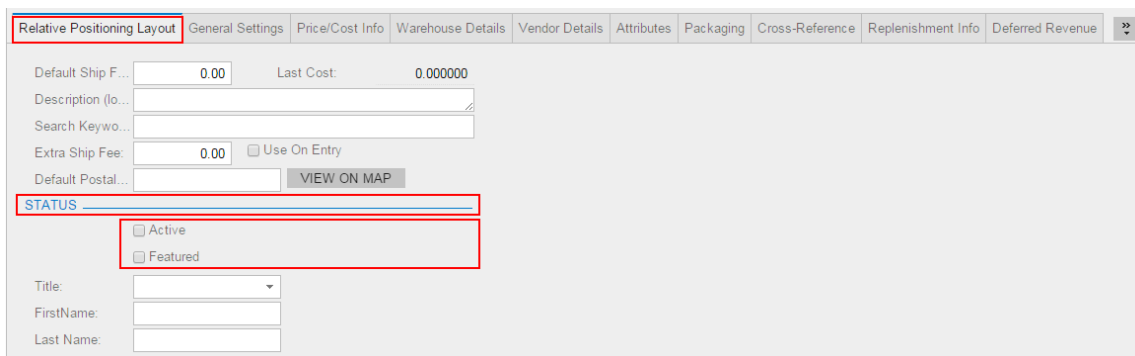


Figure: Viewing the group of controls

SupressLabel Property

Every control contains both a label and a user input control, except for buttons and check boxes. A check box has a label, which is displayed right of it. When you add a check box onto a form, it is automatically aligned toward other input controls within the appropriate column. As a result, the area left of a check box is empty.

To hide control labels placed within a column, you should set the **SupressLabel** property value to *True* of the `PXLayouRule` subnode. All control labels placed within the column are hidden and check boxes are placed without any space to the left of the input control. If needed, you can place any separate check box to the left by setting the **AlignLeft** property value to *True*, or you can set the **SupressLabel** property value to *True* for any other control to hide its label.

In this example, you use the **SupressLabel** property to hide the empty area to the left of the check boxes included in the **Status** group.



: The **SuppressLabel** property affects all controls of the group that are placed under the `PXLayoutRule` component with the `True` value of this property. The **SuppressLabel** property value must be defined for every `PXLayoutRule` component for the controls placed beneath the component and included in the same column; this property is never inherited from the previously declared property. The **SuppressLabel** property value is never applied to `PXLayoutRule` components with the **ColumnSpan** property value specified.

To suppress labels for the check boxes within the same column, you should perform the following actions (shown in the following screenshot):

1. Select the **Row** node with specified the **StartGroup** property.
2. Select the **Properties** tab of the Layout Editor.
3. Set the **SuppressLabel** property value to `True`, as shown in the screenshot below.
4. Click **Save** on the toolbar of the Layout Editor to save changes to the customization project.

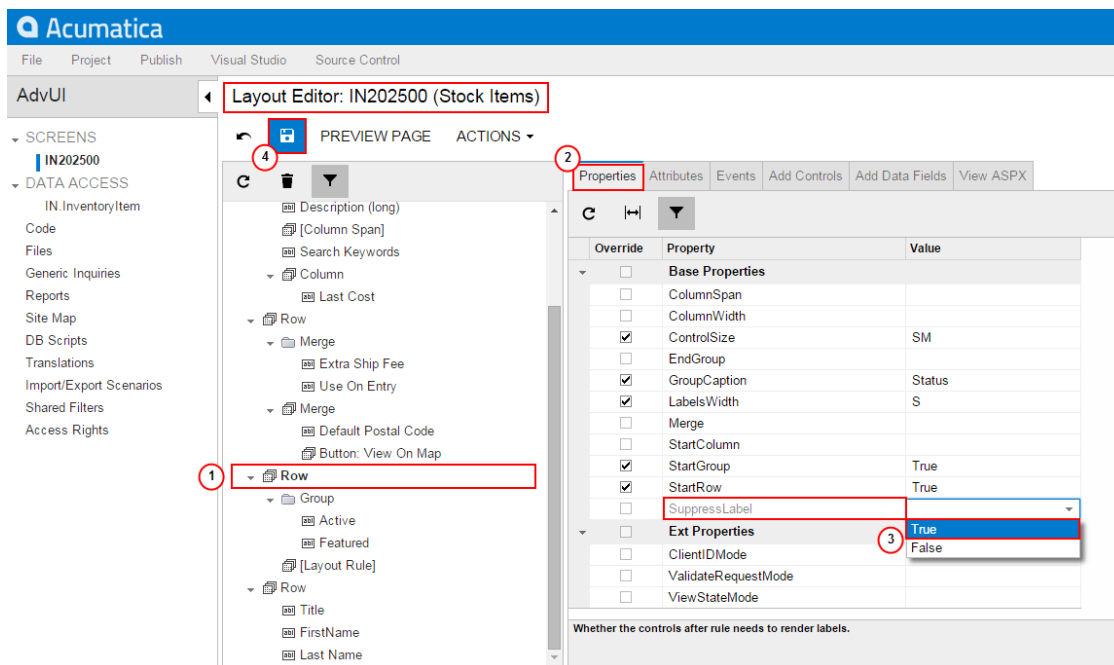


Figure: Setting the SuppressLabel property for the Status group of check boxes

As the screenshot below illustrates, the **Active** and **Featured** check boxes have been shifted to the left.

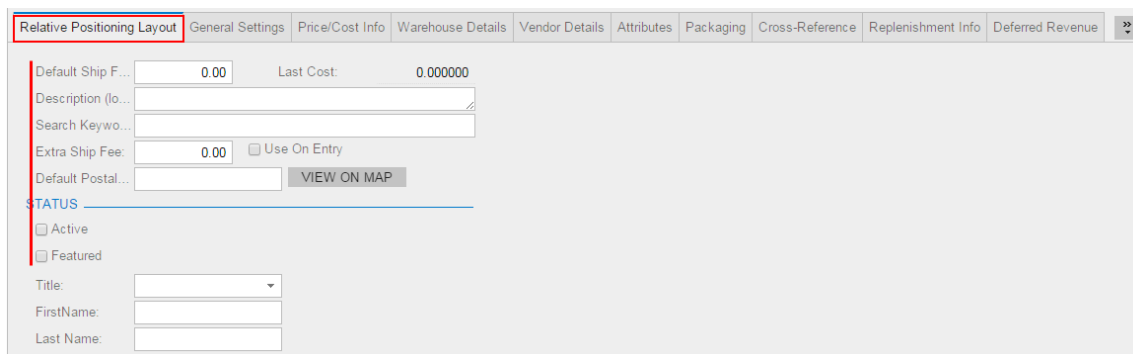


Figure: Viewing the shifted check boxes

Nested Container Controls

Next, you need to add a panel (the **PXPanel** control) onto the **Relative Positioning Layout** tab, add a few controls onto the panel, place them into two columns, and hide the empty area to the left of check boxes.

After you add the **PXPanel** control, the system creates a node. All the aforementioned layout rules apply to this node, because it also consists of the separate set of rows and columns. Hence, every nested container control is arranged toward the parent node layout but contains the separate layout of subnodes within.

To perform this complex step, you would perform the following actions:

1. Select the **Add Controls** tab of the Layout Editor.
2. On the opened tab item, drag and drop the **Panel (PXPanel)** control to the position under the Layout Rule node which is the end of the group, as shown in the screenshot below.

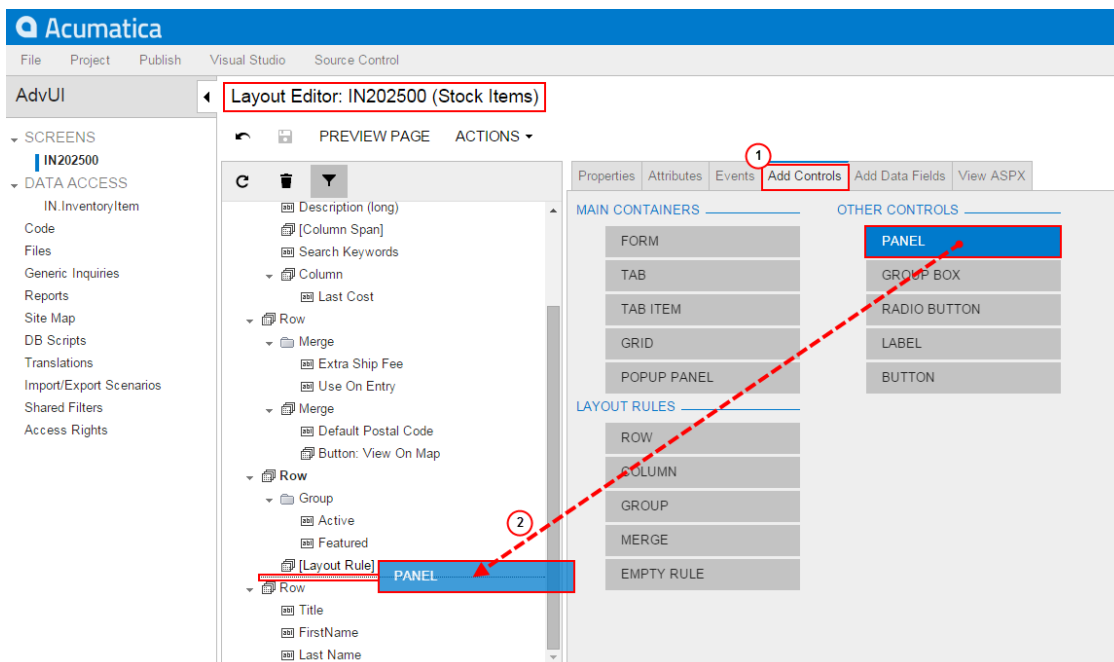


Figure: Adding the PXPanel control

3. Select the created *PXPanel* node in the tree.
4. Select the **Properties** tab of the Layout Editor.
5. Select the **Caption** property and enter `Pending Values` to specify the name of the new panel.
6. Open the **Add Data Fields** tab of the Layout Editor.
7. Select **Visible** filter for the list to display the visible fields of the `Inventory Item` data access class.
8. In the list, select the **PendingStdCost** and **PendingStdCostDate** fields.
9. Click **Create Controls** on the tab toolbar (see the screenshot below).

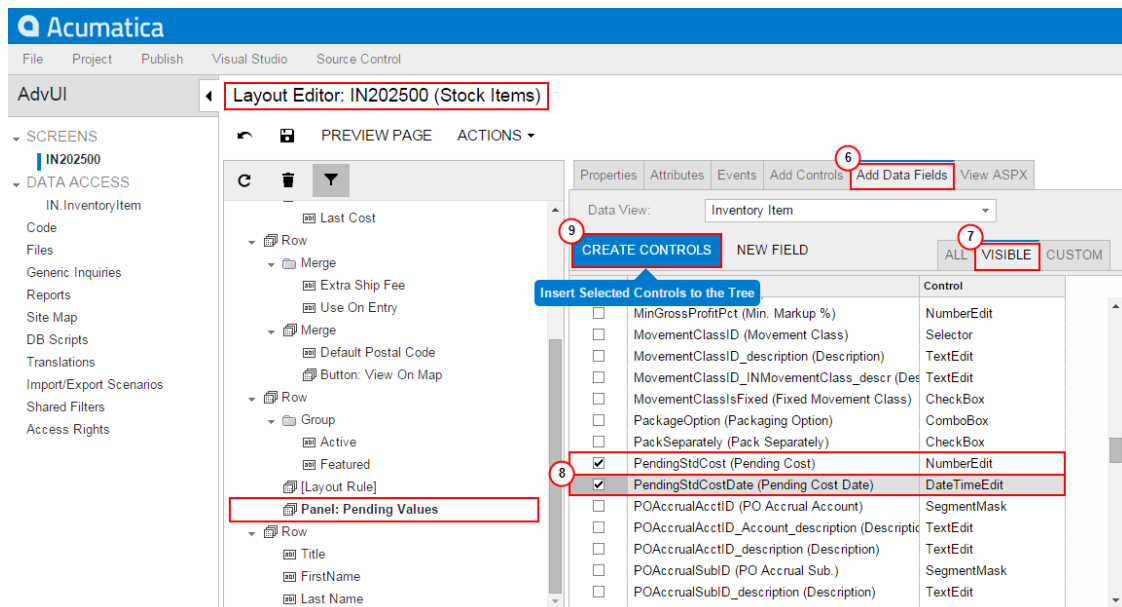


Figure: Adding the fields onto the PXPanel container control

10. Select **Custom** filter for the list to display the custom fields of the `Inventory Item` data access class.

11. In the list, select the **UsrUseOnEntryPndPrice** and **UsrUseOnEntryPndCost** fields.

12. Click **Save** on the toolbar of the Layout Editor to save changes to the customization project.

After adding a column layout rule for the **PXPanel** control, placing two new column layout rules onto the panel and specifying properties for these rules, you can view the final result of the UI customization, as the following screenshot illustrates.

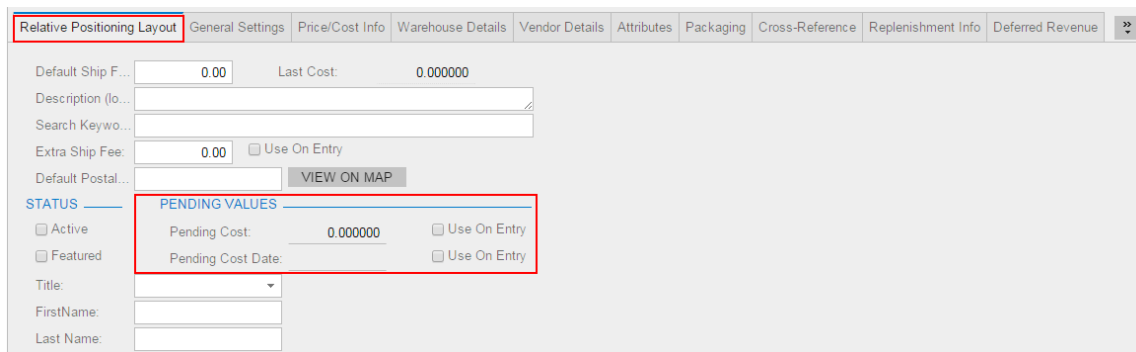


Figure: Analyzing the final view of the form

The Results of UI Customization that Uses the `PXLayoutRule` Component

After modifying the UI and publishing the customization project, you should analyze the results of the UI customization by doing the following:

- Explore the changeset of the project
- Analyze the new layout of the form at run time and check the final positioning of the controls

To view the fragment of the `.aspx` code that represents the customization result, open the form in the Layout Editor and select the **View ASPX** tab. Then select a node on the tree of controls to display

the code fragment for the node. All the changes are highlighted in yellow, as the screenshot below illustrates.

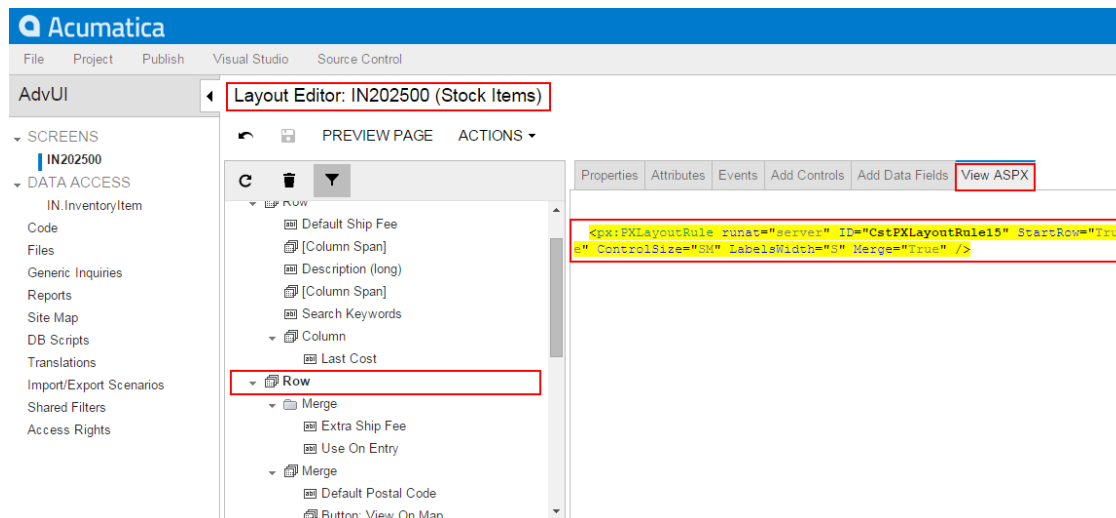


Figure: Exploring the .aspx changeset

If you need to analyze the added content (changeset) of the customization project, open the project in the Project Editor, choose the **Edit Project Items** command on the **File** menu item, and select the **Object Name** field of the customized object (see the screenshot below). The `Page` item represents the changes to the .aspx code.

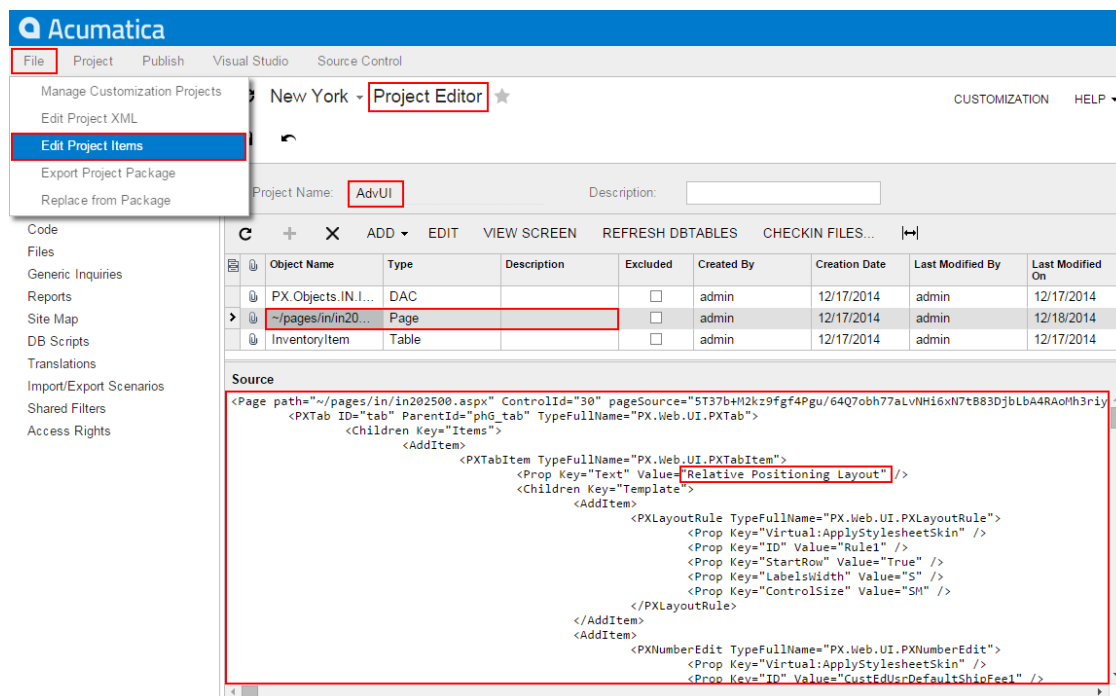


Figure: Exploring the content of the Page item of the customization project

After publishing the customization project, open the **Relative Positioning Layout** tab of the *Stock Items* form to analyze and test the new structure of this tab at run time (see the screenshot below).

The screenshot displays the Acumatica user interface. At the top, the navigation bar includes 'Acumatica', 'ORGANIZATION', 'FINANCE', 'DISTRIBUTION' (highlighted), 'CONFIGURATION', 'SYSTEM', '13 (10)', '12/18/2014 12:42 PM', and 'ADMIN'. Below this, a secondary navigation bar shows 'Inventory', 'Sales Orders', 'Purchase Orders', and 'Purchase Requisitions'. The main content area is titled 'Inventory' and shows 'New York - Stock Items' with a search bar and navigation icons. A left-hand menu lists various inventory-related actions like 'ENTER', 'MANAGE', and 'EXPLORE'. The 'Stock Items' menu item is highlighted in red. The main form area has several tabs: 'Relative Positioning Layout' (highlighted in red), 'General Settings', 'Price/Cost Info', 'Warehouse Details', 'Vendor Details', 'Attributes', 'Packaging', and 'Cross-Reference'. The form contains fields for 'Inventory ID', 'Item Status' (set to 'Active'), 'Description', 'Default Ship Fee' (0.00), 'Last Cost' (0.000000), 'Search Keyword', 'Extra Ship Fee' (0.00), 'Default Postal Code', and a 'VIEW ON MAP' button. Below these are sections for 'STATUS' and 'PENDING VALUES' with checkboxes for 'Active', 'Featured', and 'Use On Entry'.

Figure: Analyzing the final view of the Relative Positioning Layout tab

Examples of Functional Customization

Functional customization means modifications of the data structure and original application business logic. All these modifications are related to the code of the appropriate data access classes (DACs) or business logic controllers (BLCs, also referred to as *graphs*).

The Acumatica Customization Platform provides the tools and the framework that is based on class extensions and provides the following key features:

- You can deploy multiple projects that extend DACs or BLCs.
- Extensions are precompiled, which provides a measure of protection for your source code and intellectual property.
- The framework provides an advanced level of control over the business logic and a multilevel extension model.
- The platform implements an auto-discovery mechanism, which makes the deployment and upgrade processes straightforward.

The topics of this section describe the rules and methods of functional customization and give examples of customization projects that resolve typical functional customization tasks:

- [Adding Data Fields](#)
- [Customizing DAC Attributes](#)
- [Modifying a BLC Action](#)
- [Modifying a BLC Data View](#)
- [Declaring or Altering a BLC Data View Delegate](#)
- [Extending BLC Initialization](#)
- [Altering the BLC of a Processing Form](#)
- [Adding or Altering BLC Event Handlers](#)
- [Altering BLC Virtual Methods](#)

Adding Data Fields

You can add to a form a UI control based on a new data field. To do this, you can use the following ways:

- Use the Data Class Editor to create the data field and use the Layout Editor to add the control onto the required area of a form.
- Define the data field in code, add the Table customization item for adding the column to the database table if the field is bound, and use the Layout Editor to add the control onto the required area of a form.

These approaches are described in greater detail below.

- [Adding a Data Field From the Data Class Editor](#)
- [Adding a Data Field From Code](#)

Adding a Data Field From the Data Class Editor

Suppose that you have to add an input UI text box for a bound data field to the [Stock Items](#) (IN.20.25.00) form (**Distribution > Inventory > Work Area > Manage**). To do this, you have to (see the diagram below):

- Add a data field to the code of the appropriate DAC (the functional customization step)
- Add a column to the database table (a customization change to the database structure that is a part of functional customization)



: You have to add a database column only for bound data fields. Bound data field means the field values are saved in the database. If you define an unbound field, skip this step.

- Add a control for the field onto the form area (the UI customization step)

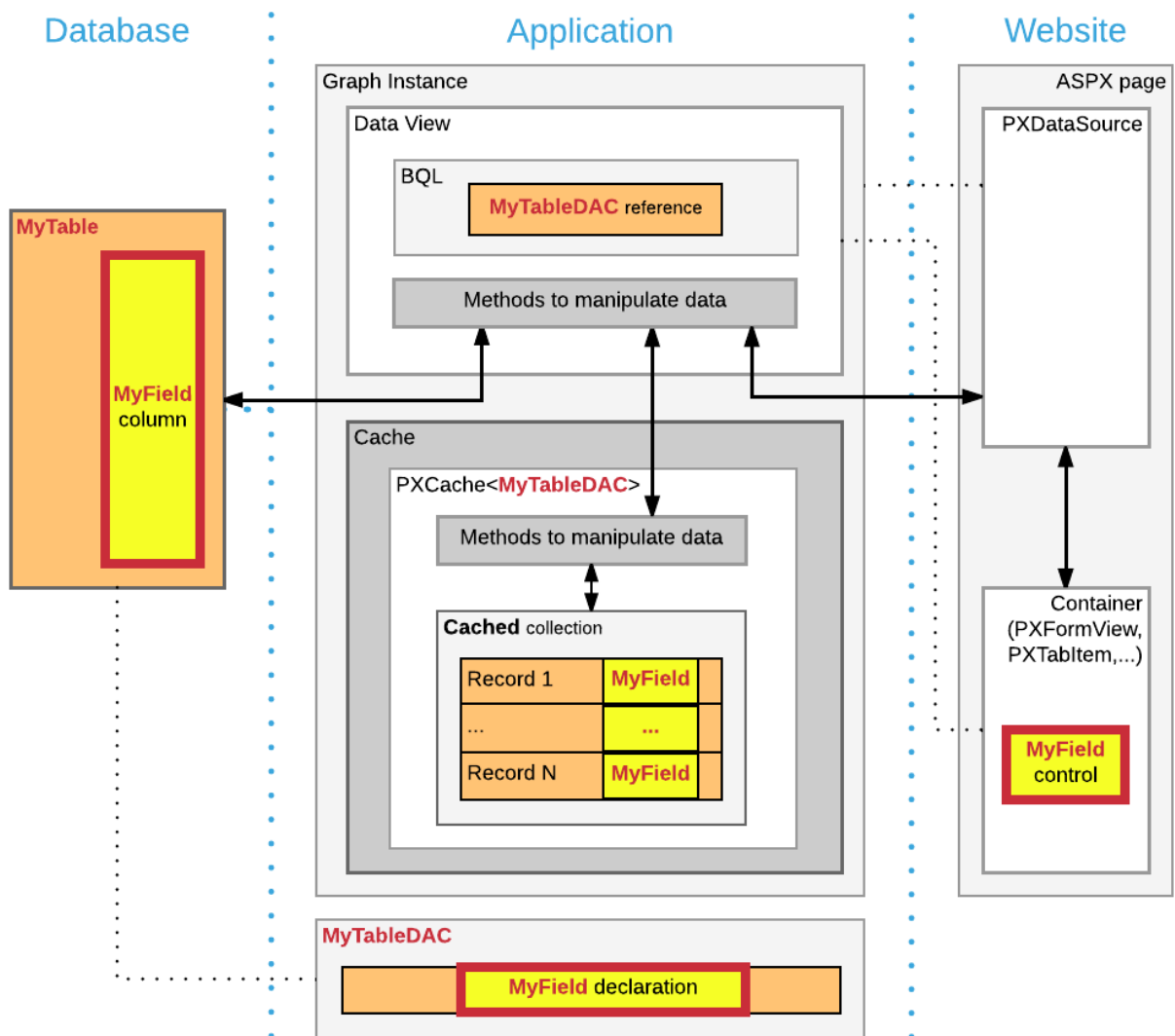


Figure: Adding a bound field to a webpage

Suppose you have to add the new **Search Keywords** text box to the *Stock Items* form, as the screenshot below shows.

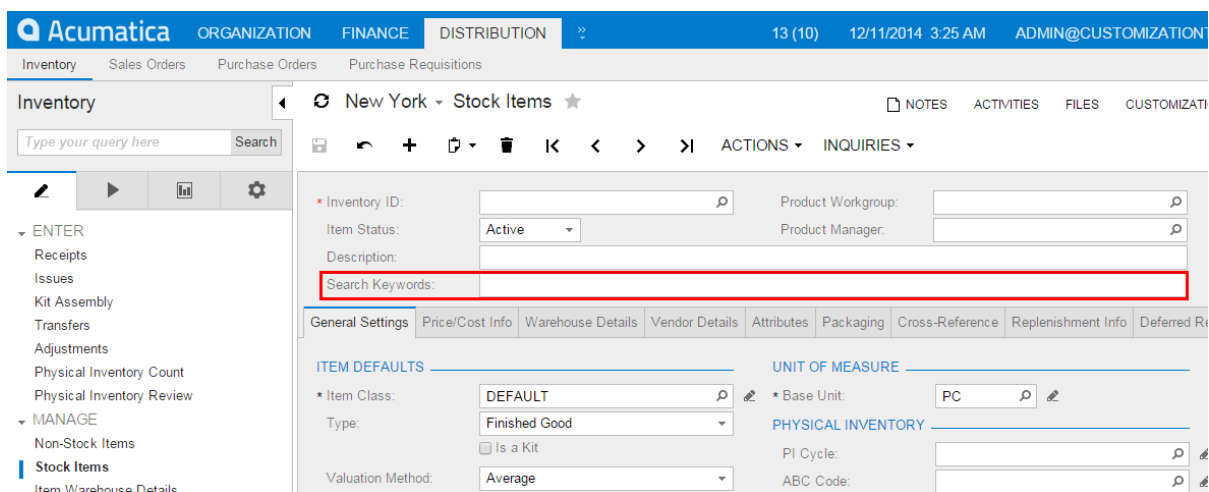


Figure: The customization task of adding the new field to the form

To open the data access class that provides fields for the form area on the *Stock Items* form in Data Class Editor, open the form, select **Customization > Inspect Element** on the form title bar, and click the form area of the *Stock Items* form.

The following information of the control and the data access class should appear in the **Element Properties** dialog box:

- **Control Type:** *Form View*
- **Data Class:** *InventoryItem*
- **Business Logic:** *InventoryItemMaint*

InventoryItem is the data access class that provides the data fields for UI controls on the form. Therefore, you have to add the new data field to the extension of this class.

In the **Element Properties** dialog box, select **Actions > Customize Data Fields** to open the Data Class Editor for the *InventoryItem* class (see the screenshot below).



: If you haven't selected the current project yet, you will be asked to select the customization project to which the customization of the data access class will be added. You can select the current project to automatically add all customizations that you initiate from the Element Inspector by using the **Customization > Select Project** menu to the project. For more information, see [Select Customization Project Dialog Box](#).

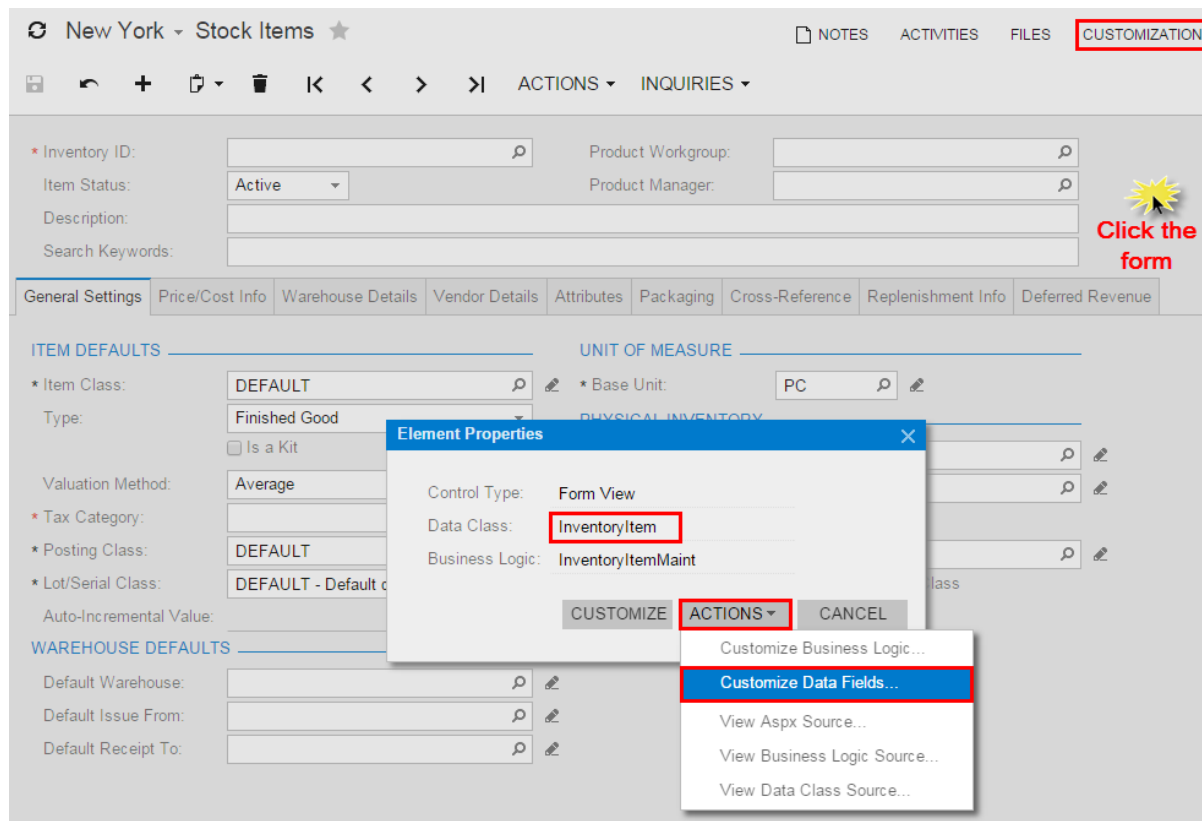


Figure: Selecting the data access class for customization

In the Data Class Editor, select **Add Field > Create New Field** on the toolbar.

In the **Create New Field** dialog box, specify the following parameters for the new field:

- **Field Name:** *SearchKeywords*
- **DisplayName:** *Search Keywords*

- **Storage Type:** *DBTableColumn*
- **Data Type:** *string*
- **Length:** 255

As soon as you move the focus out of the **Field Name** box, the system adds the *Usr* prefix to the field name (see the screenshot below), which provides the distinction between the base fields and new custom fields that you add to the class. Keep the prefix in the field name and click **OK** to add the data field to the class.



: If you select the *DBTableColumn* storage type, the system automatically adds schema for the new database column to the customization project. For more information on parameters that you specify for new fields, see [Create New Field Dialog Box](#).

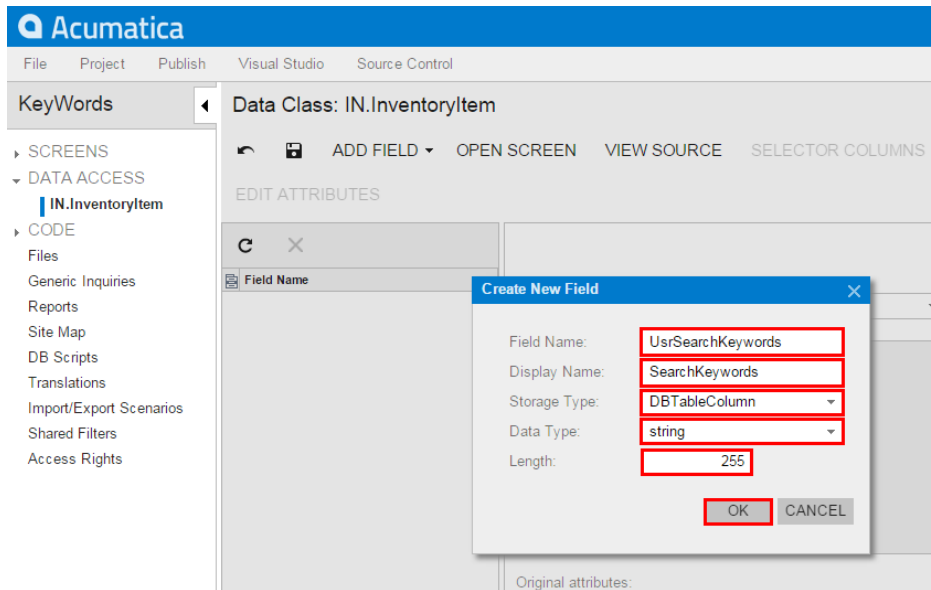


Figure: Adding the new data field to the class

To the customization project, the system adds the new field as the extension to the *InventoryItem* data access class, and adds the *Table* item for adding the corresponding column to the database table. Click **Save** on the toolbar of the Data Class Editor to save the changes to the project.



: To view the customization item generated for the database schema, select **File > Edit Project Items** and find the *Table* item for the *InventoryItem* database table in the list of project items.

To be able to add the UI control for the new field, you have to publish the customization project to make the system compile the customization code and update the database schema with the new column. To publish the customization project from Project Editor, select **Publish > Publish Current Project** and click **Publish** in the **Compilation** window. For more information, see [Publishing Customization Projects](#).

After the publication is complete, in Project Editor add the *Stock Items* form to the customization project. To do this, you can use the Element Inspector or select **Screens** on the navigation pane of the Project Editor, and add the *Stock Items* form to the list by selecting **Add Screen > Customize Existing Screen** on the toolbar.

Open the form in Layout Editor. In the tree of controls, expand the **Form: Item** container and select and expand the first column within the container to set up the position to which the new control will be added.

To add the control, select the **Add Data Fields** tab of Layout Editor. Leave the **Data View** box empty. (The box specifies the data access class by which the fields are filtered in the table below.)

Select the **Custom** filter on the tab and select the check box for the *UsrSearchKeywords* field in the table, as the following screenshot shows.

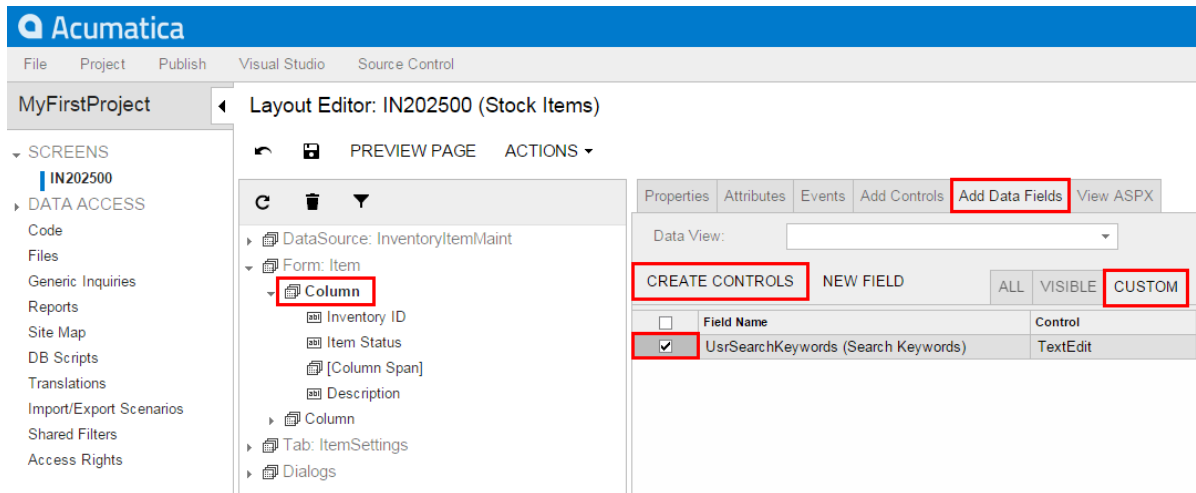


Figure: Adding the control for the data field

Click **Create Controls** on the toolbar of the tab. The control for the **Search Keywords** field will be added to the tree.

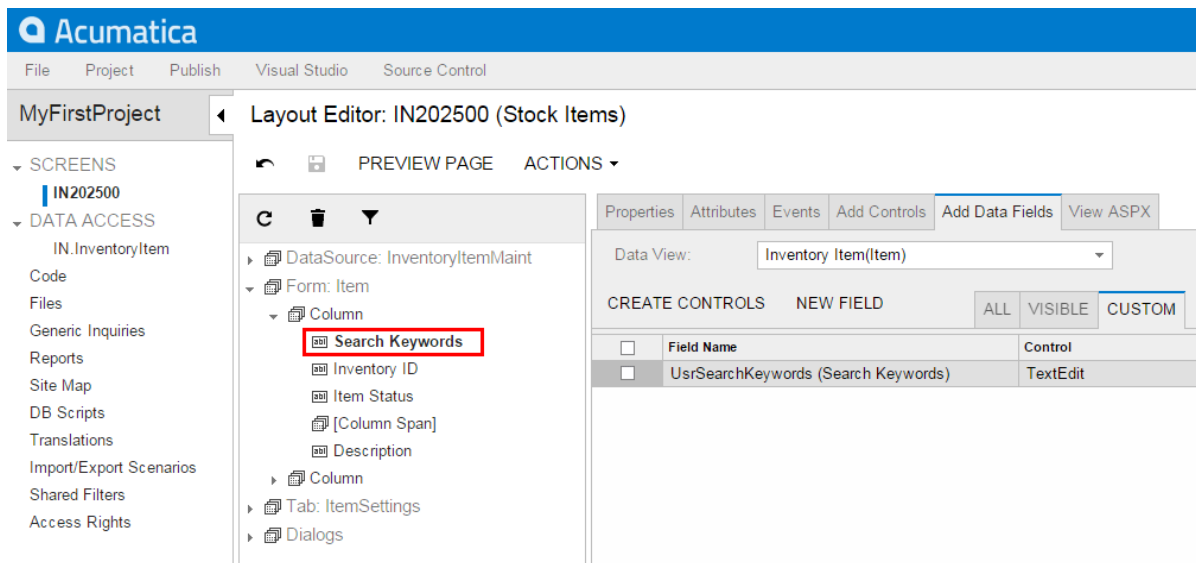


Figure: The control added to the tree

Now you have to adjust the position and size of the control on the form.

Drag and drop the **Search Keywords** control at the end of the column after the **Description** control and click **Save** to save the changes to the project.

To adjust the size of the control, you have to place an instance of the `PXLayoutRule` component right above the control.

Select the **Add Controls** tab of Layout Editor and drag and drop an **Empty Rule** component to the position above the **Search Keywords** control in the tree, as the screenshot below shows.

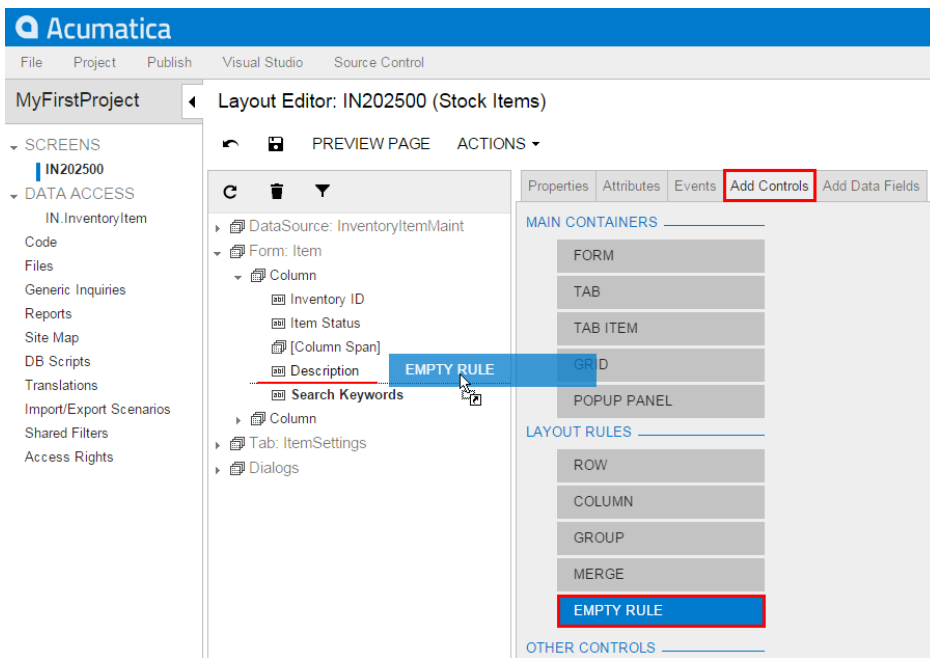


Figure: Adding an Empty Rule component to the tree to adjust the size of the control

Select the **Layout Rule** component that has been added to the tree and switch to the **Properties** tab of Layout Editor.

On the **Properties** tab, set the **ColumnSpan** property to 2, which specifies the span of the control on two columns of the layout on the form. Click **Save** to save the changes to the project.



: The controls are organized in two columns on the form. For more information about positioning controls, see [Layout Rule \(PXLayoutRule\)](#).

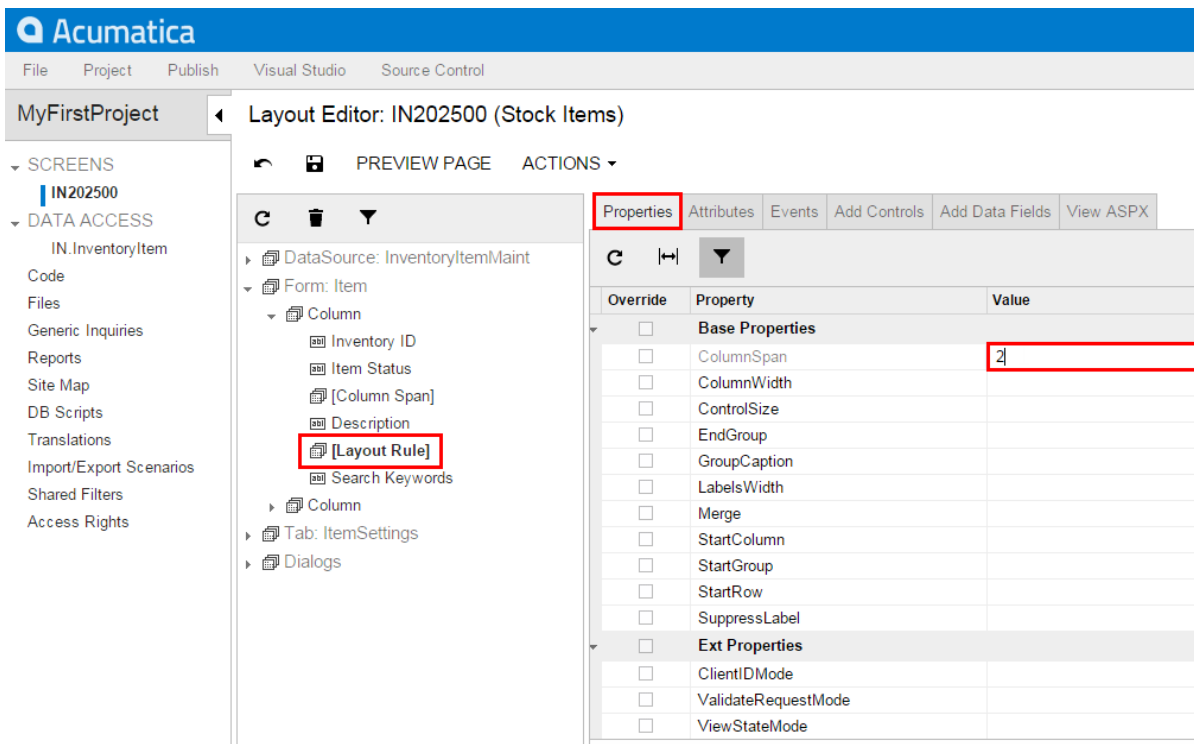


Figure: Specifying the span of the control for the number of columns in the layout

The customization to the layout of the form is ready.



: While working with Layout Editor, you can click **Preview Page** to view how the changes will affect the layout of the form.

Publish the project again to view the changes applied to the *Stock Items* (IN.20.25.00) form (**Distribution > Inventory > Work Area > Manage**). After you publish the project, verify that the customized Stock Items form looks as needed according to the customization task (see above). The customization items for form layout, data access class, and database schema have been added to the project. The new **Search Keywords** text box will appear on the *Stock Items* form on every system where you publish the customization project.

The DAC extension code that has been generated for the new field is given below. The system has generated a DAC extension class for `InventoryItem` and added the data field definition to the class. For more information about extension classes, see [DAC Extensions](#).

```
public class PX_Objects_IN_InventoryItem_Extension_AddColumn:
    PXCacheExtension<PX.Objects.IN.InventoryItem>
{
    #region UsrSearchKeywords
    public abstract class usrSearchKeywords : IBqlField{}
    [PXDBString(255)]
    [PXUIField(DisplayName="Search Keywords")]
    public virtual string UsrSearchKeywords
    {
        get; set;
    }
    #endregion
}
```

You can open the customization code in MS Visual Studio and work with the code there. For more information, see [Integrating the Project Editor with Microsoft Visual Studio](#).

Adding a Data Field From Code

If you work with the customization code in MS Visual Studio, you can define a new data field in DAC extension code and then create the control on the form.

Suppose that you have to add an input UI text box for a bound data field to the *Stock Items* (IN.20.25.00) form (**Distribution > Inventory > Work Area > Manage**). To do this, you have to:

- Add a data field to the code of the appropriate DAC (the functional customization step)
- Add a column to the database table (a customization change to the database structure that is a part of functional customization)



: You have to add a database column only for bound data fields. Bound data field means the field values are saved in the database. If you define an unbound field, skip this step.

- Add an input UI text box onto the form area of the form (the UI customization step)

Suppose you have to add the new **Local Tax Category** selector to the *Stock Items* form, as the screenshot below shows.

The screenshot shows the Acumatica Inventory Item form for 'New York Stock Items'. The 'Local Tax Category' field is highlighted with a red box. A 'Select - Local Tax Category' dialog box is open, showing a list of tax categories. The dialog box has a search bar and a table with the following data:

Tax Category ID	Description
EXEMPT	Exempt Tax Category
INCLUSIVE	inclusive
INCLUSIVE2	INCLUSIVE2
INCLUSIVE3	INCLUSIVE3
PURCHCAT	Purch Tax Cat
TAXABLE	Taxable Goods and Services
WITHHOLDIN	Withholding1

Figure: The customization task of adding the new field to the form

If you want to work with the DAC extension code in the integrated MS Visual Studio solution without use of Data Class Editor, you can develop the code in a .cs file added to the customization project as a custom Code item or in a .cs file in a separate add-on project in the solution. We recommend that you use either Data Class Editor or develop the entire customization code in an add-on project in MS Visual Studio, but do not mix up the approaches. For more information, see [Integrating the Project Editor with Microsoft Visual Studio](#).

To add the UI control for the bound data field created from code, perform the following actions:

1. In MS Visual Studio, define the custom `usrLocalTaxCategoryID` data field in the DAC extension class, as listed below.

```
using PX.Data;
using PX.Objects.IN;
using PX.Objects.TX;

namespace PX.Objects.IN
{
    public class InventoryItemExtension: PXCACHEExtension<InventoryItem>
```

```

{
    #region UsrLocalTaxCategoryID
    public abstract class usrLocalTaxCategoryID : PX.Data.IBqlField
    {
    }
    [PXDBString(10, IsUnicode = true)]
    [PXUIField(DisplayName = "Local Tax Category")]
    [PXSelector(typeof(TaxCategory.taxCategoryID),
        DescriptionField = typeof(TaxCategory.descr))]
    public string UsrLocalTaxCategoryID { get; set; }
    #endregion
}

```



: In this example, the .cs file with the DAC extension code is added to the customization project as a custom Code item. When you develop extensions in custom Code items, define the extension classes in the namespace of the original class. For more information about adding custom Code items to the project, see [Code](#).

2. Update the customization project with the new customization code. To do this, open the customization project that corresponds to the MS Visual Studio solution in the Customization Project Editor, select the **Files** list of project items and click **Detect Modified Files**.
3. In the **Modified Files Detected** dialog box, select the conflicting file of the custom Code item and click **Update Customization Project** to update the code in the project.



: If you build an assembly with the DAC extension code, update the assembly file in the customization project. For more information, see [To Update a File Item in a Project](#).

4. Add the database schema for the custom field to the project. To do this, select **File > Edit Project Items** on the menu of Project Editor.
5. On the toolbar of the list of project items, select **Add > Database Table Field**.
6. In the **Add UsrField to Database Table** dialog box, specify the parameters for the new database column in the `InventoryItem` table and click **OK** to add the database schema to the project:
 - **Table Name:** `InventoryItem`
 - **Field Name:** `LocalTaxCategoryID` (without the `Usr` prefix)
 - **Field Type:** `DBString(nvarchar)`
 - **Length:** 10

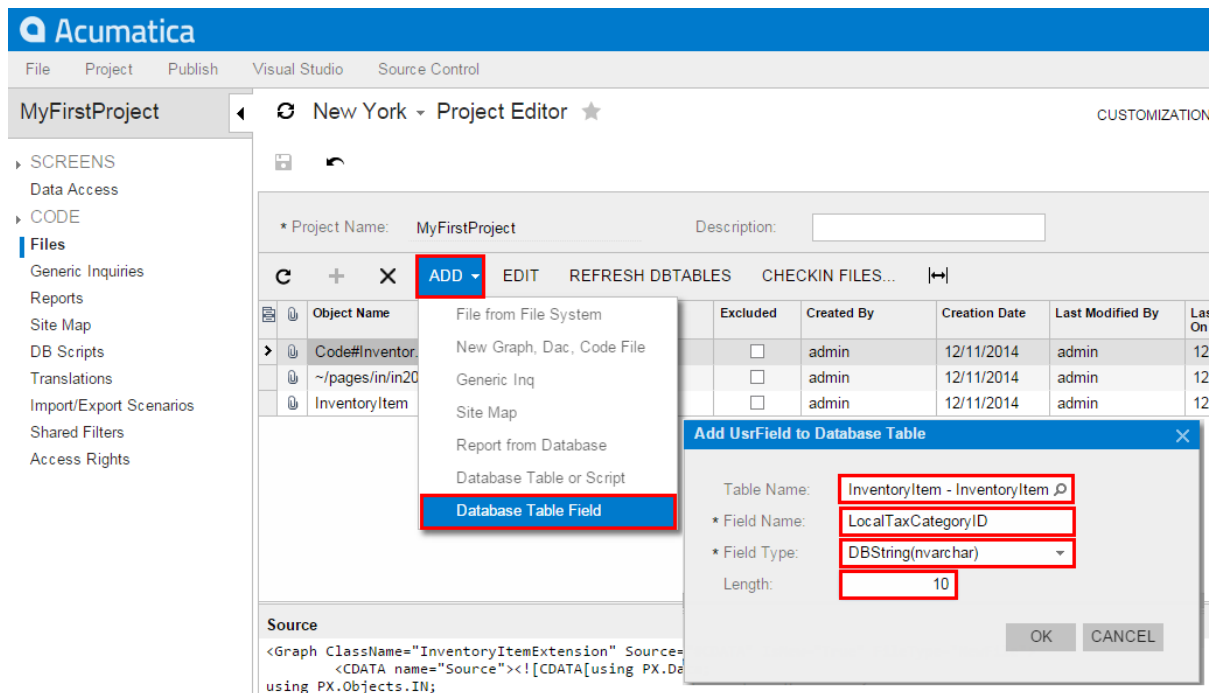


Figure: Adding the database schema for a custom DAC field created from code



: We recommend that you add the database schema for the a custom column, as shown above, and do not alter the original schema of the table by a custom SQL script.

7. Publish the customization project to make the system add the column to the database and compile the customization code. To do this, select **Publish > Publish Current Project** on the menu of Project Editor and publish the project.
8. Add the UI control for the custom field to the *Stock Items* form. To do this, open the form in Layout Editor.



: You can add an existing form to the customization project by using the Element Inspector or directly from Project Editor. See [Customized Screens](#) for details.

9. Select the **Tax Category** control in the tree to add the new control beneath it.
10. On the **Add Data Fields** tab, select the check box for the **Local Tax Category** selector in the table and click **Create Controls** to add the control to the tree (see the screenshot below).

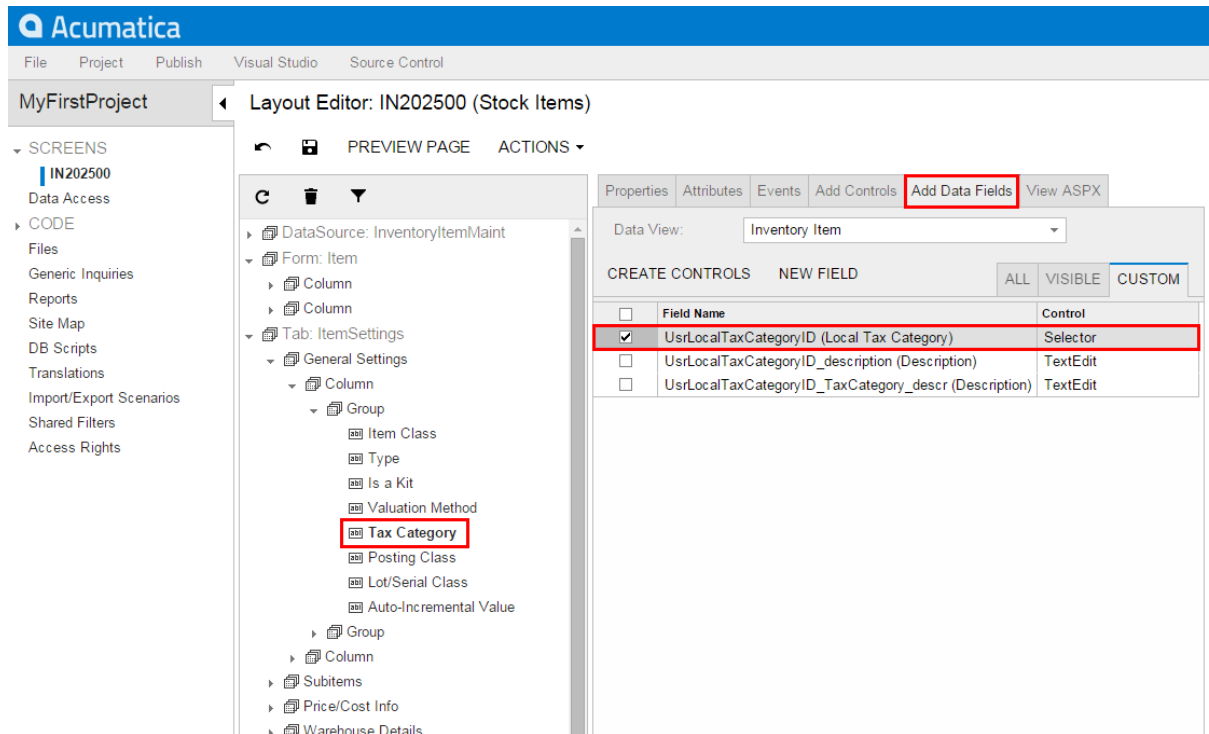


Figure: Adding the control for the data field

11. Click **Save** on the toolbar to save the layout changes.
12. Preview the customized form by clicking **Preview Page** on the toolbar or publish the customization project to view the changes applied to the *Stock Items* (IN.20.25.00) form (**Distribution > Inventory > Work Area > Manage**).

Verify that the customized *Stock Items* form looks as needed according to the customization task (see above). The customization items for the form layout, custom code (or external assembly file), and database schema have been added to the customization project. The new **Local Tax Category** text box will appear on the *Stock Items* form on every system where you publish the customization project.

Customizing DAC Attributes

To modify or extend the behavior that is defined within a data access class (DAC), you should append or override the field attributes declared within the DAC.

Suppose that you have a customization task to modify the *Invoices* form, and that this task consists of the following requirements:

1. The **Customer Order** data field on the form area (see the screenshot below) must have the *None* value when the user has not specified the reference number of the customer.
2. The values of the **Discount Percent** column must be limited to a range of -30 to 25 percent instead of the original range of -100 to 100 percent.

The screenshot shows the Acumatica 'Invoices' form for 'New York'. The form includes fields for Type (Invoice), Reference Nbr. (000656), Customer (SO00000003 - SO customer #003), Currency (USD), Status (Open), Terms (07D - 7 Days), Due Date (3/16/2009), Date (3/9/2009), and Post Period (02-2009). A 'Customer Order' field is highlighted with a red box. Below the form is a table with columns for Branch, Shipment Nbr., Order Type, Order Nbr., Inventory ID, Transaction Descr., Quantity, UOM, Unit Price, and Discount Percent. The 'Discount Percent' column is also highlighted with a red box.

Branch	Shipment Nbr.	Order Type	Order Nbr.	Inventory ID	Transaction Descr.	Quantity	UOM	Unit Price	Discount Percent	Dis Ar
MAIN	000005	SO	000006	SO000000...	SO #003	6.00	PC	20.000000	0.000000	
MAIN	000005	SO	000006	SO000000...	SO #103	3.00	PC	15.000000	0.000000	
MAIN	000005	SO	000008	SO000000...	SO #103	1.00	PC	16.000000	0.000000	
MAIN	000007	SO	000010	SO000000...	SO #003	2.00	PC	20.000000	0.000000	
MAIN	000007	SO	000012	SO000000...	SO #103	1.00	PC	18.000000	0.000000	
MAIN	000007	SO	000012	SO000000...	SO #103	1.00	PC	15.000000	0.000000	
MAIN	000010	SO	000015	SO000000...	SO #103	2.00	PC	15.000000	0.000000	
MAIN	000010	SO	000015	SO000000...	SO #003	1.00	PC	0.000000	0.000000	

Figure: Viewing the original form

To begin implementing the first requirement, inspect the properties of the **Customer Order** UI control. To do this, on the **Customization** menu, select **Inspect Element** and click the box or label of the control on the form. The system should retrieve the following information of the control that will display in the **Element Properties** dialog box:

- **Control Type:** *Text Edit*. The type of the UI control.
- **Data Class:** *ARInvoice*. The data access class that contains the inspected field.
- **Data Field:** *InvoiceNbr*. The field that is represented by the **Customer Order** control on the form.
- **Business Logic:** *SOInvoiceEntry*. The business logic controller (BLC, also referred to as *graph*) that provides the logic executed on the *Invoices* form.

To modify attributes of the DAC field, select **Actions > Customize Data Fields** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the data field and click **OK**.

Data Class Editor opens for customization of the *InvoiceNbr* field of the *ARInvoice* data access class, as the screenshot below shows.

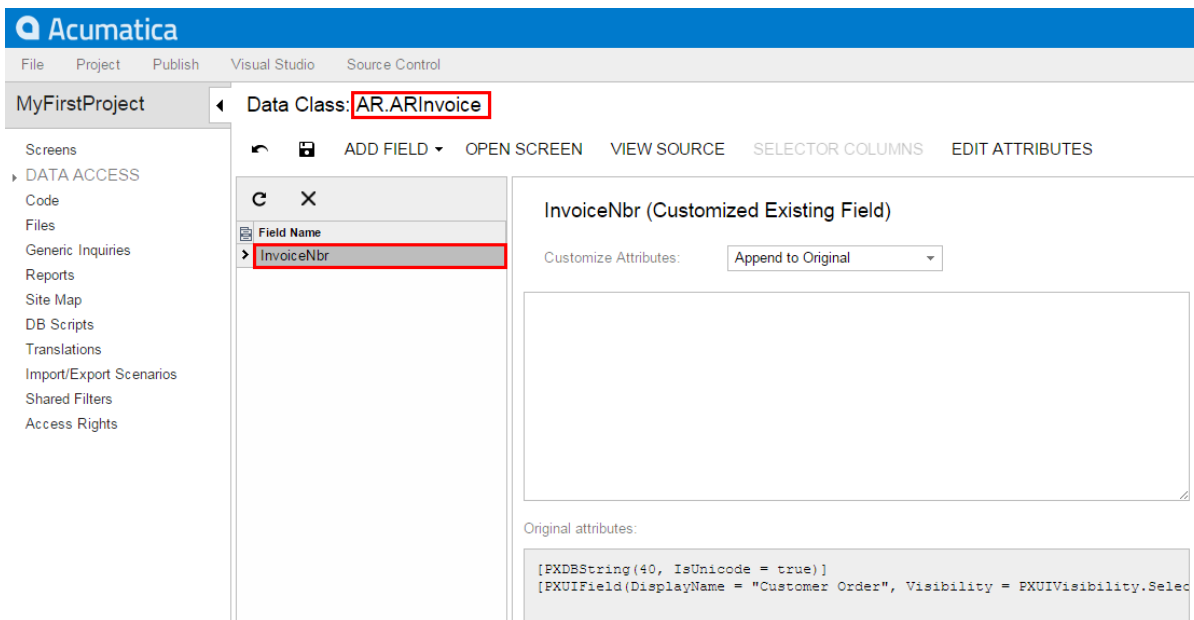


Figure: The data field added to the customization project

Suppose that you decided to append an attribute to the DAC field to specify the default value for the field. The attribute that you have to add to the field is `PXDefault`.



: Acumatica Framework 6.1 provides advanced possibilities to control the field customization by using additional attributes in the DAC extension. See the [Customization of Field Attributes in DAC Extensions](#) for details.

To append the attribute to the list of attributes of the data field, in the **Customize Attributes** box, select *Append to Original*. In the text area, type `[PXDefault("None")]` and save the changes in Layout Editor (see the screenshot below).

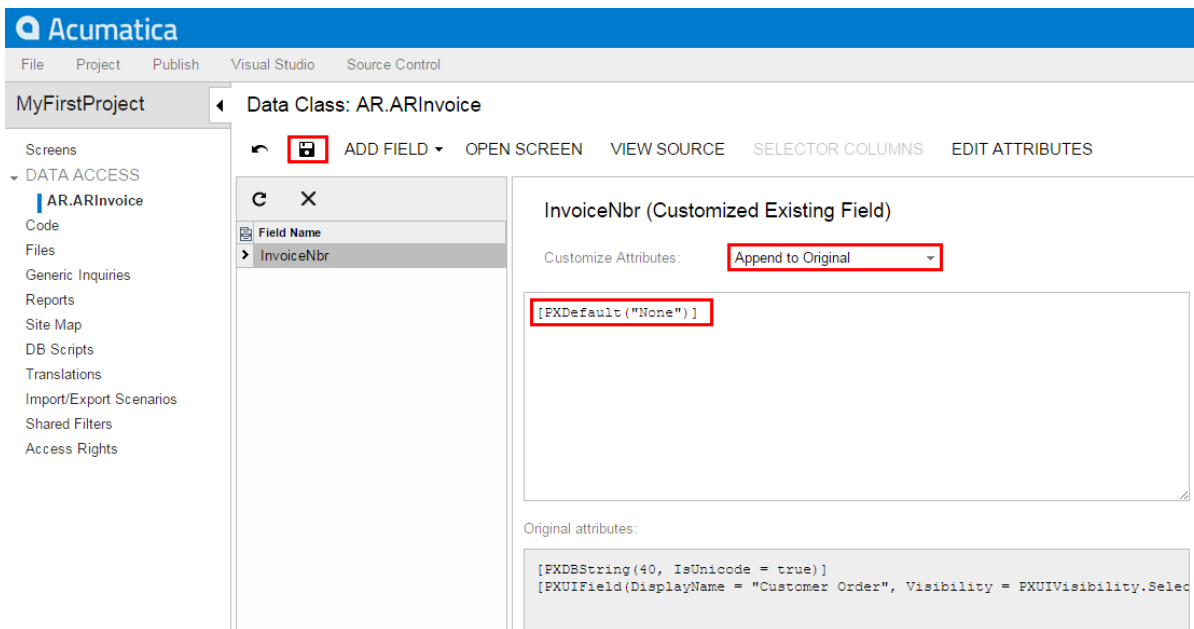


Figure: Appending an attribute to the DAC field

The system adds the customized data access class to the **Data Access** list of project items. For more information, see [Customized Data Classes](#).

To view the changes applied to the *Invoices* form, publish the customization project and add a new document on the *Invoices* form. When you add a new document, the system will insert the *None* default value into the **Customer Order** box as you have specified with the `PXDefault` attribute.

To implement the second requirement of the task, inspect the **Discount Percent** column on the **Document Details** tab of the *Invoices* form. To do this, on the **Customization** menu, select **Inspect Element** and click the area of the **Discount Percent** column header. The system should retrieve the following information of the control that will display in the **Element Properties** dialog box:

- **Control Type:** *Grid Column*. The type of the UI control.
- **Data Class:** *ARTran*. The data access class that contains the inspected field.
- **Data Field:** *DiscPct*. The field that is represented by the **Discount Percent** column in the table.
- **Business Logic:** *SOInvoiceEntry*. The business logic controller (BLC, also referred to as *graph*) that provides the logic executed on the *Invoices* form.

To modify attributes of the DAC field, select **Actions > Customize Data Fields** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the data field and click **OK**.

Data Class Editor opens for customization of the `DiscPct` field of the `ARTran` data access class, as the screenshot below shows.

The `PXDBDecimal` attribute specifies the range of valid values for the field. According to the customization task, you have to modify the minimum and maximum values specified in the attribute. To do this, you have to replace the original attributes with the custom ones.

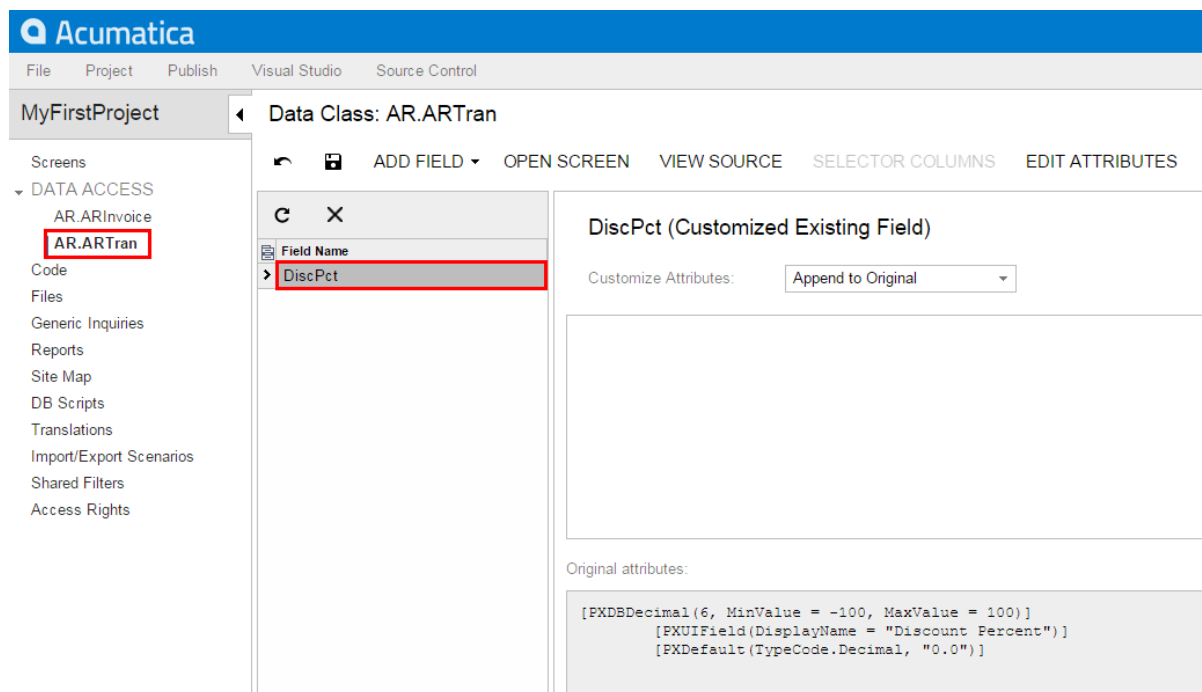


Figure: The data field added to the customization project

To replace the attributes, in the **Customize Attributes** box, select *Replace Original*. The original attributes will be copied to the text area where you enter custom attributes.

In the `PXDBDecimal` attribute, change the minimum and maximum values, as listed below, and click **Save** in Data Class Editor to save the changes.



: If you replace attributes, you have to repeat all attributes that you want to remain on the data field.

```
[PXDBDecimal(6, MinValue = -30, MaxValue = 25)]
[PXUIField(DisplayName = "Discount Percent")]
[PXDefault(TypeCode.Decimal, "0.0")]
```

The screenshot shows the Acumatica interface for customizing a data field. The left sidebar shows the project structure under 'MyFirstProject' and 'DATA ACCESS'. The main area is titled 'Data Class: AR.ARTran'. A 'Field Name' table lists 'DiscPct'. The 'Customize Attributes' dropdown is set to 'Replace Original'. The 'Original attributes' section shows the default code: `[PXDBDecimal(6, MinValue = -100, MaxValue = 100)]`, `[PXUIField(DisplayName = "Discount Percent")]`, and `[PXDefault(TypeCode.Decimal, "0.0")]`. The 'Customized Existing Field' section shows the updated code: `[PXDBDecimal(6, MinValue = -30, MaxValue = 25)]`, `[PXUIField(DisplayName = "Discount Percent")]`, and `[PXDefault(TypeCode.Decimal, "0.0")]`.

Figure: Replacing attributes on the data field

The system adds the customized data access class to the **Data Access** list of project items. For more information, see [Customized Data Classes](#).

To test the result of the customization, publish the customization project, open the *Invoices* form, insert a new document, and add a record to the details table, as shown in the screenshot below. If you try to specify a value in the **Discount** column greater than 25 percent or less than -30 percent, the system will automatically correct the value to the maximum 25 or minimum -30, respectively.

Modifying a BLC Action

To modify the behavior of an action that is defined within a business logic controller (BLC, also referred to as *graph*), you should override an action delegate.

Suppose that you need to rename the **Release** button to **Extended Release** on the *Receipts* form (**Distribution > Inventory > Enter > Receipts**), which is shown in the following screenshot.

Inventory New York - Receipts

Reference Nbr.: **R100081** Transfer Nbr.: Total Qty.: 200.00
 Status: **Balanced** External Ref.: Total Cost: 30,000.00
 * Date: 10/1/2012 * Post Period: 09-2012
 Description: TXTestScenario_2

Transaction Details

* Branch	* Inventory ID	* Warehouse	Location	Quantity	* UOM	Unit Cost	Ext. Cost	Reason Code	* Proj
MAIN	TX200001	RETAIL	1	100.00	PC	200.000000	20,000.00		X
MAIN	TX200002	RETAIL	1	100.00	PC	100.000000	10,000.00		X

Figure: Viewing the original Receipts form

To do this, you have to find the business logic controller for the form and modify the display name of the action that corresponds to the **Release** button.

To find the business logic controller for the form, on the **Customization** menu, select **Inspect Element** and click the button caption on the form. The system should retrieve the following information of the button, which appears in the **Element Properties** dialog box:

- **Control Type:** *DS Callback Command*. The type of the UI control.
- **Business Logic:** *INReceiptEntry*. The business logic controller that contains the code of the action that is represented by the **Release** button.

Thus, to rename the button, you have to change the `DisplayName` parameter of the `PXUIField` attribute for the `Release` action in the BLC extension for the `INReceiptEntry` class.

To modify the attributes of the action, select **Actions > Customize Business Logic** in the Element Inspector. In the **Select Customization Project** dialog, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

The Code Editor opens for customization of the business logic code of the form (see the screenshot below). The system generates the BLC extension class in which you can develop the customization code. (See [Graph Extensions](#) for details.)

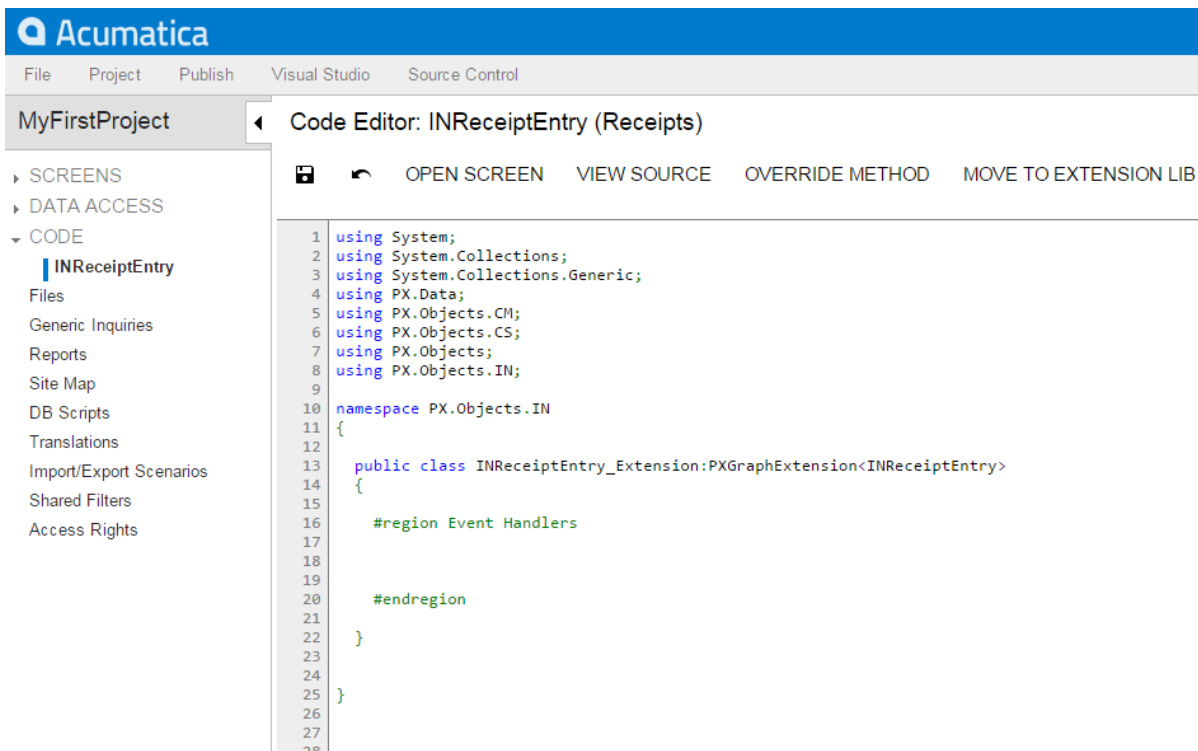


Figure: The BLC extension class generated for customization of the business logic code executed for the Receipts form

To modify the `DisplayName` parameter of the `PXUIField` attribute for the `Release` action, you have to redefine the action in the BLC extension class, to which the attributes are attached. Insert the code listed below into the `INReceiptEntry_Extension` class, as shown in the screenshot below. In the `DisplayName` parameter, you specify the new name for the button, *"Extended Release"*.

```

#region Customized Actions
public PXAction<INRegister> release;
[PXUIField(DisplayName = "Extended Release",
           MapEnableRights = PXCacheRights.Update,
           MapViewRights = PXCacheRights.Update)]
[PXProcessButton]
protected IEnumerable Release(PXAdapter adapter)
{
    return Base.release.Press(adapter);
}
#endregion

```

The redefined action delegate must have exactly the same signature—that is, the return value, the name of the method, and any method parameters—as the base action delegate has. You always have to redefine the action delegate to alter either its delegate or the attributes attached to the action. To use the action declared within the base BLC or a lower-level extension from the action delegate, you should redefine the generic `PXAction<TNode>` data member within the BLC extension, as shown in the code above. You do not need to redefine the data member when it is not meant to be used from the action delegate. For details, see [Graph Extensions](#). When you redefine an action delegate, you have to repeat the attributes attached to the action. Every action delegate must have `PXButtonAttribute` (or the derived attribute) and `PXUIFieldAttribute` attached.

Click **Save** in Code Editor to save the changes.

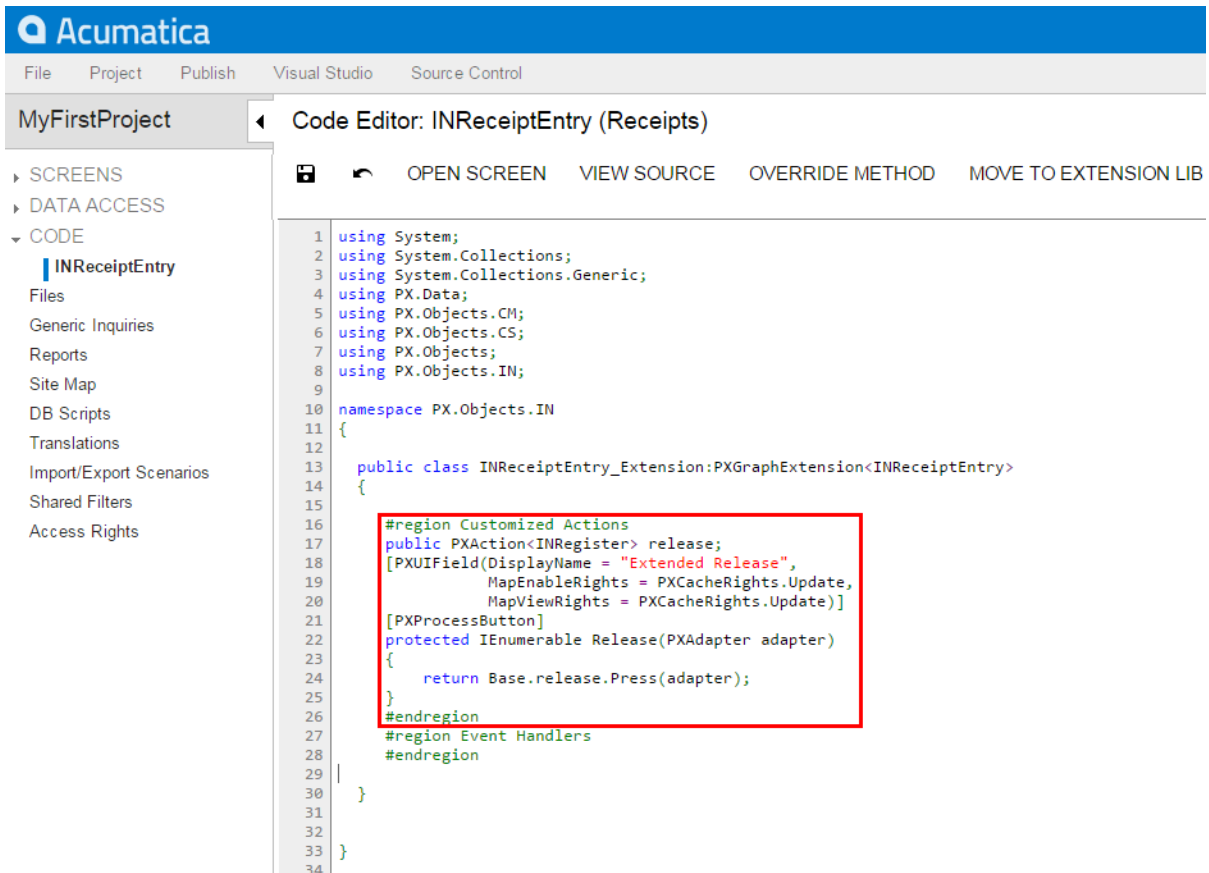


Figure: The Release action redefined in the BLC extension class with the modified DisplayName

The system adds the customization to the business logic code to the **Code** list of project items. See [Code](#) for details.

To view the result of the customization, publish the customization project and open the *Receipts* form. Verify that the button caption has changed to **Extended Release**, as the following screenshot shows. Click this button for a document with the Balanced status and verify that the release procedure works exactly as it worked before publication of the current customization project.

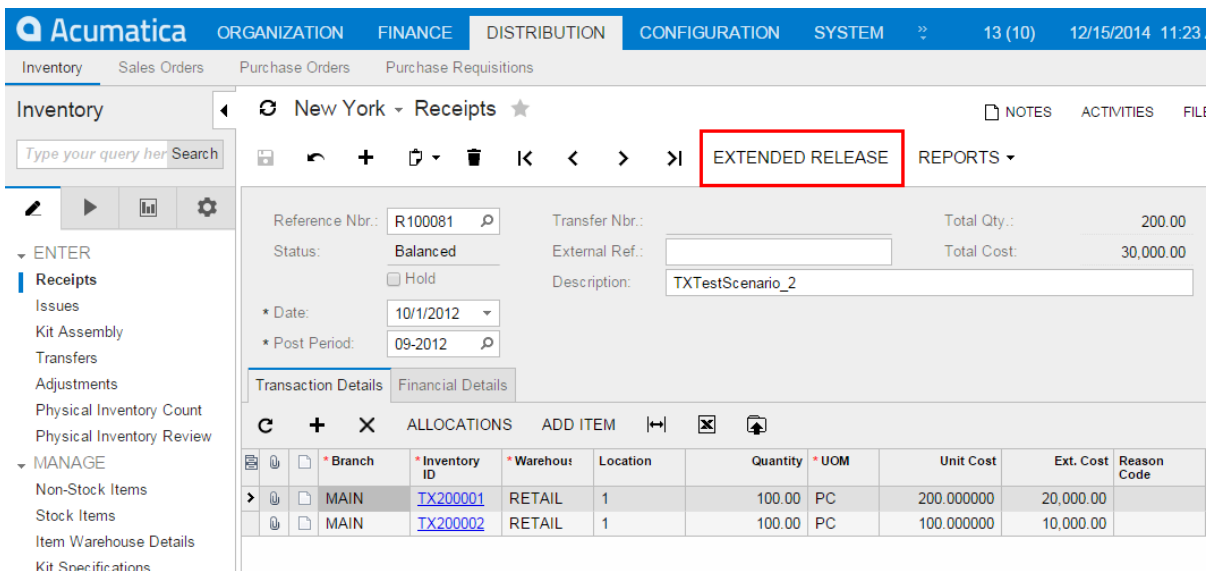


Figure: Viewing the result of the customization

Modifying a BLC Data View

A *data view* is a `PXSelect` BQL expression declared in a business logic controller (BLC, also referred to as *graph*) for accessing and manipulating data. A data view may contain a delegate that is used to extend data access. To modify the data view, you have to alter the appropriate `PXSelect` BQL expression.

Suppose that you need to filter the rows on the **Vendor Details** tab of the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**). Open the original *Stock Items* form and create an item that has multiple vendor records on the **Vendor Details** tab, one or more of which has a **Lot Size** column value that equals 0.00, as the screenshot below illustrates.

Your task is to modify the selection of records for the **Vendor Details** tab so that only the records that have a non-zero **Lot Size** value are retrieved to the tab. To resolve this customization task, you have to redeclare the data view that provides the data displayed on the tab.

The screenshot shows the Acumatica interface for the 'Stock Items' form. The 'Vendor Details' tab is active, displaying a table of vendor information. The table has columns for 'Acti', 'Defc', '* Vendor ID', 'Vendor Name', 'Warehouse', '* Purch. Unit', 'Lot Size', and 'Vend Inven'. The 'Lot Size' column is highlighted with a red box, showing values of 2.00, 1.00, 0.00, and 5.00 for the four vendors listed.

Acti	Defc	* Vendor ID	Vendor Name	Warehouse	* Purch. Unit	Lot Size	Vend Inven
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	ACITAISSYST	Acitai Systems - Computer Ser...		PC	2.00	
<input checked="" type="checkbox"/>	<input type="checkbox"/>	ADPSERVICE	Automatoc Data Processing Inc.		PC	1.00	
<input checked="" type="checkbox"/>	<input type="checkbox"/>	ARKTAK	Arktak Networks		PC	0.00	
<input checked="" type="checkbox"/>	<input type="checkbox"/>	BETAIR	Beta Air Lines		PC	5.00	

Figure: Viewing the original Stock Items form

The data for the **Vendor Details** tab is provided by the `VendorItems` data view that is defined in the `InventoryItemMaint` class—that is, the business logic controller for the *Stock Items* form. To resolve the customization task, you have to redefine the `VendorItems` data view in the BLC extension for the business logic controller of the form.

To select the business logic controller for customization, on the **Customization** menu, select **Inspect Element** and click the **Vendor Details** tab. The system should retrieve the following information that appears in the **Element Properties** dialog box:

- **Business Logic:** `InventoryItemMaint`. The business logic controller that provides the logic for the *Stock Items* form.

Select **Actions > Customize Business Logic** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

The Code Editor opens for customization of the business logic code of the form (see the screenshot below). The system generates the BLC extension class in which you can develop the customization code. (See [Graph Extensions](#) for details.)

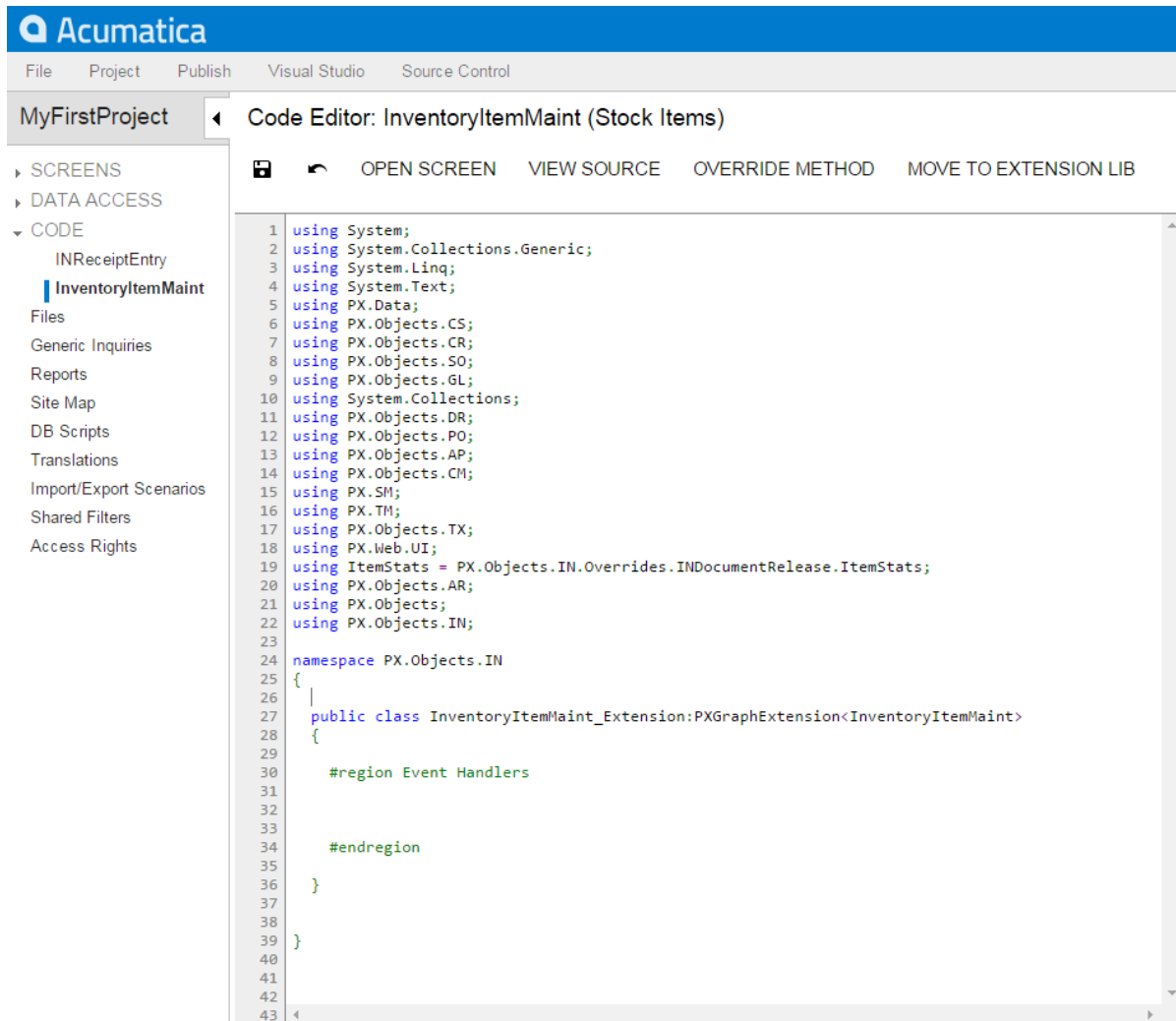


Figure: The BLC extension class generated for customization of the business logic code executed for the Stock Items form

In the BLC extension class for `InventoryItemMaint`, redefine the `VendorItems` data view, as listed below.

```

public POVendorInventorySelect<POVendorInventory,
    LeftJoin<InventoryItem,
        On<InventoryItem.inventoryID, Equal<POVendorInventory.inventoryID>>,
    LeftJoin<Vendor,
        On<Vendor.bAccountID, Equal<POVendorInventory.vendorID>>,
    LeftJoin<Location,
        On<Location.bAccountID, Equal<POVendorInventory.vendorID>,
        And<Location.locationID,
    Equal<POVendorInventory.vendorLocationID>>>>>>,
    Where<POVendorInventory.inventoryID, Equal<Current<InventoryItem.inventoryID>>,
        And<POVendorInventory.lotSize, Greater<decimal0>>>, InventoryItem>
    VendorItems;

```

To modify a data view, you have to redefine the data view in the BLC extension class. The data view redefined within a BLC extension completely replaces the base data view within the `Views` collection of a BLC instance, including all attributes attached to the data view declared within the base BLC. You can either attach the same set of attributes to the data view or completely redeclare the attributes. For details, see [Graph Extensions](#). The data view must have exactly the same identifier, `VendorItems`, which is referred in the UI control.

Click **Save** in Code Editor to save the changes.

The screenshot shows the Acumatica Code Editor interface. The left sidebar displays a project tree with 'InventoryItemMaint' selected under the 'CODE' folder. The main editor area shows the following C# code:

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using PX.Data;
6 using PX.Objects.CS;
7 using PX.Objects.CR;
8 using PX.Objects.SO;
9 using PX.Objects.GL;
10 using System.Collections;
11 using PX.Objects.DR;
12 using PX.Objects.PO;
13 using PX.Objects.AP;
14 using PX.Objects.CM;
15 using PX.SM;
16 using PX.TM;
17 using PX.Objects.TX;
18 using PX.Web.UI;
19 using ItemStats = PX.Objects.IN.Overrides.INDocumentRelease.ItemStats;
20 using PX.Objects.AR;
21 using PX.Objects;
22 using PX.Objects.IN;
23
24 namespace PX.Objects.IN
25 {
26
27     public class InventoryItemMaint_Extension:PXGraphExtension<InventoryItemMaint>
28     {
29         #region Customized Data Views
30         public POVendorInventorySelect<POVendorInventory,
31             LeftJoin<InventoryItem,
32                 On<InventoryItem.inventoryID, Equal<POVendorInventory.inventoryID>>,
33                 LeftJoin<Vendor,
34                     On<Vendor.bAccountID, Equal<POVendorInventory.vendorID>>,
35                     LeftJoin<Location,
36                         On<Location.bAccountID, Equal<POVendorInventory.vendorID>,
37                             And<Location.locationID, Equal<POVendorInventory.vendorLocationID>>>>>>,
38                 Where<POVendorInventory.inventoryID, Equal<Current<InventoryItem.inventoryID>>,
39                     And<POVendorInventory.lotSize, Greater<decimal0>>>, InventoryItem> VendorItems;
40         #endregion
41
42         #region Event Handlers
43         #endregion
44
45     }
46
47 }
48
49

```

Figure: The data view redefined in the BLC extension class

The system adds the customization to the business logic code to the **Code** list of project items. See [Code](#) for details.

To view the result of the customization, publish the customization project and open the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**). Verify that the records are filtered on the **Vendor Details** tab as required according to the customization task (only the records with non-zero **Lot Size** are retrieved).

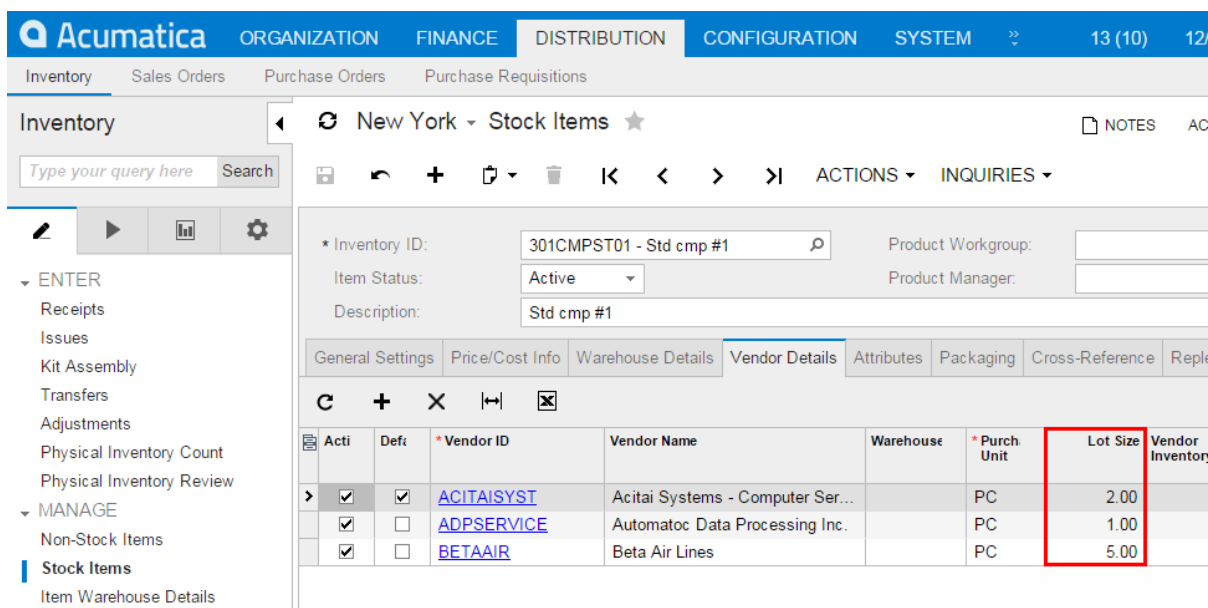


Figure: Viewing the result of the customization

Declaring or Altering a BLC Data View Delegate

You can modify the data view delegate—that is, the method that is invoked when the records are retrieved through the data view.



: For this example, you have to enable the *Warehouses* feature in the system to view the **Warehouse Details** tab on the *Stock Items* form.

For the *Stock Items* form, suppose that you have to make the system display the default warehouse in the **Default Receipt To** column for those records for which this column is empty. Your customization task specifies that the default warehouse for such records is the **Default Receipt To** warehouse of the default record selected on the **Warehouse Details** tab.

Open the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**) and select the **Warehouse Details** tab to view the original tab, which is shown in the screenshot below.

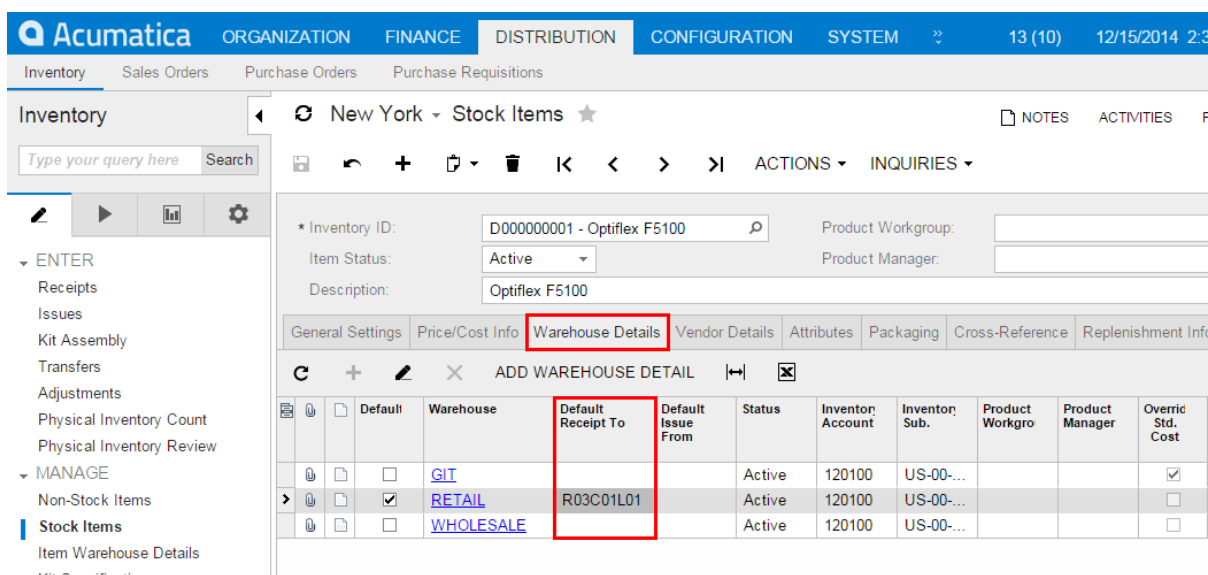


Figure: Viewing the original Stock Items form

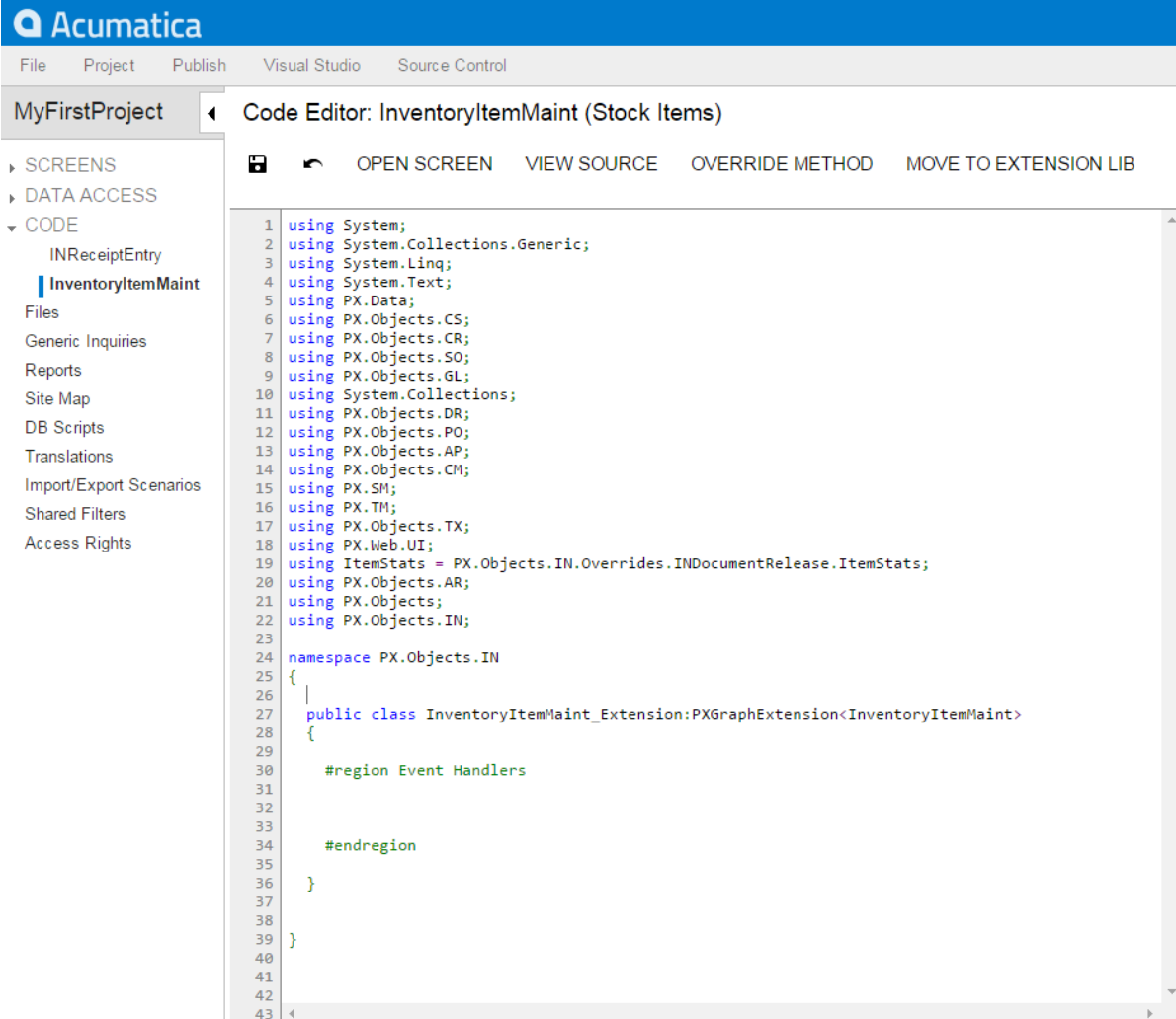
To resolve this task, you have to modify the logic that retrieves the records to the **Warehouse Details** tab—that is, the modify the delegate of the data view that provides data for the tab. The data view that provides data for the tab is `itemsiterecords` that is defined in the business logic controller of the form. The business logic controller of the form is the `InventoryItemMaint` class.

To select the business logic controller for customization, on the **Customization** menu, select **Inspect Element** and click the **Warehouse Details** tab. The system should retrieve the following information that appears in the **Element Properties** dialog box:

- **Business Logic:** `InventoryItemMaint`. The business logic controller that provides the logic for the `Stock Items` form.

Select **Actions** > **Customize Business Logic** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

The Code Editor opens for customization of the business logic code of the form (see the screenshot below). The system generates the BLC extension class in which you can develop the customization code. (See [Graph Extensions](#) for details.)



```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using PX.Data;
6 using PX.Objects.CS;
7 using PX.Objects.CR;
8 using PX.Objects.SO;
9 using PX.Objects.GL;
10 using System.Collections;
11 using PX.Objects.DR;
12 using PX.Objects.PO;
13 using PX.Objects.AP;
14 using PX.Objects.CM;
15 using PX.SM;
16 using PX.TM;
17 using PX.Objects.TX;
18 using PX.Web.UI;
19 using ItemStats = PX.Objects.IN.Overrides.INDocumentRelease.ItemStats;
20 using PX.Objects.AR;
21 using PX.Objects;
22 using PX.Objects.IN;
23
24 namespace PX.Objects.IN
25 {
26     |
27     public class InventoryItemMaint_Extension:PXGraphExtension<InventoryItemMaint>
28     {
29         |
30         #region Event Handlers
31
32
33
34         #endregion
35
36     }
37
38
39 }
40
41
42
43

```

Figure: The BLC extension class generated for customization of the business logic code executed for the Stock Items form

To resolve the customization task, you have to redefine the `itemsiterecords` data view and implement the needed logic in the `itemSiteRecords()` data view delegate.

To the BLC extension class for `InventoryItemMaint`, add the code that is listed below.

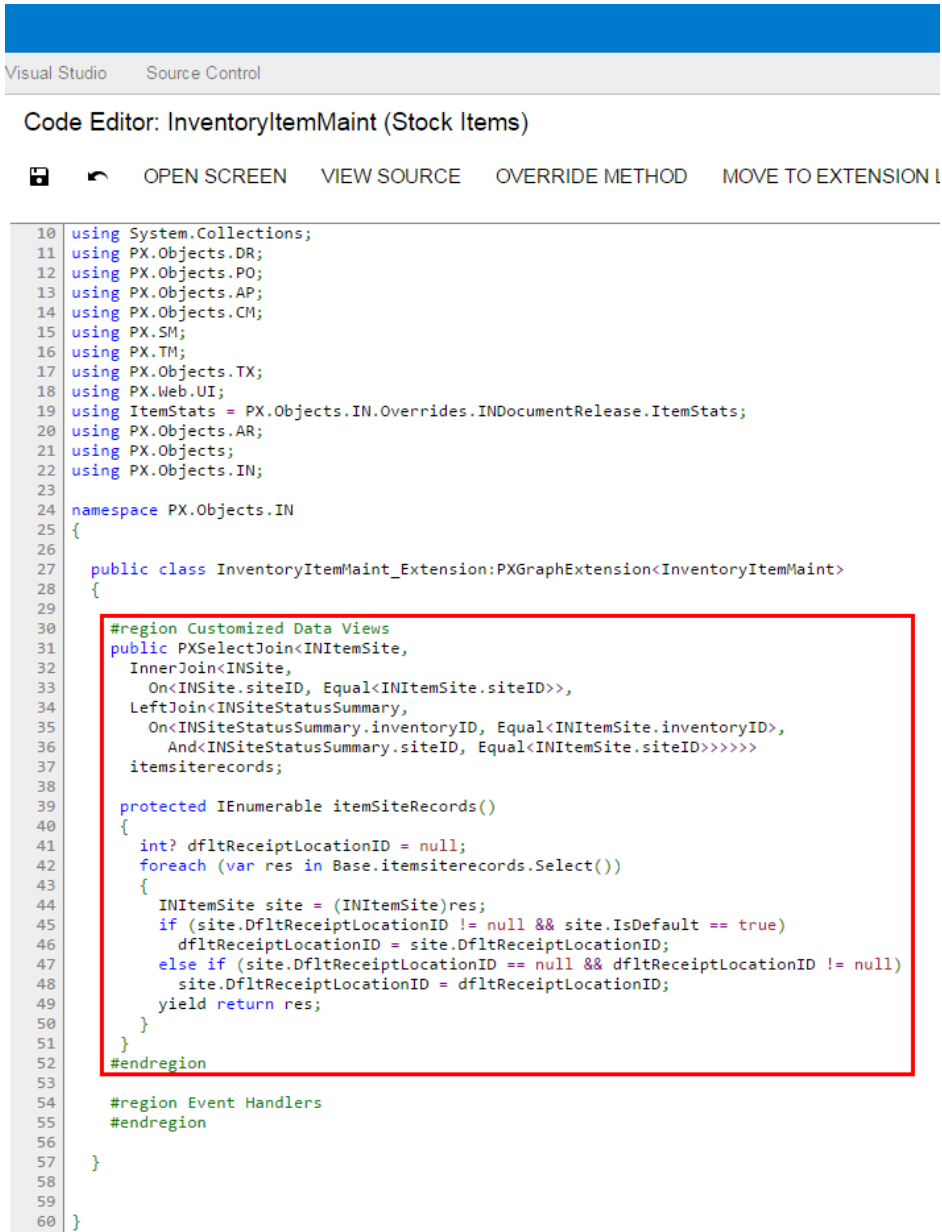
```
#region Customized Data Views
public PXSelectJoin<INItemSite,
    InnerJoin<INSite,
        On<INSite.siteID, Equal<INItemSite.siteID>>,
    LeftJoin<INSiteStatusSummary,
        On<INSiteStatusSummary.inventoryID, Equal<INItemSite.inventoryID>,
            And<INSiteStatusSummary.siteID, Equal<INItemSite.siteID>>>>>>
    itemsiterecords;

protected IEnumerable itemSiteRecords()
{
    int? dfltReceiptLocationID = null;
    foreach (var res in Base.itemsiterecords.Select())
    {
        INItemSite site = (INItemSite)res;
        if (site.DfltReceiptLocationID != null && site.IsDefault == true)
            dfltReceiptLocationID = site.DfltReceiptLocationID;
        else if (site.DfltReceiptLocationID == null && dfltReceiptLocationID != null)
            site.DfltReceiptLocationID = dfltReceiptLocationID;
        yield return res;
    }
}
#endregion
```

When you declare or alter a data view delegate within a BLC extension, the new delegate is attached to the corresponding data view. To query a data view declared within the base BLC or lower-level extension from the data view delegate, you should redeclare the data view within the BLC extension. You do not need to redeclare a generic `PXSelect<Table>` data member when it will not be used from the data view delegate. For details, see [Graph Extensions](#).

The redeclared data view delegate must have exactly the same signature—the return value, the name of the method, and any method parameters— as the base data view delegate.

Click **Save** in Code Editor to save the changes.



```

10 using System.Collections;
11 using PX.Objects.DR;
12 using PX.Objects.PO;
13 using PX.Objects.AP;
14 using PX.Objects.CM;
15 using PX.SM;
16 using PX.TM;
17 using PX.Objects.TX;
18 using PX.Web.UI;
19 using ItemStats = PX.Objects.IN.Overrides.INDocumentRelease.ItemStats;
20 using PX.Objects.AR;
21 using PX.Objects;
22 using PX.Objects.IN;
23
24 namespace PX.Objects.IN
25 {
26
27     public class InventoryItemMaint_Extension:PXGraphExtension<InventoryItemMaint>
28     {
29
30         #region Customized Data Views
31         public PXSelectJoin<INItemSite,
32             InnerJoin<INSite,
33                 On<INSite.siteID, Equal<INItemSite.siteID>>,
34                 LeftJoin<INSiteStatusSummary,
35                     On<INSiteStatusSummary.inventoryID, Equal<INItemSite.inventoryID>,
36                     And<INSiteStatusSummary.siteID, Equal<INItemSite.siteID>>>>>>
37             itemsiterecords;
38
39         protected IEnumerable itemSiteRecords()
40         {
41             int? dfltReceiptLocationID = null;
42             foreach (var res in Base.itemsiterecords.Select())
43             {
44                 INItemSite site = (INItemSite)res;
45                 if (site.DfltReceiptLocationID != null && site.IsDefault == true)
46                     dfltReceiptLocationID = site.DfltReceiptLocationID;
47                 else if (site.DfltReceiptLocationID == null && dfltReceiptLocationID != null)
48                     site.DfltReceiptLocationID = dfltReceiptLocationID;
49                 yield return res;
50             }
51         }
52         #endregion
53
54         #region Event Handlers
55         #endregion
56
57     }
58
59 }
60

```

Figure: The data view and the data view delegate redefined in the BLC extension class

The system adds the customization to the business logic code to the **Code** list of project items. See [Code](#) for details.

To view the result of the customization, publish the customization project and open the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**). Verify that the default warehouse is displayed for all records as needed according to the customization task. The default warehouse is taken from the Default Receipt To warehouse of the record selected as Default in the table (see the screenshot below).

The screenshot shows the Acumatica interface for the 'Inventory' module, specifically the 'New York - Stock Items' form. The 'Warehouse Details' tab is active, displaying a table of warehouse information for the item 'D00000001 - Optiflex F5100'. The table has columns for Default, Warehouse, Default Receipt To, Default Issue From, Status, Inventor Account, Inventor Sub., Product Workgro, Product Manager, and Overrid Std. Cost. Three rows are listed: GIT, RETAIL, and WHOLESALE, all with a 'Default Receipt To' of R03C01L01. The 'Default Receipt To' column is highlighted with a red box.

Default	Warehouse	Default Receipt To	Default Issue From	Status	Inventor Account	Inventor Sub.	Product Workgro	Product Manager	Overrid Std. Cost
<input type="checkbox"/>	GIT	R03C01L01		Active	120100	US-00-...			<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	RETAIL	R03C01L01		Active	120100	US-00-...			<input type="checkbox"/>
<input type="checkbox"/>	WHOLESALE	R03C01L01		Active	120100	US-00-...			<input type="checkbox"/>

Figure: Viewing the result of the customization

Extending BLC Initialization

During the initialization of a business logic controller (BLC, also referred to as *graph*), you can retrieve additional preferences, change field settings, or configure the processing form. You can extend the BLC initialization by implementing the needed logic in a BLC extension. To extend the initialization process, you have to override the `Initialize()` method of the `PXGraphExtension<T>` class and do not use the constructor of the BLC extension class for that. During the BLC initialization, the system calls the `Initialize()` methods of all extension levels, from the lowest to the highest. See [Graph Extensions](#) for details.

In this example, you will create an extension of the `InventoryItemMaintExtension` BLC, which overrides the `Initialize()` method to change the display name of the `InventoryItem` data access class (DAC) field from *Inventory ID* to *Inventory Item ID* during the initialization of the base (original) BLC instance.

If you open the *Stock Items* form, you can see its original view with the **Inventory ID** control (see the following screenshot) that corresponds to the `InventoryItem` DAC field.

The screenshot shows the Acumatica web interface for the 'Inventory' module. The top navigation bar includes 'ORGANIZATION', 'FINANCE', 'DISTRIBUTION', 'CONFIGURATION', 'SYSTEM', and user information. The main header shows 'New York - Stock Items'. A search bar is present with the text 'Type your query here'. The left sidebar contains a tree view with categories: ENTER (Receipts, Issues, Kit Assembly, Transfers, Adjustments, Physical Inventory Count, Physical Inventory Review), MANAGE (Non-Stock Items, Stock Items, Item Warehouse Details, Kit Specifications), and EXPLORE (Inventory Summary, Inventory Allocation Details, Inventory Transaction Summary, Inventory Transaction History, Inventory Transaction Details, Inventory Transactions by Account). The 'Stock Items' option is highlighted with a red box. The main content area shows the 'Stock Items' form for item '301CMPST03 - Std cmp #3'. The 'Inventory ID' field is highlighted with a red box. The form includes fields for Item Status (Active), Description (Std cmp #3), Product Workgroup, and Product Manager. Below these are tabs for 'General Settings', 'Price/Cost Info', 'Warehouse Details', 'Vendor Details', 'Attributes', 'Packaging', 'Cross-Reference', 'Replenishment Info', and 'Deferred Revenue'. The 'General Settings' tab is active, showing sections for ITEM DEFAULTS, UNIT OF MEASURE, PHYSICAL INVENTORY, and WAREHOUSE DEFAULTS. ITEM DEFAULTS includes Item Class (MISC - Miscellaneous), Type (Component Part), Valuation Method (Standard), Tax Category (EXEMPT - Exempt Tax Category), and Posting Class (WCLASS - Warehouse-oriented class). UNIT OF MEASURE shows Base Unit (PC). PHYSICAL INVENTORY includes PI Cycle, ABC Code, and Movement Class. WAREHOUSE DEFAULTS includes Default Warehouse (WHOLESALE - Wholesale warehouse), Default Issue From (R01C01L01 - Row 1, column 1, level), and Default Receipt To (R01C01L01 - Row 1, column 1, level).

Figure: Viewing the original Stock Items form

To select the business logic controller for customization, on the **Customization** menu, select **Inspect Element** and click the **Inventory ID** label. The system should retrieve the following information that appears in the **Element Properties** dialog box:

- **Business Logic:** *InventoryItemMaint*. The business logic controller that provides the logic for the *Stock Items* form.

Select **Actions** > **Customize Business Logic** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

The Code Editor opens for customization of the business logic code of the form (see the screenshot below). The system generates the BLC extension class in which you can develop the customization code. (See [Graph Extensions](#) for details.)

The screenshot shows the Acumatica development environment. The left sidebar displays a project tree for 'MyFirstProject' with a tree view expanded to 'InventoryItemMaint'. The main window is a code editor titled 'Code Editor: InventoryItemMaint (Stock Items)'. The code is a BLC extension class for 'InventoryItemMaint' in the 'PX.Objects.IN' namespace. It includes various system and Acumatica namespaces and defines a class 'InventoryItemMaint_Extension' that inherits from 'PXGraphExtension<InventoryItemMaint>'. The class contains a region for event handlers.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using PX.Data;
6  using PX.Objects.CS;
7  using PX.Objects.CR;
8  using PX.Objects.SO;
9  using PX.Objects.GL;
10 using System.Collections;
11 using PX.Objects.DR;
12 using PX.Objects.PO;
13 using PX.Objects.AP;
14 using PX.Objects.CM;
15 using PX.SM;
16 using PX.TM;
17 using PX.Objects.TX;
18 using PX.Web.UI;
19 using ItemStats = PX.Objects.IN.Overrides.INDocumentRelease.ItemStats;
20 using PX.Objects.AR;
21 using PX.Objects;
22 using PX.Objects.IN;
23
24 namespace PX.Objects.IN
25 {
26
27     public class InventoryItemMaint_Extension:PXGraphExtension<InventoryItemMaint>
28     {
29
30         #region Event Handlers
31
32
33
34         #endregion
35
36     }
37
38
39 }
40
41
42
43

```

Figure: The BLC extension class generated for customization of the business logic code executed for the Stock Items form

To resolve the customization task, override the `Initialize()` method; in the method, set the display name of the `InventoryItem` data access class (DAC) field to *Inventory Item ID*. (For demonstrative purposes, the code also retrieves the `INSetup` data record and shows how you can implement the logic depending on values of the record.)



: Do not define constructors of BLC extension classes; always use the `Initialize()` method to implement the needed initialization logic.

To the BLC extension class for `InventoryItemMaint`, add the code that is listed below.

```

#region Extended initialization
public override void Initialize()
{
    PXUIFieldAttribute.SetDisplayName<InventoryItem.inventoryCD>(
        Base.Item.Cache,
        "Inventory Item ID");
    //retrieval of the setup record that stores preferences of the Inventory module

    INSetup setup = Base.insetup.Current;
    if (setup.UpdateGL != true)
    {
        // Do some actions here
    }
}

```

```

    }
}
#endregion



```

In this method, you change the display name of the `InventoryCD` field of the `InventoryItem` DAC. You do this by calling the `SetDisplayName()` static method of `PXUIFieldAttribute`. Also, you obtain the `INSetup` DAC instance by using the `Current` property of the `INSetup` data view. In your customization tasks, you may need to perform some actions that depend on field values of this setup DAC instance.

You do not need to explicitly invoke the `Initialize()` method on the previous extension levels; these methods are called automatically. Invoking `base.Initialize()` makes no sense, because the `base` variable points to the base class, which is `PXGraphExtension` (not the base graph). The `PXGraphExtension` class defines `Initialize()` as an empty method.

Click **Save** in Code Editor to save the changes.

Code Editor: InventoryItemMaint (Stock Items)



 OPEN SCREEN VIEW SOURCE OVERRIDE METHOD MOVE TO EXTEN

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using PX.Data;
6  using PX.Objects.CS;
7  using PX.Objects.CR;
8  using PX.Objects.SO;
9  using PX.Objects.GL;
10 using System.Collections;
11 using PX.Objects.DR;
12 using PX.Objects.PO;
13 using PX.Objects.AP;
14 using PX.Objects.CM;
15 using PX.SM;
16 using PX.TM;
17 using PX.Objects.TX;
18 using PX.Web.UI;
19 using ItemStats = PX.Objects.IN.Overrides.INDocumentRelease.ItemStats;
20 using PX.Objects.AR;
21 using PX.Objects;
22 using PX.Objects.IN;
23
24 namespace PX.Objects.IN
25 {
26     public class InventoryItemMaint_Extension:PXGraphExtension<InventoryItemMaint>
27     {
28         #region Extended initialization
29         public override void Initialize()
30         {
31             PXUIFieldAttribute.SetDisplayName<InventoryItem.inventoryCD>(
32                 Base.Item.Cache,
33                 "Inventory Item ID");
34             INSetup setup = Base.insetup.Current;
35             if (setup.UpdateGL != true)
36             {
37                 //Do some actions here
38             }
39         }
40     }
41     #endregion
42
43     #region Event Handlers
44     #endregion
45
46 }
47 }
48

```

Figure: The initialization method implemented in the BLC extension class

The system adds the customization to the business logic code to the **Code** list of project items. See [Code](#) for details.

To view the result of the customization, publish the customization project and open the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**). Verify that the label has changed to **Inventory Item ID**, as shown in the screenshot below.

The screenshot shows the Acumatica ERP interface for the 'Stock Items' form. The top navigation bar includes 'DISTRIBUTION', 'CONFIGURATION', 'SYSTEM', and user information. The main form is titled 'New York - Stock Items' and features a toolbar with icons for save, refresh, add, delete, and navigation. The form fields are organized into sections: 'General Settings' (with tabs for Price/Cost Info, Warehouse Details, Vendor Details, Attributes, Packaging, Cross-Reference, Replenishment Info, and Deferred Revenue), 'ITEM DEFAULTS', 'UNIT OF MEASURE', 'PHYSICAL INVENTORY', and 'WAREHOUSE DEFAULTS'. The 'Inventory Item ID' field is highlighted with a red box. The 'Item Class' is set to 'MISC - Miscellaneous', 'Type' is 'Component Part', 'Valuation Method' is 'Standard', 'Tax Category' is 'EXEMPT - Exempt Tax Category', and 'Posting Class' is 'WCLASS - Warehouse-oriented class'. The 'Warehouse Defaults' section shows 'Default Warehouse' as 'WHOLESALE - Wholesale warehouse' and 'Default Issue From' and 'Default Receipt To' as 'R01C01L01 - Row 1, column 1, level'.

Figure: Viewing the result of the customization

Altering the BLC of a Processing Form

For most processing forms of Acumatica ERP, the processing method, the button captions that are displayed on the toolbar of the form are set up during the BLC initialization (*business logic controller*, also referred to as *graph*)—that is, in the constructor of the BLC that provides the logic for the processing form, or in the `RowSelected` event handler of the main view data access class (DAC). To modify the logic that is executed during the BLC initialization, you have to create a BLC extension class and implement the needed logic in the `Initialize()` method overridden in the extension class; do not use the constructor for that.

For example, to disable the **Release All** button and modify the functionality of the **Release** button of the *Release IN Documents* process form, you create an extension on the `INDocumentRelease` BLC and override the `Initialize()` method.

Open the *Release IN Documents* form (**Distribution > Inventory > Processes > Daily**) to see its original view (see the screenshot below). Notice that the **Release** and **Release All** buttons are both available for clicking.

Figure: Analyzing the original Release IN Documents form

To select the business logic controller for customization, on the **Customization** menu, select **Inspect Element** and click any element on the form, for example, the **Release** button. The system should retrieve the following information that appears in the **Element Properties** dialog box:

- **Business Logic:** *INDocumentRelease*. The business logic controller that provides the logic for the *Release IN Documents* form.

Select **Actions** > **Customize Business Logic** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

The Code Editor opens for customization of the business logic code of the form (see the screenshot below). The system generates the BLC extension class in which you can develop the customization code. (See [Graph Extensions](#) for details.)

The screenshot shows the Acumatica Visual Studio interface. The top menu bar includes File, Project, Publish, Visual Studio, and Source Control. The main window is titled 'Code Editor: INDocumentRelease (Release IN Documents)'. On the left, a project tree shows 'MyFirstProject' with sub-items: SCREENS, DATA ACCESS, and CODE. Under CODE, 'INDocumentRelease' is selected, showing its sub-items: INReceiptEntry, InventoryItemMaint, Files, Generic Inquiries, Reports, Site Map, DB Scripts, Translations, Import/Export Scenarios, Shared Filters, and Access Rights. The code editor displays the following C# code:

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Runtime.Serialization;
5 using System.Text;
6 using PX.Common;
7 using PX.Data;
8 using PX.Objects.CS;
9 using PX.Objects.GL;
10 using PX.Objects.CM;
11 using PX.Objects.IN.Overrides.INDocumentRelease;
12 using PX.Objects.SO;
13 using PX.Objects.AP;
14 using PX.Objects;
15 using PX.Objects.IN;
16
17 namespace PX.Objects.IN
18 {
19
20     public class INDocumentRelease_Extension:PXGraphExtension<INDocumentRelease>
21     {
22
23         #region Event Handlers
24
25
26
27         #endregion
28
29     }
30
31 }
32
33

```

Figure: The BLC extension class generated for customization of the business logic code executed for the Release IN Documents form

To resolve the customization task, override the `Initialize()` method. In the method, disable the **Release All** button on the toolbar of the form, set the processing method, and set the method for asking the confirmation from the user before running the process.



: Do not define constructors of BLC extension classes; always use the `Initialize()` method to implement the needed initialization logic.

To the BLC extension class for `INDocumentRelease`, add the code that is listed below.

```

#region Extended initialization
public override void Initialize()
{
    // Disable the Process All button
    Base.INDocumentList.SetProcessAllEnabled(false);
    //If set by the SetParametersDelegate, the method is executed before the
    processing is run and
    //Provides conditional processing depending on the confirmation from the user
    //If the method returns true, the system executes the processing method that is
    set by SetProcessDelegate
    Base.INDocumentList.SetParametersDelegate(delegate(List<INRegister> documents)
    {
        return Base.INDocumentList.Ask("IN Document Release",
            "Are you sure?",
            MessageButtons.YesNo) == WebDialogResult.Yes;
    });
}

```

```

Base.INDocumentList.SetProcessDelegate(delegate(INRegister doc)
{
    INDocumentRelease.ReleaseDoc(new List<INRegister>() { doc }, true);
});
}
#endregion

```

For another example of extending the BLC initialization, see [Extending BLC Initialization](#).

Click **Save** in Code Editor to save the changes.

The screenshot shows the Visual Studio Code editor interface. The title bar indicates 'Visual Studio' and 'Source Control'. The editor window title is 'Code Editor: INDocumentRelease (Release IN Documents)'. Below the title bar are several icons and menu items: 'OPEN SCREEN', 'VIEW SOURCE', 'OVERRIDE METHOD', and 'MOVE TO EXTENSION LIB'. The main editor area displays C# code for the 'INDocumentRelease_Extension' class. A red rectangular box highlights the 'Initialize' method implementation, which is enclosed in a '#region Extended Initialization' block. The code within this block includes comments and logic to disable the 'Process All' button, prompt the user for confirmation via a dialog box, and then call 'SetProcessDelegate' to execute the 'ReleaseDoc' method on a list of documents.

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Runtime.Serialization;
5 using System.Text;
6 using PX.Common;
7 using PX.Data;
8 using PX.Objects.CS;
9 using PX.Objects.GL;
10 using PX.Objects.GH;
11 using PX.Objects.IN.Overrides.INDocumentRelease;
12 using PX.Objects.SO;
13 using PX.Objects.AP;
14 using PX.Objects;
15 using PX.Objects.IN;
16
17 namespace PX.Objects.IN
18 {
19
20     public class INDocumentRelease_Extension:PXGraphExtension<INDocumentRelease>
21     {
22
23         #region Extended Initialization
24         public override void Initialize()
25         {
26             // Disable the Process All button
27             Base.INDocumentList.SetProcessAllEnabled(false);
28             //If set by the SetParametersDelegate, the method is executed before the processing is run and
29             //Provides conditional processing depending on the confirmation from the user
30             //If the method returns true, the system executes the processing method that is set by SetProcessDelegate
31             Base.INDocumentList.SetParametersDelegate(delegate(List<INRegister> documents)
32             {
33                 return Base.INDocumentList.Ask("IN Document Release",
34                 "Are you sure?",
35                 MessageButtons.YesNo) == WebDialogResult.Yes;
36             });
37             Base.INDocumentList.SetProcessDelegate(delegate(INRegister doc)
38             {
39                 INDocumentRelease.ReleaseDoc(new List<INRegister>() { doc }, true);
40             });
41         }
42         #endregion
43         #region Event Handlers
44         #endregion
45     }
46 }
47
48
49 }
50

```

Figure: The initialization method implemented in the BLC extension class

The system adds the customization to the business logic code to the **Code** list of project items. See [Code](#) for details.

To view the result of the customization, publish the customization project and open the *Release IN Documents* form (**Distribution > Inventory > Processes > Daily**). Notice that the **Release All** button has become unavailable. Select a few check boxes for documents to be released and click **Release**. The dialog box appears with the question that you have added in the BLC code (see the following screenshot). Click **Yes** to release the documents or **No** to cancel the operation and exit the dialog box.

The screenshot shows the Acumatica interface for 'New York - Release IN Documents'. The 'RELEASE' button is highlighted with a red box. Below it, a table lists document types and their details:

Document Type	Reference Nbr.	Status	Date	Post Period	Total Qty.	Total Cost	Total Amount	Description
Adjustment	A060189	Balanced	1/24/2007	12-2006	0.00	0.00	0.00	IN017-04-01
Production	000004	Balanced	2/15/2012	01-2012	0.00	0.00	0.00	IN301-02-17

A dialog box titled 'IN Document Release' is open, asking 'Are you sure?' with 'YES' and 'NO' buttons.

Figure: Viewing the result of the customization

Adding or Altering BLC Event Handlers

The following examples of customization tasks demonstrate how you can implement custom handlers for events in extension classes to business logic controllers (BLCs, also referred to as *graphs*).

- [Implementing a Handler That is Appended to the Collection of Base Handlers](#)
- [Implementing a Handler That Replaces the Collection of Base Handlers](#)
- [Adding an Event Handler From the Layout Editor](#)

Implementing a Handler That is Appended to the Collection of Base Handlers

When you define an event handler in the BLC extension class with the same declaration, as it is defined in the base (original) BLC, this handler is added to the appropriate event handler collection. Depending on the event type, the event handler is appended to either the end of the collection or the start of it. When the event occurs, all event handlers in the collection are executed, from the first one to the last one. For details, see [Event Handlers](#).

Suppose that you need to add validation of **Local Tax Category** on the **General Settings** tab of the **Stock Items** form (**Distribution > Inventory > Work Area > Manage**). **Local Tax Category** that is shown in the screenshot below is a custom field that has been added to the form, as described in the example of [Adding Data Fields](#). **Local Tax Category** is linked to the custom `UsrLocalTaxCategoryID` data field defined in the DAC extension for the `InventoryItem` data access class.

Your task is to check whether the selected **Local Tax Category** is the same as the selected **Tax Category** of the item and add a warning message that appears for the **Local Tax Category** box if they are the same.

The screenshot shows the Acumatica interface for the 'Stock Items' form. The 'Local Tax Category' field is highlighted with a red box. A modal dialog box titled 'Select - Local Tax Category' is open, showing a table of tax categories. The table has two columns: 'Tax Category ID' and 'Description'. The 'EXEMPT' category is selected.

Tax Category ID	Description
EXEMPT	Exempt Tax Category
INCLUSIVE	inclusive
INCLUSIVE2	INCLUSIVE2
INCLUSIVE3	INCLUSIVE3
PURCHCAT	Purch Tax Cat
TAXABLE	Taxable Goods and Services
WITHHOLDIN	Withholding1

Figure: The control for the custom field on the Stock Items form

To resolve the task, you implement validation of fields on the `RowUpdating` event handler that you will add to the BLC extension for the *Stock Items* form. In this variant of implementation, the validation will occur when the user attempts to save the `InventoryItem` record. In the handler, you compare the `TaxCategoryID` and `UsrLocalTaxCategoryID` fields and return an error message that displays for the **Local Tax Category** box if the values are equal.

To select the business logic controller for customization, on the **Customization** menu, select **Inspect Element** and click the **Warehouse Details** tab. The system should retrieve the following information that appears in the **Element Properties** dialog box:

- **Business Logic:** *InventoryItemMaint*. The business logic controller that provides the logic for the *Stock Items* form.

Select **Actions** > **Customize Business Logic** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

The Code Editor opens for customization of the business logic code of the form (see the screenshot below). The system generates the BLC extension class in which you can develop the customization code. (See [Graph Extensions](#) for details.)

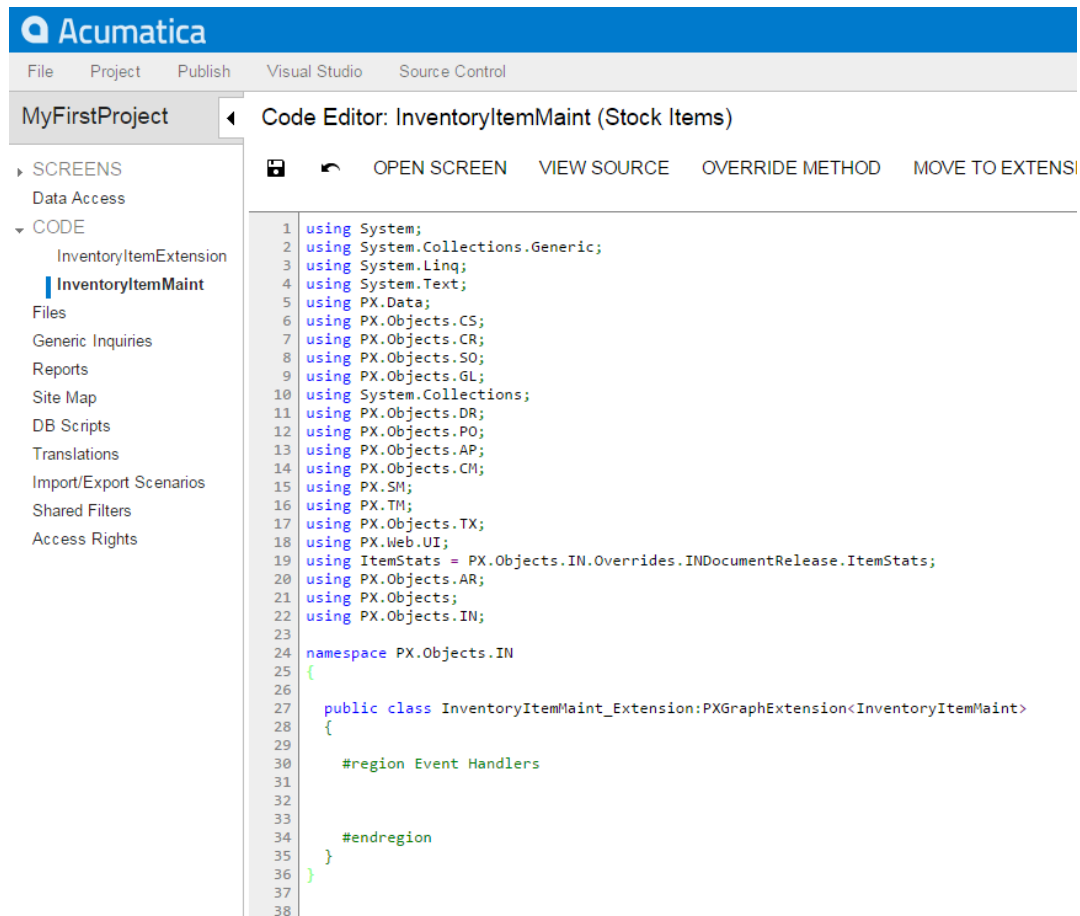


Figure: The BLC extension class generated for customization of the business logic code executed for the Stock Items form

To the BLC extension class for `InventoryItemMaint`, add the code that is listed below.

```

#region Event Handlers
protected void InventoryItem_RowUpdating(PXCache sender, PXRowUpdatingEventArgs e)
{
    InventoryItem row = e.NewRow as InventoryItem;
    InventoryItemExtension rowExt =
    cache.GetExtension<InventoryItemExtension>(row);
    if (row.TaxCategoryID != null && rowExt.UsrLocalTaxCategoryID != null &&
        row.TaxCategoryID == rowExt.UsrLocalTaxCategoryID)
    {
        cache.RaiseExceptionHandling<InventoryItemExtension.usrLocalTaxCategoryID>(
            row,
            rowExt.UsrLocalTaxCategoryID,
            new PXSetPropertyException("Tax category and local tax category should
            differ",
                PXErrorLevel.Warning));
    }
}
#endregion

```

The event handler checks whether the `TaxCategoryID` and `UsrLocalTaxCategoryID` field values are not null and do not equal each other. If these conditions are not satisfied, the handler issues

the warning that will be shown on the **Local Tax Category** box in the UI, which corresponds to the `UsrLocalTaxCategoryID` field.



: The field is accessed by its string name by using the `GetValue()` method on the cache. There is a number of ways how you can access customization objects from code. See [Access to a Custom Field](#) for details.

Click **Save** in Code Editor to save the changes.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using PX.Data;
6  using PX.Objects.CS;
7  using PX.Objects.CR;
8  using PX.Objects.SO;
9  using PX.Objects.GL;
10 using System.Collections;
11 using PX.Objects.DR;
12 using PX.Objects.PO;
13 using PX.Objects.AP;
14 using PX.Objects.CM;
15 using PX.SM;
16 using PX.TM;
17 using PX.Objects.TX;
18 using PX.Web.UI;
19 using ItemStats = PX.Objects.IN.Overrides.INDocumentRelease.ItemStats;
20 using PX.Objects.AR;
21 using PX.Objects;
22 using PX.Objects.IN;
23
24 namespace PX.Objects.IN
25 {
26
27     public class InventoryItemMaint_Extension:PXGraphExtension<InventoryItemMaint>
28     {
29
30         #region Event Handlers
31
32         protected void InventoryItem_RowUpdating(PXCache cache, PXRowUpdatingEventArgs e)
33         {
34             InventoryItem row = e.NewRow as InventoryItem;
35             InventoryItemExtension rowExt = cache.GetExtension<InventoryItemExtension>(row);
36             if (row.TaxCategoryID != null && rowExt.UsrLocalTaxCategoryID != null &&
37                 row.TaxCategoryID == rowExt.UsrLocalTaxCategoryID)
38             {
39                 cache.RaiseExceptionHandling<InventoryItemExtension.usrLocalTaxCategoryID>(
40                     row,
41                     rowExt.UsrLocalTaxCategoryID,
42                     new PXSetPropertyException("Tax category and local tax category should differ",
43                                             PXErrorLevel.Warning));
44             }
45         }
46         #endregion
47     }
48 }

```

Figure: The event handler implemented in the BLC extension class

The system adds the customization to the business logic code to the **Code** list of project items. See [Code Editor](#) for details.

To view the result of the customization, publish the customization project and open the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**). To test the validation, select an item on the form, set the **Local Tax Category** the same as the **Tax Category** of the item and click **Save**. The warning message appears for the **Local Tax Category** box, as it was required by the task. To view the warning message, point the mouse to the warning icon, as shown on the screenshot below.

The screenshot shows the Acumatica interface for the 'Stock Items' form. The 'General Settings' tab is active, and the 'Tax Category' and 'Local Tax Category' fields are both set to 'EXEMPT - Exempt Tax Category'. A yellow warning box with a red border is displayed, stating 'Tax category and local tax category should differ'. The left sidebar shows navigation options like 'ENTER', 'MANAGE', and 'EXPLORE'.

Figure: The warning message appears on saving attempt

Implementing a Handler That Replaces the Collection of Base Handlers

In a BLC extension, you can define an event handler so that the handler replaces the base BLC event handler collection. To do this, you declare the event handler with an additional parameter, as described below. When the event is raised, the system calls the event handler with an additional parameter of the highest-level BLC extension. The system passes the link to the event handler with an additional parameter from the extension of the previous level, if such an event handler exists, or to the first item in the event handler collection (also described in [Event Handlers](#)). In the event handler that replaces the collection of handlers of the base BLC, you can invoke the collection of base handlers as well as implement the logic to be executed before and after the base collection.

Suppose that you need to resolve the same customization task, as described above—that is, you have to add validation of **Local Tax Category** on the **General Settings** tab of the *Stock Items* form. To resolve the task, you implement validation in the `RowUpdating` event handler that you define in the BLC extension for the `InventoryItemMaint` class.

If you define the event handler, as listed below, the handler will replace the collection of handlers of the base BLC. The code of the replacing handler that is given below introduces the same behavior of the *Stock Items* form as the code of the event handler that is added to the collection, as described above in [Adding a Handler to the Collection of Base Handlers](#). In the handler below, you invoke the base handlers first and then validate the `TaxCategoryID` and `UsrLocalTaxCategoryID` fields and return an error message that displays for the **Local Tax Category** box if the validation fails.

```
#region Event Handlers
protected void InventoryItem_RowUpdating(
    PXCache cache, PXRowUpdatingEventArgs e, PXRowUpdating InvokeBaseHandler)
{
    //execute the collection of base handlers
    if(InvokeBaseHandler != null) InvokeBaseHandler(cache, e);
    //add the validation of Local Tax Category
    InventoryItem row = e.NewRow as InventoryItem;
    InventoryItemExtension rowExt = cache.GetExtension<InventoryItemExtension>(row);
    if (row.TaxCategoryID != null && rowExt.UsrLocalTaxCategoryID != null &&
        row.TaxCategoryID == rowExt.UsrLocalTaxCategoryID)
    {

```

```

        cache.RaiseExceptionHandling<InventoryItemExtension.usrLocalTaxCategoryID>(
            row,
            rowExt.UsrLocalTaxCategoryID,
            new PXSetPropertyException("Tax category and local tax category should
differ",
                PXErrorLevel.Warning));
    }
}
#endregion

```



: Notice the three parameters within the event handler declaration. When the event is raised, the system calls the event handler, passing a link to the first item in the base BLC event handler collection. You execute `InvokeBaseHandler()` to invoke the collection of event handlers defined in the base BLC. Depending on a customization task, you can skip the execution of base handlers or implement the needed logic before the base handlers are invoked.

Adding an Event Handler From the Layout Editor

To add a custom handler for a row or field event to the customization project, you can use the Layout Editor, as described below.



: For this example, you have to enable the *Lot and Serial Tracking* feature in the system to view the **Lot/Serial Class** box on the **General Settings** tab of the *Stock Items* form.

Open the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**). Suppose that you need to replace the error message that appears on the **Lot/Serial Class** box for an incompatible class selected in the box (see the screenshot below) with a warning message.

The screenshot shows the Acumatica interface for the 'New York - Stock Items' form. The 'Lot/Serial Class' field is highlighted with a red box, and a red error message is displayed below it: 'Lot/serial class cannot be changed when its tracing method is not compatible with the previous class and the item is in use.' The form includes fields for Inventory ID (301KITSTD1 - Std kit #1), Item Status (Active), Description (Std kit #1), and various settings under the 'ITEM DEFAULTS' section, such as Item Class (MISC - Miscellaneous), Type (Finished Good), Valuation Method (Standard), Tax Category (EXEMPT - Exempt Tax Category), Posting Class (CCLASS - Self-centered class), and Lot/Serial Class (PLNRE). The error message is positioned over the Lot/Serial Class field and its dropdown arrow.

Figure: The original error message on the Stock Items form

To resolve the task, you have to modify the logic of validation of the DAC field that corresponds to the **Lot/Serial Class** box. You have to implement a custom handler for the `FieldVerifying` event on the DAC field.

To find the DAC field and add the event handler to the customization project for it, open the form in Layout Editor. To do this, on the **Customization** menu, select **Inspect Element** and click the label or the box area of the **Lot/Serial Class** control. The system should retrieve the following information that appears in the **Element Properties** dialog box:

- **Control Type:** *Selector*. The type of the inspected UI control.
- **Data Class:** *InventoryItem*. The data access class that contains the field corresponding to the inspected control.
- **Data Field:** *LotSerClassID*. The data field that is linked to the inspected control.
- **Business Logic:** *InventoryItemMaint*. The business logic controller that provides the logic for the *Stock Items* form.

Click **Customize** in the Element Inspector to open the form in Layout Editor. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

Layout Editor opens for the form already navigated to the inspected control, as the screenshot below shows.

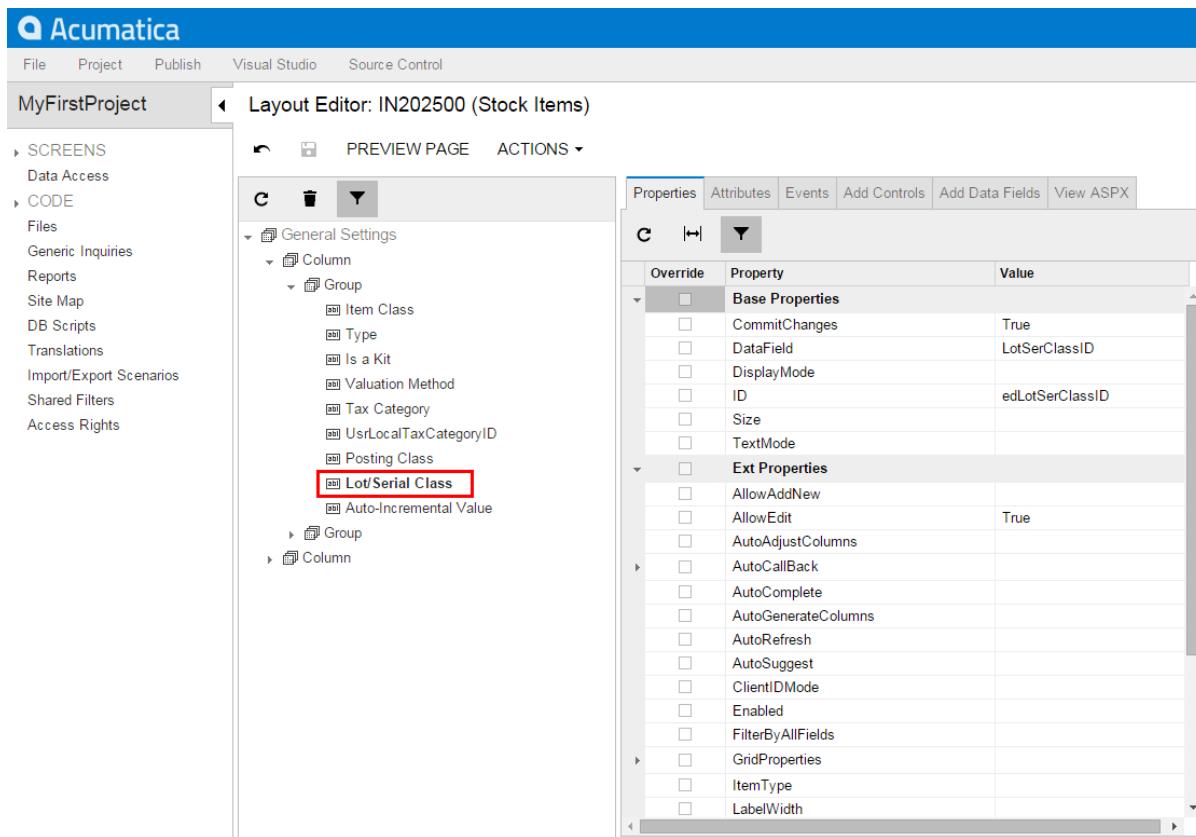


Figure: Layout Editor opened for the inspected element

To add the `FieldVerifying` event handler for data field underlying the **Lot/Serial Class** control, select the **Events** tab. On the tab, the selected check box **Handled in Source** means that the `FieldVerifying` event is already handled for the `LotSerClassID` field in the original business logic controller, `InventoryItemMaint` class.



: The **Events** tab displays the events depending on the control selected in the tree. The tab shows the field events if you select a control that is linked to a data field (that is, has the `DataField` property specified) and the row events of the parent container that is linked to a data view (that is, has the `DataMember` property specified). If you select a container that is not linked to a data view or data field, the tab shows no events.

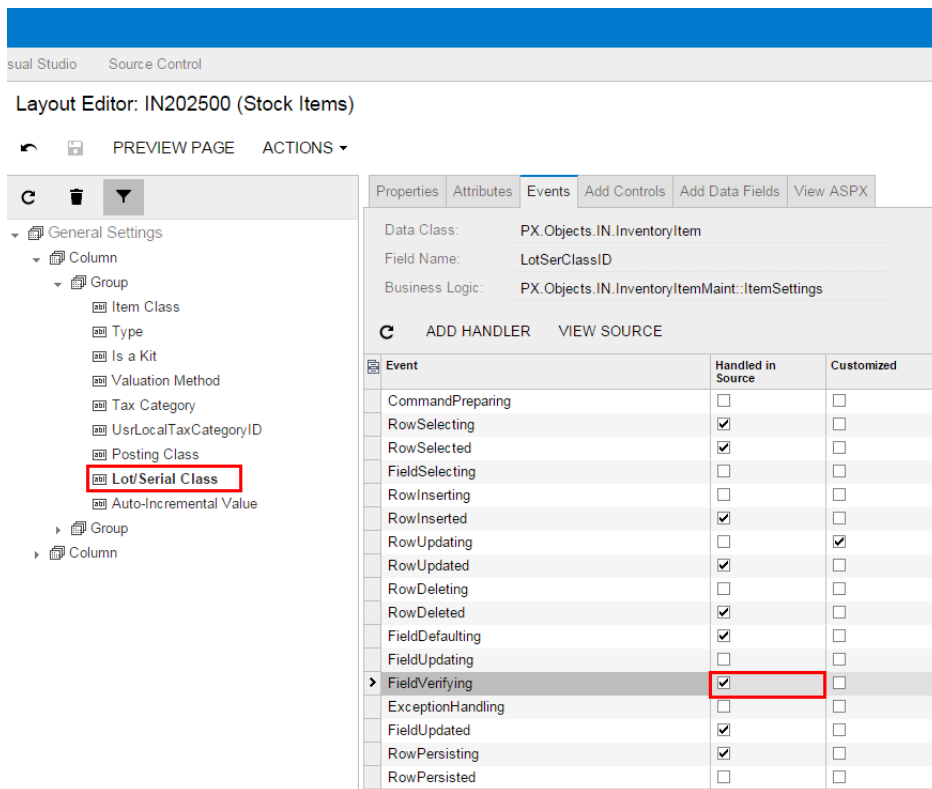


Figure: The event handled in the original business logic controller

To add a custom handler for the LotSerClassID field to the customization project, click the **FieldVerifying** event in the table and click **Add Handler** on the toolbar (see the screenshot below).

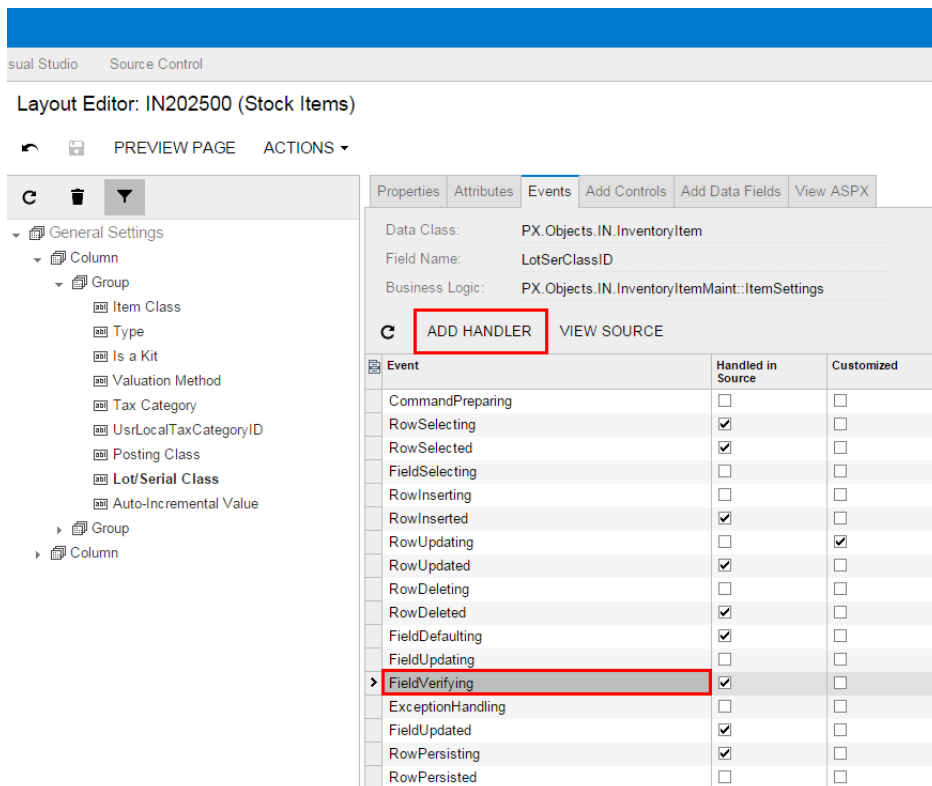


Figure: Adding an event handler to the customization project

The system generates the definition of the event handler, adds the definition to the BLC extension class for the form, and opens the Code Editor with the generated code (see the screenshot below). The system generates handlers with three input parameters; a handler with such definition replaces the collection of event handlers implemented in the original BLC, which is described in [Implementing a Handler That Replaces the Collection of Base Handlers](#). You can modify the generated code as needed.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using PX.Data;
6 using PX.Objects.CS;
7 using PX.Objects.CR;
8 using PX.Objects.SO;
9 using PX.Objects.GL;
10 using System.Collections;
11 using PX.Objects.DR;
12 using PX.Objects.PO;
13 using PX.Objects.AP;
14 using PX.Objects.CM;
15 using PX.SM;
16 using PX.TM;
17 using PX.Objects.TX;
18 using PX.Web.UI;
19 using ItemStats = PX.Objects.IN.Overrides.INDocumentRelease.ItemStats;
20 using PX.Objects.AR;
21 using PX.Objects;
22 using PX.Objects.IN;
23
24 namespace PX.Objects.IN
25 {
26
27     public class InventoryItemMaint_Extension:PXGraphExtension<InventoryItemMaint>
28     {
29
30         #region Event Handlers
31
32         protected void InventoryItem_LotSerClassID_FieldVerifying(PXCache cache, PXFieldVerifyingEventArgs e, PXFieldVerifying InvokeBaseHandler)
33         {
34             if(InvokeBaseHandler != null) InvokeBaseHandler(cache, e);
35             var row = (InventoryItem)e.Row;
36         }
37
38
39         protected void InventoryItem_RowUpdating(PXCache cache, PXRowUpdatingEventArgs e)
40         {
41
42
43

```

Figure: The generated definition of the event handler

Modify the generated code, as listed below, and click **Save** in Code Editor to save the changes to the customization project. In the code, you catch the exception that could be thrown from the original handlers and create another exception with the warning message instead of the original one.

```

protected void InventoryItem_LotSerClassID_FieldVerifying(
    PXCache cache, PXFieldVerifyingEventArgs e, PXFieldVerifying InvokeBaseHandler)
{
    try
    {
        if(InvokeBaseHandler != null) InvokeBaseHandler(cache, e);
    }
    catch (PXSetPropertyException ex)
    {
        cache.RaiseExceptionHandling<InventoryItem.lotSerClassID>(
            e.Row,
            e.NewValue,
            new PXSetPropertyException(
                ex.MessageNoPrefix, PXErrorLevel.Warning));
    }
}

```

To view the result of the customization, publish the customization project and open the *Stock Items* form (**Distribution > Inventory > Work Area > Manage**). To test the validation, select an incompatible class in the **Lot/Serial Class** box, as shown in the screenshot below. The warning message appears for the box instead of an error message.

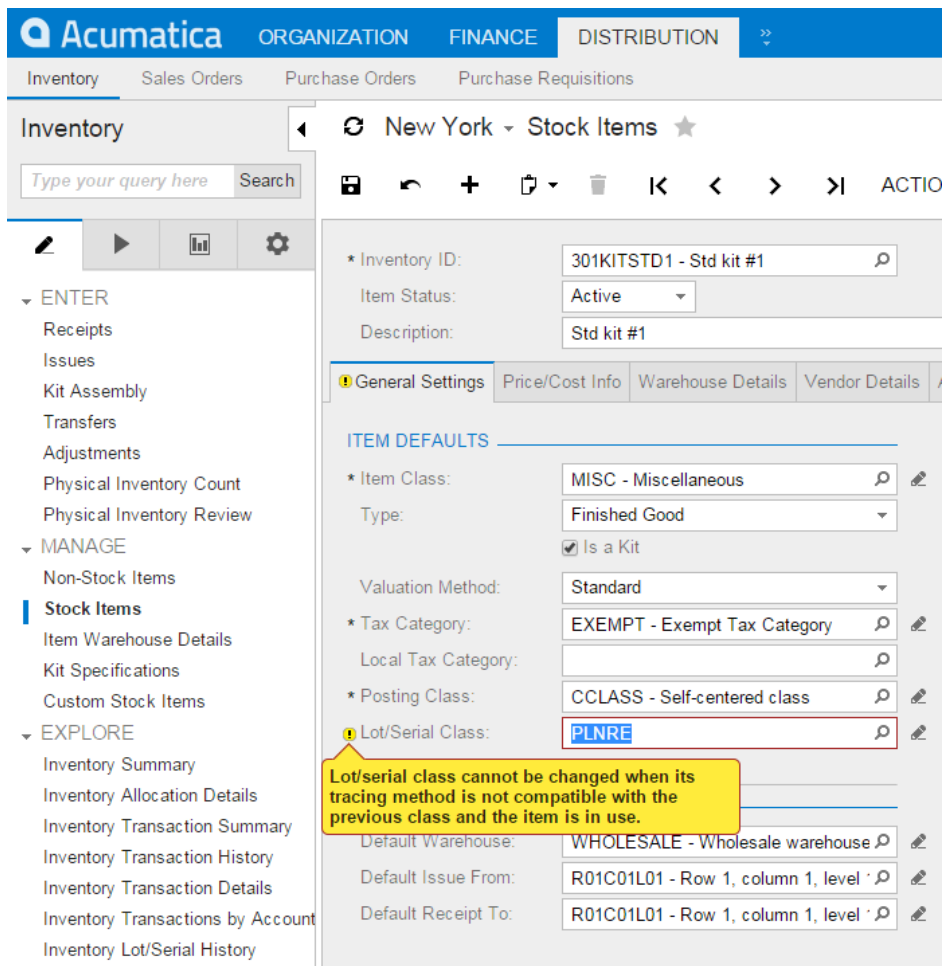


Figure: The warning message that appears on selecting an incompatible class

Altering BLC Virtual Methods

In a BLC extension, you can override virtual methods defined within a business logic controller (BLC, also referred to as *graph*). As with the event handlers, you have two options:

- You can define the override method with exactly the same signature—that is, the return value, the name of the method, and all method parameters—as the overridden base virtual method. As a result, the method is added to the queue of all override methods. When the system invokes the base method, all methods in the queue are executed sequentially, from the first to the last one. The lower the level the BLC extension has, the earlier the system invokes the override method.
- You can define the override method with an additional parameter, which represents the delegate for one of the following:
 - The override method with an additional parameter from the extension of the previous level, if such a method exists.
 - The base virtual method, if no override methods with additional parameters declared within lower-level extensions exist.

In both cases, you should attach the `PXOverrideAttribute` to the override method declared within the BLC extension. See the topics below for details.

- [Override Method That is Added to the Override Method Queue](#)
- [Override Method That Replaces the Original Method](#)

Override Method That is Added to the Override Method Queue

By declaring an override method with exactly the same signature as the overridden base virtual method, you extend the base method execution. The base BLC method is replaced at run time with the queue of methods that starts with the base BLC method. When the system invokes the base method, all methods in the queue are executed sequentially, from the first to the last one. The lower the level the BLC extension has, the earlier the system invokes the override method. If the system has invoked the base method, you have no option to prevent the override method queue from execution. To prevent the base method executions, see [Override Method That Replaces the Original Method](#) below.

Suppose that you need to modify the behavior of the *Journal Transactions* form (**Finance > General Ledger > Work Area > Enter**).

Open *Journal Transactions* and explore its original behavior. Start by adding a new journal transaction, selecting an account, and setting the **Debit Amount** to 2000.00. Then add one more journal transaction and select a different account, and notice that the **Credit Amount** is set by default to 2000.00 to balance the batch, as the following screenshot illustrates.

The screenshot shows the Acumatica Journal Transactions form. The form is titled "New York - Journal Transactions" and is in the "ENTER" state. The "Journal Transactions" menu item is highlighted in red. The form displays the following fields:

- Module: GL
- Branch: MAIN - New York
- Batch Number: <NEW>
- Ledger: ACTUAL
- Status: Balanced
- Currency: USD 1.00
- Transaction Date: 12/19/2014
- Post Period: 11-2014
- Debit Total: 2,000.00
- Credit Total: 2,000.00

The table below shows the journal entries:

Branch	Account	Description	Subaccount	Debit Amount	Credit Amount	Transaction Description
MAIN	100000	Petty Cash USD	US-00-00-00-000	2,000.00	0.00	
MAIN	710000	Office Expense	US-00-00-00-000	0.00	2,000.00	

Figure: The Credit Amount by default is set to balance the batch

Suppose that you need this form to work differently, so that if the user first inserts a debit entry, by default each **Credit Amount** value equals 0.00, to make the user add the required values manually. If the user first inserts a credit entry, a **Debit Amount** value should also equal 0.00 by default, so that the user must add the needed values. You can implement this behavior in the `PopulateSubDescr` method that you have to override in a BLC extension for the `JournalEntry` class.

To select the business logic controller for customization, on the **Customization** menu, select **Inspect Element** and click any element on the form, for example, the form area. The system should retrieve the following information that appears in the **Element Properties** dialog box:

- **Business Logic:** `JournalEntry`. The business logic controller that provides the logic for the *Journal Transactions* form.

Select **Actions > Customize Business Logic** in the Element Inspector. In the **Select Customization Project** dialog box, specify the project to which you want to add the customization item for the business logic controller of the form and click **OK**.

The Code Editor opens for customization of the business logic code of the form (see the screenshot below). The system generates the BLC extension class in which you can develop the customization code. (See [Graph Extensions](#) for details.)

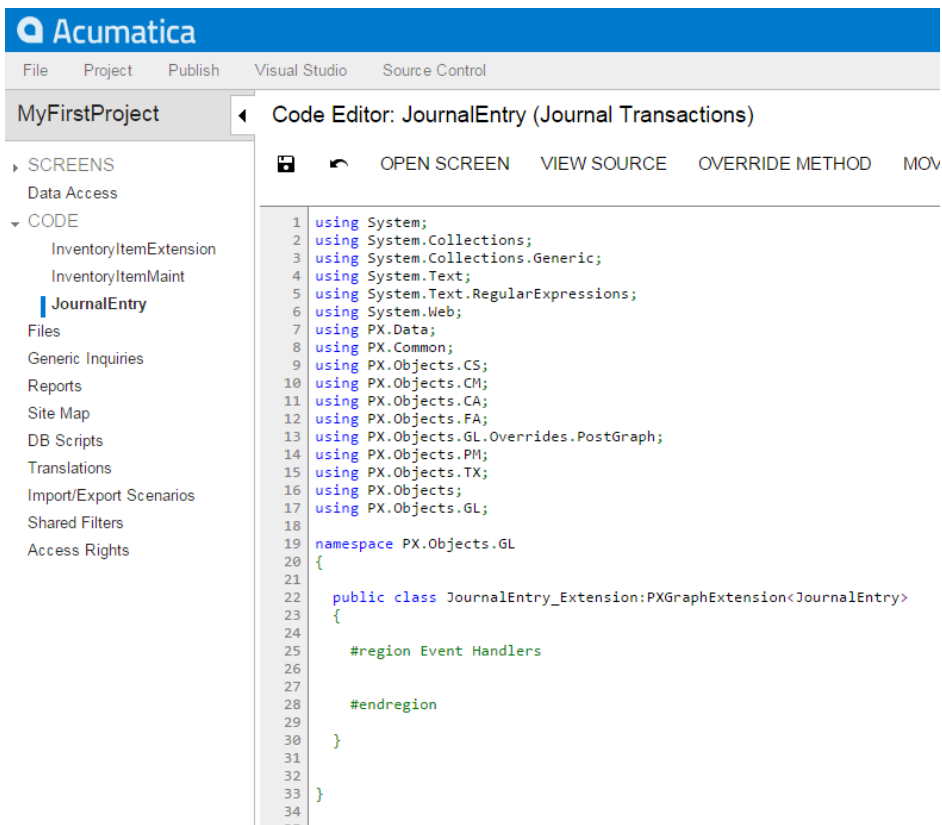


Figure: The BLC extension class generated for customization of the business logic code executed for the Journal Transactions form

To the BLC extension class for `JournalEntry`, add the code that is listed below.

```

[PXOverride]
public void PopulateSubDescr(PXCache sender, GLTran row, bool externalCall)
{
    decimal difference = (Base.BatchModule.Current.CuryCreditTotal ?? decimal.Zero)
-
    (Base.BatchModule.Current.CuryDebitTotal ?? decimal.Zero);
    if (difference != 0)
    {
        if (row.CuryCreditAmt == Math.Abs(difference))
        {
            row.CuryCreditAmt = 0;
        }
        else if (row.CuryDebitAmt == Math.Abs(difference))
        {
            row.CuryDebitAmt = 0;
        }
    }
}

```

In the code, you override the `PopulateSubDescr` method of the original BLC by adding the new method to the queue of override methods to be executed. The system invokes the base method first, and then executes the override method. The override method resets the value of the **Credit Amount** or the **Debit Amount** to 0.00.

Click **Save** in Code Editor to save the changes.

```

7 using PX.Data;
8 using PX.Common;
9 using PX.Objects.CS;
10 using PX.Objects.CM;
11 using PX.Objects.CA;
12 using PX.Objects.FA;
13 using PX.Objects.GL.Overrides.PostGraph;
14 using PX.Objects.PM;
15 using PX.Objects.TX;
16 using PX.Objects;
17 using PX.Objects.GL;
18
19 namespace PX.Objects.GL
20 {
21
22     public class JournalEntry_Extension:PXGraphExtension<JournalEntry>
23     {
24
25         #region Event Handlers
26         [PXOverride]
27         public void PopulateSubDescr(PXCache sender, GLTran row, bool externalCall)
28         {
29             decimal difference = (Base.BatchModule.Current.CuryCreditTotal ?? decimal.Zero) -
30                                 (Base.BatchModule.Current.CuryDebitTotal ?? decimal.Zero);
31             if (difference != 0)
32             {
33                 if (row.CuryCreditAmt == Math.Abs(difference))
34                 {
35                     row.CuryCreditAmt = 0;
36                 }
37                 else if (row.CuryDebitAmt == Math.Abs(difference))
38                 {
39                     row.CuryDebitAmt = 0;
40                 }
41             }
42         }
43         #endregion
44     }
45 }
46
47
48 }

```

Figure: The override method implemented in the BLC extension class

To view the result of the customization, publish the customization project and open the *Journal Transactions* form (**Finance > General Ledger > Work Area > Enter**).

Test the modified behavior of *Journal Transactions*. Add a new journal transaction, select an account, and set the **Debit Amount** to *2000.00*. Then add one more journal transaction and select a different account, and notice that the **Credit Amount** is set by default to 0.00. You must manually specify the amount for every journal entry to make the batch balanced. Otherwise, you will not be able to save the batch with the Balanced status, and you will see the appropriate error message on the **Debit Total** box.



: You should also test the case when the user first adds a transaction with a **Credit Amount**: Then the **Debit Amount** must equal 0.00 by default.

Branch	Account	Description	Subaccount	Debit Amount	Credit Amount	Transaction Description
MAIN	100000	Petty Cash USD	US-00-00-00-000	2,000.00	0.00	
MAIN	710000	Office Expense	US-00-00-00-000	0.00	0.00	

Figure: Viewing the result of the customization

Override Method That Replaces the Original Method

The override method with an additional parameter replaces the base BLC virtual method. When the virtual method is invoked, the system invokes the override method with an additional parameter of the highest-level BLC extension. The system passes a link to the override method with an additional parameter from the extension of the previous level, if such a method exists, or to the base virtual method.

You use a delegate as an additional parameter to encapsulate the method with exactly the same signature as the base virtual method. If the base virtual method contains a `params` parameter, then you should not use the `params` keyword when you declare the override method with an addition parameter. For example, to declare an `ExecuteInsert` override method with an additional parameter within a BLC extension, you can use the following code.

```
public class BaseBLCExt : PXGraphExtension<BaseBLC>
{
    [PXOverride]
    public int ExecuteInsert(string viewName, IDictionary values, object[]
parameters,
                            Func<int, string, IDictionary, object[]> del)
    {
        if (del != null)
        {
            del(viewName, values, parameters)
        }
    }
}
```

You can decide whether to call the method pointed to by the delegate. By invoking the base method, you also start the override method queue execution.

Suppose that you need to modify the logic of generating batches on release of Accounts Receivable documents.

For example, open the *Invoices and Memos* form (**Finance > Accounts Receivable > Work Area > Enter**) and explore its original behavior. Create and save an invoice, clear the **Hold** check box, click the **Release** button, and notice that a new GL batch has been generated by the system while the system releases the invoice. You can see the batch number on the **Financial Details** tab of the *Invoices and Memos* form, as the following screenshot shows.

The screenshot shows the Acumatica Accounts Receivable form for a New York invoice. The 'RELEASE' button is highlighted in red. The form displays the following details:

Type:	Invoice	Customer:	CUSTOMER1 - Customer1	Detail Total:	100.00
Reference Nbr.:	001013	Currency:	USD 1.00	Discount Total:	0.00
Status:	Open	Terms:	07D - 7 Days	VAT Taxable ...	0.00
	<input type="checkbox"/> Hold	* Due Date:	12/26/2014	VAT Exempt ...	0.00
Date:	12/19/2014	* Cash Discoun...:	12/19/2014	Tax Total:	0.00
Post Period:	11-2014	Project:	X - Non-Project Code.	Balance:	100.00
Customer Order:				Amount:	100.00
Description:				Cash Discount:	0.00

The 'Financial Details' tab is active, showing the following information:

LINK TO GL	TAX AND COMMISSION
Batch Nbr.:	0004482
* Branch:	MAIN - New York
* AR Account:	110000 - AR Trade - Local
* AR Subaccount:	US-00-00-00-000 - US Default
Customer Tax Zone:	
Customer Usage Type:	
Salesperson ID:	
Total Commissionable:	0.00
Commission Amt.:	0.00
DEFAULT PAYMENT INFO	ASSIGNED TO
Payment Method:	CHECK - Check
Workgroup:	
Card/Account No.:	
Owner:	
Cash Account:	102000 - Bank of America Chec

Figure: The batch has been generated during release of the Accounts Receivable invoice

Suppose that you need to prevent generation of batches during release of Accounts Receivable documents. You can implement the new behavior in the `Persist` method that you have to override in a BLC extension for the `JournalEntry` class. In the override method, you need to handle execution of the `Persist` method of the original BLC. Therefore, you have to implement the override method with an additional parameter, as described below.

To the BLC extension class for `JournalEntry`, add the code that is listed below. For example, you can add the override method to the `JournalEntry_Extension` BLC extension class generated by the system, as described above in [Override Method That is Added to the Override Method Queue](#).

```
[PXOverride]
public void Persist(Action del)
{
    if (Base.BatchModule.Current != null &&
        Base.BatchModule.Cache.GetStatus(Base.BatchModule.Current) ==
        PXEntryStatus.Inserted &&
        Base.BatchModule.Current.Module == "AR")
        return;
    if (del != null)
        del();
}
```

The code modifies the logic so that the system will not invoke the base virtual method if the batch is being generated from the Accounts Receivable module; otherwise, the system executes the base business logic.

To view the result of the customization, publish the customization project and open the *Invoices and Memos* form (**Finance > Accounts Receivable > Work Area > Enter**).

To test the modified behavior of the *Invoices and Memos* form, add and save a new invoice, clear the **Hold** check box, and click the **Release** button. Notice that no new GL batch is generated by the system

while it releases the invoice. Open the **Financial Details** tab. No batch number is displayed, as the following screenshot illustrates.

The screenshot shows a software interface for managing invoices. At the top, there are navigation tabs: 'FINANCE', 'DISTRIBUTION', and others. Below these, there are icons for various actions, with a 'RELEASE' button highlighted in a red box. The main area displays invoice details for 'CUSTOMER1 - Customer1'. A summary table on the right shows 'Detail Total: 100.00', 'Discount Total: 0.00', 'VAT Taxable ...: 0.00', 'VAT Exempt ...: 0.00', 'Tax Total: 0.00', 'Balance: 100.00', 'Amount: 100.00', and 'Cash Discount: 0.00'. Below the invoice details, there are several tabs: 'Document Details', 'Financial Details', 'Billing Address', 'Tax Details', 'Salesperson Commission', 'Discount Details', and 'Applications'. The 'Financial Details' tab is selected. Under this tab, there are sections for 'LINK TO GL', 'TAX AND COMMISSION', 'DEFAULT PAYMENT INFO', and 'ASSIGNED TO'. The 'Batch Nbr.' field under 'LINK TO GL' is empty and highlighted with a red box. Other fields include 'Branch: MAIN - New York', 'AR Account: 110000 - AR Trade - Local', 'AR Subaccount: US-00-00-00-000 - US Default', 'Customer Tax Zone', 'Customer Usage Type', 'Salesperson ID', 'Total Commissionable: 0.00', 'Commission Amt.: 0.00', 'Payment Method: CHECK - Check', 'Card/Account No.', 'Cash Account: 102000 - Bank of America ChecI', 'Workgroup', and 'Owner'.

Figure: No batch has been generated by the system for the Accounts Receivable Invoice

Appendix

The appendix provides some reference information relevant for this document. The additional information in this section is a useful source for readers who need some reference material that is related to system forms and tables, as well as running reports.

In this section:

- [Reports](#)
- [Form Toolbar](#)
- [Table Toolbar](#)
- [Glossary](#)

Reports

In addition to offering a comprehensive collection of reports for each module, Acumatica ERP gives you a high degree of control over each report.

A typical report form, described in [Report Form](#), lets you adjust the report settings to meet your specific informational needs. You can specify sorting and filtering options and select the data by using report-specific settings—such as financial period, ledger, and account—and configure additional processing settings for each report. The settings can be saved as a report template for later use. For details, see [To Run a Report](#) and [To Create a Report Template](#).

After you run a report, the prepared report appears on your screen. You can print the report, export the report to a file, or send the report by email.

This chapter describes a typical report form and the main tasks related to using reports.

In This Chapter

- [Report Form](#)
- [To Run a Report](#)
- [To Configure an Ad Hoc Filter on a Report Form](#)
- [To Modify an Ad Hoc Filter on a Report Form](#)
- [To Create a Report Template](#)

Report Form

Before you run a report, you set a variety of parameters on the report form. You can select a template or manually make selections that affect the information collected. Also, you can specify appropriate settings to print or email the finished report.

The following screenshot shows a typical report form.

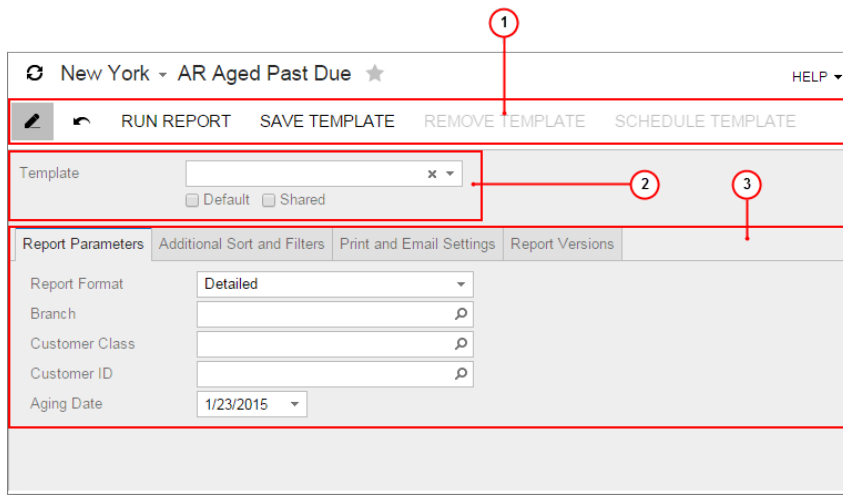


Figure: Parameters View of Report Form

1. Report Form Toolbar
2. Parameters Toolbar
3. Template Area
4. Details Area


Report Form Toolbar

The following table lists the buttons of the report form toolbar when you are configuring a report.

Button	Description
Cancel	Clears any changes you have made and restores default settings.
Run Report	Initiates data collection for the report and displays the generated report.
Save Template	Gives you the ability to save the currently selected report as a template with all the selected settings.
Remove Template	Removes the previously saved template. This button is available only when you select a template.
Schedule Template	Opens the Select Schedule Name dialog box, which you can use to schedule report processing. This button is available only when you select a template.










Select Schedule Name Dialog Box

Element	Description
Schedule	The schedule for report processing. Select an existing schedule, or leave the box blank and click OK to open the Automation Schedules (SM.20.50.20) form to create a new schedule for running the report. For more information on scheduling, see To Schedule Processing in the Acumatica ERP User Guide.
Merge Reports	A check box that indicates (if selected) that this report will be merged with the other reports selected for merging into one net report when processed.

Element	Description
	 : You can check the reports that will be merged when processed on the Send Reports (SM.20.50.60) form.
Merging Order	The number of the report in the net report.

Report Toolbar

The following table lists the buttons of the toolbar after you run the configured report.

Buttons	Icon	Description
Parameters		Navigates back to the report form to let you change the report parameters.
Refresh		Refreshes the information displayed in the report (if any data changes were made).
Groups		Adds to the report a left pane where the report structure is shown. Click a report node to highlight the pertinent data in the right pane.
View PDF / View HTML	 / 	Displays the report as a PDF, or displays the report in HTML format. The available button depends on the current report view; if you're viewing a PDF, for instance, you will see the View HTML button.
First		Displays the first page of the report.
Previous		Displays the previous page.
Next		Displays the next page.
Last		Displays the last page of the report.
Print		Opens the browser dialog box so you can print the report.
Send		Opens the Email Activity dialog box, which you use to send the report file (in the chosen format) to the specified email address.
Export		Enables you to export the data in the chosen format (Excel or PDF).

Template Area

Use the elements in this area to select an existing template and then use the template, share it with other users, or use it as your default report settings.

The Template area elements, which are available for all reports, are described in the following table.

Template Area Elements

Element	Description
Template	The template to be used for the report. If any templates were created and saved, you can select a template to use its settings for the report.

Element	Description
Default	A check box that indicates (if selected) that the selected template is marked as the default one for you. A default template cannot be shared.
Shared	A check box that indicates (if selected) that the selected template is shared with other users. A shared template cannot be marked as the default.
Locale	A locale that you select to indicate to the system that the report should be prepared with the data translated to the language associated with this locale. This box is displayed if there are multiple active locales in the system. For details, see Locales and Languages .

Report Parameters Tab

The **Report Parameters** tab includes sections where you can specify the contents of the report depending on the current report and vary in the following regards:

- How many elements and which elements are available on a particular report
- Whether elements contain default values
- Whether specific elements require values to be selected
- Whether elements may be left blank to let you display a broader range of data

Additional Sort and Filters Tab

The **Additional Sort and Filter** tab contains additional sorting and filtering conditions:

- **Additional sorting conditions:** Defines the sorting order. You can add a line, select one of the report-specific properties, and select the *Descending* or *Ascending* sort order for the column.
- **Additional filtering conditions:** Defines the report filter. You can add a line, select one of the report-specific properties, and define a condition and its value. The list of conditions include one-operand and two-operand conditions. To create a more complicated logical expression, you can use brackets and logical operations between brackets. For more information on creating filters, see [Creation of Ad Hoc and Reusable Filters](#) in Acumatica ERP User Guide. For detailed procedures on using ad hoc filters, see [To Configure an Ad Hoc Filter on a Report Form](#) and [To Modify an Ad Hoc Filter on a Report Form](#).

Print and Email Settings Tab


If you plan to print the report or save the report as a PDF, select the appropriate settings in the **Print Settings** area.

Print Settings Section

Element	Description
Deleted Records	Selects the visibility of the data deleted from the database.
Print All Pages	Causes all pages of the report to be printed.
Print in PDF format	Displays the report in PDF format.
Compress PDF file	Indicates that the system will generate a compressed PDF.
Embed fonts in PDF file	Indicates that the system will generate the PDF with fonts embedded.

If you plan to send the report as an email, in the **Email Settings** area, specify the format in which the report will be sent, as well as the email subject, the recipients of copies of the report, and the email account of the recipient.

Email Settings Section

Field	Description
Format	The format (<i>HTML</i> , <i>PDF</i> , or <i>Excel</i>) in which the report will be emailed.  : Merge function for reports in Excel format is not supported. If you want to merge a report with other reports and send an aggregated report by email, you should select either the HTML or PDF format for the report.
Email Account	The email address of the recipient.
CC	An additional addressee to receive a carbon copy (CC) of the email.
BCC	The email address of a person to receive a blind carbon copy (BCC) of the email; an address entered in this box will be hidden from other recipients.
Subject	The subject of the email.

Report Versions Tab

If the report has multiple versions, you can select one of them.

Report Versions Tab Toolbar

Button	Description
Refresh	Refreshes the list of report versions.
Select	Temporarily activates the selected report version.




Report







Once you click **Run Report**, the prepared report appears on your screen. You can print the report, export the report to a file, or send the report by email.

The prepared report is displayed in the report view of the report form. For more information about setting up the report parameters and the parameters view of the report form, see [Report Form](#).

Report Toolbar

The following table lists report toolbar buttons.

Buttons	Icon	Description
Parameters		Navigates back to the report form to let you change the report parameters.
Refresh		Refreshes the information displayed in the report (if any data changes were made).
Groups		Adds to the report a left pane where the report structure is shown. Click a report node to highlight the pertinent data in the right pane.

Buttons	Icon	Description
View PDF / View HTML	 / 	Displays the report as a PDF, or displays the report in HTML format. The available button depends on the current report view; if you're viewing a PDF, for instance, you will see the View HTML button.
First		Displays the first page of the report.
Previous		Displays the previous page.
Next		Displays the next page.
Last		Displays the last page of the report.
Print		Opens the browser dialog box so you can print the report.
Send		Opens the Email Activity dialog box, which you use to send the report file (in the chosen format) to the specified email address.
Export		Enables you to export the data in the chosen format (Excel or PDF).

Form Toolbar

The form toolbar, available on most forms, is located near the top of the form, under the form title bar (see the screenshot below). The form toolbar may include standard and form-specific buttons.

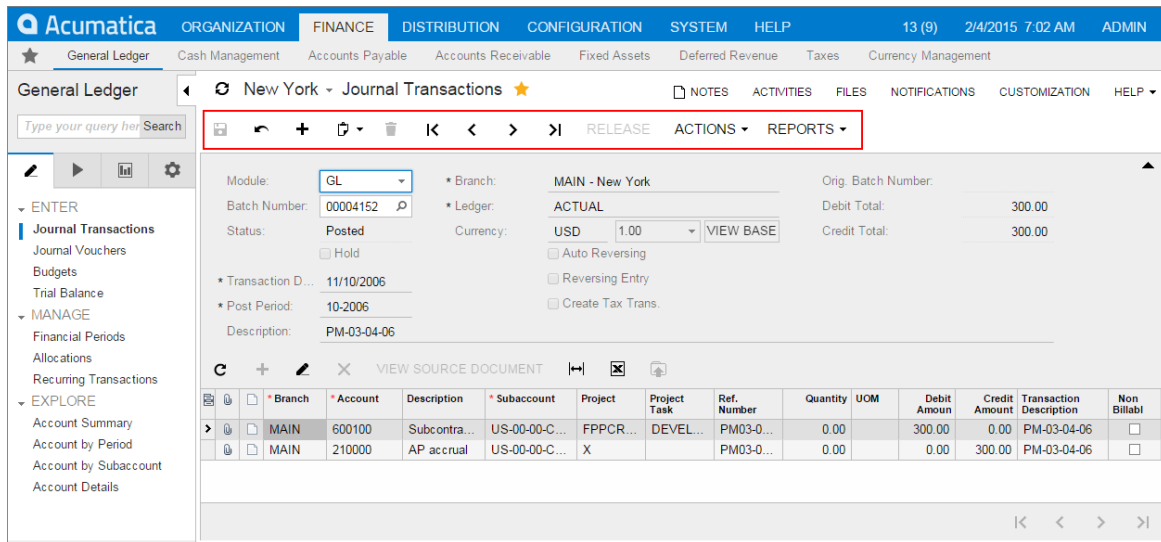


Figure: Form toolbar












You use the standard buttons on the form toolbar to navigate through objects and entities that were created by using the current form, insert or delete an object or entity, use the clipboard, save the data you have entered, or cancel your work on the form.

In addition to standard buttons, a form toolbar on a particular form may include form-specific buttons. These buttons usually provide navigation to other forms, take specific actions, and perform modifications or processing related to the functionality of the form.

Standard Form Toolbar Buttons

The following table lists the standard buttons of the form toolbar. A form toolbar may include some or all of these buttons.

Standard Form Toolbar Buttons

Button	Icon	Description
Save		Saves the changes made to the object or entity.
Cancel		Depending on the context, does one of the following: <ul style="list-style-type: none"> Discards any unsaved changes you have made to objects or entities and retrieves the last saved version. Clears all changes and restores the default settings.
Add New Record		Clears any values you've specified on the form, restores any default values, and initiates the creation of a new object or entity.
Clipboard		Provides options to do the following: <ul style="list-style-type: none"> <i>Copy</i>: Copy the selected object or entity to the clipboard. <i>Paste</i>: Paste an object, entity, or template from the clipboard. <i>Save as Template</i>: Create a template based on the selected object or entity. <i>Import from XML</i>: Import an object, entity, or template from an .xml file. <i>Export to XML</i>: Export the selected object or entity to an .xml file. <p>For more information on templates and copy-and-paste operations in Acumatica ERP, see Using Forms. For more information on importing and exporting .xml files, see System-Wide Actions in Acumatica ERP in the Acumatica ERP User Guide.</p>
Delete		Deletes the currently selected object or entity, clears any values you've specified on the form, and restores default values. <p> : You can delete a document that is not linked with another document.</p>
Go to First Record		Displays the first object or entity (in the list of objects or entities of the specific type) and its details.
Go to Previous Record		Displays the previous object or entity and its details.
Go to Next Record		Displays the next object or entity and its details.
Go to Last Record		Displays the last object or entity (in the list of objects or entities of the specific type) and its details.
Schedules		Gives you the ability to schedule the processing. For more information, see To Schedule Processing topic in the Acumatica ERP User Guide.

Inquiry Form Toolbar Buttons

Acumatica ERP inquiry forms present the data in a tabular format. These forms can be designed by a user with the appropriate access rights by using the Generic Inquiry tool (for details, see [Managing Generic Inquiries](#) in the Acumatica ERP User Guide), or can be initially configured in your system. A toolbar of an inquiry form contains both the standard form toolbar buttons (described in the table above) and additional buttons described below.




Button	Icon	Description
Fit to Screen		Expands the form to fit on the screen and adjusts the column widths proportionally.
Export to Excel		Exports the data to an Excel file. For more information, see Integration with Excel in the Acumatica ERP User Guide.
Filter Settings		Opens the Filter Settings dialog box, which you can use to define a new filter. After the filter has been created and saved, the corresponding tab appears on the table. For more information about filtering, see Filters .

Table Toolbar

Each table on an Acumatica ERP form, tab, or dialog box has a table toolbar, which contains the search box and buttons you can use to work with the details or objects of the table.

The table toolbar, shown in the following screenshot, can include the following sections:

- *Action section*: Contains buttons that are specific to the table, standard buttons that most table toolbars have, and the search box.
- *Footer section*: Displays navigation buttons if there are too many details or objects (that is, table rows) to fit on one page.

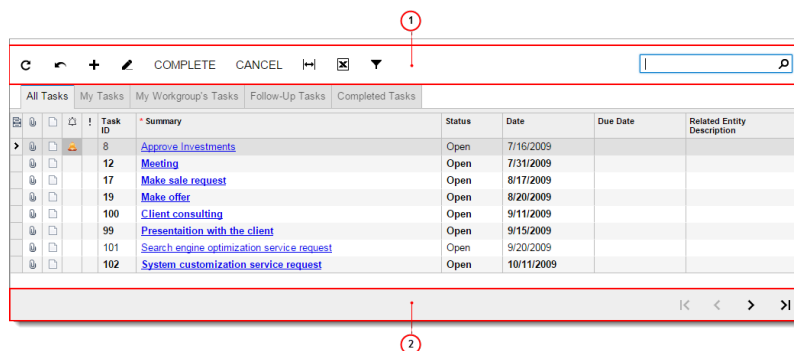


Figure: Table toolbar sections










1. Action section
2. Footer section

Action Section of Table Toolbar

The action section, commonly located at the top of a table, can contain standard and table-specific buttons. If a table toolbar includes table-specific buttons, they are described in the form reference help topic.

The following table describes the standard table toolbar buttons. A table toolbar may include some or all of those buttons.


Standard Table Toolbar Buttons

Button	Icon	Description
Refresh		Refreshes the data in the table.
Switch Between Grid and Form		Controls how the elements are displayed on the form: in a table (grid) with rows and columns; or as separately arranged elements for one table row on a form, with navigation tools you use to move between rows.
Add Row		Appends a new blank row to the table so you can define a new detail or object. A row may contain some default values.
Delete Row		Deletes the selected row.
Fit to Screen		Adjusts the table to the screen width and makes the column width proportional.
Export to Excel		Exports the data in the table to an Excel file. For more information, see Integration with Excel in the Acumatica ERP User Guide.
Filter Settings		Opens the Filter Settings dialog box, which you can use to define a new filter. After the filter is created and saved, the corresponding tab appears on the table. For more information about filtering, see Filters .
Load Records from File		Opens the File Upload dialog box, described in detail below, so you can locate and upload a local file for import. You can use this option to import data from Excel spreadsheets (.xlsx) and .csv files. For the detailed procedure, see To Import Data from a Local File to a Table .
Search		A box in which you can type a word, part of a word, or multiple words. As you type, the system filters the contents of the table to reflect the string you have typed.

File Upload Dialog Box

The **File Upload** dialog box gives you the ability to upload a file of one of the supported formats (.csv or .xlsx) and import data from the file. You import the file by using the **File Upload** dialog box, specify the import settings by using the **Common Settings** dialog box, and then match the columns in the imported file to the columns in the Acumatica ERP table to which you import data by using the **Columns** dialog box.


Element	Description
File Upload dialog box	
File Path	The path to the file you want to upload. To select the file, click Browse , and then find and select the file you want to upload.
The dialog box has the following button:	
Upload	The button you click to upload the selected file and to open the Common Settings dialog box.


Element	Description
Common Settings dialog box	
Separator Chars	The character that is used as the separator in the imported file. Specify the separator character if the imported file uses an atypical separator. This box appears on the interface only when you import data from a .csv file.
Null Value	Optional. The value that is used to mark an empty column in the imported file. Specify the null value if the value in the imported file differs from the standard empty value.
Encoding	The encoding that is used in the imported file. This box appears on the interface only when you import data from a .csv file.
Culture	The regional format that has been used to display the time, currency, and other measurements in the imported file..
Mode	<p>The mode defining which rows of the uploaded file will be imported into the table. The following options are available:</p> <ul style="list-style-type: none"> • <i>Update Existing</i>: The rows already present in the table will be updated, and the rows not present in the table will be added. • <i>Bypass Existing</i>: Only the new rows that are not present in the table will be imported. The rows that are already present in the table will not be updated. • <i>Insert All Records</i>: All the rows from the file will be imported into the table. <p> : If you select this option, you may get duplicated rows because the system won't check for duplicates when importing rows from the file.</p>
The dialog box has the following buttons:	
OK	Closes the dialog box and opens the Columns dialog box.
Cancel	Closes the dialog box without importing the data from the file.
Columns dialog box	
Column Name	The name of the column in the uploaded file.
Property Name	The name of the corresponding column in the table in Acumatica ERP.
The dialog box has the following buttons:	
OK	Closes the dialog box and import the selected file.
Cancel	Closes the dialog box without importing the data from the file.

Shortcut Menu

Right-clicking within the rows of a table opens a shortcut menu. The commands you see in the menu, which depend on the table you are working with, are mostly duplicates of actions on the table toolbar, but they offer easier access to them. The unique menu commands are described in the following table.

Shortcut Menu Commands





Option	Icon	Description
Clear Column Filter		Clears the simple filter that you have applied to the selected column.

Option	Icon	Description
Filter by This Cell Value		Filters the data in the table by the value of the selected cell. For details, see To Use Simple Filters .

Footer Section of Table Toolbar

If a particular table has too many details (table rows) to fit on one page, you use the footer to browse the table pages.

Standard Table Navigation Buttons

Element	Icon	Description
Go to First Page		Displays the first page of the table.
Go to Previous Page		Displays the previous page of the table.
Go to Next Page		Displays the next page of the table.
Go to Last Page		Displays the last page of the table.

Glossary

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A	
account	A General Ledger entity that holds a detailed record of similar transactions involving a particular item, such as a source of cash or a recipient of income. supports several account types: <i>Asset</i> , <i>Liability</i> , <i>Income</i> , and <i>Expense</i> . All of a company's accounts are listed on its chart of accounts. In Acumatica ERP, accounts are used with subaccounts, and at system setup, you choose whether account identifiers should be composed of segments. See also General Ledger , chart of accounts , subaccount , segment .
account class	A user-defined class that is used to group related accounts of the same type and that can be used in reports and inquiries for convenient grouping, sorting, and filtering of information associated with accounts. For example, you can create a class for long-term liabilities and another for middle-term liabilities.
Accounts Payable (AP)	An Acumatica ERP module that provides functionality for efficient management of your company's vendors, primarily for goods and services purchased by your company. See also module , vendor .
Accounts Receivable (AR)	An Acumatica ERP module that provides functionality for efficient management of your company's customers, primarily for goods and services sold by your company. See also module , customer .
home page	The first page a user sees upon signing in to the web site. By default, the home page displays the documentation list, but it can be configured to display the home page dashboard. See also dashboard .

adjustment period	A financial period with the same start and end dates (and, thus, a duration of zero days) that is the last period in the financial year. An adjustment period can be added only at initial setup. On data entry forms, an adjustment period is available only when you directly type it in the Fin. Period box. See also financial period , financial year .
aging period	One of a group of time intervals used for sorting a company's open AR documents by age or its open AP documents by days outstanding. For example, aging periods may be defined as follows: -999 to 0 days past due, 1 to 30 days past due, 31 to 60 days past due, and 61 to 91 days past due.
aggregating value	A special value of a segment of a subaccount that indicates the sum of appropriate budget articles whose subaccounts have other values in this segment. Aggregating values let you build a hierarchical structure of budget articles. See also segment , subaccount .
allocation template	A group of settings, saved as a whole, used in to help you automate allocations; allocation templates can be defined using the Allocations (GL.20.45.00) form. You can create an allocation source by using input masks to specify multiple accounts and subaccounts at once, and you can set up rules for distributing the source amount over multiple accounts. See also input mask , account , subaccount .
AP bill	A document created for each vendor invoice that includes information about the vendor, location, and currency used for the transaction. A bill may contain either line items or one summary line with a tax category specified. Some taxes are calculated for the entire document, and some are calculated for each line item, depending on each tax's <i>Calculate On</i> setting on the Taxes (TX.20.50.00) form. See also vendor , location , tax category , tax .
AP check	A payment document created for a vendor. A separate AP check may be required for each AP document, depending on your selected configuration options. AP checks should always have zero unapplied balance; that is, the amount of a check should be exactly the amount of the bill or bills it pays for. See also vendor .
AP credit adjustment	A clearing document created to correct errors on an existing bill, or in response to a vendor's overdue charges or a debit memo. Posting a credit adjustment increases the balance of Accounts Payable.
AP debit adjustment	An AP document created on the Bills and Adjustments (AP.30.10.00) form for a vendor refund. It may be applied to any bills of the same vendor. Discounts cannot be taken on debit adjustments. See also vendor .
approval	The act of formally giving permission for a document to proceed to the next step in its workflow. An organization may require documents, such as purchase orders or expense claims, to be approved by authorized persons before they may be paid. Also, you can require that wiki articles be approved before they may be published. In Acumatica ERP, you can configure approvals by assigning documents to specific persons for approvals and by giving only authorized persons access to certain processing forms. See also form .
AR credit memo	A clearing document created for damaged goods or a previous overcharging invoice. A credit memo may have one summary line or

	multiple line items. A credit memo may be applied against invoices, debit memos, and overdue charges. See also overdue charges .
AR debit memo	A document that adjusts the amount in one or several previous undercharging invoices. It doesn't contain a direct reference to any original invoices; if needed, you can reference the original invoice in the <i>Description</i> box. Debit memos may be numbered differently from invoices.
AR invoice	An itemized request for payment for goods sold or services rendered. An invoice includes the customer information, location, currency, and any reference number in the original customer document. The due date of the document is calculated based on the credit terms associated with the customer. An AR invoice may have a single summary line or multiple line items. For each line, a tax category may be specified. See also customer , location , credit terms , tax category .
assignment map	A structure representing the hierarchy of workgroups involved in processing or approval. See also approval .
assignment rules	A set of rules you can configure—based on the properties of an entity (such as a lead or case) or a document (such as an expense claim, sales order, or purchase order)—to enable automatic assignment to appropriate employees for processing or approval. For the selected entity type, you facilitate automatic assignment of entities to workgroups and to particular users by creating an assignment map and rules based on properties of the entity or document. See also approval , assignment map .
attribute	A custom element that your company can add to forms to keep additional information about products, leads, customers, inventory items, and other entities. Attributes—which are used by the Inventory, Customer Management, and Common Settings modules—allow you to gather details that are meaningful for your business. See also form , customer , inventory item , Inventory , module .
attribute class	A grouping of your company's leads, opportunities, customers, or cases by a specific set of attributes. For more information, see Attribute Classes. See also attribute , attribute class .
authentication	The process by which establishes a potential user as valid and grants access to the system. A user must use a valid user name and password pair for successful authentication. See also user .
authorization	The process by which verifies whether a user has sufficient access rights to particular forms, elements, and actions. The system makes this determination for a user who has successfully signed in, based on the roles assigned to the user and the restriction groups that include the user as a member. See also user , form , role , restriction group .
auto-reversing batch	A batch for which the General Ledger module automatically creates another batch that reverses debit and credit entries into the next financial period; the debit entry is reversed as a credit entry and vice versa. Auto-reversing entries are used to reverse some period-end adjustments. See also batch , General Ledger , module , financial period .
automation definition	A complete set of all automation steps defined for all forms created using the Automation Definitions (SM.20.50.10) form. You can use definitions created before system upgrades (or before major changes in

	automation steps) as backups of various states of automation in your system. See also automation steps , automation definition .
automation schedule	A schedule defined for an processing form to direct the system to perform specific processing periodically. You can create automation schedules using the Automation Schedules (SM.20.50.20) form. See also automation schedule
automation steps	Steps to be executed for specific records or objects on a particular form, depending on the record or object properties. Automation steps allow you to extend the functionality of data entry and processing forms. You can add new object statuses, associate statuses with certain actions, and enable or disable actions, depending on object properties. See also form .
B	
bank account	A cash account associated with a specific bank (which in is defined as a vendor) and with one or more linked clearing accounts. Bank charges can be configured as entry types associated with the bank account. Bank accounts generally require periodic reconciliations to be performed. See also vendor , clearing account , entry type , reconciliation .
base currency	The currency of the environment in which the company generates and expends cash. A base currency can be the only currency used in the system or one of multiple currencies used. The base currency is the default currency for recording transactions, budgets, and other GL data, and it is used for reporting, income statement, and balance sheet calculations. For General Ledger accounts denominated to a foreign currency, maintains the history of transactions and balances in both the currency of denomination and the base currency. See also General Ledger , account .
base price	A price for an item set with respect to the base unit, expressed in the base currency and offered to customers of the base price class (those not associated with any specific price class by default). Base prices can be defined and maintained directly, following your company's pricing policy, via the Inventory or Sales Orders module. See also base unit , base currency , customer , Inventory .
base unit	The unit of measure in which a particular item is tracked from the moment it is received at a warehouse or produced at one of your facilities until it is picked for shipping. Generally, the base unit is the smallest unit defined in the system for a particular item. Also, the base unit is the unit of measure used for calculating item costs. See also warehouse .
batch	A set of related transactions or journal entries that are in the same currency, refer to the same ledger, and occur in the same financial period. A batch's debit total and credit total are calculated over all the transactions. Some batches are generated by the system automatically, such as those implementing revaluations. Only balanced batches (those for which the credit total equals the debit total) can be released and posted to the specified ledger. See also journal entry , financial period , revaluation
batch control total	A user-entered total that is used for batch status validation if it is enforced in your system—that is, if the Validate Batch Control Totals

	<p>on Entry check box on the <i>General Ledger Preferences</i> (GL.10.20.00) is selected. A batch can be saved with the <i>Balanced</i> status only if the user types into this box a value that is equal to the batch's debit total and credit total amounts. We recommend that you use batch control total validation to reduce data input errors. See also <i>batch</i>.</p>
budget article	<p>An account-subaccount pair recorded in a budget ledger with a budget amount. A budget article has subarticles if its subaccount has at least one aggregating value in any segment. See also <i>account</i>, <i>subaccount</i>, <i>aggregating value</i>, <i>segment</i>.</p>
business account	<p>A set of information about one of your company's vendors or customers, including its locations, contacts, and payment and shipping options. Acumatica ERP uses this information, which is specified on data entry forms, in a variety of modules. Your company also has a business account to record its own locations, contacts, and shipping settings. See also <i>vendor</i>, <i>customer</i>, <i>location</i>, <i>module</i>.</p>
C	
cash account	<p>A special type of General Ledger account used to record various monetary transactions in a specific currency. You can specify entry types and payment methods with which the cash account is associated. Each cash account is assigned to a specific branch. See also <i>General Ledger</i>, <i>account</i>, <i>entry type</i>, <i>payment method</i>.</p>
cash-in-transit account	<p>An account used for cash that is being moved from one currency to another. Because cross-rates are not used in Acumatica ERP, currency conversion is performed via the base currency with the use of the cash-in-transit account and subaccount. See also <i>account</i>, <i>base currency</i>, <i>subaccount</i>.</p>
cash discount	<p>A deduction from the total payable amount, allowed if the amount owed is paid within a specified time period on or before a due date. Cash discounts available for your organization are defined by credit terms assigned to vendors, and cash discounts available for customers are defined by credit terms assigned to customers. See also <i>credit terms</i>, <i>vendor</i>, <i>customer</i>.</p>
Cash Management	<p>An Acumatica ERP module that manages cash and bank accounts, cash transactions (including funds transfer), and bank statement reconciliations. The module is integrated with the Accounts Payable and Accounts Receivable modules for smooth payment processing. See also <i>module</i>, <i>cash account</i>, <i>bank account</i>, <i>funds transfer</i>, <i>reconciliation</i>.</p>
chart of accounts	<p>A listing of the accounts in the system to which you will record accounting transactions. The chart of accounts, which you maintain in Acumatica ERP via the <i>Chart of Accounts</i> (GL.20.25.00) form, consists of balance sheet accounts (assets and liabilities) and income statement accounts (income and expenses). The chart of accounts should follow national and industry standards while also reflecting the operations of your company. See also <i>account</i>.</p>
clearing account	<p>A cash account that temporarily holds customer payments included in a deposit. Once the money is actually deposited to the bank and the deposit is released, a batch of transactions will be generated to move the payment amounts from clearing accounts to the bank account and to record the charges incurred as expenses. See also <i>cash account</i>, <i>customer</i>, <i>deposit</i>, <i>batch</i>, <i>bank account</i>.</p>

combined subaccount	A subaccount that can be combined from multiple involved subaccounts, based on rules you create, for certain transactions; use elements with labels such as Combine Subaccount From to set up such rules. A box for a combined subaccount displays a subaccount mask, such as ----.---.----.----, in accordance with the segmented structure of subaccounts defined for your company. For each segment, you can specify one of the involved subaccounts as the source of the segment value. See the Combined Subaccounts article for more information. See also subaccount , combined subaccount .
commission	A payment made to a salesperson for goods and services sold. Commission is calculated once in a commission period based on total invoice amounts or payments received, depending on your configuration choice. For each document, calculates the commission amount as a percentage. For an invoice, the commission may be split among multiple salespersons.
Common Settings	A module used to control global system settings. Some configuration settings—such as company information, segmented keys, and numbering sequences—must be provided during initial system setup and cannot be modified later. Other information, such as credit terms and logistics settings, can be added at any time. See also module , segmented key , numbering sequence , credit terms .
consolidation	The process of combining separate accounting data into one set of data by importing data from subsidiaries to the parent company. Whether your organization is a parent company or a subsidiary of a larger company, the General Ledger module enables you to prepare and consolidate the data into one consolidation ledger in the parent company. See also General Ledger , module .
consolidation data	The data imported to the parent company (as GL batches with system-generated descriptions) to enable consolidation. Consolidation data prepared by consolidation units is available via the SOAP gateway as a set of GLConsolRead instances. See also General Ledger , batch , consolidation .
consolidation mapping	The process of matching accounts and subaccounts in a subsidiary against those of the parent company for consolidation. Mapping of subaccounts can be performed across subaccount segments. See also account , subaccount , segment .
credit terms	Conditions and stipulations used by vendors in their relations with your company and by your company in its relations with customers when any outstanding balance is paid. Credit terms include an installment option (one payment or multiple installments), a payment schedule, and terms for cash discount (for only the single-installment option). Also, credit terms can be used as a schedule for overdue charges. For more information about setting up credit terms in Acumatica ERP, see Credit Terms. See also vendor , customer , cash discount , credit terms .
Currency Management	An Acumatica ERP module that lets you define multiple currencies, enabling foreign currency transactions throughout other modules. You can maintain the lists of currencies, track exchange rate fluctuations, and perform periodical revaluations. See also module , revaluation .
currency rate	The rating of one currency valued against another. You use the Currency Rates (CM.30.10.00) form to enter foreign currencies' exchange rates and the base currency rates. Rates for each rate type

	are specified with respect to the base currency and are recorded to the database with the type of arithmetic operation required to apply the rate. Each record contains the rate and the date when it becomes effective. The rate is used for currency conversions for documents that have later dates until a new rate is recorded. See also currency rate , base currency .
customer	One of your company's trade debtors. Acumatica ERP lets you set up default values for individual customers, customer classes, and customer documents to help make data entry easier and less error-prone. When you enter a new invoice for a customer, calculates the due date, discount date, and amount automatically, based on its credit terms. Tax settings are by default those of the tax zone associated with the customer location. See also customer class , credit terms , tax , tax zone .
customer class	A group of settings that provides default values when users create new customer accounts, thus saving them time. You divide customers into classes based on the types of goods or services they purchase from you. For more details, see Customer Defaults and Overrides. See also customer .
Customer Management	An Acumatica ERP module that helps your company set up customer service based on contracts and effectively track service issues reported by customers. Also, the module provides tools to help salespeople generate quality leads, track and analyze sales opportunities, and manage marketing campaigns by sending personalized emails in bulk. See also module , customer .
customer price class	A group of customers that may be offered special prices because of their buying habits. All customers of the same customer price class are charged the same price for the same item, and you can set different prices for the same item for different customer price classes. See also customer .
customer statement	A complete record of the customer's invoices, debit and credit memos, payments, prepayments, and overdue charges for a specific period. A statement includes all new activity for a statement cycle, from the previous statement date to the current statement date. Any open debit items from prior periods are included in the statement, grouped by days outstanding. See also statement cycle , prepayment .
D	
dashboard	An interface that organizes and presents key information in a format that users can interpret easily. A dashboard can be configured for the home page and for each module web page. System administrators can design a set of company-specific template dashboards. A user can modify any of them or create from scratch a set of personalized dashboards that display information tailored to job and information needs. For details, see Dashboards. See also user , module , dashboard .
deferral code	A code used in Acumatica ERP to configure how revenues and expenses are recognized. If a line amount in an invoice or a bill should be recognized over several periods, a deferral code (of the revenue or expense type, respectively) is assigned to this line.

deferral schedule	A number of related transactions automatically generated for the documents whose lines have deferral codes assigned. See also deferral code .
Deferred Revenue	An Acumatica ERP module that stores definitions of deferral codes, while allowing you to view and edit deferral schedules generated for AP and AR documents and recognize parts of deferred amount according to these schedules. See also module .
deposit	<ol style="list-style-type: none"> 1. An instance of physically placing money in a bank. 2. In Acumatica ERP, an internal document created using the Bank Deposits (CA.30.50.00) form. Batch deposits are used to group customer payments deposited to the bank in bulk. When you enter payments intended for such deposits into Acumatica ERP, they are recorded to special clearing accounts, which temporarily hold payments drawn from customers' AR accounts. Before you make a deposit to the bank, you create a deposit in Acumatica ERP, list the payments and cash to be deposited, and print a deposit slip. After the money was actually deposited, the deposit can be corrected to contain only payments accepted by the bank and released. On the deposit's release, a batch of transactions will be generated to move the payment amounts from clearing accounts to the bank account and to record the charges incurred as expenses. <p>See also deposit, batch, customer, clearing account, bank account.</p>
discount	A means of reducing sales prices. Acumatica ERP allows your company to configure various types of discounts applicable to sales orders and intended to attract customers: document-level discounts that are subtracted from the document total, item-level discounts that apply to a document line, and flat-price discounts, which are special discounted prices that depend on the quantities of goods purchased. See also customer .
E	
entry type	A user-defined type of transaction (which can be a cash receipt or cash disbursement) entered by using the Cash Management module. Entry types are used to categorize cash transactions. See also Cash Management , module .
event	An activity that has a specific start time and duration. You create events for all or several users and invite users, leads, contacts, and customers as attendees. You can send invitation or rescheduling emails to attendees.
F	
Favorites	Links to the forms the particular user accesses most frequently. Any user can create his or her own list of favorites for personal use.
financial period	A part of a financial year defined by its start date and end date. A financial year can be divided into monthly, bimonthly, quarterly, or custom-defined periods. For each next new year, generates financial periods in accordance with initial system settings. We recommend that you not change financial settings in once transactions have been posted to any of the periods. See also financial year .

financial year	A time interval used for calculating annual financial statements. The year is defined by its start date, which you determine, and lasts 12 months. For example, the U.S. government's financial year begins on October 1 of the previous calendar year and ends on September 30 of the year that gives the financial year its number. A financial year consists of a number of financial periods and may include an additional adjustment period. See also financial period , adjustment period .
FOB point	A destination at which the vendor delivers the goods to be loaded to the transportation provided by the carrier. The customer covers the freight and other expenses for the cargo from a FOB (meaning <i>freight on board</i>) point. See also vendor , customer .
foreign currency translation	The process of restating the account balances in a reporting currency. For accounts denominated to currencies other than the reporting currency, the balances expressed in the base currency are recalculated to the reporting currency. For more details, see Overview of Translations. See also account , base currency .
form	A screen in Acumatica ERP that lets you, using various actions and elements, enter needed data and perform functions that are key to the capabilities of the module. In most modules, forms are grouped into the following categories: Data Entry, Processing, Inquiries, Maintenance, and Setup. See also module .
form toolbar	A toolbar present on most forms with data navigation and processing actions that apply to the entire form. For example, its actions allow you to cancel or save changes you've made, to insert or delete objects, or to navigate through the objects created via the form. See also form .
funds transfer	A transaction that moves an amount from one cash account to another, with related service charges. Funds can be transferred between accounts denominated to different foreign currencies in two steps, using the cash in transit account for currency conversion to the base currency and from the base currency. See also cash account , cash-in-transit account , base currency .
G	
General Ledger (GL)	An Acumatica ERP module that serves as the central application where all financial information is collected for analyzing, summarizing, and reporting. You use the module to set up your company's financial structure through the chart of accounts and subaccounts, collect information through transactions entered by users and imported from other modules, and prepare data for generating various financial statements. See also module , chart of accounts , subaccount .
H	
historical rate	An exchange rate for the foreign currency with respect to the base currency that was effective during a certain past period. The General Ledger module uses historical rates for foreign currency translations and "past-date" transactions. See also base currency , General Ledger , module .
I	

inline editor	An editing tool you can use to edit a section of a Wiki article or its full text. The inline editor contains both a text box (which contains the text of the applicable section) and a Formatting toolbar.
input mask	A mask implemented to govern what a user may enter into a box, so that the required format is used on data entry forms. Masks are used, for example, for phone numbers, postal codes, and tax registration IDs. Moreover, input masks can be created using regular expressions to validate entered values, since the values for some elements must follow not only input format requirements but also specific rules. For more details, see Input Validation Options .
integration services	The powerful capabilities, provided as part of the Integration module, that allow you to filter and import data from external sources, converting it into internal format, and configure data synchronization between and third-party applications to be performed on schedule. Also, you can configure data export with conversion to required formats. See also module .
inventory item	A stock or non-stock item defined and tracked in Acumatica ERP. The record's unique identifier, Inventory ID , as with other identifiers in Acumatica ERP, can be segmented, with special meaning assigned to each segment. (The INVENTORY key is used to configure inventory IDs.) Well-designed inventory IDs can help you sort and group items in operational and management reports. See also stock item , non-stock item .
Inventory	An Acumatica ERP module that provides real-time access to item availability data configured in accordance with your company's policies. The Inventory module lets you maintain a perpetual inventory system as well as performing physical inventories, which can be performed as full inventory and by cycles. You can use subitems as an additional means of tracking special types of inventory items, and you can track inventory items by either lot or serial numbers and expiration dates. (See lot serial numbers for more details.) Advanced functionality of the module includes flexible posting settings, multiple warehouses with multiple specialized locations, and automatic replenishments. See also module , subitems , inventory item , lot or serial numbers .
inventory price class	A class used to group inventory items by the method of their price calculation. An inventory price class may include items of one or more item classes. See also inventory item , item class .
item class	A class used to group stock or non-stock items with similar properties and to provide default settings for new items. See also stock item , non-stock item .
J	
journal entry	A record of debit or credit to any account in General Ledger. Journal entries (or transactions), which are added in batches, must follow the generalized double-entry rule: The debits total must be equal to the credits total through all the entries in a batch. The batch contains the date, the accounts and subaccounts to be debited, the accounts and subaccounts to be credited, and the debit or credit amounts for each transaction. A batch of journal entries can be marked as recurring or auto-reversing. See also account , General Ledger , batch , subaccount , auto-reversing batch .

K	
kit	An inventory item that consists of other stock or non-stock items as components and requires assembling (or packaging) to become a salable good. You enter a kit as a stock or non-stock item using either the Stock Items (IN.20.25.00) or #unique_265 (IN.20.20.00) form and select the Is a Kit option. You can specify the kit's components (with their quantities) using the Kit Specifications (IN.20.95.00) form. A kit may include a number of stock and non-stock components. See also inventory item , non-stock item , stock item .
L	
landed costs	All extra costs—beyond the prices at which the goods are purchased from vendors—associated with acquiring products and “landing” them at one of your company's locations. These costs might include customs duties, handling fees, freight charges, value-added taxes, and other costs for a particular product. In Acumatica ERP, you can define these costs via the Landed Cost Codes (PO.20.20.00) form. See also vendor , tax .
location	<ol style="list-style-type: none"> 1. One of multiple places of business for a particular company. Each location is assigned to a tax zone and, as a business entity, may have a separate tax registration ID from that of the main location of the company. 2. A warehouse location. <p>See also tax zone, warehouse location.</p>
location table	The list of a particular warehouse's locations and their properties. Use the location table to configure your warehouse to fit the logistical processes established in your company. For each location table, you can specify whether to include the quantities of stock items stored at this location in the quantity of available items calculated for the warehouse, whether to cost the inventory on this location separately, what inventory operations are allowed for the location, and what the location's pick priority is. Users can consult this table for reference when they're creating receipts, issues, or transfers. See also warehouse , stock items .
lot or serial numbers	Identifying numbers through which you track goods in your inventory. Serial numbers are used when you need to trace each item of the same inventory ID, while lot numbers are used to trace items (of the same ID) that were purchased or produced together and have the same expiration dates (if applicable). You can segment lot and serial numbers via the Lot/Serial Classes (IN.20.70.00) form. Acumatica ERP supports the following types of segments for lot/serial numbers: constant, date, and auto-incrementing. See also segment .
low seasons	Time intervals in which the decreasing factors (used to divide the standard replenishment quantity to get lower replenishment quantities during low seasons) should be applied to quantities on purchase orders generated to replenish the stock. Each inventory item may have multiple low seasons, each with different decreasing factors. See also inventory item .
M	

main menu	A menu, that fits across the top on each page, containing functions that are not specific to the form or wiki article. The toolbar allows you to navigate to a Help article for the form (if available), add the form to Favorites and dashboards, among other capabilities. For details, see Main Menu . See also form , Favorites , dashboard
module	A software component of Acumatica ERP that consists of a variety of forms. While each module provides specialized functionality, it is tightly integrated with other modules. See also form .
multi-currency	A mode in which Acumatica ERP can function to support multiple currencies. If you have activated multi-currency support, you can manage transactions in various foreign currencies, record exchange rates for multiple rate types as needed, report in a specific foreign currency, and revalue GL, AP, and AR accounts in the base currency. The base currency is used for reporting and income statement calculation. See also base currency , Currency Management , module .
N	
navigation pane	A pane, located on the left side of any page, that allows you to select the form, or article you wish to display in the right pane. The navigation pane contains the tree structure of a module or wiki, which shows the hierarchy of its forms or articles and contains links to those items. When you click a link, the requested form or article opens in the right pane. See also form , dashboard , module .
negative inventory	An option, offered in Acumatica ERP, allowing a negative inventory balance for an inventory item or a group of items. This can occur when the inventory issue is made before the necessary quantity of the item arrives at the warehouse. To calculate the balance of over-issued inventory items, the most recent historical cost will be used until the item is received. When the inventory is received, the system will match the receipt cost with the issue cost, and generate a cost adjustment for the difference. Appropriate warnings are issued on transactions that will result in negative inventory balances. See also inventory item , warehouse .
non-stock item	An inventory item that is not stored in a warehouse. Such items can be of different types: labor, service (such as product assembly, installation, or personalization), charge, expense, and actual non-stock items, such as goods used only for drop-shipments. For a non-stock item, you can specify the following information in Acumatica ERP: base, sales, and purchase units; conversion coefficients; price and cost information; and the default GL accounts and subaccounts to be used for transactions with the items. See also inventory item , warehouse , General Ledger , account .
notification template	An article in the Notification Templates Wiki that is an email template with variables denoting values in an employee or contact record. When the email is sent, the system replaces variables with values from the database record associated with each addressee for email personalization.
numbering sequence	A set of rules the system uses to generate the next unique identifier when you create a new object of certain type (such as a batch). provides a number of predefined numbering sequences you can use. A numbering sequence may have subsequences. If one numbering sequence is used for multiple object types, all the objects get numbers

	according to the order in which they were created, so successive numbers can be assigned to objects of different types. See also batch .
O	
overdue charges	Charges calculated on open Accounts Receivable items that are past due. Acumatica ERP calculates overdue charges and displays them on customer statements. You can configure these charges to be compound charges (charges calculated on charges) or not. Overdue charges are based on terms that provide a schedule for payment. See also Accounts Receivable , customer statement .
P	
payment method	A way in which customers pay for goods they purchase from your organization. For each payment method, you can use a number of predefined elements: define the element names as you want them to appear on the interface, and set up input validation for these elements (input masks or regular expressions). Payment methods are based on the following general means of payment: credit cards, gift certificates, purchase orders, cash cards, and custom methods. See also customer , input mask .
PI cycle	The physical inventory cycle assigned to the stock item. PI cycles are used to arrange the items into groups for periodic counting. For more details on using PI cycles, see Planning for Physical Inventory. See also stock item .
posting class	A group of items in the Inventory module that defines the default account to be used and the rules for composing the default subaccount for transactions with the applicable inventory items. Accounts and subaccounts for transactions can be obtained from the following sources: inventory item, warehouse, or posting class. See also combined subaccount , Inventory , module , account , subaccount , combined subaccount , warehouse , inventory item .
prepayment	A document that represents amounts paid in advance for future purchases. In the Accounts Payable module, a vendor's request for prepayment is processed as follows: You use the Checks and Payments (AP.30.20.00) form to enter the prepayment. Then the prepayment is paid in full by an AP check in the same currency as the default cash account. If the payment method associated with the default cash account requires printing a check, print it and release the AP check, which changes its status to <i>Closed</i> and creates a payment of the <i>Prepayment</i> type with the reference number of the original prepayment request. After that, you can apply the prepayment to bills and adjustments. See also Accounts Payable , module .
price list	A list of sales prices that is set for goods sold in a specific currency, offered to customers of a particular customer price class, and specified with respect to various units of measure available for the items. In Acumatica ERP, you can maintain multiple price lists. See also sales price , customer , customer price class .
Purchase Orders	An Acumatica ERP module that provides functionality for efficient management of your company's supply chain and optimization of the cost of acquiring materials or services. See also module .

Purchase Requisitions	An Acumatica ERP module that allows you to streamline and customize the process of requesting needed items. You can request goods and services, approve requests, and prevent cost overruns. See also module .
R	
reason code	A code used to provide additional information regarding transactions in the system. When you configure a reason code via the Reason Codes (CS.21.10.00) form, you can specify whether this code is used in the Inventory module and, if so, how it is used. Inventory-related reason codes allow you to post transactions related to direct inventory operations (such as receipts, issues, transfers, adjustments, and physical inventory counts) to specific accounts and assign particular subaccounts to them to allow for more detailed reporting. See also reason code , Inventory, module , account , subaccount .
reconciliation	The process of matching the cash transactions recorded in Acumatica ERP against those presented on a bank statement. Theoretically, the balance of the cash account associated with the bank should reconcile to the balance of the bank statement, but there may be some discrepancy between account balances. The goal of reconciliation is to find discrepancies and determine whether each is due to error or timing. In Acumatica ERP, you mark documents as cleared as you receive preliminary information from the bank. Later, when you have received the bank statement, you reconcile transactions with the bank statement.
recurring GL transactions	GL transactions that repeat regularly. To automate the entering of recurring transactions, such as depreciation transactions, allows you to create schedules for them. A schedule defines how many times and how often specific batches should be repeated. One or several batches can be assigned to a schedule, but only batches with the <i>Balanced</i> status can be scheduled. Once a batch is assigned to a schedule, its status changes to <i>Scheduled</i> . To create schedules, use the Recurring Transactions (GL.20.35.00) form. See also schedule , batch .
replenishment policy	Settings that define how automatic replenishment for the inventory item is initiated, as well as its source, quantity, and time intervals, including low seasons, during which replenishment is initiated in smaller quantities. See also inventory item .
restriction group	A set of objects (such as users, accounts, and subaccounts) of two or more types created to, if the group includes users, restrict users' access to only objects in the same group; if the group doesn't include users, the restriction group relates its objects in a way that limits their use. For instance, one restriction group may include two users and a number of special-use accounts that only these two users can update, and another restriction group may include several GL expense accounts and a subaccount that should be used only with these particular accounts. If a restriction group is defined as inverse, the objects in the group instead cannot be used with one another. To learn more about restriction groups, see the Overview of Restriction Groups section in User Guide.
Retained Earnings account	A special system-maintained accounts that is of the <i>Liability</i> type and must be created before any actual data is entered. The Retained Earnings account accumulates the company's net income (or loss)

	after the dividends have been paid. Retained earnings are summarized over the years since the first year of company operations. During the financial year closing, this account is updated by the amount accumulated on the YTD Net Income account. See also YTD Net Income .
revaluation	The process of revising the value of AP, AR, or GL accounts that are maintained in a foreign currency. For more information, see Overview of Revaluations.
role	A set of access rights to certain system objects—such as specific Wiki articles, forms, form elements, and toolbar actions—to which you assign users. When you define roles, give only the access rights necessary to perform typical tasks. Sets of access rights by different roles should not intersect. We recommend that you assign to a user several roles rather than creating a more complicated role with the same privileges as multiple already-defined roles. Acumatica ERP has several preconfigured roles. For more information about roles, see Role-Based Security.
S	
Sales Orders	An Acumatica ERP module with the functionality required to manage sales-related activities, such as maintaining multiple price lists, configuring the system to calculate discounts, entering quotes, fulfilling sales orders, generating pick lists, creating shipments, and adding landed costs. See also module .
sales price	A price you set for a particular item that you sell in a specific currency, offer to customers of a particular customer price class, and specify with respect to an appropriate unit of measure. Sales prices can be maintained with regard to items' sales units or base units. See also customer , customer price class , sales unit , base unit .
sales unit	The unit of measure in which a particular item is sold to a customer. See also customer .
schedule	A definition in Acumatica ERP of how many times and how often specific AP batches, AR documents should be generated for recurring transactions. Once a batch or a document is assigned to a schedule, its status changes to <i>Scheduled</i> . The system uses the original documents or batches as templates to generate similar documents or batches with only transaction dates being changed as dictated by the schedule. See also batch .
Search text box	A text box, located on the top of the navigation pane on any page, that allows you to perform a quick search in the entities. You can click the Search icon to open the Search form, which offers more extensive capabilities to search the wikis, files, or entities in the system. See also navigation pane .
segment	<ol style="list-style-type: none"> 1. In Acumatica ERP, one of the parts of an identifier of an entity—such as account, subaccount, inventory item, subitem, warehouse or location reserved to carry special meaning. Segments should be populated with values before entities are created. Segment values are alphanumeric strings of the fixed length, and one of the segments may be assigned a numbering sequence. Several input validation options can be used to verify the segment values when users create new entities of the type.

	<p>2. To break the identifier into segments (as described above).</p> <p>See also account, subaccount, inventory item, location, warehouse.</p>
segmented key	<p>A system entity that lets you define the structure of identifiers for a certain type of object and then serves as a template when a user creates an identifier for a new object. The current version of Acumatica ERP provides the following segmented keys: <i>ACCOUNT</i>, for GL accounts; <i>SUBACCOUNT</i>, for GL subaccounts; <i>BIZACCT</i>, for vendor and customer accounts in the Accounts Payable and Accounts Receivable modules; <i>INVENTORY</i>, for inventory items; and <i>SALESPER</i>, for salesperson accounts. For more detailed information, see Identifier Segmentation in the Acumatica ERP User Guide. See also General Ledger, account, subaccount, vendor.</p>
standard cost method	<p>A method for inventory item valuation in which standard cost is calculated outside the system using company-specific policies. With this method, the currently effective standard costs are assigned to inventory items on their receipt, issue, adjustment or transfer, regardless of their actual costs. When items assigned to this method are received at the warehouses, any differences between the actual and standard costs are recorded to the specified standard cost variance accounts and posted to the General Ledger. Standard costs can be updated as often as is needed. See also inventory item, warehouse.</p>
stock item	<p>An inventory item stored and maintained in steady volumes at some warehouse. For each stock item, Acumatica ERP tracks a basic set of item properties, such as the item's identifier, description, price, cost, units of measure, and default warehouse and vendor information. Stock items can have many additional properties, known as attributes in Acumatica ERP, that do not affect item processing but may be important for analyzing the stock movements or item sales. See also inventory item, attribute.</p>
statement cycle	<p>The schedule for customer statements. You can also set up four aging periods that sort open documents by days past due. You can use the aging periods to prepare an AR aging schedule at the end of each month, which you can analyze to identify potential cash flow problems. Statement cycles can be assigned to customer classes and to individual customers. See also customer statement, aging period, customer class, customer.</p>
subaccount	<p>A subcategory of the account that carries identifying information; in Acumatica ERP, you use subaccounts with accounts to virtually split accounts into smaller, more specific ones. This gives you finer classification within the account for reporting and internal management purposes. While account identifiers carry the information about the account type along with the actual account number, subaccount identifiers can provide such information as the division, department, and cost center. Each journal entry is recorded with the appropriate account and subaccount combination. See the Hierarchy of Accounts and Subaccounts article for more details. See also account, journal entry.</p>
subitems	<p>Codes that allow further categorization of an inventory items. Subitems are used in the system if you have otherwise-identical products with different colors, sizes, or other properties tracked because of their importance to customers. Thus, under the same inventory ID, there may be a number of subitems—records about products that share all</p>

	settings of the inventory item record but have additional properties that differ. If your site uses subitems, they should be specified for each inventory ID related to a stock item. See also inventory item , customer , stock item .
Management	An Acumatica ERP module that lets you define users, roles, and restriction groups for security management. It also provides site management, Wiki management, task management, customization management, and file management capabilities, as well as integration services. See also module , user , role , restriction group , integration services .
T	
table	An arrangement of similar objects or details, each displayed with the same number of properties, on many forms. In a details table, each row represents an object or detail (for example, an account, subaccount, document line, or journal entry) and its properties; elements specifying properties are grouped into columns.
table toolbar	A toolbar on most forms, located above (and sometimes above and below) the Details table, that allows you to perform detail-related actions, including the following: add, edit, or delete details; filter details; perform custom actions; and rearrange details by changing the order of values in any column.
task	An activity that you have to complete before a due date but that doesn't have a specific time or duration. By default, you create tasks for yourself, but you also can create tasks and assign them to other employees.
tax	A compulsory financial contribution imposed by a government. In Acumatica ERP, you can configure taxes of the following major types: <i>Sales</i> , <i>Use</i> , <i>VAT</i> , and <i>Withholding</i> . The definition of each tax includes the tax rate (used to calculate the tax amount), the method of calculation, the effective date, and the accounts to which the tax amounts are posted. Each tax is reported to a specific tax agency and is paid to or claimed from the agency. See also account , tax agency .
tax agency	A tax authority, defined in as a vendor, that requires tax reports to be filed regularly. For your convenience, you can create a vendor class for tax agencies (local and federal). Each tax agency requires tax reports to be filed regularly. See also vendor , vendor class .
tax category	A list of taxes associated with a product or a service when it is purchased or sold. See also Taxes .
Taxes	An Acumatica ERP module that stores definitions of taxes, tax categories, and tax zones that are used across Acumatica ERP for automatic tax calculation for every document and transaction. See also module , Taxes , tax category , tax zone .
tax reporting group	An entity used to accrue taxable amounts and tax amounts charged on GL, AP, and AR transactions for tax reporting purposes. For example, a VAT requires two groups (input and output): one for tax amounts charged on sales, and another for tax amounts charged on purchases. A sales tax requires one output group for taxes on sales. Tax reporting groups are used to calculate the report lines for a report to a tax authority. For more information, see Tax Report Configuration in the Acumatica ERP User Guide.

tax report lines	Lines configured for a tax agency as a combination of output and input reporting groups for various taxes associated with the same tax agency. See also tax agency , Taxes .
tax zone	An area or tax jurisdiction where the same taxes are enforced. In Acumatica ERP, a tax zone includes a list of taxes to be applied to a customer's invoice or a vendor's bill depending on the location. Tax zones are used in other modules, such as General Ledger, Accounts Payable and Accounts Receivable.
U	
user	A person who uses the ERP system. Once a user has been authenticated, the system checks the user's membership in roles. Users can view only the forms, articles, and elements authorized by their roles, and can perform only the actions permitted by these roles. Users may be members of restriction groups, which let them access specific entities included in the groups. See also role , form , restriction group .
V	
vendor	One of your company's trade creditors. For ease of use, you can set up default values for vendor classes, individual vendors, and vendor documents. When users enter new bills, they must specify a vendor for each bill. Once they choose the vendor, certain elements on the form will be automatically populated with the vendor's default values. The due date and available discount are calculated automatically, based on the vendor's credit terms. See also vendor class , credit terms .
vendor class	A group of settings that provides default values when users create new vendor accounts. Divide vendors into classes based on the types of goods they sell or services they provide. For details, see Vendor Defaults and Overrides. See also vendor .
W	
warehouse	A place where goods are stored. A warehouse in Acumatica ERP does not necessarily represent one physical building where your inventory is stocked; you can divide a large physical storage space into several areas and define each as a warehouse in Acumatica ERP. A warehouse can even be virtual: For example, all goods that are on the way to you from the supplier can be considered as located in the virtual goods-in-transit warehouse.
warehouse location	An actual or virtual place in a warehouse that can be used to receive, store, or issue specific goods or all goods. Each warehouse can include several locations. Warehouse location IDs are defined with the <i>INLOCATION segmented key</i> . See also warehouse .
wiki article	An entity that consists of digital content on a particular topic and, along with other articles, makes up a wiki. Articles can be organized in folders in ways that best fit your needs.
wiki editor	The form, invoked when you click Edit for an open wiki article, that lets you edit both the article text and its properties.
wiki markup	<ol style="list-style-type: none"> 1. The syntax used to create wiki articles. Using wiki markup, you can create articles, add headings, tables of contents, hint boxes, and warning boxes.

	<p>2. A mode in which you can edit wiki articles, which lets you view the wiki markup.</p>
Wiki toolbar	<p>A toolbar, appearing below the main menu when you open a wiki article, that provides a variety of actions you can use as you browse the wiki and work with articles. These actions include creating a new article, moving to the previous or next article in the wiki tree, and printing or deleting the current article.</p>
Y	
YTD (Year-to-Date) Net Income account	<p>A special account, automatically maintained by the system, that records the net income (the difference between the amounts posted on income and expense GL accounts) accumulated since the beginning of the financial year. This difference is updated by every transaction posted. During closing of the financial year, the balance of the YTD Net Income account is transferred to the Retained Earnings Account and is reset to zero for a new financial year. The YTD Net Income account should be of the <i>Liability</i> type and must be created before any actual data is entered. See also account, financial year.</p>