Developer Guide

UI Components
2026 R1

Contents

Copyright.......7
UI Component Guide....... 8
Address Lookup.......9
Address Lookup: General Information....... 9
Address Lookup: Layout Examples.......10
Barcode Editor.......12
Barcode Editor: General Information....... 12
Barcode Editor: Configuration of Sound Effects.......13
Barcode Editor: Conversion from ASPX to HTML and TypeScript.......14
Button....... 17
Button: General Information.......17
Button: Configuration.......18
Button: Layout Examples....... 22
Button: Conversion from ASPX to HTML and TypeScript....... 24
Caption....... 27
Caption: General Information....... 27
Caption: Caption of a Fieldset, Table, or a Tab.......27
Caption: Caption of a Template.......28
Check Box....... 29
Check Box: General Information.......29
Check Box: Layout Examples....... 31
Check Box: Conversion from ASPX to HTML and TypeScript....... 32
Collapsible Area.......36
Collapsible Area: General Information....... 36
Collapsible Area: Configuration....... 36
Color Picker....... 39
Color Picker: General Information.......39
Color Picker: Conversion from ASPX to HTML and TypeScript.......40
Combo Box.......41
Combo Box: General Information.......41
Combo Box: Configuration.......42
Combo Box: Conversion from ASPX to HTML and TypeScript.......44
Currency....... 47
Currency: General Information....... 47

Currency: Configuration of the Currency Control.....49
Currency: Conversion from ASPX to HTML and TypeScript.....51
Data Feed.....54
Data Feed: General Information.....54
Data Feed: Configuration of Toolbar Buttons.....57
Data Feed: Configuration of Tiles.....59
Data Feed: Configuration of Bottom Toolbar.....70
Data Feed: Configuration of Layout.....71
Date and Time.....73
Date and Time Control: General Information.....73
Date and Time Control: Configuration.....75
Date and Time Control: Conversion from ASPX to HTML and TypeScript.....76
Dialog Box.....80
Dialog Box: General Information.....80
Dialog Box: Configuration.....83
Dialog Box: Field Validation.....87
Dialog Box: Opening a Dialog Box.....88
Dialog Box: Buttons on the Dialog Box.....90
Dialog Box: Executing Action on Dialog Box Closing.....93
Dialog Box: Files.....93
Dialog Box: Notes.....95
Dialog Box: Conversion from ASPX to HTML and TypeScript.....96
Error, Warning, or Informational Notification.....100
Error, Warning, or Informational Notification: General Information.....100
Error, Warning, or Informational Notification: Configuration.....102
Fieldset.....104
Fieldset: General Information.....104
Fieldset: Field Configuration.....110
Fieldset: Layout Examples.....111
Formula Editor.....118
Formula Editor: General Information.....118
Formula Editor: Implementation of a Standard Formula Editor.....121
Formula Editor: Configuration of a Combo Box Field for the Formula Editor.....121
Formula Editor: Implementation of Custom Option Providers.....122
Formula Editor: Customization of Option Providers.....124
Icon.....126

Icon: General Information 126

Icon: Conversion from ASPX to HTML and TypeScript 127

Image Uploader 128

Image Uploader: General Information 128

Image Uploader: Conversion from ASPX to HTML and TypeScript 128

Image Viewer 131

Image Viewer: General Information 131

Image Viewer: Loading of Images from a Third-Party Source 132

Image Viewer: Conversion from ASPX to HTML and TypeScript 133

Label 134

Label: General Information 134

Label: A Combo Box Control as a Label 134

Label: A Single Label for Multiple Controls 135

Label: Conversion from ASPX to HTML and TypeScript 136

Link Editor 138

Link Editor: General Information 138

LinkEditor: Conversion from ASPX to HTML and TypeScript 139

Mail Editor 141

Mail Editor: General Information 141

Mail Editor: Configuration 143

Mail Editor: Conversion from ASPX to HTML and TypeScript 143

Mask Editor 146

Mask Editor: General Information 146

Mask Editor: Conversion from ASPX to HTML and TypeScript 146

Menu 148

Menu: General Information 148

Menu: Configuration of the Menu Control 150

Menu: Handling the MenuSelected Event 152

Radio Button (Option Button) 155

Radio Button: General Information 155

Radio Button: Configuration 157

Radio Button: Conversion from ASPX to HTML and TypeScript 159

Record Title 163

Record Title: General Information 163

Record Title: Configuration 165

Resizer 166

Resizer: General Information........ 166

Rich Text Editor........ 169

Rich Text Editor: General Information........ 169

Rich Text Editor: Configuration and Layout........ 171

Rich Text Editor: Conversion from ASPX to HTML and TypeScript........ 174

Selector........ 179

Selector Control: General Information........ 179

Selector Control: Configuration from Backend........ 181

Selector Control: Configuration of a Link........ 183

Selector Control: Configuration of the Control's Text........ 185

Selector Control: Configuration of the Lookup Table........ 186

Selector Control: Manual Configuration of the Value and Text........ 187

Selector Control: Multiple-Selection Mode........ 188

Selector Control: Displaying of the Value in the Selector Control........ 188

Selector Control: Selector Parameters........ 188

Selector Control: Conversion from ASPX to HTML and TypeScript........ 189

Splitter........ 194

Splitter: General Information........ 194

Splitter: Configuration........ 197

Splitter: Conversion from ASPX to HTML........ 198

Tab........ 201

Tab: General Information........ 201

Tab: Configuration........ 204

Tab: Conversion from ASPX to HTML and TypeScript........ 205

Table (Grid)........ 208

Table (Grid): General Information........ 208

Table (Grid): Configuration of the Table and Its Columns........ 211

Table (Grid): Configuration of the Table Toolbar........ 213

Table (Grid): Configuration of the Search in the Table........ 217

Table (Grid): Table with Highlighted Contents........ 218

Table (Grid): Changing of Column Properties Dynamically........ 219

Table (Grid): Layout Examples........ 220

Table (Grid): Conversion from ASPX to HTML and TypeScript........ 223

Time Span........ 248

Time Span: General Information........ 248

Time Span: Conversion from ASPX to HTML and TypeScript........ 250

Text Box ... **252**
Text Box: General Information ... 252
Text Box: Multiline Text Box ... 253
Text Box: Conversion from ASPX to HTML and TypeScript ... 254
Tree ... **255**
Tree: General Information ... 255
Tree: Configuration ... 256
Tree: Markup of a Tree Node Using HTML ... 257
Tree: Conversion from ASPX to HTML and TypeScript ... 258
Tree Selector ... **271**
Tree Selector Control: General Information ... 271
Tree Selector Control: Configuration ... 272
Tree Selector Control: Conversion from ASPX to HTML and TypeScript ... 273
Upload Dialog Box ... **281**
Upload Dialog Box: General Information ... 281
Upload Dialog Box: Configuration of an Upload Dialog Box ... 282
Upload Dialog Box: Conversion from ASPX to HTML and TypeScript ... 283
Upload Files Button ... **286**
Upload Files Button: General Information ... 286
Upload Files Button: Conversion from ASPX to HTML and TypeScript ... 287
Wizard ... **289**
Wizard: General Information ... 289
Wizard: Configuration of Buttons ... 292
Wizard: Conversion from ASPX to HTML and TypeScript ... 293

Copyright

© 2026 Acumatica, Inc.

ALL RIGHTS RESERVED.

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

3075 112th Avenue NE, Suite 200, Bellevue, WA 98004, USA

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: 2026 R1

Last Updated: 03/15/2026

UI Component Guide

In this guide, you can find details about the configuration of UI components. For each UI component, you can find design guidelines, naming conventions, and information about the configuration of the related UI controls.

Address Lookup

In this chapter, you will learn about the configuration of **Address Lookup** buttons. You will learn when to use an **Address Lookup** button and how to organize a layout that includes such a button.

Address Lookup: General Information

An **Address Lookup** button is a control that a user can click to search for addresses that are available in the entered records. When a user clicks this control, the system opens the corresponding dialog box, where the user can perform a search operation. In this dialog box, the user can also add a new address, update an existing address, and fill in the missing address information in a record.

An example of this control is available on the **Addresses** tab of the *Sales Orders* (SO301000) form, as shown in the following screenshot.

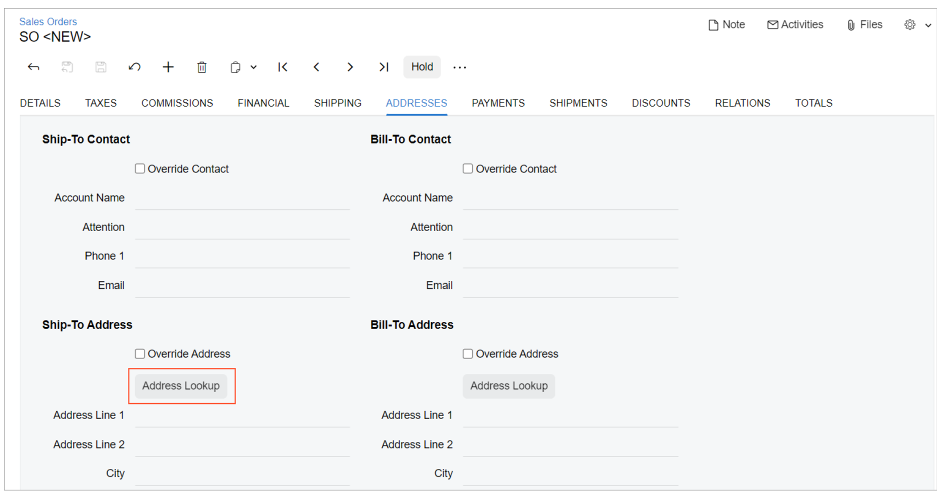


Figure: The Address Lookup button

In the Classic UI, **Address Lookup** button is defined by `PXButton`, with the corresponding value specified for the button's `CommandName` property. You also need to separately define the code for the corresponding dialog box that opens when this button is clicked. In the Modern UI, an **Address Lookup** button is defined by the `qp-address-lookup` HTML tag. This tag already includes the code for both the button and the corresponding dialog box.

Learning Objectives

In this chapter, you will learn the following about an **Address Lookup** button:

- The design guidelines for an **Address Lookup** button
- The proper configuration of a button for specific cases, such as when a button is to be placed below another element

Applicable Scenarios

You configure an **Address Lookup** button on a form when a user needs to look up addresses that are available in the entered records.

Address Lookup ID

An ID of a address lookup in HTML consists of three parts: the `addressLookup` prefix, the location of the **Address Lookup** button, and the semantic name separated with a dash. The semantic name describes the purpose of the element. For example, the **Address Lookup** button on the **Billing** tab that specifies an address may have the `addressLookupBilling-Addresses` ID, as the following code shows.

```
<qp-address-lookup id="addressLookupBilling-Addresses"
  view.bind="Billing_Address" class="col-12"></qp-address-lookup>
```

Address Lookup: Layout Examples

In this topic, you can find examples of ways to configure various layouts with **Address Lookup** buttons.

Address Lookup Button at the Top of a Fieldset

You can add an **Address Lookup** button at the top of a fieldset, as shown in the following screenshot.

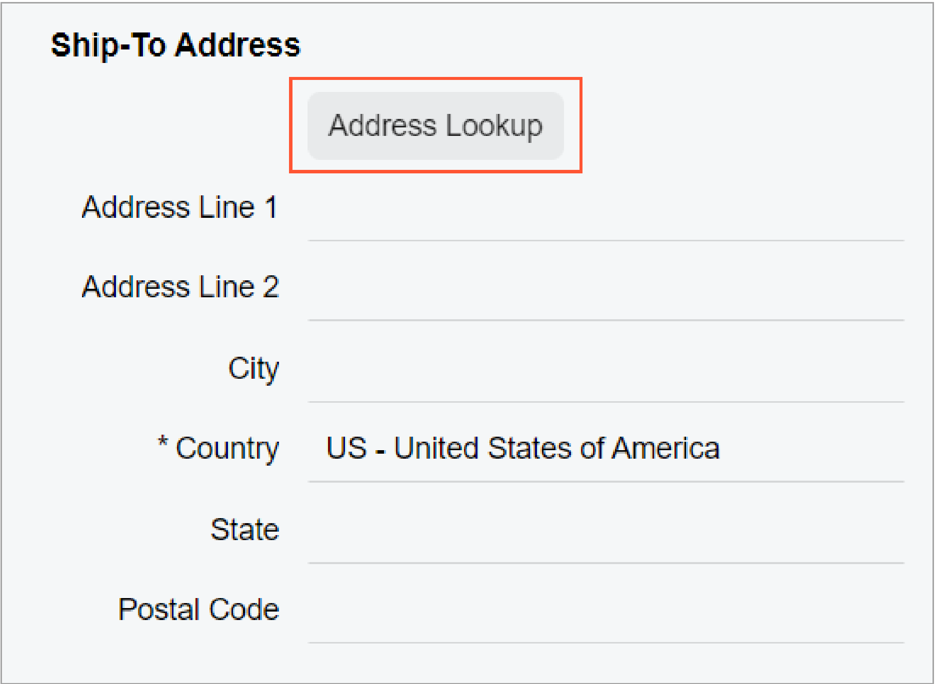


Figure: The Address Lookup button

To add an **Address Lookup** button, you add a `field` tag with the `unbound` attribute and nest the `qp-address-lookup` tag within it, as shown in the following code example.

```
<field name="FakeField" unbound>
  <qp-address-lookup class="col-12" view.bind="Shipping_Address">
  </qp-address-lookup>
</field>
```

Address Lookup Button Below a UI Element

You can add a button after another control so that it is displayed on the next line below the control. To do that, you need to add the button control inside the field tag and specify `class="col-12"` for the button control.

For example, suppose that you need to place the **Address Lookup** button on the next line after a check box, as shown in the following screenshot.

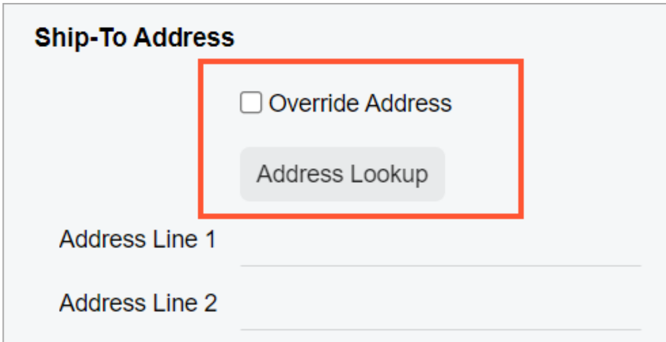


Figure: The Address Lookup button below a check box

The **Address Lookup** button is implemented by using the `qp-address-lookup` tag. The following code shows an example of moving the **Address Lookup** button to the next line.

```
<field name="OverrideAddress">
  <qp-address-lookup class="col-12" view.bind="Shipping_Address">
  </qp-address-lookup>
</field>
```

Barcode Editor

In this chapter, you will learn about the configuration of a barcode editor. You will learn when to use a barcode editor and how to organize a layout that includes a barcode editor.

Barcode Editor: General Information

A barcode editor is a control in which a user can input a barcode and view its representation in the UI. The barcode editor uses the functionality of the barcode-driven engine provided by Acumatica ERP.

A barcode editor is defined by `PXTextEdit` in the Classic UI. In the Modern UI, you define a barcode editor in one of the following ways:

- By using the `field` or `qp-field` tag with the `qp-barcode-input` control type specified
- Explicitly by using the *`qp-barcode-input`* control

Learning Objectives

In this chapter, you will learn the following about the barcode editor:

- The design guidelines for the barcode editor, including the naming conventions and layout recommendations
- The proper configuration of the barcode editor for specific cases, such as enabling the sound signal

Applicable Scenarios

You configure the barcode editor when you want to give a user the capability to scan a barcode.

Overview of the Barcode Editor

The barcode editor looks like a text box with a text value that corresponds to the barcode values. The system submits the text value to the server when a user presses Enter by using the action specified in the `submitCommand` property. Note that the default action `Scan` has already been implemented and specified in the control, so normally, you do not need to specify it. For details, see *Barcode-Driven Engine Guide*.

To facilitate back-to-back barcode scanning, the control prevents defocusing when a user presses Enter.

You can also configure the sound signal that the computer plays when a user scans the barcode. For details, see *Barcode Editor: Configuration of Sound Effects*.

The declaration of the field for the barcode editor in TypeScript must not include `<PXFieldOptions.CommitChanges>` because that setting incurs extra requests to the server. A fatal error will be thrown if the `CommitChanges` setting is specified.

UI Naming Convention

The following table shows the UI naming convention for a barcode editor.

Naming Convention	Example
Use a noun or a noun phrase to describe the contents of a barcode editor. Preferably, the name of a barcode editor should consist of one or two words.	The **Scan** barcode editor on the *Scan Materials* (AM300030) form, which is shown in the following screenshot Scan Materials Reset User Settings Remove Set Qty Release Scan Reference Nbr. Warehouse The Scan Material mode is in use. The active mode is set to Scan Material. Scan the Production Order ID.

Use of the BarcodeProcessingScreen Component

The source code of Acumatica ERP provides a reusable component, named `BarcodeProcessingScreen`, that you can use to insert the barcode editor with all necessary logic. The component is defined in the `Screen/src/screens/barcodeProcessing` folder of the instance, and it uses the functionality defined in the barcode-driven engine.

The reusable component includes the following:

- The Summary area with the scanned barcode and the message indicating the result of the scanning
- The **Scan Log** tab, which lists all attempts to scan the barcode
- The logic to produce the sound signal when the barcode is scanned

For details on including the reusable component in the UI, see *Reusing a UI Definition*.

For details on configuring the backend part of the form that uses the barcode-driven engine, see *Barcode-Driven Engine Guide*.

Related Links

- *Barcode-Driven Engine Guide*

Barcode Editor: Configuration of Sound Effects

In the barcode editor, you can configure a sound signal that the computer plays when a barcode is being scanned. You can also specify the location of the sound signal library to specify a custom sound.

Specifying a Sound Signal

To specify a sound signal, you need to pair it with an invisible control. This control should be mapped to a field that contains the name of sound files. Do the following:

1. In TypeScript, define a view and a field that contains a sound file, as shown in the following example.

```
export class ScanInfo extends PXView {
    MessageSoundFile: PXFieldState<PXFieldOptions.Disabled>;
    ...
}

export abstract class BarcodeProcessingScreen extends PXScreen {
    @viewInfo({ containerName: "Scan Information" })
    Info = createSingle(ScanInfo);
```

```
    ...
}
```

2. In the HTML template, add a field that contains a sound file, as shown in the following example. The field should be invisible.

```
<qp-fieldset id="fsInfo-Header" view.bind="Info">
  <field name="MessageSoundFile" show.bind="false"></field>
</qp-fieldset>
```

3. Specify the field name in the `soundControl` property of the barcode editor control, as shown in the following example.

```
<field name="Barcode" control-type="qp-barcode-input"
  config-sound-control.bind="Info.MessageSoundFile" ...></field>
```

Configuring the Location of the Sound Signal Library

The `config` parameter of the `qp-barcode-input` control provides the following ways to specify the sound signal and its location:

- The `soundMapper` function: The function accepts the name of the file as an argument and returns the file with its path. An example of the function definition is shown in the following code.

```
soundMapper : (name) => `https://contoso.com/themes/business/${name}.webm`
```

- The `soundPath` and `soundControl` properties: The properties specify the URL of the folder with the sound files and the field that contains the file name without an extension, respectively.

  If the `soundMapper` function is defined, the system does not analyze the values defined in these properties. If the `soundMapper` function is not defined, the system determines the sound file location as follows: `"soundPath" + "value_from_soundControl" + ".wav"`. For example, suppose that you specify the `soundPath` and `soundControl` values as shown in the following code.

```
@controlConfig({
  soundPath: "https://contoso.com/themes/business/",
  soundControl: Info.MessageSoundFile
})
Barcode : PXFieldState;
```

  The name of the file in the `Info.MessageSoundFile` field is *"Success"*. The resulting file URL will look as follows.

```
https://contoso.com/themes/business/Success.wav
```

Barcode Editor: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the barcode editor to HTML or TypeScript elements.

PXTextEdit

The following table shows the correspondence between the `PXTextEdit` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXTextEdit`

```
<px:PXTextEdit ID="edBarcode" runat="server"
  DataField="Barcode">
  <AutoCallBack Command="Scan" Target="ds">
    <Behavior CommitChanges="True" />
  </AutoCallBack>
  <ClientEvents Initialize="Barcode_Initialize"/>
</px:PXTextEdit>
```

</td><td>

Replace it with `field` and specify `control-type="qp-barcode-input"` in the `field` tag. In rare cases, you should replace it explicitly with the `qp-barcode-input` control.

```
<field name="Barcode" control-type="qp-barcode-input"
  config-sound-control.bind="Info.MessageSoundFile"
  config-submit-command.bind="Scan"></field>
```

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

`DataField`

```
<px:PXTextEdit ... DataField="Barcode">
```

</td><td>

Use the `name` attribute of the `field` tag.

```
<field name="Barcode" ... >
```

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

`ID`

```
<px:PXTextEdit ID="edBarcode" ... >
```

</td><td>

Replace it with the `id` attribute of the `qp-field` tag if this tag is used as a replacement. In other cases, the ID is not necessary for a barcode editor.

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

AutoCallBack in PXTextEdit

The following table shows the correspondence between the `AutoCallBack` element located inside the `PXTextEdit` tag and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`AutoCallBack > Command`

```
<px:PXTextEdit ...>
  <AutoCallBack Command="Scan" Target="ds">
    <Behavior CommitChanges="True" />
  </AutoCallBack>
</px:PXTextEdit>
```

</td><td>

In the `fieldConfig` decorator or in the `field` tag, specify the action name in the `submitCommand` property of the control.

```
<field name="Barcode" control-type="qp-barcode-input"
  config-submit-command.bind="Scan"></field>
```

The `Scan` action is defined in the barcode-driven engine, so you do not need to define it in TypeScript.

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

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the barcode editor. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties/Comments
`PXTextEdit`	• `runat`
`AutoCallBack`	• `Target`
`ClientEvents`	The tag is not necessary.

Button

In this chapter, you will learn about the configuration of buttons. You'll learn when to use buttons, how to name them, and how to organize a layout that includes buttons.

Button: General Information

A button is a control that users can interact with to trigger an action or execute a command.

A button is defined by `PXButton` in the Classic UI. In the Modern UI, a button is defined by the *`qp-button`* HTML tag.

You do not need to manually add the standard buttons to the form toolbar of an Acumatica ERP form. The buttons representing actions that are available on the form toolbar are added automatically to the form. For details about adding button to various parts of the form, see *Button: Configuration*.

Learning Objectives

In this chapter, you will learn the following about a button:

- The design guidelines for a button, including the naming conventions and layout recommendations
- The proper configuration of a button for specific cases, such as when a button is to be placed below another element
- The key details of each property of a button

Applicable Scenarios

You configure buttons in the following user scenarios:

- A user needs to trigger an action or execute a command, such as initiating invoice creation.
- A user needs to confirm or cancel an action or a command, such as canceling the release of an invoice.

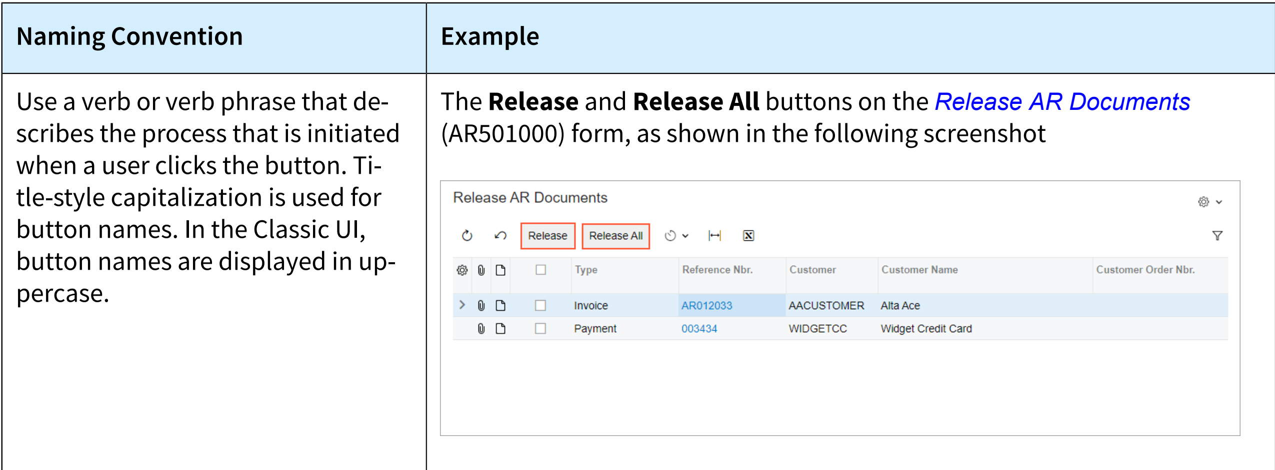
Button ID

An ID of a button in HTML consists of two parts, the `button` prefix and the semantic name. The semantic name describes the purpose of the element. For example, you may have a button that adjusts the document amount. You can set its ID to `buttonAdjustDocAmt`, as the following code shows.

```
<qp-button id="buttonAdjustDocAmt"></qp-button>
```

UI Naming Conventions

The following table shows the UI naming conventions for a button.

<table>
<tr><th>Naming Convention</th><th>Example</th></tr>
<tr><td>Use a verb or verb phrase that describes the process that is initiated when a user clicks the button. Title-style capitalization is used for button names. In the Classic UI, button names are displayed in uppercase.</td><td>The Release and Release All buttons on the Release AR Documents (AR501000) form, as shown in the following screenshot

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

Button: Configuration

In this topic, you can learn how to adjust a button for specific cases.

Action Definition in TypeScript

The actions that are defined in the graph or in the workflow have corresponding commands displayed on the More menu of the form toolbar by default. You do not need to define them in the TypeScript code of the Acumatica ERP form.

However, if you need to place a button for an action on an area of the form other than the toolbar or the More menu, you need to do the following:

- To place a button on a table toolbar, you specify the property with the name of the corresponding action in the view class of the table, as shown below.

```
import {
    gridConfig,
    PXView,
    PXActionState
} from "client-controls";

@gridConfig({
    preset: GridPreset.Details
})
export class SOLine extends PXView {
    AddInvBySite: PXActionState;
}
```

- To place a button in a dialog box or somewhere on the form (outside of the form toolbar or table toolbar), you specify the property with the name of the corresponding action in the screen class, as the following code shows. You then add the `qp-button` tag for the action in HTML. For details, see *Dialog Box: Configuration*.

```
import {
    graphInfo,
    PXScreen,
    PXActionState
} from "client-controls";
```

```
@graphInfo({
    graphType: "PX.Objects.GL.AccountHistoryEnq",
    primaryView: "Filter"
})
export class GL401000 extends PXScreen {
    AddInvoiceOK: PXActionState;
}
```

- To execute an action when a user clicks the link in a column of a grid, you specify the property with the name of the corresponding action in the screen class and use the `linkCommand` decorator for the column that displays the link. An example of such action is shown in the following code.

```
@graphInfo({
    graphType: "PX.Objects.FA.CalcDeprProcess",
    primaryView: "Filter"
})
export class FA502000 extends PXScreen {
    ViewAsset: PXActionState;
    Balances = createCollection(FABookBalance);
}

@gridConfig({
    preset: GridPreset.Processing
})
export class FABookBalance extends PXView {
    @linkCommand("ViewAsset")
    @columnConfig({ allowUpdate: false })
    AssetID: PXFieldState;
    ...
}
```

- To execute an action when a user clicks the link in a field in a fieldset, you specify the property with the name of the corresponding action in the screen class and use the `editCommand` property in the `controlConfig` decorator for the field that displays the link. An example of such action is shown in the following code.

```
@graphInfo({
    graphType: "PX.Objects.EP.EPAssignmentMaint",
    primaryView: "AssigmentMap"
})
export class EP205000 extends PXScreen {
    OpenForm: PXActionState;
    AssigmentMap = createSingle(EPAssignmentMap);
}

export class EPAssignmentMap extends PXView {
  @controlConfig({editCommand: "OpenForm"})
  GraphType: PXFieldState<PXFieldOptions.CommitChanges>;
}
```

When you include the property of the `PXActionState` type for the action in the TypeScript code of a form, this action is automatically bound by name to an action in the graph. The action is not displayed on the form toolbar by default. To specify explicitly whether the button for the action is displayed on the form toolbar, you can use the `PXButton.DisplayOnMainToolbar` property in the graph's action declaration.

Button for a Graph Action in HTML

If you need to use an action defined in a graph in your HTML code, you must specify the `state` attribute for the button that corresponds to this action, as shown in the following code.

```
<qp-button id="buttonAddBlanketLine" state.bind="AddBlanketLineOK">
</qp-button>
```

In the code above, you have used the `state` attribute and specified the *AddBlanketLineOK* value for its `bind` property, which is the name of the action that is defined in the graph.

Configuration of Button Properties

You use the `actionConfig` decorator to specify the properties of an action explicitly defined in a view class, such as an action that corresponds to a button on the table toolbar or a button in a dialog box. The full list of properties is defined in the *`IActionConfig`* interface. The following code uses this decorator.

```
@localizable
export class LocalizableStrings {
    static btn_specifyDatabaseEngine = "Specify Database Engine";
}

export class AU209000 extends PXScreen {
    localizableStrings = LocalizableStrings;

    @actionConfig({text: LocalizableStrings.btn_specifyDatabaseEngine})
    actionAddSqlAttribute: PXActionState;
}
```

You can also specify the properties of any button by using the `config` attribute of the *`qp-button`* tag in HTML, as the following code shows.

```
<qp-button id="pdfPrevPageBtn"
  config.bind="{images: { normal: 'svg:main@arrowLeft' } }">
</qp-button>
```

You can also handle the state and appearance of any action that corresponds to a button or command on the table toolbar by using the `actionsConfig` property of the `gridConfig` decorator, as shown in the following examples.

```
// Hides the Refresh button from the table toolbar.
@gridConfig({
  actionsConfig: { refresh: { hidden: true } }
})
export class SOLine extends PXView

// Adds the Custom Refresh button.
@gridConfig({
  actionsConfig: {
    refresh: {
      renderAs: MenuItem.RENDER_TEXT,
      images: {},
      text: "Custom Refresh" }
  }
})
```

```
export class POLine extends PXView
```

Standard Buttons of a Dialog Box

For the standard buttons of a dialog box—such as **OK** or **Cancel**—you must have the `dialog-result` attribute specified. Also, if you do not specify the `caption` attribute, the button's caption will be set to the value of the `dialog-result` attribute, and the caption will be localizable by general rules. You can override the default caption by specifying the `caption` attribute. You can omit the `caption` attribute if its value is the same as the value of the `dialog-result` attribute.

The following code example shows an approach to declaring the **OK** button of a dialog box that has a caption.

```
<qp-button id="buttonOK" caption="Confirm" dialog-result="OK"> </qp-button>
```

The following code example shows an approach to declaring the **Cancel** button of a dialog box that does not have a caption.

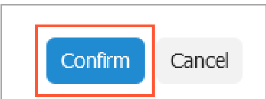
```
<qp-button id="buttonCancel" dialog-result="Cancel"></qp-button>
```

For more details on buttons of a dialog box, see *Dialog Box: Buttons on the Dialog Box*.

Configuration of Button Connotation

You can configure a button connotation from frontend by using the `class` property of the `qp-button` tag. The property can have the following values:

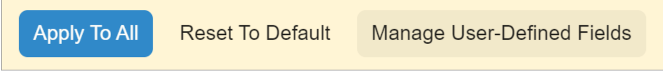
- `major-button`: The button is highlighted with blue, as shown in the following screenshot.

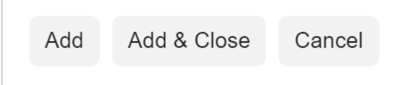
  An example implementation of this button is shown in the following code.

  ```
  <qp-button
      id="btnOK" dialog-result="OK"
      config.bind="{ validateInput: true }"
      caption="Confirm"
      class="major-button">
  </qp-button>
  ```

- `minor-button`: The button does not have any frame, gray or blue. When a user hovers over the button with this class, the button gets a gray frame. In the following screenshot, the **Reset to Default** and **Manage User-Defined Fields** buttons have `class="minor-button"` specified. The **Manage User-Defined Fields** button has a gray frame because a mouse hovers over it.

By default, a button has a gray frame as shown in the following screenshot.



Button: Layout Examples

In this topic, you can find examples of ways to configure various layouts with buttons.

Button Below a UI Element

You can add a button after another control so that it is displayed on the next line below the control. To do that, you need to add the button control inside the field tag and specify `class="col-12"` for the button control.

For example, suppose that you need to place the **Address Lookup** button on the next line after a check box, as shown in the following screenshot.

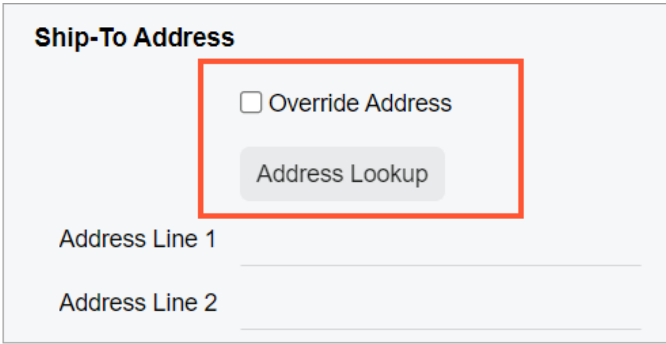


Figure: The Address Lookup button below a check box

The **Address Lookup** button is implemented by using the `qp-address-lookup` tag. The following code shows an example of moving the **Address Lookup** button to the next line.

```
<field name="OverrideAddress">
  <qp-address-lookup class="col-12" view.bind="Shipping_Address">
  </qp-address-lookup>
</field>
```

Button Above a Set of UI Elements

Suppose that you need to display the **Shop for Rates** button above a set of boxes, as shown in the following screenshot.

Figure: A button placed above other elements

You need to put the `qp-button` tag within the `field` tag with the `replace-content` and `unbound` attributes, as shown in the following code example.

```
<qp-fieldset slot="A" id="currentDocument_formDeliverySettings"
  wg-container view.bind="CurrentDocument" caption="Delivery Settings">
```

```
  <field name="fakename" replace-content unbound>
    <qp-button id="buttonShopRates" class="col-12" state.bind="ShopRates"></qp-button>
  </field>
  <field name="FOBPoint"></field>
  <field name="Priority"></field>
  <field name="ShipTermsID"></field>
</qp-fieldset>
```

Button with a Fixed Width

Suppose that you want to set a fixed width to be used for the **Shop for Rates** button regardless of the width of the screen, as shown in the following screenshots.

Figure: A button with a fixed width on a narrow screen

Figure: A button with a fixed width on a wide screen

You need to specify the `class="col-auto"` attribute to set a fixed width for the button, as shown in the following code example.

```
<field name="ShipVia">
  <qp-button id="btnShopRates" state.bind="ShopRates" class="col-auto">
  </qp-button>
</field>
```

Button with an Image

An image for a button is an SVG icon that is stored in the `\Content\svg_icons` folder of your instance. You can add a button with an image by using one of the following approaches:

- Specify the `ImageSet` and `ImageKey` properties in the `PXButton` attribute of the corresponding action in the graph code.
- Specify the `config` attribute and provide a value for the `imageSet` and `imageKey` properties in it.

The `ImageSet` (declared in a graph) or `imageSet` (declared in HTML) property specifies one of the icon sets available in the `\Content\svg_icons` folder. The `ImageKey` (declared in a graph) or `imageKey` (declared in HTML) property specifies a particular item from this icon set.

The following code example shows you how to define the `Refresh` image for a button whose action is defined in the graph code.

```
[PXButton(ImageKey = Web.UI.Sprite.Main.Refresh, ImageSet = Web.UI.Sprite.AliasMain)]
public IEnumerable Refresh(PXAdapter adapter) {}
```

The following code example shows you how to define the `Refresh` image for a button in HTML. This code adds a button to the right of an existing field.

```
<field name="CuryOrigDocAmt">
```

```
  <qp-button id="buttonAdjustDocAmt" state.bind="model.viewModel.AdjustDocAmt"
 class="col-4"
    config.bind="{imageSet: 'main', imageKey: 'Refresh'}"></qp-button>
</field>
```

Button: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert ASPX elements that are related to buttons to HTML or TypeScript elements.

PXButton

The following table shows the correspondence between `PXButton` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXButton` `<px:PXButton ... />`	Is replaced by the `qp-button` tag. `<qp-button id="buttonAdjustDocAmt"></qp-button>`
`CommandName` `<px:PXButton CommandName= "AddOK"   ... />`	Is replaced by the `state.bind` attribute, which binds the button to the specified action defined in a graph. `<qp-button id="buttonAddOK" state.bind="AddOK"></qp-button>`
`DialogResult` `<px:PXButton DialogResult= "Cancel"   ... />`	Is replaced by `dialog-result` attribute, which specifies the response returned by an action. This attribute is required for the standard buttons of a dialog box. `<qp-button id="buttonCancel" dialog-result="Cancel"></qp-button>` This attribute can have any of the following values: • *Abort* • *Cancel* • *Ignore* • *No* • *None* • *OK* • *Retry* • *Yes*

ASPX	HTML or TypeScript

`Enabled`
 	
`<px:PXButton`
`Enabled= "True"  ... />`
	Is replaced by the `enabled` attribute, which indicates (if set to *true*) that the button is enabled on the UI.
	
	`<qp-button id="buttonAdjustDocAmt">`
	`enabled="true" </qp-button>`

`ID`
 	
`<px:PXButton`
`ID= "button1"  ... />`
	Is replaced by the `id` attribute. This is a required attribute.
	
	`<qp-button id="button1">`
	`</qp-button>`

`Text`
 	
`<px:PXButton`
`Text= "Add More"  ... />`
	Is replaced by the `caption` attribute, which defines the text that is displayed on the button. The string specified in this attribute is localizable.
	
	`<qp-button id="button1"`
	`caption="Add More" dialog-result="Add">`
	`</qp-button>`

`Tooltip`
 	
`<px:PXToolBarButton ...>`
`  <Tooltip="Add Related Tables">`
`</px:PXToolBarButton>`
	Use the `tooltip` attribute of the `qpbutton` tag.
	
	`<qp-button id="buttonAddTable"`
	`  config.bind="{tooltip="Add a related ta-`
	`ble for the selected table"}">`
	
	For buttons on a table toolbar, use the `tooltip` parameter while defining an action in the `topBarItems` property of the `gridConfig` decorator. For details, see *Table (Grid): Configuration of the Table Toolbar*
	
	`@gridConfig({`
	`  topBarItems: {`
	`    addRelatedTableRelations: {`
	`      index: 5,`
	`      config: {`
	`        commandName: "addRelatedTableRela-`
	`tions",`
	`        text: ActionCaption.AddRelated-`
	`TableRelations,`
	`        toolTip: ActionToolTip.AddRelated-`
	`TableRelations`
	`    }`
	`  }`
	`}`
	`})`
	`export class GIRelation extends PXView {`
	`  ... }`

`Size`	Is replaced by the `class` attribute, which indicates how many columns of width the button should occupy.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`Visible`

```
<px:PXButton
Visible= "True"  ... />
```

</td><td>

Is replaced by the `hidden` attribute, which indicates (if set to *true*) that the button is not visible on the UI.

```
<qp-button id="buttonAdjustDocAmt">
hidden="false" </qp-button>
```

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

PXDSCallbackCommand

The following table shows the correspondence between `PXDSCallbackCommand` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PopupCommand`

</td><td>

Use the `popupCommand` property of the `action-Config` decorator.

```
@actionConfig({
  popupCommand: 'Refresh'
})
CreatePrepayment: PXActionState;
```

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

`PopupCommandTarget`

</td><td>

Use the `target` property of the `actionConfig` decorator.

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

`PopupPanel`

</td><td>

Use the `popupPanel` property in the `config` attribute of the `qp-button` tag.

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

Obsolete ASPX Controls and Properties

The following table lists obsolete ASPX elements that are related to buttons. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXButton`	• `CommandSourceID` • `runat` • `SyncVisible`
`PXDSCallbackCommand`	• `CommitChanges` • `DependOnGrid` • `Name` • `PostData` • `StartNewGroup` • `Visible`

Caption

In this chapter, you will learn about the configuration of captions. You will learn when to use captions and how to organize a layout that includes them.

Caption: General Information

A caption is a text that precedes a group of fields or a table. The caption is defined by the `qp-caption` tag in HTML.

Learning Objectives

In this chapter, you will learn the following about a caption:

- The design guidelines for the caption, including the naming conventions and layout recommendations
- The proper configuration of the caption

Applicable Scenarios

You configure a caption when you need to add a title for a group of controls inside a template.

Caption: Caption of a Fieldset, Table, or a Tab

To specify a caption for a fieldset, table, or tab, you use the `caption` attribute of the respective tag. The `caption` attribute is localizable. For details, see *Fieldset*, *Table (Grid)*, and *Tab*.

The following screenshot shows the **Attributes** caption specified for a table.

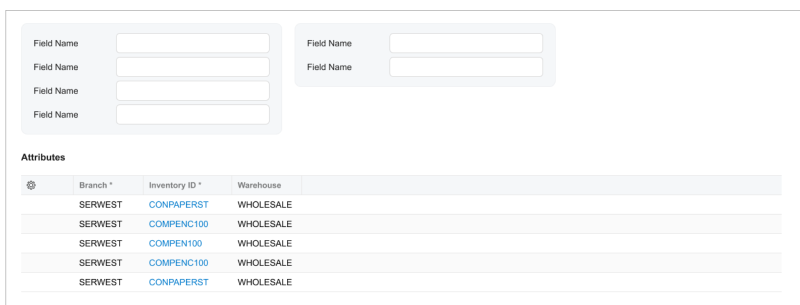


Figure: A caption for a table

The following code example adds a caption to a grid.

```
<qp-grid id="ordersGrid"
  view.bind="BlanketOrderChildrenDisplayList"
  caption="Child Orders">
</qp-grid>
```

Caption: Caption of a Template

To specify a caption for a group of fields inside a template, as shown with the **Attribute** caption in the following screenshot, you use the `qp-caption` control.

Figure: A caption for a template

The following code example implements this approach.

```
<qp-caption caption="Contacts"></qp-caption>
<qp-template name="1-1-1" id="SyncPolicy_tab_Contacts" wg-container>
    <qp-fieldset id="contacts_left" slot="A" view.bind="SyncPolicy">
        <field name="ContactsSync"></field>
        <field name="ContactsSeparated"></field>
        <field name="ContactsMerge"></field>
        <field name="ContactsSkipCategory"></field>
        <field name="ContactsGenerateLink"></field>
    </qp-fieldset>
    <qp-fieldset id="contacts_right" slot="B" view.bind="SyncPolicy">
        <field name="ContactsDirection"></field>
        <field name="ContactsFolder"></field>
        <field name="ContactsFilter"></field>
        <field name="ContactsClass" config-allow-edit.bind="true"></field>
    </qp-fieldset>
</qp-template>
```

Check Box

In this chapter, you will learn about the configuration of check boxes. You'll learn when to use check boxes, how to name them, and how to organize a layout that includes them.

Check Box: General Information

A check box is a control in which a user makes a choice between two mutually exclusive options.

A check box is defined by `PXCheckBox` in the Classic UI. In the Modern UI, a check box is defined by the `field` tag (whose control type is automatically defined as a check box from the backend code). In rare cases, a check box in the Modern UI is defined explicitly by the *`qp-check-box`* control.

Learning Objectives

In this chapter, you will learn the following information about a check box:

- The design guidelines for a check box, including the naming conventions and layout recommendations
- The proper configuration of a check box for specific cases, such as when a check box is located next to another element in the same row.
- A detailed description of each property of a check box

Applicable Scenarios

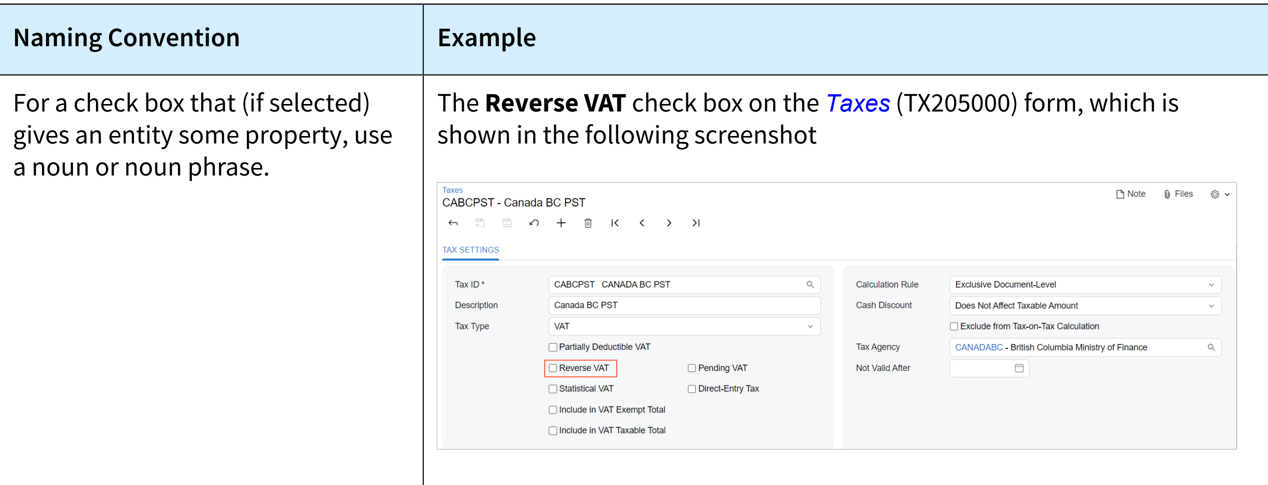
You configure check boxes in the following user scenarios:

- A user can select multiple items in a list of items, such as selecting items from a list of preferences.
- A user chooses one option from two possible options, such as enabling or disabling a feature.
- A user selects a specific criterion to apply to the displayed information to filter or sort data.

UI Naming Conventions

The following table shows the UI naming conventions for a check box.

Naming Convention	Example
For a check box that (if selected) enables an action, use a verb or verb phrase that describes this action.	The **Copy Tag Number from Asset ID** check box on the *Fixed Assets Preferences* (FA101000) form, which is shown in the following screenshot

Naming Convention	Example
For a check box that (if selected) gives an entity some property, use a noun or noun phrase.	The **Reverse VAT** check box on the *Taxes* (TX205000) form, which is shown in the following screenshot 

Recommendations for Organizing the Layout

The following table shows the recommendations for organizing the layout for check boxes.

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">When a fieldset contains only check boxes, align the check boxes without left padding.
To remove the left padding, specify <code>class="no-label"</code> in <code>qp-fieldset</code>, as shown below.
<code><qp-fieldset slot="C" class="no-label" ...>...</qp-fieldset></code></td></tr>
<tr><td>Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Figure: A correct layout</td><td>Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Figure: An incorrect layout</td></tr>
<tr><td colspan="2">Keep the default left padding for check boxes in a section if the section contains other controls and has a title.</td></tr>
<tr><td>Title
Field Name Value
Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Figure: A correct layout</td><td>Title
Field Name Value
Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Checkbox Name
Figure: An incorrect layout</td></tr>
</table>

Check Box: Layout Examples

In this topic, you can find examples of configurations of layouts with check boxes.

Check Box Next to a UI Element

Suppose that you want to place the **Canceled** check box next to the **Cancel By** box in the same row, as shown in the following screenshot.

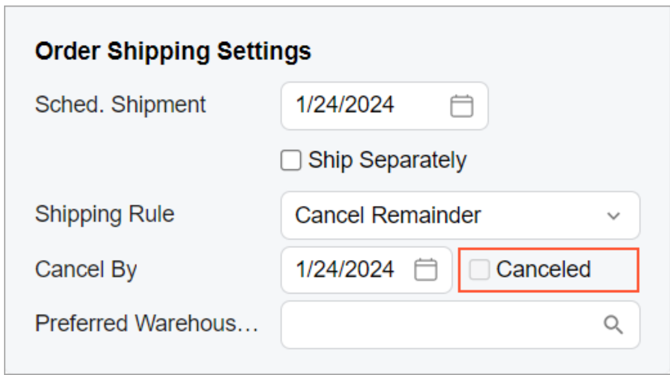


Figure: A check box next to a box

You use the following code.

```
<field name="CancelDate">
  <qp-field control-state.bind="CurrentDocument.Cancelled"
    config-enabled.bind="false"></qp-field>
</field>
```

You do not need to specify `class="no-label"` for the second control to omit the empty label of the check box because `class="no-label"` is used by default for a `qp-field` tag inside a `field` tag.

Multiple Check Boxes in a Row

Suppose that you need to display three check boxes in a row, as shown in the following screenshot.

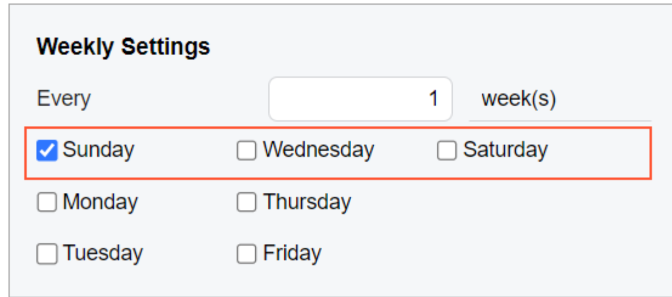


Figure: Multiple check boxes in a row

The following code implements this layout.

```
<field name="fake01" unbound replace-content>
    <qp-field control-state.bind="StaffScheduleSelected.WeeklyOnSun"
      config-enabled.bind="false" class="col-4"></qp-field>
    <qp-field control-state.bind="StaffScheduleSelected.WeeklyOnWed"
      config-enabled.bind="false" class="col-4"></qp-field>
    <qp-field control-state.bind="StaffScheduleSelected.WeeklyOnSat"
      config-enabled.bind="false" class="col-4"></qp-field>
</field>
```

You do not need to set `class="no-label"` to remove the label before each check box because this class is used by default in this case.

You have distributed the check boxes equally with `class="col-4"`, which means that each check box occupies four (12 / 3 = 4) columns out of 12 columns that the replaced field occupies. (If you need to have four columns and need to distribute them equally, you should use `class="col-3"`.)

Check Box: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to check boxes to HTML or TypeScript elements.

PXCheckBox

The following table shows the correspondence between the `PXCheckBox` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXCheckBox` `<px:PXCheckBox ... />`	Replace with `field` (whose control type is automatically defined as a check box from the backend code). In rare cases, you should replace it explicitly with the `qp-check-box` control.
`AlignLeft` `<px:PXCheckBox AlignLeft="True" ... />`	Use the `textAlign` property of the `config` attribute of the `qp-check-box` control. The property specifies whether the text of the check box should be aligned right or left. The following values are available: `right` and `left`. `<field ... control-type="qp-check-box" config.bind="{textAlign: 'right'}"> </field>`

ASPX	HTML or TypeScript

`CommitChanges`
`<px:PXCheckBox`
`  CommitChanges="True" ... />`
	Use `PXFieldOptions.CommitChanges` in Type-Script code.
	`export class DefLocation`
	`  extends PXView {`
	`  OverrideAddress:`
	`    PXFieldState<`
	`      PXFieldOptions.CommitChanges>;`
	`}`

`DataField`
`<px:PXCheckBox`
`  DataField="AutoPost" ... />`
	Use the `name` attribute of the `field` tag.
	`<field name="AutoPost">`
	`</field>`

`FalseValue`
`<px:PXCheckBox`
`  FalseValue="D" ... />`
	Use the `falseValue` property of the `config` at-tribute of the `qp-check-box` control. The property specifies the string and the boolean value that is used as the field value when the check box is cleared.
	`<field ...`
	`  control-type="qp-check-box"`
	`  config.bind="{falseValue: 'D'}">`
	`</field>`

`ID`
`<px:PXCheckBox`
`  ID="chkAutoPost" ... />`
	Replace with the `id` attribute of the `qp-field` tag if this tag is used as a replacement. In other cases, the ID is not necessary for a check box.

`RenderStyle`
`<px:PXCheckBox`
`  RenderStyle="Button" ... />`
	Use the `falseValue` property of the `config` at-tribute of the `qp-check-box` control. The property specifies how the check box is rendered. If you need to render the check box as a button, set the property to `button`.
	`<field name="ProcessingSucceeded"`
	`  config.bind="{`
	`    renderStyle: 'button'`
	`  }"`
	`>`
	`</field>`

ASPX	HTML or TypeScript
`TrueValue`	Use the `trueValue` property of the `config` attribute of the `qp-check-box` control. The property specifies the string or the boolean value that is used as the field value when the check box is selected.

```
<px:PXCheckBox
  TrueValue="A" ... />
```

```
<field ...
  control-type="qp-check-box"
  config.bind="{trueValue: 'A'}">
</field>
```

ASPX	HTML or TypeScript
`SuppressLabel`	Use `class="no-label"` for the `field` tag. (You do not need to use `class="no-label"` with `qp-field`, which does not have a label by default.)

```
<px:PXCheckBox
  SuppressLabel="True" ... />
```

```
<field name="Two" class="no-label">
</field>
```

CheckImages

The following table shows the correspondence between the `CheckImages` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`CheckImages`	Use the `checkImages` property of the `config` attribute of the `qp-check-box` control. The property specifies the image that should be used for the selected state of the check box if this check box is rendered as a button (see `renderStyle`). The image is an SVG icon from the `\Content\svg_icons` folder of the Acumatica ERP instance.

```
<px:PXCheckBox
  ID="edSuccessRecognition"
  runat="server"
  DataField="NumCheck"
  RenderStyle="Button"
>
  <CheckImages Normal="main@Success" />
</px:PXCheckBox>
```

```
<field name="ProcessingSucceeded"
  config.bind="{
    checkImages: {
      normal: 'main@Success' }
  }"
>
</field>
```

UncheckImages

The following table shows the correspondence between the `UncheckImages` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`UncheckImages`

```
<px:PXCheckBox
  ID="chkIsValid"
  runat="server"
  DataField="Valid"
  RenderStyle="Button"
>
  <UncheckImages Normal="main@Fail" />
</px:PXCheckBox>
```

</td><td>

Use the `uncheckImages` property of the `config` attribute of the `qp-check-box` control. The property specifies the image that should be used for the selected state of the check box if this check box is rendered as a button (see `renderStyle`). The image is an SVG icon from the `\Content\svg_icons` folder of the Acumatica ERP instance.

```
<field name="ProcessingSucceeded"
  config.bind="{
    uncheckImages: {
      normal: 'main@Fail' }
  }"
>
</field>
```

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

Obsolete ASPX Control and Property

The following table lists the obsolete ASPX element that is related to check boxes. You do not need to replace this ASPX element with any HTML or TypeScript elements.

ASPX Control	Property
`PXCheckBox`	`runat`

Collapsible Area

In this chapter, you will learn how to configure a collapsible area on an Acumatica ERP form. You will also learn how to configure the UI elements in this area to be pinned or unpinned by default.

Collapsible Area: General Information

A collapsible area is an area on an Acumatica ERP form that users can expand or collapse to show or hide its content. When the area is collapsed, only the pinned elements are displayed, saving screen space and directing users' attention toward important elements. When expanded, the collapsible area reveals additional content. Collapsible areas are useful when you want to manage screen space effectively while providing access to secondary or optional information.

In the Modern UI, a collapsible area is defined by the `qp-collapsible` attribute, which is added to the tag that defines the area.

Learning Objectives

In this chapter, you will learn about the proper configuration of a collapsible area and the elements of the area that are pinned by default.

Applicable Scenarios

You configure a collapsible area in the following cases:

- You need to manage a large number of elements on any part of an Acumatica ERP form. For example, if an area on an Acumatica ERP form contains multiple sections with more than five UI elements, you can use a collapsible area to give users the ability to expand the key information while keeping other content hidden.
- You want to hide optional or less frequently used content on an Acumatica ERP form.
- You want to give users control over the content of some part of an Acumatica ERP form. With collapsible areas, users can control what information they want to see.

Collapsible Area: Configuration

You can configure an area on an Acumatica ERP form to be collapsible. You can define either of the following as a collapsible area:

- A section that corresponds to a container
- A part of the form that corresponds to a template

When a part of the form is collapsed, only the pinned elements are displayed.

Configuring a Collapsible Area

To make an area of an Acumatica ERP form collapsible, you add the `qp-collapsible` attribute to the tag that defines the area, as shown in the following code example.

```
<qp-template name="7-10-7" id="document_form"
  qp-collapsible class="equal-height">
</qp-template>
```

If the form has any collapsible areas, the arrow button is displayed at the top right corner of the form, as shown in the following screenshot.

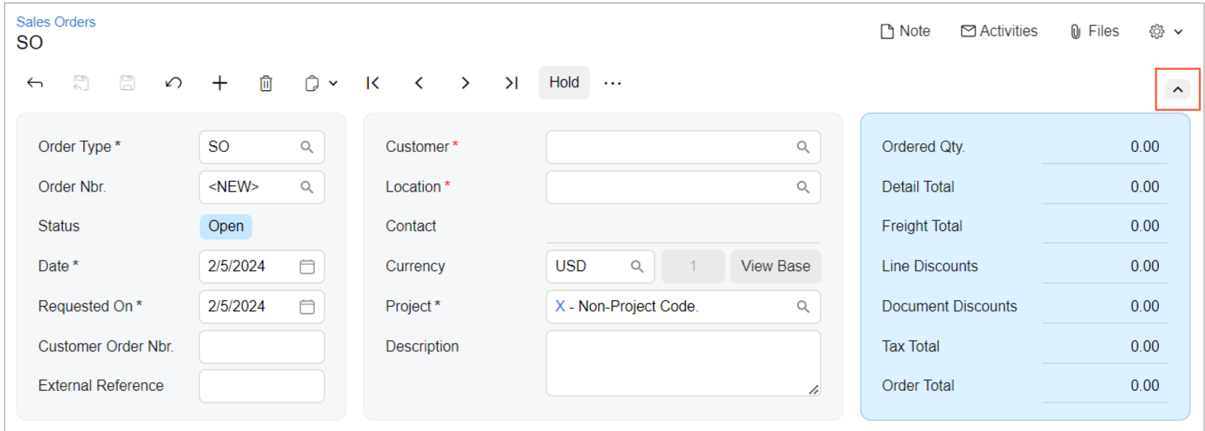


Figure: A Summary area with collapsible sections

When a user clicks the button, all containers with the `qp-collapsible` attribute are collapsed; only the pinned elements are displayed.

Sections with no pinned elements in the Summary or Selection area are displayed but empty (that is, they contain no elements), and their height is adjusted to the height of the other sections, as shown in the following screenshot. If the section has a title, the title is displayed.

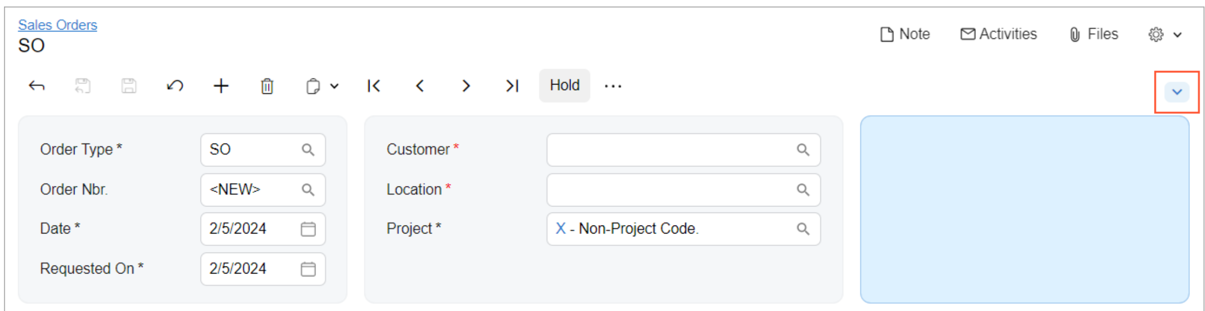


Figure: The Summary area with the collapsed sections

If the section is displayed on a tab and it has no pinned elements, only the title of the section is displayed without the space normally used by its elements, as shown with the sections on the right side of the form in the following screenshot.

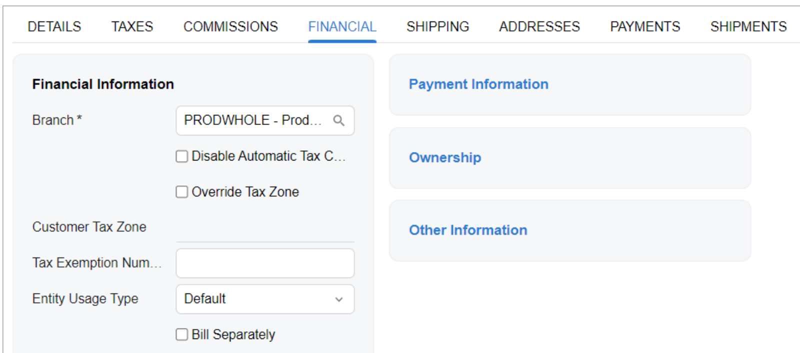


Figure: Collapsible sections on a tab

Defining Pinned Elements

By default, all required elements—that is, the elements that correspond to the fields with the `[PXUIFields(Required = true)]` attribute in the DAC—are pinned. A user can specify which elements are pinned in the **Section Configuration** dialog box, as shown in the following screenshot.

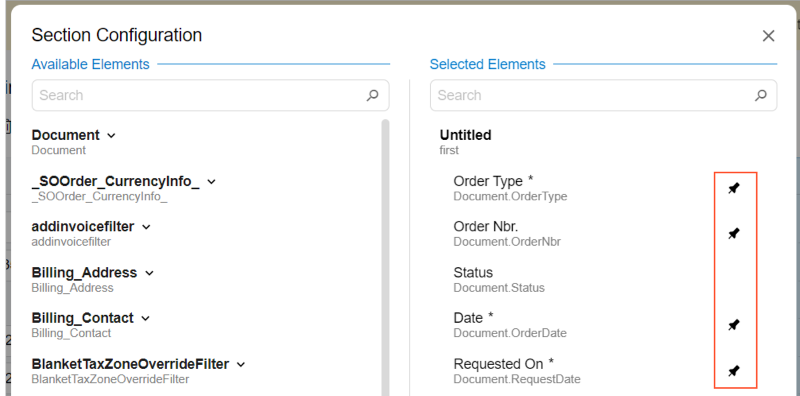


Figure: The Section Configuration

You can configure other elements to be pinned by default. To pin an element that corresponds to a field from code, you use the `pinned` attribute of the `field` tag, as shown in the following example.

```
<field name="OrderQty" pinned></field>
<field name="CuryDetailExtPriceTotal" pinned="true"></field>
```

To unpin an element that corresponds to a field, you specify `pinned="false"`, as shown in the following code.

```
<field name="OrderDate" pinned="false"></field>
```

Color Picker

In this chapter, you will learn about the configuration of the color picker control. You'll learn when to use this control and how to organize it in a layout.

Color Picker: General Information

A color picker is a group of controls and the Color Picker dialog box, which give a user the ability to select a color.

The following screenshot shows an example of this control.

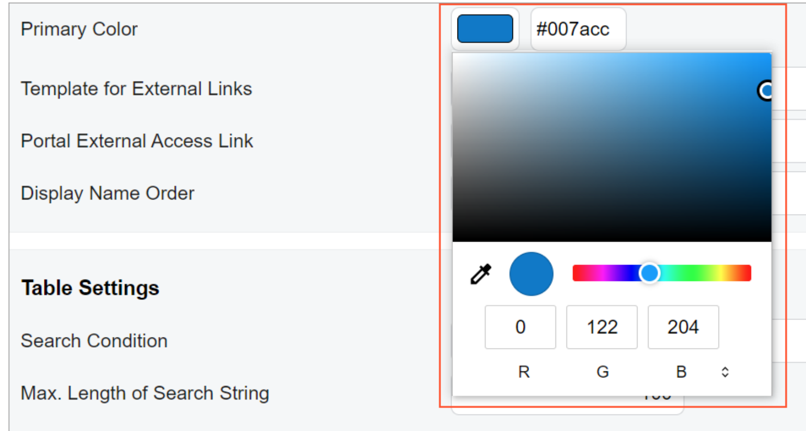


Figure: The color picker group of controls

In the Classic UI, the color picker control is defined by the `PXTextEdit` tag with the `TextMode="Color"` attribute. In the Modern UI, the color picker control is defined by the *`qp-color-picker`* tag.

Learning Objectives

In this chapter, you will learn the following about the color picker:

- The design guidelines for the color picker control
- The proper configuration of the color picker control

Applicable Scenarios

You configure the color picker control when you want a user select an arbitrary color from a dialog box.

Design Guidelines

A color picker is implemented as a `field` tag with a `control-type="qp-color-picker"` attribute, as shown in the following code.

```
<field name="PrimaryColor" control-type="qp-color-picker"></field>
```

The default value of the color picker is specified in the backend code.

You do not need to specify `control-type="qp-color-picker"` for fields that have the `PXColorState` state, such as the fields that have `PXColorListAttribute` specified on the DAC field.

Color Picker: Conversion from ASPX to HTML and TypeScript

The following table will help you to convert the ASPX element for the color picker control to an HTML element.

PXTextEdit

The following table shows the correspondence between the `PXTextEdit` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

PXTextEdit
 	
<px:PXTextEdit
  ID="edPrimaryColor"
  runat="server"
  DataField="PrimaryColor"
  CommitChanges="True"
  TextMode="Color" />
	HTML template:
	
	<qp-fieldset>
	  <field name="PrimaryColor">
	  </field>
	</qp-fieldset>
	
	TypeScript:
	
	PrimaryColor:
	  PXFieldState<PXFieldOptions.Com-
	mitChanges>;
	
	You need to specify con-
	trol-type="qp-color-picker"
	in HTML unless the field has the PXCol-
	orList attribute specified in the DAC.

Combo Box

In this chapter, you will learn about the configuration of combo boxes. You'll learn when to use combo boxes, how to name them, and how to organize a layout that includes them.

Combo Box: General Information

A combo box is a drop-down control that combines a text box with a list of options. It can consist of any number of values that a user can select. You can configure a combo box so that the user can select more than one value.

A combo box is defined by `PXDropDown` in the Classic UI. In the Modern UI, a combo box is defined by the `field` tag, whose control type is automatically defined as a combo box from the backend code. In rare cases, a combo box in the Modern UI is defined explicitly by the *`qp-drop-down`* control.

Learning Objectives

In this chapter, you will learn the following about a combo box:

- The design guidelines for a combo box, including the naming conventions
- The details of each property of a combo box

Applicable Scenarios

You configure combo boxes in the following user scenarios:

- A user needs to select a single item or multiple items from a finite list of items, such as selecting items from a list of preferences.
- A user needs to selects a specific criterion from a predefined list of options to apply a filter or sort data.

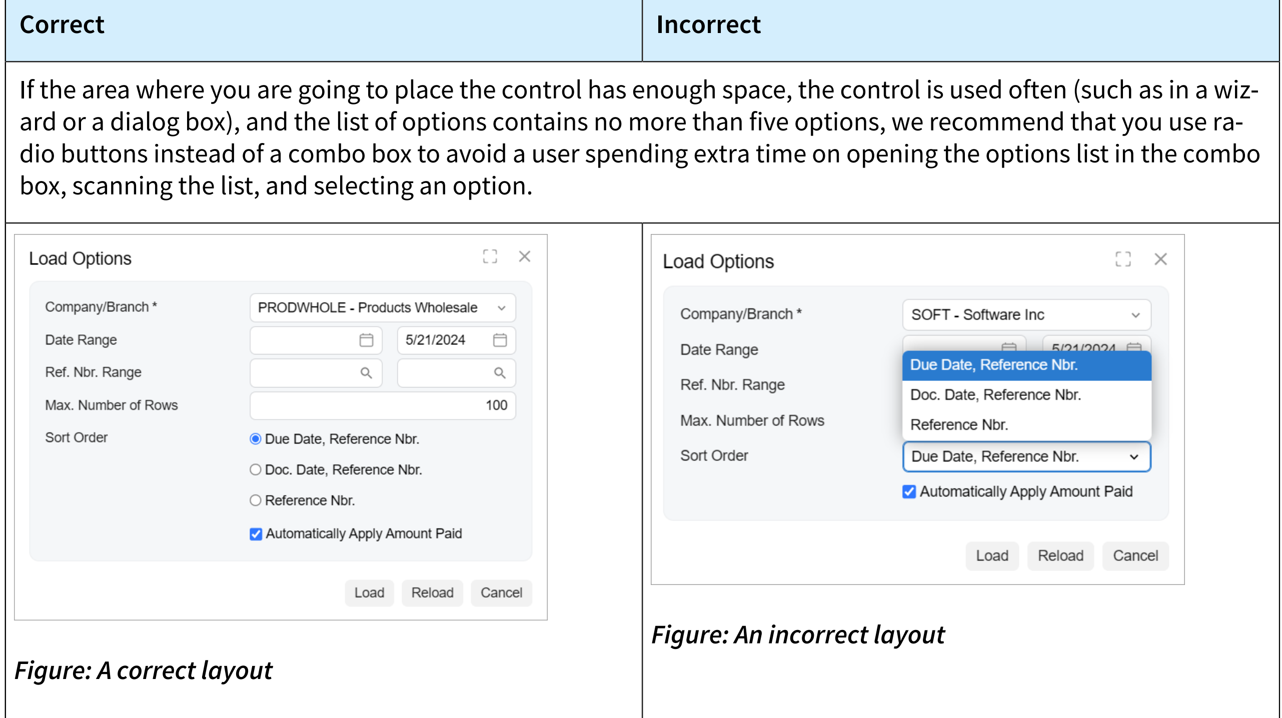
UI Naming Convention

The following table shows the UI naming convention for a combo box.

Naming Convention	Example
Use a noun or a noun phrase to describe the contents of a combo box. Preferably, combo box names should consist of one or two words.	The **Customer Status** combo box on the *Customers* (AR303000) form, which is shown in the following screenshot.

Recommendations for Organizing the Layout

The following table shows the recommendations for organizing the layout for combo boxes.



Correct	Incorrect
If the area where you are going to place the control has enough space, the control is used often (such as in a wizard or a dialog box), and the list of options contains no more than five options, we recommend that you use radio buttons instead of a combo box to avoid a user spending extra time on opening the options list in the combo box, scanning the list, and selecting an option.	
Figure: A correct layout	*Figure: An incorrect layout*

Combo Box: Configuration

In this topic, you can learn how to adjust a combo box for specific cases.

Defining a Drop-Down List

To configure a drop-down list, you use the `PXStringList` or `PXIntList` attribute in the definition of the data field in the data access class (DAC), as shown in bold type in the following example.

```
[PXDBString(1)]
[PXDefault(ShipmentStatus.OnHold)]
[PXUIField(DisplayName = "Status")]
[PXStringList(
    new string[]
    {
        ShipmentStatus.OnHold, ShipmentStatus.Shipping,
        ShipmentStatus.Cancelled, ShipmentStatus.Delivered
    },
    new string[]
    {
        "On Hold", "Shipping", "Cancelled", "Delivered"
    })]
public virtual string Status
{
    get;
    set;
```

```
}
```

You use `PXStringList` when the values that are assigned to the field are strings, and you use `PXIntList` when the values are integers.

In this example, `ShipmentStatus` is an enumeration defined in the following way.

```
public static class ShipmentStatus
{
    public const string OnHold = "H";
    public const string Shipping = "S";
    public const string Cancelled = "C";
    public const string Delivered = "D";
}
```

As parameters, you provide two arrays of strings:

- The array of values assigned to the field and saved to the database with the data record
- The array of labels displayed in the user interface

Modifying a Drop-Down List at Run Time

You can modify a drop-down list at runtime by using the `SetList<>()` static method of the `PXStringList` attribute. You can do this in the `RowSelected` event handler or graph constructor.

The following code example shows the use of the `SetList<>()` method.

```
PXStringListAttribute.SetList<Shipment.status>(
    sender, row,
    new string[]
    {
        ShipmentStatus.OnHold,
        ShipmentStatus.Shipping,
    },
    new string[]
    {
        "On Hold",
        "Shipping",
    });
```

This code sets a new list of values and labels for the `Status` field.

In the type parameter, you specify the data field associated with the control. You also provide the cache object, the data record that will be affected by the method, the list of values, and the list of labels.

If the list of possible values of a drop-down control is changed dynamically at runtime, you should use the `RowSelected` event handler to manage the list. Otherwise, we recommend that you create the list in the graph constructor.

Inserting a Not-Listed Value

If a drop-down list is configured with the `PXStringList` attribute, you can allow a user to enter values that are not options in the list. You do this by setting the `AllowEdit` property of the `PXDropDown` control to `True` on the ASPX page (see the setting in bold type in the following code).

```
<px:PXDropDown ID="edStatus" runat="server" DataField="Status"
               AllowEdit="True">
```

```
</px:PXDropDown>
```

Selecting Multiple Values

By default, a user can select one value from a drop-down list. The user will be able to select multiple values if you do all of the following:

- Set the `allowMultiSelect` property of the `config` attribute of the *`qp-drop-down`* control.

  ```
  @controlConfig({allowMultiSelect: true})
  TimeZone: PXFieldState;
  ```

- Set the `MultiSelect` property of the `PXStringList` attribute to `true`, as shown in the following code.

  ```
  [PXString(20)]
  [PXUIField(DisplayName = "Priority")]
  [PXStringList(
      new string[]
      {
      WorkOrderPriorityConstants.High,
      WorkOrderPriorityConstants.Medium,
      WorkOrderPriorityConstants.Low
      },
      new string[]
      {
      Messages.High,
      Messages.Medium,
      Messages.Low
      },
      MultiSelect = true)]
  public virtual string Priority { get; set; }
  ```

The selected values are displayed in the control separated by a comma. You can change the character that is used as a separator by using the `valuesSeparator` property of the `config` attribute of the *`qp-drop-down`* control.

Combo Box: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to combo boxes to HTML or TypeScript elements.

PXDropDown

The following table shows the correspondence between `PXDropDown` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXDropDown` `<px:PXDropDown ... />`	Replace it with `field` (whose control type is automatically defined as a combo box from the backend code). In rare cases, you should replace it explicitly with the `qp-drop-down` control.

ASPX	HTML or TypeScript

`AllowEdit`
 	
`<px:PXDropDown`
`  AllowEdit="True" ... />`
	Use the `allowEdit` property of the `config` attribute of the `qp-drop-down` control. If the property is set to *true*, a user can enter values that are not available in the combo box.
	
	`<field`
	`  config-allow-edit.bind="true" >`
	`</field>`

`AllowMultiSelect`
 	
`<px:PXDropDown`
`  AllowMultiSelect="True" ... />`
	Use the `allowMultiSelect` property of the `config` attribute of the `qp-drop-down` control. If the property is set to *true*, a user can select multiple values from the combo box.
	
	`@controlConfig({allowMultiSelect: true})`
	`TimeZone: PXFieldState;`

`AllowNull`
 	
`<px:PXDropDown`
`  AllowNull="True" ... />`
	Use the `allowNull` property of the `config` attribute of the `qp-drop-down` control. If the property is set to *true*, an empty value is allowed for the combo box.
	
	`<field`
	`  config-allow-null.bind="true" >`
	`</field>`

`AutoSuggest`
 	
`<px:PXDropDown`
`  AutoSuggest="True" ... />`
	Use the `autoSuggest` property of the `config` attribute of the `qp-drop-down` control. If the property is set to *true*, while a user is typing to search for a value in the combo box, the system should suggest matching values available in the combo box.
	
	`<field`
	`  config-auto-suggest.bind="true" >`
	`</field>`

`CommitChanges`
 	
`<px:PXDropDown`
`  CommitChanges="True" ... />`
	Use `PXFieldOptions.CommitChanges` in TypeScript code.
	
	`export class Account extends PXView {`
	`  Type:`
	`  PXFieldState<PXFieldOptions.Com-`
	`mitChanges>;`
	`}`

`DataField`
 	
`<px:PXDropDown`
`  DataField="AutoPost" ... />`
	Use the `name` attribute of the `field` tag.
	
	`<field name="AutoPost">`
	`</field>`

ASPX	HTML or TypeScript

Enabled
 	
<px:PXDropDown
  Enabled="False" ... />
	Use PXFieldOptions.Disabled in TypeScript code. The property specifies that the combo box is unavailable for editing.
	
	export class Account extends PXView {
	  Type:
	  PXFieldState<PXFieldOptions.Disabled>;
	}

ID <px:PXDropDown ID="chkAutoPost" ... />	If the qp-field tag is being used to replace the PXDropDown ASPX element, specify the id attribute of this tag. In other cases, the ID is not necessary for a combo box.
ValuesSeparator <px:PXDropDown ValuesSeparator=";" ... />	Use the valuesSeparator property of the config attribute of the qp-drop-down control. The property specifies the character that is used to separate selected values in the combo box while the allowMultiSelect property is set to *true*. <field config-value-separator.bind=";" > </field>

Obsolete ASPX Control and Properties

The following table lists the obsolete ASPX element that is related to combo boxes. You do not need to replace this ASPX element with any HTML or TypeScript element.

ASPX Control	Properties
PXDropDown	• NullText • runat • Size • SuppressLabel

Currency

In this chapter, you will learn about the configuration of the currency control. You will develop an understanding of when to use a currency control and how to configure it.

Currency: General Information

You use a currency control to display a currency on an Acumatica ERP form. This control provides advanced capabilities: Users can select a currency, view and select an exchange rate, and toggle between the foreign and base currencies.

In Classic UI a curremcy control is defined by `PXCurrencyRate`. In the Modern UI, a currency control appears and is used slightly differently on where it is defined on an Acumatica ERP form: as an advanced lookup box in a `qp-template` control or in a column of a table. You define a currency control by using the following controls in the Modern UI:

- `qp-currency` for the lookup box
- `qp-currency-selector` for the table column

Learning Objectives

In this chapter, you will learn the following:

- The design guidelines for the currency control, including the naming conventions and layout recommendations
- The proper configuration of the currency control

Applicable Scenarios

You configure the currency control in the following cases:

- Users need to specify a currency and a currency rate in a `qp-template` control.
- Users need to specify a currency in a table column.

Currency Control for the Lookup Box

As a looup box, a currency control has two modes:

- The foreign currency mode, which displays the foreign currency:

  The foreign currency mode is shown by default.
- The base currency mode

These views are described and shown below.

Foreign Currency Mode

The foreign currency mode displays the following:

- A currency selector (see Item 1 in the example in the following screenshot).

  When a user clicks the magnifier button of the currency selector, they can select a currency that has been defined on the Currencies (CM202000) form.
- An exchange rate between the selected foreign currency and the base currency (Item 2).

- The **View Base** button (Item 3), which a user can click to switch between the base currency and the current currency.

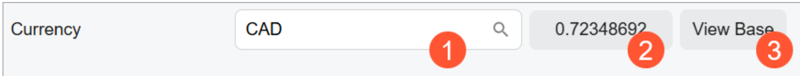


Figure: The default mode of the currency control

When a user clicks the exchange rate box (Item 2), the system opens the **Rate Selection** dialog box (see the following screenshot), where a user can configure the currency rate. The user selects the ID of the currency rate type and the date when the rate becomes effective. In the **Currency Unit Equivalents** section, the user can see the exchange rates between the selected foreign currency and the base currency.

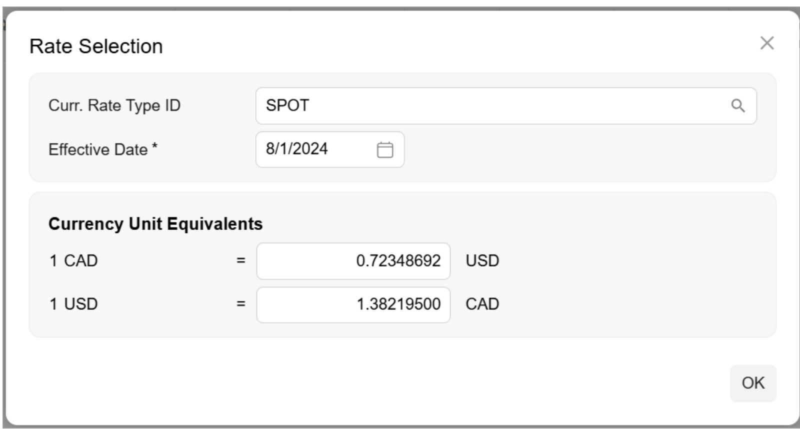


*Figure: The **Rate Selection** dialog box*

Base Currency Mode

The base currency mode is displayed when a user clicks **View Base** in the currency control. The base currency mode displays the following:

- The base currency.
- The **View Cury** button.

  By clicking the **View Cury** button, the user can switch to the default mode of the control (with **View Base** displayed).

Figure: The base currency mode of the currency control

Currency Control for the Table Column

A currency control in a table column is represented by a currency selector. When a user clicks the magnifier button, the **Currency Selection** dialog box is opened, as shown in the following screenshot. The user selects a currency and then the system displays the currency unit equivalencies. That is, it shows the exchange rates between the selected foreign currency and the base currency.

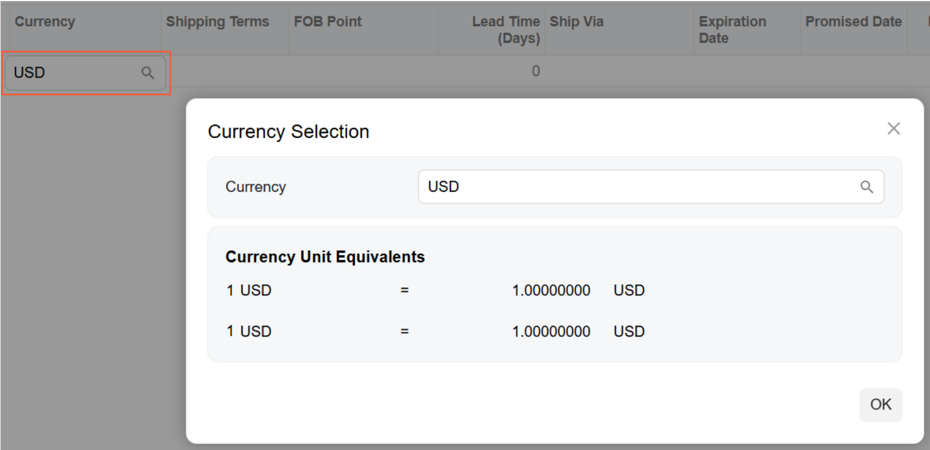


Figure: The Currency Selection dialog box

Currency ID

An ID of a currency control in HTML consists of two parts: the `cury` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a control that specifies the vendor currency may have the `curyVendorCurrency` ID, as the following code shows.

```
<qp-currency id="curyVendorCurrency"></qp-currency>
```

UI Naming Convention

The following table shows the UI naming conventions for the currency.

Naming Convention	Example
Use a noun or a noun phrase to describe the UI name of the currency box. Preferably, currency box names should consist of one or two words.	The **Currency** box on the *Sales Orders* (SO301000) form, which is shown in the following screenshot. Currency CAD 0.72348692 View Base

Currency: Configuration of the Currency Control

To configure a currency control, you need to do the following:

- Map a view for the control in TypeScipt.
- Add a field for the control in HTML or TypeScript.
  The way you add the control depends on whether it is located in a form view, or a table. See *Defining a Currency Control in the Form View* and *Defining the Currency Control in a Table*.
- Configure the type of the control and specify the view for it.

Defining the View

Suppose that the view for a currency control is defined in the graph as follows.

```
public PXSelect<CurrencyInfo, Where<CurrencyInfo.curyInfoID,
  Equal<Current<SOOrder.curyInfoID>>>> currencyinfo;
```

To define the view for the currency control in TypeScript, do the following:

1. Define a view class that implements the `ICurrencyInfo` interface, as shown in the following code.

```
export class CurrencyInfo extends PXView implements ICurrencyInfo {
 CuryInfoID: PXFieldState;
 BaseCuryID: PXFieldState;
 BaseCalc: PXFieldState;
 CuryID: PXFieldState<PXFieldOptions.CommitChanges>;
 DisplayCuryID: PXFieldState;
 CuryRateTypeID: PXFieldState<PXFieldOptions.CommitChanges>;
 BasePrecision: PXFieldState;
 CuryRate: PXFieldState<PXFieldOptions.CommitChanges>;
 CuryEffDate: PXFieldState<PXFieldOptions.CommitChanges>;
 RecipRate: PXFieldState<PXFieldOptions.CommitChanges>;
 SampleCuryRate: PXFieldState<PXFieldOptions.CommitChanges>;
 SampleRecipRate: PXFieldState<PXFieldOptions.CommitChanges>;
}
```

2. Create the instance of the view in the screen class, as shown in the following code.

```
export class SO301000 extends PXScreen {
  @viewInfo({ containerName: "Process Order" })
  _SOOrder_CurrencyInfo_ = createSingle(CurrencyInfo);
}
```

Defining a Currency Control in the Form View

Suppose that you need to define a currency control in the form view, and it should look as shown in the following screenshot.

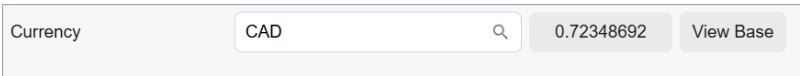


Figure: Currency control

To add a currency control to a form, you use the `field` tag inside a fieldset in the HTML template. In the `field` tag, you need to specify the following parameters:

- The type of the control: `qp-currency`
- The view that defines the currency data, which you defined in TypeScript

The following code shows implementation of the currency control in the screenshot above.

```
<field name="CuryID"
  control-type="qp-currency"
  view.bind="_SOOrder_CurrencyInfo_">
</field>
```

Defining the Currency Control in a Table

Suppose that you need to define a currency control in a table—that is, a **Currency** column with a currency selector, as shown in the following screenshot.

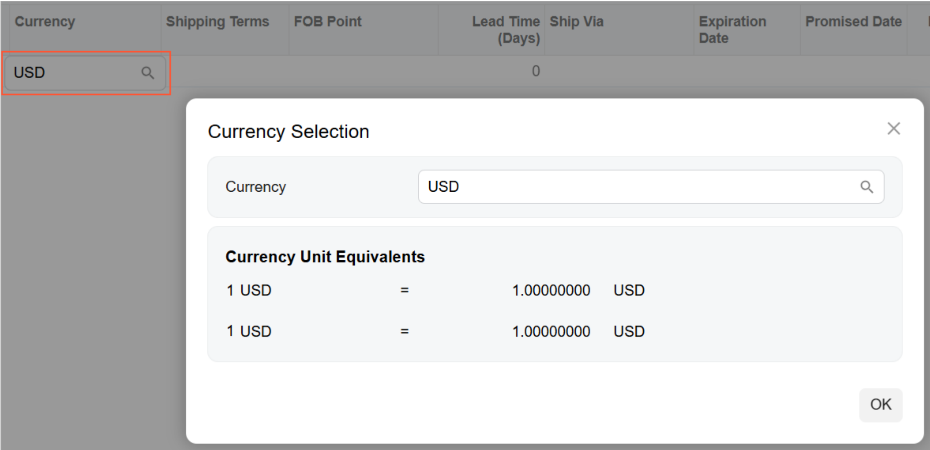


Figure: Currency selector

To add a currency control in a table column, you use the `columnConfig` decorator in TypeScript, where you need to specify the following parameters:

- The type of the control: `qp-currency-selector`.

  For this purpose, you need to use the `editorType` property: `editorType: "qp-currency-selector"`.
- The view that defines the currency data, which you defined in TypeScript.

  For this purpose, you need to use the `viewName` property of the `editorConfig` property.

The following code shows the implementation of the currency control in the screenshot above.

```
@columnConfig({
  editorType: "qp-currency-selector",
  hideViewLink: true,
  editorConfig: {
    viewName "_RQRequisition_CurrencyInfo_"
  },
})
CuryID: PXFieldState<PXFieldOptions.CommitChanges>;
```

Currency: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the currency control to HTML or TypeScript elements.

PXCurrencyRate

The following table shows the correspondence between the `PXCurrencyRate` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

PXCurrencyRate
 	
<pxa:PXCurrencyRate
  DataField="CuryID"
  ID="edCury"
  runat="server"
  DataSourceID="ds"
  RateTypeView="_SOOrder_CurrencyInfo_"
  DataMember="_Currency_">
</pxa:PXCurrencyRate>
	Use the field tag with control-type="qp-currency" in the HTML template.
	
	<field name="CuryID"
	  control-type="qp-currency"
	  view.bind="_SOOrder_CurrencyInfo_">
	</field>

DataField <pxa:PXCurrencyRate DataField="CuryID" ... > </pxa:PXCurrencyRate>	Use the name attribute of the field tag. <field name="CuryID" ...> </field>

RateTypeView
 	
<pxa:PXCurrencyRate
  ...
  RateTypeView="_SOOrder_CurrencyInfo_">
</pxa:PXCurrencyRate>
	Use the view.bind property of the field tag.
	
	<field name="CuryID" ...
	  view.bind="_SOOrder_CurrencyInfo_">
	</field>
	
	⚠ The view should be defined in TypeScript and should implement the ICurrencyInfo interface.

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the currency control. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
PXCurrencyRate	• DataMember • DataSourceID • runat

ASPX Control	Properties
`CurrencyEditor`	The control is no longer supported because the form view is no longer supported for a table. To define a column with the currency control, use the `qp-currency-selector` control. • `ID` • `SuppressLabel` • `Hidden` • `DataField` • `runat` • `DataSourceID` • `DataMember`

Data Feed

In this chapter, you'll learn about the data feed control and its configuration on Acumatica ERP forms. You'll also learn how to define data feed control templates and configure the control's toolbar.

Data Feed: General Information

In this topic, you'll learn what a data feed control is and what its basic components are.

Learning Objectives

In this chapter, you'll learn the following about the data feed control:

- Its design guidelines, including naming conventions and layout recommendations
- The proper configuration of the control for specific cases, such as defining templates of the control

Applicable Scenarios

You configure the data feed control when you want to display a list of entities, such as activities or tasks, where each item in the list contains a short summary of the entity.

Overview of the Data Feed Control

A data feed control lets users view multiple entities of different types and apply actions to these entities. The control offers flexibility in displaying, filtering, and organizing entities through configuration in both HTML and TypeScript. For example, you can configure a data feed control to show each entity on a separate tile and group entities by one of their properties.

The following example shows a data feed control configured to display activities of different types.

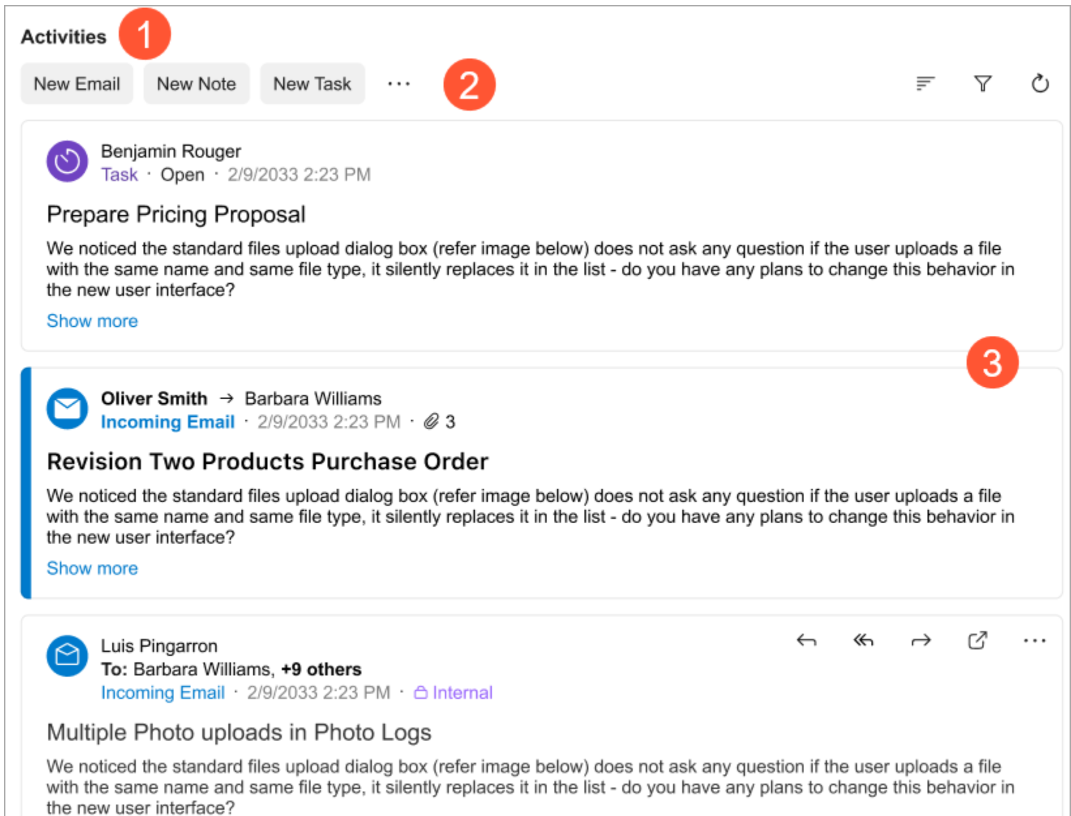


Figure: The data feed control

The data feed control includes the following components:

- A title (Item 1 above).
- The top toolbar, also called a header (Item 2).

  For information about configuring toolbar buttons, see *Data Feed: Configuration of Toolbar Buttons*.
- Tiles (Item 3).

  A *tile* is a container that displays any valid HTML. This HTML can contain information about a record, custom text, or values from multiple records that describe an entity. You can use predefined templates to organize content on a tile. For details, see *Data Feed: Configuration of Tiles*.
- The bottom toolbar (footer).

  Typically, the data feed control is configured to use infinite scroll, so the bottom toolbar isn't displayed.

  For information about configuring the bottom toolbar, see *Data Feed: Configuration of Bottom Toolbar*.

You can group entities in the data feed control based on a condition—for example, the entity's status. For each group, you can configure an HTML template that specifies which properties of the entity and which commands are displayed on the tile. The following example shows such a data feed where tiles are grouped by the status of an entity.

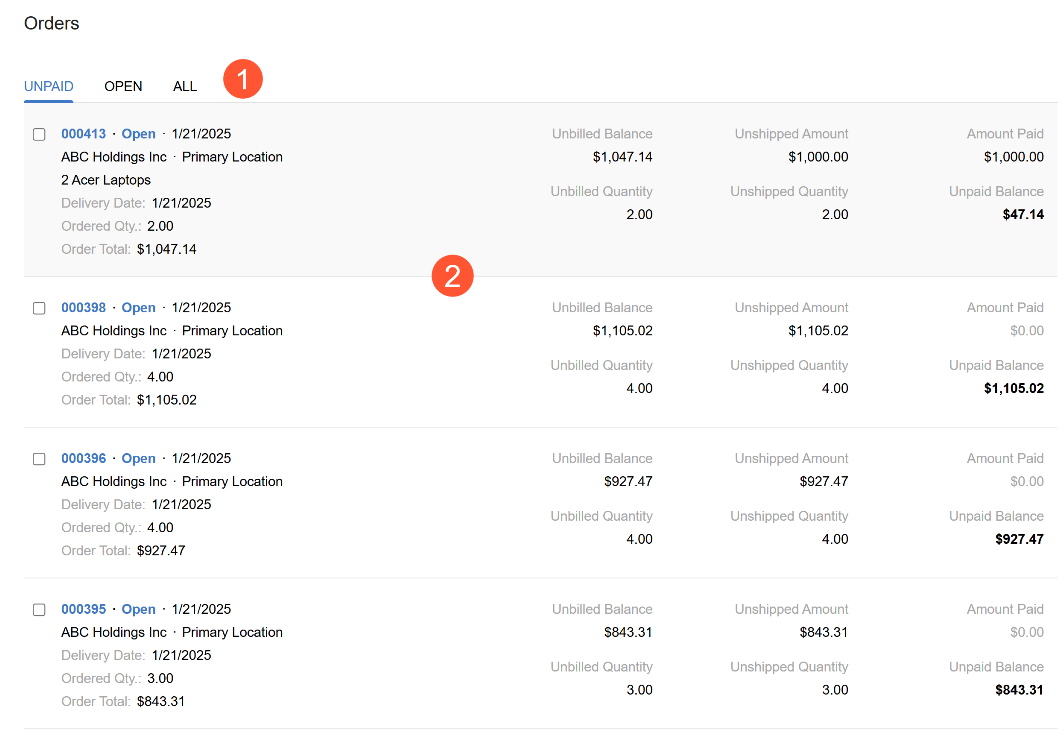


Figure: A group with a list of entities

Each container includes of the following components:

- A group title (Item 1 above)
- An optional menu with the available commands for the entities
- The list of entities of the container type (Item 2)
- An optional bottom toolbar of the tile

For information on configuring tiles, see *Data Feed: Configuration of Tiles*.

The data feed control is implemented with the *qp-data-feed* control in the Modern UI. The control does not have an analogue in the Classic UI.

Defining the Data Feed Control

The implementation of the data feed control is based on a table (grid). So to define a data feed control in TypeScript, you use the same approach as you use to define a table: Define a view class with the table's columns and actions, and specify the table properties in the `gridConfig` decorator. In HTML, you add the data feed control by using the `qp-data-feed` tag. For information on configuring the table in TypeScript, see *Table (Grid): Configuration of the Table and Its Columns*.

The data feed is bound to a particular view—a collection of records in the context of a screen—initialized by the `createCollection` method. But in the context of a tile (the `template` tag inside the data feed control), the view is treated as a single record, as if the `createSingle` method was used to initialize it.

To define the data feed control:

1. In TypeScript, define the view whose records should be displayed in the data feed.
2. If necessary, map the actions for the data feed control.
3. Initialize the view by using the `createCollection` method.

4. In HTML, add the `qp-data-feed` tag.
5. In the `qp-data-feed` tag, specify the needed properties of the control, including the following:
   - The view whose records should be displayed in the control
   - The view model
6. Optional: To add buttons to the toolbar of the data feed control, add the `toolbar` tag in the `qp-data-feed` tag and specify the corresponding actions in the `action` tag. For details, see *Data Feed: Configuration of Toolbar Buttons*.
7. For each entity type, define the template for the tile. For details, see *Data Feed: Configuration of Tiles*.
8. Optional: To define a bottom toolbar, add the `footer` tag in the `qp-data-feed` tag and specify the toolbar layout in it. For details, see *Data Feed: Configuration of Bottom Toolbar*.
9. Configure the message displayed in an empty data feed: In the `qp-data-feed` tag, specify the message in the `placeholderTemplate` property.

Data Feed ID

The data feed control is based on a table control and was represented by a table in the Classic UI. That's why the data feed control's ID should have the same structure and consist of:

- The `grid` prefix.
- The semantic name, which describes the control's purpose. For example, a table that shows transactions can have the `gridActivities` ID, as the following code shows.

  ```
  <qp-data-feed caption="Activities" id="gridActivities">
  ```

The controls inside a `qp-data-feed` control must have unique IDs. To ensure this, you specify IDs for all nested elements with the `$index` suffix, as shown in the following example.

```
<qp-data-feed id="MyDataFeed" ... >
    <template>
        <qp-template id="template_${$index}" name="record-1">
             <qp-fieldset id="fieldset_${$index}" slot="A1"
                   view.bind="LinkedFilesByEntity">
                   <field name="LinkedDocumentType""></field>
             </qp-fieldset>
         </qp-template>
         ...
    </template>
</qp-data-feed>
```

You don't need to specify IDs for fields inside a nested fieldset of `qp-template`. Their IDs are automatically appended with indexing suffixes as long as the fieldset is nested inside the `<qp-data-feed>` tag.

Related Links

- *Table (Grid)*
- *gridConfig*
- *QpDataFeedControl*

Data Feed: Configuration of Toolbar Buttons

By default, the toolbar of the data feed control contains standard table buttons that let users filter and sort records. To define a toolbar, you use the `toolbar` tag inside the `qp-data-feed` tag.

You can add buttons that correspond to actions implemented in a graph. To add toolbar buttons, you define each button with the `action` tag inside the `toolbar` tag.

Defining a Toolbar Button Without TypeScript

To define a button without mapping it in TypeScript, do the following:

1. In HTML, add the `toolbar` tag in the `qp-data-feed` control.

```
<qp-data-feed ...>
  <toolbar>
  </toolbar>
</qp-data-feed>
```

2. In the `toolbar` tag, add the `action` tag. In the `action` tag, specify the action's name in the `name` property.

   The value of the `name` property must be identical to the name of the action in the graph.

```
<toolbar>
  <action name="NewTask"></action>
  <action name="NewEvent"></action>
  <action name="NewMailActivity"></action>
  <action name="NewActivity"></action>
</toolbar>
```

Defining a Toolbar Button with TypeScript

To define a button by mapping it in TypeScript, do the following:

1. In TypeScript, in the view class for the data feed control, map the action implemented in the graph.

```
@gridConfig({ ... })
export class CRActivity extends PXView {
 NewTask: PXActionState;
 NewEvent: PXActionState;
 NewMailActivity: PXActionState;
 NewActivity: PXActionState;
 ...
}
```

   Here you can also configure the action with the `actionConfig` decorator.

2. In HTML, in the `qp-data-feed` control, add the `toolbar` tag.

```
<qp-data-feed ...>
  <toolbar>
  </toolbar>
</qp-data-feed>
```

3. In the `toolbar` tag, add the `action` tag. Then in the `action` tag, specify the name of the action in the `state` property.

   The value of the `name` property must be identical to the name of the action in TypeScript.

```
<toolbar>
  <action state="NewTask"></action>
  <action state="NewEvent"></action>
  <action state="NewMailActivity"></action>
```

```
  <action state="NewActivity"></action>
</toolbar>
```

You can specify an icon for an action or configure it in more detail by using the `config` property of the `qp-data-feed` control and the properties of the following interfaces:

- `IToolBarMenuButton`: For a regular menu button
- `IToolBarMenuOptions`: For a button that opens a menu
- `IToolBarMenuPopupButton`: For a button that opens a dialog box

Configuring a Button

You can further configuring a button in the `action` tag by using the properties in the following table.

Property	Example	Description
`index`	`<action state="NewTask" index="-80">`	Specifies the button's position on the toolbar.
`text`	`<action state="NewTask" text="New Task">`	Specifies the display name of the action.
`tooltip`	`<action state="NewTask" tooltip="Creates a new task">`	Specifies the button's tooltip text.
`icon`	`<action state="NewTask" icon="svg:main@plus">`	Specifies the icon.
`isSystem`	`<action state="NewTask" isSystem="true">`	Overrides the `isSystem` property of the action.

Related Links

- *QpDataFeedControl*

Data Feed: Configuration of Tiles

You can define each tile's layout and the list of properties displayed on it. You define tiles inside the `template` tag of the `qp-data-feed` tag.

To define the type of tile, you need to specify a condition in the `if.bind` property. For example, to define a tile for entities of the `Email` class, specify `<template if.bind="Activities.ClassID.cellText == 'Email'">...</template>`. If no condition is specified, the `template` tag defines tiles for all other entity types.

The following example shows definition of a tile.

```
<template width="100%">
  <div class="activity-card">
    <template if.bind="Activities.ClassID.cellText == 'Email'">
```

```
      <div style="background-color:${Activities.Color.cellText};">
${Activities.OwnerID.cellText}
          </div>
      <div>${Activities.ClassInfo.cellText} &middot; ${Activities.StartDate.cellText}
 &middot;
          ${Activities.Location.cellText}</div>
      <div>${Activities.Subject.cellText}</div>
      <qp-rich-text-editor state.bind="Activities.Body"
        config.bind="{readOnly: true, expandToContent: true, expandToContentMinHeight: 1}">
      </qp-rich-text-editor>
    </template>
  </div>
</template>
```

Overview of Tile Templates

Acumatica ERP provides several predefined templates that you can use to organize data on a tile. The name of a tile template always starts with `record-`.

You can use templates that start with `record-` not only in a data feed but anywhere on Acumatica ERP forms for formatting content.

Each template has the following types of slots:

- A prefix slot
- A number of columns with *primary* slots. A tile can have up to four columns with slots which are indicated by letters from A to D.
- A postfix slot

The following diagram shows the basic structure of the template.

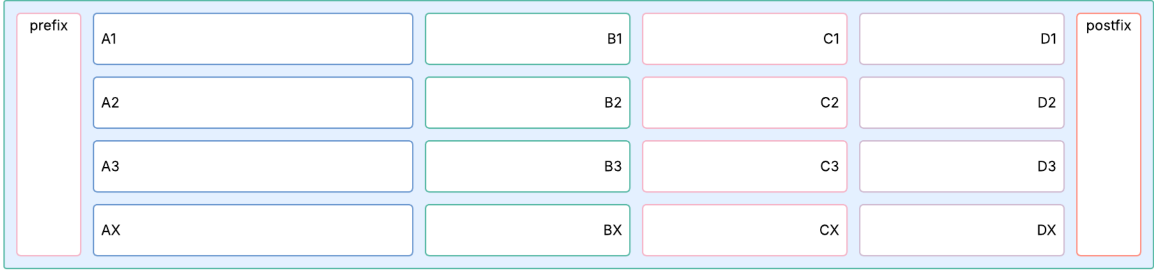


Figure: The base structure of a tile template

Every column can contain from one to four slots:

- Each slot's name consists of the combination of the letter which indicates a column and the row number—for example, *A1* for the first slot in column A is *A1*, and the second slot in column C is *C2*.
- Slots with numbers *1* to *3* (such as *A1, A2,* and *A3*) can display values separated by a gray dot.
- Each column also has the *X* slot (for example, *AX* or *BX*), where all values are placed one below the other.
- You can combine slots (*A1* and *A2*) and the *X* slot (for example, *A1* and *AX*), or just one type of slots.

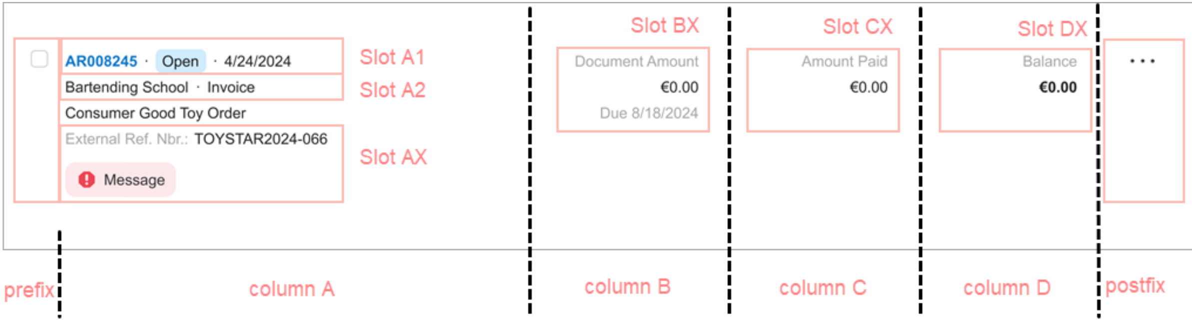


Figure: An example of the slot usage on a tile

Labels on tiles: On numbered slots (*A1–A3*), labels are not displayed. On *X* slots, labels are displayed by default.

Text alignment: By default, text is left-aligned in *AX* slots and right-aligned in *BX*, *CX*, and *DX* slots. All number values without labels (*A1–A3*) are left-aligned. To right-align text in a slot, use the *fieldset--left* CSS class.

Text formatting: By default, text on tiles gets the same formatting as in a regular fieldset. For example, status fields are highlighted with connotation colors and links appear in blue. For formatting, you can also use such CSS classes as *fieldset--text*, *fieldset--long-text*, *fieldset--header*, *fieldset--secondary*, and *fieldset--bold*.

Predefined Templates

For the predefined templates, the number of primary slots is indicated in the template name. For example, `record-1-1` and `record 1-2` both contain two slots (in addition to the prefix and postfix). The digit in the template name indicates the relative width of each slot for a full-width screen. That means that the second slot in `record-1-2` is wider than the second slot in `record-1-1`.

The following templates describe the default position of slots for a full-width screen. When the screen width changes, the width of primary slots automatically changes as well. The slots may even be visually merged.

The following table lists the predefined templates, along with examples and descriptions.

Name	Example	Description
`record-1`	VISA ****-1111 Exp. Date 01/26 · Default ... A tile with multiple fields in the *A1* slot and a menu in the postfix slot	The template contains one primary slot.
`record-1-1`	Travis Harper · Active Officer Portal Case Management Portal Company Management Portal Financial Management Portal Manager Portal Order Management Portal User tharper@abcstudios.con +1 (777) 228-0220 harper 10/7/2025 A tile with multiple fields in the *A1*, *AX*, and *BX* slots	The template contains two primary slots of equal width.

Name	Example	Description
`record-1-2`	5/22/2025 Branch: Products Retail Statement Balance $151.45 A tile with a prefix and fields in the *AX* and *BX* slots	The template contains two primary slots; the second primary slot is wider than the first one.
`record-1-2-compact`		The template contains two primary slots; the second primary slot is wider than the first one. This template is intended for narrow spaces (for example, when the data feed occupies one-third of the screen).
`record-1-1-1`		The template contains three primary slots of equal width.
`record-1-1-1-compact`		The template also contains three primary slots of equal width but is intended for narrow spaces (for example, when a data feed occupies one-third of the screen).
`record-3-2-2-2`	AR008245 · Open · 4/24/2024 Bartending School · Invoice Consumer Good Toy Order External Ref. Nbr.: TOYSTAR2024-066 Message Document Amount €0.00 Due 8/18/2024 Amount Paid €0.00 Balance €0.00	The template contains four primary slots. The first primary slot is wider than the others.

The following code shows the use of the `record-1-2` template that results in the screenshot in the table above.

```
<template width="100%">
 <div class="sp-documents-document no-animation">
  <qp-template id="dfCase" wg-container name="record-1-2">
   <qp-fieldset slot="Prefix" id="fsCasePrefix" view.bind="{{$viewName}}"
    class="record-item__prefix--date">
    <field name="StatementDateText" class="field--bold"></field>
   </qp-fieldset>
   <qp-fieldset slot="AX" id="fsCaseA" view.bind="{{$viewName}}"
    class="fieldset--condensed-sm fieldset--condensed-md fieldset--condensed-lg">
    <field name="AcctName"></field>
   </qp-fieldset>
   <qp-fieldset slot="BX" id="fsCaseB" view.bind="{{$viewName}}"
    class="fieldset--label-above-sm">
    <field name="CuryEndBalance_Text"></field>
    <field name="EndBalance_Text"></field>
   </qp-fieldset>
  </qp-template>
 </div>
</template>
```

CSS Classes for the record- Templates

The following table shows CSS classes that you can use to format content inside the `record-` templates. These classes cannot be used outside those templates.

Class Name	Example	Description
feed-vertical-divider		Displays a vertical line before the template. The class must be specified in a tag higher in the hierarchy than `qp-template`. <div class="feed-vertical-divider"> <qp-template id="dfCase" name="record-1-2"></qp-template> </div>
field--header		Intended for a header of the tile. The class should be applied to the `field` tag or, if `replace-content` is used, in nested tags.
field--text		Intended for plain text inside the template. Depending on the container size, the text may be limited to two, three, or four lines. The class should be applied to the `field` tag or, if `replace-content` is used, in nested tags.

Class Name	Example	Description
field--long-text	Company Profile Save Cancel scarlett@abc.co +12334445 scarlett@abc.co Travis Harper · Active **CMO** Portal Case Management Portal Company Management Portal Financial Management Portal Manager Portal Order Management tharper@abcstudios.con +1 (777) 228-0221 harper 2/5/2025	Formats plain text that should occupy multiple lines. The class should be applied to the `field` tag or, if `replace-content` is used, in nested tags.
field--left		Left-aligns the label text when the label is displayed above the value (see *Displaying of Labels and Values*). The class should be applied to the `field` tag or, in case `replace-content` is used, in nested tags.
field--bold	Unbilled Balance $1,047.14 $1,000.00 $1,000.00 Unbilled Quantity 2.00 Unshipped Quantity 2.00 Unpaid Balance **$47.14**	Makes text bold. The class is applied to the `field` tag or, in case `replace-content` is used, in nested tags.

Class Name	Example	Description
fieldset--condensed- • *fieldset--condensed-xs* • *fieldset--condensed-sm* • *fieldset--condensed-md* • *fieldset--condensed-lg*		Displays labels and values in the condensed mode (see *Displaying of Labels and Values*). If condensed mode is needed only for particular size of screen, specify the suffix. For example, if the field should be displayed in condensed mode only for *SM* and *XS* screens, specify `class="fieldset--condensed-xs fieldset-condensed-sm"`. The class is applied to the `qp-fieldset` tag.

<table>
<tr><th>Class Name</th><th>Example</th><th>Description</th></tr>
<tr><td>record-item-compound</td><td></td><td>Allows you to use multiple templates in a single <code>div</code> tag. By using this CSS class, you can define templates for different entities.
<pre><code><qp-data-feed id="fileList-
FeedByTag"
 view.bind="LinkedFilesBy-
Tag"
 class="no-toolbar sp-in-
voices-feed compact-feed">
 <template width="100%">
   <div
   class="record-item-com-
pound ..."
   >
     <div class="col-lg-4">
        <qp-template
 id="fileFormByTag1"
 name="record-1">
         ...
        </qp-template>
     </div>

     <div class="col-lg-4">
       <qp-template
 id="fileFormByTag2"
 name="record-1">
         ...
       </qp-template>
     </div>

     <div class="col-lg-4">
       <qp-template
 id="fileFormByTag3"
 name="record-1">
         ...
       </qp-template>
     </div>
   </div>
 </template>
</qp-data-feed></code></pre></td></tr>
</table>

Displaying of Labels and Values

The following table describes how field labels and values can be displayed on a tile.

<table>
<tr><th>Notation and Description</th><th>Example</th></tr>
<tr><td>Compressed: The label and value are displayed in one line, separated with a single space.

This notation is applied to fieldsets of the SM or XS size.</td><td>000413 · Open · 1/21/2025
ABC Holdings Inc · Primary Location
2 Acer Laptops
Delivery Date: 1/21/2025
Ordered Qty.: 2.00
Order Total: $1,047.14</td></tr>
<tr><td>Dot notation: The label and value are displayed in one line, separated with dots. This notation is used when the tile is displayed on a smaller screen.</td><td>Unbilled Balance: $1,047.14
Unbilled Quantity: 2.00
Unshipped Amount: $1,000.00
Unshipped Quantity: 2.00
Amount Paid: $1,000.00
Unpaid Balance: .... $47.14</td></tr>
<tr><td>With the label above the value: The label without a colon is displayed above the value.

This notation is applied to fieldsets of the MD or LG size.

By default, the labels in the left column are left-aligned, and the labels in the rest of the columns are right-aligned.

To left-align the label text, use the <code>field-left</code> class.</td><td>Unbilled Balance / $1,047.14 | Unshipped Amount / $1,000.00 | Amount Paid / $1,000.00
Unbilled Quantity / 2.00 | Unshipped Quantity / 2.00 | Unpaid Balance / $47.14</td></tr>
</table>

Recommendations for Organizing the Layout

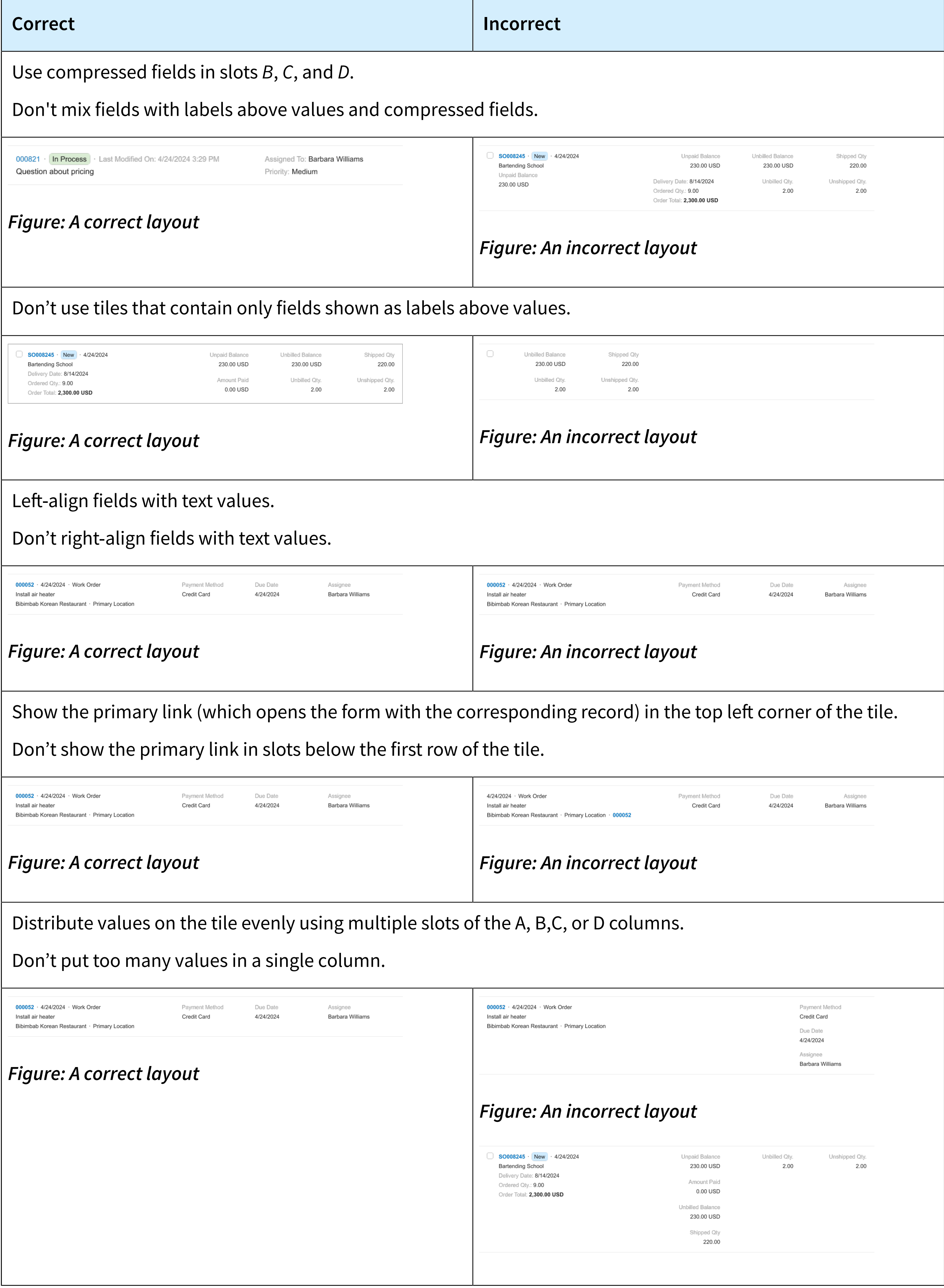
The following table shows layout recommendations for a data feed.

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">Use slots B, C, or D for fields with vertical alignment (values under their labels) and show them on the right side of the card.

Don't show compressed fields on the right side and fields with labels above values on the left side.</td></tr>
<tr><td>SO008245 · New · 4/24/2024
Bartending School
Delivery Date: 8/14/2024
Ordered Qty.: 9.00
Order Total: 2,300.00 USD
Unpaid Balance 230.00 USD · Unbilled Balance 230.00 USD · Shipped Qty 220.00
Amount Paid 0.00 USD · Unbilled Qty. 2.00 · Unshipped Qty. 2.00

Figure: A correct layout</td><td>Unpaid Balance 230.00 USD · Unbilled Balance 230.00 USD · Shipped Qty 220.00
Amount Paid 0.00 USD · Unbilled Qty. 2.00 · Unshipped Qty. 2.00
SO008245 · New · 4/24/2024
Bartending School
Delivery Date: 8/14/2024
Ordered Qty.: 9.00
Order Total: 2,300.00 USD

Figure: An incorrect layout</td></tr>
</table>



Correct	Incorrect
Use compressed fields in slots *B*, *C*, and *D*. Don't mix fields with labels above values and compressed fields.	
Figure: A correct layout	*Figure: An incorrect layout*
Don't use tiles that contain only fields shown as labels above values.	
Figure: A correct layout	*Figure: An incorrect layout*
Left-align fields with text values. Don't right-align fields with text values.	
Figure: A correct layout	*Figure: An incorrect layout*
Show the primary link (which opens the form with the corresponding record) in the top left corner of the tile. Don't show the primary link in slots below the first row of the tile.	
Figure: A correct layout	*Figure: An incorrect layout*
Distribute values on the tile evenly using multiple slots of the A, B,C, or D columns. Don't put too many values in a single column.	
Figure: A correct layout	*Figure: An incorrect layout*

Correct	Incorrect
Use compressed notation when you're displaying values in multiple columns. Don't use dot notation in a single slot when other columns have a different notation.	
Figure: A correct layout	*Figure: An incorrect notation*
Use dot notation for tiles displayed in narrow areas (such as side panels or narrow fieldsets). Don't display labels above values in narrow areas.	
Figure: A correct layout	*Figure: An incorrect notation*

Adding a Menu on a Tile

A tile can have a drop-down menu.

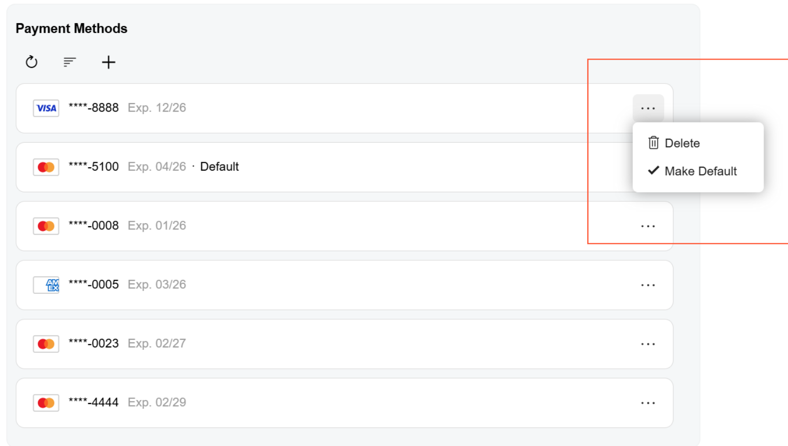


Figure: A menu of a tile

To add a menu to a tile, use the `qp-menu` control. The menu control should be located in the `postfix` slot of a tile, as shown above.

To configure the menu control, you need to define its configuration and the `MenuSelected` event handler. For details, see *Menu*.

The following code implements the data feed from the screenshot above. The `qp-menu` tag is located in the `postfix` slot of the `record-1` template.

```
<qp-data-feed id="dfPayMethods" view.bind="PaymentMethods" auto-resize="true">
  <template width="100%">
    <qp-template id="dfPayMethod" name="record-1" class="no-hover">
      <qp-fieldset slot="A1" id="fsContactA1" view.bind="{{$viewName}}">
    ...
      </qp-fieldset>
      <qp-menu slot="Postfix" config.bind="paymentMethodCommands({{$viewName}})"
        menuselected.delegate="processPaymentMethodMenuCommand({{$viewName}}, $event)">
      </qp-menu>
    </qp-template>
  </template>
</qp-data-feed>>
```

Data Feed: Configuration of Bottom Toolbar

You can configure the layout of a data feed's bottom toolbar in the `footer` tag inside the `qp-data-feed` tag.

The following example shows the definition of a bottom toolbar that shows either the **More Records** button or a text message. Each variant of the toolbar is defined in a `div` tag and is displayed based on the condition specified in the `if.bind` property.

```
<qp-data-feed ... >
  ...
  <footer>
    <div if.bind="controller.hasNextPage" class="ou-item ou-item--footer">
      <div
        id.bind="id & attr"
        class="ou-item__show-more"
        click.delegate="controller.appendPage()"
      >More Records</div>
    </div>
    <div if.bind="controller.isEmpty" class="ou-item ou-item--footer">
      <div class="ou-item__content">
        <div class="ou-item__no_records ou-item--disabled">No records found</div>
      </div>
    </div>
  </footer>
</qp-data-feed>
```

Configuring a Fixed Number of Tiles

You can configure a data feed to display a fixed number of tiles, even if the view may contain more records, as shown below.

The footer of the data feed control is not displayed in this case, and scrolling within the control isn’t available.

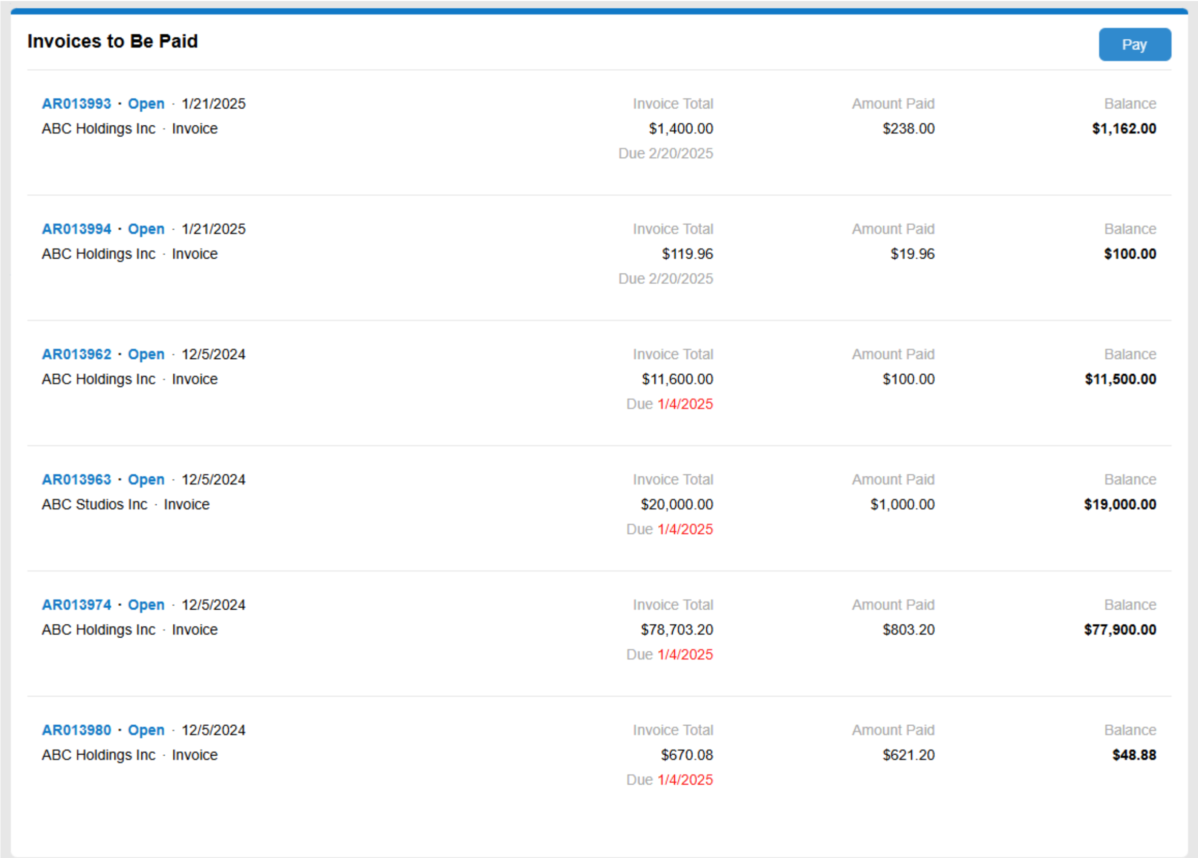


Figure: A data feed with no footer

To configure a data feed control that displays no more than a predefined number of records:

1. Specify the number of records in the `page-size` property of the `qp-data-feed` control.
2. Add an empty footer at the end of the `qp-data-feed` tag.

```
<qp-data-feed id="dfOpenInvoices" class="..." view.bind="OpenInvoices"
  page-size.bind="7">
    <template width="100%">
  ...
    </template>
    <footer></footer>
</qp-data-feed>
```

Data Feed: Configuration of Layout

You can configure the layout of tiles and field values in a data feed by adjusting how text is displayed and how tiles are arranged.

Field Values as Plain Text

If you want field values to be displayed as a plain text without the control functionality—for example, for the `qp-text-edit` or `qp-selector` control—do the following:

1. In the DAC, make sure to disable all fields that should be displayed as plain text.

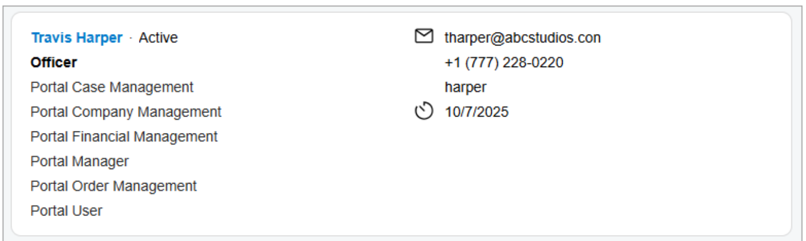
If one of the controls should remain available for editing, enable that control in the DAC.

2. Put the `qp-data-feed` tag inside the `qp-text-light-scope` tag.

The following code shows the use of the `qp-text-light-scope` tag.

```
<qp-text-light-scope>
  <qp-data-feed id="dfContacts" view.bind="Contacts" auto-resize="true">
  ...
  </qp-data-feed>
</qp-text-light-scope>
```

The resulting data feed is shown below.



Tiles in a Single Line

To put multiple tiles in a single line, specify the width of each tile, as shown below.

```
<qp-data-feed id="dfItems" class="no-toolbar" width="180px"
  view.bind="FilteredItems" page-size.bind="20">
  <template>
    <qp-include url="../catalog/catalog.html" view="{{$viewName}}">
    </qp-include>
  </template>
</qp-data-feed>
```

You can also create a CSS class that specifies the tile width. The following example shows a class that specifies the default width and calculates the width for screens that are narrower than 600 pixels.

```
.sp-feed-item {
  width: 186px;

  @media (width < 600px) {
    width: calc(round(down, 50vw - var(--sp-column-gap) * 1.5, 1px));
  }
}
```

Date and Time

In this chapter, you will learn about the configuration of date and time controls. You will develop an understanding of when to use date and time controls and how to name them.

Date and Time Control: General Information

A date and time control displays the date and the time, only the time, or only the date. By clicking the calendar button in the date box, a user can select the date in the calendar, as shown in the following screenshot. By clicking the clock button in the time box, a user can select the time from the drop-down list.

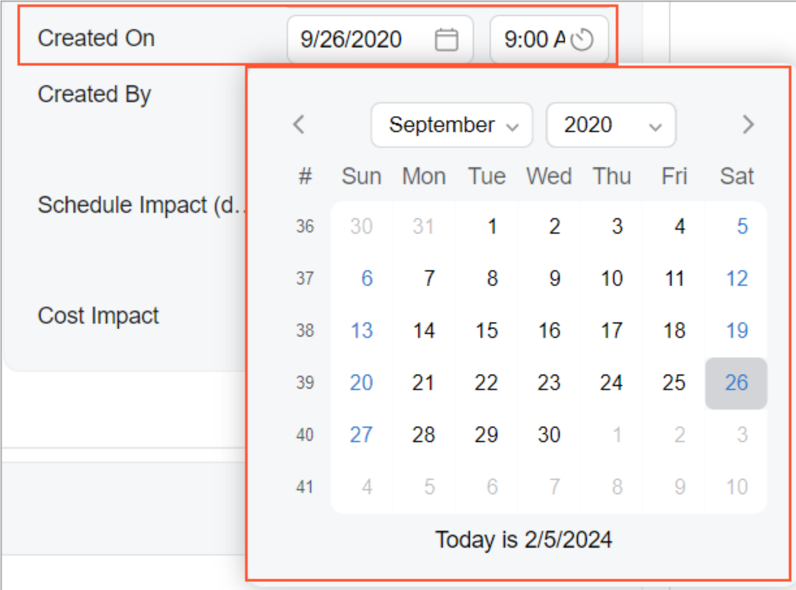


Figure: A date and time control

A date and time control is defined by `PXDateTimeEdit` in the Classic UI. In the Modern UI, you use the `field` tag inside a fieldset. If you need to display the date and the time separately in two boxes, you also use a nested `qp-field` control inside the `field` tag. The control type in the Modern UI is defined automatically from the backend code. In rare cases, you may need to explicitly use the *`qp-datetime-edit`* or *`qp-datetime-edit-utc`* tag.

Learning Objectives

In this chapter, you will learn the following about a date and time control:

- The design guidelines for a date and time control, including the naming conventions
- The proper configuration of a date and time control for specific cases, such as a control with both the date and the time
- A detailed description of each property of a date and time control

Applicable Scenarios

You use date and time controls in the following scenarios:

- On data entry forms where users need to input or modify temporal information. The use of date and time controls ensures accurate and standardized input formats, reducing errors and enhancing data consistency.

- On forms that involve scheduling, calendar management, or event planning. Users can select specific dates and times for appointments, meetings, or other time-sensitive activities.
- On forms that are designed for reporting or analytical purposes, to filter or aggregate data based on temporal parameters.
- In the implementation of notifications. Users can make the system trigger automated actions at specified dates and times.
- In automated processes, such as scheduled tasks, which frequently rely on date and time controls to initiate actions at predefined intervals.

Date and Time Control ID

The `field` tag doesn't have an ID. If the control is defined as a `field` with a nested `qp-field`, the `qp-field` tag has an ID that consists of two parts, the `field` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a time field for the contact export date and time control can have the `fieldContactExportTime` ID, as the following code shows.

```
<field name="ContactsExportDate_Date">
  <qp-field
    id="fieldContactExportTime"
    control-state.bind=
      "EmailSyncAccountFilter.ContactsExportDate_Time"
  </qp-field>
</field>
```

UI Naming Conventions

The following table shows the UI naming conventions for date and time controls.

Naming Convention	Example
Use nouns or noun phrases that describe the content of a date and time control. Preferably, box names should consist of no more than two words.	The **Date** box on the *Invoices* (SO303000) form, which is shown in the following screenshot

Date Format Settings

When the year of the date is displayed with two digits—for example, *25*—the actual year is calculated as follows:

- A number between 50 and 99 is treated as the year in the XXth century. For example, *50* specifies the year 1950.

- A number between 0 and 49 is treated as the year in the XXIth century. For example, *49* specifies the year 2049.

This setting corresponds to the current setting in .NET.

Date and Time Control: Configuration

In this topic, you can learn how to adjust a date and time control.

Date and Time Control Definition

To define a date and time control, in the data access class (DAC), you need to add one of the date and time attributes (such as *PXDate*, *PXDateAndTime*, *PXDBDate*, *PXDBTime*, or *PXDBDateAndTime*) to the property field, as shown in the following code.

```
public abstract class effectiveAsOfDate :
  PX.Data.BQL.BqlDateTime.Field<effectiveAsOfDate> { }
[PXDBDate]
[PXUIField(DisplayName = "Effective As Of")]
public virtual DateTime? EffectiveAsOfDate { get; set; }
```

In the TypeScript and HTML code, you then define a field with no additional settings specified. For details, see *UI Definition in HTML and TypeScript: General Information*.

Separate Boxes for Date and Time

If you need to display the date and the time in two separate boxes, you define the date and time control, as shown in the following example. You use a nested `qp-field` control with the `timeMode` property set to *true* in `config`.

```
<field name="ContactsExportDate_Date">
  <qp-field
    class="col-3 no-label"
    control-state.bind=
      "EmailSyncAccountFilter.ContactsExportDate_Time"
    config-time-mode.bind="true">
  </qp-field>
</field>
```

Input Mask and Display Mask

The date and time mask configuration are taken from the locale preferences and do not require any additional setup. However, if you need to specify the input mask and the display mask, you specify the value of the `InputMask` or `DisplayMask` property of the date and time attribute that is assigned to the DAC property field. You use the *standard* and *custom* date and time format strings.

The following example shows the use of the `InputMask` and `DisplayMask` properties.

```
public abstract class parameter1 :
  PX.Data.BQL.BqlDateTime.Field<parameter1> { }
[PXDateAndTime(DisplayMask = "D", InputMask ="d")]
[PXUIField(DisplayName = "Parameter 1")]
public virtual DateTime? Parameter1 { get; set; }
```

Relative Dates

The date and time control can display relative dates—that is, values such as *@Today* or *@WeekStart* (shown below).

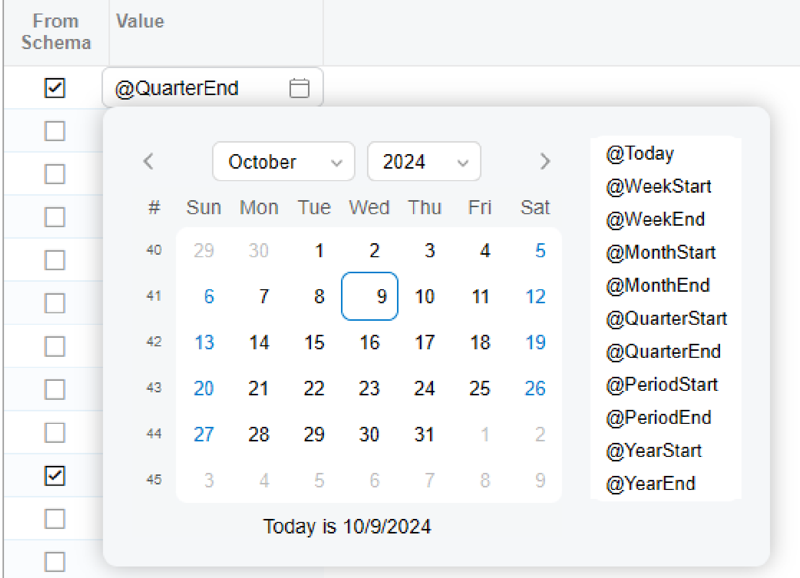


Figure: Relative dates

To specify that a user can enter relative dates in the control, you need to implement the `RowSelected` event handler. In the event handler, you call the `PXDatetimeFieldState.showRelativeDates` method to indicate that the values are allowed.

An example of such a handler is shown in the following code.

```
@handleEvent(CustomEventType.RowSelected, { view: "Rules" })
  onEPRuleConditionSelected(
    args: RowSelectedHandlerArgs<PXViewCollection<EPRuleCondition>>)
{
    const ar = args.viewModel.activeRow;

    ar.Value?.to(PXDatetimeFieldState).showRelativeDates();
}
```

Date and Time Control: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert ASPX elements that are related to date and time controls to HTML or TypeScript elements.

PXDateTimeEdit

The following table shows the correspondence between the `PXDateTimeEdit` ASPX control and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXDateTimeEdit`

```
<px:PXDateTimeEdit
  CommitChanges="True"
  ID="edExportedDate_Date"
  runat="server"
  DataField="ExportDate_Date" />
<px:PXDateTimeEdit
  CommitChanges="True"
  ID="edExportedDate_Time"
  runat="server"
  DataField="ExportDate_Time"
  TimeMode="true"
  SuppressLabel="True"
  Width="84" />
```

</td><td>

To display two separate boxes for date and time, in the TypeScript code of the form, define two fields—a field for the date and a field for the time—as shown in the following code.

```
export class EmailSyncFilter
  extends PXView {
  ExportDate_Date:
    PXFieldState<
      PXFieldOptions.CommitChanges>;
  ExportDate_Time:
    PXFieldState<
      PXFieldOptions.CommitChanges>;
}
```

In HTML code of the form, use the `field` tag for the date field and a nested `qp-field` tag for the time field, as the following code shows.

```
<field name="ExportDate_Date">
  <qp-field
    id="fieldExportTime"
    class="col-3 no-label"
    control-state.bind=
      "EmailSyncFilter.ExportDate_Time"
    config-time-mode.bind="true">
  </qp-field>
</field>
```

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

`CommitChanges`

```
<px:PXDateTimeEdit
  CommitChanges="True"
  DataField="ExportDate_Date" />
```

</td><td>

Use the `CommitChanges` option of `PXFieldState` in TypeScript, as the following code shows. With this option, the control commits changes to the server each time a user has changed the value in the box and focus is no longer on the box.

```
export class EmailSyncFilter
  extends PXView {
  ExportDate_Date:
    PXFieldState<
      PXFieldOptions.CommitChanges>;
}
```

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

ASPX	HTML or TypeScript

`DataField`
 	
`<px:PXDateTimeEdit`
`  DataField="ExportDate_Date"/>`
`<px:PXDateTimeEdit`
`  DataField="ExportDate_Time"/>`
	To specify the data fields to be used for the date and time control, use the `name` attribute of the `field` tag for the date field, and `control-state.bind` of the `qp-field` tag for the time field, as the following code shows.
	
	`<field name="ExportDate_Date">`
	`  <qp-field`
	`    control-state.bind=`
	`      "EmailSyncFilter.ExportDate_Time"`
	`  </qp-field>`
	`</field>`

`ID`
 	
`<px:PXDateTimeEdit`
`  ID="edExportedDate_Date"`
`/>`
`<px:PXDateTimeEdit`
`  ID="edExportedDate_Time"`
`/>`
	Use the `id` attribute of the nested `qp-field` tag for the time field, as the following code shows. The `id` attribute is not used for `field` tags.
	
	`<field name="ExportDate_Date">`
	`  <qp-field`
	`    id="fieldExportTime"`
	`    control-state.bind=`
	`      "EmailSyncFilter.ExportDate_Time"`
	`  </qp-field>`
	`</field>`

`SuppressLabel`
 	
`<px:PXDateTimeEdit`
`  DataField="ExportDate_Time"`
`  SuppressLabel="True" />`
	To hide the label for a field, use `class="no-label"` for the field, as the following code shows. For details about the class, see *Form Layout: CSS Classes*.
	
	`<field name="ExportDate_Date">`
	`  <qp-field`
	`    control-state.bind=`
	`      "EmailSyncFilter.ExportDate_Time"`
	`    class="no-label"`
	`  </qp-field>`
	`</field>`

`TimeMode`
 	
`<px:PXDateTimeEdit`
`  DataField="ExportDate_Time"`
`  TimeMode="true"/>`
	To indicate that the field contains time, use the `timeMode` property of `config` of the field, as shown in the following code.
	
	`<field name="ExportDate_Date">`
	`  <qp-field`
	`    control-state.bind=`
	`      "EmailSyncFilter.ExportDate_Time"`
	`    config-time-mode.bind="true">`
	`  </qp-field>`
	`</field>`

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`Width`

```
<px:PXDateTimeEdit
  DataField="ExportDate_Time"
  Width="84" />
```

</td><td>

Use the `col-N` class to define the width of the field, as the following code shows. In `col-N`, *N* is the number of columns the field occupies, which can be a whole number between 1 and 12. For details about the class, see *Form Layout: CSS Classes*.

```
<field name="ExportDate_Date">
  <qp-field
    class="col-3"
    control-state.bind=
      "EmailSyncFilter.ExportDate_Time"
  >
  </qp-field>
</field>
```

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

Obsolete ASPX Controls and Properties

The following table lists obsolete ASPX elements that are related to date and time controls. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Property
`PXDateTimeEdit`	`runat`

Dialog Box

In this chapter, you will learn about the configuration of dialog boxes—that is, pop-up panels or windows that are opened on Acumatica ERP forms when a user clicks a button or link. You'll learn when to use dialog boxes, how to name them, and how to organize their layout.

Dialog Box: General Information

A dialog box is a pop-up panel or window that opens when a user clicks a button or link. In Acumatica ERP, it can be either of the following:

- A modal pop-up panel that gives the user information and prompts them to enter settings or click a button (or both) before continuing. This type of dialog box may be opened when the user clicks a button or a command.
- A pop-up window that contains an Acumatica ERP form. This type of dialog box is not modal and is usually opened when the user clicks a link on a form. You define the form that opens in the pop-up window as you define any other form.

A dialog box is defined by `PXSmartPanel` in the Classic UI. In the Modern UI, a dialog box can be defined by the *`qp-panel`* control, or in rare cases, by the `qp-dialog` control.

The `qp-panel` control is not specifically a control for creating a dialog box; it is a generic placeholder for an HTML template that can be used in various other contexts, such as creating a tab or wizard.

Learning Objectives

In this chapter, you will learn the following:

- The design guidelines for a dialog box, including the naming conventions and layout recommendations
- The details of each property of a dialog box that is created by using the `qp-panel` control

Applicable Scenarios

You configure a dialog box in the following user scenarios:

- A user needs to enter data on a form that is related to another entity without having to navigate to that entity's form.
- A user has clicked a button or command, and you want the user to provide additional information before the system invokes the associated action.
- A user has clicked a button, and you want the user to verify or cancel the intention to continue before the action is invoked.
- A user has clicked a link with a record's identifier, and the system should open a pop-up window to show the selected record on the corresponding form.

Types of Dialog Boxes

The following table lists different types of dialog boxes and the corresponding controls.

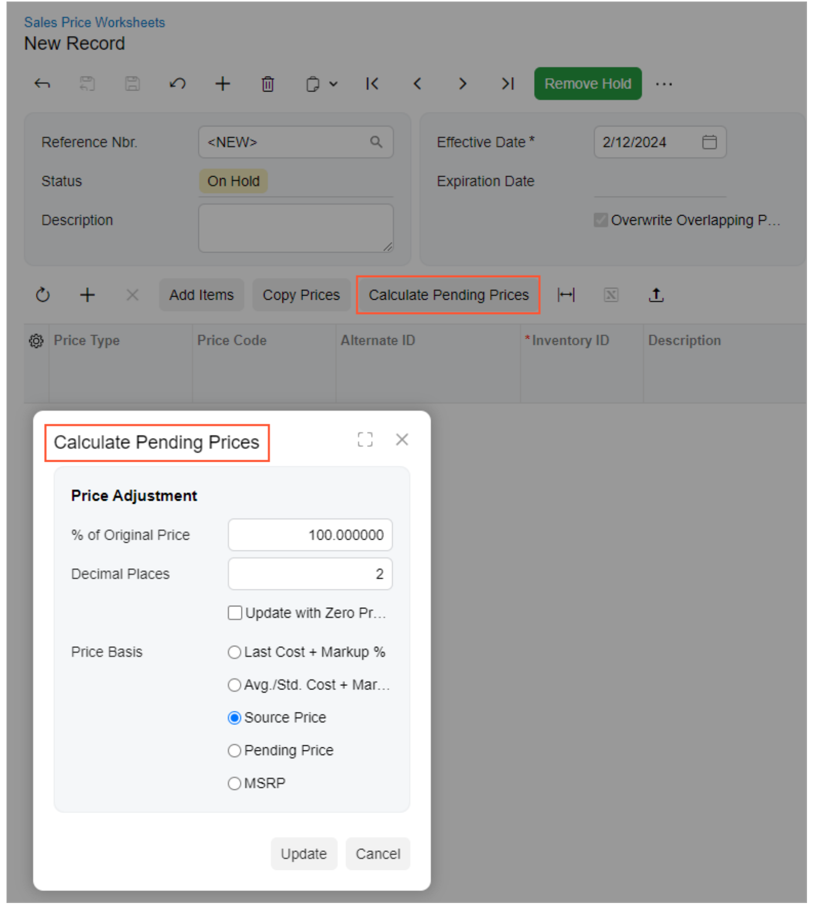
Control	Purpose	Documentation
`qp-panel`	Smart panel, simple dialog box	*Dialog Box*
No control is needed.	Workflow dialog box	*Implementing Workflow Dialog Boxes*
`qp-upload-dialog`	Dialog box for uploading files	*Upload Dialog Box*
`qp-multi-upload`	A button and a dialog box for uploading multiple files at a time	*Upload Files Button*

UI Naming Conventions

The following table shows the UI naming conventions for a dialog box.

Naming Convention	Example
Dialog box used to confirm an action that the user has invoked by clicking a button or command: Use the button or command name (or a part of it) as the name of the dialog box caption.	When a user clicks the **Switch to Modern UI** button on the *Site Map* (SM200520) form, the dialog box shown below opens. Thus, the dialog box is also named **Switch to Modern UI**.

<table>
<tr><th>Naming Convention</th><th>Example</th></tr>
<tr><td>For a dialog box used to provide additional information before the system can execute the action that the user has invoked by clicking a button or command: Use the button or command name (or a part of it) as the name of the dialog box caption.</td><td>When a user clicks the Calculate Pending Prices button on the Sales Price Worksheets (AR202010) form, the following dialog box opens. It should be named Calculate Pending Prices (as shown below) or Calculate Prices.


</td></tr>
<tr><td>For a dialog box used to confirm a change, such as a change in the value of a box that causes changes in other boxes: Use a clear summary as the name of the dialog box caption. The name should describe the change that the user needs to confirm.</td><td>Suppose that a user has made a change on a form that will cause an address on the form to be replaced. The name of the dialog box that confirms this change should be Replace Address.</td></tr>
</table>

Naming Convention	Example
For a dialog box that asks the user a question or offers multiple choices: Use a short summary of the choices or the question that you want to present to the user.	When a user clicks the **Create Contact** command on the More menu of the *Leads* (CR301000) form, the dialog box should be named **Create Contact**, as shown below.
For a dialog box that opens a form in a pop-up window: You do not need to make any changes to the form's name.	Suppose that a user has clicked the link in the **Customer** box for the selected record on the *Sales Orders* (SO301000) form. The system launches a pop-up window and displays the selected record on the *Customers* (AR303000) form. Notice that the original name of this form is maintained in the pop-up window.

Related Links

- *Upload Dialog Box*

Dialog Box: Configuration

A dialog box can contain a Summary area, a grid, and a footer with buttons, as shown in the following screenshot.

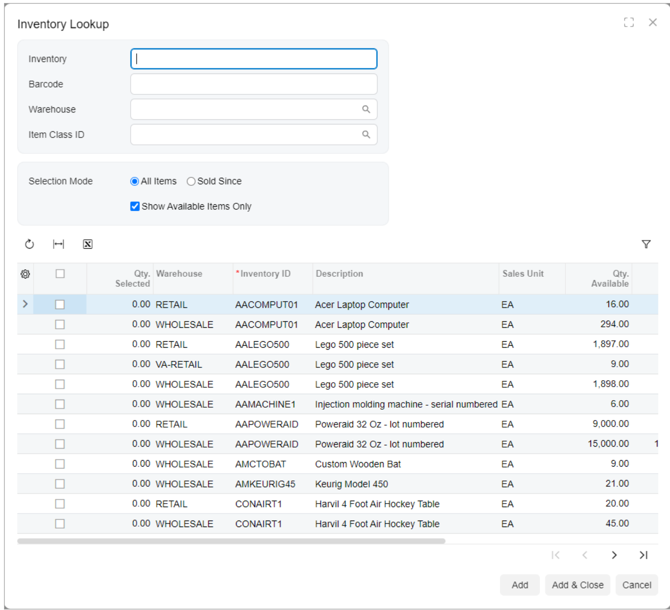


Figure: Sample layout of a dialog box

View Definition of a Dialog Box

The following code shown an example of a view declaration and initialization for a dialog box.

```
@graphInfo({graphType: 'PX.Objects.SO.SOOrderEntry', primaryView: 'Document',
  bpEventsIndicator: true, udfTypeField: 'OrderType', showActivitiesIndicator: true})
export class SO301000 extends PXScreen {
  ...
  AddSelectedItems: PXActionState;

  @viewInfo({containerName: "Inventory Lookup"})
  ItemFilter = createSingle(SOSiteStatusFilter);

  @viewInfo({containerName: "Inventory Lookup"})
  ItemInfo = createCollection(SOSiteStatusSelected);
}

export class SOSiteStatusFilter extends PXView {
    Inventory: PXFieldState<PXFieldOptions.CommitChanges>;
    BarCode: PXFieldState<PXFieldOptions.CommitChanges>;
    SiteID: PXFieldState<PXFieldOptions.CommitChanges>;
    ItemClass: PXFieldState<PXFieldOptions.CommitChanges>;
    SubItem: PXFieldState<PXFieldOptions.CommitChanges>;

    Mode: PXFieldState<PXFieldOptions.CommitChanges>;
    HistoryDate: PXFieldState<PXFieldOptions.CommitChanges>;
    OnlyAvailable: PXFieldState<PXFieldOptions.CommitChanges>;
    DropShipSales: PXFieldState<PXFieldOptions.CommitChanges>;

    CustomerLocationID: PXFieldState<PXFieldOptions.CommitChanges>;
}

@gridConfig({
    preset: GridPreset.ReadOnly,
    syncPosition: true,
```

```
    adjustPageSize: true
})
export class SOSiteStatusSelected extends PXView {
    @columnConfig({allowCheckAll: true}) Selected: PXFieldState;
    QtySelected: PXFieldState;
    @columnConfig({hideViewLink: true}) SiteID: PXFieldState;
    @columnConfig({hideViewLink: true}) ItemClassID: PXFieldState;
    ItemClassDescription: PXFieldState;
    @columnConfig({hideViewLink: true}) PriceClassID: PXFieldState;
    PriceClassDescription: PXFieldState;
    @columnConfig({hideViewLink: true}) PreferredVendorID: PXFieldState;
    PreferredVendorDescription: PXFieldState;
    @columnConfig({hideViewLink: true}) InventoryCD: PXFieldState;
    @columnConfig({hideViewLink: true}) SubItemID: PXFieldState;
    Descr: PXFieldState;
    @columnConfig({hideViewLink: true}) SalesUnit: PXFieldState;
    QtyAvailSale: PXFieldState;
    QtyOnHandSale: PXFieldState;
    CuryID: PXFieldState;
    QtyLastSale: PXFieldState;
    CuryUnitPrice: PXFieldState;
    LastSalesDate: PXFieldState;
    DropShipLastQty: PXFieldState;
    DropShipCuryUnitPrice: PXFieldState;
    DropShipLastDate: PXFieldState;
    AlternateID: PXFieldState;
    AlternateType: PXFieldState;
    AlternateDescr: PXFieldState;
}
```

Layout of a Dialog Box

To define the layout of the Summary area of a dialog box, you use one of the predefined templates. For details, see *Form Layout: Predefined Templates*. The following code shown an example of the HTML for the dialog box in the preceding screenshot.

```
<qp-panel id="ItemInfo"
  caption="Inventory Lookup" auto-repaint="true" width="lg" height="85vh">
    <qp-template id="ItemFilter_formSitesStatus" name="17-17-14"
      class="equal-height" wg-container>
        <qp-fieldset id="fs-sitestatusfilter-first" slot="A"  view.bind="ItemFilter">
            <field name="Inventory"></field>
            <field name="BarCode"></field>
            <field name="SiteID"></field>
            <field name="ItemClass"></field>
            <field name="SubItem"></field>
        </qp-fieldset>
        <qp-fieldset id="fs-sitestatusfilter-second" slot="B"  view.bind="ItemFilter">
            <field name="Mode" control-type="qp-radio"></field>
            <field name="HistoryDate"></field>
            <field name="OnlyAvailable"></field>
            <field name="DropShipSales"></field>
        </qp-fieldset>
        <qp-fieldset id="fs-sitestatusfilter-third" slot="C"  view.bind="ItemFilter">
            <field name="CustomerLocationID"></field>
        </qp-fieldset>
```

```
    </qp-template>
    <qp-grid id="ItemInfo_gridSiteStatus" view.bind="ItemInfo">
    </qp-grid>
    <footer>
        <qp-button id="btnAdd" state.bind="AddSelectedItems">
        </qp-button>
        <qp-button id="btnAddClose" caption="Add & Close" dialog-result="OK">
        </qp-button>
        <qp-button id="btnCancel" caption="Cancel" dialog-result="Cancel">
        </qp-button>
    </footer>
</qp-panel>
```

Width and Height of a Dialog Box

Generally, you do not need to specify the width and height of a dialog box. If you need to change the width and height of the dialog box that are detected by default, you can use the `width` and `height` attributes of the `qp-panel` tag.

To specify the width of the dialog box, you use the preferred values for the property, which are *sm*, *md*, and *lg*. Only if the predefined sizes do not suit your needs can you can use other standard CSS units to specify the value for this property, such as pixels (px), percentage (%), or viewport width (vw). For a complete list of standard CSS values and units, see *CSS values and units*.

You may need to specify the height of a dialog box if the dialog box contains a grid, tab, or rich text editor. You set the height by using pixels (px) or the viewport height (vh).

The following code shows some examples of the usage of different values and units for the `width` and `height` attributes.

```
<qp-panel
  id="ItemInfo" width="md" height="85vh">
...
</qp-panel>

<qp-panel
  id="ItemInfo" width="120px" height="85vh">
...
</qp-panel>

<qp-panel
  id="ItemInfo" width="100%" height="85vh">
...
</qp-panel>
```

If a dialog box contains a text message in the `qp-text-box` control, its parent, or the `field` tag that corresponds to the `qp-text-box` control, you can make the height of the dialog box adaptive to the size of the message. For this purpose, do the following:

1. Specify the `adaptive-height` CSS class for the `qp-text-box` control or its parent. For details, see *adaptive-height*.
2. Specify the multiline mode for the `qp-text-box` control. For details, see *Text Box: Multiline Text Box*.

The following code shows an example of configuring adaptive height in a dialog box.

```
<qp-panel
  id="LogFileFilterRecord" caption="Log" auto-repaint="true" width="50vw">
  <qp-fieldset id="frmLockout" view.bind="LogFileFilterRecord">
```

```
    <field
      name="Text"
      class="label-size-xxs adaptive-height"
      config-type.bind="1"></field>
  </qp-fieldset>
  <footer>
    <qp-button id="btnCancel" dialog-result="Cancel" caption="Close"></qp-button>
  </footer>
</qp-panel>
```

Buttons of a Dialog Box

To add a button to the dialog box, you do the following:

1. In the `qp-panel` tag, you specify the `qp-button` tag.

   For the button to be at the bottom of the dialog box, add the `qp-button` tag inside the `footer` tag.
2. You bind the button to an action by specifying action in the `state` attribute or specify the `dialog-result` attribute.

For more details on adding buttons to a dialog box, see *Dialog Box: Buttons on the Dialog Box*.

Related Links

- *Table Toolbar Button That Opens a Dialog Box*

Dialog Box: Field Validation

If you want the system to perform validation of values (for example, in event handlers or by using the `VerifyRequired` method) when a user clicks a button in a dialog box, you need to enable this validation in the frontend.

The validation can be performed both for the fields of a dialog box and for the fields on the form.

To enable this validation, do the following:

1. In the HTML template, for the button that returns `WebDialogResult.OK`, specify `validateInput = true`, as shown in the following code.

```
<footer id="footer-CreateSalesOrder">
  <qp-button id="buttonOK-CreateSalesOrder" dialog-result="OK"
    config.bind="{validateInput: true}">
  </qp-button>
</footer>
```

2. If you need to validate values when a user clicks the **Close** button in the dialog box, in the `qp-panel` tag, specify `closeButtonValidateInput = true`, as shown in the following code.

```
<qp-panel id="CreateSalesOrder" caption="Create Sales Order"
  auto-repaint="true" width="sm"
  close-button-dialog-result="No" close-button-validate-input="true">
```

Related Links

- *Interface IButtonControlConfig*
- *Interface IGeneralDialogConfig*

Dialog Box: Opening a Dialog Box

You can define an action that opens a dialog box by using the following types of actions:

- A frontend action whose corresponding button is anywhere on the form
- A frontend action whose corresponding button is on the table toolbar
- A backend action mapped in the frontend

A *frontend action* is an action that is defined only in the frontend and does not have a corresponding action in a graph or a graph extension.

The Classic UI provides only the ability to trigger a backend action in the frontend. By using the Modern UI, you can define frontend actions that open a dialog box.

Frontend Action Anywhere on a Form

To define a fronend action with a button located anywhere on a form, do the following:

1. In TypeScript, define a `PXActionState` property for this action.

   The following code shows an example of defining an action in the screen class.

   ```
   export class SO301000 extends PXScreen {
     viewInquiryParams: PXAcionState;
   }
   ```

2. Add the `actionConfig` decorator.
3. In the `popupPanel` property of the `actionConfig` decorator, specify the dialog box that his action opens, as shown in the following code.

   ```
   @actionConfig({popupPanel: 'InquiryParams'})
   viewInquiryParams: PXAcionState;
   ```

   If you also need to specify an action that should be executed when the dialog box is closed, use the `popupCommand` property.

By default, this action is displayed on the More menu. To specify its location, add the `qp-button` tag for this action in HTML template or use the `topBarItems` property of the `gridConfig` decorator. The following code shows an example of the `qp-button` tag being used to define the action parameters.

```
<qp-button id="edInquiryParameters" caption="Inquiry Parameters"
  if.bind="viewInquiryParams.visible==true"></qp-button>
```

Frontend Action on the Table Toolbar

To define an action that opens a dialog box and whose button is located on the table toolbar, you can use one of the following approaches:

- Define an action in the view class for the grid and specify the `popupPanel` property in the `actionConfig` decorator.
- Define an action in the screen class and specify it in the `topBarItems` property of the `gridConfig` decorator. For more details, see *Table Toolbar Button That Opens a Dialog Box*.

Backend Action Mapped in the Frontend

To define a backend action that is mapped in the frontend and opens a dialog box, do the following:

1. In the backend, define an action that opens a dialog box.

   To invoke the dialog box, use the `AskExt()` method of the view, as shown in the following example.

```
public PXAction<SOOrder> addBlanketLine;
[PXUIField(DisplayName = "Add Blanket SO Line",
  MapEnableRights = PXCacheRights.Update,
  MapViewRights = PXCacheRights.Select, Visible = true)]
[PXLookupButton()]
public virtual IEnumerable AddBlanketLine(PXAdapter adapter)
{
  if (Base.Document.Current != null && BlanketSplits.AskExt() == WebDialogResult.OK)
  {
    ...
  }
}
```

2. In the backend, define an action that closes the dialog box with the `WebDialogResult.OK` result, as shown in the following example.

```
public PXAction<SOOrder> addBlanketLineOK;
[PXUIField(DisplayName = "Add",
  MapEnableRights = PXCacheRights.Select,
  MapViewRights = PXCacheRights.Select)]
[PXLookupButton()]
public virtual IEnumerable AddBlanketLineOK(PXAdapter adapter)
{
  BlanketSplits.View.Answer = WebDialogResult.OK;
  return AddBlanketLine(adapter);
}
```

3. In the frontend, in TypeScript, define the `PXActionState` properties for the action that opens the dialog box. An example is shown in the following code.

```
export class SOLine extends PXView {
  AddBlanketLine: PXActionState;
}

export class SO301000_AddBlanketOrderLine {
    AddBlanketLineOK: PXActionState;
}
```

4. If you need to specify the location of the button that opens the dialog box, in the HTML template, add the `qp-button` tag for the action or use the `topBarItems` property of the `gridConfig` decorator.
5. In the HTML template, specify the location of the button that closes the dialog box. For more details on actions on the dialog box, see *Dialog Box: Buttons on the Dialog Box*.

   The following code shows an example of a button being placed in the footer of the dialog box.

```
<footer id="footer-AddBlanketOrderLine">
  <qp-button id="buttonAdd-AddBlanketOrderLine" state.bind="AddBlanketLineOK"></qp-
button>
```

```
</footer
```

Dialog Box: Buttons on the Dialog Box

You can add two types of button to a dialog box:

- A standard button, such as **OK** or **Cancel**
- A button whose action is defined in the backend

Processing a Dialog Box Action

In a dialog box, a button's logic can be determined in one of the following ways:

- By binding the button to a graph action
- By specifying the value of the `dialog-result` attribute
- By combining the ways listed above

This section describes the differences between these three approaches.

An action in a dialog box is processed as follows: Suppose that a user invokes an action, `OpenDialog`, that opens a dialog box. In the backend, in the `OpenDialog` action body, the dialog box is opened. In the UI, a user sees the dialog box and clicks one of the buttons in it. Then the system acts as follows, based on the action bound to the clicked button and the specified `dialog-result` value:

- By default, the frontend invokes the same action that opened the dialog box (in this case, `OpenDialog`).
- If no action is bound and a `dialog-result` is specified, the frontend invokes the action that opened the dialog (in this case, `OpenDialog`) with the specified `dialog-result` value. You can process the returned `dialog-result` value in the action body in the graph.
- If some action is bound to the clicked button and no `dialog-result` value is specified, the system invokes this action.
- If both the `dialog-result` value is specified and an action is bound to the button, both the `OpenDialog` action and the bound action will be executed on the server when this button is clicked. The bound action will be executed first. If no exceptions are thrown, the `OpenDialog` action with the specified `dialog-result` value will also be executed in the same server round trip. The following code shows examples of this type of buttons.

The functionality described in this list item is not widely used in Acumatica ERP. We recommend that you not use this approach to execute both actions simultaneously. But if your legacy code uses this approach, you can migrate the code to the Modern UI without server-side modifications.

```
<qp-panel ...>
  ...
  <footer>
    <qp-button id="CreatePaymentRefundButton"
       state.bind="CreatePaymentRefund" dialog-result="Abort">
    </qp-button>
    <qp-button id="CreatePaymentCaptureButton"
       state.bind="CreatePaymentCapture" dialog-result="Yes">
    </qp-button>
    <qp-button id="CreatePaymentAuthorizeButton"
       state.bind="CreatePaymentAuthorize" dialog-result="No">
    </qp-button>
```

```
    <qp-button id="CreatePaymentOKButton"
       state.bind="CreatePaymentOK" dialog-result="OK">
    </qp-button>
    <qp-button id="CreatePaymentCancelButton"
       state.bind="CreatePaymentCancel" dialog-result="Cancel">
    </qp-button>
  </footer>
</qp-panel>
```

Thus, we recommend only the following approaches for actions in the dialog box:

- Specify the `dialog-result` value, and in the graph, process it in the body of the action that opens the dialog box.
- Bind the action to the button of the dialog box by specifying the action in the `state.bind` attribute

If the `dialog-result` attribute is specified for a button, clicking this button will always close the dialog box. If you have not specified the `dialog-result` attribute for a button, clicking this button will never close the dialog box.

Adding a Button with dialog-result

To add a button with the `dialog-result` value, in the HTML template, add the `qp-button` tag in the `footer` tag of the `qp-panel` tag and specify the following attributes:

- `id`
- `dialog-result`
- `caption`

  If you do not specify the `caption` attribute, the button's caption will be set to the value of the `dialog-result` attribute, and the caption will be localizable by general rules. You can override the default caption by specifying the `caption` attribute. You can omit the `caption` attribute if its value is the same as the value of the `dialog-result` attribute.

Suppose that an action that opens a dialog box is defined in a graph as shown in the following code. You process the `dialog-result` value in the body of the action that invoked the dialog box.

```
public PXAction<ARInvoice> selectARTran;
[PXUIField(DisplayName = "Add Return Line",
  MapEnableRights = PXCacheRights.Insert, MapViewRights = PXCacheRights.Select)]
[PXLookupButton]
public virtual IEnumerable SelectARTran(PXAdapter adapter)
{
  if (Base.Document.Cache.AllowInsert)
  {
    WebDialogResult result = arTranList.AskExt();
    if (result == WebDialogResult.OK)
      AddARTran(adapter);
  }
  return adapter.Get();
}
```

The following code shows an example of the **Add & Close** and **Cancel** buttons being added to this dialog box.

```
<qp-panel>
  <footer id="footer-AddReturnLine">
    <qp-button id="buttonAddClose-AddReturnLine" dialog-result="OK"
               caption="Add & Close"></qp-button>
```

```
    <qp-button id="buttonCancel-AddReturnLine" dialog-result="Cancel"></qp-button>
  </footer>
</qp-panel>
```

Adding a Button with a Bound Action

Suppose that an action that opens a dialog box is defined in a graph as shown in the following code.

```
public PXAction<Document> addARTran;
[PXButton(DisplayOnMainToolbar = false), PXUIField(DisplayName = "Add Return Line")]
protected virtual IEnumerable AddARTran(PXAdapter adapter)
{
    // do some work
}
```

To add a button whose action is defined in backend, do the following:

1. In TypeScript, in the screen class, add a member of the `PXActionState` type with the same name as the `PXAction` property in the graph.

   The following code shows an example of adding the `AddARTran` action.

   ```
   export class SO303000_AddReturnLine {
     AddARTran: PXActionState;
     ...
   }
   ```

2. In the HTML template, add the `qp-button` tag in the `qp-panel` tag. You can place the tag either among the controls of the dialog box or in the footer. In the `qp-button` tag, specify the following attributes:

   - `id`
   - `state`

     In this attribute, specify the name of the action added in the screen class.

   The following code shows an example of the `AddARTran` action being added in HTML.

   ```
   <qp-panel>
     <footer>
       <qp-button id="buttonAdd-AddReturnLine" state.bind="AddARTran"></qp-button>
     </footer>
   </qp-panel>
   ```

If you define an action in the screen class in TypeScript, the action is hidden from the More menu and the form toolbar. That is, this approach cancels the default display behavior of the button and hands over the ability to manage this behavior to the developer. As a result, you can specify the location of the button anywhere on the form, such as in the Summary area of a dialog box or in the footer. If you do not add the `qp-button` tag in HTML, the button for the action will not be displayed anywhere on the form.

Specifying the Response Returned by the Close Button

For a dialog box, you can specify the response returned by the Close button (which is located at the top right corner of the dialog box). To do this, in the `qp-panel` tag, specify the `close-button-dialog-result` attribute, as shown in the following code.

```
<qp-panel id="rollsettings" caption="BOM Cost Settings"
```

```
  auto-repaint="true"
  close-button-dialog-result="Abort">
```

You can process the response the same way as you processed the `dialog-result` value in the body of the action that opened the dialog box.

Related Links

- *Button: Layout Examples*
- *Button: Configuration*

Dialog Box: Executing Action on Dialog Box Closing

You can specify the action to be executed when a dialog box is closed. To do this, specify the name of the action method in the `command` property of the `qp-panel` tag.

Suppose that you have defined the following action in a graph.

```
public PXAction<GLBudgetTree> Preload;
[PXButton]
[PXUIField(MapEnableRights = PXCacheRights.Update, Visible = false)]
public virtual IEnumerable preload(PXAdapter adapter)
{ ... }
```

In the frontend, you only need to provide the name of the action in the `qp-panel` tag, as shown in the following code. You do not need to declare a property of the `PXActionState` type in TypeScript unless you want to hide it from the More menu.

```
<qp-panel id="Details" caption="Preload Accounts" auto-repaint="true"
  command="preload">
```

Related Links

- *Interface IGeneralDialogConfig*

Dialog Box: Files

The **Files** dialog box gives a user the ability to attach files to the primary record displayed on a form. To open this dialog box, the user clicks the **Files** button on the right side of the form title bar. Below you can see an example of the **Files** button and the corresponding dialog box.

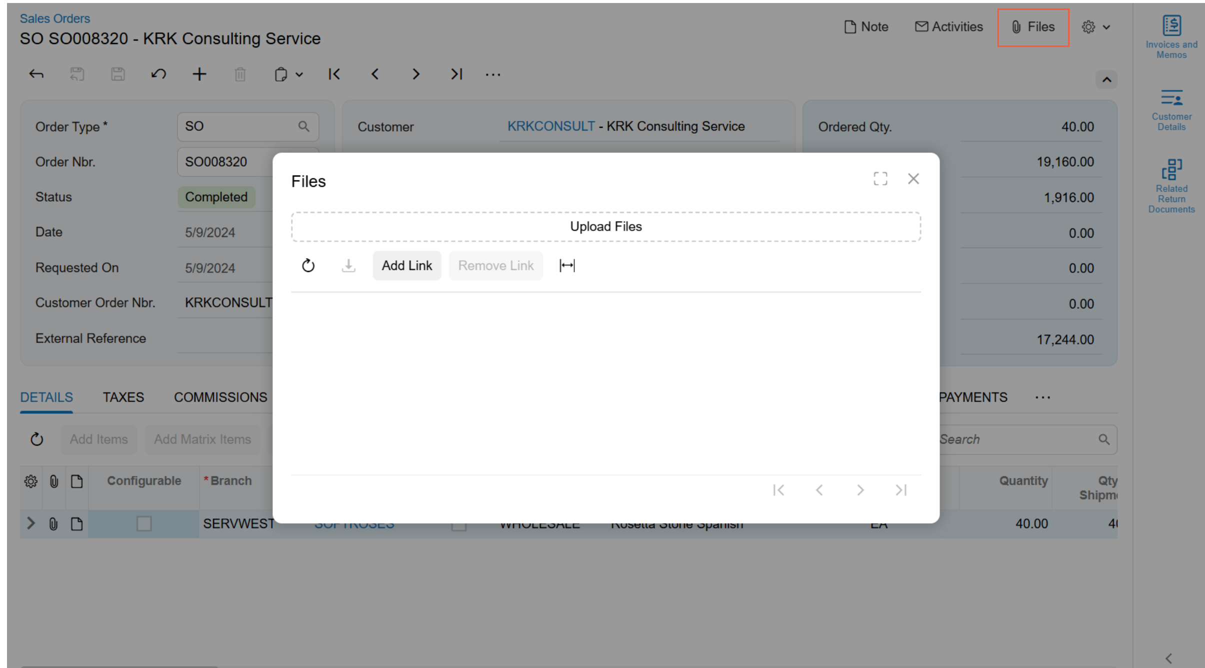


Figure: The Files dialog box

The Files dialog box is implemented using the `qp-upload-dialog` control.

Hiding the Files Button

The **Files** button is displayed by default. To hide this button from the form title bar, specify `hideFilesIndicator : true` in the `graphInfo` decorator.

Default Closing Behavior

By default, a user can close the **Files** dialog box only by clicking the X button in the dialog box. A user cannot close the dialog box by clicking outside the dialog box.

Opening the Files Dialog Box from a Custom Button

You can implement a button anywhere on a form, and specify that this button should open the **Files** dialog box. For this purpose, do the following:

1. In TypeScript code, define the following method, which opens the **Files** dialog box.

```
filesShow() {
  (this.activeScreenVM?.screenService
    .getDataComponent(NoteMenuDataComponent.dataComponentName) as
 NoteMenuDataComponent).filesShow();
}
```

2. In HTML, define a button by using the `qp-button` tag.

```
<qp-button id="btnFiles" caption="Attach Files"></qp-button>
```

3. In the `qp-button` tag, specify the method that opens the **Files** dialog box in the `click.delegate` attribute, as the following code shows.

```
<qp-button id="btnFiles" caption="Open Files" click.delegate="filesShow()">
```

```
</qp-button>
```

Related Links

- *Attachments: File Upload and Attachment*

Dialog Box: Notes

The Notes dialog box provides a user with the ability to specify notes for a particular record displayed on a form. To open the Notes dialog box, the user clicks the **Note** button on the right side of the form title bar. Below you can see an example of the Note button and the corresponding dialog box.

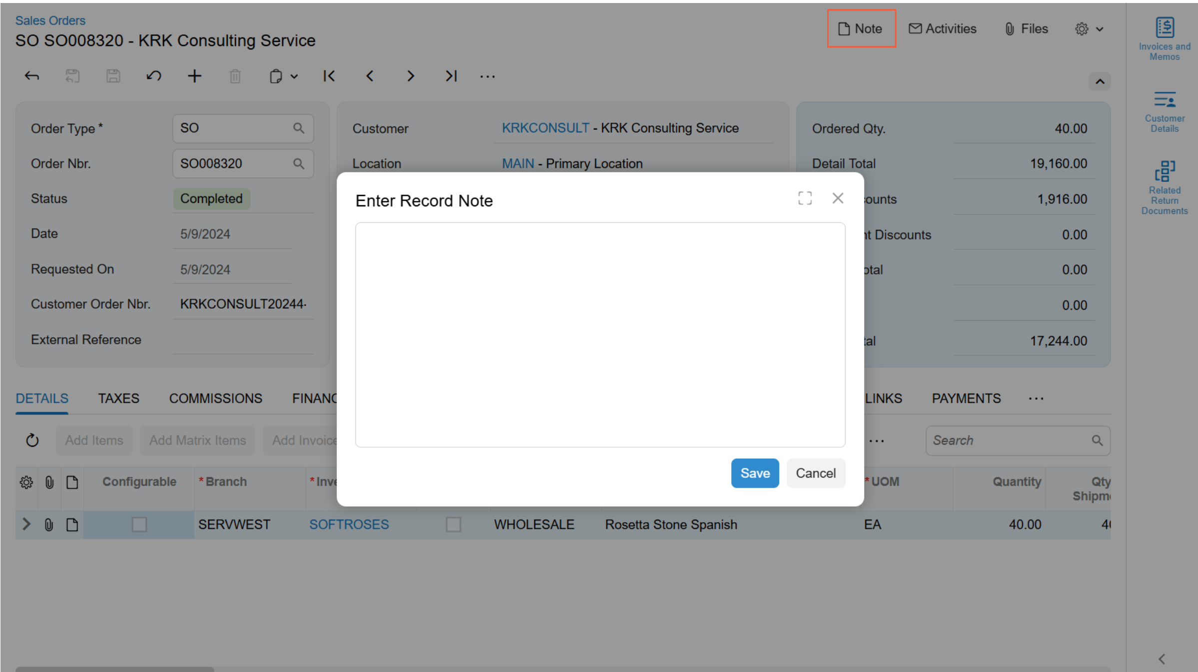


Figure: The Notes dialog box

The Notes dialog box is implemented using the `qp-dialog` control.

Hiding the Note Button

The **Note** button is displayed by default. To hide this button from the form title bar, specify `hideNotesIndicator : true` in the `graphInfo` decorator.

Default Closing Behavior

By default, a user can close the Note dialog box in one of the following ways:

- If no changes have been made in the notes attached to the record, the dialog box is closed when a user clicks anywhere outside the dialog box.
- If any modification has been made to the notes attached to the record, a user can close the dialog box only by clicking the **Save** button in the dialog box.

Opening Notes from a Custom Button

You can implement a button anywhere on a form, and specify that this button should open the Notes dialog box. For this purpose, do the following:

1. In TypeScript code, define the following method, which opens the Notes dialog box.

```
noteShow() {
  (this.screenService.getDataComponent(NoteMenuDataComponent.dataComponentName)
    as NoteMenuDataComponent).noteShow();
}
```

2. In HTML, define a button by using the `qp-button` tag.

```
<qp-button id="notes" caption="Open Notes"></qp-button>
```

3. In the `qp-button` tag, specify the method that opens the Notes dialog box in the `click.delegate` attribute, as the following code shows.

```
<qp-button id="notes" caption="Open Notes" click.delegate="noteShow()">
</qp-button>
```

Related Links

- *Attachments: Note Attachments*

Dialog Box: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to dialog boxes to HTML or TypeScript elements.

PXSmartPanel

The following table shows the correspondence between `PXSmartPanel` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
PXSmartPanel	Is replaced by the qp-panel tag.

AutoRepaint
 	
<px:PXSmartPanel
  AutoRepaint="True" ... />
	Is replaced by the auto-repaint attribute, which indicates (if set to *true*) that the system should initiate a callback to the server with all the updated data. The server then processes the data and shows the processed data in the dialog box.
	
	<qp-panel
	  id="ItemInfo" auto-repaint="true">
	...
	</qp-panel>

ASPX	HTML or TypeScript

`Caption`
 	
`<px:PXSmartPanel`
`  Caption="Inventory Lookup"`
`... />`
	Is replaced by the `caption` attribute, which sets the caption of the dialog box. The caption is localizable.
	
	`<qp-panel`
	`  id="ItemInfo" caption="Inventory`
	` Lookup">`
	`...`
	`</qp-panel>`

`CloseButtonDialogResult`	Is replaced by the `close-button-dialog-result` attribute, which specifies the dialog result for the Close button of the dialog box.
	
	`<qp-panel id="ItemInfo"`
	`  close-button-dialog-result="Abort">`
	`...`
	`</qp-panel>`

`CommandName`
 	
`<px:PXSmartPanel`
`  CommandName="preload" ... />`
	Is replaced by the `command` attribute, which specifies the action that should be executed on dialog closing. For more details, see *Dialog Box: Executing Action on Dialog Box Closing*.
	
	`<qp-panel id="Details" command="preload">`

`Height`	Is replaced by the `height` attribute, which sets the height of the dialog box.
	
	`<qp-panel`
	`  id="ItemInfo" width="85vw"`
	` height="85vh">`
	`...`
	`</qp-panel>`

`ID`
 	
`<px:PXSmartPanel`
`  ID="ItemInfo" ... />`
	Is replaced by the `id` attribute, which sets the ID of the dialog box. Use the `key` value from ASPX for the `id` in HTML if the `id` and `key` in ASPX do not match.
	
	`<qp-panel`
	`  id="ItemInfo">`
	`...`
	`</qp-panel>`

`Key`	Is replaced by the `id` attribute which sets the id of the dialog box. Use the `key` value from ASPX for the `id` in HTML if the `id` and `key` in ASPX do not match. This is a requirement to maintain the testability of the UI.

ASPX	HTML or TypeScript
`Width`	Is replaced by the `width` attribute, which sets the width of the dialog box.

```
<qp-panel
  id="ItemInfo" width="1632px"
  height="918px">
...
</qp-panel>
```

PXPanel

The following table shows the correspondence between the `PXPanel` ASPX element and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXPanel`	Is replaced by the `footer` tag. The `footer` tag contains the buttons at the bottom of the dialog box. This tag can contain any number of `qp-button` tags.

```
<px:PXPanel runat="server"
  ID="PXPanelAddToProjects"
  SkinID="Buttons">
    <px:PXButton
      runat="server"
      ID="btnCancel"
      Text="Cancel"
      DialogResult="Cancel" >
</px:PXPanel>
```

```
<footer>
  <qp-button
    id="buttonCancel-addItemLookup"
    dialog-result="Cancel">
  </qp-button>
</footer>
```

Obsolete ASPX Controls and Properties

The following table lists obsolete ASPX elements that are related to dialog boxes. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXSmartPanel`	• `AcceptButtonId` • `AllowMove` • `AllowResize` • `AlreadyLocalized` • `AutoCallBack-Command` • `AutoCallBack-Target` • `AutoReload` • `AutoSize-Container` • `AutoSize-Enabled` • `CallbackArgument` • `CallBackMode-CommitChanges` • `CallBackMode-PostData`

ASPX Control	Properties
	• CancelButtonID • CommandSourceID • DependsOnView • HideAfterAction • LoadOnDemand • Overflow • Position • RenderVisible • runat • SkinID
PXPanel	• ID

Error, Warning, or Informational Notification

In this chapter, you will learn about the configuration of error, warning, and informational notifications. You will develop an understanding of when to use these notifications and what is the best way to phrase them.

Error, Warning, or Informational Notification: General Information

A notification on an Acumatica ERP form is a control containing an error message, a warning message, or an informational message that is displayed to a user to convey important information related to the data displayed on the form. The following screenshots show examples of an informational notification, a warning notification, and an error notification.

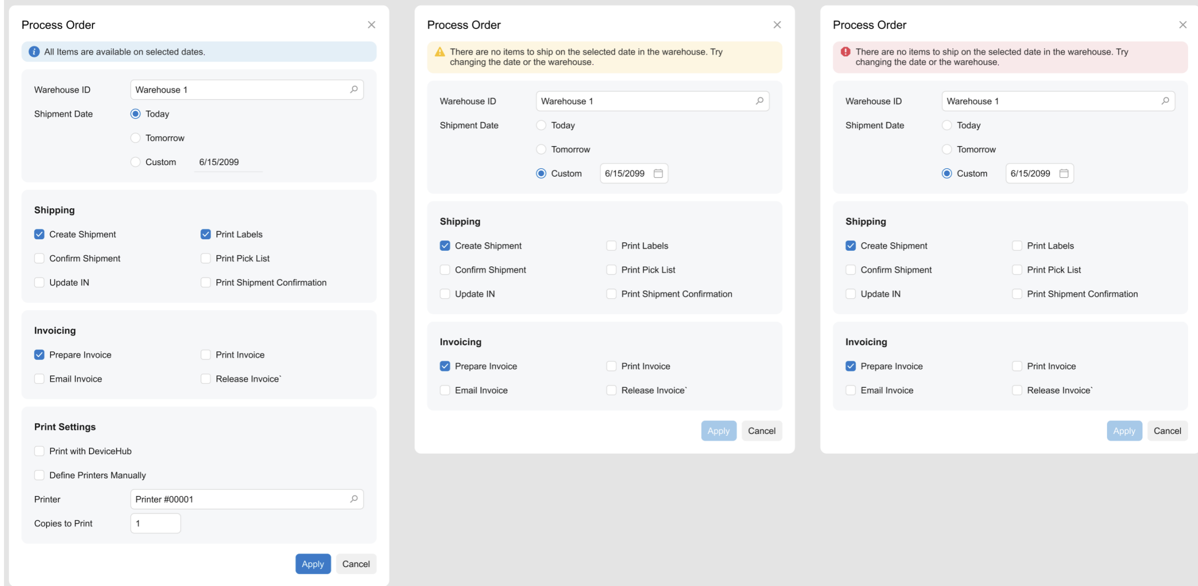


Figure: Notifications

In the Modern UI, a notification is defined by the `qp-info-box` control. The Classic UI does not have an analog of this control.

Learning Objectives

In this chapter, you will learn the following about error, warning, and informational notifications:

- The guidelines for error, warning, and informational messages
- The proper configuration of a notification for specific cases, such as a notification that appears in specific place of an Acumatica ERP form

Applicable Scenarios

You configure an error, warning, or informational notification in the following cases:

- An error or a warning has been generated during validation of the data that the user has entered on an Acumatica ERP form.
- Some information related to the data displayed on an Acumatica ERP form should be shown to a user.

Message Guidelines

The main idea of any error message is to describe the problem that is stopping the user or system from completing a task and to educate the user on how to solve this problem. Before finalizing an error message and inserting it in the code, you should consider whether your message answers the following questions:

- What problem has occurred? (State the problem clearly.)
- Why did it happen? (Explain what has gone wrong.)
- How can the user solve the problem? (Provide the user with a solution to the problem.)

Your error message should answer at least two of these questions in a given situation. It must include enough information for the user to understand the problem and to overcome it.

In a warning message, you inform a user about a problem that could arise so that they can avoid it. Clearly state what could go wrong and how the user can avoid it. Do not include any unneeded details.

You use an informational message when you want to provide helpful information to help the user understand what is happening (for example, informing them about the successful completion of a process). Keep the message concise and useful; the goal is to support the user in completing the task at hand, not to slow their work with unneeded or confusing information.

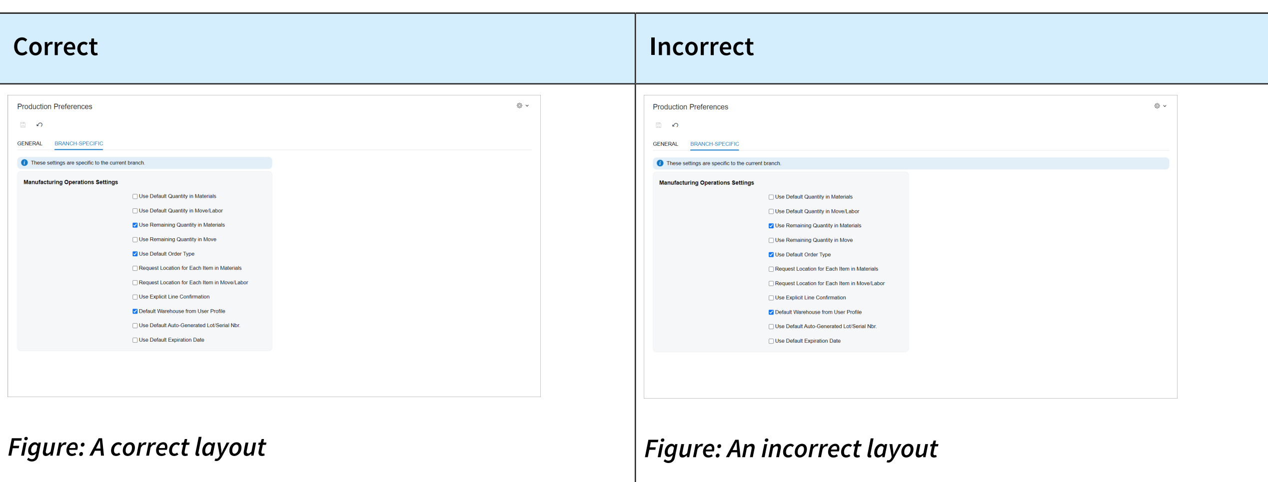
In all types of messages, you should use complete sentences and the proper punctuation. Clear, concise messages enhance and smooth the user experience.

Recommendations for Organizing the Layout

The following table shows the recommendations for organizing the layout with error, warning, and informational notifications.

Correct	Incorrect
When a template contains fields in only one fieldset and an error, warning, or informational notification should be displayed for this fieldset, align the notification with the fieldset. Put both the `qp-info-box` tag and the `qp-fieldset` tags into a single `div` tag with the specified slot value, as shown below.	

```
<qp-template id="scanSetupTemplate" name="1-1" class="label-size-xm">
  <div slot="A">
    <qp-info-box caption="These settings are specific to the current branch."
      type="info"></qp-info-box>
    <qp-fieldset slot="A" id="ScanSetup_formScanSetup"
      view.bind="ScanSetup" caption="Manufacturing Operations Settings">
    </qp-fieldset>
  </div>
</qp-template>
```



Correct	Incorrect
Figure: A correct layout	*Figure: An incorrect layout*

Error, Warning, or Informational Notification: Configuration

In this topic, you can find information about how to configure an error, warning, or informational notification to be displayed on an Acumatica ERP form.

Configuration from the Backend

To display an error, warning, or informational notification on an Acumatica ERP form, you generally do not need to explicitly add the `qp-info-box` control in the frontend code of an Acumatica ERP form. Instead, you should validate the data on the backend and throw an exception of the `PXSetPropertyException` type, as described in *Data Validation: Validation of Field Values* and *Data Validation: Validation of a Data Record*. In this case, the notification is displayed automatically for the field or data record for which the exception has been thrown.

Notification Defined in the Frontend

You may need to display a static notification on an Acumatica ERP form or display a notification in a specific place on the form. In this case, you can explicitly define the `qp-info-box` control in the HTML code of an Acumatica ERP form.

In the following example, the notification displays the message that is defined by the state of the field specified in the control. You need to define this field in the data access class that provides data for the Acumatica ERP form and in the TypeScript code of the form.

The notification is not displayed if the field specified in the `state` attribute does not have any error, warning, or informational message attached (that is, its state is *none*).

```
<qp-info-box state.bind="QuickProcessParameters.AvailabilityMessage">
</qp-info-box>
```

The notification in the following code displays the error, warning, or informational message that is assigned to any of the fields or records of the view specified in the control. If no error, warning, or informational messages are assigned to the fields or records of the view, the notification is not displayed.

```
<qp-info-box view.bind="QuickProcessParameters"></qp-info-box>
```

You can also specify the type and text of the notification message directly in the control, as the following example shows.

```
<qp-info-box caption="An informational message" type="info">
</qp-info-box>
```

You define the position of the notification on an Acumatica ERP form by placing the `qp-info-box` control in the needed location in HTML code of the form. By default, the notification is stretched through the whole container where it is added. For example, if the `qp-info-box` tag is added inside the `qp-template` tag, the notification has the same width as the whole template.

Fieldset

In this chapter, you will learn about defining fieldsets and configuring fields inside them.

Fieldset: General Information

A fieldset is a control that is a container of fields—that is, it displays one field or multiple fields.

A fieldset is defined by the `qp-fieldset` tag and contains any number of `field` tags or, in rare cases, `qp-field` tags. The control does not have an analog in the Classic UI.

Learning Objectives

In this chapter, you will learn the following about fieldsets:

- The design guidelines for a fieldset, including the naming conventions and layout recommendations
- Examples of fieldsets for particular layouts

Applicable Scenarios

You configure a fieldset when you want to display any number of fields.

Overview of a Fieldset and Its Contents

A fieldset can represent any group of fields in the UI, such as a column of fields in the Summary area, a framed section with a title, or fields inside a dialog box.

A fieldset is also used to organize layout—that is, it can be used as a slot of a `qp-template` tag. For details, see *Form Layout: Predefined Templates*.

A fieldset can include one field or multiple fields. You use the `field` tag to define a field inside `qp-fieldset`.

Fields defined in the HTML code of the form by using `field` tags are displayed by default. You can remove fields from a fieldset or add fields to one by using the **Section Configuration** dialog box. You open this dialog box by clicking **UI Configuration** on the Settings menu and then clicking the Settings icon in the upper right corner of the fieldset.

When you use the `field` tag, the system will display the control that corresponds to the DAC field specified in the name attribute. If you need to add another control, you need to specify it manually inside the `field` tag.

The following example shows a definition of a fieldset with three fields inside it.

```
<qp-fieldset slot="A" id="fsFinancial" view.bind="CurrentDocument" caption="Financial
 Information">
  <field name="BranchID"></field>
  <field name="BranchBaseCuryID"></field>
  <field name="DisableAutomaticTaxCalculation"></field>
</qp-fieldset>
```

Fieldset ID

An ID of a fieldset in HTML consists of two parts, the fs prefix and the semantic name. The ID depends on purpose of the fieldset:

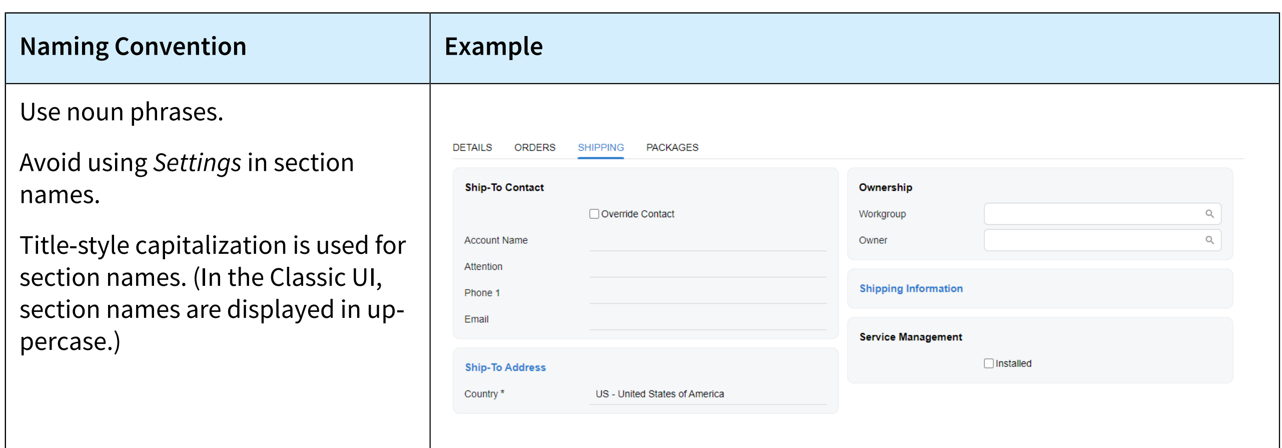
- For a fieldset that represents a column of fields (such as in the Summary area), the ID has the following structure: fsColumnN-<SemanticName> where N is the name of the slot in the template. The semantic name describes the purpose of the element and should be the same for all columns of the template. An example of the fieldsets for three columns is shown in the following code.

```
<qp-template name="7-10-7" id="document_form" wg-container
   qp-collapsible class="equal-height">
   <qp-fieldset slot="A" id="fsColumnA-Order" view.bind="Document">
   ...
   </qp-fieldset>
   <qp-fieldset slot="B" id="fsColumnB-Order" view.bind="Document">
   ...
   </qp-fieldset>
   <qp-fieldset slot="C" label-size="col-lg-6" id="fsColumnC-Order"
      view.bind="Document" class="highlights-section">
   ...
   </qp-fieldset>
   </qp-template>
```

- For all other fieldsets, the ID has the following structure: fs<SematicName>, such as fsShipToAddress. If a fieldset has a title specified in the caption attribute, the semantic name should repeat the title without spaces.

UI Naming Conventions

A fieldset can have a title that is specified using the caption attribute. If a fieldset represents a column, it can have an optional title that a user can change in the **Section Configuration** dialog box. The following table shows the UI naming conventions for the title of a fieldset.

Naming Convention	Example
Use noun phrases. Avoid using *Settings* in section names. Title-style capitalization is used for section names. (In the Classic UI, section names are displayed in up-percase.)	

Recommendations for Organizing the Layout Inside a Fieldset

The following table shows recommendations for organizing the layout of the fieldset.

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">Inside the fieldset, you can put check boxes, fields, and buttons right after a field. For details, see Form Layout: An Element Next to Another Element.

Do not put two or more sets of a label and a control in the same row.</td></tr>
<tr><td>Figure: A correct layout

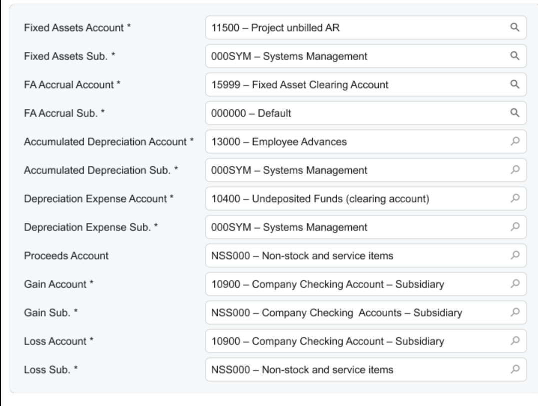
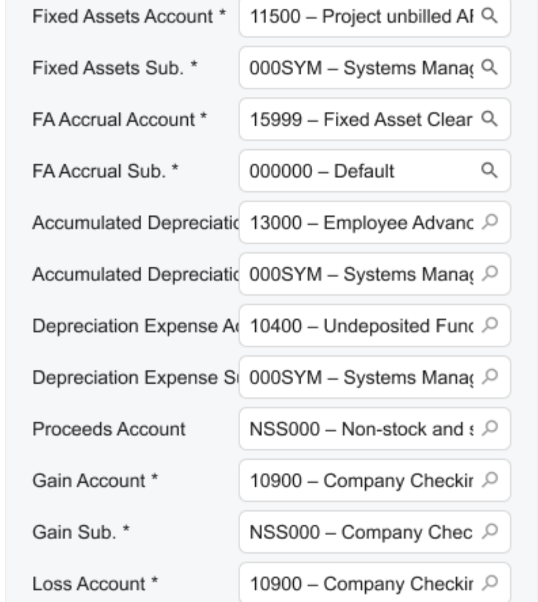
Figure: Another correct layout</td><td>Figure: An incorrect layout</td></tr>
<tr><td colspan="2">Allocate more space for long labels and fields by using the following approaches:
• Use a proper template that has a wider section for your controls
• Specify the length of labels and fields by using the following CSS classes (for details on these classes, see Form Layout: CSS Classes):
  • <code>class="label-size-<SIZE>"</code> for the length of labels
  • <code>class="col-lg-XX"</code> or <code>class="col-md-XX"</code> for the length of fields
Do not use narrow templates for wide labels and fields.</td></tr>
<tr><td>

Figure: A correct layout</td><td>Figure: An incorrect layout

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

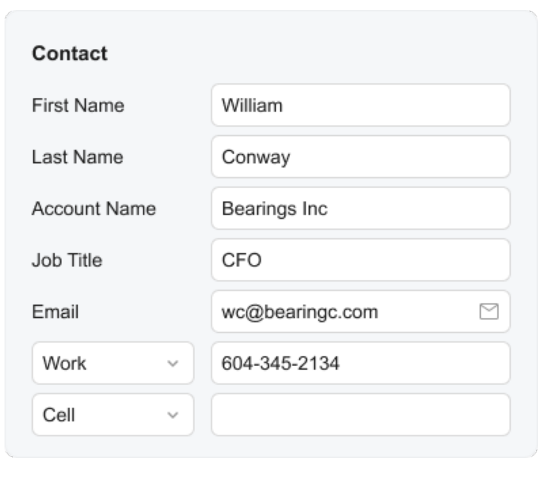
<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">When you need to use a combo box, a check box, or a radio button as a label for a field, align them as labels.
For the field to be used as a label, specify <code>slot="label"</code>.
Do not use combo boxes, check boxes, or radio buttons along with other fields when they are used as labels.</td></tr>
<tr><td>

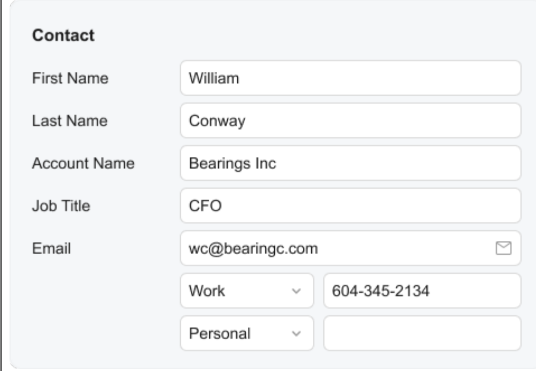
Figure: A correct layout</td><td>

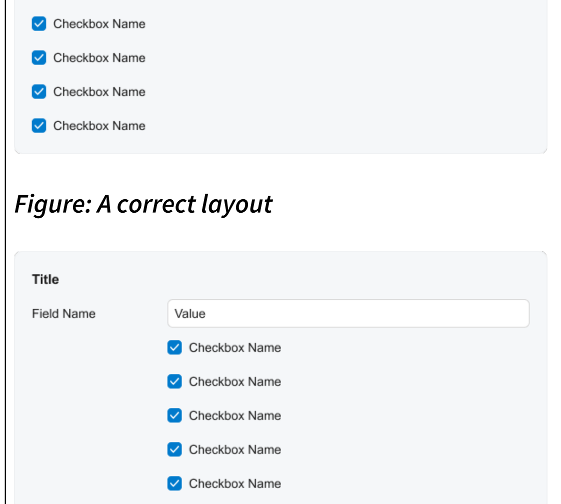
Figure: An incorrect layout</td></tr>
<tr><td colspan="2">When a fieldset contains only check boxes and does not have a title, align check boxes without a left padding.
To remove left padding, specify <code>class="no-label"</code> in <code>qp-fieldset</code>. See Check Box.
Do not leave the default left padding for check boxes or radio buttons when there are no other controls in the fieldset and there is no title.</td></tr>
<tr><td>

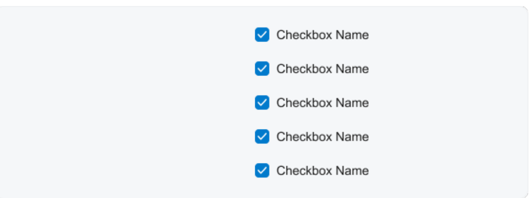
Figure: A correct layout
Figure: Another correct layout</td><td>

Figure: An incorrect layout</td></tr>
<tr><td colspan="2">In a single fieldset, make sure that the length of all labels is the same and the length of all fields is the same.
In a single fieldset, do not specify different sizes of labels and different sizes of fields.</td></tr>
</table>

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td>Contact
First Name William
Last Name Conway
Account Name Bearings Inc
Job Title CFO

Figure: A correct layout</td><td>Contact
First Name William
Last Name Conway
Account Name Bearings Inc
Job Title CFO

Figure: An incorrect layout</td></tr>
<tr><td colspan="2">Configure two or three lines in a text box for the description, summary, subject, or other similar field.

For details about how to create a multiline text box, see Text Box: Multiline Text Box.

Do not use a single-line text box for the description, summary, subject, or similar field for data entry forms. Do not span the text box over multiple columns as was done in ASPX.</td></tr>
<tr><td>Order
Customer
Location
Contact
Description

Figure: A correct layout

You can also make the whole fieldset longer.

Order
Customer
Location
Contact
Description

Figure: A correct layout with longer fields</td><td>Order
Customer
Location
Contact
Description

Figure: An incorrect layout</td></tr>
<tr><td colspan="2">On processing forms, use a single <code>field</code> with the Date Range label and two date and time controls for the selection of the start date and the end date.

Do not use two separate fields in a fieldset for Start Date and End Date boxes on processing forms.</td></tr>
<tr><td>Action
Select By Ship Date
Date Range

Figure: A correct layout</td><td>Action
Select By Ship Date
Start Date
End Date

Figure: An incorrect layout</td></tr>
</table>

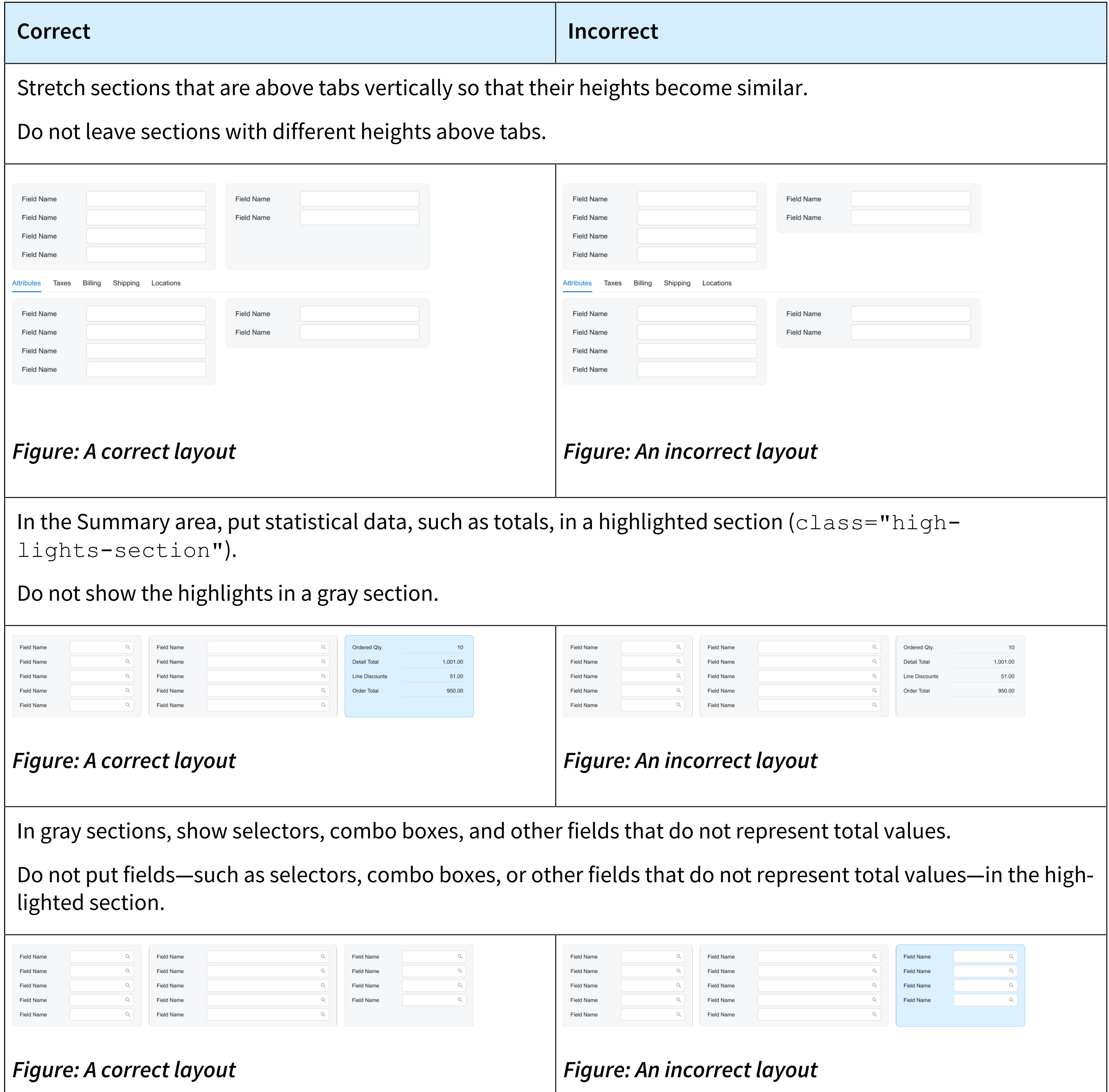
Recommendations for Organizing the Layout of Multiple Fieldsets

The following table shows recommendations for organizing a layout that includes multiple fieldsets.

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">Try to occupy slot A and slot B equally in order to balance the form.
Do not put far more fieldsets into slot A as compared to slot B and the other way round.</td></tr>
<tr><td>Figure: A correct layout</td><td>Figure: An incorrect layout</td></tr>
<tr><td colspan="2">Use a caption instead of showing a single tab.
Do not confuse the table caption and template caption.
Do not show a single tab when you can replace it with a grid caption.</td></tr>
<tr><td>Figure: A correct layout

Figure: Another correct layout</td><td>Figure: An incorrect layout

Figure: Another incorrect layout</td></tr>
</table>



Correct	Incorrect
Stretch sections that are above tabs vertically so that their heights become similar. Do not leave sections with different heights above tabs.	
Figure: A correct layout	*Figure: An incorrect layout*
In the Summary area, put statistical data, such as totals, in a highlighted section (`class="high-lights-section"`). Do not show the highlights in a gray section.	
Figure: A correct layout	*Figure: An incorrect layout*
In gray sections, show selectors, combo boxes, and other fields that do not represent total values. Do not put fields—such as selectors, combo boxes, or other fields that do not represent total values—in the highlighted section.	
Figure: A correct layout	*Figure: An incorrect layout*

Related Links

- *WG Containers for the qp-info-box Control*

Fieldset: Field Configuration

You can define properties of a field in a fieldset by using the *`fieldConfig`* decorator or by using the *`controlConfig`* decorator, which is a shortcut for the `fieldConfig` decorator.

The `controlConfig` decorator accepts configuration of the associated control as a parameter. If you do not need to specify the type of the control (in the `controlType` property), you can use the `controlConfig` decorator instead of the `fieldConfig` decorator.

Example

Suppose that you have the following code in HTML.

```
<field name="FormatLocale"
    control-type="qp-selector" config.bind="{ displayMode: 'text',
        suggester: { descriptionName:'CultureReadableName' } }">
</field>
```

We recommend that you rewrite the code above by using the `fieldConfig` decorator as follows.

```
@fieldConfig({
    controlType: "qp-selector",
    controlConfig: {
      displayMode: 'text',
      suggester: { descriptionName: 'CultureReadableName' },
    }
})
FormatLocale: PXFieldState;
```

If you do not need to change the default type of the control, you can make the TypeScript code shorter by using the `controlConfig` decorator, as shown in the following example.

```
@controlConfig({
    displayMode: 'text',
    suggester: { descriptionName: 'CultureReadableName' },
})
FormatLocale: PXFieldState;
```

If you have defined the control properties in TypeScript by using one of the decorators as shown above, the code of the field in HTML looks as shown in the following code.

```
<field name="FormatLocale"></field>
```

Fieldset: Layout Examples

In this topic, you will find cases of complicated layouts inside a fieldset and instructions on how to implement them.

Check Boxes Next to Particular Controls

If you need to put check boxes next to particular controls and define them to occupy half of the control space, as shown in the following screenshot, add the `qp-field` tag inside the `field` tag, and specify `class="col-6"` for the nested control.

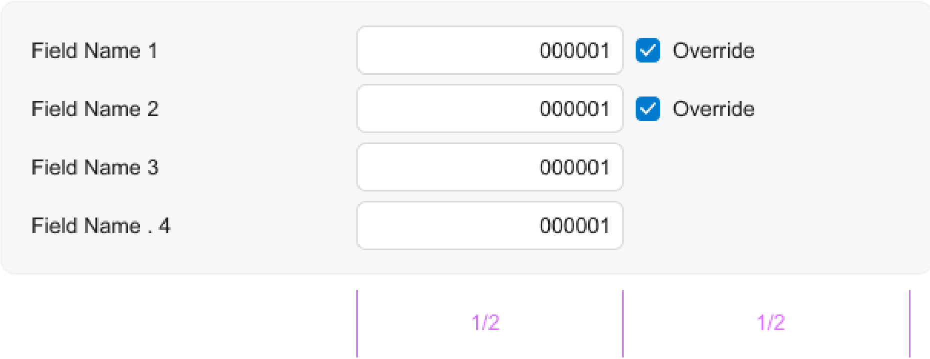


Figure: Check boxes next to particular controls

The following code implements this approach.

```
<qp-fieldset id="groupFields" view.bind="View1">
  <field name="Field1">
    <qp-field control-state.bind="View1.Override1" class="col-6" >
    </qp-field>
  </field>
  <field name="Field2">
    <qp-field control-state.bind="View1.Override2" class="col-6" >
    </qp-field>
  </field>
  <field name="Field3">
    <span class="col-6"></span>
  </field>
  <field name="Field4">
    <span class="col-6"></span>
  </field>
</qp-fieldset>
```

Check Boxes Next to All Controls

If you need to put check boxes next to all controls (as the following screenshot shows), add the `qp-field` tag inside the `field` tag. There is no need to specify the class.

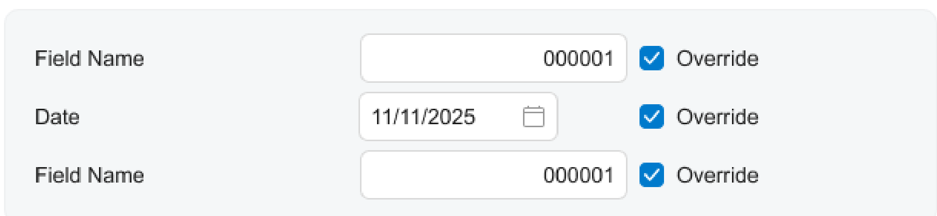


Figure: Check boxes next to all controls

The following code illustrates this approach.

```
<qp-fieldset id="groupFields" view.bind="View1">
  <field name="Field1">
    <qp-field control-state.bind="View1.Override1"></qp-field>
  </field>
  <field name="Date">
    <qp-field control-state.bind="View1.Override2"></qp-field>
  </field>
```

```
  <field name="Field3">
    <qp-field control-state.bind="View1.Override3"></qp-field>
  </field>
</qp-fieldset>
```

Complicated Layout with Multiple Controls in a Row

Suppose that you need to organize a complicated layout with multiple controls in a row, as shown in the following screenshot.

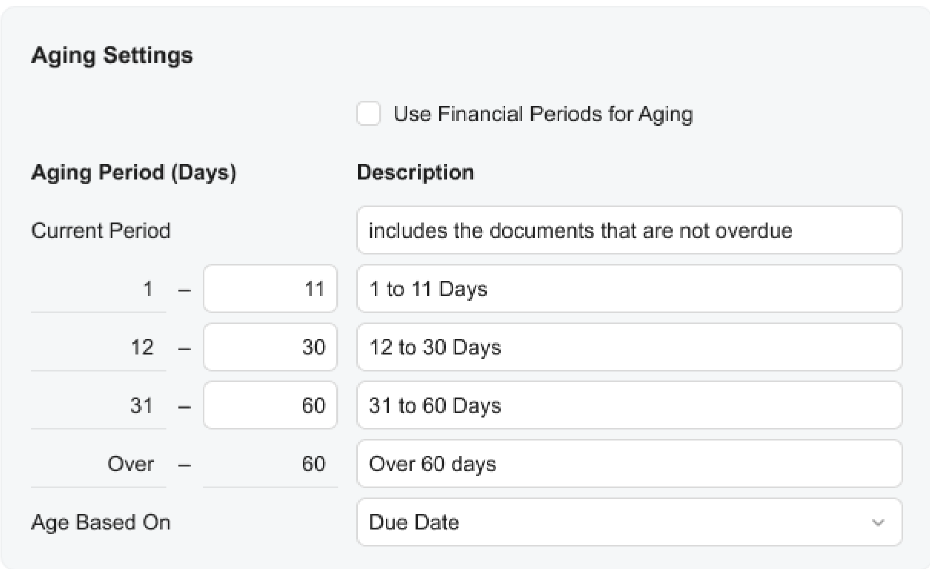


Figure: Complicated layout

The following code implements the layout above.

```
<qp-fieldset id="groupAgingSettings-ARStatementCycleRecord"
  view.bind="ARStatementCycleRecord" Caption="Aging Settings">
    <field name="UseFinPeriodForAging">
    </field>
    <field name="AgingPeriodsCaption" unbound>
        <qp-label slot="label" caption="Aging Period (Days)">
        </qp-label>
        <qp-label caption="Description">
        </qp-label>
    </field>
    <field name="AgeMsgCurrent">
        <qp-label slot="label" caption="Current Period">
        </qp-label>
    </field>
    <field name="AgeMsg00">
        <div slot="label" class="no-label h-stack">
            <qp-field
              control-state.bind=
                "ARStatementCycleRecord.Bucket01LowerInclusiveBound"
              class="col-4">
            </qp-field>
            -
            <qp-field
              control-state.bind="ARStatementCycleRecord.AgeDays00"
```

```
                class="col-4">
            </qp-field>
        </div>
    </field>
    <field name="AgeMsg01">
        <div slot="label" class="no-label h-stack">
            <qp-field
              control-state.bind=
                "ARStatementCycleRecord.Bucket02LowerInclusiveBound"
              class="col-4">
            </qp-field>
            -
            <qp-field
              control-state.bind="ARStatementCycleRecord.AgeDays01"
              class="col-4">
            </qp-field>
        </div>
    </field>
    <field name="AgeMsg02">
        <div slot="label" class="no-label h-stack">
            <qp-field
              control-state.bind=
                "ARStatementCycleRecord.Bucket03LowerInclusiveBound"
              class="col-4">
            </qp-field>
            -
            <qp-field
              control-state.bind="ARStatementCycleRecord.AgeDays02"
              class="col-4">
            </qp-field>
        </div>
    </field>
    <field name="AgeMsg03">
        <div slot="label" class="no-label h-stack">
            Over
            <qp-field
              control-state.bind=
                "ARStatementCycleRecord.Bucket04LowerInclusiveBound"
              class="col-4">
            </qp-field>
        </div>
    </field>
    <field name="AgeBasedOn">
    </field>
</qp-fieldset>
```

Multiple Columns in a Fieldset

Suppose that you need to implement the layout that is shown in the following screenshot.

Execution

Start day	Start time	Trips per day
☑ Sunday	10:00 AM	222
☑ Monday	10:00 AM	2
☐ Tuesday		
☐ Thursday		
☐ Wednesday		
☐ Friday		
☐ Saturday		

Figure: Multiple columns in a fieldset

The following code implements the layout above.

```
<qp-fieldset id="RouteSelected_10"
  view.bind="RouteSelected">
    <field name="fake01" unbound replace-content>
        <qp-label caption="Start day" class="col-4">
        </qp-label>
        <qp-label caption="Start time" class="col-4">
        </qp-label>
        <qp-label caption="Trips per day" class="col-4">
        </qp-label>
    </field>
    <field name="fake02" unbound replace-content>
        <qp-field control-state.bind="RouteSelected.ActiveOnSunday"
          no-label="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.BeginTimeOnSunday_Time"
          no-label="true"
          config-time-mode.bind="true" class="col-4">
        </qp-field>
        <qp-field control-state.bind="RouteSelected.NbrTripOnSunday"
          no-label="true" class="col-4">
        </qp-field>
    </field>
    <field name="fake03" unbound replace-content>
        <qp-field control-state.bind="RouteSelected.ActiveOnMonday"
          no-label="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.BeginTimeOnMonday_Time"
          no-label="true"
          config-time-mode.bind="true" class="col-4">
        </qp-field>
        <qp-field control-state.bind="RouteSelected.NbrTripOnMonday"
          no-label="true" class="col-4">
        </qp-field>
```

```
    </field>
    <field name="fake04" unbound replace-content>
        <qp-field control-state.bind="RouteSelected.ActiveOnTuesday"
          no-label="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.BeginTimeOnTuesday_Time"
          no-label="true"
          config-time-mode.bind="true" class="col-4">
        </qp-field>
        <qp-field control-state.bind="RouteSelected.NbrTripOnTuesday"
          no-label="true" class="col-4">
        </qp-field>
    </field>
    <field name="fake05" unbound replace-content>
        <qp-field control-state.bind="RouteSelected.ActiveOnWednesday"
          no-label="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.BeginTimeOnWednesday_Time"
          no-label="true"
          config-time-mode.bind="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.NbrTripOnWednesday"
          no-label="true" class="col-4">
        </qp-field>
    </field>
    <field name="fake06" unbound replace-content>
        <qp-field control-state.bind="RouteSelected.ActiveOnThursday"
          no-label="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.BeginTimeOnThursday_Time"
          no-label="true"
          config-time-mode.bind="true" class="col-4">
        </qp-field>
        <qp-field control-state.bind="RouteSelected.NbrTripOnThursday"
          no-label="true" class="col-4">
        </qp-field>
    </field>
    <field name="fake07" unbound replace-content>
        <qp-field control-state.bind="RouteSelected.ActiveOnFriday"
          no-label="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.BeginTimeOnFriday_Time"
          no-label="true"
          config-time-mode.bind="true" class="col-4">
        </qp-field>
        <qp-field control-state.bind="RouteSelected.NbrTripOnFriday"
          no-label="true" class="col-4">
        </qp-field>
    </field>
    <field name="fake08" unbound replace-content>
        <qp-field control-state.bind="RouteSelected.ActiveOnSaturday"
          no-label="true" class="col-4">
```

```
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.BeginTimeOnSaturday_Time"
          no-label="true"
          config-time-mode.bind="true" class="col-4">
        </qp-field>
        <qp-field
          control-state.bind="RouteSelected.NbrTripOnSaturday"
          no-label="true" class="col-4">
        </qp-field>
    </field>
</qp-fieldset>
```

Formula Editor

In this chapter, you will learn about the configuration of the formula editor. You'll learn when to use this control and how to organize it in a layout.

Formula Editor: General Information

The formula editor is a control that gives a user the ability to select or compose a formula. The formula editor control is composed of a *formula box* and *Formula Editor dialog box*.

The formula editor control is implemented with the `qp-formula-editor` control in the Modern UIand replaces the following controls from the Classic UI:

- `CTFormulaInvoiceEditor`
- `CTFormulaTransactionEditor`
- `RMFormulaEditor`
- `PMFormulaEditor`
- `PXFormulaEditor`
- `PXFormulaCombo`

Learning Objectives

In this chapter, you will learn the following about the formula editor:

- The design guidelines for the formula editor, including the naming conventions and layout recommendations
- The proper configuration of the formula editor for specific cases, such as custom options in the Formula Editor dialog box

Applicable Scenarios

You configure the formula editor in the following cases:

- You need to give users the ability to select an existing formula from a combo box
- You need to give users the ability to compose custom formulas

Formula Box

A *formula box* is a box of the formula editor where the formula is displayed. The formula box can be added as a column or as a box in a fieldset.

The formula box can behave like a text box or a combo box depending on the attributes on the corresponding DAC field. The following screenshot shows an example of formula boxes.

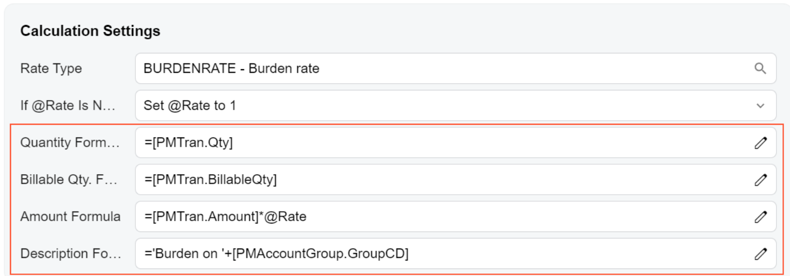


Figure: Formula boxes

In the following screenshot, you can see an example of a formula box in a column.

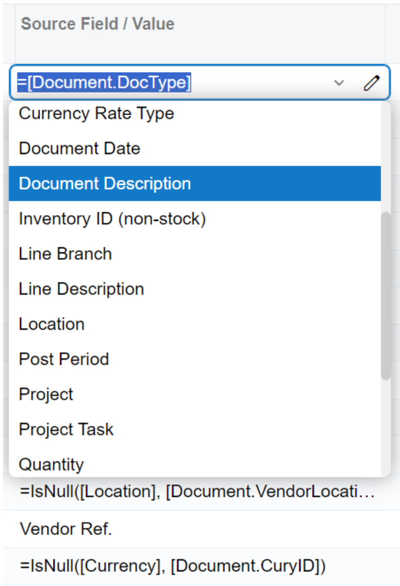


Figure: A formula box in a column

You can also configure the formula editor control to be displayed as a combo box, as shown in the following screenshot. For details, see *Formula Editor: Configuration of a Combo Box Field for the Formula Editor*.

Formula Editor Dialog Box

A *Formula Editor dialog box* is a dialog box where a user can compose a formula. The dialog box (see the following screenshot) opens when a user clicks the Edit button in a formula box.

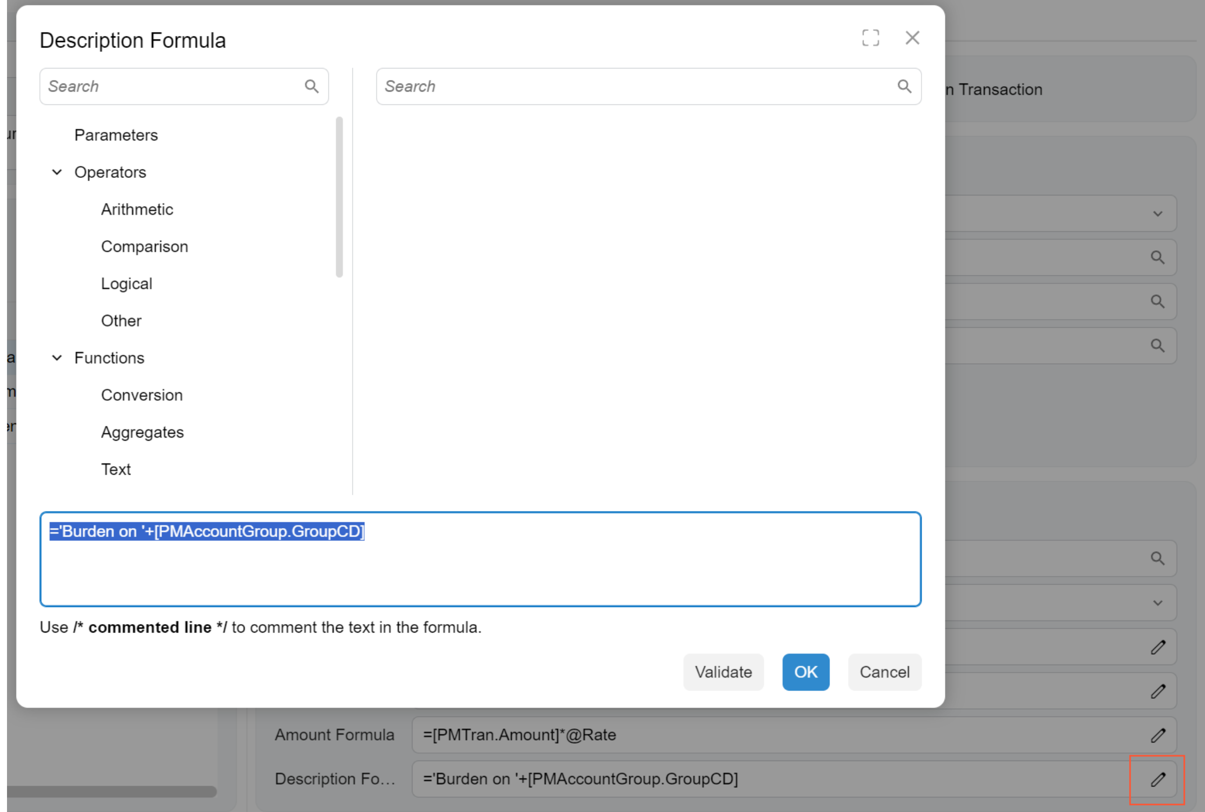


Figure: The Formula Editor dialog box

The formula editor can be implemented with the following tags:

- `qp-formula-dialog` for a text box and a formula editor dialog box
- `qp-formula-combo` for a drop-down control and a formula editor dialog box

However, you do not need to add these tags to the HTML template; the dialog box is displayed by default.

Configuration of a Formula Editor

You can configure the formula box and Formula Editor dialog box in the following locations:

- In the DAC field attributes—specifically, by using the *`PXFormulaEditor`* attribute.

> We recommend that you configure the control by using DAC field attributes. For details, see *Formula Editor: Implementation of a Standard Formula Editor*.

- In TypeScript, in the *`columnConfig`* or *`fieldConfig`* decorator.
- In HTML template, in the `config` attributes.

If you have used the `PXFormulaEditor` attribute, you do not need to specify the control type anywhere. It will be determined automatically as `"qp-formula-editor"`. In case the control has a formula editor functionality configured using the attributes and a list of combo box values (for example, configured by using the `PXStringList` attribute), the type of the control is determined automatically as `"qp-formula-combo"`.

If you do not use the `PXFormulaEditor` attribute on a DAC field, to add the formula editor, you need to specify the type of the control in the view definition in TypeScript as follows:

- If the control is a fieldset, in the `fieldConfig` decorator, specify `control-type="qp-formula-editor"`.

- If the control is in a table, in the `columnConfig` decorator, specify `editor-type="qp-formula-editor"`.

Formula Editor: Implementation of a Standard Formula Editor

To configure the formula editor, you should use the following attributes on the DAC field:

- The `PXFormulaEditor` attribute, which indicates that the control is a formula editor.

  By using this attribute, you can configure the display name for the formula box and specify the data type of the field. The attribute does not configure the Formula Editor dialog box or fill it with formula parameters.
- Descendants of `PXFormulaEditor.OptionsProviderAttribute`. These attributes provide options that are independent from the graph and are called *option providers*.

  Currently, the following descendants of `PXFormulaEditor.OptionsProviderAttribute` are available to configure the formula editor:
  - The `AddFunctions` attribute, which adds standard functions to the formula editor.
  - The `AddOperators` attribute, which adds standard operations to the formula editor.
  - The `AddStyles` attribute, which adds standard styles to the formula editor.

Suppose that you need to configure a formula editor for a field, and it should be displayed as a combo box in a column in the UI. The configuration of this field in the backend and frontend consists of the following parts:

1. The field configuration in the DAC, as shown below.

```
#region Field
public abstract class field : PX.Data.BQL.BqlString.Field { }
[PXFormulaEditor(DisplayName = "Data Field", IsDBField = true)]
[PXFormulaEditor.AddOperators]
[PXFormulaEditor.AddFunctions]
[PXStringList(new string[] { null }, new string[] { "" }, ExclusiveValues = false)]
[PXDefault]
public string Field { get; set; }
#endregion
```

You do not need to specify the `PXUIField` or `PXDBString` attribute when you are using the `PXFormulaEditor` attribute.

2. The field configuration in TypeScript, as shown in the following code.

```
@columnConfig({
  editorType: "qp-formula-editor",
  comboBox: true
})
SourceFieldOrValue: PXFieldState<PXFieldOptions.CommitChanges>;
```

In this case, the configuration of the field in HTML is not necessary because the field is included in the `qp-grid` tag.

Formula Editor: Configuration of a Combo Box Field for the Formula Editor

To configure the formula editor to be displayed as a combo box, you need to do the following:

1. In the DAC field, add the `PXStringList` attribute, as shown in the following code.

```
#region Field
public abstract class field : PX.Data.BQL.BqlString.Field { }
[PXFormulaEditor(DisplayName = "Data Field", IsDBField = true)]
[PXFormulaEditor.AddOperators]
[PXFormulaEditor.AddFunctions]
[PXStringList(new string[] { null }, new string[] { "" }, ExclusiveValues = false)]
[PXDefault]
public string Field_Name { get; set; }
#endregion
```

2. If the values in the combo box are dynamic, populate the field with the combo box values by using the `PXStringListAttribute.SetList` method in the graph code—for example, in the `RowSelected` event handler.

   If the values are static, you can populate them in the `PXStringList` attribute in the DAC, as the following code shows.

```
PXStringListAttribute.SetList<DAC_Name.Field_Name>(
  sender, row,
  new string[]
  {
    OrderStatus.OnHold,
    OrderStatus.Shipping,
  },
  new string[]
  {
    "On Hold",
    "Shipping",
});
```

3. In the HTML template or TypeScript code, enable the combo box mode for the field, as shown below.

```
@columnConfig({comboBox: true})
field_Name : PXFieldState;
```

Related Links

- *Combo Box: Configuration*

Formula Editor: Implementation of Custom Option Providers

In the Formula Editor dialog box, you can add options that are dependent on the logic in the graph (such as a list of available fields). To do this, you need to define a *custom option provider* and apply it to the DAC field by using the `CacheAttached` event handler.

To define a custom option provider, you do the following:

1. In the graph, define the option provider as a nested class of the graph.

   The class should inherit from `PXFormulaEditor.OptionsProviderAttribute`, as shown in the following example. Also, the class should be defined inside the graph. This way, you make sure that the option provider will not be used outside the graph. Later, the new class will be used as an attribute on the `CacheAttached` event handler.

```
public class GenericInquiryDesigner : PXGraph<GenericInquiryDesigner>
{
```

```
  ...
  public class PXFormulaEditor_AddFieldsAttribute :
    PXFormulaEditor.OptionsProviderAttribute
  {
  }
}
```

2. In the new class, override the `ChangeOptionsSet(PXGraph graph, ISet options)` method.
3. In the `ChangeOptionsSet` method, populate the array of options. Each element of the array is an object of the `FormulaObject` type that has two fields: `Category` and `Value`. `Value` is the actual component of the formula (operand or operator), and `Category` is the path to the component in the tree. In the following screenshot, the `Category` values are shown in Item 1, and `Value` values are shown in Item 2.

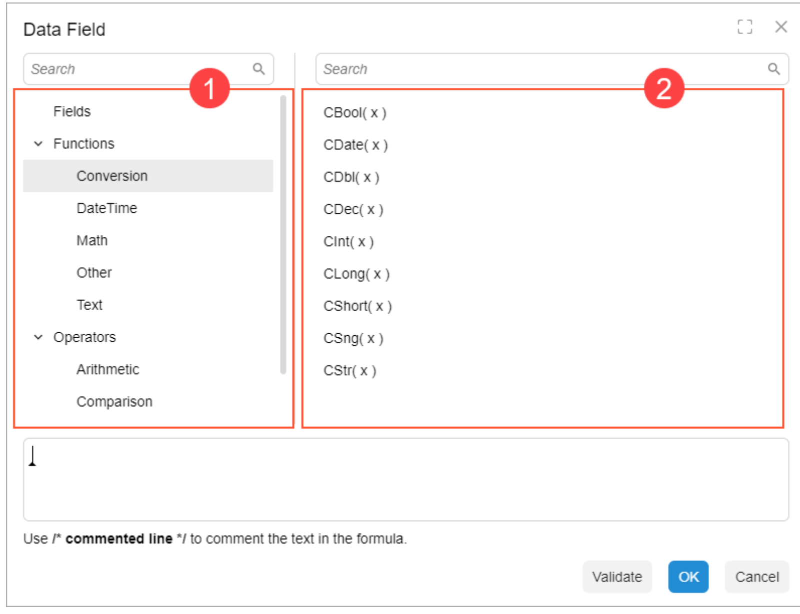


Figure: Category and value in the Formula Editor dialog box

The `Category` field should have the following structure: `<Option_name>/<Category_name>`. For example, if you are defining the set of fields for the formula editor, the `Category` value is `"Fields/<Table_name>"`.

The order of categories is determined by the system: All categories are sorted in alphabetic order. You cannot change the order of categories in the code.

The following example from the `GenericInquiryDesigner` class shows how to add a set of parameters from multiple tables.

```
public class PXFormulaEditor_AddParametersAttribute :
  PXFormulaEditor.OptionsProviderAttribute
{
  public string CategoryName { get; set; }
  public override void ChangeOptionsSet(PXGraph graph, ISet<FormulaOption> options)
  {
    var designer = (GenericInquiryDesigner)graph;
    var parameters = designer.GetAllParameters();
    var fields = designer.GetAllFields();

    foreach (var parameter in parameters.Concat(fields))
    {
```

```
        options.Add(new FormulaOption
        {
          Category = CategoryName,
          Value = parameter
        });
      }
    }
}
```

4. Connect the option provider by using the `CacheAttached` mechanism to the DAC field in the graph. To do that, in the graph, define the `CacheAttached` event handler for the field where you want to configure the formula editor. On the event handler, add the `PXMergeAttributes` attribute and the new attribute you defined in the nested class.

```
public class GenericInquiryDesigner : PXGraph<GenericInquiryDesigner>
{
  public class PXFormulaEditor_AddParametersAttribute :
    PXFormulaEditor.OptionsProviderAttribute {...}

  [PXMergeAttributes]
  [PXFormulaEditor_AddParameters(CategoryName = "Fields")]
  protected virtual void _(Events.CacheAttached<GIDesign.rowStyleFormula> e) { }
}
```

Formula Editor: Customization of Option Providers

You can customize the option sets provided by the `AddFunctions`, `AddOperators`, and `AddStyles` attributes by doing one of the following:

- Defining a custom option provider that inherits from `PXFormulaEditor.OptionsProviderAttribute` and defining the option set, as described in *Formula Editor: Implementation of Custom Option Providers*.
- Defining a custom option provider that inherits from one of the existing attributes and modifying the existing option set. You do this by adding, updating, or deleting options from the set.

Do not use both the custom option provider and its parent attribute (`AddFunctions`, `AddOperator`, or `AddStyles`) together. This could cause the changes to be applied unpredictably in the child attribute.

You can use any combination of standard attributes and descendants of `PXFormulaEditor.OptionsProviderAttribute` together.

For example, you can use this approach to remove multiple values from the `AddStyles` attribute, as shown in the following code.

```
public class RemoveRedStylesAttribute : OptionsProviderAttribute
{
    public override void ChangeOptionsSet(PXGraph graph, ISet<FormulaOption> options)
    {
        options.Remove(DefaultFormulaOptions.Styles.Red);
        options.Remove(DefaultFormulaOptions.Styles.Red60);
        options.Remove(DefaultFormulaOptions.Styles.Red40);
        options.Remove(DefaultFormulaOptions.Styles.Red20);
        options.Remove(DefaultFormulaOptions.Styles.Red00);
    }
```

```
}
```

You can attach a custom option provider in the following places:

- On the DAC field in the DAC definition.

  We recommend this approach when you write original code and the option set does not depend on the graph logic.
- On the `CacheAttached` event handler in the graph code.

  We recommend this approach when you customize existing code and the option set depends on the graph logic.
- On the DAC extension.

  We recommend this approachwhen the change should be applied to the DAC regardless of where it is used.

The following diagram shows how you should select the approach to define a custom option provider.

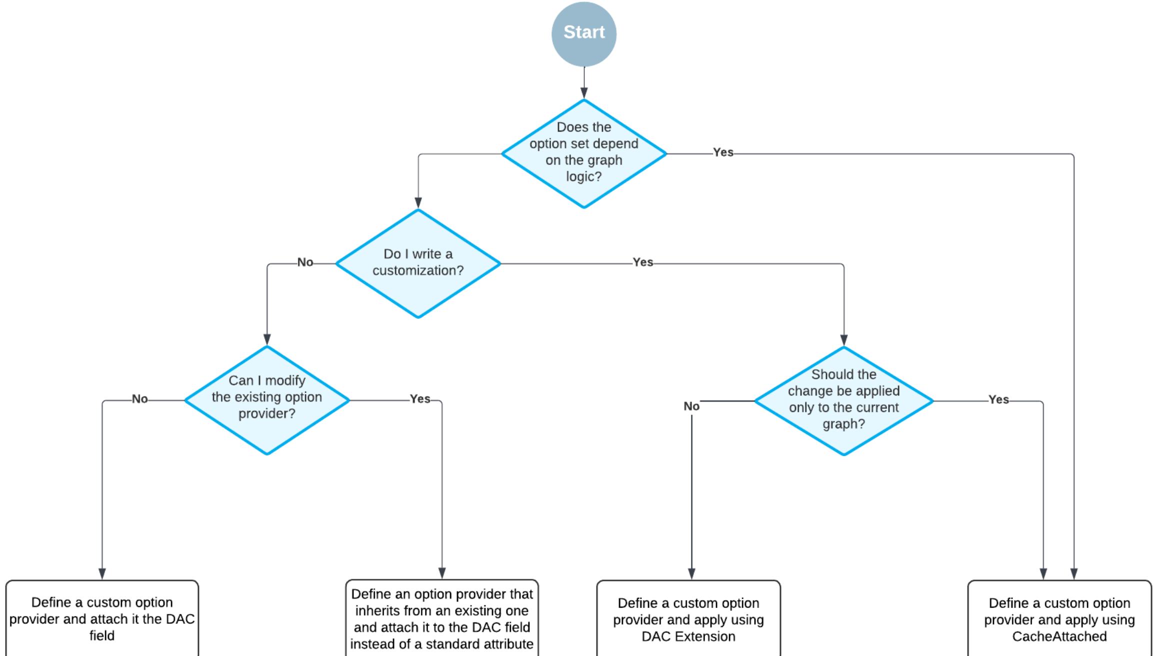


Figure: Selecting a way to customize an option provider

Icon

In this chapter, you will learn about the configuration of icons and the key details of each property of an icon.

Icon: General Information

An icon is an image that provides a visual representation of a particular function of the system. An icon is generally placed over a UI element that a user interacts with, such as a button. An example is the icon used to represent the **Save** button on the form toolbar of a form.

An icon is defined by `PXImage` in the Classic UI. In the Modern UI, an icon is defined by the *qp-icon* HTML tag.

Learning Objectives

In this chapter, you will learn the following about an icon:

- The proper configuration of an icon
- The key details of each property of an icon

Applicable Scenario

You configure an icon when you want to use an image to visually represent an action on the UI of a form.

Icon ID

An ID of an icon in HTML consists of two parts, the `icon` prefix and the semantic name. The semantic name describes the purpose of the element. For example, you may have an icon that represents drag-and-drop functionality in a file upload dialog box. You can set its ID to `iconDragDrop`, as the following code shows.

```
<qp-icon id="iconDragDrop"></qp-icon>
```

Configuration of an Icon

To configure an icon, you use the *qp-icon* HTML tag and specify its `imagesrc` property. The image source can be a Font Awesome glyph, an SVG icon, or a simple raster icon, such as a PNG or a GIF file. The following code shown an example of the configuration of a `qp-icon` control that uses a Font Awesome glyph as the image source.

```
<qp-icon imagesrc="main@Excel"></qp-icon>
```

The height and the width of the icon are adjusted automatically in most cases, and we recommend that you not specify the `height` and `width` attributes of the `qp-icon` control manually.

The following example shows how you can specify an SVG icon as the image source.

```
<qp-icon imagesrc="svg:main@filledKey"></qp-icon>
```

The following example shows how you can specify a simple raster icon as the image source.

```
<qp-icon imagesrc="icons/disk.png"></qp-icon>
```

Icon: Conversion from ASPX to HTML and TypeScript

The following table will help you to convert the ASPX elements that are related to an icon to HTML or TypeScript elements.

PXImage

The following table shows the correspondence between the `PXImage` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXImage`

```
<px:PXImage
  ID="PXImage1"
  runat="server"
  ImageUrl="main@Excel"
  Width="16"
  Height="16"
/>
```

</td><td>

Use the `qp-icon` tag, as shown in the following code.

```
<qp-icon
  imagesrc="main@Excel">
</qp-icon>
```

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

`ID`

</td><td>

Use the `id` attribute of the `qp-icon` tag.

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

`ImageUrl`

</td><td>

Use the `imagesrc` attribute of the `qp-icon` tag. The image source can be a Font Awesome glyph, an SVG icon, or a simple raster icon, such as a PNG or a GIF file.

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

`Height`

</td><td>

Use the `height` attribute of the `qp-icon` tag.

```
<qp-icon  ...
  height="16px">
</qp-icon>
```

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

`Width`

</td><td>

Use the `width` attribute of the `qp-icon` tag.

```
<qp-icon  ...
  width="16px">
</qp-icon>
```

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

Image Uploader

In this chapter, you will learn about the configuration of image uploader, which is used to select and upload images.

Image Uploader: General Information

An image uploader is a control that is used to select and upload images. The following screenshot shows an example of an image uploader control.

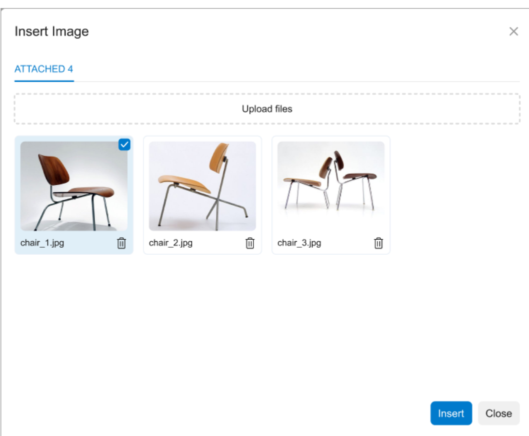


Figure: The image uploader control

An image uploader control is defined by `PXImageUploader` in the Classic UI. In the Modern UI, an image uploader control is defined by the *`qp-image-uploader`* tag.

Learning Objectives

In this chapter, you will learn how to convert the ASPX elements of an image uploader control to HTML or TypeScript.

Applicable Scenarios

You configure an image uploader control when a user needs to attach images to a record.

Image Uploader: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the image uploader control to HTML or TypeScript elements.

PXImageUploader

The following table shows the correspondence between `PXImageUploader` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXImageUploader`

```
<px:PXImageUploader
  Height="320px"
  Width="430px"
  ID="imgUploader"
  runat="server"
  DataField="ImageUrl"
  AllowUpload="True"
  AllowNoImage="True"
  ShowComment="True"
  DataMember="ItemSettings"
/>
```

</td><td>

Use the `qp-image-uploader` tag, as shown in the following code.

```
<qp-image-uploader
  id="ImageUrl"
  state.bind="Item.ImageUrl"
  value.two-way="Item.ImageUrl.value"
  width="300px"
  height="300px">
</qp-image-uploader>
```

Alternatively, you can use the `field` tag and specify `qp-image-uploader` as the value of the `control-type` attribute, as shown in the following code.

```
<field
  name="ImageUploader"
  control-type="qp-image-uploader"
/>
```

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

`AllowNoImage`

```
<px:ImageUploader ID="imgUploader" ...
  AllowNoImage="True" />
```

</td><td>

Use the `allowNull` property of the `config` attribute.

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

`AllowUpload`

```
<px:ImageUploader ID="imgUploader" ...
  AllowUpload="True" />
```

</td><td>

Use the `readOnly` property of the `config` attribute.

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

`DataField`

</td><td>

Use the `state` and `value` attributes of the `qp-image-uploader` tag, as shown in the following code.

```
<qp-image-uploader
  state.bind="Item.ImageUrl"
  value.two-way="Item.ImageUrl.value"
>
</qp-image-uploader>
```

Alternatively, you can use the `name` property of the `field` tag in HTML to specify the DAC field name, as shown in the following code.

```
<field
  name="Item.ImageUrl" ... >
  control-type="qp-image-uploader"
</field>
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>DataMember
<code><px:ImageUploader ID="imgUploader" ...
  DataMember="ItemSettings" /></code></td><td>Use the <code>viewName</code> property of the <code>config</code> attribute.</td></tr>
<tr><td>Height</td><td>Use the <code>height</code> attribute of the <code>qp-image-uploader</code> tag.
<code><qp-image-uploader
  ...
  height="300px">
</qp-image-uploader></code></td></tr>
<tr><td>Width</td><td>Use the <code>width</code> attribute of the <code>qp-image-uploader</code> tag.
<code><qp-image-uploader
  ...
  width="300px">
</qp-image-uploader></code></td></tr>
</table>

Obsolete ASPX Control and Property

The following table lists the obsolete ASPX element that is related to the image uploader control. You do not need to replace this ASPX element with any HTML or TypeScript element.

ASPX Control	Properties
PXImageUploader	• runat • SuppressLabel

Image Viewer

In this chapter, you will learn about the configuration of image viewers.

Image Viewer: General Information

An image viewer is a control that is used to display an image. This control displays the image by using a URL that is stored in a field that is bound to this control. The following screenshot shown an example of an image viewer control.

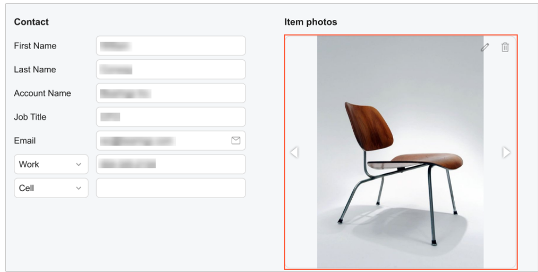


Figure: The Image Viewer control

An image viewer control is defined by `PXImageView` in the Classic UI. In the Modern UI, an image viewer control is defined by the *qp-image-view* tag.

Learning Objectives

In this chapter, you will learn the following about an image viewer control:

- The various approaches to specifying the size of an image viewer control
- The conversion of the ASPX elements of an image viewer control to HTML or TypeScript

Applicable Scenarios

You configure an image viewer control when a user needs to view images that are associated with a record.

Specifying the Size of an Image Viewer Control

You can specify the size of an image viewer control in the following ways:

- By specifying a fixed size: You can specify the size in pixels or as a percentage. You can also specify a fixed size by using the viewport height or width. The following code shows an example with each of these units.

  ```
  <qp-image-view width="400px" height="280px">
  </qp-image-view>
  <qp-image-view width="50%" height="50vw">
  </qp-image-view>
  <qp-image-view width="100vh" height="50px">
  </qp-image-view>
  ```

- By specifying the size proportionally based on one dimension: You can specify one dimension (either the width or the height) as a specific proportion of the overall size of the image that is to be displayed. Before the image is loaded and displayed in the control, the control will have a size of *0* for the dimension that was

not specified. After the image is loaded, the control's size will be based on the specified proportion of the image size. The following code shows an example.

```
<qp-image-view width="400px"></qp-image-view>
<qp-image-view height="100%"></qp-image-view>
<qp-image-view width="100vh"></qp-image-view>
```

- By specifying the size proportionally based on width and height: You can specify the proportions by using fractions, which are represented by *fr* in code. To specify the width as *400px*, and the height as *800px*, you can use the following code.

```
<qp-image-view width="400px 1fr" height="2fr"></qp-image-view>
```

- By specifying the size by using the `aspect-ratio` property: You can specify one dimension explicitly, and the other dimension will be calculated by the system based on the specified value for the `aspect-ratio` property. If none of the dimensions have been specified, the system will calculate the size based on the dimensions of the parent container while considering the value specified for the `aspect-ratio` property. The following code shows an example.

```
<qp-image-view height="100%" aspect-ratio="2/3"></qp-image-view>
```

  In the code above, the specified the value for the `height` property is *100%* and the value for the `aspect-ratio` property is *2/3*. The number *2* represents the proportion of the width, whereas *3* represents the proportion of the height. For more details, see *aspect-ratio*.

The values of the following properties are determined in the same way as the `width` and `height` properties are, as described in the preceding list:

- `max-width` and `max-height`
- `min-width` and `min-height`

You can also control the way an image is scaled inside an image viewer control by using the `object-fit` property. This property has the following values:

- *contain*
- *cover*
- *fill*
- *scale-down*

The following code shows an usage example.

```
<qp-image-view aspect-ratio="2/3" object-fit="cover">
</qp-image-view>
```

In the code above, the size of the control is determined by the parent control. The `aspect-ratio` of the dimensions is set to *2/3*. This means that the image will always cover the whole control and keep its proportions. The image will be cropped if the ratio of the dimensions is something other than *2/3*. For me details, see *object-fit*.

Image Viewer: Loading of Images from a Third-Party Source

When a user specifies an image from a third-party source (such as a third-party integration), the Cross-Origin Resource Sharing (CORS) policy may prevent the system from loading it.

To resolve this issue, in the `qp-image-view` control, specify `showProgress: false`, as shown in the following code. This setting disables the CORS policy and the progress bar for uploading the image.

```
export class SendGridDesignImportParameters extends PXView  {
 ...
```

```
  @controlConfig({showProgress: false})
  ThumbnailUrl : PXFieldState;
}
```

Image Viewer: Conversion from ASPX to HTML and TypeScript

The following table will help you to convert the ASPX elements that are related to image viewer control to HTML or TypeScript elements.

PXImageView

The following table shows the correspondence between the `PXImageView` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

`PXImageView`
`<px:PXImageView ID="edTeamsMemberPhoto" runat="server" DataField="PhotoFileName" Width="64" Height="64" />`
	Use the `qp-image-view` tag, as shown in the following code.
	`<qp-image-view id="PhotoFileName" width="64px" height="64px"></qp-image-view>`
	Alternatively, you can use the `field` tag and specify `qp-image-view` as the value of the `control-type` attribute, as shown in the following code.
	`<field name="PhotoFileName" control-type="qp-image-view"></field>`

`Height`	Use the `height` attribute of the `qp-image-view` tag. `<qp-image-view ... height="300px"></qp-image-view>`
`Width`	Use the `width` attribute of the `qp-image-view` tag. `<qp-image-view ... width="300px"></qp-image-view>`

Label

In this chapter, you will learn about the configuration of labels. You'll learn when to use labels and how to organize a layout that includes them.

Label: General Information

A label is a text that precedes a field value. You can specify a field to occupy the width of a label. Any tag can be used as a label, including `qp-label`.

Learning Objectives

In this chapter, you will learn the following about a label:

- The design guidelines for the label, including the naming conventions and layout recommendations
- The proper configuration of a label for specific cases, such as adding a single label for multiple controls

Applicable Scenarios

You configure a label in the following cases:

- You need to add a label for multiple UI controls located in a single line.
- You need to add explanatory text (at least one sentence) before a UI element such as a table

Label ID

An ID of a label in HTML consists of two parts, the `label` prefix and the semantic name separated by a hyphen. The semantic name repeats or summarizes the label text. For example, if a label contains a sentence and precedes a table on the **Billing** tab of a form, it may have the `label-Billing` ID, as the following code shows.

```
<qp-label id="label-Billing" caption="..."/>
```

UI Naming Conventions

For labels that precede one or multiple UI elements, you use the naming conventions for this type of element.

For labels that contain a whole sentence, use general rules of the language.

Label: A Combo Box Control as a Label

You can define a control, such as a combo box to occupy the space of a label.

Suppose that you need to display a combo box in place of a label for another control, as shown in the following screenshot.

Figure: A combo box in place of a label

In the HTML template, you need to add a field for that label by using the `qp-field` tag and specify the following attributes for the field:

- `slot="label"` to display that control in space for the label of the parent control
- `class="no-label"` to remove the space allocated for the label part of that control

The following code shows an example that implements the layout from the screenshot above.

```
<field name="Phone1">
  <qp-field slot="label" class="no-label"
    control-state.bind="PrimaryContactCurrent.Phone1Type">
  </qp-field>
</field>
```

Alternatively, you can add both fields explicitly, as shown in the following code.

```
<field name="groupPhone1" unbound>
  <qp-field slot="label" class="no-label"
    control-state.bind="DefContact.Phone1Type"></qp-field>
  <qp-field class="col-12 no-label"
    control-state.bind="DefContact.Phone1" ></qp-field>
</field>
```

Label: A Single Label for Multiple Controls

You can add a single label for multiple controls.

Suppose that you need to display a single label for two text boxes and both of these text boxes need to have equal widths, as shown in the following screenshot.

Figure: A single label for two boxes

In the HTML template, you need to do the following:

- Add the `field` tag for the first field, as usual
- Add the nested `qp-label` tag and specify `slot="label"` for that tag
- For the `qp-label` tag, specify `class="no-label"` to remove the space allocated for the label part of each control
- Add the nested `qp-field` control for the second field
- For the `qp-field` tag, specify `class="col-6"` so that each text box occupies 50% of the space

The following code implements the layout from the screenshot above.

```
<field name="FirstName">
  <qp-label slot="label" caption="Name"
    if.bind="PrimaryContactCurrent.FirstName.visible == true" class="no-label">
  </qp-label>
  <qp-field control-state.bind="PrimaryContactCurrent.LastName" class="col-6">
  </qp-field>
</field>
```

The system puts the controls in the following order:

1. The control with `slot="label"`
2. The control from the parent `field` tag
3. The control specified in the nested `qp-field` tag

Label: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the label control to HTML or TypeScript elements.

PXLabel

The following table shows the correspondence between the `PXLabel` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXLabel` `<px:PXLabel runat="server" ID="space1" />`	For labels with no text that are used to organize layout, use the appropriate templates instead. For details, see *Form Layout: Predefined Templates*.

`PXLabel`
`<px:PXLabel runat="server" ID="lblRecalcTaxes" Text="Taxes will be recalculated ..." Height="40px" Width="350px">`
`</px:PXLabel>`
	For labels that display an informational message, use the `qp-info-box` tag. For more details, see *Error, Warning, or Informational Notification: General Information*.
	`<qp-info-box caption="Taxes will be recalculated ..." type="info">`
	`</qp-info-box>`

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`Text`

```
<px:PXLabel
  Text="Taxes will be recalculated ..." >
</px:PXLabel>
```

</td><td>

Use the `caption` property of the `qp-info-box` control.

```
<qp-info-box
  caption="Taxes will be recalculat-
ed ...">
</qp-info-box>
```

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

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the label control. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXLabel`	• `ID` • `runat`

Link Editor

In this chapter, you will learn about the configuration of a link editor. You'll learn when to use a link editor and how to organize a layout that includes a link editor.

Link Editor: General Information

A link editor represents a text box with a link that a user can click and be redirected to another page.

A link editor is defined by `PXLinkEdit` in the Classic UI. In the Modern UI, you define a link editor either by using the `field` or `qp-field` tag with the *`qp-link-editor`* control type specified or explicitly by using the `qp-link-editor` control.

Learning Objectives

In this chapter, you will learn the following about the link editor:

- The design guidelines for the link editor, including the naming conventions
- The guidelines for converting the element from the Classic UI to Modern UI

Applicable Scenarios

You configure the link editor when you want to add a control with a link that a user can click and be redirected to another page.

Overview of the Link Editor

A link editor consists of a text box that contains the link and the dialog box where a user can specify the link name and URL. The text box part of the link editor contains a link icon at the right side of it, as shown in the following screenshot.

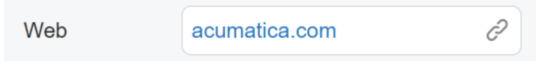


Figure: A link editor

When a user click the link icon, the **Edit Link** dialog box opens, as shown in the following screenshot. In this dialog box, a user can specify the link text which will be displayed in the link editor, and the link URL.

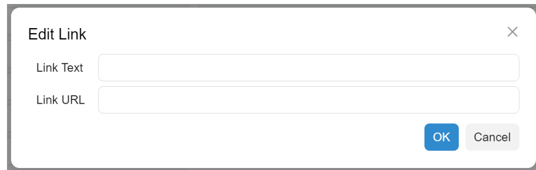


Figure: The Edit Link dialog box

After the user clicks **OK**, the link text is displayed in the link editor text box.

UI Naming Convention

The following table shows the UI naming convention for a link editor.

Naming Convention	Example
Use a noun or a noun phrase to describe the contents of a link editor. Preferably, link editor names should consist of one or two words.	The **URL** link editor on the *Webhooks* (SM304000) form, which is shown in the following screenshot. Webhooks Webhook Name * Active Implementation Class * Predefined URL

LinkEditor: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the link editor to HTML or TypeScript elements.

PXLinkEdit

The following table shows the correspondence between the `PXLinkEdit` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXLinkEdit` `<px:PXLinkEdit ID="edDefContactWebSite" runat="server" DataField="WebSite" CommitChanges="True" />`	Replace it with `field` with `control-type="qp-link-editor"`. In rare cases, you should replace it explicitly with the `qp-link-editor` control. `<field name="WebSite" control-type="qp-link-editor"></field>`
`DataField` `<px:PXLinkEdit DataField="WebSite" ... />`	Use the `name` attribute of the `field` tag. `<field name="WebSite" ...></field>`
`ID` `<px:PXLinkEdit ID="edDefContactWebSite" .../>`	Replace it with the `id` attribute of the `qp-field` tag if this tag is used as a replacement. In other cases, the ID is not necessary for a link editor.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>CommitChanges</code>
<pre><code><px:PXLinkEdit ... CommitChanges="True" /></code></pre></td><td>In the view class in TypeScript, specify <code>PXFieldOptions.CommitChanges</code>.
<pre><code>export class Contact extends PXView {
  WebSite: PXFieldState<PXFieldOptions.CommitChanges>;
  ...
}</code></pre></td></tr>
</table>

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the link editor. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXLinkEdit`	• `runat`

Mail Editor

In this chapter, you will learn about the configuration of the mail editor control, including when and how to use a mail editor.

Mail Editor: General Information

The mail editor control consists of a text field and a button in the UI. In the text field, a user can enter an email address. You can assign an action to the button. If you do not specify an action, then by default, the button launches the default mail client application that is configured on the system running the Acumatica ERP instance.

A mail editor is defined by `PXMailEdit` in the Classic UI. In the Modern UI, you define a mail editor either by using the `field` tag with the control type specified or explicitly by using the *`qp-mail-editor`* control.

Learning Objectives

In this chapter, you will learn the following about the mail editor:

- The common uses of the mail editor
- The proper configuration of the mail editor

Applicable Scenarios

You configure the mail editor when you want to give a user the ability to enter an email address and execute the appropriate action by clicking the button of this control.

Uses of the Mail Editor

You generally use the mail editor control on forms that store an email address. As mentioned, you can specify an action for the button associated with this control. Typically, this action launches either the default email client application of your system or the *Email Activity* (CR306015) form in a pop-up window.

The following screenshot shows an example of the mail editor control on the **General** tab of the *Customers* (AR303000) form.

Item 1 in the following screenshot shows the text field of the control, in which the user types an email address. Item 2 shows the button associated with the control, which the user clicks to execute the action you have configured.

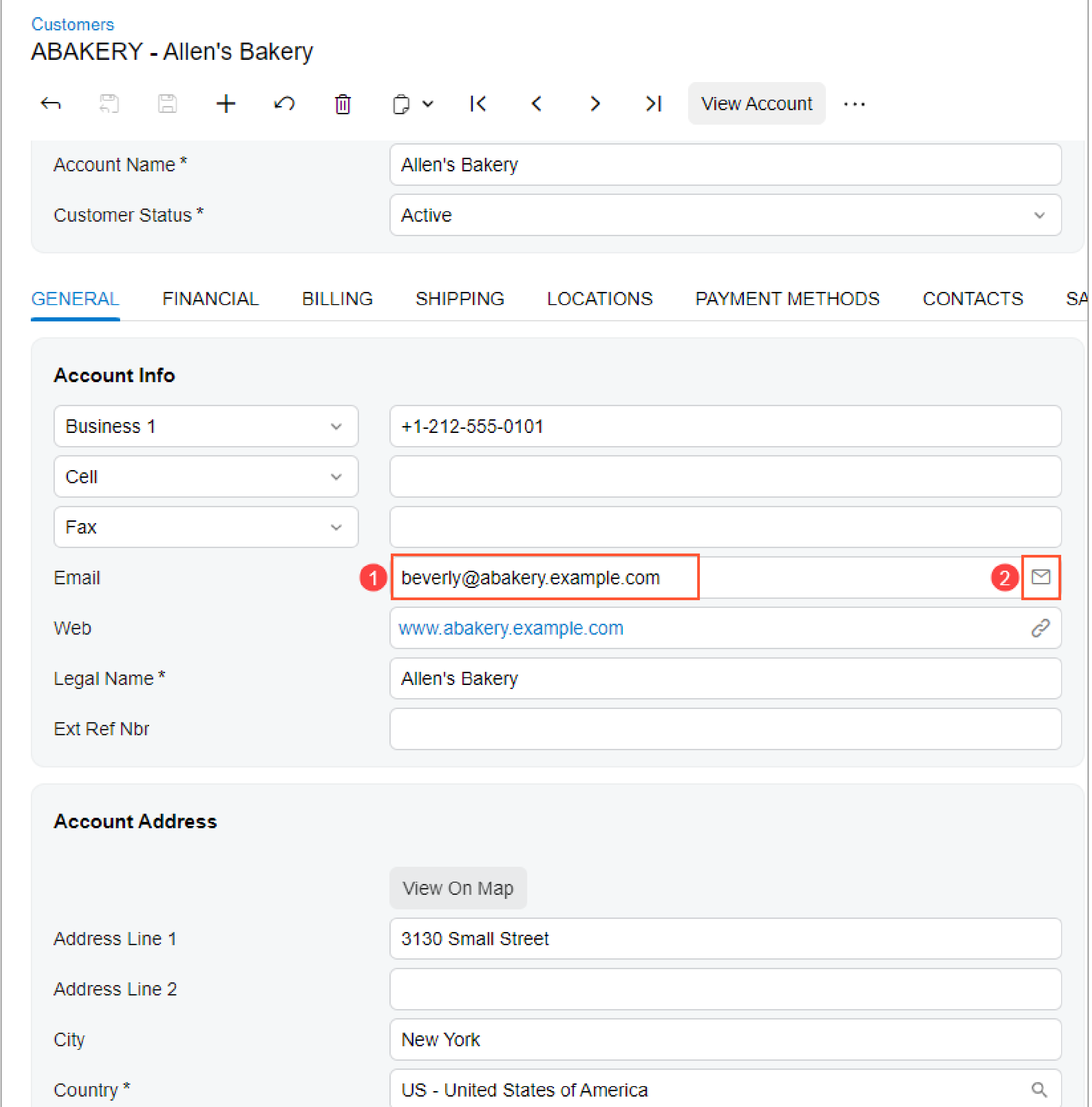


Figure: The Mail Editor control on the Customers form

The following screenshot shows the result of the user clicking the button of the control shown in the previous screenshot: The action specified for the button has opened the *Email Activity* form in a pop-up window. Notice that the email address that was specified in the text field of the control in the preceding screenshot has been automatically inserted into the **To** box.

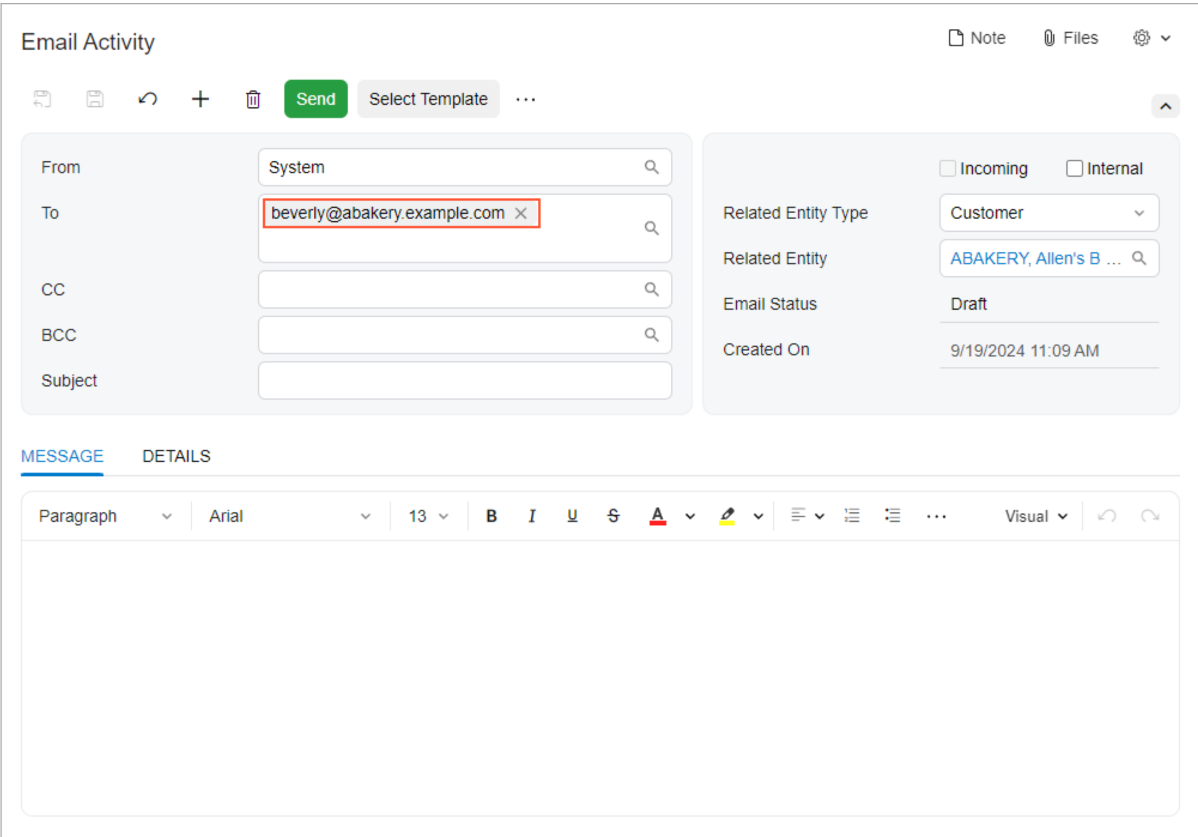


Figure: The pop-up window with the Email Activity form

Mail Editor: Configuration

The configuration parameters for the mail editor control are defined in the `IMailEditorControlConfig` interface. To configure a mail editor, you typically use the `field` tag with the control type specified as `qp-mail-editor`, as shown in the following code example.

```
<field name="Email" control-type="qp-mail-editor"></field>
```

Alternatively, you can use the `qp-mail-editor` tag explicitly to configure the mail editor, as shown in the following code example.

```
<qp-mail-editor state.bind="Email"></<qp-mail-editor>
```

Mail Editor: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to a mail editor to HTML or TypeScript elements.

PXMailEdit

The following table shows the correspondence between the `PXMailEdit` element and the HTML or TypeScript element. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXMailEdit`

```
<px:PXMailEdit ... />
```

</td><td>

Replace with `field` and specify `qp-mail-editor` as the control type, as shown in the following code.

```
<field
  ...
  control-type="qp-mail-editor">
</field>
```

Alternatively, you can use the `qp-mail-editor` tag to explicitly define the control, as shown in the following code.

```
<qp-mail-editor
  ... >
</<qp-mail-editor>
```

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

`CommandName`

```
<px:PXMailEdit
  CommandName="RunEmailActivity"
  ... />
```

</td><td>

Use the `action` property of the `config` attribute of the `qp-mail-editor` control, as shown below.

```
<field
  ...
  control-type="qp-mail-editor"
  config.bind="{action: 'RunEmailActivi-
ty'}">
</field>
```

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

`DataField`

```
<px:PXMailEdit
  DataField="Email"
  ... />
```

</td><td>

Use the `name` property of the `field` tag, as shown in the following code.

```
<field name="Email"
  control-type="qp-mail-editor">
</field>
```

If you defined the control explicitly by using the `qp-mail-editor` tag, use the `state.bind` property of this tag, as shown in the following code.

```
<qp-mail-editor
  state.bind="Email">
</<qp-mail-editor>
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`ID`

```
<px:PXMailEdit
  ID="edDefContactEmail"
  ... />
```

</td><td>

Use the `id` property of the `config` attribute of the `qp-mail-editor` control, as shown below.

```
<field name="Email"
  control-type="qp-mail-editor"
  config.bind="{id: 'edDefContactEmail'}">
</field>
```

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

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to mail editors. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXMailEdit`	• `runat` • `CommandSourceID`

Mask Editor

In this chapter, you will learn about the configuration of the mask editor control, including when and how to use a mask editor.

Mask Editor: General Information

The mask editor control represents a text box with an input mask.

A mask editor is defined by `PXMaskEdit` in the Classic UI. In the Modern UI, you define a mask editor by using the `field` tag with the control type specified in TypeScript or explicitly by using the *`qp-mask-editor`* control.

Learning Objectives

In this chapter, you will learn the following about the mask editor:

- The common usages of the mask editor
- The proper configuration of the mask editor

Applicable Scenarios

You configure the mask editor when you want to ensure that a user enters a text value correctly based on a predefined format (mask). For example, you might want the user to enter a postal code that consists of two letters and four numerical digits.

Configuration of a Mask Editor

To configure a mask editor, you need to specify the control type in TypeScript as follows:

- If the control is a text box inside a template, use the `fieldConfig` decorator, as shown in the following code.

  ```
  @fieldConfig({controlType: "qp-mask-editor"})
  PostalCode: PXFieldState<PXFieldOptions.CommitChanges>;
  ```

- If the control is a column in a table, use the `columnConfig` decorator, as shown in the following code.

  ```
  @columnConfig({
    width: 200,
    editorType: "qp-mask-editor",
    editorConfig: { emptyChar: "0"} })
  StartIPAddress: PXFieldState;
  ```

You do not need to specify the control type as `"qp-mask-editor"` if the mask is applied as a part of the selector control.

Mask Editor: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the mask editor to HTML or TypeScript elements.

PXMaskEdit

The following table shows the correspondence between the `PXMaskEdit` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXMaskEdit`

```
<px:PXMaskEdit
  ID="edStartIPAddress"
  runat="server"
  DataField="StartIPAddress"
  EmptyChar="0"
  InputMask="###.###.###.###" />
```

</td><td>

Use the `fieldConfig` or `columnConfig` decorator to specify the type of the control and its properties.

```
@columnConfig({
  editorType: "qp-mask-editor",
  editorConfig: { emptyChar: "0"} })
StartIPAddress: PXFieldState;
```

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

`EmptyChar`

```
<px:PXMaskEdit ...
 EmptyChar="0" />
```

</td><td>

Use the `config.emptyChar` property of the `qp-mask-editor` control.

```
@columnConfig({ ...
  editorConfig: { emptyChar: "0"} })
StartIPAddress: PXFieldState;
```

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

`InputMask`

```
<px:PXMaskEdit ...
  InputMask="###.###.###.###" />
```

</td><td>

Use the `config.inputMask` property of the `qp-mask-editor` control. However, by default, the `inputMask` property is assigned the value from the `PXUIField` attribute on the corresponding DAC field, so you do not need to specify it.

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

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to mask editor. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXMaskEdit`	• `ID` • `runat`

Menu

In this chapter, you'll learn about the configuration of the menu control, including how to configure menu commands and event handlers.

Menu: General Information

A menu is a drop-down control that provides a list of commands.

A menu is defined by the `qp-menu` tag in the Modern UI. (The control is not supported by the Classic UI.)

Learning Objectives

In this chapter, you'll learn the following about the menu:

- Its design guidelines, including naming conventions and layout recommendations
- The configuration of the menu commands (options)
- The proper configuration of the menu for specific cases, such as the implementation of an event handler for the `menuselected` event

Applicable Scenarios

- You want to add a drop-down menu to a specific area of a form, such as `qp-template` or a table toolbar
- You want to add a menu to a tile of the data feed control

Overview of the Menu Control

The menu control consists of a button that opens the menu and the drop-down menu. The drop-down menu contains menu commands. An example of an opened menu control is shown below.

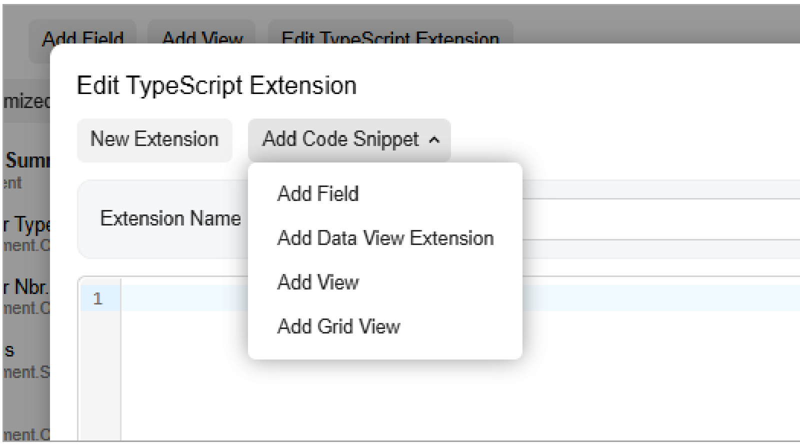


Figure: The menu control in a dialog box

You can place the menu control anywhere on a form, such as a tab toolbar, dialog box, or tile of a data feed control.

To define the menu control:

1. In the TypeScript file, define the configuration of the control. For details, see*Menu: Configuration of the Menu Control*.

2. Optional: Define the `menuSelected` event handler. For details, see *Menu: Handling the MenuSelected Event*.
3. In the HTML template, add the `qp-menu` tag in the needed location. In the `qp-menu` tag, specify the configuration and the `menuSelected` event handler.

Menu ID

A menu's ID consists of two parts: the `menu` prefix and the semantic name., which describes the menu's purpose. For example, a menu that contains file options may have the `menuFileOptions` ID, as the following code shows.

```
@observable fileMenuConfig:IMenuControlConfig = {
  id: "menuFileOptions,
  ...
}
```

A menu item's ID consists of two parts: the semantic name and the `Action` postfix. The semantic name should be identical to the graph action's name. For example, a menu item that adds a field may have `addFieldAction` ID, as the following code shows.

```
options: [
  {
    id: "addFieldAction",
    commandName: "addField",
  },
  ...
]
```

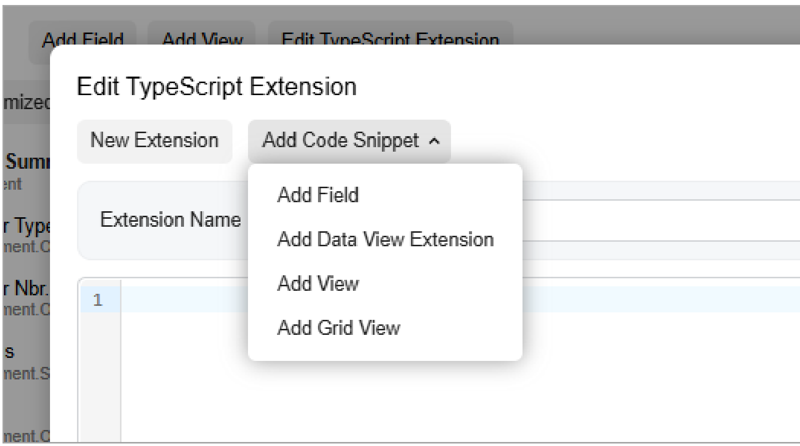
UI Naming Conventions

The following table shows the UI naming conventions for menu commands.

Naming Convention	Example
Use a verb or verb phrase that describes the process that's initiated when a user clicks the command. Use title-style capitalization for command names.	The tile menu on the Company Profile (SP201000) form of the Self-Service Portal

Recommendations for Organizing the Layout

When you want the `qp-menu` control to be displayed as a button, use the `qp-button` class. The button that opens the menu will be displayed on a gray background. Below you can see an example of the **Add Code Snippet** button, which is a `qp-menu` control.



When the `qp-menu` control is displayed on a tile of a data feed control, follow the *layout recommendations for the data feed control*.

Related Links

- *Data Feed*
- *Button*

Menu: Configuration of the Menu Control

The configuration of the `qp-menu` control is stored in the `config: IMenuControlConfig` property and includes these properties:

- `id`: The identifier of the `qp-menu` control
- `text`: The text displayed on the button that a user clicks to open the menu
- `options`: The list of menu commands

To configure the `qp-menu` control, you perform these general steps, described in more detail below:

1. In TypeScript, define the configuration. You can do this either statically in a property of the `IMenuControlConfig` type or in a method if the set of options depends on some logic.
2. In the HTML template, specify the property or the method in the `config.bind` property of the `qp-menu` control.

Defining the Configuration Statically

To define a static set of options:

1. In the screen class, define a property of the `IMenuControlConfig` type.
2. In the property, define the following properties:
   - `id`: The string identifier of the menu.
   - `text`: The text displayed on the button that opens the menu.
   - `options`: The set of available commands on the menu. Options are defined as an array of objects of the `IMenuItem` type.
3. For each option, specify these properties:
   - `id`: The string identifier the menu command
   - `text`: The text displayed as the command
   - `type`: The type of the menu command

- `commandName`: The string name of the graph action to be executed

The following example shows a configuration definition.

```
@observable extensionsMenuConfig:IMenuControlConfig = {
 id: "addSnippetActionsMenu",
 text: Names.AddCodeSnippet,
 options: [
  {
   id: "addFieldAction",
   text: Names.AddField,
   type: "MenuButton",
   commandName: "addFieldSnippet",
  }, {
   id: "addBaseViewAction",
   text: Names.AddDataViewExtension,
   type: "MenuButton",
   commandName: "addBaseViewSnippet",
  }, {
   id: "addViewAction",
   text: Names.AddView,
   type: "MenuButton",
   commandName: "addViewSnippet",
  }, {
   id: "addGridViewAction",
   text: Names.AddGridView,
   type: "MenuButton",
   commandName: "addGridViewSnippet",
  }
 ]
};
```

After you've defined the configuration, you can bind it to the `config` property of the `qp-menu` control in the HTML template, as shown below.

```
<qp-menu class="qp-button" hidden.bind="isReadonly"
  config.bind="extensionsMenuConfig" ...></qp-menu>
```

Defining the Configuration Dynamically

If the set of options depends on a condition or other logic, you can define the configuration and populate the option array in a method. To define the configuration method:

1. In the screen class, declare a method.
2. In the method, define the option array and populate it with objects of the `IMenuItem` type by using the `push` method, as shown in the following example.

```
getCustomizationTreeNodeMenuConfig() {
  const options = [];
  options.push({
    type: "MenuButton",
    text: "",
    commandName: "edit",
    images: { normal: "svg:main@edit" }
  });
  ...
```

```
}
```

3. In the method, return an `IMenuControlConfig` object with the list of options.

The following example shows a configuration method.

```
getCustomizationTreeNodeMenuConfig(field: ICustomizationFieldListItem) {
  const options = [];
  if (field.deletable) {
    // Add the Remove command
    options.push({
    type: "MenuButton",
    text: "Names.RemoveCustomField",
    commandName: "remove",
    images: { normal: "svg:main@delete" } });
  }
  // Add the Edit command
  options.push({
    type: "MenuButton",
    text: "Names.EditInEditor",
    commandName: "edit",
    images: { normal: "svg:main@edit" }
  });
  return {
    ...this.customizationTreeNodeMenuConfig,
    id: `customization_editor_node_menu_${field.id}`,
    options: options,
  };
}
```

The method above adds two menu commands to the option array. Then in the return statement, it destructures the existing static configuration stored in the `customizationTreeNodeMenuConfig` property, and adds the `id` and the `options` properties to it. For details about destructuring, see *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring*.

After you've defined the configuration method, you can bind it to the `config` property of the `qp-menu` control in the HTML template. When specifying the name of the method in the `config` property, you can provide parameter values, as shown below.

```
<qp-menu config.bind="getCustomizationTreeNodeMenuConfig(childField)" ...>
</qp-menu>
```

Menu: Handling the MenuSelected Event

The `qp-menu` control fires a specific event, `menuSelected`, when a user clicks a menu command. You can implement a custom handler for this event and specify it for your `qp-menu` control. In the handler, you can invoke the action corresponding to the menu command that was clicked. You can also execute a method in the menu asynchronously.

Implementing the menuSelected Event Handler

To implement a handler, in your screen's TS file, declare a method with the following signature.

```
handlerName (event: CustomEvent)
```

You can declare additional parameters, such as `config: IEditableElementView`, to the handler.

In the `event: CustomEvent` parameter, you can access the information about the menu command that has been clicked: The `event.detail.id` and `event.detail.commandName` properties hold the identifier and the name of the command that has been clicked.

The following example shows an event handler.

```
processCustomizationTreeNodeMenu(e: CustomEvent,
                                 config: IEditableElementView) {
  const fieldId = config.id;

  switch (e.detail.commandName) {
    case "edit":
      this.editInEditor(fieldId);
      break;
    case "remove":
      this.removeCustomField(fieldId);
      break;
  }
}
```

The code above executes a method corresponding to the menu command (`e.detail.CommandName`) that has been clicked and provides a value as a parameter.

Specifying the Handler for the Control

To define the handler for the `menuSelected` event in the HTML template, you need to:

- Specify the name of the handler in the `menuselected.delegate` property of the `qp-menu` control in the HTML template.
- Provide any needed parameter values. For the parameter of the `CustomEvent` type, specify `$event`.

The following code shows an example of binding an event handler to the `qp-menu` control.

```
<qp-menu config.bind="getCustomizationTreeNodeMenuConfig(childField)"
 class="customization-tree__field-menu"
 menuselected.delegate="processCustomizationTreeNodeMenu($event,
        { id: childField.id })">
</qp-menu>
```

Defining an Asynchronous Event Handler

You can implement a handler to be executed asynchronously. You should declare an asynchronous event handler if the code in the handler can perform a request to the server—for example, if a graph action is executed. To define an asynchronous handler:

1. Add the `async` keyword to the handler declaration.
2. In the handler body, define an `await` expression that executes the method that requests the server.

The following example shows an asynchronous handler. In the handler, you execute an action whose name is identical to the `commandName` property of the menu command that has been clicked.

```
async processSnippetMenuSelected(e: CustomEvent) {
  await this.screenService.executeCommand(e.detail.commandName);
```

```
}
```

Radio Button (Option Button)

In this chapter, you will learn about the configuration of radio buttons (also known as *option buttons*). You'll learn when to use radio buttons and how to organize a layout that includes them.

Radio Button: General Information

A radio button is a small circle that a user can click to turn on or turn off an action, as shown in the following screenshot. Radio buttons are generally organized in groups in which a user is permitted to select only one of the radio buttons.

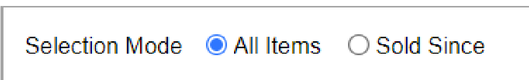


Figure: A group of radio buttons

A group of radio buttons is defined by `PXGroupBox` with nested `PXRadioButton` elements in the Classic UI. In the Modern UI, you define a group of radio buttons by using the `field` tag with the control type specified in TypeScript or explicitly by using the `qp-radio` control.

Learning Objectives

In this chapter, you will learn the following about radio buttons:

- The design guidelines for radio buttons, including the naming conventions and layout recommendations
- The proper configuration of radio buttons for specific cases, such as including an additional box next to a radio button option

Applicable Scenarios

You configure radio buttons when a user needs to select one of the buttons from a set of mutually exclusive options and you want all options to be visible at once.

UI Naming Conventions

The following table shows the UI naming conventions for radio buttons.

<table>
<tr><th>Naming Convention</th><th>Example</th></tr>
<tr><td>Use either of the following phrases as the names of a group of radio buttons:
Noun phrases.
Use noun phrases if they satisfy the context—the name that precedes the group of radio buttons they follow and the meaning they are intended to convey.
Adverbs or prepositional phrases.
Use adverbs or prepositional phrases when the radio buttons should tell a user when or under what circumstances something should happen. Usually these radio buttons follow the verb phrase used as the name that precedes the group of radio buttons.
The radio button names should be parallel with one another.</td><td>The Equipment Class radio buttons on the Stock Items (IN202500) form, which is shown in the following screenshot</td></tr>
</table>

Recommendations for Organizing the Layout

The following table shows the recommendations for organizing the layout for radio buttons.

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">If there are more than two radio buttons, list them vertically. Do not arrange radio buttons in multiple columns if there are more than two of them.</td></tr>
<tr><td>Figure: A correct layout</td><td>Figure: An incorrect layout</td></tr>
<tr><td colspan="2">If the area where you are going to place the control has enough space, the control is used often (such as in a wizard or a dialog box), and the list of options contains no more than five options, use radio buttons instead of a combo box to avoid a user spending extra time on opening the option list in the combo box, scanning the list, and selecting an option.</td></tr>
</table>

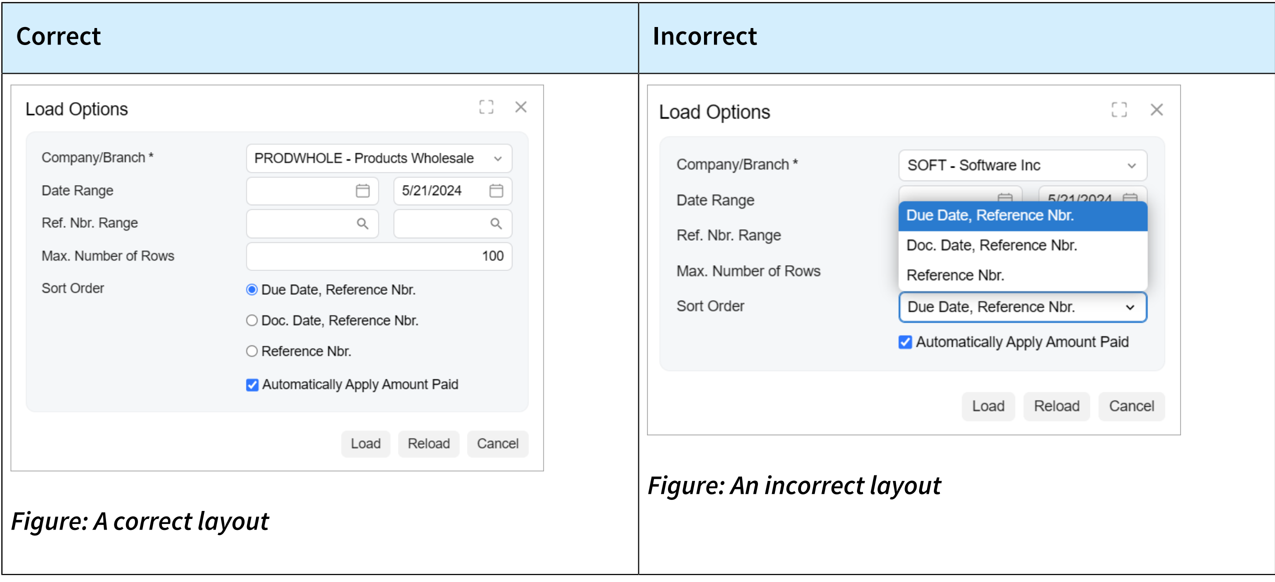


Figure: A correct layout

Figure: An incorrect layout

Radio Button: Configuration

In this topic, you will learn how to configure radio buttons for various layouts.

Vertical Group of Radio Buttons

A simple group of radio buttons is defined in HTML with a `field` tag. You need to specify the `qp-radio` control type in the configuration of the control. By default, radio buttons are rendered horizontally.

In the following example, the radio buttons are configured to be rendered vertically.

```
<field name="PriceBasis" caption="Price Basis"
  control-type="qp-radio"
  config-class.bind="'vertical'">
</field>
```

The following screenshot shows the resulting layout.

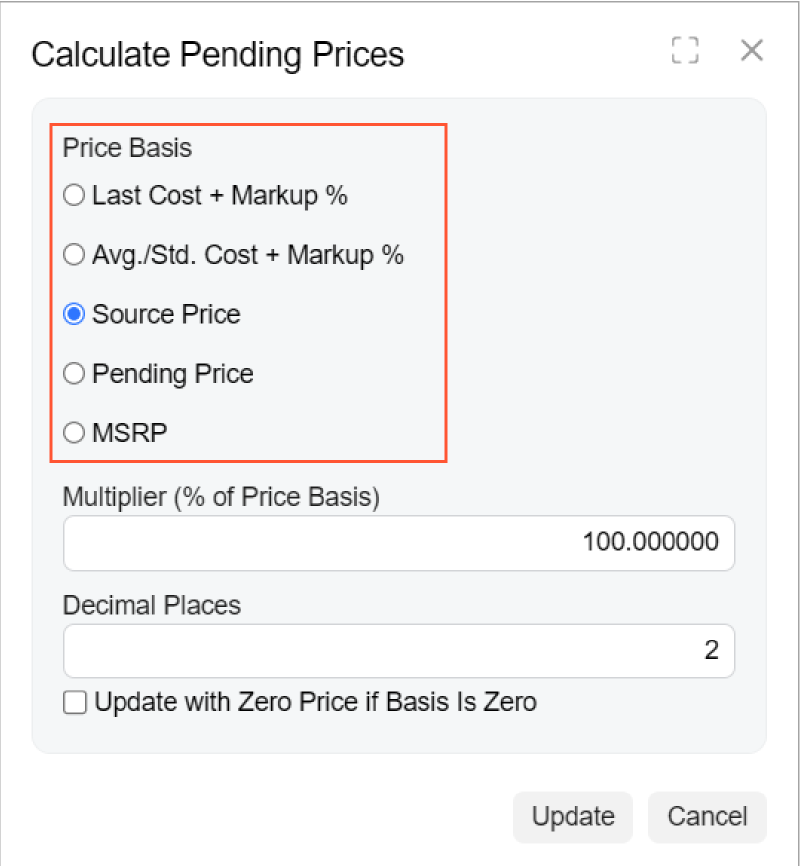


Figure: Vertical group of radio buttons

Group of Radio Buttons with a Box Next to a Radio Button

In some cases, you might need the user to provide additional information if a particular radio button is selected. You can do this by placing a box right of the name of the radio button.

To insert a box next to a radio button, you perform the following steps:

1. For the `field` tag that corresponds to a radio button, you add the `replace-content` attribute.
2. Inside the `field` tag, you do the following:
   a. You insert a `qp-field` tag that has the `qp-radio` control type and is bound to the needed field. You also specify the name of the control in the `name` property of the `config` attribute.
   b. You add a `qp-radio-item` tag for the radio button for which you need to add a box in the same row. For the tag, you specify the following attributes:
      - `name`: The same name that you have specified for the `qp-field` tag with the `qp-radio` control type
      - `value`: The value that corresponds to the radio button in the database
   c. You add a control, such as `qp-field`, for the box that you need to add in the same row.

The following example implements this approach.

```
<field name="ScheduleOption" replace-content>
  <qp-field
    control-type="qp-radio"
    control-state.bind="DeferredCode.ScheduleOption"
    config-name.bind="'ScheduleOption'"
```

```
    config-class.bind="'vertical'"
  >
    <span class="col-6"></span>
    <qp-radio-item name="ScheduleOption" value="D" class="col-6">
    </qp-radio-item>
    <qp-field control-state.bind="DeferredCode.FixedDay" class="col-6">
    </qp-field>
  </qp-field>
</field>
```

The code above results in the layout shown in the following screenshot.

Schedule Options

On Start of Financial Period

On End of Financial Period

On Fixed Day of Financial Period 1

Figure: A box next to a radio button

Radio Button: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to radio buttons to HTML or TypeScript elements.

PXGroupBox

The following table shows the correspondence between the `PXGroupBox` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXGroupBox`

```
<px:PXGroupBox
  CommitChanges="True"
  RenderStyle="Simple"
  ID="gpMode"
  runat="server"
  Caption="Selected Mode"
  DataField="Mode"
  Width="280px"
  Height="25px">
  <Template>
    <px:PXLayoutRule
      runat="server"
      Merge="True"
      LabelsWidth="S"
      ControlSize="S" />
    <px:PXRadioButton
      runat="server"
      ID="rModeSite"
      Value="0"
      Text="By Site State" />
    <px:PXRadioButton
      runat="server"
      ID="rModeCustomer"
      Value="1"
      Text="By Last Sale" />
    <px:PXDateTimeEdit
      CommitChanges="True"
      ID="edHistoryDate"
      runat="server"
      DataField="HistoryDate"
      SuppressLabel="true" />
  </Template>
</px:PXGroupBox>
```

</td><td>

Use the `qp-radio` and `qp-radio-item` tags in HTML instead. For details about this approach, see *Radio Button: Configuration*.

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

`Caption`

```
<px:PXGroupBox
  Caption="Selected Mode" />
```

</td><td>

Specify the `DisplayName` property in the *`PXUIField`* attribute of the respective DAC field.

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

ASPX	HTML or TypeScript

CommitChanges
 	
<px:PXGroupBox
  CommitChanges="True"/>
	Use the CommitChanges option of PXFieldState in TypeScript, as the following code shows. With this option, the control commits changes to the server each time a user has changed the value in the box and focus is no longer on the box.
	
	export class Document
	  extends PXView {
	  Mode:
	    PXFieldState<
	      PXFieldOptions.CommitChanges>;
	}

DataField <px:PXGroupBox DataField="Mode"/>	Use the name attribute of the field tag or the control-state attribute of the qp-field tag in HTML to bind the control to a data field.
ID <px:PXGroupBox ID="gpMode"/>	Use the id attribute of the qp-field tag if this tag is used as a replacement. For the field tag, the ID is not used.

PXRadioButton

The following table shows the correspondence between the PXRadioButton element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

PXRadioButton
 	
<px:PXGroupBox ...>
  <Template>
    <px:PXLayoutRule .../>
    <px:PXRadioButton
      runat="server"
      ID="rModeSite"
      Value="0"
      Text="By Site State" />
    <px:PXRadioButton .../>
    <px:PXDateTimeEdit .../>
  </Template>
</px:PXGroupBox>
	Use the qp-radio-item tag or the options property of the config attribute of the qp-radio tag, as shown in the following code, in HTML instead. For details about the approach, see *Radio Button: Configuration*.
	
	<qp-field
	  control-state.bind=
	    "QuickProcessParameters.ShipDateMode"
	  control-type="qp-radio"
	  config-options.bind=
	    "[{ value: 0, text: 'Today' },
	    { value: 1, text: 'Tomorrow' },
	    { value: 2, text: 'Custom' }]">
	</qp-field>

Text <px:PXRadioButton Text="By Site State" />	Use the text property inside the options property of the config attribute, or use the text attribute of the qp-radio-item tag in HTML.

ASPX	HTML or TypeScript

`Value`
`<px:PXRadioButton`
`  Value="0"/>`
	Use the `value` property inside the `options` property of the `config` attribute, or use the `text` attribute of the `qp-radio-item` tag in HTML.

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to radio buttons. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXGroupBox`	• `Height` • `RenderStyle` • `runat` • `ValidateRequestMode` • `Width`
`PXRadioButton`	• `runat` • `ID`

Record Title

In this chapter, you will learn about the configuration of a record title—that is, a short description of the current record that is displayed under the name of an Acumatica ERP form. You will learn when to use record titles and how to include field values in them.

Record Title: General Information

In Acumatica ERP, data entry and maintenance forms can have a record title, which is a short description of the current record that is displayed under the form name on the form title bar. The record title usually includes the record's ID along with the its name, its description, or additional information about the record.

The record title is also called the *header description*. In this topic, for simplicity, it will be further referred to as the *record title*.

The following screenshot shows an example of the record title for the *Shipments* (SO302000) form. Notice that the record title shows both the shipment number and the customer name.

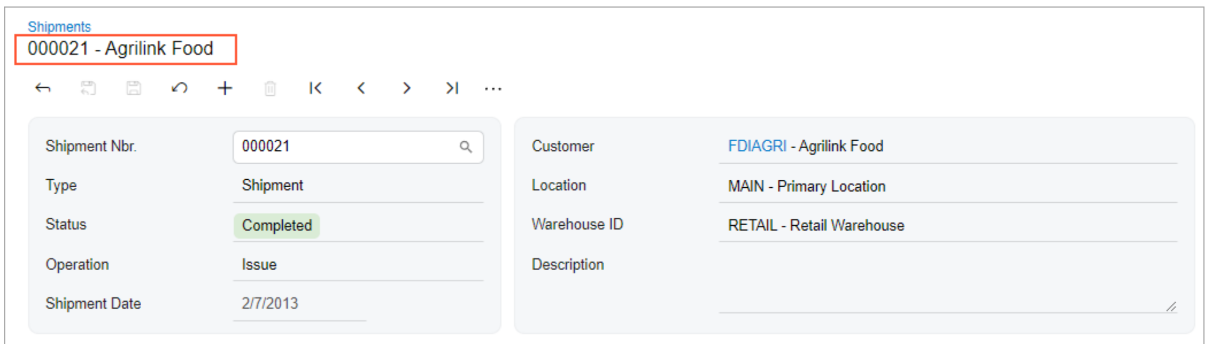


Figure: The record title of the Shipments form

Learning Objectives

In this chapter, you will learn the following:

- The design guidelines for a record title, including the naming conventions and layout recommendations
- The proper configuration of a record title

Applicable Scenarios

You configure a record title if you want to display a short description of the current record on a data entry or maintenance form.

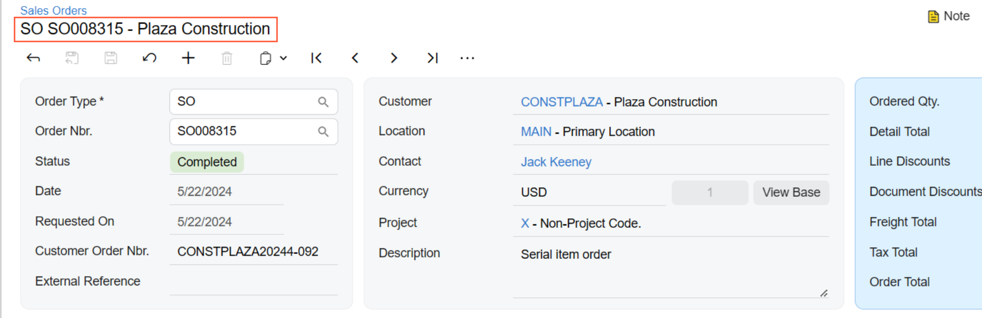
UI Naming Conventions

The following table shows the UI naming conventions for record titles.

<table>
<tr><th>Naming Convention</th><th>Example</th></tr>
<tr><td>
For a data entry form that is used to enter documents, include the following information:

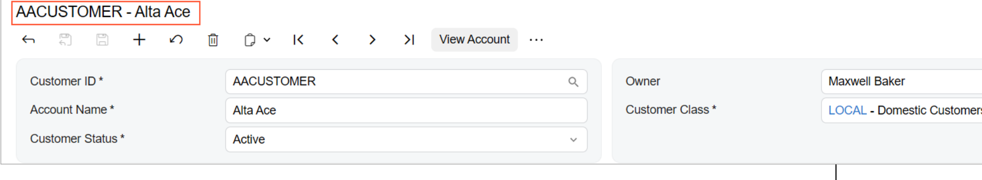
- The document identifier.

  For documents that have multiple key fields (such as sales orders or invoices), you need to include all key fields.
- The account name of the customer or vendor account related to the document.
</td><td>
The record title on the Sales Orders (SO301000) form, which is shown in the following screenshot. On this form, the record title displays the key fields of the sales order (its order type and number) and the customer name.



</td></tr>
<tr><td>
For a maintenance or data entry form that is used to enter maintenance records or profiles (such as inventory items, customers, and contacts) that are used in data entry records, include the following information:

- The record identifier
- The record name
</td><td>
The record title on the Customers (AR303000) form, which is shown in the following screenshot. On this form, the record title displays the identifier of the customer record and the customer name.

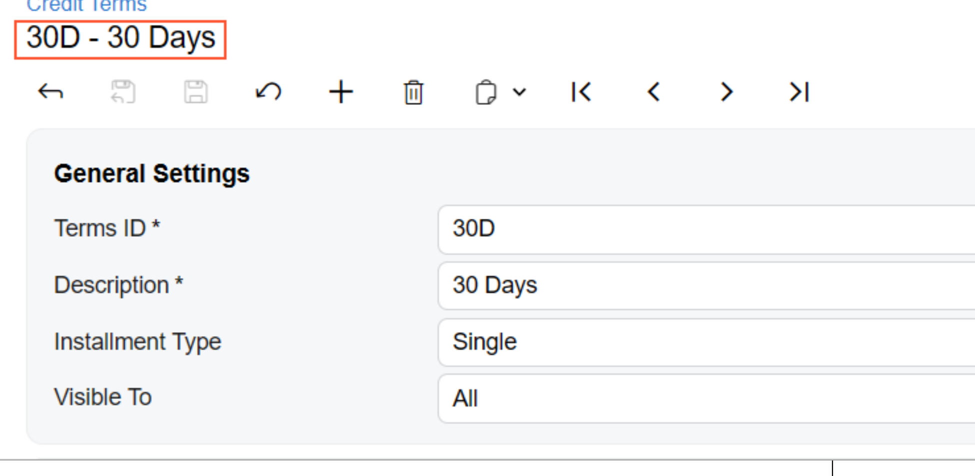


</td></tr>
<tr><td>
For other data entry or maintenance forms, include the following information:

- The record's identifier.

  For records that have compound identifiers, you need to include all parts of the identifier.
- The record's description.

  If the record does not have a description, you can use any other field that can help a user to identify the record.
</td><td>
The record title on the Credit Terms (CS206500) form, which is shown in the following screenshot. On this form, the record title displays the identifier of the credit term record and its description.



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

Record Title: Configuration

For data entry and maintenance forms, the record title includes the values of key fields by default. If you need to adjust the automatically defined record title, you can use one of the following approaches:

- Use the *headerDescription* decorator (recommended)
- Implement the *ICaptionable* interface in the graph of the form

Using the headerDescription Decorator

To define a record title, in the view class that corresponds to the primary view of the graph, you need to add the *headerDescription* decorator to every field whose value should be specified in the record title. For details on defining view classes, see *View Classes in TypeScript*.

The decorator is applicable to any field in the primary view.

For example, the record title for the *Shipments* (SO302000) form is composed of the following values: **Shipment Nbr.** (the `ShipmentNbr` field, which is the key field of the form) and **Customer** (the `CustomerID` field). To define the record title, the `headerDescription` decorator is added to the `CustomerID` field in the definition of the view class, as shown in the following code.

```
export class SOShipmentHeader extends PXView {
  ShipmentNbr: PXFieldState;

  @headerDescription
  CustomerID: PXFieldState<PXFieldOptions.CommitChanges>;

  ...
}
```

You do not need to add the `headerDescription` decorator to the key field (`ShipmentNbr` in the example above) because the key fields are added to the record title by default.

Depending on which fields are marked with the `headerDescription` decorator, the record title is composed according to the following rules:

- If no field is marked with the `headerDescription` decorator, the record title is received from the server. If the server does not send the record title or sends information that it should be empty, the record title is composed of the set of key fields of the view and the description of the last key field if this description exists.
- If only the non-key fields are marked with the `headerDescription` decorator, the record title is composed of the key fields of the view and the description of the fields marked with the `headerDescription` decorator.
- If only the key fields are marked with the `headerDescription` decorator, the record title is composed of these key fields and the description of the last key field, regardless of whether it is marked with the `headerDescription` decorator.
- If both the key fields and the non-key fields are marked with the `headerDescription` decorator, the record title is composed based on these fields.

You can specify how a field is shown in the record title by specifying the parameter of the `headerDescription` decorator on this field. For details, see *Enumeration HeaderDescription*.

Resizer

In this chapter, you will learn about the configuration of a resizer control, including when and how to use a resizer.

Resizer: General Information

A resizer is a control that gives you the ability to adjust the height of a control that is located inside the resizer. You can put any control whose height can be changed into a resizer, such as a table or a tree. If a user adjusts the height of the control, the system stores the adjusted height in the user's preferences. When the user opens the form at any later time, the control will be shown with the adjusted height.

The control looks like a line with three dots in the middle (see the first screenshot below). The line is located under the control that has been placed inside the resizer. When a user hovers over the line with three dots (see the second screenshot below), the mouse pointer turns into a double-headed arrow with two lines in the middle, which means that the lines can be moved.

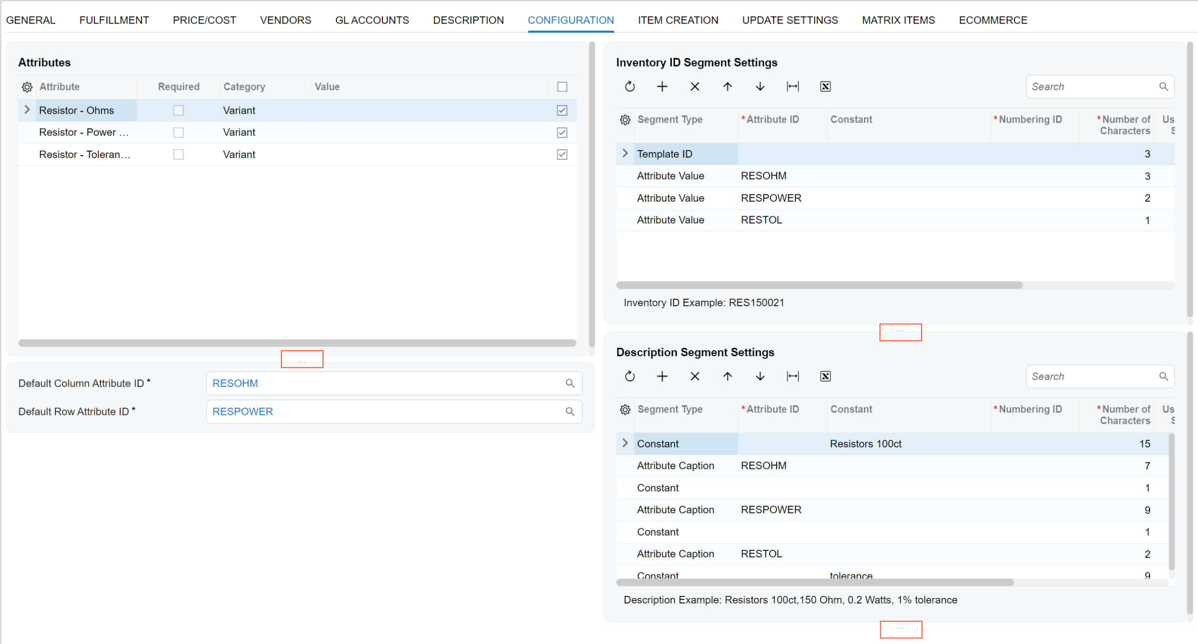


Figure: The three dots on a resizer

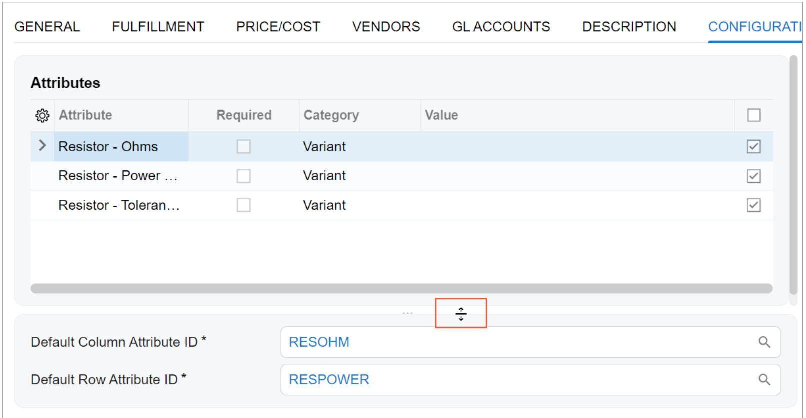


Figure: Changing of the height of the control by using a resizer

Learning Objectives

In this chapter, you will learn the following about the resizer:

- The design guidelines for the resizer, including the naming conventions and layout recommendations
- The proper configuration of the resizer

Applicable Scenarios

You configure a resizer when you want to give a user the ability to change the height of a control manually.

Resizer Overview

The resizer control is defined by the `qp-resizer` tag in HTML.

To add a control inside a resizer, you place the tag for the control inside the `qp-resizer` tag, as shown in the following code. You can specify the initial height of the control in the resizer by using the `initialSize` attribute of the `qp-resizer` tag.

```
<qp-tab id="tab-UpdateExcludeSettings" caption="Update Settings">
  <qp-resizer id="resizer-gridFields" initial-size="400px">
    <qp-grid id="gridFields-UpdateExcludeSettings"
      view.bind="FieldsExcludedFromUpdate"
      caption="Fields Excluded from Update"
      class="framed-section">
    </qp-grid>
  </qp-resizer>
  <qp-resizer id="resizer-gridAttributes" initial-size="400px">
    <qp-grid id="gridAttributes-UpdateExcludeSettings"
      view.bind="AttributesExcludedFromUpdate"
      caption="Attributes Excluded from Update"
      class="framed-section">
    </qp-grid>
  </qp-resizer>
</qp-tab>
```

Resizer ID

The ID of a resizer in HTML consists of two parts with a hyphen between them: the `resizer` prefix and the semantic name. The semantic name describes the element that is located inside the resizer. For example, a resizer that adjusts the height of a table with item attributes may have the `resizer-gridAttributes` ID, as the following code shows.

```
<qp-resizer id="resizer-gridAttributes" initial-size="400px">
  <qp-grid id="gridAttributes-UpdateExcludeSettings" ...>
  </qp-grid>
</qp-resizer>
```

Rich Text Editor

In this chapter, you will learn about the configuration of a rich text editor. You will learn when to use a rich text editor and how to organize a layout that includes a rich text editor.

Rich Text Editor: General Information

A rich text editor represents a text box with the formatting toolbar in the UI. (For details about the formatting toolbar, see *Formatting Toolbar*.)

A rich text editor is defined by `PXRichTextEdit` in the Classic UI. In the Modern UI, you define a rich text editor either by using the `field` or `qp-field` tag with the control type specified or explicitly by using the *`qp-rich-text-editor`* control.

Learning Objectives

In this chapter, you will learn the following about a rich text editor:

- The design guidelines for a rich text editor, including the naming conventions and layout recommendations
- The proper configuration of a rich text editor for specific cases, such as when two rich text editors are used on one Acumatica ERP form

Applicable Scenarios

You use a rich text editor on Acumatica ERP forms where users need to create, edit, and format text with various styles and media elements, such as in the following scenarios:

- Composing and formatting emails that may use different fonts, colors, and embedded images
- Formatting project descriptions, updates, and collaborative notes
- Creating product descriptions and reviews with rich formatting

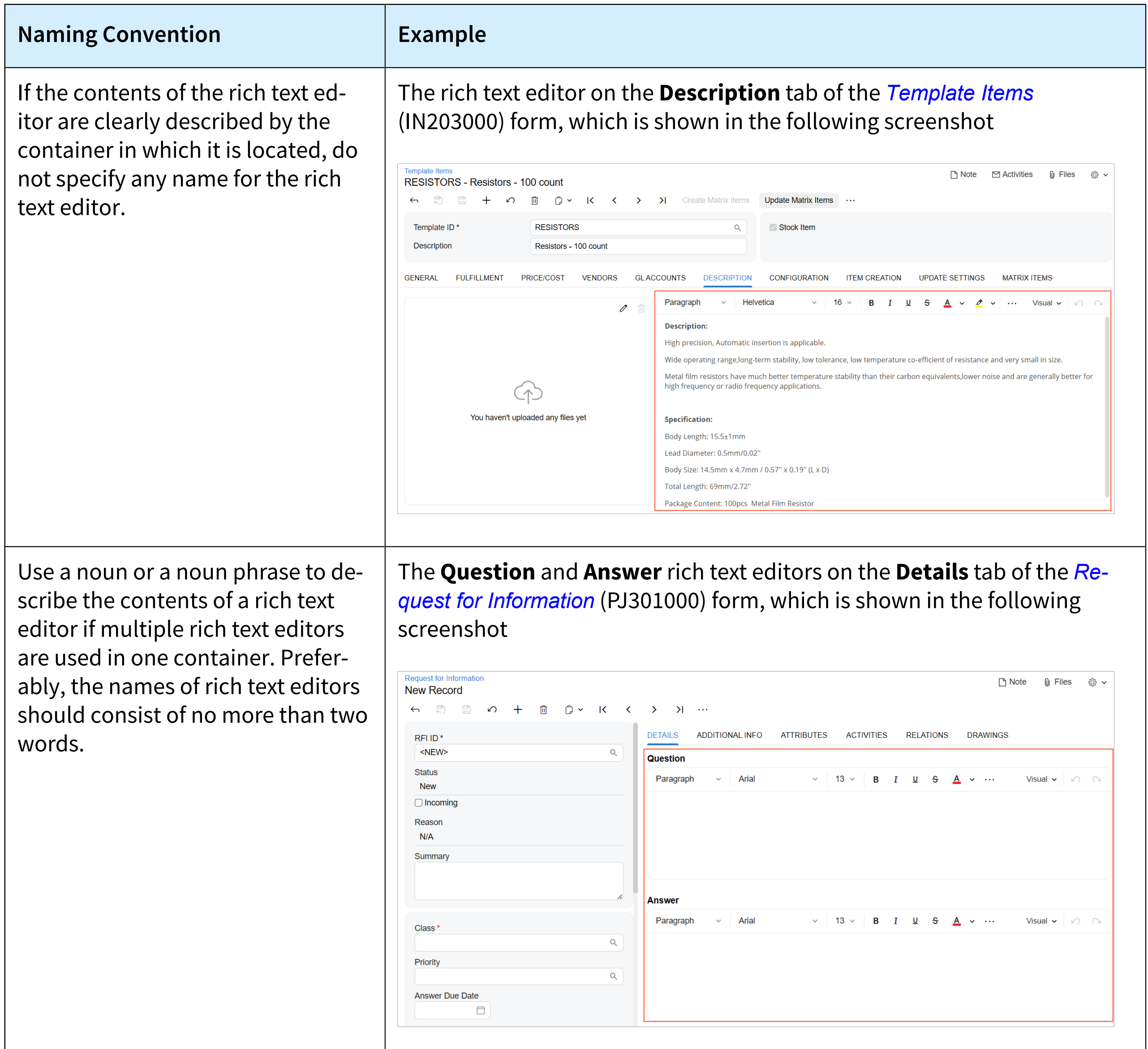
Rich Text Editor ID

The ID of a rich text editor in HTML consists of two parts: the `ed` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a rich text editor where a user provides the project issue description may have the `edProjectIssueDescription` ID, as the following code shows.

```
<qp-rich-text-editor id="edProjectIssueDescription"></qp-rich-text-editor>
```

UI Naming Conventions

The following table shows the UI naming conventions for a rich text editor.



Naming Convention	Example
If the contents of the rich text editor are clearly described by the container in which it is located, do not specify any name for the rich text editor.	The rich text editor on the **Description** tab of the *Template Items* (IN203000) form, which is shown in the following screenshot
Use a noun or a noun phrase to describe the contents of a rich text editor if multiple rich text editors are used in one container. Preferably, the names of rich text editors should consist of no more than two words.	The **Question** and **Answer** rich text editors on the **Details** tab of the *Request for Information* (PJ301000) form, which is shown in the following screenshot

Recommendations for Organizing the Layout

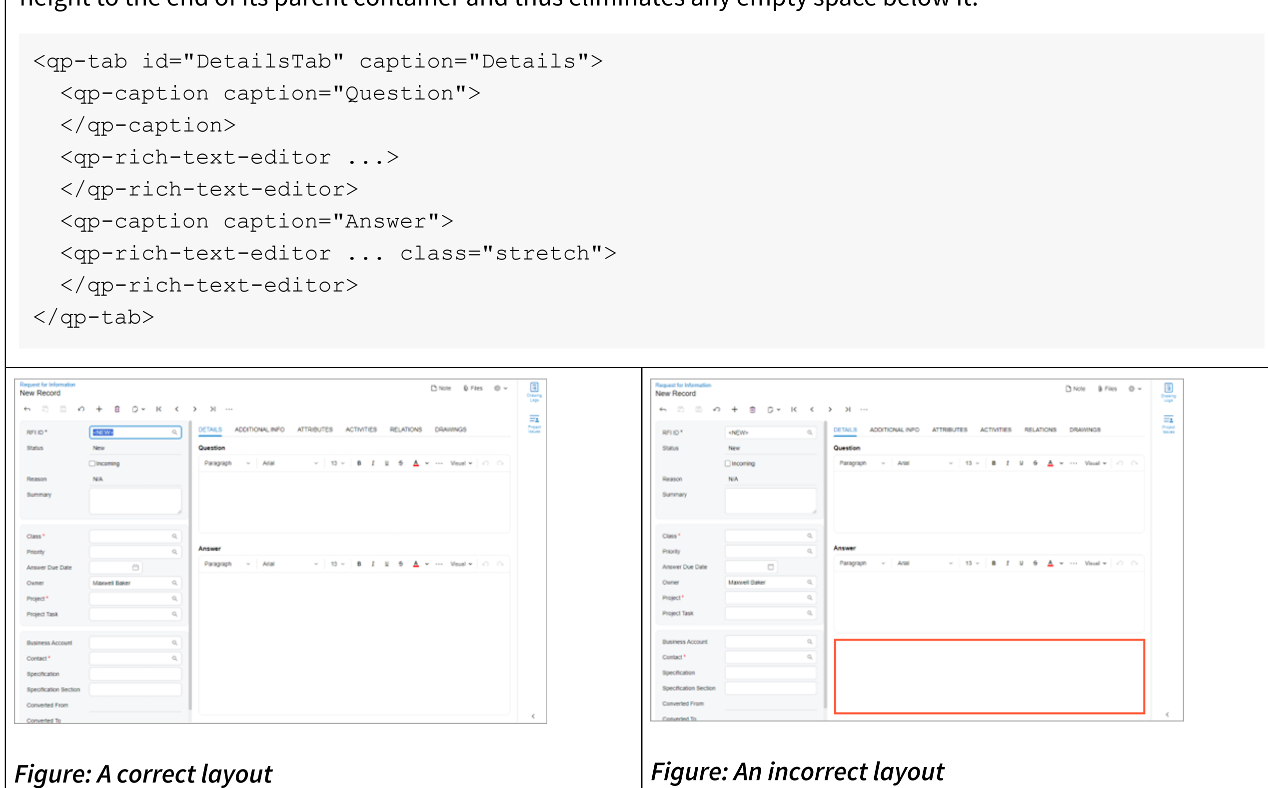
The following table shows the recommendations for organizing the layout for a rich text editor.

Correct	Incorrect

If two rich text editors are displayed on the UI vertically, we recommend that you explicitly stretch at least one of them.

This will prevent empty spaces from being created between or after the rich text editors. Depending on your specific context, you may want to stretch either one or both of the rich text editors.

To stretch the controls, you specify `class="stretch"` in the `qp-rich-text-editor` tag that represents the second rich text editor, as the following example shows. This causes the second rich text editor to expand its height to the end of its parent container and thus eliminates any empty space below it.



```
<qp-tab id="DetailsTab" caption="Details">
  <qp-caption caption="Question">
  </qp-caption>
  <qp-rich-text-editor ...>
  </qp-rich-text-editor>
  <qp-caption caption="Answer">
  <qp-rich-text-editor ... class="stretch">
  </qp-rich-text-editor>
</qp-tab>
```

Figure: A correct layout

Figure: An incorrect layout

Rich Text Editor: Configuration and Layout

In this topic, you can find general information about configuration of the rich text editor control as well as about layout adjustments.

Configuration

To configure a rich text editor, you can use the *`qp-rich-text-editor`* tag, as shown in the following code. The configuration parameters for the control are defined in the *`IRichTextEditorConfig`* interface.

```
<qp-rich-text-editor
  id="edBody"
  state.bind="EstimateRecordSelected.Body"
  class="stretch">
</qp-rich-text-editor>
```

You can also use the `field` or `qp-field` tag with the control type specified instead of the `qp-rich-text-editor` tag.

Addition of Buttons to the Formatting Toolbar

While using a rich text editor, users may need to insert the previous or current value of a data field into the editing area. You can add the **Data Field** and **Previous Data Field** buttons to the formatting toolbar (shown below) to provide these capabilities..

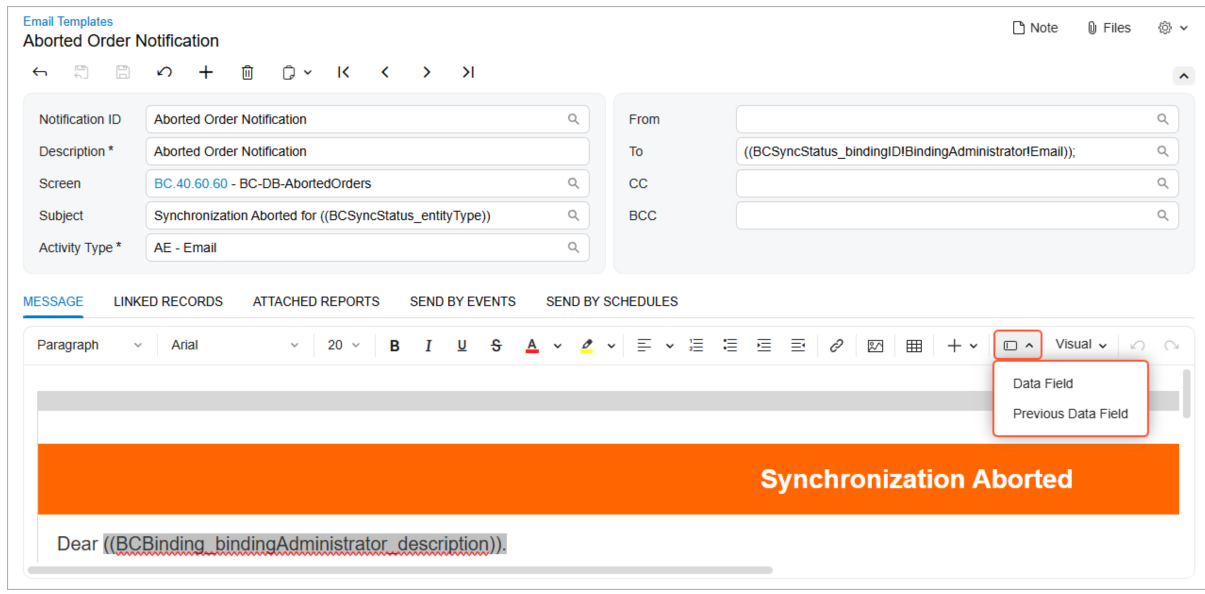


Figure: The Data Field and Previous Data Field buttons

To add these buttons to the formatting toolbar of the `qp-rich-text-editor` control, you use the `insert-data-field` and `insert-data-field-prev` tags. You add these tags within the opening and closing tags of the `qp-rich-text-editor` control, as shown below.

```
<qp-rich-text-editor ...>
  <insert-data-field
    data-member="EntityItems"
    text-field="Name"
    value-field="Path"
    icon-field="Icon">
  </insert-data-field>
  <insert-data-field-prev
    data-member="EntityItemsWithPrevious"
    text-field="Name"
    value-field="Path"
    icon-field="Icon">
  </insert-data-field-prev>
</qp-rich-text-editor>
```

This code adds the **Data Field** and **Previous Data Field** buttons to the formatting toolbar.

When you click either button, the **Select - Data Field** dialog box opens. This dialog box lists the available views and their fields, which you can insert into the editing area.

The list of views in the dialog box depend on the value of the `data-member` property of the `insert-data-field` and `insert-data-field-prev` tags. If you want to make the fields from only a specific view available for selection, you can use the `data-field-preview` tag to specify:

- The name of the view
- The name of the graph in which this view is declared

See the following example.

```
<qp-rich-text-editor ...>
  <insert-data-field
   data-member="EntityItems"
   text-field="Name"
   value-field="Path"
   icon-field="Icon">
  </insert-data-field>
  <data-field-preview
   graph="PX.Objects.CR.ContactMaint"
   view="Contact">
  </data-field-preview>
</qp-rich-text-editor>
```

Based on this code, when you click the **Data Field** button (Item 1 below), the *Contact* view and its fields are displayed in the **Select - Data Field** dialog box (Item 2).

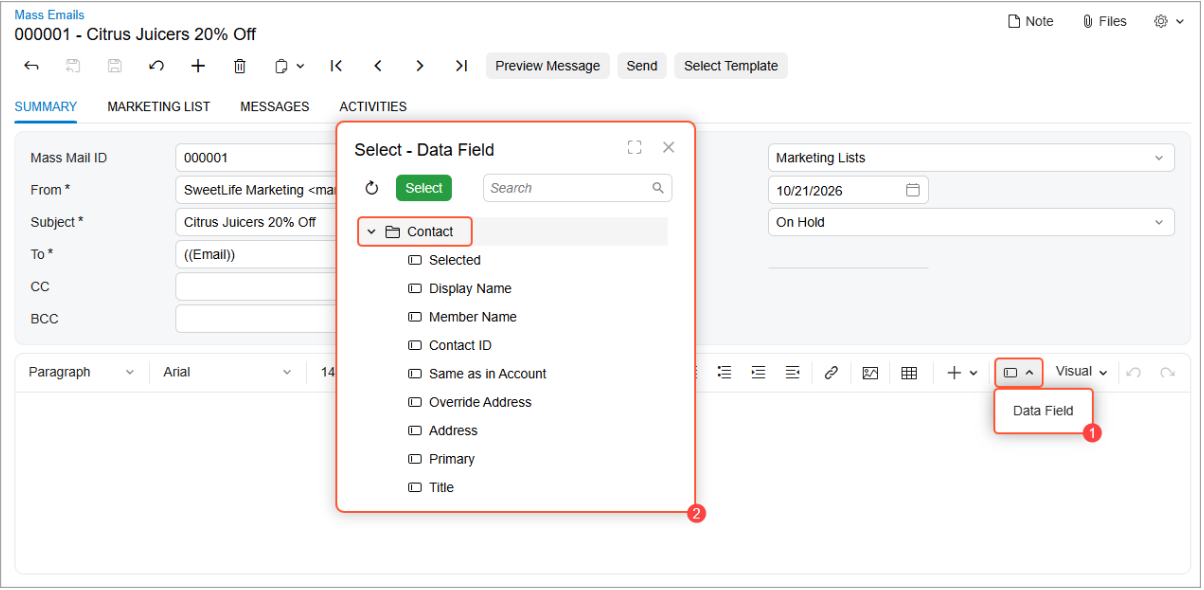


Figure: The Select - Data Field dialog box

Vertical Scroll in a Complex Layout

We do not recommend that you have three vertical scroll lines on a form, such as for a Summary area, grid, and rich text editor.

To avoid three vertical scroll lines, you make the following adjustments to the grid and rich text editor controls:

1. Set the `expandToContent` property to *true* for the rich text editor control.
2. Do not use resize in the grid.
3. For the grid, specify `autoGrowInHeight` equal to `Fit`.
4. For the grid, specify `PageSize` equal to *10*.

As a result, the form will include one scroll for the whole form. The grid will grow in height until it has 10 lines. If it has more than 10 lines, it will have pages.

Rich Text Editor: Conversion from ASPX to HTML and TypeScript

The information in the following tables will help you to convert the ASPX elements that are related to a rich text editor to HTML or TypeScript elements.

AutoSize

The following table shows the correspondence between the AutoSize element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

AutoSize
 	
<px:PXRichTextEdit ...>
  <AutoSize
    Enabled="True"
    MinHeight="216" />
</px:PXRichTextEdit>
	Use the properties in the config attribute of the qp-rich-text-editor control.
	
	<qp-rich-text-editor
	  id="edBody"
	  state.bind="ItemSettings.Body"
	  class="stretch">
	</qp-rich-text-editor>

Enabled
 	
<px:PXRichTextEdit ...>
  <AutoSize
    Enabled="True" ... />
</px:PXRichTextEdit>
	Use the expandToContent property in the config attribute of the qp-rich-text-editor control.
	
	<qp-rich-text-editor
	  config.bind="{expandToContent: true}">
	</qp-rich-text-editor>

MinHeight
 	
<px:PXRichTextEdit ...>
  <AutoSize
    MinHeight="216" ... />
</px:PXRichTextEdit>
	Use the expandToContentMinHeight property in the config attribute of the qp-rich-text-editor control.
	
	<qp-rich-text-editor
	  config.bind=
	    "{expandToContentMinHeight: 216}">
	</qp-rich-text-editor>

InsertDatafield

The following table shows the correspondence between the InsertDatafield element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>
<code>InsertDatafield</code>
<pre>
<px:PXRichTextEdit ...>
  <InsertDatafield
    DataMember="EntityItems"
    DataSourceID="ds"
    TextField="Name"
    ValueField="Path"
    ImageField="Icon" />
</px:PXRichTextEdit>
</pre>
</td><td>
Add the <code>insert-data-field</code> tag between the opening and closing <code>qp-rich-text-editor</code> tag in HTML.
<pre>
<qp-rich-text-editor ...>
  <insert-data-field
    data-member="EntityItems"
    id-field="Key"
    text-field="Name"
    value-field="Path"
    icon-field="Icon">
  </insert-data-field>
</qp-rich-text-editor>
</pre>
</td></tr>
<tr><td>
<code>DataMember</code>
<pre>
<px:PXRichTextEdit ...>
  <InsertDatafield
    DataMember="EntityItems" ... />
</px:PXRichTextEdit>
</pre>
</td><td>
Use the <code>data-member</code> attribute of the <code>insert-data-field</code> tag in HTML. This attribute specifies the data view whose fields should be displayed when the Data Field command is clicked on the toolbar of the <code>qp-rich-text-editor</code> control.
<pre>
<qp-rich-text-editor ...>
  <insert-data-field
    data-member="EntityItems" ...>
  </insert-data-field>
</qp-rich-text-editor>
</pre>
</td></tr>
<tr><td>
<code>DataSourceID</code>
<pre>
<px:PXRichTextEdit ...>
  <InsertDatafield
    DataSourceID="ds" ... />
</px:PXRichTextEdit>
</pre>
</td><td>
Use the <code>id-field</code> attribute of the <code>insert-data-field</code> tag in HTML. This attribute specifies the unique identifier field that belongs to the data view specified in the <code>datamember</code> attribute.
<pre>
<qp-rich-text-editor ...>
  <insert-data-field
    id-field="Key" ...>
  </insert-data-field>
</qp-rich-text-editor>
</pre>
</td></tr>
<tr><td>
<code>TextField</code>
<pre>
<px:PXRichTextEdit ...>
  <InsertDatafield
    TextField="Name" ... />
</px:PXRichTextEdit>
</pre>
</td><td>
Use the <code>text-field</code> attribute of the <code>insert-data-field</code> tag in HTML.
<pre>
<qp-rich-text-editor ...>
  <insert-data-field
    text-field="Name" ...>
  </insert-data-field>
</qp-rich-text-editor>
</pre>
</td></tr>
</table>

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`ValueField`

```
<px:PXRichTextEdit ...>
  <InsertDatafield
    ValueField="Path" ... />
</px:PXRichTextEdit>
```

</td><td>

Use the `value-field` attribute of the `insert-data-field` tag in HTML.

```
<qp-rich-text-editor ...>
  <insert-data-field
    value-field="Path" ...>
  </insert-data-field>
</qp-rich-text-editor>
```

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

`ImageField`

```
<px:PXRichTextEdit ...>
  <InsertDatafield
    ImageField="Icon" ... />
</px:PXRichTextEdit>
```

</td><td>

Use the `icon-field` attribute of the `insert-data-field` tag in HTML.

```
<qp-rich-text-editor ...>
  <insert-data-field
    icon-field="Icon" ...>
  </insert-data-field>
</qp-rich-text-editor>
```

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

InsertDatafieldPrev

The following table shows the correspondence between the `InsertDatafieldPrev` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`InsertDatafieldPrev`

```
<px:PXRichTextEdit ...>
  <InsertDatafieldPrev
    DataMember="PreviousEntityItems"
    DataSourceID="ds"
    TextField="Name"
    ValueField="Path"
    ImageField="Icon" />
</px:PXRichTextEdit>
```

</td><td>

Add the `insert-data-field-prev` tag between the opening and closing `qp-rich-text-editor` tag in HTML.

```
<qp-rich-text-editor ...>
  <insert-data-field-prev
    data-member="EntityItemsWithPrevious"
    id-field="Key"
    text-field="Name"
    value-field="Path"
    icon-field="Icon">
  </insert-data-field-prev>
</qp-rich-text-editor>
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>DataMember</code>
<pre><code><px:PXRichTextEdit ...>
  <InsertDatafieldPrev
    DataMember="PreviousEntityItems" ... />
</px:PXRichTextEdit></code></pre></td>
<td>Use the <code>data-member</code> attribute of the <code>insert-data-field-prev</code> tag in HTML. This attribute specifies the data view whose fields should be displayed when the Previous Data Fieldcommand is clicked on the toolbar of the <code>qp-rich-text-editor</code> control.
<pre><code><qp-rich-text-editor ...>
  <insert-data-field-prev
    data-member="EntityItemsWithPrevious" ...>
  </insert-data-field-prev>
</qp-rich-text-editor></code></pre></td></tr>
<tr><td><code>DataSourceID</code>
<pre><code><px:PXRichTextEdit ...>
  <InsertDatafieldPrev
    DataSourceID="ds" ... />
</px:PXRichTextEdit></code></pre></td>
<td>Use the <code>id-field</code> attribute of the <code>insert-data-field-prev</code> tag in HTML. This attribute specifies the unique identifier field that belongs to the data view specified in the <code>data-member</code> attribute.
<pre><code><qp-rich-text-editor ...>
  <insert-data-field-prev
    id-field="Key" ...>
  </insert-data-field-prev>
</qp-rich-text-editor></code></pre></td></tr>
<tr><td><code>TextField</code>
<pre><code><px:PXRichTextEdit ...>
  <InsertDatafieldPrev
    TextField="Name" ... />
</px:PXRichTextEdit></code></pre></td>
<td>Use the <code>text-field</code> attribute of the <code>insert-data-field-prev</code> tag in HTML.
<pre><code><qp-rich-text-editor ...>
  <insert-data-field-prev
    text-field="Name" ...>
  </insert-data-field-prev>
</qp-rich-text-editor></code></pre></td></tr>
<tr><td><code>ValueField</code>
<pre><code><px:PXRichTextEdit ...>
  <InsertDatafieldPrev
    ValueField="Path" ... />
</px:PXRichTextEdit></code></pre></td>
<td>Use the <code>value-field</code> attribute of the <code>insert-data-field-prev</code> tag in HTML.
<pre><code><qp-rich-text-editor ...>
  <insert-data-field-prev
    value-field="Path" ...>
  </insert-data-field-prev>
</qp-rich-text-editor></code></pre></td></tr>
<tr><td><code>ImageField</code>
<pre><code><px:PXRichTextEdit ...>
  <InsertDatafieldPrev
    ImageField="Icon" ... />
</px:PXRichTextEdit></code></pre></td>
<td>Use the <code>icon-field</code> attribute of the <code>insert-data-field-prev</code> tag in HTML.
<pre><code><qp-rich-text-editor ...>
  <insert-data-field-prev
    icon-field="Icon" ...>
  </insert-data-field-prev>
</qp-rich-text-editor></code></pre></td></tr>
</table>

PXRichTextEdit

The following table shows the correspondence between the `PXRichTextEdit` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

`PXRichTextEdit`
 	
`<px:PXRichTextEdit`
`  ID="edBody"`
`  runat="server"`
`  DataField="Body"`
`  Style="border-width: 0px;"`
`  AllowAttached="true"`
`  AllowSearch="true"`
`  AllowLoadTemplate="false"`
`  AllowSourceMode="true">`
`</px:PXRichTextEdit>`
	Use the `qp-rich-text-editor` tag in HTML.
	
	`<qp-rich-text-editor`
	`  id="edBody"`
	`  state.bind="ItemSettings.Body"`
	`  class="stretch">`
	`</qp-rich-text-editor>`

`ID`
 	
`<px:PXRichTextEdit`
`  ID="edBody">`
`</px:PXRichTextEdit>`
	Use the `id` attribute of the `qp-rich-text-editor` tag in HTML.
	
	`<qp-rich-text-editor`
	`  id="edBody">`
	`</qp-rich-text-editor>`

`DataField`
 	
`<px:PXRichTextEdit`
`  DataField="Body">`
`</px:PXRichTextEdit>`
	Use the `state` attribute of the `qp-rich-text-editor` tag in HTML.
	
	`<qp-rich-text-editor`
	`  state.bind="ItemSettings.Body">`
	`</qp-rich-text-editor>`

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to rich text editors. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXRichTextEdit`	• `AllowAttached` • `AllowLoadTemplate` • `AllowSearch` • `AllowSourceMode` • `runat` • `Style`

Selector

In this chapter, you will learn about the configuration of selector controls, including when to use a selector control and how to name it.

Selector Control: General Information

A selector control that consists of a box to hold a selected value and a lookup table where a user can select a value, as shown in the following screenshot.

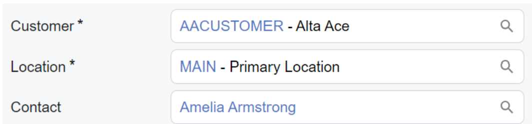


Figure: The Selector controls

A selector control is defined by `PXSelector` in the Classic UI. In the Modern UI, a selector control is defined by the `field` tag (whose control type is automatically defined as a selector from the backend code). In rare cases, a selector control in the Modern UI is defined explicitly by the *`qp-selector`* control.

Learning Objectives

In this chapter, you will learn the following about a selector control:

- The design guidelines for the selector control, including the naming conventions and layout recommendations
- The proper configuration of the selector control for specific cases, such as when it should display a link to a record
- A detailed description of each property of the selector control
- Differences between the selector control in the Classic UI and in the Modern UI

Applicable Scenarios

You configure a selector control in the following cases:

- A user needs to select a record from a list of possible records.
- You need to display a box or a column with a record identifier.
- You need to display the unique identifier of the existing record or the new record.
- A user needs to be able to open the record by clicking on the link.

Differences between the Classic UI and the Modern UI

The following screenshots demonstrate the difference between the `PXSelector` control in the Classic UI and the `qp-selector` control in the Modern UI.

The new selector control does not have the Edit button, which gave users of the Classic UI the ability to open the selected record for editing. Instead, in the Modern UI, the selector control displays a link to the selected record. A user clicks the link to open the record in a new window. For details on configuring a link in the selector control, see *Selector Control: Configuration of a Link*.

To create a new record for a selector control in the Modern UI, a user needs to click the magnifier button and then click + on the toolbar of the lookup table. (In the Classic UI, a user could do this by clicking the Edit button for an empty selector control.)

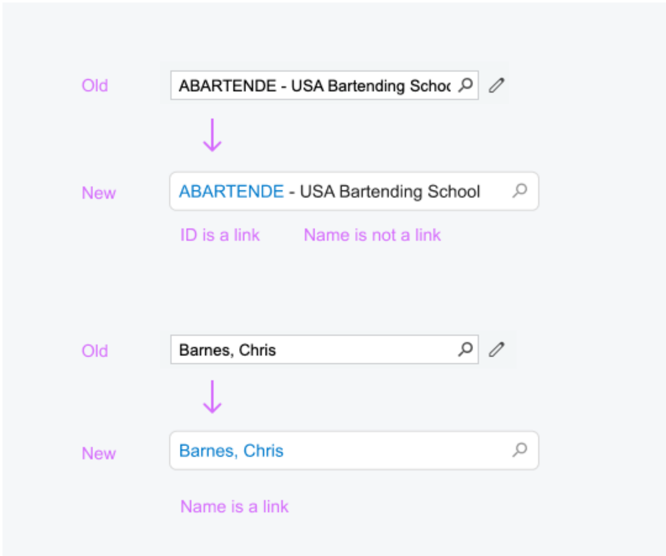
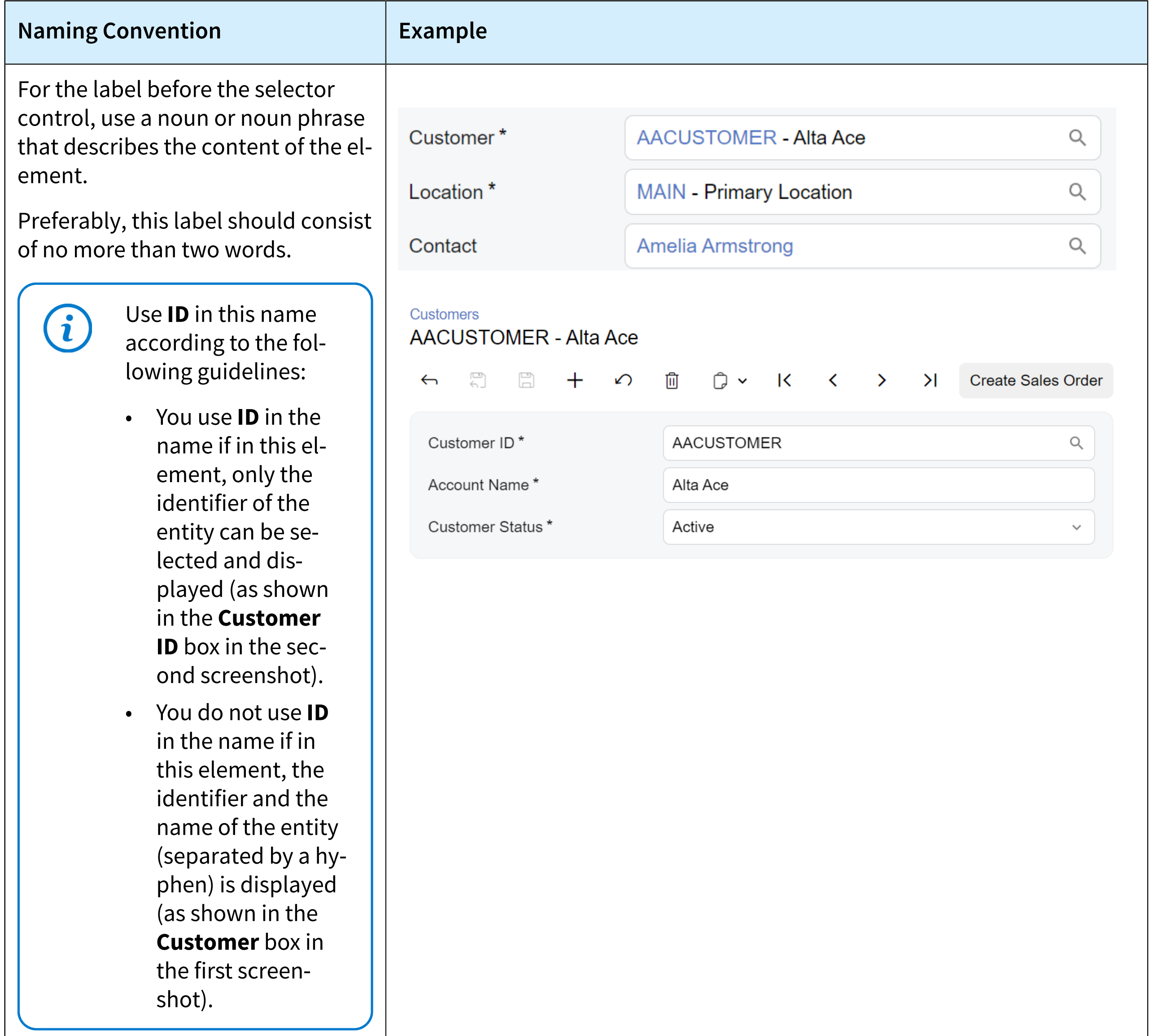


Figure: Differences in the selector control between the Classic UI and the Modern UI

UI Naming Conventions

The following table shows the UI naming conventions for a selector control.



Naming Convention	Example
For the label before the selector control, use a noun or noun phrase that describes the content of the element. Preferably, this label should consist of no more than two words. Use **ID** in this name according to the following guidelines: • You use **ID** in the name if in this element, only the identifier of the entity can be selected and displayed (as shown in the **Customer ID** box in the second screenshot). • You do not use **ID** in the name if in this element, the identifier and the name of the entity (separated by a hyphen) is displayed (as shown in the **Customer** box in the first screenshot).	

Selector Control: Configuration from Backend

You use selector controls to provide a list from which the user can select a data record and then to set the ID of the selected data record as the data field value.

Defining a Selector Control

To configure a selector control, you use the `PXSelector` attribute in the definition of the data field in the data access class (DAC), as shown in bold type in the following example.

```
[PXDBInt(IsKey = true)]
[PXDefault]
[PXUIField(DisplayName = "Product ID")]
[PXSelector(typeof(Search<Product.productID>),
            typeof(Product.productCD),
            typeof(Product.productName),
```

```
            typeof(Product.unitPrice),
            SubstituteKey = typeof(Product.productCD))]
public virtual int? ProductID
...
```

In the first parameter, you specify a `Search<>` BQL query to select data records for the control. The `Search<>` command has the same syntax as the `Select<>` command, except that you specify the data field of the main DAC. In the `Search<>` command, you can specify conditions and join data from other DACs. When a user selects a data record in the control, the control assigns the value of the specified field to the data field.

You can omit `Search<>` in the first parameter of `PXSelector`, if you specify only a DAC field without a complex expression that may contain `WHERE`, `JOIN`, `ORDER BY`, or `GROUP BY` conditions. Thus, in the example above, you can specify `typeof(Product.productID)` instead of `typeof(Search<Product.productID>)` in the first parameter.

Defining the List of Columns

You configure the columns that should be shown in the control by providing the types of the fields after the `Search<>` command; see the code in bold type in the following example.

```
[PXSelector(typeof(Search<Product.productID>),
            typeof(Product.productCD),
            typeof(Product.productName),
            typeof(Product.unitPrice),
            SubstituteKey = typeof(Product.productCD))]
public virtual int? ProductID
...
```

The code above defines three columns.

You can join multiple DACs in the `Search<>` command and specify fields from the joined DACs as columns. The following code example shows in bold type the `LeftJoin` clause and the fields from the joined DAC added to the selector control as columns.

```
[PXDBString(10, IsKey = true, IsUnicode = true, InputMask = "")]
[PXDefault]
[PXUIField(DisplayName = "Shipment Nbr.")]
[PXSelector(typeof(
    Search2<Shipment.shipmentNbr,
        LeftJoin<Customer, On<Customer.customerID,
                              Equal<Shipment.customerID>>>>),
            typeof(Shipment.shipmentNbr),
            typeof(Shipment.customerID),
            typeof(Customer.customerCD),
            typeof(Customer.companyName))]
public virtual string ShipmentNbr
...
```

If you do not specify any columns, the control will display all columns that have the `Visibility` property of the `PXUIField` attribute set to `PXUIVisibility.SelectorVisible`.

Replacing the Displayed Key Value

The `SubstituteKey` property specifies the field whose value should be shown in the control in the UI instead of the field specified in the `Search<>` command.

The `SubstituteKey` property is shown in the bold type in the following code.

```
[PXSelector(typeof(Search<Product.productID>),
            typeof(Product.productCD),
            typeof(Product.productName),
            typeof(Product.minAvailQty),
            SubstituteKey = typeof(Product.productCD))]
public virtual int? ProductID
...
```

In the example above, the `ProductID` field of a shipment line stores the `ProductID` value of the selected product, while in the UI. the control shows the `ProductCD` value. Conversion between the `ProductID` and `ProductCD` values happens in the `FieldUpdating` and `FieldSelecting` event handlers, which are implemented within the `PXSelector` attribute.

Selector Control: Configuration of a Link

A selector control can display a value as a link to the record whose identifier is displayed in the selector control. The link is configured differently depending on the location of the selector control.

Selector Control in a Fieldset

In a fieldset, you add the link by specifying `allowEdit: true` in the `controlConfig` decorator for the field in TypeScript, as shown in the following example.

```
@controlConfig({allowEdit: true, })
CustomerID: PXFieldState<PXFieldOptions.CommitChanges>;

@controlConfig({allowEdit: true, })
CustomerLocationID: PXFieldState;

@controlConfig({allowEdit: true, })
ContactID: PXFieldState;
```

The code above displays links, as shown below.

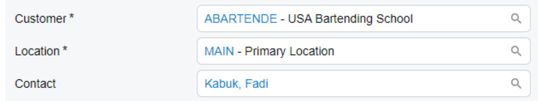


Figure: Links in the selector controls

The `allowEdit: true` setting also adds the **+** (**Add Row**) button to the lookup table of the selector control.

Selector Control in a Table

In a table, a link is displayed by default for a selector control. To remove the link, specify `hideViewLink: true` in the `columnConfig` decorator in TypeScript, as shown in the following example.

```
@columnConfig({ hideViewLink: true })
BranchID: PXFieldState<PXFieldOptions.CommitChanges>;
```

The code above removes the link from the **Branch** column, as shown below.

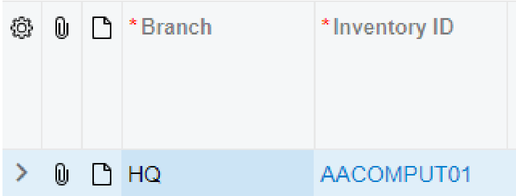


Figure: Link in the grid

Link Behavior

You can configure which action is executed when a user clicks the link in the selector control. By default, the system opens the record that is selected by the user in the selector control. To specify a custom action, you use one of the following:

- For a selector control located in a fieldset, the *`editCommand`* property in the *`controlConfig`* decorator, as shown in the following code

```
export class EPAssignmentMap extends PXView {
  @controlConfig({editCommand: "OpenForm"})
  GraphType: PXFieldState<PXFieldOptions.CommitChanges>;
}
```

- For a selector control located in a grid, the *`linkCommand`* decorator

```
@gridConfig({
 preset: GridPreset.Inquiry
})
export class RSSVWorkOrderToPay extends PXView {
 @linkCommand<RS401000>("ViewOrder")
 OrderNbr: PXFieldState;
}
```

To use a custom action for the link in the selector control, you also need to do the following:

1. In the graph, define an action that opens the entered form with the new record.
2. In the TypeScript file of the form, declare a property of the `PXActionState` type for this action in the screen class.

Selector Control: Configuration of the Control's Text

The text in a selector control is determined by the parameters specified in the PXSelector attribute (or a derived attribute, such as Customer) on the DAC field. By default, the value of the SubstituteKey parameter of the PXSelector attribute is displayed in the selector control.

Adding Additional Text to the Default Text

You can add additional text to the selector element. This text will be shown after the default text, separated by a hyphen. To add the additional text to the text box, specify the field that contains the additional text in the DescriptionField parameter of the PXSelector attribute (or its descendants), as shown in the following code.

```
[PXDefault]
[Customer(
    typeof(Search<BAccountR.bAccountID, Where<True, Equal<True>>>),
    Visibility = PXUIVisibility.SelectorVisible,
    DescriptionField = typeof(Customer.acctName),
    Filterable = true)]
...
public virtual Int32? CustomerID
```

Replacing Default Text

You can replace the identifier from the SubstituteKey parameter with more human-readable text that is specified in the DescriptionField parameter value of the PXSelector attribute. To replace the default text (within the link) with a human-readable field, you can do one of the following:

- In the DAC code, on the DAC field, in the PXSelector attribute, specify SelectorMode = PXSelectorMode.DisplayModeText.
- In the HTML code, in the field tag, specify displayMode = "text" (see the descriptions of other possible values in qp-selector).
- Only for a grid column: In the TypeScript code, in the columnConfig decorator, specify displayMode: GridColumnDisplayMode.Text.

If you need to specify which field should be displayed in the column selector explicitly, you can specify the field in the textField tag, as shown in the following example.

```
@columnConfig({
  hideViewLink: true,
  textField: "ShipmentPlanType_INPlanType_LocalizedDescr" })
```

Entering a Value in the Selector Box

You can configure the selector box so that when a user enters a value by typing characters in the selector box, the entered string is assigned as the value of the field—that is, the valueField property of the control. For this purpose, you need to do the following in the controlConfig decorator:

1. For the displayMode property, specify "text".
2. For the selectorMode property, specify PXSelectorMode.TextModeEditable.

You can use the `selectorMode` property to configure the autocomplete feature, the display mode, and the `"text"` mode of the selector box.

For example, suppose that you need a user to be able to enter the URL of a link manually in a selector column. The following code implements this behavior.

```
@columnConfig({
  displayMode: GridColumnDisplayMode.Both,
  editorConfig: {
    displayMode: "text",
    selectorMode: PXSelectorMode.TextModeEditable
  }
})
Link: PXFieldState<PXFieldOptions.CommitChanges>;
```

Related Links

- *PXSelector*

Selector Control: Configuration of the Lookup Table

You can configure the lookup table that is displayed when a user clicks the magnifier icon in the selector box.

Configuring the Add New Record Button

You can specify which action is executed when a user clicks the **Add New Record** button in the lookup table, as shown in the following screenshot. By default, the system opens the relevant form with an empty record (that is, no values except default ones are filled in). To specify the action, use the `addCommand` property of the `qp-selector config` property.

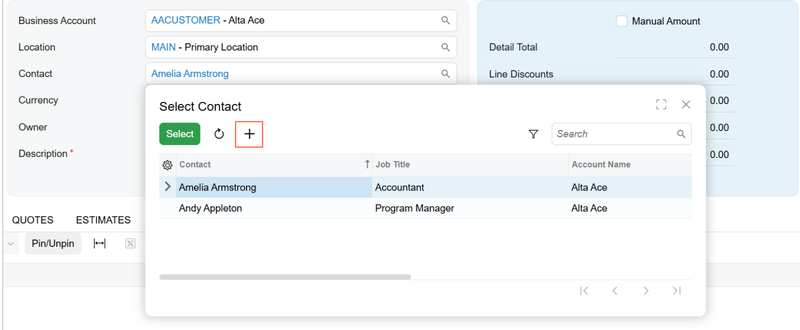


Figure: The lookup table of the selector control

Suppose that when a user clicks the **Add New Record** button in the selector control, the system needs to open a new record with prefilled values that depend on the user's data. For that purpose, you need to do the following:

1. In the backend, define an action that opens a form with a new record and prefills the values depending on the user's data.
2. In the frontend, declare a `PXActionState` property in the screen class for this action.
3. In the `controlConfig` decorator for the form field, specify the action in the `addCommand` property, as shown in the following example.

```
export class CROpportunityHeader extends PXView {
  @controlConfig({addCommand: "CreateNewContact"})
```

```
  ContactID: PXFieldState<PXFieldOptions.CommitChanges>;
}
```

Sorting Rows in the Lookup Table

By default, the rows in the lookup table are sorted as follows:

- If only plain text is displayed in the selector box (`displayMode = "text"`), the lookup table is sorted by the value of the `textField` property. The default value of the `textField` property is the `SubstituteKey` value of the `PXSelector` attribute on the DAC field.
- Otherwise, the sorting is performed by `valueField`. The default value of the `valueField` property is the field in the `Search<>` command of the `PXSelector` attribute on the DAC field.

You can specify the field for sorting rows in the lookup table by specifying the column name in the `textField` property of the `qp-selector config` property. An example is shown in the following code.

```
@columnConfig({
  displayMode: GridColumnDisplayMode.Both,
  editorConfig: {
    textField: "title", displayMode: "text"
  }
})
ScreenID : PXFieldState;
```

For more details on the `valueField` and `textField` properties, see *Selector Control: Manual Configuration of the Value and Text*.

Related Links

- *qp-selector*
- *Interface ISelectorControlConfig*

Selector Control: Manual Configuration of the Value and Text

By default, the values selected and displayed in the selector control are determined by the settings in the `PXSelector` attribute on the DAC field as follows:

- The value selected in the lookup table is determined by the `Search<>` command of the `PXSelector` attribute.
- The text displayed in the selector box is determined by the `SubsituteKey` property of the `PXSelector` attribute.

For more details on using the `PXSelector` attribute, see *Selector Control: Configuration from Backend*.

However, if the DAC field does not have the `PXSelector` attribute for some reason (for example, when a custom field state is created in an event handler of the graph), you can use the following properties of the `qp-selector config` property to specify these settings manually:

- `valueField`: The field whose value is selected in the lookup table
- `textField`: The field whose value is displayed in the selector box after the link text

The following code shows an example from the *Filters* (CS209010) form that illustrates the usage of these properties.

```
export class FilterHeader extends PXView {
  @controlConfig({ valueField: "Name", textField: "DisplayName" })
  ViewName: PXFieldState<PXFieldOptions.CommitChanges>;
```

```
  ...
}
```

These properties also affect how the sorting is performed in the lookup table. For details, see *Sorting Rows in the Lookup Table*.

Related Links

- *Selector Control: Configuration of a Link*
- *Class PXSelectorAttribute*

Selector Control: Multiple-Selection Mode

You can configure a selector control so that a user can select multiple values in it. To configure the multiple-selection mode, in the `controlConfig` decorator of the field that corresponds to the selector control, you specify `multiSelect = true` and an optional template in the `valueTemplate` parameter.

For example, if you want a user to select multiple emails in a selector control, you can use the following code. In the value template below, *{0}* is the value (that is, a email address) and *{1}* is the description text.

```
@controlConfig({ multiSelect: true, valueTemplate: '{1} <{0}>' })
  MailTo: PXFieldState<PXFieldOptions.CommitChanges>;
```

Selector Control: Displaying of the Value in the Selector Control

In some cases, you may want to explicitly specify the control that is used to display the value selected in a selector control. By default, the `qp-text-editor` control is used to display the selected value. To specify a control explicitly, you can use the `editorType` property of the `config` attribute of the selector control, as shown in the following code example.

```
<field name="MailTo" control-type="qp-selector"
  config-editor-type.bind="'qp-mail-editor'">
</field>
```

In the code above, you have specified that the selected value in the selector control should be displayed by using the `qp-mail-editor` control. For details about this control, see *Mail Editor*.

Selector Control: Selector Parameters

You can configure parameters for the selector control. The provided parameter value is used in `PXSelector` attribute on the DAC field.

You specify selector parameters in the `config` property of the `qp-selector` control in one of the following ways:

- If you need to specify dynamic values, use the `parameters` function, which is shown in the following code.

  ```
  parameters(screen: PXScreen): { [k: string]: any };
  ```

  In the `parameters` function, you specify the screen class as an argument so that the validation and calculation can be performed for the provided parameters. The `parameters` function should return the dictionary, where each object is a name and a value of the selector parameter.

To provide the value of the selector parameter, you can refer to the views of the current form by using the `screen` property. To refer to the currently selected record, you can use the `activeRow` property of the view. An example is shown in the following code.

```
parameters: (screen: AP203500) => ({
  "APRegister.docType": screen.Document_Detail.activeRow.DocType.value })
```

In the code above, the parameter name is `APRegister.docType`, and the parameter value is in the `AP203500` screen, `Document_Detail` view, and `DocType` property of the currently selected record.

- If you can specify static values, use the `parameters` property, which is shown below.

```
parameters: { [k: string]: any }
```

As the property value, you specify the dictionary, where each object is a name and a value of the selector parameter.

Example: Defining a Selector Column that Depends on Another Column

Suppose that you need show different records in the `refNbr` selector depending on a document type specified in the `docType` field in the same table row.

To implement this dependency, do the following:

1. In the backend: In the `PXSelector` attribute for the `refNbr` DAC field, specify the `APRegister.docType` parameter, as shown in the following code.

```
[PXSelector(typeof(Search<APRegister.refNbr,
  Where<APRegister.docType, Equal<Optional<APRegister.docType>>>>),
  Filterable = true)]
public virtual String RefNbr {...}
```

2. In the frontend: In the view definition in TypeScript, in the definition for the `RefNbr` field, specify the parameters in the `columnConfig` decorator, as shown in the following code.

```
@columnConfig({
  editorConfig: {
    parameters: (screen: AP203500) => ({
      "APRegister.docType": screen.Document_Detail.activeRow.DocType.value })
  }
})
RefNbr: PXFieldState;
```

Selector Control: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the selector control to HTML or TypeScript elements.

PXSelector

The following table shows the correspondence between `PXSelector` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

PXSelector

```
<px:PXSelector
  runat="server"
  ID="edContactID"
  DataField="ContactID"
  AllowEdit="True"
  DataSourceID="ds"
  AllowAddNew="True"
  AutoRefresh="True"
  CommitChanges="True" />
```

</td><td>

Use the `field` tag, as shown in the following code. The control represents the selector control if the `PXSelector` attribute is specified (or its descendants are specified) for the DAC field in the `name` attribute.

```
<field name="ContactID"
  config-allow-edit.bind="true">
</field>
```

In rare cases, you can use the `qp-selector` tag.

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

AllowAddNew

```
<px:PXSelector ID="edContactID" ...
  AllowAddNew="True" />
```

</td><td rowspan="2">

Use the `config-allow-edit` property in HTML, as shown in the following code. The property enables the link and add the **+** (**Add Row**) button to the lookup table of the selector control.

```
<field name="ContactID"
  config-allow-edit.bind="true">
</field>
```

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

AllowEdit

```
<px:PXSelector ID="edContactID" ...
  AllowEdit="True" />
```

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

CommitChanges

```
<px:PXSelector ID="edContactID" ...
  CommitChanges="True" />
```

</td><td>

In the TypeScript file, use the `PXFieldOptions.CommitChanges` type parameter when defining the field, as shown in the following code.

```
ContactID:
  PXFieldState<PXFieldOptions.CommitChanges>;
```

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

DataField

```
<px:PXSelector
  DataField="ContactID" ... />
```

</td><td>

Use the `name` property of the `field` tag in HTML to specify the DAC field name, as shown in the following code.

```
<field name="ContactID" ... ></field>
```

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

DisplayMode

```
<px:PXSelector ID="edSrvOrdType"
  DataField="SrvOrdType"
  DisplayMode="Text" ... />
```

</td><td>

Specify what should be displayed in the selector by using the `config-display-mode` property in HTML, as shown in the following code. The possible values are "`both`", "`id`", and "`text`".

```
<field name="EntityTypeName"
  config-display-mode.bind="'text'">
</field>
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>TextField</code><pre><px:PXSelector ID="edViewName"
  DataField="ViewName"
  ValueField="Name"
  TextField="DisplayName"
  ... /></pre></td><td>Use the <code>textField</code> property in the <code>controlConfig</code> decorator.<pre>@controlConfig({
  valueField: "Name", textField: "DisplayName" })
ViewName: PXFieldState<PXFieldOptions.CommitChanges>;</pre></td></tr>
<tr><td><code>TextMode</code><pre><px:PXSelector ID="edNavScreen"
  DatField="Link"
  DisplayMode="Text"
  TextField="Title"
  TextMode="Editable"/></pre></td><td>Use the <code>selectorMode</code> property in the <code>controlConfig</code> decorator.<pre>@columnConfig({
  displayMode: GridColumnDisplayMode.Both,
  editorConfig: {
    displayMode: "text",
    selectorMode: PXSelectorMode.TextModeEditable
  }
})
Link: PXFieldState<PXFieldOptions.CommitChanges>;</pre></td></tr>
<tr><td><code>ValueField</code><pre><px:PXSelector ID="edViewName"
  DataField="ViewName"
  ValueField="Name"
  TextField="DisplayName"
  ... /></pre></td><td>Use the <code>valueField</code> property in the <code>controlConfig</code> decorator.<pre>@controlConfig({
  valueField: "Name", textField: "DisplayName" })
ViewName: PXFieldState<PXFieldOptions.CommitChanges>;</pre></td></tr>
</table>

PXSegmentMask

The following table shows the correspondence between `PXSegmentMask` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

PXSegmentMask

```
<px:PXSegmentMask
  CommitChanges="True"
  ID="edCustomerID"
  runat="server"
  DataField="CustomerID"
  AllowAddNew="True"
  AllowEdit="True"
  DataSourceID="ds"
  AutoRefresh="True"/>
```

</td><td>

Use the `field` tag, as shown in the following code. The control represents the selector control if the `PXSelector` attribute is specified (or its descendants are specified) for the DAC field in the `name` attribute.

```
<field name="CustomerID"
  config-allow-edit.bind="true"></field>
```

In rare cases, you can use the `qp-selector` tag.

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

Wildcard

```
<px:PXSegmentMask
  DataField="SubCD"
  SelectorMode="Segment"
  WildCard="?" />
```

</td><td>

Use the `wildCard` property of the `controlConfig` or `columnConfig` decorator.

```
@columnConfig({
  wildcard: "?",
  editorConfig: {
    wildcard: "?"
  }
})
SubCD: PXFieldState;
```

You need to specify the `wildCard` property twice: first for the column value when the table just shows the value, and second in the `editorConfig` decorator, when a user actually selects the control inside the column and edits the value in the control.

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

PXMultiSelector

The following table shows the correspondence between the `PXMultiSelector` ASPX element and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXMultiSelector`

```
<px:PXMultiSelector
  ID="edMailTo"
  runat="server"
  SkinID="email"
  TextField="SearchSuggestion"
  ValueField="EMail"
  DataSourceID="ds"
  DataField="MailTo"
  Width="100%"
  AutoGenerateColumns="false"
  Hidden="True"
  CommitChanges="True">
```

</td><td>

Use the `multiSelect` property, which is available in the `config` attribute of the `qp-selector` control, as shown in the following code. The property indicates whether multiple values can be selected in the control.

```
@controlConfig({
  multiSelect: true,
  valueTemplate: '{1} <{0}>' })
MailTo: PXFieldState;
```

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

Parameters

The following table shows the correspondence between `Parameters` and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`Parameters`

```
<px:PXSelector ... DataField="RefNbr">
  <Parameters>
    <px:PXControlParam ControlID="grid"
      Name="APRegister.docType"
      PropertyName="DataValues[&quot;Doc-
Type&quot;]"
      Type="String" />
  </Parameters>
</px:PXSelector>
```

</td><td>

Use the `parameters` property in the `config` property of the `qp-selector` control. For details, see *Selector Control: Selector Parameters*.

```
@columnConfig({
  editorConfig: {
    parameters: (screen: AP203500) => ({
      "APRegister.docType": screen.Docu-
ment_Detail.activeRow.DocType.value })
  }
})
RefNbr: PXFieldState;
```

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

Splitter

In this chapter, you will learn about the configuration of a splitter control, including when to use a splitter and how to name it.

Splitter: General Information

A *splitter* is a control that gives you the ability to organize two panes with controls so that the panes can occupy the same part of the form, and give a user an ability to adjust the width of the panes and hide and show them separately.

Learning Objectives

In this chapter, you will learn the following about a splitter:

- The design guidelines for the splitter, including the naming conventions and layout recommendations
- The proper configuration of the splitter for specific cases, such as removing scrollers
- The details of each property of the splitter

Applicable Scenarios

You configure a splitter when you need to organize two panes with controls on a form and give users the ability to close each of the areas.

Splitter Control

The splitter control looks like a line with three dots in the middle and arrows that can be used to open or close the encased controls. When you hover over the three dots (see the first screenshot below), the two arrows appear next to it (as shown in the second screenshot).

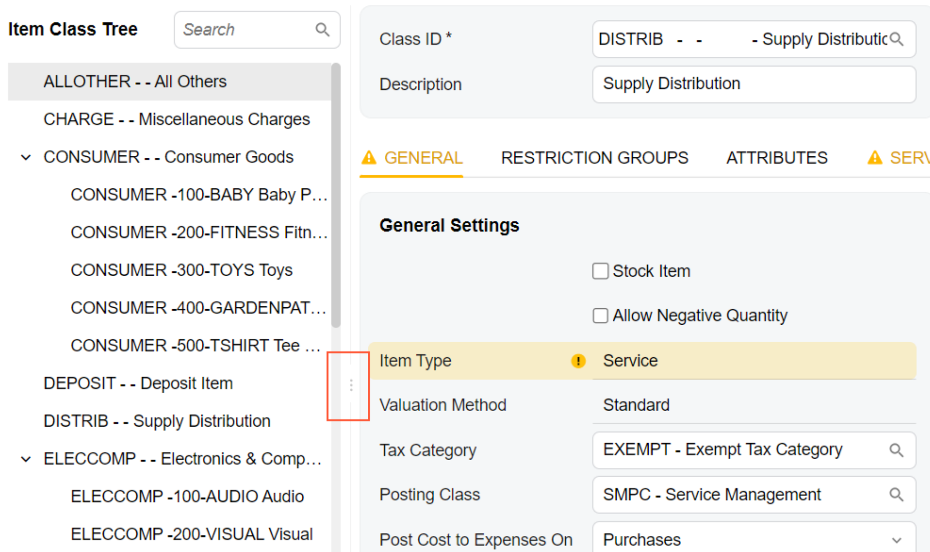


Figure: The three dots on a splitter

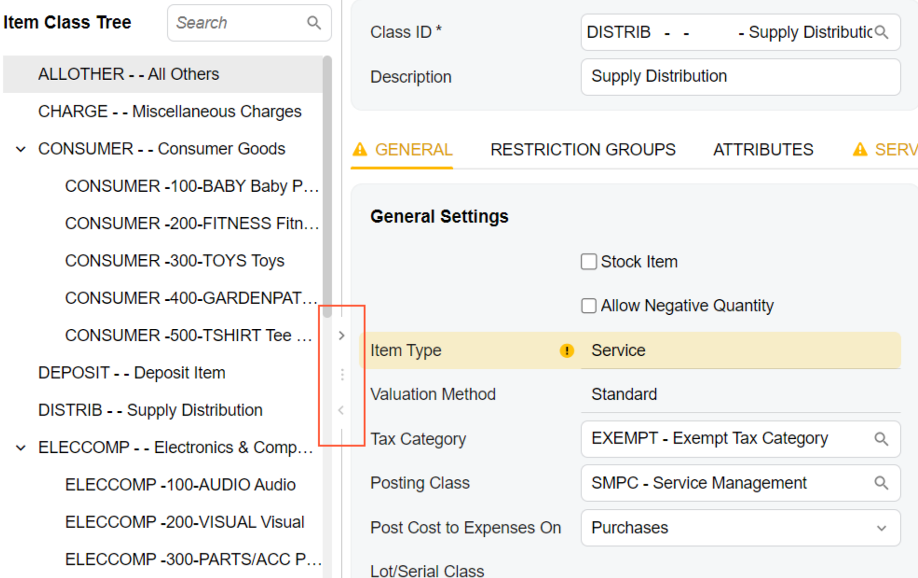


Figure: The arrows on a splitter

When you click the top arrow, you collapse the right pane, when you click the bottom arrow, you collapse the left pane. The following screenshot shows the left pane collapsed.

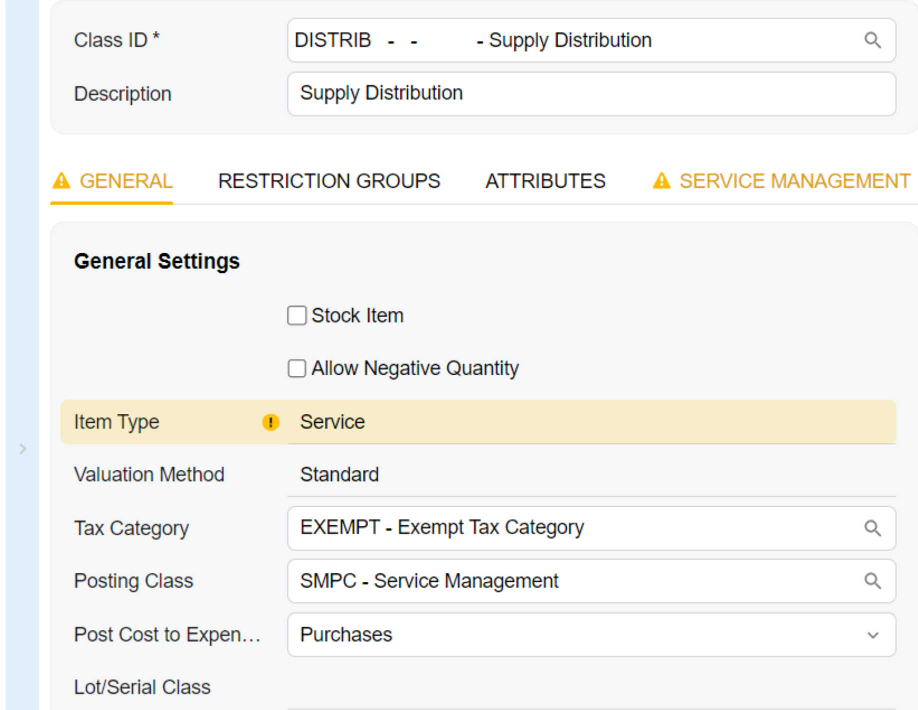


Figure: A splitter with a closed pane

The splitter control is defined by the *`qp-splitter`* tag in HTML.

A splitter contains panes that can be closed or opened by the user. A splitter can include exactly two panes. Each pane can include any number of controls.

Each pane is defined by the `split-pane` tag (so that the `qp-splitter` tag contains two sets of the `split-pane` tags). To put a control in a pane, you add the `split-pane` tag inside the `qp-splitter` tag, and then add the control inside the `split-pane` tag that corresponds to the pane where you want to put the control.

The following example shows two tables located in two panes.

```
<qp-splitter id="sp1">
  <split-pane>
    <qp-grid id="formReceipts" view.bind="Receipts">
    </qp-grid>
  </split-pane>
  <split-pane>
    <qp-grid id="formAPDocs" view.bind="APDocs">
    </qp-grid>
  </split-pane>
</qp-splitter>
```

Splitter ID

An ID of a splitter in HTML consists of two parts, the `splitter` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a splitter that includes two tables on the **PO History** tab may have the `splitter-POHistory` ID, as the following code shows.

```
<qp-splitter id="splitter-POHistory"></qp-splitter>
```

For each pane, the id should consist of two parts, the `splitPaneA` or `splitPaneB` prefix and the semantic name. The letter A or B in the prefix corresponds to the first pane and the second pane, respectively. The semantic name should be identical to the semantic name specified in the `id` of the `qp-splitter` tag. An example is shown in the following code.

```
<qp-splitter id="splitter-POHistory">
<split-pane id="splitPaneA-POHistory">
...
</split-pane>
<split-pane id="splitPaneB-POHistory">
...
</split-pane>
</qp-splitter>
```

Recommendations for Organizing the Layout

The following table shows the recommendations for organizing the layout for the splitter.

Correct	Incorrect
When each pane of a splitter contains a table, and the panes are organized horizontally, make the splitter visible. To make the splitter visible, specify `fading = "false"` in the `qp-splitter` tag, as shown below. `<qp-splitter fading="false">...</qp-splitter>` Do not configure a splitter between two horizontal tables with any of the following parameters: `fading="true"` and `invisible="true"`.	

Correct	Incorrect
Figure: A correct layout	*Figure: An incorrect layout*

Splitter: Configuration

In this topic, you will learn how to configure a splitter control for different cases.

A splitter has many configuration options, which can be set up by means of the `config` property and the set of attributes in the HTML template. The full list of options is located in the `ISplitterConfig` interface.

Splitting the Width and the Height

You can organize panes vertically and horizontally by specifying the value of the `split` attribute. By default, the panes are stacked vertically (`split = "width"`), which means that the width of the control is split into columns. When you specify `split="height"`, the height of the control is split into rows.

When the panes are stacked vertically (`split = "width"`) and the user adjusts the width of one of the panes, the other pane's width is adjusted accordingly. The width of the splitter itself does not change.

When the panes are stacked horizontally (`split="height"`) and the user adjusts the height of one of the panes, this pane's height is changed, but the height of the other pane remains fixed.

Removing Scrollers From Splitter Panes

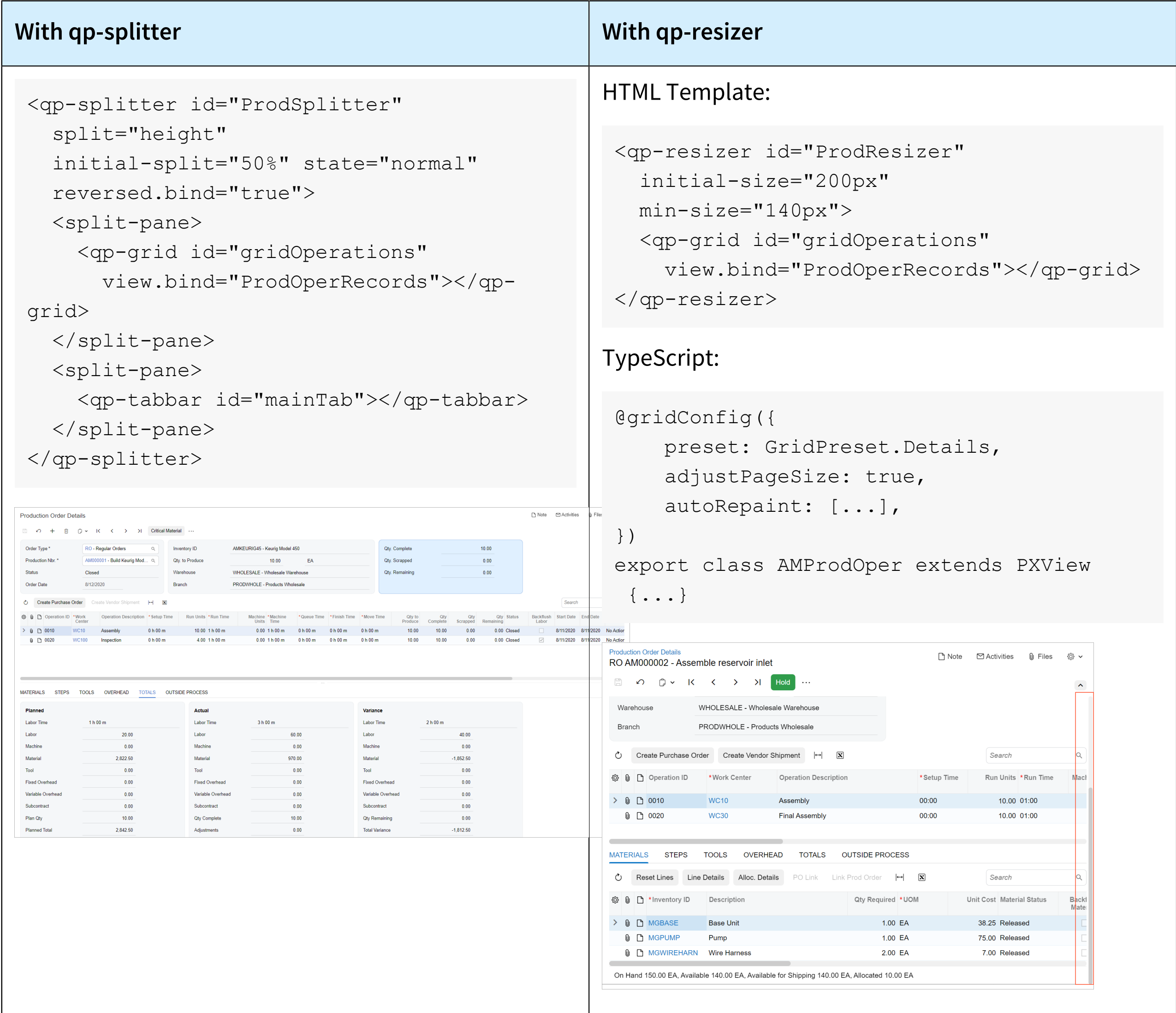
If you are using a splitter, the system may display scrollers for all panes, such as two horizonal scrollers and two vertical scrollers. This may not look very appealing.

As a possible solution, instead of using the splitter, you can use the `qp-resizer` control for one part or both parts of the splitter. In this case, a single vertical scroller will be displayed for the whole form.

For example, suppose that you need to display a table and a tab control below the table. You should put the table inside the `qp-resizer` control and then specify `autoAdjustColumns: true` in the `gridConfig` decorator for this table.

As the result, the paging buttons will appear for the table, and no scroller will be displayed for the table or the tabs.

The following table shows two implementations of this example, along with screenshots showing the result: the first column shows the `qp-splitter` control, and the second column illustrates the `qp-resizer` control.



With qp-splitter

```
<qp-splitter id="ProdSplitter"
  split="height"
  initial-split="50%" state="normal"
  reversed.bind="true">
  <split-pane>
    <qp-grid id="gridOperations"
      view.bind="ProdOperRecords"></qp-
grid>
  </split-pane>
  <split-pane>
    <qp-tabbar id="mainTab"></qp-tabbar>
  </split-pane>
</qp-splitter>
```

With qp-resizer

HTML Template:

```
<qp-resizer id="ProdResizer"
  initial-size="200px"
  min-size="140px">
  <qp-grid id="gridOperations"
    view.bind="ProdOperRecords"></qp-grid>
</qp-resizer>
```

TypeScript:

```
@gridConfig({
    preset: GridPreset.Details,
    adjustPageSize: true,
    autoRepaint: [...],
})
export class AMProdOper extends PXView
  {...}
```

Splitter: Conversion from ASPX to HTML

The following tables will help you to convert the ASPX elements that are related to the splitter to HTML elements.

PXSplitContainer

The following table shows the correspondence between the `PXSplitContainer` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML, you need to replace these ASPX elements with their analogs in HTML.

ASPX	HTML or TypeScript

PXSplitContainer
 	
<px:PXSplitContainer runat="server"
 ID="sp1"
  SplitterPosition="300" SkinID="Horizontal"
  Panel1MinSize="150" Panel2MinSize="150">
	Use the qp-splitter tag, as shown in the following code.
	
	<qp-splitter id="Relations_sp"
	  split="height" initial-split="300"
	  first-pane-min-size="150"
	  last-pane-min-size="150">

<Template1> and <Template2>
 	
<Template1>
  <px:PXGrid ID="grdJoins" ...></px:PXGrid>
</Template1>
<Template2>
  <px:PXGrid ID="grdOns" ...></px:PXGrid>
</Template2>
	Use the <split-pane> tag, as shown in the following code.
	
	<split-pane>
	  <qp-grid id="Relations_grdJoins"
	    view.bind="Relations" wg-container>
	  </qp-grid>
	</split-pane>
	<split-pane>
	  <qp-grid id="JoinConditions_grdOns"
	    view.bind="JoinConditions" wg-container>
	  </qp-grid>
	</split-pane>

ID <px:PXSplitContainer ID="sp1">	Use the id attribute of the qp-splitter tag. <qp-splitter id="splitter-ItemClass">

SplitterPosition
 	
<px:PXSplitContainer ID="sp1"
  SplitterPosition="300">
	Use the initial-split attribute of the qp-splitter tag.
	
	<qp-splitter id="splitter-ItemClass"
	  initial-split="20%">

SkinID
 	
<px:PXSplitContainer ID="sp1"
  SkinID="Horizontal">
	Use the split attribute of the qp-splitter tag.
	
	<qp-splitter id="Relations_sp"
	  split="height">

Panel1MinSize
 	
<px:PXSplitContainer ID="sp1"
  Panel1MinSize="150">
	Use the first-pane-min-size attribute of the qp-splitter tag.
	
	<qp-splitter id="Relations_sp"
	  first-pane-min-size="150">

Panel2MinSize
 	
<px:PXSplitContainer ID="sp1"
  Panel2MinSize="150">
	Use the last-pane-min-size attribute of the qp-splitter tag.
	
	<qp-splitter id="Relations_sp"
	  last-pane-min-size="150">

Obsolete ASPX Control

The following table lists the obsolete ASPX element that is related to the splitter. You do not need to replace this ASPX elements with any HTML element.

ASPX Control	Properties
`PXSplitContainer`	• `runat`

Tab

In this chapter, you will learn about the configuration of tabs. You'll learn when to use tabs and how to name them.

Tab: General Information

A tab is a control that looks like a bookmark with a textual title, as shown in the following screenshot. A tab is usually located in a group of tabs, which is called *tab bar*. Users click tabs to navigate between multiple views within a single window.

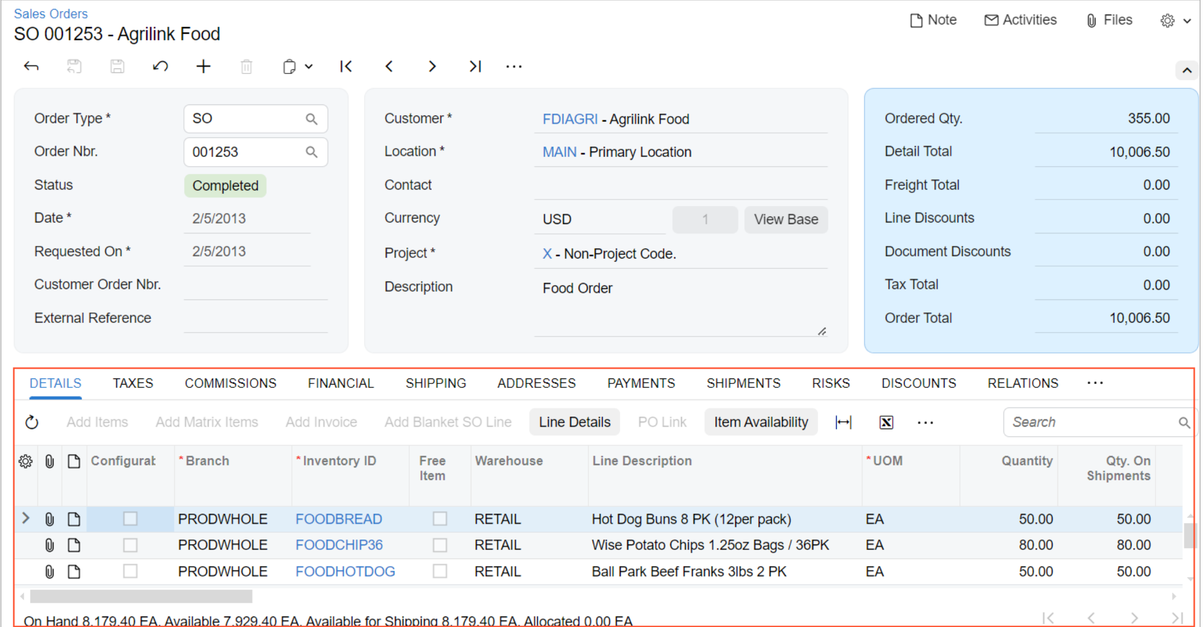


Figure: A form with multiple tabs

A tab is defined by `PXTabItem` in the Classic UI and by the *`qp-tab`* that is nested in the `qp-tabbar` tag in the Modern UI. A tab bar is defined by `PXTab` in the Classic UI and by *`qp-tabbar`* in the Modern UI.

Learning Objectives

In this chapter, you will learn the following information about a tab:

- The design guidelines for a tab, including the naming conventions and layout recommendations
- The proper configuration of a tab for specific cases, such as conditional visibility of the tab
- A detailed description of each property of API elements that are related to a tab

Applicable Scenarios

You configure a tab in the following cases:

- You want to keep a clear distinction between various sets of information, so that a user can easily switch between different contexts within the same Acumatica ERP form.
- You need to guide a user through a step-by-step workflow and simplify complex tasks. You organize a layout with multiple tabs where each tab represents a sequential stage.

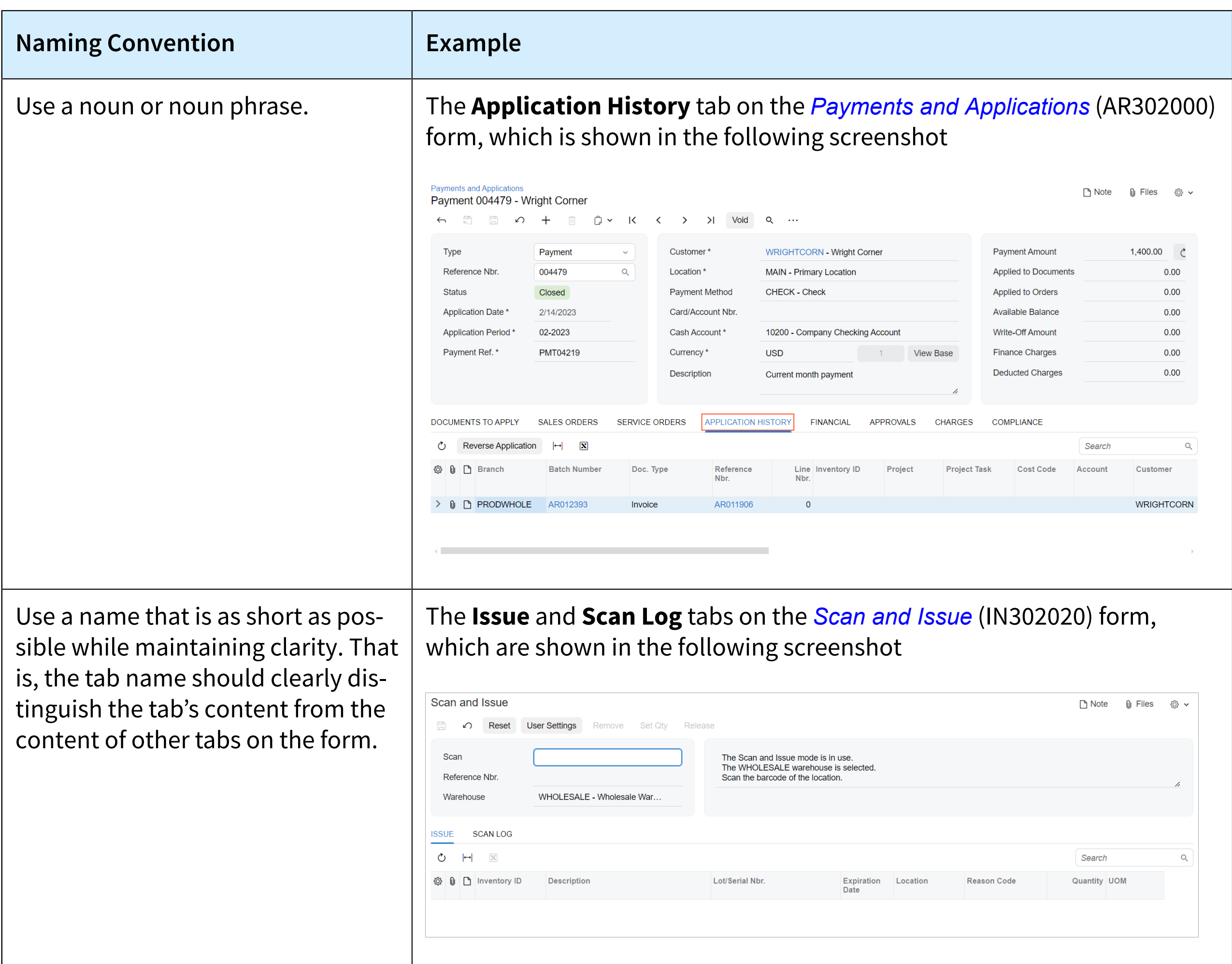
Tab ID

An ID of a tab in HTML consists of two parts, the `tab` prefix and the semantic name. An ID of a tab bar has the `tabs` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a tab that displays invoices may have the `tabInvoices` ID, and the whole tab bar that displays the settings of an sales order may have the `tabsSOOrder` ID, as the following code shows.

```
<qp-tabbar id="tabsSOOrder">
  <qp-tab id="tabInvoices">
  </qp-tab>
</qp-tabbar>
```

UI Naming Conventions

The following table shows the UI naming conventions for tabs.



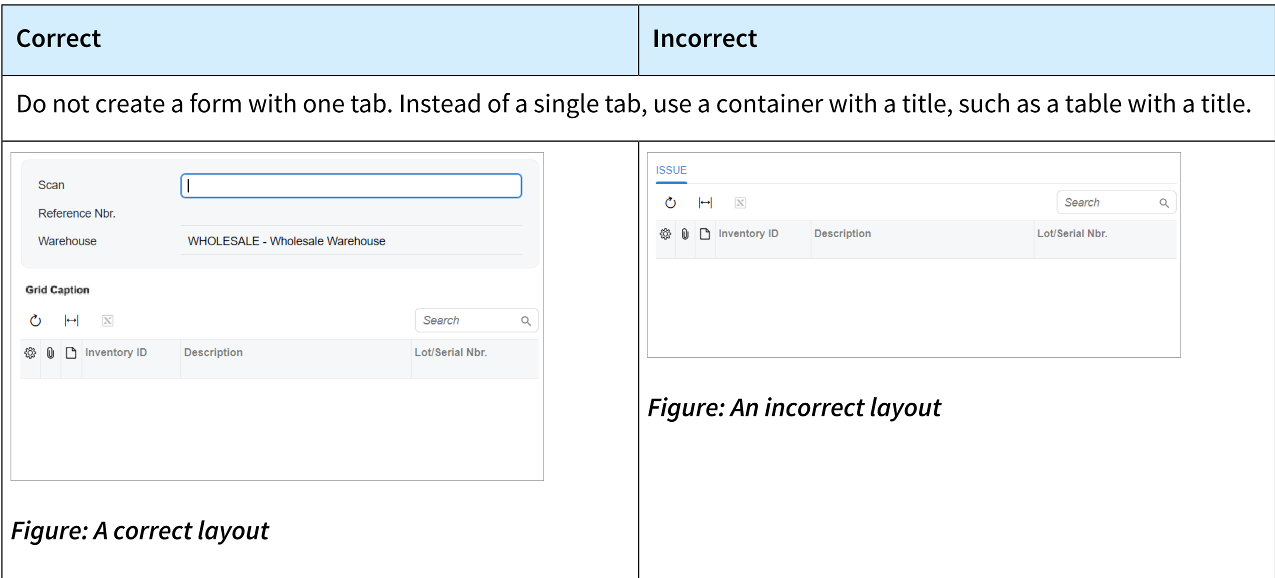
Naming Convention	Example
Use a noun or noun phrase.	The **Application History** tab on the *Payments and Applications* (AR302000) form, which is shown in the following screenshot
Use a name that is as short as possible while maintaining clarity. That is, the tab name should clearly distinguish the tab's content from the content of other tabs on the form.	The **Issue** and **Scan Log** tabs on the *Scan and Issue* (IN302020) form, which are shown in the following screenshot

Naming Convention	Example
For tabs that contain lists or tables with links to related records, use the plural form of the related record's name.	The **Payments** and **Shipments** tabs on the *Sales Orders* (SO301000) form, which are shown in the following screenshot
For tabs with standard content, use the following standard names: • **General**: The core information about a record • **Details**: The detail lines of the record • **Addresses**: The addresses and contact information related to the record • **Financial**: The financial settings of the record • **Shipping**: The shipping settings of the record • **Mailing & Printing**: The mailing and printing settings that can be used for the record, class, or functional area • **Taxes**: The taxes applied to the record • **Discounts**: The discounts applied to the record • **Approvals**: The history of approvals for the record • **Activities**: Activities related to the record	The **General**, **Financial**, **Shipping**, and **Activities** tabs on the *Customers* (AR303000) form, which are shown in the following screenshot

Recommendations for Organizing Layout

The following table shows the recommendations for organizing the layout for tabs.

Correct	Incorrect
Do not create a form with one tab. Instead of a single tab, use a container with a title, such as a table with a title.	
Figure: A correct layout	*Figure: An incorrect layout*



Tab: Configuration

In this topic, you can learn how to define the layout of a tab and how to adjust the tab visibility.

Tab Layout

You organize the layout of a tab as follows:

- If the tab will contain only a table, see *Table (Grid)*.
- If the tab will not contain only a table, you use a nested `qp-template` tag. For details about the organization of the layout with `qp-template`, see *Form Layout: Predefined Templates*.

Tab Layout in an Extension

The `ref` attribute specifies the ID of the tab that is defined in an extension, as shown in the following code.

```
<qp-tab ref="tabUserInfo_Content">
</qp-tab>
```

The tab in an extension can be defined as the following code shows. In this example, you need to use nested `template` tags. The first `template` tag is a template for the whole extension, and the second one is a template for the tab content.

```
<template>
  <template id="tabUserInfo_Content">
    <qp-template ...>
    </qp-template>
  </template>
</template>
```

Conditional Visibility of a Tab

To display the tab conditionally, you can use `visible.bind='<condition>'` for the `qp-tab` tag, as shown in the following example.

```
<qp-tab id="tabDuplicates" caption="Duplicates" ref="tabDuplicates_Content"
  visible.bind="Lead.DuplicateFound.value === true"></qp-tab>
```

However we recommend that you define conditional visibility of a tab in the graph code. For details, see *Configuration of the User Interface in Code*.

Tab: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert tab-related ASPX elements to HTML or TypeScript elements.

PXTab

The following table shows the correspondence between the `PXTab` ASPX control and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

`PXTab`
`<px:PXTab ID="tab" runat="server"`
`  Width="100%" Height="100%">`
`  <Items>`
`    <px:PXTabItem Text="Material">`
`    </px:PXTabItem>`
`    <px:PXTabItem Text="Tools">`
`    </px:PXTabItem>`
`  </Items>`
`  <AutoSize Enabled="True"/>`
`</px:PXTab>`
	Use the `qp-tabbar` control, as shown in the following code. The control represents a tab bar in HTML.
	`<qp-tabbar id="tab">`
	`</qp-tabbar>`

`ID`
`<px:PXTab ID="tabsSOOrder">`
`</px:PXTab>`
	Use the `id` attribute of the `qp-tabbar` control. The attribute specifies the ID of the tab bar, as shown in the following example.
	`<qp-tabbar`
	`  id="tabsSOOrder"`
	`</qp-tabbar>`

`SelectedIndex`
`<px:PXTab SelectedIndex="1">`
`</px:PXTab>`
	Use the `active-tab-id` attribute of the `qp-tabbar` control. The attribute specifies the ID of the active tab on the tab bar, as shown in the following example.
	`<qp-tabbar`
	`  active-tab-id="tabSettings"`
	`</qp-tabbar>`

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>TabIndex</code><pre><px:PXTab TabIndex="23">
</px:PXTab></pre></td><td>Use the <code>tabIndex</code> property of the <code>config</code> attribute of the <code>qp-tabbar</code> control.</td></tr>
</table>

PXTabItem

The following table shows the correspondence between the `PXTabItem` ASPX control and HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>PXTabItem</code><pre><px:PXTabItem
  Text="Estimates"
  BindingContext="form"
  RepaintOnDemand="false">
  <Template>
    <px:PXGrid>
     </px:PXGrid>
  </Template>
</px:PXTabItem></pre></td><td>Use the <code>qp-tab</code> control, as shown in the following code. The control represents a tab in HTML.<pre><qp-tab
  id="tabEstimates"
  caption="Estimates">
  <qp-grid>
  </qp-grid>
</qp-tab></pre></td></tr>
<tr><td><code>LoadOnDemand</code><pre><px:PXTabItem
  LoadOnDemand="true">
</px:PXTabItem></pre></td><td>Use the <code>load-on-demand</code> attribute of the <code>qp-tab</code> control. The attribute makes the system load the tab, even if the tab is inactive. The following code uses this attribute.<pre><qp-tab load-on-demand="true">
</qp-tab></pre></td></tr>
<tr><td><code>Text</code><pre><px:PXTabItem Text="Options">
</px:PXTabItem></pre></td><td>Use the <code>caption</code> attribute of the <code>qp-tab</code> control. The attribute specifies the title of the tab. The follow-ing code uses this attribute.<pre><qp-tab caption="Options">
</qp-tab></pre></td></tr>
<tr><td><code>Visible</code><pre><px:PXTabItem Visible="True">
</px:PXTabItem></pre></td><td>Use the <code>visible</code> attribute of the <code>qp-tab</code> control. The attribute specifies whether the tab appears on the form. The following code uses this attribute.<pre><qp-tab visible="true">
</qp-tab></pre></td></tr>
</table>

ASPX	HTML or TypeScript

VisibleExp
 	
<px:PXTabItem
  VisibleExp="DataControls["edDupli-cateFound"]
    .Value == true"
  LoadOnDemand="True">
	Use visible.bind='<condition>' for the qp-tab tag, as shown in the following example.
	
	<qp-tab
	  visible.bind="Lead.DuplicateFound.value
	 === true">
	</qp-tab>

Obsolete ASPX Controls and Properties

The following table lists obsolete ASPX elements that are related to tabs. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
AutoSize <px:PXTab> <AutoSize Enabled="True"/> </px:PXTab>	All properties

PXTab
 	
<px:PXTab
  AllowAutoHide="True"
  BindingContext="form"
  DataKeyNames="ProdOrdID"
  DataMember="AMBSetupRecord"
  DataSourceID="ds"
  DefaultControlID="edBatchID"
  DynamicTabs="False"
  FilesIndicator="False"
  Height="100%"
  MarkRequired="Dynamic"
  NoteIndicator="False"
  runat="server"
  SelectedIndexExpr=''
  Style="z-index: 100"
  Width="100%" >
</px:PXTab>
	• AllowAutoHide
	• BindingContext
	• DataKeyNames
	• DataMember
	• DataSourceID
	• DefaultControlID
	• DynamicTabs
	• FilesIndicator
	• Height
	• MarkRequired
	• NoteIndicator
	• runat
	• SelectedIndexExpr
	• Style
	• Width

PXTabItem
 	
<px:PXTabItem
  BindingContext="TreeNodeSelectedForm"
  DependsOnView="ComplianceDocuments"
  RepaintOnDemand="true">
</px:PXTabItem>
	• BindingContext
	• DependsOnView
	• RepaintOnDemand

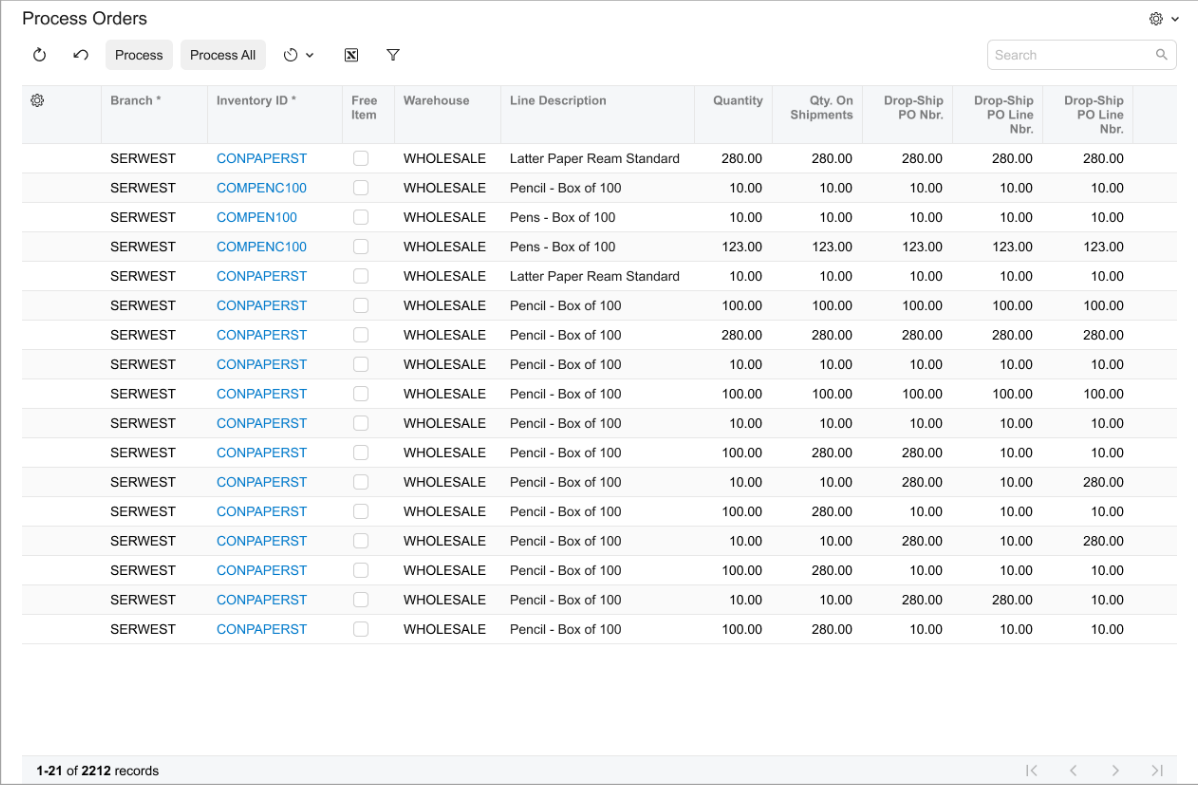
Table (Grid)

In this chapter, you will learn how to configure tables (which are also called *grids*) on Acumatica ERP forms. You will learn which names you need to assign to columns in tables and to buttons on the table toolbar. You will also learn how to organize a layout that includes tables.

Table (Grid): General Information

In a table (or *grid*), each row represents an object or detail,) and each column shows a text title describing a property of the object or detail in the row. A table can be located on a tab or in a dialog box and have its own toolbar. Alternatively. a table can occupy the whole form, as shown in the screenshot below. In this case, the table toolbar is merged with the form toolbar.

A table is defined by `PXGrid` in the Classic UI and by *`qp-grid`* in the Modern UI.



Branch *	Inventory ID *	Free Item	Warehouse	Line Description	Quantity	Qty. On Shipments	Drop-Ship PO Nbr.	Drop-Ship PO Line Nbr.	Drop-Ship PO Line Nbr.
SERWEST	CONPAPERST		WHOLESALE	Latter Paper Ream Standard	280.00	280.00	280.00	280.00	280.00
SERWEST	COMPENC100		WHOLESALE	Pencil - Box of 100	10.00	10.00	10.00	10.00	10.00
SERWEST	COMPEN100		WHOLESALE	Pens - Box of 100	10.00	10.00	10.00	10.00	10.00
SERWEST	COMPENC100		WHOLESALE	Pens - Box of 100	123.00	123.00	123.00	123.00	123.00
SERWEST	CONPAPERST		WHOLESALE	Latter Paper Ream Standard	10.00	10.00	10.00	10.00	10.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	100.00	100.00	100.00	100.00	100.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	280.00	280.00	280.00	280.00	280.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	10.00	10.00	10.00	10.00	10.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	100.00	100.00	100.00	100.00	100.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	10.00	10.00	10.00	10.00	10.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	100.00	280.00	280.00	10.00	10.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	10.00	10.00	280.00	10.00	280.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	100.00	280.00	10.00	10.00	10.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	10.00	10.00	280.00	10.00	280.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	100.00	280.00	10.00	10.00	10.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	10.00	10.00	280.00	280.00	10.00
SERWEST	CONPAPERST		WHOLESALE	Pencil - Box of 100	100.00	280.00	10.00	10.00	10.00

Figure: A table

Learning Objectives

In this chapter, you will learn the following information about a table:

- The design guidelines for a table, including the naming conventions and layout recommendations
- The proper configuration of a table for specific cases, such as a table with a title or a table with highlighted content
- A detailed description of each property of API elements that are related to a table

Applicable Scenarios

You configure a table in the following cases:

- You need to display multiple database records on an Acumatica ERP form, such as information about inventory items included in a sales order.
- You need users to sort and filter data easily. Users can click column headers to sort data based on that column or use filter options to narrow down the displayed information.
- For scenarios where users need to input or edit data in a structured format, such as on the *Chart of Accounts* (GL202500) form, tables provide a familiar and efficient way to do so.

Table ID

An ID of a table in HTML consists of two parts, the `grid` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a table that shows transactions can have the `gridTransactions` ID, as the following code shows.

```
<qp-grid id="gridTransactions"></qp-grid>
```

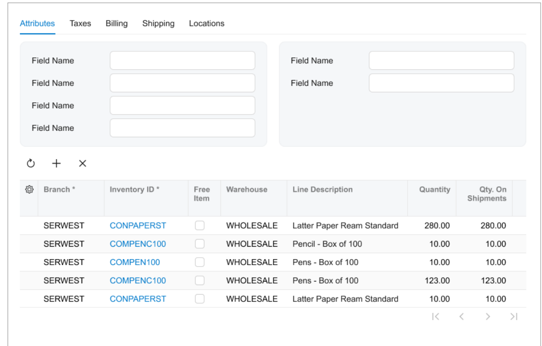
UI Naming Conventions

The following table shows the UI naming conventions for tables.

Naming Convention	Example
For column names, use a noun or noun phrase (except for the case when the column header contains a check box). Preferably, the phrase should include no more than two words.	The **Location Name** column on the **Locations** tab of the *Customers* (AR303000) form, which is shown in the following screenshot
For names of buttons of table toolbars, use verbs or verb phrases that describe the process initiated when the user clicks the button.	The **Release** and **Release All** buttons on the *Release AR Documents* (AR501000) form, which is shown in the following screenshot

Recommendations for Organizing Layout

The following table shows the recommendations for organizing the layout for tables.

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">Show a table without any backgrounds when the table occupies the entire width of the form or popup (including the situation when you have two tables on the form).</td></tr>
<tr><td>
Figure: A correct layout with one table

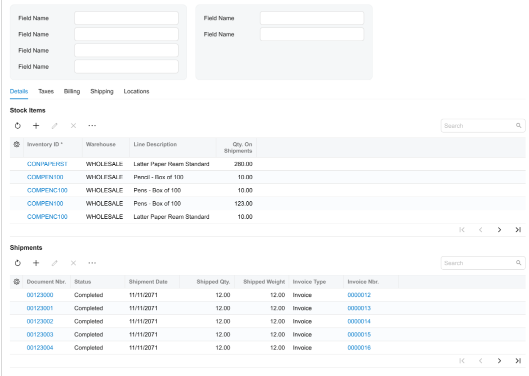
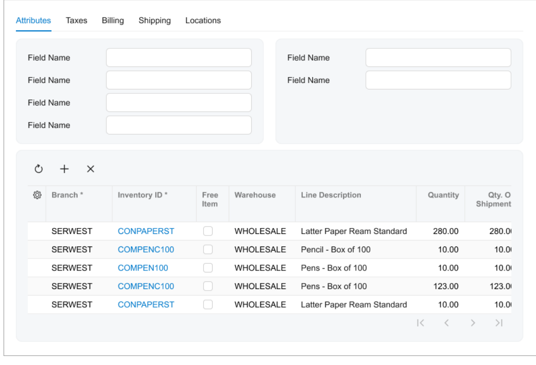



Figure: A correct layout with two tables</td><td>
Figure: An incorrect layout with one table

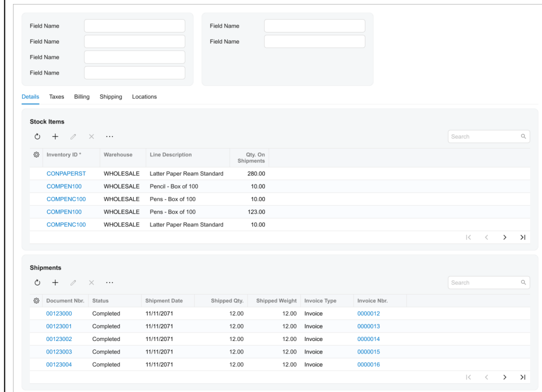
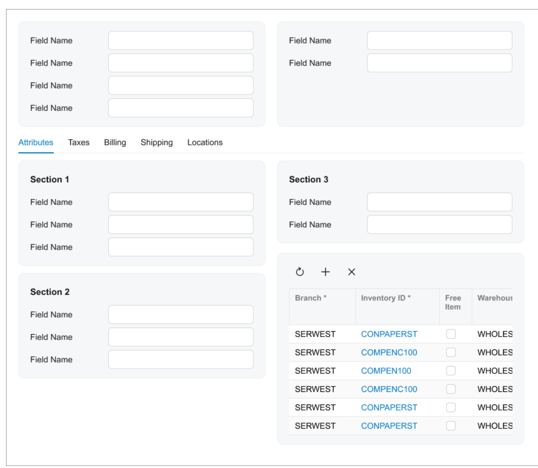
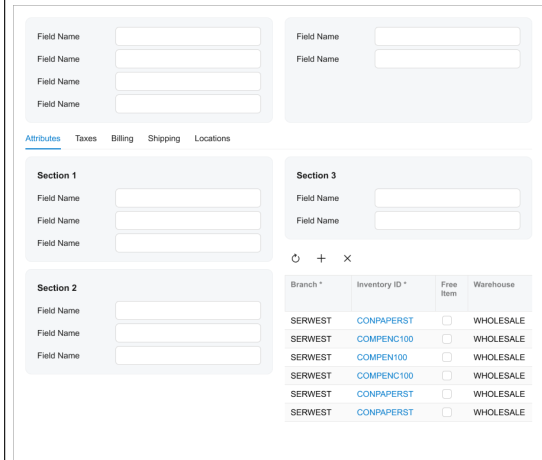
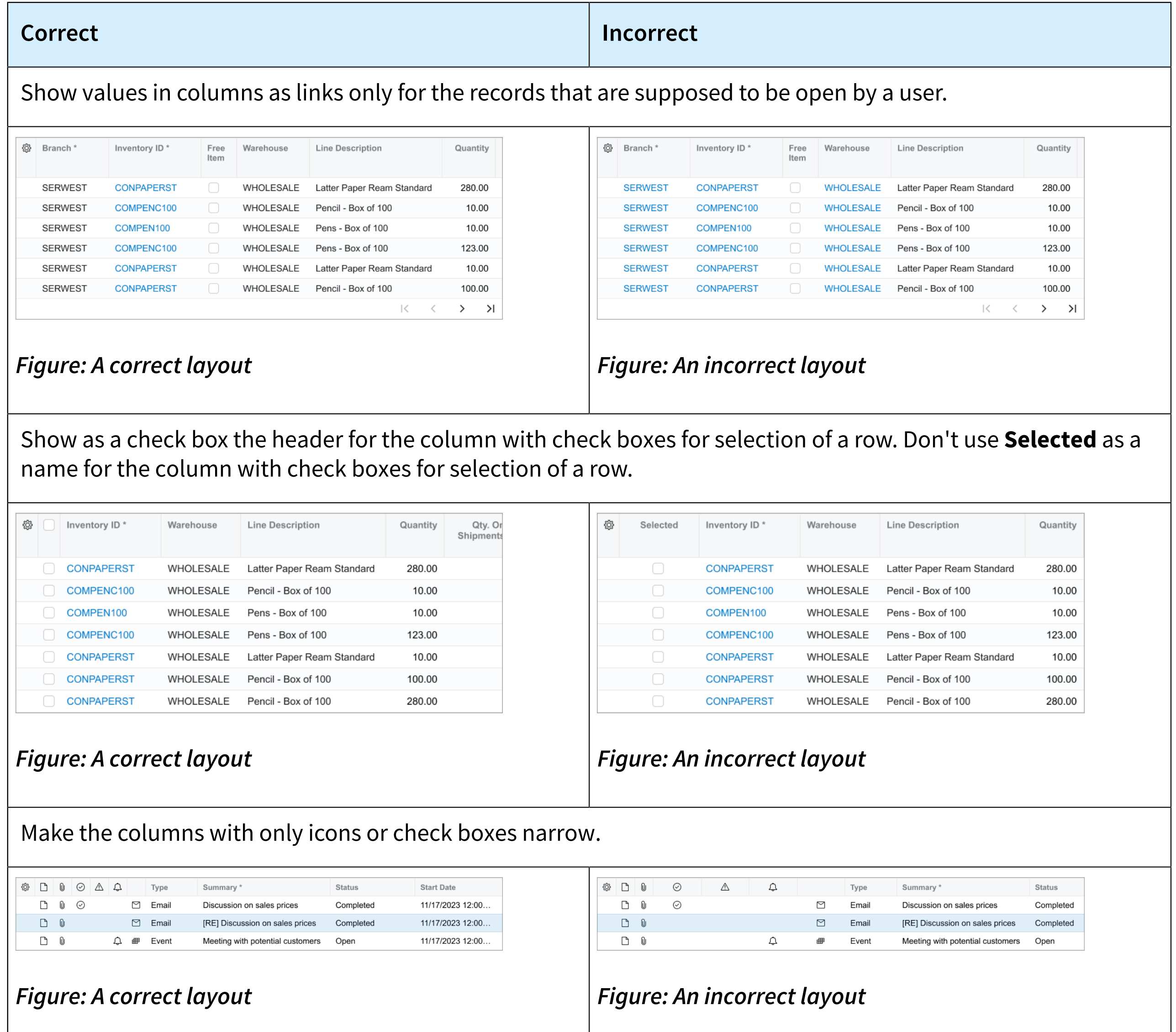



Figure: An incorrect layout with two tables</td></tr>
<tr><td colspan="2">Put a table in a gray section when the table is surrounded by fieldsets. To put a table in a gray section, in the `qp-grid` tag, specify `class="framed-section"`.</td></tr>
<tr><td>
Figure: A correct layout</td><td>
Figure: An incorrect layout</td></tr>
</table>



Correct	Incorrect
Show values in columns as links only for the records that are supposed to be open by a user.	
Figure: A correct layout	*Figure: An incorrect layout*
Show as a check box the header for the column with check boxes for selection of a row. Don't use **Selected** as a name for the column with check boxes for selection of a row.	
Figure: A correct layout	*Figure: An incorrect layout*
Make the columns with only icons or check boxes narrow.	
Figure: A correct layout	*Figure: An incorrect layout*

Table (Grid): Configuration of the Table and Its Columns

In this topic, you can learn how to configure a table and its columns.

Table Configuration

To specify the configuration parameters of a grid, you use the `gridConfig` decorator in TypeScript. You put the decorator on the definition of the view class for the table, as shown in the following example.

```
@gridConfig({
  preset: GridPreset.Inquiry,
  initNewRow: true,
  quickFilterFields: ['AccountClassID', 'Type', 'PostOption', 'CuryID']
})
export class AccountRecords extends PXView {
}
```

For each table, you must specify a preset in the `preset` property of the `gridConfig` decorator. A preset is a predefined set of properties of the `gridConfig` decorator that define how the table is displayed. For more information about presets, see *Form Layout: Grid Presets*.

Table Columns

To specify the configuration parameters of a table column, you use the `columnConfig` decorator, as shown in the following example.

```
export class SOLine extends PXView {
    @columnConfig({ allowShowHide: GridColumnShowHideMode.Server })
    ExcludedFromExport: PXFieldState;
    IsConfigurable: PXFieldState;
    @columnConfig({ hideViewLink: true })
    BranchID: PXFieldState<PXFieldOptions.CommitChanges>;
}
```

Configuring Password Value in Matrix Mode

Suppose that you are configuring a table in the matrix mode, and in one of the columns, a value may contain a password. In case a cell contains a password, the value should be hidden with asterisk (*) symbols.

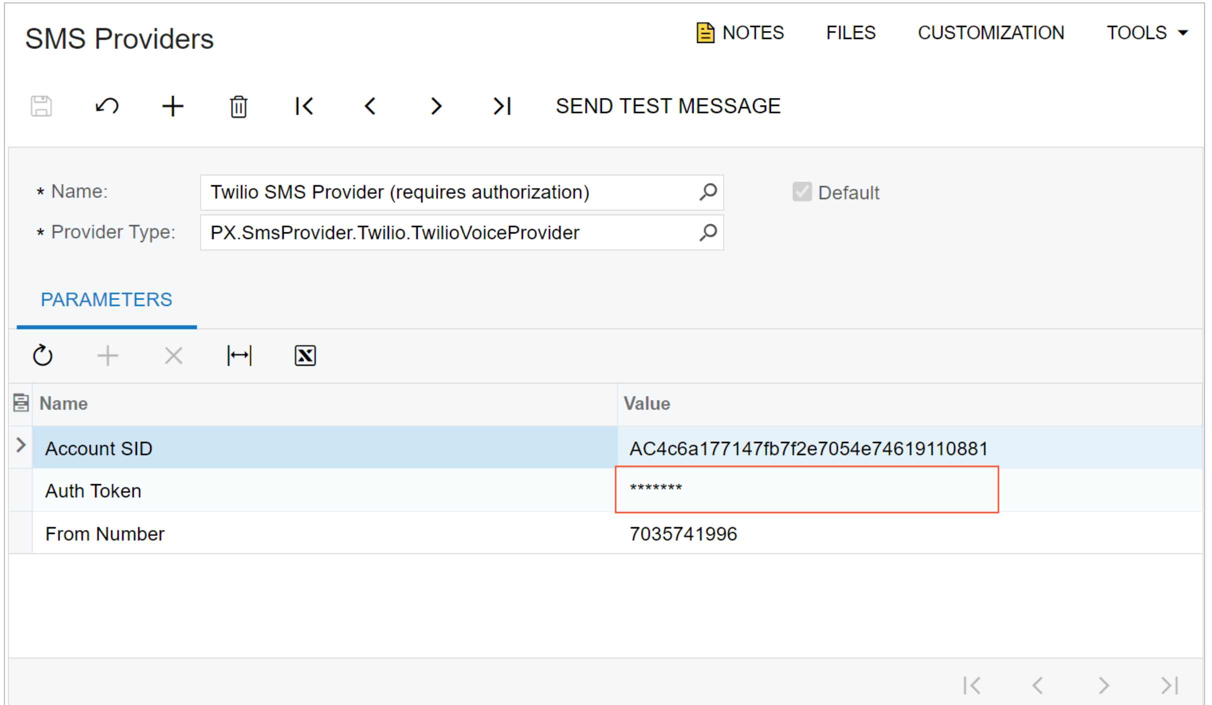


Figure: A password value

This behavior is configured only in backend: When assigning the value that contains a password (`PXStringState`), for example, in the `FieldSelecting` event handler, call the `IsPassword(true)` method for the value.

In frontend, the table and its view are configured as usual without any specific changes such as `config.type == "2"` (which indicates a password value) for the `qp-text-editor` control.

Configuring a Column Footer to Show Totals

To show the totals of all the lines for particular columns in each column's footer area in the UI, you use the `footerText` property of a table column. The following code illustrates the property's use.

```
protected async onSummaryGridDataReadyHandler(grid: QpGridCustomElement,
   args: QpGridEventArgs) {
  if (!grid.view) return;

  const defaultTotal = "00:00";
  const fieldName = 0;
  const fieldValue = 1;
  const totals = [
     ["Sun", this.Document.SunTotal?.value],
     ["Mon", this.Document.MonTotal?.value],
     ["Tue", this.Document.TueTotal?.value],
     ["Wed", this.Document.WedTotal?.value],
     ["Thu", this.Document.ThuTotal?.value],
     ["Fri", this.Document.FriTotal?.value],
     ["Sat", this.Document.SatTotal?.value],
     ["TimeSpent", this.Document.WeekTotal?.value]
  ];
  totals.forEach((total) => {
     const column = grid.getColumn(total[fieldName]);
     if (column) column.footerText = this.formatTime(total[fieldValue] ?? defaultTotal);
  });
}
```

In the code above, the `footerText` property is set to the total value for each column of the grid that is specified in the `totals` collection.

Table (Grid): Configuration of the Table Toolbar

In this topic, you can learn how to configure actions on the table toolbar.

Definition of Actions

The actions of the table toolbar must be defined in TypeScript. You use the `PXActionState` in the grid view class, which is the inheritor of the `PXView` class. (See the example below.) Actions of the table toolbar are not displayed on the form toolbar.

```
export class SOLine extends PXView {
    AddInvoice: PXActionState;
}
```

State and Appearance of Actions

You can also handle the state and appearance of any action that corresponds to a button or command on the table toolbar by using the `actionsConfig` property of the `gridConfig` decorator, as shown in the following examples.

```
// Hides the Refresh button from the table toolbar.
@gridConfig({
```

```
  actionsConfig: { refresh: { hidden: true } }
})
export class SOLine extends PXView

// Adds the Custom Refresh button.
@gridConfig({
  actionsConfig: {
    refresh: {
      renderAs: MenuItem.RENDER_TEXT,
      images: {},
      text: "Custom Refresh" }
  }
})
export class POLine extends PXView
```

You can also use the `actionConfig` decorator to specify properties of an action that is explicitly defined in the view class for the table.

More Button

If all buttons of the table toolbar cannot be displayed on the screen because of the screen's width, you can make the system show the More button in the table toolbar, as shown in the following screenshot.

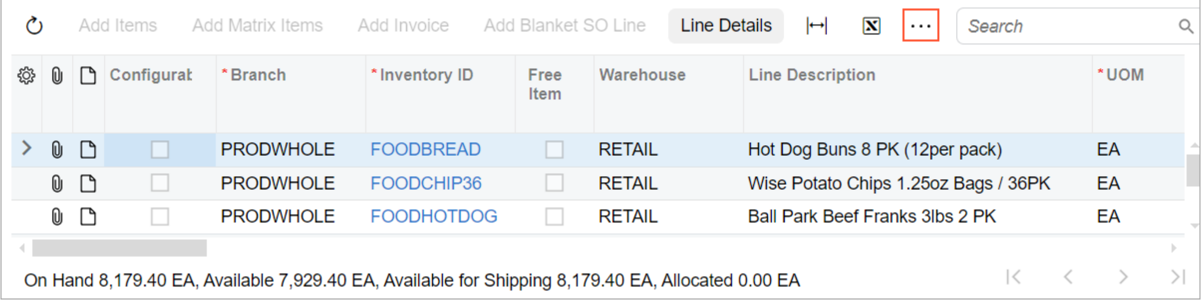


Figure: The More button

To display the More button for the table toolbar, you use the `wrapToolbar` property of the `gridConfig` decorator, as the following code shows.

```
@gridConfig({
    wrapToolbar: true
})
export class SOLine extends PXView
```

Menu Button

You may need to group multiple commands under one menu button, as shown with the **Create Activity** menu button in the following screenshot.

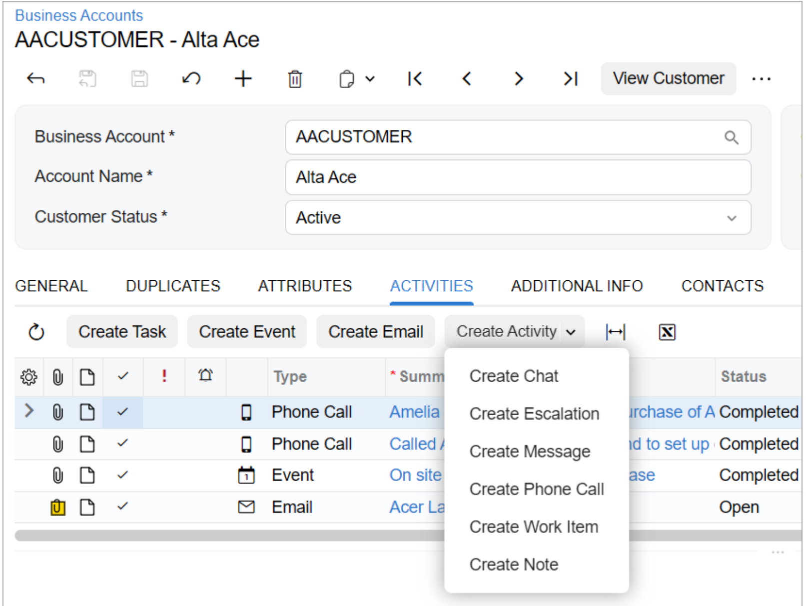


Figure: Menu button

If you need to configure a menu button with static commands for the table toolbar, you use the `topBarItems` property of the `gridConfig` decorator, as the following example shows.

```
@gridConfig({
  topBarItems: {
    TestMenu: {
      type: "menu-options",
      index: 1,
      config: {
        images: {
          normal: "svg:main@external" },
        options: {
          first: {
            text: "First",
            commandName: "First"
          },
          second: {
            text: "Second",
            commandName: "Second"
          }
        }
      }
    }
  }
})
export class SOLine extends PXView
```

If you need to define the list of menu commands dynamically, you use the `PXAction.SetMenu` method in the constructor of the graph or in the overridden `Initialize` method in the graph extension, as shown in the following example. In this case, you do not need to specify the menu commands in the `topBarItems` property of the `gridConfig` decorator.

```
public class MyGraphMaint : PXGraph<MyGraphMaint>
{
  public PXAction<MyDAC> MyMenuAction;
```

```
  private const string MenuAction1 = "FirstAction";
  private const string MenuAction2 = "SecondAction";

  public MyGraphMaint()
  {
    MyMenuAction.SetMenu(new[]
    {
      new ButtonMenu(MenuAction1, Messages.Command_MenuAction1, null),
      new ButtonMenu(MenuAction2, Messages.Command_MenuAction2, null),
    });
  }
}

[PXLocalizable]
public static class Messages
{
  public const string Command_MenuAction1 = "First Command";
  public const string Command_MenuAction2 = "Second Command";
}
```

Table Toolbar Button That Opens a Dialog Box

You can add a button that opens a dialog box to the table toolbar by using the `topBarItems` property of the `gridConfig` decorator. The action that corresponds to this button can be defined only in frontend and has no corresponding action in the graph.

To implement an action that opens a dialog box, in the `topBarItems` property of the `gridConfig` decorator, you specify the following:

1. The internal name of the action
2. The index of the action on the table toolbar
3. The configuration of the action:
   - `commandName`: *ExecuteCommand*
   - `popupPanel`: The name of the dialog box
   - `text`: The text on the button

For example, suppose that you need to define an action that opens the `PanelRef` dialog box (which is defined with the `qp-panel` control in HTML). An example of such action is shown in the following code.

```
@gridConfig({
  topBarItems: {
    PanelRef: {
      index: 0,
      config: {
        commandName: "ExecuteCommand",
        popupPanel: "PanelRef",
        text: Labels.ReferenceDesignators,
      }
    }
  ...
})
```

For details about wrappers for such action, see *Testing of the Modern UI: Frontend Actions in Wrappers*.

Standard Buttons

When you define a table on the form, a set of buttons is added to the table toolbar by default. The following table lists the names of the buttons and names of corresponding actions in code.

Button Name	Action Name
Refresh	`refresh`
Add Row	`insert`
Delete Row	`delete`
Fit to Screen	`adjust`
Export to Excel	`exportToExcel`
Load Records from File	`import`
Filter Setting	`filter`

You can refer to a standard button by its action name in the TypeScript code, for example, to modify visibility in the `actionConfig` decorator. For more details, see *State and Appearance of Actions*.

Table (Grid): Configuration of the Search in the Table

To configure the search in the table, you need to specify fast filter settings for the table. A fast filter is applied to the table when a user enters a value in the Search box of the table or in another box on an Acumatica ERP form if the field that corresponds to this box is used for fast filtering.

Hiding the Search Box or Changing Its Position

By default, the Search box of a table is displayed on the filter bar of the table. You can hide the Search box or display it on the table toolbar by using the `showFastFilter` property in the `gridConfig` decorator, as the following example shows.

```
@gridConfig({
  showFastFilter:
    GridFastFilterVisibility.False
})
export class FieldValue
  extends PXView
```

Including Columns in the List of Searched Fields

To include columns in the list of searched fields, you use the `allowFastFilter` property in the `columnConfig` decorator for the needed columns or the `fastFilterByAllFields` property in the `gridConfig` decorator. Take into account the following considerations:

- By default, the search is performed in all string columns.

- To exclude particular columns from the search, `fastFilterByAllFields` is not needed; you need to use `allowFastFilter: false`.
- To have only particular columns searched, you can set `fastFilterByAllFields: false` and `allowFastFilter: true` for only these columns.

Using a Field Value for Fast Filtering

You may need to filter records in the table by a value that a user has specified in a box on the Acumatica ERP form. To use a field value for fast filtering, you do the following:

1. Specify an alias for the view model of the `qp-grid` element, as the following code shows.

```
<qp-grid id="gridSiteStatus" view.bind="ItemInfo"
  view-model.ref="gridSiteStatusVM">
</qp-grid>
```

2. Use this alias in the `qp-fast-filter` attribute of the field that should be used as a fast filter for the table, as shown below.

```
<field name="Inventory" qp-fast-filter.bind="gridSiteStatusVM">
</field>
```

You can also use one field in multiple fast filters by providing an array of view model aliases in the `qp-fast-filter` attribute, as the following code shows.

```
<field name="Inventory"
  qp-fast-filter.bind="[gridSiteStatusVM, gridRelatedItemsVM]">
</field>
```

Table (Grid): Table with Highlighted Contents

You may need to highlight the contents of a table. For example, you may need to make a row displayed as bold, such as for a row with total values on an inquiry form. To highlight the contents of the table, you implement a method that modifies the CSS for the row and assigns the `handleEvent` decorator to it. In the parameters of the decorator, you specify the event type that the method handles and the view or the view and column for which the event is handled.

The following example makes the row of the `Transactions` view bold.

```
@handleEvent(CustomEventType.GetRowCss, { view: 'Transactions' })
getTransactionsRowCss(args: RowCssHandlerArgs) {
    if (args?.selector?.rowIndex === 1) {
        // see static/custom.css
        return 'bold-row';
    }
    return undefined;
}
```

The following examples makes the `CuryLineAmt` cell of the `Transactions` view colored red if its value is greater than 1000.

```
@handleEvent(CustomEventType.GetCellCss,
  { view: 'Transactions', column: 'CuryLineAmt' })
```

```
getTransactionsCellCss(args: CellCssHandlerArgs): string | undefined {
    if (args?.selector?.rowIndex === 0 &&
        args?.selector?.cellValue > 1000) {
        // see static/custom.css
        return 'red-cell';
    }
    return undefined;
}
```

Table (Grid): Changing of Column Properties Dynamically

You can change column properties dynamically in a table by using an event handler.

The visibility of a column is managed by two properties:

- The `allowShowHide` property of a column, which determines whether a user can show or hide the column in the table
- The `visible` property, which indicates whether the column should be displayed in the table by default

Suppose that you need to hide some of the columns depending on a condition. When the columns are loaded into the table, the `onFilterColumns` event occurs. So to hide a column completely, you need to set values for the `allowShowHide` and `visible` properties in the event handler for the `onFilterColumns` event.

To implement the dynamic change of these properties when the table is loaded, you need to do the following:

1. Implement an event handler for the `onFilterColumns` event in the screen class.

   The following code shows an example of changing the `allowShowHide` and `visible` properties. The event handler is defined in the screen class and accepts the condition for showing the column as a parameter.

   ```
   @graphInfo({
     graphType: "PX.Objects.AM.MRPDisplay",
     primaryView: "Detailrecs"
   })
   export class AM400000 extends PXScreen {
     ...
     protected detailrecsOnFilterColumns(column: IGridColumn, show : boolean):
       boolean {
       if (column.field === "CreateSubAssemblyOrders" ||
           column.field === "OrderType" ||
           column.field === "ProdOrdID")
       {
         column.allowShowHide = show ? GridColumnShowHideMode.Server
           : GridColumnShowHideMode.False;
         column.visible = show;
       }
       return true;
     }
   }
   ```

2. In the `qp-grid` tag of the HTML template, specify the name of the event handler in the `onFilterColumns` attribute. In the attribute value, specify the parameter values for the event handler.

   The following examples show the specification of the event handler with different parameter values.

   ```
   <qp-grid id="gridPurchaseDisplay" view.bind="ProcessRecords"
   ```

```
  on-filter-columns.call="detailrecsOnFilterColumns(col, false)"></qp-grid>
```

```
<qp-grid id="gridManufactureDisplay" view.bind="ProcessRecords"
  on-filter-columns.call="detailrecsOnFilterColumns(col, true)"></qp-grid>
```

Table (Grid): Layout Examples

In this topic, you can find examples of various layouts with tables.

Table in a Gray Section

Suppose that you need to place a table in a gray section, as shown in the following screenshot.

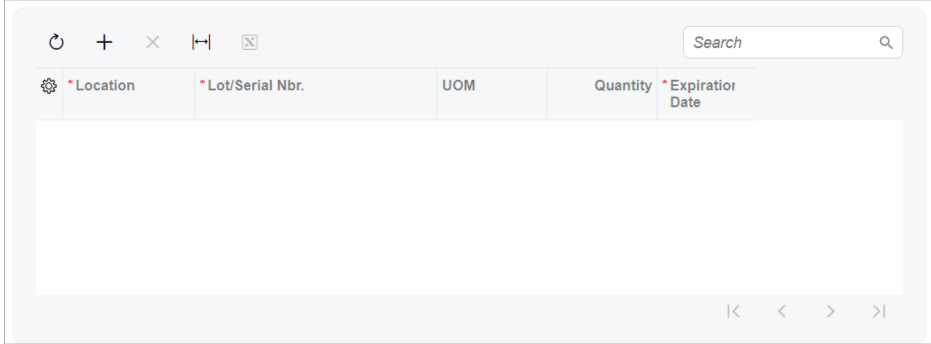


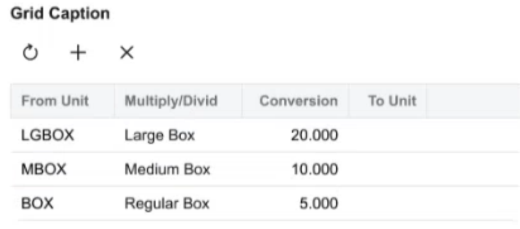
Figure: A table in a gray section

To define a table in a gray section, you specify `class="framed-section"` for the `qp-grid` tag, as shown in the following code.

```
<qp-grid slot="B" id="gridSerialNumbers"
  view.bind="MasterSplits" class="framed-section">
</qp-grid>
```

Table with a Title

Suppose that you need to define a table with a title, as shown in the following screenshot.


Grid Caption

From Unit	Multiply/Divid	Conversion	To Unit
LGBOX	Large Box	20.000	
MBOX	Medium Box	10.000	
BOX	Regular Box	5.000	

Figure: A table with a title

To define a table with a title, you specify the `caption` attribute in the `qp-grid` tag, as shown in the following code.

```
<qp-grid id="gridOrders"
  view.bind="BlanketOrderChildrenDisplayList"
```

```
  caption="Grid Caption">
</qp-grid>
```

Table with Elements Above It in a Gray Section

Suppose that for a table in a gray section, you need to place elements, such as boxes, above the table and inside the gray section, as shown in the following screenshot.

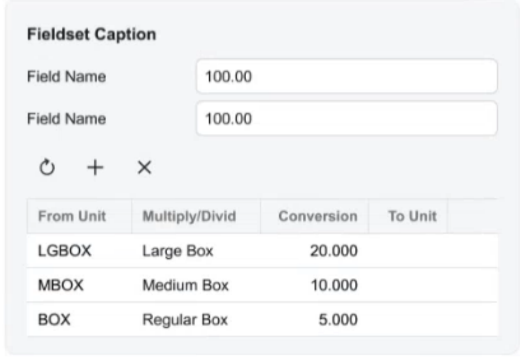


Figure: A table with boxes in a gray section

To define a table with elements above it in a gray section, you do the following in the HTML code:

1. Add a `qp-fieldset` tag.
2. Specify the `caption` attribute for the `qp-fieldset` tag.
3. Put the following N + 1 `field` tags in the `qp-fieldset` tag, where N is the number of elements you need to define above the table:
   - N fields for the elements above the table
   - One field for the table with the `replace-content` and `unbound` attributes
4. Put the `qp-grid` tag inside the field with `replace-content` and `unbound`.

You do not need to specify any classes for coloring because the gray section is displayed for the entire fieldset by default.

The following code implements this approach.

```
<qp-fieldset id="groupID" view.bind="View1" caption="Fieldset Caption">
  <field name="Field1"></field>
  <field name="Field2"></field>
  <field name="FakeField" replace-content unbound>
    <qp-grid id="gridSomeGrid" view.bind="View2"></qp-grid>
  </field>
</qp-fieldset>
```

You generally don't need to explicitly specify the width for the table. However, you may need to do so in cases where the table's width doesn't automatically match the width of its outer container. You can specify `class="col-12"` in the `qp-grid` tag to force the table to match the width of its outer container. Thus, in the preceding code example, you would specify the `qp-grid` tag as follows.

```
<qp-grid id="gridSomeGrid" view.bind="View2" class="col-12"></qp-grid>
```

Table with a Title and Elements Above It in a Gray Section

Suppose that for a table in a gray section, you need to specify a table title; further suppose that you need to add elements above the table inside the gray section, as shown in the following screenshot.

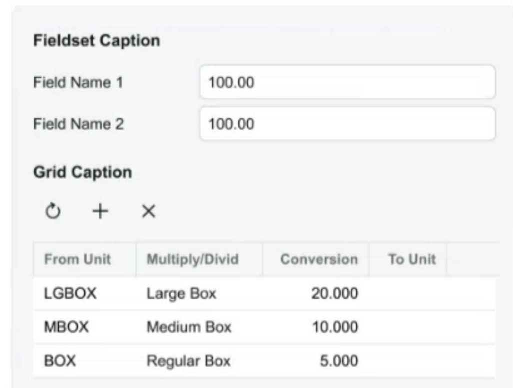


Figure: A table with a title and with elements above it in a gray section

To implement this layout, you need to do the following in HTML code:

1. Add a `qp-fieldset` tag.
2. Add the title for the fieldset by specifying the `caption` attribute for the `qp-fieldset` tag.
3. Put the following N + 1 `field` tags in the `qp-fieldset` tag, where N is the number of elements you need to define above the table:
   - N fields for the elements above the table
   - One field for the table with the `replace-content` and `unbound` attributes
4. Put the `qp-grid` tag inside the field with `replace-content` and `unbound`.
5. Add the title for the table by specifying the `caption` attribute for the `qp-grid` tag.

You do not need to specify any classes for coloring because the gray section is displayed for the entire fieldset by default.

The following code implements this approach.

```
<qp-fieldset id="groupID" view.bind="View1" caption="Fieldset Caption">
  <field name="Field1"></field>
  <field name="Field2></field>
  <field name="FakeField" replace-content unbound>
    <qp-grid id="gridSomeGrid" view.bind="View2" caption="Grid Caption">
    </qp-grid>
  </field>
</qp-fieldset>
```

You generally don't need to explicitly specify the width for the table. However, you may need to do so in cases where the table's width doesn't automatically match the width of its outer container. You can specify `class="col-12"` in the `qp-grid` tag to force the table to match the width of its outer container. Thus, in the preceding code example, you would specify the `qp-grid` tag as follows.

```
<qp-grid
  id="gridSomeGrid"
  view.bind="View2"
  caption="Grid Caption"
  class="col-12">
</qp-grid>
```

Table with a Fixed Height

By default, the grid is stretched over the whole width of the area. If a grid should not occupy the whole tab or form, you specify `class="fixed-height"` for it, as shown in the following example.

```
<qp-grid id="gridSomeGrid" view.bind="GridView" class="fixed-height">
</qp-grid>
```

Table (Grid): Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert ASPX elements that are related to tables to HTML or TypeScript elements.

ActionBar

The following table shows the correspondence between the `ActionBar` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`ActionBar`

```
<px:PXGrid>
  <ActionBar>
    <CustomItems>
      <px:PXToolBarButton
        Text="Add Invoice"
        CommandSourceID="ds"
        CommandName="AddInvoice" />
    </CustomItems>
  </ActionBar>
</px:PXGrid>
```

</td><td>

To specify the toolbar buttons for the grid, define actions in the view class of the grid in TypeScript, as shown in the following code.

```
export class SOLine extends PXView {
    AddInvoice: PXActionState;
}
```

You can also specify particular action properties with the `actionsConfig` and `defaultAction` properties in the `gridConfig` decorator.

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

ASPX	HTML or TypeScript

`DefaultAction`
 	
<px:PXGrid>
  <ActionBar
    DefaultAction="EditDetail"/>
</px:PXGrid>
	Use the `defaultAction` property of the `gridConfig` decorator, as shown in the following code. The property specifies the action that is executed when a user double-clicks a record in the table.
	
	@gridConfig({
	  defaultAction: "EditDetail"
	})
	export class EPApproval
	  extends PXView

AutoCallback

The following table shows the correspondence between the `AutoCallback` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

`AutoCallback`
 	
<px:PXGrid>
  <AutoCallBack
    Command="save"
    Target="view1">
  </AutoCallBack>
</px:PXGrid>
	Use the `autoRepaint` property in the `gridConfig` decorator, as shown in the following code. The property refreshes the specified data view depending on the record selected in the table.
	
	@gridConfig({
	  autoRepaint: ["view1"]
	})
	export class Approvals extends PXView

CallbackCommands

The following table shows the correspondence between the `CallbackCommands` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

`CallbackCommands`
 	
<px:PXGrid>
  <CallbackCommands
    PasteCommand="PasteLine">
    <Save PostData="Container" />
  </CallbackCommands>
</px:PXGrid>
	Use the `pasteCommand` property in the `gridConfig` decorator, as shown in the following code. The property specifies the name of the command that performs the paste operation.
	
	@gridConfig({
	  pasteCommand: "PasteLine",
	})
	export class GIResult extends PXView

Mode

The following table shows the correspondence between the `Mode` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>AllowAddNew</code>
<pre><px:PXGrid>
  <Mode AllowAddNew="True"/>
</px:PXGrid></pre></td><td>Use the <code>allowInsert</code> property of the <code>gridConfig</code> decorator. If the value is true, this property indicates that a user can insert new records directly in the table. The following code uses this property.
<pre>@gridConfig({
  allowInsert: true
})
export class Approvals extends PXView</pre></td></tr>
<tr><td><code>AllowDelete</code>
<pre><px:PXGrid>
  <Mode AllowDelete="True"/>
</px:PXGrid></pre></td><td>Use the <code>allowDelete</code> property of the <code>gridConfig</code> decorator. If the value is true, this property indicates that a user can delete records directly in the table. The following code uses this property.
<pre>@gridConfig({
  allowDelete: true
})
export class Approvals extends PXView</pre></td></tr>
<tr><td><code>AllowDragRows</code>
<pre><px:PXGrid>
  <Mode AllowDragRows="True"/>
</px:PXGrid></pre></td><td>Use the <code>allowDragRows</code> property of the <code>gridConfig</code> decorator. If the value is true, this property indicates that a user can drag rows in the table. The following code uses this property.
<pre>@gridConfig({
  allowDragRows: true
})
export class GIResult extends PXView</pre></td></tr>
<tr><td><code>AllowRowSelect</code>
<pre><px:PXGrid>
  <Mode AllowRowSelect="True"/>
</px:PXGrid></pre></td><td>Use the <code>allowRowSelect</code> property of the <code>gridConfig</code> decorator. If the value is true, this property indicates that a user can select a row in the table. The following example uses this property.
<pre>@gridConfig({
  allowRowSelect: false
})
export class Approvals extends PXView</pre></td></tr>
</table>

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>AllowUpdate</code>
<pre><px:PXGrid>
  <Mode AllowUpdate="True"/>
</px:PXGrid></pre></td><td>Use the <code>allowUpdate</code> property of the <code>gridConfig</code> decorator. If the value is true, this property indicates that a user can update records in the table. The following code uses this property.<pre>@gridConfig({
  allowUpdate: true
})
export class Approvals extends PXView</pre></td></tr>
<tr><td><code>AutoInsert</code>
<pre><px:PXGrid>
  <Mode AutoInsert="True"/>
</px:PXGrid></pre></td><td>Use the <code>autoInsert</code> property of the <code>gridConfig</code> decorator. If the value is true, a user can save default values inserted into a row without changing it. The following code uses this property.<pre>@gridConfig({
  autoInsert: true
})
export class PriceBreak extends PXView</pre></td></tr>
<tr><td><code>InitNewRow</code>
<pre><px:PXGrid>
  <Mode InitNewRow="True"/>
</px:PXGrid></pre></td><td>Use the <code>initNewRow</code> property of the <code>gridConfig</code> decorator. If the value is true, this property indicates that the webpage should perform a server callback upon the insertion of a new row.

Normally, when a user clicks the Add Row button on a table toolbar, the new empty row is created only on the webpage, and the data is not sent to the application server. This means that the C# code with all event handlers is not executed. For the data to be sent to the server, you need to specify <code>initNewRow: true</code>. Also, by specifying this property, you can avoid an extra row being created automatically when a user clicks the Save button.

The following code uses this property.<pre>@gridConfig({
    initNewRow: true
})
export class AccountRecords
  extends PXView</pre></td></tr>
</table>

PXGrid

The following table shows the correspondence between the `PXGrid` ASPX control and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXGrid`

```
<px:PXGrid
  ID="grid"
  runat="server"
  Height="350px"
  Width="100%"
  AdjustPageSize="Auto"
  SkinID="Primary"
  AllowSearch="True"
  FastFilterFields=
    "AccountCD,Description"
  SyncPosition="True">
  <Levels>
    <px:PXGridLevel
      DataMember="AccountRecords">
    </px:PXGridLevel>
  </Levels>
  <Layout
    FormViewHeight="250px" />
  <AutoSize
    Container="Window"
    Enabled="True"
    MinHeight="200" />
  <Mode
    AllowFormEdit="True"
    AllowUpload="True" />
</px:PXGrid>
```

</td><td>

Is replaced by `qp-grid` in HTML. To define a table, you need to instantiate a view class with the `gridConfig` decorator in TypeScript and add a `qp-grid` tag in HTML, as shown in the following code fragments.

```
@graphInfo({ ... })
export class GL103001
  extends PXScreen {
AccountRecords =
  createCollection(GLConsolAccount);
}

@gridConfig({
  preset: GridPreset.Inquiry,
  initNewRow: true
})
export class GLConsolAccount
  extends PXView
```

```
<qp-grid
  id="gridAccountRecords"
  view.bind="AccountRecords">
</qp-grid>
```

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

`AdjustPageSize`

```
<px:PXGrid AdjustPageSize="Auto">
</px:PXGrid>
```

</td><td>

Use the `adjustPageSize` property in the `gridConfig` decorator, as shown in the following example. If the value is *true*, this property makes the table show only one page of rows, without the vertical scrollbar. The following code uses this property.

```
@gridConfig({
    adjustPageSize: true
})
export class ActivityTypes
  extends PXView
```

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

`AllowFilter`

```
<px:PXGrid AllowFilter="True">
</px:PXGrid>
```

</td><td>

Use the `allowStoredFilters` in the `gridConfig` decorator, as shown in the following code. If the value is *true*, the system adds the Search box for the table. The following code uses this property.

```
@gridConfig({
  allowStoredFilters: false
})
export class AppointmentData
  extends PXView
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`AllowPaging`

```
<px:PXGrid AllowPaging="True">
</px:PXGrid>
```

</td><td>

Use the `pagerMode` property in the `gridConfig` decorator, as shown in the following code. The property defines whether pages should be displayed for the table. The property can have one of the following values, which are defined by the `GridPagerMode` enum:

- `InfiniteScroll`: No pages for the table are displayed. A user can only scroll the records of the table.
- `NextPrevFirstLast`: The *Next*, *Previous*, *First*, and *Last* links are displayed for the pages of the table. A user can use these links to switch between the pages of the table.
- `Numeric`: The page numbers are displayed as links that a user can click to switch between the pages of the table.

The following code uses this property.

```
@gridConfig({
  pagerMode:
    GridPagerMode.NextPrevFirstLast
})
export class FASheetHistory
  extends PXView
```

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

`AutoAdjustColumns`

```
<px:PXGrid
  AutoAdjustColumns="True">
</px:PXGrid>
```

</td><td>

Use the `autoAdjustColumns` property in the `gridConfig` decorator. If the value is *true*, the property makes the system adjust the column width automatically. The following code uses this property.

```
@gridConfig({
  autoAdjustColumns: true
})
export class ARDiscount
  extends PXView
```

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

ASPX	**HTML or TypeScript**

`AutoGenerateColumns`
 	
`<px:PXGrid`
`  AutoGenerateColumns="AppendDynamic">`
`</px:PXGrid>`
	Use the `generateColumns` property of the `gridConfig` decorator. The property defines the mode of column generation for the table. You can use one of the following modes, which are defined in the `GridColumnGeneration` enum:
	• *None*
	• *Append*
	• *Recreate*
	• *AppendDynamic*
	• *Selector*
	The following example assigns the *AppendDynamic* mode for the table.
	
	`@gridConfig({`
	`  generateColumns:`
	`    GridColumnGeneration.AppendDynamic`
	`})`
	`export class Parameters extends PXView`

`AutoSaveLayout`
 	
`<px:PXGrid`
`  AutoSaveLayout="True">`
`</px:PXGrid>`
	Use the `autoSaveLayout` property of the `gridConfig` decorator. If the value is *true*, the property makes the system save the layout of the table after each column is resized. The following code uses this property.
	
	`@gridConfig({`
	`  autoSaveLayout: true`
	`})`
	`export class Parameters`
	`  extends PXView`

`BatchUpdate`
 	
`<px:PXGrid`
`  BatchUpdate="True">`
`</px:PXGrid>`
	Use the `batchUpdate` property of the `gridConfig` decorator, as shown in the following code. If a table contains editable fields that depend on each other, the `BatchUpdate` property should be *false*.
	
	`@gridConfig({`
	`  batchUpdate: true`
	`})`
	`export class APInvoice`
	`  extends PXView`

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`BlankFilterHeader`

```
<px:PXGrid
  BlankFilterHeader="All Activities">
</px:PXGrid>
```

</td><td>

Use the `blankFilterHeader` property of the `gridConfig` decorator, as shown in the following code. You can use the `blankFilterHeader` property to define a custom title for the **All Records** filter.

```
@localizable
class Messages {
  static ActivitiesFilterHeader = "All Ac-
tivities";
}

@gridConfig({
  preset: GridPreset.Inquiry,
  showFilterBar: GridFilterBarVisibili-
ty.OnDemand,
  blankFilterHeader: Messages.Activities-
FilterHeader
})
export class Activities extends PXView
```

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

`Caption`

```
<px:PXGrid
  Caption="Schedules">
</px:PXGrid>
```

</td><td>

Use the `caption` attribute of the `qp-grid` control in HTML. The attribute defines a title for the table. When the table is hidden, the title specified in the `caption` attribute is hidden too.

Do not use the `qp-caption` or `qp-label` tags to specify a table title.

The following code specifies a title for the table.

```
<qp-grid
  id="ordersGrid"
  view.bind=
    "BlanketOrderChildrenDisplayList"
  caption="Child Orders">
</qp-grid>
```

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

`EditPageUrl`

```
<px:PXGrid
  EditPageUrl="~/Pages/AP/AP203500.aspx">
</px:PXGrid>
```

</td><td>

Use the `gridDataUrl` property in the `gridConfig` decorator. You can use this property to change the standard API for the retrieval of the table data.

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`FastFilterFields`

```
<px:PXGrid
  FastFilterFields=
    "ClassID,Type,Description">
</px:PXGrid>
```

</td><td>

Use `allowFastFilter` in the `columnConfig` decorator for the needed columns or `fastFilterByAllFields` in the `gridConfig` decorator. Take into account the following considerations:

- By default, the search is performed in all string columns.
- To exclude particular columns from the search, `fastFilterByAllFields` is not needed; you need to use `allowFastFilter: false`.
- To have only particular columns searched, you can set `fastFilterByAllFields: false` and `allowFastFilter: true` for only these columns.

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

`FastFilterId`

```
<px:PXGrid
  FastFilterID="edInventory">
</px:PXGrid>
```

</td><td>

Use the `qp-fast-filter` attribute of the `field` tag, as described in *Table (Grid): Configuration of the Search in the Table*.

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

`FilesIndicator`

```
<px:PXGrid
  FilesIndicator="True">
</px:PXGrid>
```

</td><td>

Use the `showNoteFiles` property in the `gridConfig` decorator. The following code uses this property.

```
@gridConfig({
  showNoteFiles:
    GridNoteFilesShowMode.Suppress
})
export class RecipientProjects
  extends PXView
```

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

`ID`

```
<px:PXGrid ID="gridUsers">
</px:PXGrid>
```

</td><td>

Use the `id` attribute of the `qp-grid` control in HTML. The attribute specifies the ID of the control, as shown in the following code.

```
<qp-grid id="gridUsers">
</qp-grid>
```

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

`KeepPosition`

```
<px:PXGrid KeepPosition="True">
</px:PXGrid>
```

</td><td>

Use the `keepPosition` property in the `gridConfig` decorator. This property indicates whether the system keeps the keys of the record selected in the grid. The following code uses this property.

```
@gridConfig({
  keepPosition: true
})
export class TaxReportLine
  extends PXView
```

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

ASPX	HTML or TypeScript

`MarkRequired`
 	
`<px:PXGrid MarkRequired="Dynamic">`
`</px:PXGrid>`
	Use the `markRequired` property in the `gridConfig` decorator. This property indicates whether the columns of the table can be marked as required. The following code uses this property.
	
	`@gridConfig({`
	`  markRequired: true`
	`})`
	`export class Payments`
	`  extends PXView`

`NoteIndicator`
 	
`<px:PXGrid`
`  NoteIndicator="True">`
`</px:PXGrid>`
	Use the `showNoteFiles` property in the `gridConfig` decorator. The following code uses this property.
	
	`@gridConfig({`
	`  showNoteFiles:`
	`    GridNoteFilesShowMode.Suppress`
	`})`
	`export class RecipientProjects`
	`  extends PXView`

`PageSize`
 	
`<px:PXGrid PageSize="12">`
`</px:PXGrid>`
	Use the `pageSize` property in the `gridConfig` decorator. This property specifies the maximum number of records on a page if the table is displayed on multiple pages. The following code uses this property.
	
	`@gridConfig({`
	`  PageSize: 12`
	`})`
	`export class RecipientProjects`
	`  extends PXView`

`PreserveSortsAndFilters`
 	
`<px:PXGrid`
`  PreserveSortsAndFilters="True">`
`</px:PXGrid>`
	Use the `preserveSortsAndFilters` property in the `gridConfig` decorator. If the value is *true*, this property specifies that the current filter and sorted columns should be stored in the session. By default, the value is *false*.
	
	The following code uses this property.
	
	`@gridConfig({`
	`  preserveSortsAndFilters: true`
	`})`
	`export class BPEventHistory`
	`  extends PXView`

ASPX	HTML or TypeScript

`RepaintColumns`
 	
`<px:PXGrid`
`  RepaintColumns="True">`
`</px:PXGrid>`
	Use the `repaintColumns` property in the `gridConfig` decorator. If the value is *true*, the system repaints column on every request to the server. By default, the value is *false*.
	
	The following code uses this property.
	
	`@gridConfig({`
	`  repaintColumns: true`
	`})`
	`export class CashFlowForecast`
	`  extends PXView`

`SkinID`
 	
`<px:PXGrid`
`  SkinID="Details">`
`</px:PXGrid>`
	Use the `preset` property of the `gridConfig` decorator. The property specifies a set of settings that define the appearance of the table. The following values, which are defined by the `GridPreset` enum, are available:
	
	• `Primary`
	• `Inquiry`
	• `Processing`
	• `ReadOnly`
	• `Details`
	• `Attributes`
	• `ShortList`
	• `Empty`
	
	For more information about presets, see *Form Layout: Grid Presets*.
	
	The following code uses this property.
	
	`@gridConfig({`
	`    preset: GridPreset.Inquiry`
	`})`
	`export class AccountRecords`
	`  extends PXView`

`StatusField`
 	
`<px:PXGrid`
`  StatusField="Availability">`
`</px:PXGrid>`
	Use the `statusField` property of the `gridConfig` decorator. The property specifies the name of the column that is displayed in the footer of the table. The following code uses this property.
	
	`@gridConfig({`
	`  statusField: "Availability"`
	`})`
	`export class SOLine extends PXView`

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`SyncPosition`

```
<px:PXGrid
  SyncPosition="True">
</px:PXGrid>
```

</td><td>

Use the `syncPosition` property of the `gridConfig` decorator. If the value is *true*, the property specifies that the `Current` record must be synchronized with the record selected in the table.

If a form contains a table and the form toolbar includes an action to process a single record that is selected in the table, the action delegate method must have a reference to the selected record in the cache.

To use the `Current` property of a `PXCache` object to access the record selected in a table, the `Current` property must be synchronized with the record selected in the grid. To force the system to provide this synchronization, you have to set the `syncPosition` property of the `gridConfig` decorator to *true*.

The following code uses this property.

```
@gridConfig({
    syncPosition: true
})
export class FABookBalance
  extends PXView
```

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

`TabIndex`

```
<px:PXGrid
  TabIndex="-1">
</px:PXGrid>
```

</td><td>

Use the `tabIndex` property of the `gridConfig` decorator. The property specifies the index of the table during the TAB navigation. The following code uses this property.

```
@gridConfig({
  tabIndex: -1
})
export class SOLine extends PXView
```

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

Table: PXGridColumn

The following table shows the correspondence between the `PXGridColumn` ASPX control and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

ASPX: `PXGridColumn`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          AllowNull="False"
          DataField="Selected"
          TextAlign="Center"
          Type="CheckBox" Width="40px"
          AutoCallBack="true"
          AllowCheckAll="true"
          CommitChanges="true"
        />
        <px:PXGridColumn
          DataField="InventoryCD"
          DisplayFormat="&gt;AAAAAAAAAA"
        />
        <px:PXGridColumn
          DataField="ProductWorkgroupID"
        />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

HTML or TypeScript: Define columns in the view class of the table in TypeScript, as shown in the following example.

```
export class SOLine extends PXView {
  Availability:
    PXFieldState<PXFieldOptions.Hidden>;
  @columnConfig({
    allowShowHide:
      GridColumnShowHideMode.Server
  })
  ExcludedFromExport: PXFieldState;
}
```

ASPX: `AllowCheckAll`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DataField="Selected"
          AllowCheckAll="true" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

HTML or TypeScript: Use the `allowCheckAll` property in the `columnConfig` decorator, as shown in the code below. If the value is *true*, the system adds a check box in the header of the column, which gives the user the ability to select all check boxes in the column for the page.

```
@columnConfig({
  allowCheckAll: true
})
Selected: PXFieldState;
```

ASPX: `AllowFilter`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          AllowFilter="true" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

HTML or TypeScript: Use the `allowFilter` property in the `columnConfig` decorator, as the following code shows. If the value is *true*, the system adds column filtering in the header of the column.

```
@columnConfig({
  allowFilter: true
})
Type: PXFieldState;
```

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`AllowFocus`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          AllowFocus="true" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `allowFocus` property in the `columnConfig` decorator, as the following code shows. If the value is *true*, the system places the cursor in the column when a new record is added to the table.

```
@columnConfig({
  allowFocus: true
})
Type: PXFieldState;
```

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

`AllowNull`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          AllowNull="false" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `allowNull` property in the `gridConfig` decorator, as shown below. If the value is *false*, the system prevents the cell value from being cleared.

```
@columnConfig({
  allowNull: false
})
PaymentLeadTime: PXFieldState;
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`AllowShowHide`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          AllowShowHide="false" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `allowShowHide` property in the `columnConfig` decorator. The property manages a user's ability to show or hide the column in the table. (The user can show or hide the column in the **Column Configuration** dialog box.)

The property can have one of the following values, which are defined by the `GridColumnShowHideMode` enum:

- *False*: The user cannot show or hide the column. The column appears in the table by default.
- *True*: The user can show or hide the column. Whether the column appears in the table by default depends on the field state.
- *Server*: The possibility to show or hide the column is managed on the server. The user can show or hide the column if the server has set `visible: true`.
- *Client*: The user can show or hide the column. The column appears in the table by default. The server does not manage visibility of the column.

The following code shows an example of the use of this property.

```
@columnConfig({
  allowShowHide:
    GridColumnShowHideMode.False
})
OriginalContactID: PXFieldState;
```

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

`AllowSort`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          AllowSort="false" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `AllowSort` property of the `columnConfig` decorator, as shown in the following code. If the value is *false*, the property turns off the capability for a user to sort the values in the column.

```
@columnConfig({
  allowSort: false
})
ContactID: PXFieldState;
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`AllowUpdate`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          AllowUpdate="true" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `allowUpdate` property of the `columnConfig` decorator. If the value is *true*, the property indicates that a user can update the value in the column. The following example uses this property.

```
@columnConfig({
  allowUpdate: true
})
ContactID: PXFieldState;
```

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

`CommitChanges`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DataField="Selected"
          CommitChanges="true" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use `PXFieldOptions.CommitChanges` in TypeScript, as shown in the following code. The option indicates that the webpage commits the changes in the column to the server.

```
export class Details extends PXView {
  Selected:
    PXFieldState<
      PXFieldOptions.CommitChanges>;
}
```

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

`DataField`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DataField="ProductID" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Define data field for the column in the view class of the table in TypeScript, as shown in the following code.

```
export class Details extends PXView {
  ProductID: PXFieldState;
}
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`Decimals`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DataField="RatioPerYear"
          DataType="Decimal"
          Decimals="4" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `decimals` property of the `columnConfig` decorator. The property defines the number of decimal values to be displayed in the column cells. The following example uses this property.

```
@columnConfig({ decimals: 4})
RatioPerYear: PXFieldState;
```

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

`DisplayFormat`

```
<px:PXGridLevel
  DataKeyNames="ApplicationID,TokenID"
  DataMember="Tokens">
  <Columns>
    <px:PXGridColumn DataField="UtcExpire-
dOn"
      DisplayFormat="g"/>
    <px:PXGridColumn DataField="Bearer" />
  </Columns>
</px:PXGridLevel>
```

</td><td>

Use the format property of the columnConfig decorator. The property specifies the custom display format for date and numeric columns. For example, you can set `format="g"` if you want to see date and time in a date column.

```
@columnConfig({format: "g"})
UtcExpiredOn: PXFieldState;
```

For the list of possible values, see *Standard numeric format strings* and *Standard date and time format strings* in Microsoft documentation.

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

`DisplayMode`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DisplayMode="Hint" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `displayMode` property in the `columnConfig` decorator, as shown in the code below. The property specifies the display mode for each cell of the column. The display mode, which is defined by the `GridColumnDisplayMode` enum, can be one of the following:

- *Value* (default): The column cell contains the value of the field.
- *Text*: If there is a description defined for the field, the column cell contains the description of the field.
- *Both*: If there is a description defined for the field, the column cell contains both the value of the field and the description of the field.

```
@columnConfig({
  displayMode:
    GridColumnDisplayMode.Both
})
EMailAccountID: PXFieldState;
```

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

ASPX	HTML or TypeScript

`ForceExport`
 	
`<px:PXGrid>`
`  <Levels>`
`    <px:PXGridLevel>`
`      <Columns>`
`        <px:PXGridColumn`
`          ForceExport="True" />`
`      </Columns>`
`    </px:PXGridLevel>`
`  </Levels>`
`</px:PXGrid>`
	Use the `suppressExport` property of the `columnConfig` decorator. If the value is *true*, the property specifies that the column is not included in the results of the export of the table. The following code uses this property.
	
	`@columnConfig({`
	`  suppressExport: false`
	`})`
	`LineNbr: PXFieldState;`

`Key`
 	
`<px:PXGrid>`
`  <Levels>`
`    <px:PXGridLevel>`
`      <Columns>`
`        <px:PXGridColumn`
`          Key="valueMapping" />`
`      </Columns>`
`    </px:PXGridLevel>`
`  </Levels>`
`</px:PXGrid>`
	Use the `key` property of the `columnConfig` decorator. The property specifies the key for unbound columns, such as Notes and Files. The following code uses this property.
	
	`@columnConfig({`
	`  key: "valueMapping"`
	`})`
	`Value: PXFieldState;`

`LinkCommand`
 	
`<px:PXGrid>`
`  <Levels>`
`    <px:PXGridLevel>`
`      <Columns>`
`        <px:PXGridColumn`
`          LinkCommand=`
`            "ViewDetails" />`
`      </Columns>`
`    </px:PXGridLevel>`
`  </Levels>`
`</px:PXGrid>`
	Use the `linkCommand` decorator for the column field in TypeScript, as shown in the following example. The decorator specifies the action that is executed when a user clicks the link in the column cell.
	
	`@linkCommand("ViewDetails")`
	`AttributeID:`
	`  PXFieldState;`

`MaxLength`
 	
`<px:PXGrid>`
`  <Levels>`
`    <px:PXGridLevel>`
`      <Columns>`
`        <px:PXGridColumn`
`          MaxLength="30" />`
`      </Columns>`
`    </px:PXGridLevel>`
`  </Levels>`
`</px:PXGrid>`
	Use the `maxLength` property in the `columnConfig` decorator. The property specifies the maximum length of the string in the column. The following code uses this property.
	
	`@columnConfig({`
	`  maxLength: 30`
	`})`
	`Value: PXFieldState;`

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`NullText`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          NullText="0.0" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `nullText` property in the `columnConfig` decorator. The property specifies the value that is displayed if the column value is *NULL*. The following code uses this property.

```
@columnConfig({
  nullText: "0.0"
})
Amount: PXFieldState;
```

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

`RenderEditorText`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          RenderEditorText="True" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `renderEditorText` property in the `columnConfig` decorator. The property indicates whether the column value is rendered in the way this value is displayed in inline-edit mode. The following code uses this property.

```
@columnConfig({
  renderEditorText: true,
  editorType: "qp-time-span" })
Mon: PXFieldState;
```

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

`TextAlign`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          TextAlign="Right" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `textAlign` property in the `columnConfig` decorator. The property specifies the alignment of the values in the column. You can use one of the following values, which are defined in the `TextAlign` enum:

- *NotSet*
- *Left*
- *Center*
- *Right*
- *Justify*

The following code uses this property.

```
@columnConfig({
  textAlign: TextAlign.Center
})
Included: PXFieldState;
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`TextField`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          TextField="LocalizedDescr" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `textField` property in the `columnConfig` decorator. The property specifies explicitly which field should be displayed in the column selector control. The following code uses this property.

```
@columnConfig({
  textField: "LocalizedDescr"
})
Descr: PXFieldState;
```

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

`TimeMode`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          TimeMode="True" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `timeMode` property in the `columnConfig` decorator. If the value is *true*, the property turns on the time mode for the date and time column. The following code uses this property.

```
@columnConfig({
  timeMode: true
})
Time: PXFieldState;
```

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

`Type`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          Type="HyperLink" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `type` property in the `columnConfig` decorator. The property specifies the type of the column content. You can use one of the following values, which are defined by the `GridColumnType` enum:

- *NotSet*
- *Icon*
- *HyperLink*
- *CheckBox*

The following code uses this property.

```
@columnConfig({
  type: GridColumnType.HyperLink
})
ScreenId: PXFieldState;
```

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

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`Visible`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          Visible="False" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use `PXFieldOptions.Hidden` in TypeScript, as shown in the following code.

```
export class Details
  extends PXView {
  Selected:
    PXFieldState<
      PXFieldOptions.Hidden>;
}
```

In particular cases, you can use the `visible` property of the `columnConfig` decorator, as shown in the following code.

```
@columnConfig({
  visible: false
})
OwnerID: PXFieldState;
```

Columns with `allowShowHide: GridColumnShowHideMode.False` or `allowShowHide: GridColumnShowHideMode.Client` appear in the table by default.

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

`Width`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          Width="200px" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `width` property in the `columnConfig` decorator. The property specifies the width of the column. The width is defined in pixels. The following code uses this property.

```
@columnConfig({
  width: 200
})
ExpireDate: PXFieldState;
```

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

PXGridColumn

The following table shows the correspondence between the `PXGridColumn` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXGridColumn`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DataField="UtcExpiredOn"
          DisplayFormat="g"/>
        <px:PXGridColumn
          DataField="Bearer" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Define fields for the columns in TypeScript in the view class that corresponds to the table (grid) and use the `columnConfig` decorator to specify column properties, as shown in the following code fragment. Generally, you do not need to add any tags for columns in HTML.

```
@gridConfig({
  preset: GridPreset.ReadOnly
})
export class SignedInOauth
  extends PXView  {
 @columnConfig({format: "g"})
 UtcExpiredOn : PXFieldState;

 Bearer : PXFieldState;
}
```

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

`DataField`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DataField="Bearer" />
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Define fields for the columns in TypeScript in the view class that corresponds to the table (grid), as shown in the following code fragment.

```
export class SignedInOauth
  extends PXView  {
 Bearer : PXFieldState;
}
```

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

`DisplayFormat`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel>
      <Columns>
        <px:PXGridColumn
          DataField="UtcExpiredOn"
          DisplayFormat="g"/>
      </Columns>
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `format` property of the `columnConfig` decorator, as shown in the following example.

```
@columnConfig({format: "g"})
UtcExpiredOn : PXFieldState;
```

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

PXGridLevel

The following table shows the correspondence between the `PXGridLevel` ASPX element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`DataMember`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel
      DataMember="AccountRecords">
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

1. Define a view class in TypeScript, as shown in the following code.

```
export class RS501600
  extends PXScreen {
    Records =
      createCollection(AccountRecords);
}

@gridConfig({...})
export class AccountRecords
  extends PXView
```

2. Bind the `qp-grid` control to the instance of the view class by using the `view.bind` property as follows.

```
<qp-grid view.bind="Records">
</qp-grid>
```

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

`ImportDataMember`

```
<px:PXGrid>
  <Levels>
    <px:PXGridLevel
      ImportDataMember="ImportTemplate">
    </px:PXGridLevel>
  </Levels>
</px:PXGrid>
```

</td><td>

Use the `importView` property of the `gridConfig` decorator.

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

PXGridWithPreview

The following table shows the correspondence between the `PXGridWithPreview` ASPX control and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>
<code>PXGridWithPreview</code>
<pre>
<pxa:PXGridWithPreview
  ID="gridActivities" runat="server"
  DataSourceID="ds" Width="100%"
  AllowSearch="True"
  DataMember="Activities"
  AllowPaging="true"
  NoteField="NoteText"
  FilesField="NoteFiles"
  BorderWidth="0px"
  GridSkinID="Inquire"
  PreviewPanelSkinID="Preview"
  SyncPosition="True"
  BlankFilterHeader="All Activities"
  MatrixMode="true"
  PrimaryViewControlID="form">
</pxa:PXGridWithPreview>
</pre>
</td><td>
To define a grid with a preview text editor below, use the <code>qp-splitter</code> tag, as shown in the following example.
<pre>
<qp-splitter split="height">
  <split-pane>
    <qp-grid
      id="gridActivities"
      view.bind="Activities"
      wg-container="">
    </qp-grid>
  </split-pane>
  <split-pane>
    <qp-rich-text-editor
      id="edActivityBody"
      state.bind="Activities.Body"
      wg-field
      value.bind=
        "Activities.activeRow.Body.value"
      config.bind="{readOnly: true}">
    </qp-rich-text-editor>
  </split-pane>
</qp-splitter>
</pre>
</td></tr>
</table>

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to tables. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

<table>
<tr><th>ASPX Control</th><th>Properties</th></tr>
<tr><td>
<code>ClientEvents</code>
<pre>
<px:PXGrid>
  <ClientEvents
    CellEditorCreated="gridEditor"
    BeforeCellEdit="gridBeforeEdit" />
</px:PXGrid>
</pre>
</td><td>All properties</td></tr>
<tr><td>
<code>OnChangeCommand</code>
<pre>
<px:PXGrid>
  <OnChangeCommand
    Target="tree"
    Command="Refresh" />
</px:PXGrid>
</pre>
</td><td>All properties</td></tr>
</table>

ASPX Control	Properties

Mode
 	
`<px:PXGrid>`
`  <Mode`
`    AllowFormEdit="True"`
`    AllowUpload="True" />`
`</px:PXGrid>`
	• AllowFormEdit
	• AllowUpload

PXGrid
 	
`<px:PXGrid`
`  AllowSearch="True"`
`  BorderWidth="0px"`
`  CaptionVisible="False"`
`  DataSourceID="ds"`
`  ExportNotes="False"`
`  Height="360px"`
`  KeepPosition="True"`
`  PreservePageIndex="True"`
`  RepaintColumns="True"`
`  RestrictFields="True"`
`  runat="server"`
`  Style="height: 192px;"`
`  Width="100%"`
`>`
`</px:PXGrid>`
	• AllowSearch
	• BorderWidth
	• CaptionVisible
	• DataSourceID
	• ExportNotes
	• Height
	• PreservePageIndex
	• RepaintColumns
	• RestrictFields
	• runat
	• Style
	• Width

Time Span

In this chapter, you will learn about the configuration of the time span control. You'll learn when to use the time span control, how to name it, and how to organize a layout that includes it.

Time Span: General Information

A time span is a drop-down control that combines a text box with a list of options. The options of the control represent the time periods from 0 to 24 hours in the following format: `XX:YY`, where `XX` is the number of hours, and `YY` is the number of minutes. An example of a time span control is shown in the following screenshot.

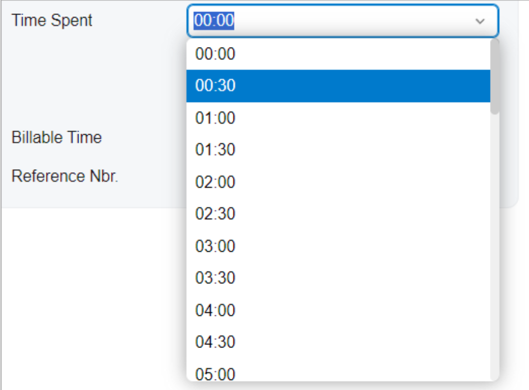


Figure: A time span control

A time span control is defined by the `PXTimeSpan` tag in the Classic UI. In the Modern UI, a time span is defined by the `field` tag, whose control type is automatically defined as a time span from the backend code. In rare cases, a time span in the Modern UI is defined explicitly by the *`qp-time-span`* tag. If the control is used in a column, the control is defined with the *qp-time-span* `editorType` in the `controlConfig` decorator.

The time span control is implemented as an extension of the selector control, so it has the same base properties and behavior.

Learning Objectives

In this chapter, you will learn the following about a time span control:

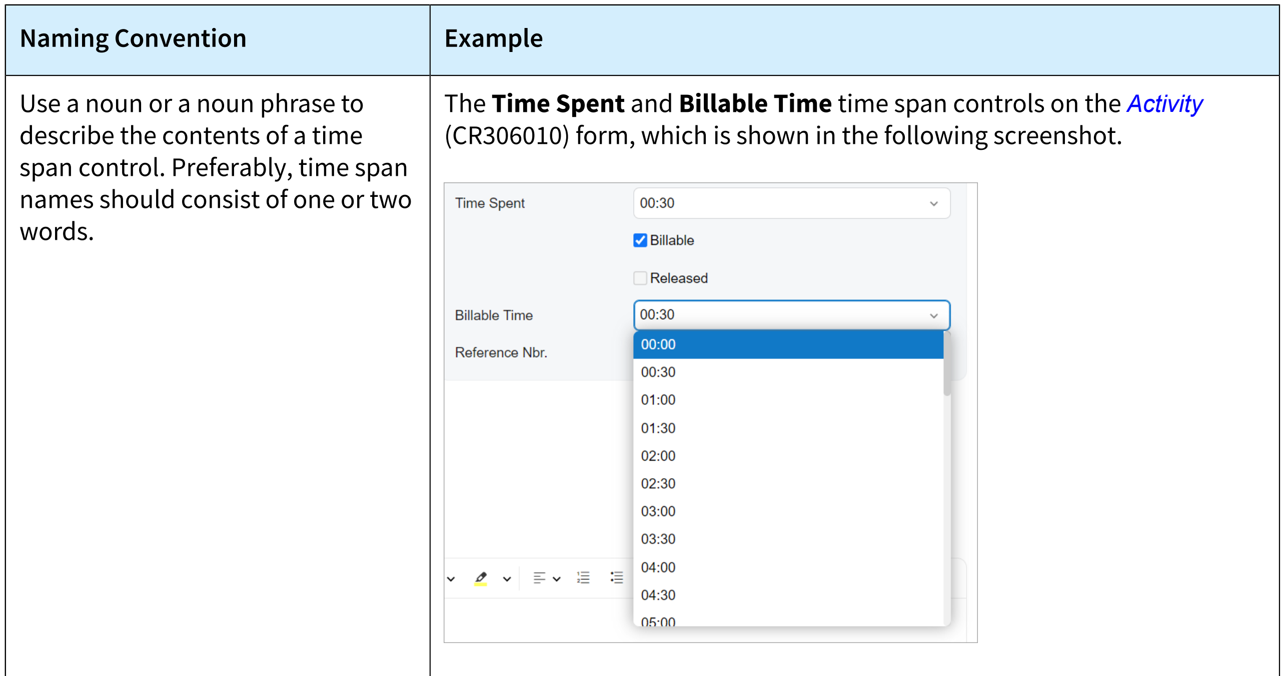
- The design guidelines for the time span, including the naming conventions and layout recommendations
- The details of each property of the time span

Applicable Scenarios

You configure a time span control when you need to add a control where a user can select a period of time, such as the amount of time spent on an activity.

UI Naming Convention

The following table shows the UI naming convention for a combo box.

Naming Convention	Example
Use a noun or a noun phrase to describe the contents of a time span control. Preferably, time span names should consist of one or two words.	The **Time Spent** and **Billable Time** time span controls on the *Activity* (CR306010) form, which is shown in the following screenshot. 

Adding the Time Span Control in a Table

To add the time span control in a column of a table, specify the `editorType` property of the `columnConfig` decorator. To allow a user to edit the value in the cell by typing it, specify `renderEditorText: true`, as shown in the following example.

```
@columnConfig({ renderEditorText: true, editorType: "qp-time-span" })
TimeSpent: PXFieldState<PXFieldOptions.CommitChanges>;
```

Differences Between the Classic UI and the Modern UI

In the Classic UI, a user could only do one of the following: enter the number of minutes or the number of hours, or select an existing value from the list.

While using the time span control in the Modern UI, a user can enter both hours and minutes at the same time and enter the colon symbol between them—for example, *1:40*, meaning 1 hour and 40 minutes. This functionality is supported by default, so you do not need to specify additional parameters.

Default Configuration

By default, the time span control has the following configuration:

- It is optional (`required: false`).
- String values are allowed in the control (`valueType: NetType.String`).
- The suggester is turned off (`autoSuggest : null`).
- Empty values are allowed (`allowNull : true`).
- The list of options in the drop-down list is empty (`options : null`).

Time Span: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the time span control to HTML or TypeScript elements.

PXTimeSpan

The following table shows the correspondence between the `PXTimeSpan` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXTimeSpan` `<px:PXTimeSpan ... />`	Replace it with `field` (whose control type is automatically defined as a time span from the backend code). In rare cases, you should replace it explicitly with the `qp-time-span` control.
`AllowEdit` `<px:PXTimeSpan AllowEdit="True" ... />`	Use the `allowEdit` property of the `config` attribute of the `qp-time-span` control. If the property is set to *true*, a user can enter values that are not available in the combo box. `<field config-allow-edit.bind="true" > </field>`
`AllowNull` `<px:PXTimeSpan AllowNull="False" ... />`	Use the `allowNull` property of the `config` attribute of the `qp-time-span` control. If the property is set to *true*, an empty value is allowed for the combo box. By default, the `allowNull` value is *true*. `<field config-allow-null.bind="false"></field>`
`Enabled` `<px:PXTimeSpan Enabled="False" ... />`	Use the `enabled` property of the `config` attribute of the `qp-time-span` control. If the property is false, a user cannot select a value in the control. `<field config-enabled.bind="false"></field>`

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the time span control. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
PXTimeSpan	• TimeMode • InputMask • runat • ID • Size

Text Box

In this chapter, you will learn about how to configure text boxes and when to use them.

Text Box: General Information

A text box is a control in which a user enters a text string.

A text box is defined by `PXTextEdit` in the Classic UI. In the Modern UI, a text box is defined by the `field` tag (whose control type is automatically defined as a text box from the backend code). In rare cases, a text box in the Modern UI is defined explicitly by the *`qp-text-editor`* control.

Learning Objectives

In this chapter, you will learn the following about a text box:

- The proper configuration of a text box for specific cases, such as when a text box needs to be configured as a multiline text box
- A detailed description of each property of a text box

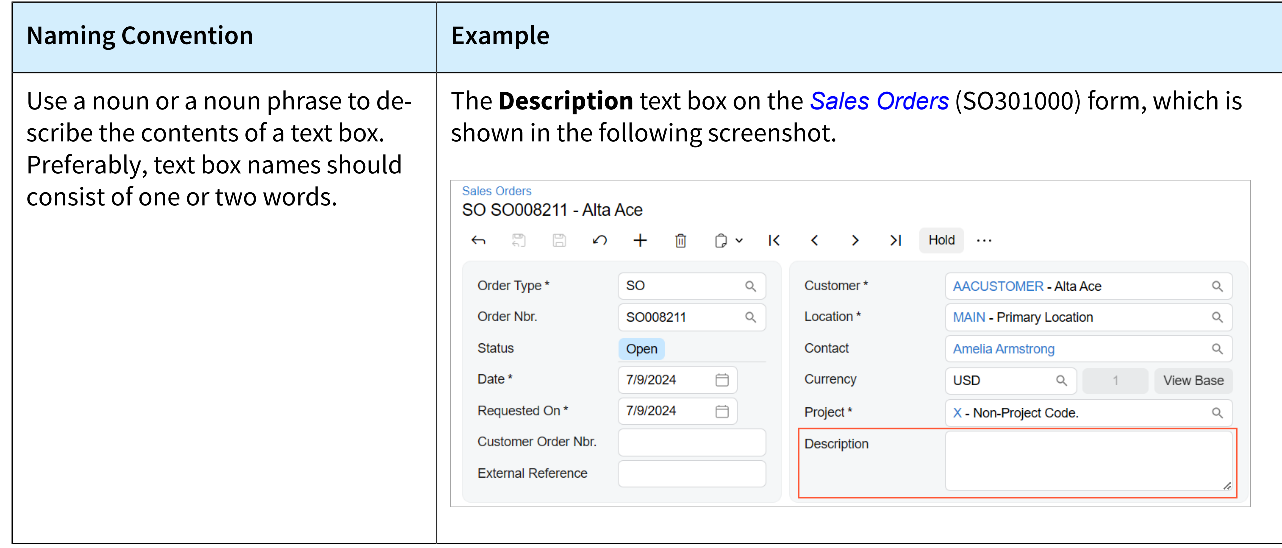
Applicable Scenarios

You configure text boxes in the following user scenarios:

- A user enters some information on a single line, such as the value of a required field.
- A user enters some detailed information on multiple lines, such as the description of a record.
- A user enters sensitive information that must be masked while the user is typing, such as the value of a password field.

UI Naming Convention

The following table shows the UI naming convention for a text box.

Naming Convention	Example
Use a noun or a noun phrase to describe the contents of a text box. Preferably, text box names should consist of one or two words.	The **Description** text box on the *Sales Orders* (SO301000) form, which is shown in the following screenshot. 

Text Box: Multiline Text Box

You can configure a multiline text box by using one of the following approaches:

- Using code in TypeScript (recommended)
- Using code in HTML

You should use multiline text boxes in the Modern UI instead of text boxes that span multiple columns in the Classic UI.

An example of a multiline text box is shown in the following screenshot.

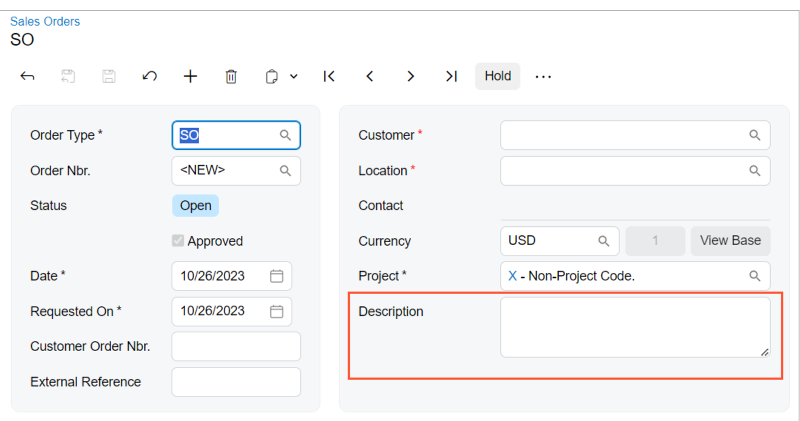


Figure: The Description multiline text box

Configuring a Multiline Text Box with TypeScript (Recommended)

To configure a multiline text box with TypeScript, you use the `controlConfig` decorator with the specified `rows` property, which specifies the number of lines in the control, and define the field with the `PXFieldOptions.Multiline` option, as the following example shows.

```
@controlConfig({rows: 2})
DocDesc: PXFieldState<PXFieldOptions.Multiline>;
```

In the HTML code, the field is defined as shown in the following code.

```
<field name="DocDesc"></field>
```

Configuring a Multiline Text Box with HTML

To configure a multiline text box only in an HTML template, you specify the following attributes in the `field` declaration:

- `config-type.bind="1"`, which indicates that the control has multiple lines.
- `config-rows.bind="N"`, which specifies the number of lines in the control (where `N` is the number of lines).

An example of a multiline box is shown in the following code.

```
<field name="OrderDesc" config-type.bind="1" config-rows.bind="2"></field>
```

Adding a New Line in the Multiline Text Box

By default, when a user clicks Enter in the multiline text box, the focus is shifted to the next control. To add a new line, the user should click Ctrl+Enter.

You can give users the ability to create a new line by pressing Enter. To do it, assign `true` to the `ITextEditorControlConfig.enterKeyAddsNewLine` property, as shown in the following code.

```
@fieldConfig({
  controlType: "qp-text-editor",
  controlConfig: {
    type: 1,
    rows: 13,
    enterKeyAddsNewLine: true
  }
})
Content: PXFieldState;
```

Related Links

- *Interface ITextEditorControlConfig*

Text Box: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the text box control to HTML or TypeScript elements.

PXTextEdit

The following table shows the correspondence between the `PXTextEdit` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXTextEdit` `<px:PXTextEdit ... />`	Replace it with `field` (whose control type is automatically defined as a text box from the backend code). In rare cases, you should replace it explicitly with the `qp-text-editor` control.
`DataField` `<px:PXTextEdit DataField="OrderDesc" ... />`	Use the `name` attribute of the `field` tag. `<field name="OrderDesc"></field>`
`ID` `<px:PXTextEdit ID="CstPXTextEdit1" ... />`	Replace it with the `id` attribute of the `qp-field` tag if this tag is used as a replacement. In other cases, the ID is not necessary for a text box.

Tree

In this chapter, you will learn how to configure trees on Acumatica ERP forms. The chapter also includes information about conversion of tree controls from ASPX to HTML and TypeScript.

Tree: General Information

A tree in the user interface represents a hierarchical data structure, which resembles an inverted tree with a root node at the top and branches extending downward. Each node in the tree represents a piece of data. The connections between nodes illustrate the relationships between them. The following screenshot shows an example of a tree.

Figure: A tree

A tree is defined by `PXTree` or `PXTreeView` in the Classic UI and by *`qp-tree`* in the Modern UI.

Learning Objectives

In this chapter, you will learn the following about a tree:

- The design guidelines for a tree
- The proper configuration of a tree

Applicable Scenarios

You configure a tree if you need to display a hierarchical data structure, such as an organizational chart, a category structure, or nested menu items.

Tree ID

An ID of a tree in HTML consists of two parts, the `tree` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a tree that displays the budget tree may have the `treeBudget` ID, as the following code shows.

```
<qp-tree id="treeBudget"></qp-tree>
```

Tree: Configuration

You configure a tree in TypeScript and define its location on an Acumatica ERP form in HTML.

Definition in TypeScript

To specify the configuration parameters of a tree, you use the `treeConfig` decorator in TypeScript. The `ITreeControlConfig` interface defines the properties that are available in the decorator. You put the decorator on a definition of the view class for the tree, as shown in the following example.

```
@treeConfig({
  parentIdField: "IDParent",
  iconField: "Icon",
  idField: "IDName",
  textField: "NodeText",
  modifiable: false,
  mode: "single",
  singleClickSelect: true,
  onSelect: "SelectNode",
  onAdd: "AddNode",
  onDelete: "DeleteNode",
  onChange: "UpdateNode",
})
export class WorkbenchTreeNode extends PXView  {
  IDName: PXFieldState;
  IDNameOriginal: PXFieldState;
  IDParent: PXFieldState;
  NodeText: PXFieldState;
  Icon: PXFieldState;
  IconColor: PXFieldState;
  Actions: PXFieldState;
  ExtraColumns: PXFieldState;
}
```

In the screen class of the form, you use the `createCollection` method to define the property for the data view that corresponds to the tree, as the following example shows.

```
@graphInfo({
  graphType: "PX.Objects.AM.EngineeringWorkbenchMaint",
  primaryView: "Documents" })
export class AM208100 extends PXScreen {
  BomTree = createCollection(WorkbenchTreeNode);
}
```

Tree Toolbar

The actions of the tree toolbar must be defined in TypeScript. You use the `PXActionState` in the tree view class, which is the inheritor of the `PXView` class. (See the example below.) Actions of the tree toolbar are not displayed on the form toolbar by default.

```
export class TreeNode extends PXView {
    AddProperty: PXActionState;
}
```

Layout in HTML

You use the `qp-tree` tag in HTML to define the position of the tree on an Acumatica ERP form, as the following example shows.

```
<qp-tree id="treeBomTree" view.bind="BomTree"></qp-tree>
```

Tree: Markup of a Tree Node Using HTML

You can mark up the text of a node using HTML. For that purpose, you need to do the following:

- In the `treeConfig` decorator, specify `renderHTML: true`
- In backend, generate the text of the node including the HTML tags

For example, suppose that in a tree node, you need to display part of the text in italics, as shown in the following screenshot.

Figure: A tree node with HTML markup

For that purpose, you need to generate the following text of the node in backend.

```
Actions (2, <span class="nodeHint">inherited 30</span>)
```

In TypeScript, you need to configure the tree, as shown in the following code.

```
@treeConfig({
 dynamic: true,
 hideRootNode: true,
 idField: "NodeID",
 textField: "Title",
 modifiable: false,
 mode: "single",
 singleClickSelect: true,
 syncPosition: true,
 renderHTML: true,
})
```

```
export class SiteMapTree extends PXView  {
 NodeID: PXFieldState;
 ...
}
```

Tree: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to trees to HTML or TypeScript elements.

Behavior

The following table shows the correspondence between the `Behavior` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

`Behavior`
`<px:PXTreeView >`
`  <AutoCallBack >`
`    <Behavior`
`      RepaintControlsIDs="form, formOper"`
`      RepaintControls="All" />`
`  </AutoCallBack>`
`</px:PXTreeView>`
	Use the `autoRepaint` property of the `treeConfig` decorator.

ExtraColumns

The following table shows the correspondence between the `ExtraColumns` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`ExtraColumns`

```
<px:PXTree
  <ExtraColumns>
    <px:ExtraColumn
      Title="Qty"
      TagName="qp-text-editor"
      Width="115"
      TextAlign="3">
    </px:ExtraColumn>
    <px:ExtraColumn
      Title="Uom"
      TagName="qp-text-editor"
      Width="85"
      TextAlign="1">
    </px:ExtraColumn>
  </ExtraColumns>
</px:PXTree>
```

</td><td>Use the `extraColumns` property of the `treeConfig` decorator.</td></tr>
</table>

Images

The following table shows the correspondence between the `Images` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`Images`

```
<px:PXTreeView>
  <Images>
    <ParentImages
      Normal="tree@Folder"
      Selected="tree@FolderS" />
    <LeafImages
      Normal="tree@Folder"
      Selected="tree@FolderS" />
  </Images>
</px:PXTree>
```

</td><td>Use the `images` property of the `treeConfig` decorator.</td></tr>
</table>

PXTree

The following table shows the correspondence between the `PXTree` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXTree`

```
<px:PXTree
  runat="server"
  ID="treeBomTree"
  CallbackUpdatable="True"
  DataMember="BomTree"
  DefaultDrag="Cut"
  Width="100%"
  Height="100%"
  ParentIDName="IDParent"
  IDName="IDName"
  Description="Description"
  ExtraColumnField="ExtraColumns"
  ActionField="Actions"
  IconField="Icon"
  IconColorField="Iconcolor"
  IconSize="22"
  AddSiblingNode="NewSiblingNodeDefault"
  AddChildNode="NewChildNodeDefault"
  OnAdd="AddNode"
  OnDelete="DeleteNode"
  OnChange="UpdateNode"
  OnSelect="SelectNode"
  CheckDropCommand="CheckDropAction"
  Modifiable="True"
  Mode="single"
  SingleClickSelect="True" >
  <ExtraColumns>
    <px:ExtraColumn
      Title="Qty"
      TagName="qp-text-editor"
      Width="115"
      TextAlign="3">
    </px:ExtraColumn>
    <px:ExtraColumn
      Title="Uom"
      TagName="qp-text-editor"
      Width="85"
      TextAlign="1">
    </px:ExtraColumn>
  </ExtraColumns>
</px:PXTree>
```

</td><td>

Use the `qp-tree` tag in HTML and the `treeConfig` decorator for a view class in TypeScript.

The following example shows a declaration of a view class for a tree in TypeScript.

```
@treeConfig({
  parentIdField: "IDParent",
  iconField: "Icon",
  idField: "IDName",
  textField: "Description",
  modifiable: false,
  mode: "single",
  singleClickSelect: true,
  onSelect: "SelectNode",
  onAdd: "AddNode",
  onDelete: "DeleteNode",
  onChange: "UpdateNode",
})
export class WorkbenchTreeNode
 extends PXView  {
  IDName: PXFieldState;
  IDNameOriginal: PXFieldState;
  IDParent: PXFieldState;
  Description: PXFieldState;
  Icon: PXFieldState;
  IconColor: PXFieldState;
  Actions: PXFieldState;
  ExtraColumns: PXFieldState;
}
```

The following HTML code includes the tree in the layout of an Acumatica ERP form.

```
<qp-tree
  id="treeBomTree"
  view.bind="BomTree">
</qp-tree>
```

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

`ActionField`

```
<px:PXTree
  ActionField="Actions" />
```

</td><td>

Use the `actionField` property of the `treeConfig` decorator.

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

ASPX	HTML or TypeScript
`AddChildNode` `<px:PXTree AddChildNode="NewChildNodeDefault" />`	Use the `addChildNode` property of the `treeConfig` decorator.
`AddSiblingNode` `<px:PXTree AddSiblingNode="NewSiblingNodeDefault" />`	Use the `addSiblingNode` property of the `treeConfig` decorator.
`CheckDropCommand` `<px:PXTree CheckDropCommand="CheckDropAction" />`	Use the `checkDropCommand` property of the `treeConfig` decorator.
`DataMember` `<px:PXTree DataMember="ItemClassNodes" />`	Use the `view` attribute of the `qp-tree` control, as the following code shows. `<qp-tree id="tree" view.bind="ItemClassNodes"></qp-tree>`
`DefaultDrag` `<px:PXTree DefaultDrag="Cut" />`	Use the `defaultDrag` property of the `treeConfig` decorator.
`Description` `<px:PXTree Description="Description" />`	Use the `textField` property of the `treeConfig` decorator.
`ExtraColumnField` `<px:PXTree ExtraColumnField="ExtraColumns" />`	Use the `extraColumnField` property of the `treeConfig` decorator.
`Height` `<px:PXTree Height="100%" />`	Do not specify the height of a tree control because it is automatically adjusted for various screen sizes. However, if you need to specify the height for some reason, you can use the `height` property in the `treeConfig` decorator.
`IconColorField` `<px:PXTree IconColorField="Iconcolor" />`	Use the `iconColorField` property of the `treeConfig` decorator.

ASPX	HTML or TypeScript

`IconField`
`<px:PXTree`
`  IconField="Icon" />`
	Use the `iconField` property of the `treeConfig` decorator.

`IconSize`
`<px:PXTree`
`  IconSize="22" />`
	Use the `iconSize` property of the `treeConfig` decorator.

`ID`
`<px:PXTree`
`  ID="tree" />`
	Use the `id` attribute of the `qp-tree` control, as the following code shows.
	`<qp-tree`
	`  id="tree">`
	`</qp-tree>`

`IDName`
`<px:PXTree`
`  IDName="IDName" />`
	Use the `idField` property of the `treeConfig` decorator.

`Mode`
`<px:PXTree`
`  Mode="single" />`
	Use the `mode` property of the `treeConfig` decorator.

`Modifiable`
`<px:PXTree`
`  Modifiable="True" />`
	Use the `modifiable` property of the `treeConfig` decorator.

`OnAdd`
`<px:PXTree`
`  OnAdd="AddNode" />`
	Use the `onAdd` property of the `treeConfig` decorator.

`OnChange`
`<px:PXTree`
`  OnChange="UpdateNode" />`
	Use the `onChange` property of the `treeConfig` decorator.

`OnDelete`
`<px:PXTree`
`  OnDelete="DeleteNode" />`
	Use the `onDelete` property of the `treeConfig` decorator.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>OnSelect</code><pre><px:PXTree
  OnSelect="SelectNode" /></pre></td><td>Use the <code>onSelect</code> property of the <code>treeConfig</code> decorator.</td></tr>
<tr><td><code>ParentIDName</code><pre><px:PXTree
  ParentIDName="IDParent" /></pre></td><td>Use the <code>parentIdField</code> property of the <code>treeConfig</code> decorator.</td></tr>
<tr><td><code>SingleClickSelect</code><pre><px:PXTree
  SingleClickSelect="True" /></pre></td><td>Use the <code>singleClickSelect</code> property of the <code>treeConfig</code> decorator.</td></tr>
<tr><td><code>Width</code><pre><px:PXTree
  Width="100%" /></pre></td><td>Do not specify the width of a tree control because it is automatically adjusted for various screen sizes. However, if you need to specify the width for some reason, you can use the <code>width</code> property in the <code>treeConfig</code> decorator.</td></tr>
</table>

PXTreeItemBinding

The following table shows the correspondence between the `PXTreeItemBinding` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>PXTreeItemBinding</code><pre><px:PXTreeView>
  <DataBindings>
    <px:PXTreeItemBinding
      DataMember="ItemClassNodes"
      TextField="SegmentedClassCD"
      ValueField="ItemClassID"
      DescriptionField="Descr" />
  </DataBindings>
</px:PXTreeView></pre></td><td>Use the properties of the <code>treeConfig</code> decorator, as shown in the following example.<pre>@treeConfig({
  textField: 'SegmentedClassCD',
})
export class ItemClassNodes
 extends PXView {
  ItemClassID: PXFieldState;
  SegmentedClassCD: PXFieldState;
  Descr: PXFieldState;
}</pre>The <code>DescriptionField</code> property from ASPX is not supported in the Modern UI.</td></tr>
</table>

ASPX	HTML or TypeScript

`DataMember`
`<px:PXTreeView>`
`<DataBindings>`
`<px:PXTreeItemBinding`
`DataMember="ItemClassNodes"`
`/>`
`</DataBindings>`
`</px:PXTreeView>`
	Use the `view` attribute of the `qp-tree` control, as the following code shows.
	`<qp-tree`
	`id="tree"`
	`view.bind="ItemClassNodes">`
	`</qp-tree>`

`DescriptionField`
`<px:PXTreeView>`
`<DataBindings>`
`<px:PXTreeItemBinding`
`DescriptionField="Descr"`
`/>`
`</DataBindings>`
`</px:PXTreeView>`
	The `DescriptionField` property from ASPX is not supported in the Modern UI.

`ImageUrlField`
`<px:PXTreeView>`
`<DataBindings>`
`<px:PXTreeItemBinding`
`ImageUrlField="Icon"`
`/>`
`</DataBindings>`
`</px:PXTreeView>`
	Use the `iconField` property of the `treeConfig` decorator.

`TextField`
`<px:PXTreeView>`
`<DataBindings>`
`<px:PXTreeItemBinding`
`TextField="SegmentedClassCD"`
`/>`
`</DataBindings>`
`</px:PXTreeView>`
	Use the `textField` property of the `treeConfig` decorator.

`ToolTipField`
`<px:PXTreeView>`
`<DataBindings>`
`<px:PXTreeItemBinding`
`ToolTipField="ToolTip"`
`/>`
`</DataBindings>`
`</px:PXTreeView>`
	Use the `toolTipField` property of the `treeConfig` decorator.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>ValueField

<pre><code><px:PXTreeView>
  <DataBindings>
    <px:PXTreeItemBinding
      ValueField="ItemClassID"
    />
  </DataBindings>
</px:PXTreeView></code></pre></td><td>Use the valueField property of the treeConfig decorator.</td></tr>
</table>

PXTreeView

The following table shows the correspondence between the PXTreeView element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXTreeView`

```
<px:PXTreeView
  ID="tree"
  runat="server"
  DataSourceID="ds"
  DataMember="ItemClassNodes"
  Height="180px"
  AutoRepaint="True"
  SyncPosition="True"
  SyncPositionWithGraph="True"
  PreserveExpanded="True"
  ExpandDepth="0"
  PopulateOnDemand="True"
  Caption="Item Class Tree"
  ShowRootNode="False"
  AllowCollapse="True">
  <CaptionStyle
    Height="17px" />
  <AutoCallBack
    Command="GoToNodeSelectedInTree"
    Target="ds" />
  <DataBindings>
    <px:PXTreeItemBinding
      DataMember="ItemClassNodes"
      TextField="SegmentedClassCD"
      ValueField="ItemClassID"
      DescriptionField="Descr" />
  </DataBindings>
  <AutoSize Enabled="True" />
</px:PXTreeView>
```

</td><td>

Use the `qp-tree` tag in HTML and the `treeConfig` decorator for a view class in TypeScript.

The following example shows a declaration of a view class for a tree in TypeScript.

```
@treeConfig({
  dynamic: true,
  hideRootNode: true,
  idField: 'ItemClassID',
  textField: 'SegmentedClassCD',
  modifiable: false,
  mode: 'single',
  singleClickSelect: true,
  selectFirstNode: true,
  syncPosition: true,
  openedLayers: 1,
})
export class ItemClassNodes
 extends PXView {
  ItemClassID: PXFieldState;
  SegmentedClassCD: PXFieldState;
  Descr: PXFieldState;
}
```

The `DescriptionField` property from ASPX is not supported in the Modern UI.

The following HTML code includes the tree in the layout of an Acumatica ERP form.

```
<qp-tree id="tree"
  view.bind="ItemClassNodes"
  caption="Item Class Tree">
</qp-tree>
```

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

`AllowCollapse`

```
<px:PXTreeView
  AllowCollapse="True">
</px:PXTreeView>
```

</td><td>

Use the `qp-collapsible` attribute as described in *Collapsible Area: Configuration*.

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

`AutoRepaint`

```
<px:PXTreeView
  AutoRepaint="True" />
```

</td><td>

Use the `autoRepaint` property of the `treeConfig` decorator.

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

ASPX	HTML or TypeScript

`Caption`
`<px:PXTreeView`
`  Caption="Item Class Tree" />`
	Use the `caption` attribute of the `qp-tree` control, as the following code shows.
	`<qp-tree id="tree"`
	`  view.bind="ItemClassNodes"`
	`  caption="Item Class Tree">`
	`</qp-tree>`

`DataMember`
`<px:PXTreeView`
`  DataMember="ItemClassNodes" />`
	Use the `view` attribute of the `qp-tree` control, as the following code shows.
	`<qp-tree`
	`  id="tree"`
	`  view.bind="ItemClassNodes">`
	`</qp-tree>`

`ExpandDepth`
`<px:PXTreeView`
`  ExpandDepth="0" />`
	Use the `openedLayers` property of the `treeConfig` decorator.

`Height`
`<px:PXTreeView`
`  Height="180px" />`
	Do not specify the height of a tree control because it is automatically adjusted for various screen sizes. However, if you need to specify the height for some reason, you can use the `height` property in the `treeConfig` decorator.

`ID`
`<px:PXTreeView`
`  ID="tree" />`
	Use the `id` attribute of the `qp-tree` control, as the following code shows.
	`<qp-tree`
	`  id="tree">`
	`</qp-tree>`

`KeepPosition`
`<px:PXTreeView`
`  KeepPosition= "True" />`
	Use the `keepPosition` property of the `treeConfig` decorator.

`PopulateOnDemand`
`<px:PXTreeView`
`  PopulateOnDemand="True" />`
	Use the `dynamic` property of the `treeConfig` decorator.

`RenderHtmlText`
`<px:PXTreeView`
`  RenderHtmlText="True" >`
`</px:PXTreeView>`
	Use the `renderHTML` property of the `treeConfig` decorator. For more details, see *Tree: Markup of a Tree Node Using HTML*.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>SelectFirstNode</code><pre><px:PXTreeView
  SelectFirstNode="True" /></pre></td><td>Use the <code>selectFirstNode</code> property of the <code>treeConfig</code> decorator.</td></tr>
<tr><td><code>ShowRootNode</code><pre><px:PXTreeView
  ShowRootNode="False" /></pre></td><td>Use the <code>hideRootNode</code> property of the <code>treeConfig</code> decorator.</td></tr>
<tr><td><code>SyncPosition</code><pre><px:PXTreeView
  SyncPosition="True" /></pre></td><td>Use the <code>syncPosition</code> property of the <code>treeConfig</code> decorator.</td></tr>
<tr><td><code>Width</code><pre><px:PXTreeView
  Width="180px" /></pre></td><td>Do not specify the width of a tree control because it is automatically adjusted for various screen sizes. However, if you need to specify the width for some reason, you can use the <code>width</code> property in the <code>treeConfig</code> decorator.</td></tr>
</table>

PXDataSource

The following table shows the correspondence between the `PXDataSource` element, which is used as a container for the `DataTrees` and `PXTreeDataMember` elements, and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXDataSource > KeySeparatorChar`

```
<px:PXDataSource ID="ds"
  ...
  PrimaryView="Filter"
  KeySeparatorChar="!">
  <DataTrees>
    <px:PXTreeDataMember
      TreeView="SiteMapTree"
      TreeKeys="NodeID" />
  </DataTrees>
  <CallbackCommands>
    ...
  </CallbackCommands>
  <ClientEvents
    CommandPerformed="afterCallback"/>
  <DataTrees>
    <px:PXTreeDataMember
      TreeView="FilesTree"
      TreeKeys="FileKey" />
  </DataTrees>
</px:PXDataSource>
```

</td><td>

Use the `keySeparatorChar` property in the `treeConfig` property of the `controlConfig` or `fieldConfig` decorator.

```
@fieldConfig({
  controlType: "qp-tree-selector",
  controlConfig: {
    treeConfig: {
      idField: "FileKey",
      parentIdField: "FileKey",
      valueField: "Path",
      dataMember: "FilesTree",
      textField: "FilePath",
      mode: "single",
      hideRootNode: true
      hideRootNode: true,
      keySeparatorChar: "!",
    }
  }
})
ParentFolder : PXFieldState;
```

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

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the tree control. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

<table>
<tr><th>ASPX Control</th><th>Properties</th></tr>
<tr><td>

`PXTree`

```
<px:PXTree
  runat="server"
  CallbackUpdatable="True"
>
</px:PXTree>
```

</td><td>

- `CallbackUpdatable`
- `runat`

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

`PXTreeView`

```
<px:PXTreeView
  CaptionVisible="False"
  DataSourceID="ds"
  PreserveExpanded="True"
  RootNodeText="BOM"
  runat="server"
  SyncPositionWithGraph="True">
</px:PXTreeView>
```

</td><td>

- `CaptionVisible`
- `DataSourceID`
- `PreserveExpanded`
- `RootNodeText`
- `runat`
- `SyncPositionWithGraph`

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

<table>
<tr><th>ASPX Control</th><th>Properties</th></tr>
<tr><td>AutoCallBack
<pre><px:PXTreeView/>
  <AutoCallBack
    Command="GoToNodeSelectedInTree"
    Target="ds" />
</px:PXTreeView></pre></td><td>All properties</td></tr>
<tr><td>AutoSize
<pre><px:PXTreeView/>
  <AutoSize Enabled="True" />
</px:PXTreeView></pre></td><td>All properties</td></tr>
<tr><td>CaptionStyle
<pre><px:PXTreeView>
  <CaptionStyle
    Height="17px" />
</px:PXTreeView></pre></td><td>All properties</td></tr>
<tr><td>PXTreeDataMember
<pre><px:PXDataSource>
  <DataTrees>
    <px:PXTreeDataMember
      TreeView="ItemClassNodes"
      TreeKeys="ItemClassID" />
  </DataTrees>
</px:PXDataSource></pre></td><td>All properties</td></tr>
</table>

Tree Selector

In this chapter, you will learn how to configure tree selector controls on Acumatica ERP forms. The chapter also includes information about conversion of tree selector controls from ASPX to HTML and TypeScript.

Tree Selector Control: General Information

A tree selector control is a control that allows a user to select a value from a tree view, as shown in the following screenshot. A tree selector control can include two tabs: one tab displays the data in the selector table, another tab displays the data as a tree view.

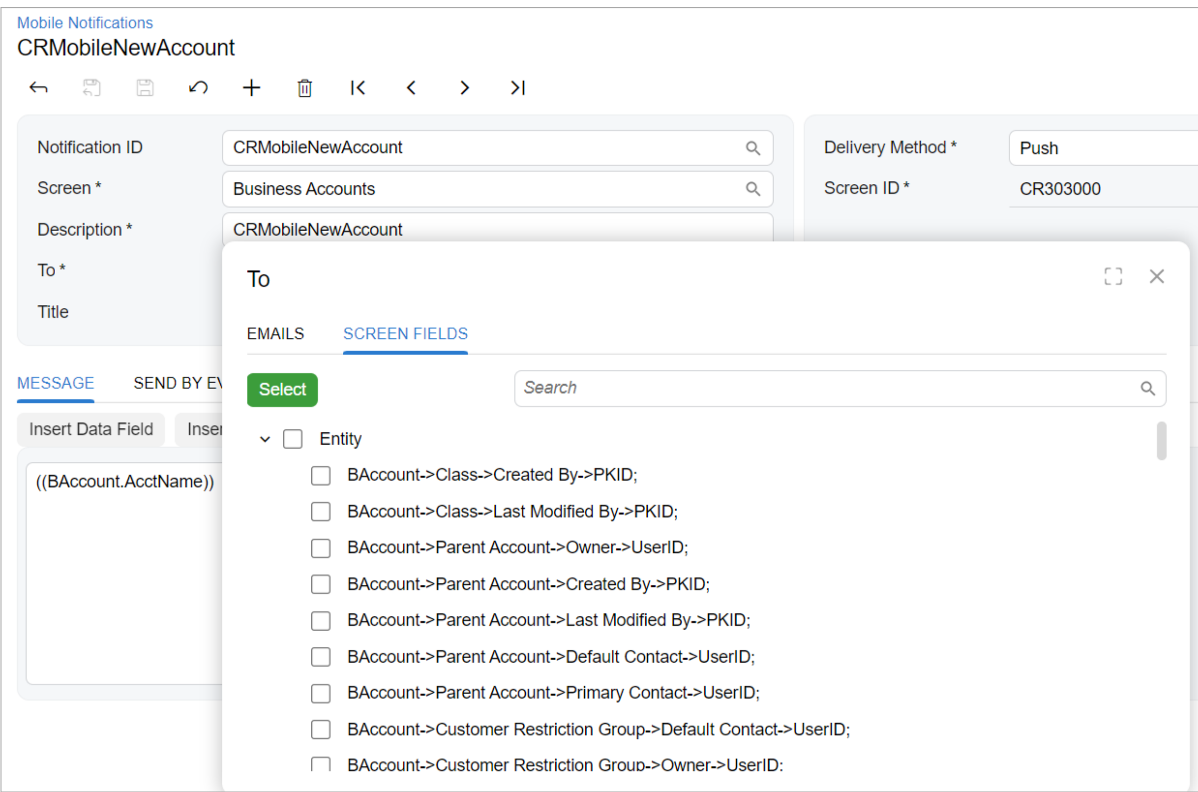


Figure: A tree selector control

In the Classic UI, a tree control with two tabs is defined with the `PXTreeSelectorWithGrid` control, a tree control without tabs is defined with the `PXTreeSelector` control. In the Modern UI, you use the `qp-tree-selector` control to define both kinds of the tree conrol.

Learning Objectives

In this chapter, you will learn about the proper configuration of a tree selector control for specific cases, such as for a tree selector control located in a table cell

Applicable Scenarios

You configure a tree if you need to display a hierarchical data structure, such as an organizational chart, a category structure, or nested menu items, in a box or in a table cell.

Tree Selector Control: Configuration

The configuration properties of the control are defined by the `ITreeSelectorConfig` interface.

Tree Selector Control with Selector Table and Tree View

To display a tree selector control with two tabs, use the `qp-tree-selector` control, as the following example shows.

```
<field name="NTo"
  control-type="qp-tree-selector"
  control-config.bind="{
    treeConfig: {
      idField: 'Key',
      valueField: 'Path',
      datamember: 'ScreenUserItems',
      descriptionField: 'Name',
      iconField: 'Icon',
    },
    selectorConfig: {
      view: 'UserItems',
      key: 'KeyUserName',
      description: 'FullName',
    },
  }">
</field>
```

Tree Selector Control with Only Tree View

To display a tree selector control with only tree view, use the `qp-tree-selector` control, as the following example shows.

```
<field name="Subject"
  control-type="qp-tree-selector"
  control-config.bind="{
    treeConfig: {
      idField: 'Path',
      dataMember: 'ItemsWithPrevious',
      descriptionField: 'Name',
      parentIdField: 'Key',
      iconField: 'Icon',
      mode: 'single'
    },
  }">
</field>
```

Tree Selector Control in a Table Column

To use a tree selector control in a table column, in the `columnConfig` decorator, you define the `editorType` property as `qp-tree-selector` and specify configuration properties in the `editorConfig` property, as shown in the following example.

```
@columnConfig({
  editorType: "qp-tree-selector",
  editorConfig: {
    treeConfig: {
      idField: "Path",
      dataMember: "EntityItemsWithPrevious",
      descriptionField: "Name",
      parentIdField: "Key",
      iconField: "Icon",
      mode: "single"
    }
  }
})
Subject: PXFieldState;
```

Tree Selector Control: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to a tree selector control to HTML or TypeScript elements.

PXTreeSelector

The following table shows the correspondence between the `PXTreeSelector` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXTreeSelector`

```
<px:PXTreeSelector
  ID="edsubject"
  runat="server"
  DataField="Subject"
  TreeDataSourceID="ds"
  PopulateOnDemand="True"
  InitialExpandLevel="0"
  ShowRootNode="False"
  MinDropWidth="468"
  MaxDropWidth="600"
  AllowEditValue="True"
  AppendSelectedValue="True"
  AutoRefresh="True"
  TreeDataMember="ItemsWithPrevious" >
  <DataBindings>
    <px:PXTreeItemBinding
      DataMember="ItemsWithPrevious"
      TextField="Name"
      ValueField="Path"
      ImageUrlField="Icon"
      ToolTipField="Path" />
  </DataBindings>
</px:PXTreeSelector>
```

</td><td>

To display a tree selector control without tabs, use the `qp-tree-selector` control, as the following example shows.

```
<field name="Subject"
  control-type="qp-tree-selector"
  control-config.bind="{
    treeConfig: {
      idField: 'Path',
      dataMember: 'ItemsWithPrevious',
      descriptionField: 'Name',
      parentIdField: 'Key',
      iconField: 'Icon',
      mode: 'single'
    },
  }">
</field>
```

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

ASPX	HTML or TypeScript

`AllowEditValue`
`<px:PXTreeSelector`
`  AllowEditValue="True"/>`
	Use the `allowEditValue` property, which is available through the `config` attribute of the `qp-tree-selector` control.

`AppendSelectedValue`
`<px:PXTreeSelector`
`  AppendSelectedValue="True"/>`
	Use the `appendSelectedValue` property, which is available through the `config` attribute of the `qp-tree-selector` control.

`CommitChanges`
`<px:PXTreeSelector`
`  CommitChanges="True"`
`  DataField="NTo" />`
	Use the `CommitChanges` option of `PXFieldState` in TypeScript, as the following code shows. With this option, the control commits changes to the server each time a user has changed the value in the box and focus is no longer on the box.
	`export class UserItems extends PXView {`
	`  NTo:`
	`    PXFieldState<`
	`    PXFieldOptions.CommitChanges>;`
	`}`

`DataField`
`<px:PXTreeSelector`
`  DataField="NTo" />`
	Use the `name` attribute of the `field` tag in HTML, as the following code shows.
	`<field name="NTo"></field>`

`InitialExpandLevel`
`<px:PXTreeSelector`
`  InitialExpandLevel="0" />`
	Use the `openedLayers` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

`PopulateOnDemand`
`<px:PXTreeSelector`
`  PopulateOnDemand="True" />`
	Use the `dynamic` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

`ShowRootNode`
`<px:PXTreeSelector`
`  ShowRootNode="False" />`
	Use the `hideRootNode` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

`TreeParentId`
`<px:PXTreeSelectorWithGrid`
`  TreeParentId="Path" />`
	Use the `parentIdField` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

PXTreeSelectorWithGrid

The following table shows the correspondence between the PXTreeSelectorWithGrid element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
PXTreeSelectorWithGrid (ASPX code 1)	To display a tree selector control with two tabs, use the qp-tree-selector control, as the following example shows. (HTML code 1)
AllowEditValue (ASPX code 2)	Use the allowEditValue property, which is available through the config attribute of the qp-tree-selector control.
AppendSelectedValue (ASPX code 3)	Use the appendSelectedValue property, which is available through the config attribute of the qp-tree-selector control.

ASPX code 1:

```
<px:PXTreeSelectorWithGrid
  ID="edNTo"
  runat="server"
  DataField="NTo"
  PopulateOnDemand="True"
  TreeDataSourceID="ds"
  CommitChanges="True"
  AllowEditValue="True"
  ShowRootNode="False"
  MinDropWidth="468"
  MaxDropWidth="600"
  InitialExpandLevel="0"
  AppendSelectedValue="True"
  AutoRefresh="True"
  TreeDataMember="ScreenUserItems"
  SelectorDataMember="UserItems"
  SelectorKeyField="KeyUserName"
  SelectorDescriptionField="FullName"
  TreeParentId="Path"
  SelectorTabTitle="Users">
  <DataBindings>
    <px:PXTreeItemBinding
      DataMember="ScreenUserItems"
      TextField="Name"
      ValueField="Key"
      ImageUrlField="Icon"
      ToolTipField="Path" />
  </DataBindings>
</px:PXTreeSelectorWithGrid>
```

HTML code 1:

```
<field name="NTo"
  control-type="qp-tree-selector"
  control-config.bind="{
    treeConfig: {
      idField: 'Key',
      valueField: 'Path',
      datamember: 'ScreenUserItems',
      descriptionField: 'Name',
      iconField: 'Icon',
    },
    selectorConfig: {
      view: 'UserItems',
      key: 'KeyUserName',
      description: 'FullName',
    },
  }">
</field>
```

ASPX code 2:

```
<px:PXTreeSelectorWithGrid
  AllowEditValue="True"/>
```

ASPX code 3:

```
<px:PXTreeSelector
  AppendSelectedValue="True"/>
```

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`CommitChanges`

```
<px:PXTreeSelectorWithGrid
  CommitChanges="True"
  DataField="NTo" />
```

</td><td>

Use the `CommitChanges` option of `PXFieldState` in TypeScript, as the following code shows. With this option, the control commits changes to the server each time a user has changed the value in the box and focus is no longer on the box.

```
export class UserItems extends PXView {
  NTo:
    PXFieldState<
    PXFieldOptions.CommitChanges>;
}
```

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

`DataField`

```
<px:PXTreeSelectorWithGrid
  DataField="NTo" />
```

</td><td>

Use the `name` attribute of the `field` tag in HTML, as the following code shows.

```
<field name="NTo"></field>
```

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

`InitialExpandLevel`

```
<px:PXTreeSelectorWithGrid
  InitialExpandLevel="0" />
```

</td><td>

Use the `openedLayers` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

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

`PopulateOnDemand`

```
<px:PXTreeSelectorWithGrid
  PopulateOnDemand="True" />
```

</td><td>

Use the `dynamic` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

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

`SelectorDataMember`

```
<px:PXTreeSelectorWithGrid
  SelectorDataMember="UserItems" />
```

</td><td>

Use the `view` property in the `selectorConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

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

`SelectorDescriptionField`

```
<px:PXTreeSelectorWithGrid
  SelectorDescriptionField="FullName" />
```

</td><td>

Use the `description` property in the `selectorConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

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

`SelectorKeyField`

```
<px:PXTreeSelectorWithGrid
  SelectorKeyField="KeyUserName" />
```

</td><td>

Use the `key` property in the `selectorConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

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

`SelectorTabTitle`

```
<px:PXTreeSelectorWithGrid
  SelectorTabTitle="Users" />
```

</td><td>

Use the `selectorTabTitle` property, which is available through the `config` attribute of the `qp-tree-selector` control.

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

ASPX	HTML or TypeScript

`ShowRootNode`
 	
`<px:PXTreeSelectorWithGrid`
`  ShowRootNode="False" />`
	Use the `hideRootNode` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

`TreeParentId`
 	
`<px:PXTreeSelectorWithGrid`
`  TreeParentId="Path" />`
	Use the `parentIdField` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control.

PXTreeItemBinding

The following table shows the correspondence between the `PXTreeItemBinding` element, which is used inside the `PXTreeSelector` or `PXTreeSelectorWithGrid` element, and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

`PXTreeItemBinding`
 	
`<px:PXTreeSelector>`
`  <DataBindings>`
`    <px:PXTreeItemBinding`
`      DataMember="ItemClassNodes"`
`      TextField="SegmentedClassCD"`
`      ValueField="ItemClassID"`
`      DescriptionField="Descr" />`
`  </DataBindings>`
`</px:PXTreeSelector>`
	Use `treeConfig` property, which is available through the `config` attribute of the `qp-tree-selector` control.

`DataMember`
 	
`<px:PXTreeSelector>`
`  <DataBindings>`
`    <px:PXTreeItemBinding`
`      DataMember="ScreenUserItems"`
`    />`
`  </DataBindings>`
`</px:PXTreeSelector>`
	Use the `dataMember` property in the `treeConfig` configuration property, which is available through the `config` attribute of the `qp-tree-selector` control. See an example in the following code.
	
	`<field name="NTo"`
	`  control-type="qp-tree-selector"`
	`  control-config.bind="{`
	`    treeConfig: {`
	`      dataMember: 'ScreenUserItems'`
	`    }`
	`  }">`
	`</field>`

ASPX	HTML or TypeScript
DescriptionField <px:PXTreeSelector> <DataBindings> <px:PXTreeItemBinding DescriptionField="Descr" /> </DataBindings> </px:PXTreeSelector>	Use the descriptionField property in the treeConfig configuration property, which is available through the config attribute of the qp-tree-selector control.
ImageUrlField <px:PXTreeSelector> <DataBindings> <px:PXTreeItemBinding ImageUrlField="Icon" /> </DataBindings> </px:PXTreeSelector>	Use the iconField property in the treeConfig configuration property, which is available through the config attribute of the qp-tree-selector control.
TextField <px:PXTreeSelector> <DataBindings> <px:PXTreeItemBinding TextField="SegmentedClassCD" /> </DataBindings> </px:PXTreeSelector>	Use the textField property in the treeConfig configuration property, which is available through the config attribute of the qp-tree-selector control.
ToolTipField <px:PXTreeSelector> <DataBindings> <px:PXTreeItemBinding ToolTipField="ToolTip" /> </DataBindings> </px:PXTreeSelector>	Use the toolTipField property in the treeConfig configuration property, which is available through the config attribute of the qp-tree-selector control.
ValueField <px:PXTreeSelector> <DataBindings> <px:PXTreeItemBinding ValueField="ItemClassID" /> </DataBindings> </px:PXTreeSelector>	Use the valueField property in the treeConfig configuration property, which is available through the config attribute of the qp-tree-selector control.

PXDataSource

The following table shows the correspondence between the `PXDataSource` element, which is used as a container for the `DataTrees` and `PXTreeDataMember` elements, and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td>

`PXDataSource > KeySeparatorChar`

```
<px:PXDataSource ID="ds"
  ...
  PrimaryView="Filter"
  KeySeparatorChar="!">
  <DataTrees>
    <px:PXTreeDataMember
      TreeView="SiteMapTree"
      TreeKeys="NodeID" />
  </DataTrees>
  <CallbackCommands>
    ...
  </CallbackCommands>
  <ClientEvents
    CommandPerformed="afterCallback"/>
  <DataTrees>
    <px:PXTreeDataMember
      TreeView="FilesTree"
      TreeKeys="FileKey" />
  </DataTrees>
</px:PXDataSource>
```

</td><td>

Use the `keySeparatorChar` property in the `treeConfig` property of the `controlConfig` or `fieldConfig` decorators.

```
@fieldConfig({
  controlType: "qp-tree-selector",
  controlConfig: {
    treeConfig: {
      idField: "FileKey",
      parentIdField: "FileKey",
      valueField: "Path",
      dataMember: "FilesTree",
      textField: "FilePath",
      mode: "single",
      hideRootNode: true
      hideRootNode: true,
      keySeparatorChar: "!",
    }
  }
})
ParentFolder : PXFieldState;
```

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

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to a tree selector control. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXTreeSelector`	• `AutoRefresh` • `ID` • `MaxDropWidth` • `MinDropWidth` • `TreeDataMember` • `TreeDataSourceID`

ASPX Control	Properties
PXTreeSelectorWithGrid	• AutoRefresh • ID • MaxDropWidth • MinDropWidth • TreeDataMember • TreeDataSourceID

Upload Dialog Box

In this chapter, you will learn about the configuration of upload dialog boxes—that is, pop-up panels or windows where a user can select and upload a file. You will learn when to use upload dialog boxes, how to name them, and how to organize their layout.

Upload Dialog Box: General Information

An upload dialog box is a pop-up panel or window where a user can select and upload a file. An upload dialog box opens when a user clicks a button or a link. This dialog box looks like a modal pop-up panel where a user can select a file from the computer file system, leave a comment, and click **Upload** or **Cancel**. The following screenshot shows an upload dialog box.

Figure: An upload dialog box

Learning Objectives

In this chapter, you will learn the following about an upload dialog box:

- The design guidelines for the upload dialog box, including the naming conventions and layout recommendations
- The proper configuration of the upload dialog box for specific cases

Applicable Scenarios

You configure the upload dialog box when you want a user to upload a file whose contents can be loaded into the form.

Users can also attach a file to a record by clicking the **Files** button on the form title bar of the form. With this approach, the contents of the file cannot be processed in code.

Upload Dialog Box ID

The ID of an upload dialog box in HTML consists of two parts: the `dlg` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a dialog box that uploads a ZIP package may have the `dlgUploadPackage` ID, as the following code shows.

```
<qp-upload-dialog id="dlgUploadPackage"></qp-upload-dialog>
```

UI Naming Convention

The following table shows the UI naming convention for the upload dialog box.

Naming Convention	Example
Ideally, the name of the button that opens the dialog box should be identical to the caption of the dialog box. If the caption is too long, the button name may be a short version of the caption.	The name of the dialog box is **Upload New File Version**. The name of the button that opens the dialog box may be either **Upload New File Version** or the shorter **Upload File Version**.

Upload Dialog Box: Configuration of an Upload Dialog Box

You define the upload dialog box in HTML by using the `qp-upload-dialog` tag.

The configuration properties of the `qp-upload-dialog` control are stored in the `IQpUploadDialogConfig` interface.

An example of defining a file upload dialog in HTML is shown in the following code.

```
<qp-upload-dialog
  id="pnlNewRev"
  key="NewRevisionPanel"
  session-key="sessionKey_fileMaintenance"
  caption="File Upload"
  render-link="true">
</qp-upload-dialog>
```

To display the dialog box, you need to invoke the dialog box in the graph code (in an action delegate) by doing the following:

1. Calling the `Ask` or `AskExt` method of the view or the `DialogManager` class
2. Specifying the dialog box properties in the method parameters.

The following code shows the invocation of the file upload dialog defined above in the graph code.

```
public IEnumerable uploadNewVersion(PXAdapter a)
{
  var askResult = DialogManager.AskExt(a.View.Graph,
                                       "NewRevisionPanel", null, null, false);
  if (askResult == WebDialogResult.OK)
  {
    var info = PXContext.SessionTyped<PXSessionStatePXData>()
      .FileInfo["sessionKey_fileMaintenance"];
    if (info != null)
    {
      this.NewRevision(info, info.CheckIn);
    }
  }

  return a.Get();
}
```

Upload Dialog Box: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the upload dialog box to HTML or TypeScript elements.

PXUploadDialog

The following table shows the correspondence between the PXUploadDialog element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

PXUploadDialog
 	
<px:PXUploadDialog ID="pnlNewRev"
Key="NewRevisionPanel"
runat="server" Height="120px"
Style="position: static"
Width="560px"
Caption="Statement File Upload"
AutoSaveFile="false"
RenderCheckIn="false"
SessionKey="ImportStatementProtoFile" />
	To display an upload dialog box, use the qp-upload-dialog tag in the HTML template.
	
	<qp-upload-dialog id="pnlNewRev"
	  key="NewRevisionPanel"
	  session-key="ImportStatementProtoFile"
	  caption="Statement File Upload"
	  height="120px"
	  width="560px"
	  autoSaveFile="false"
	  renderCheckIn="false">
	</qp-upload-dialog>

ID <px:PXUploadDialog ID="pnlNewRev" ... />	Use the id property, which is available in the config attribute of the qp-upload-dialog control. <qp-upload-dialog id="pnlNewRev" ...>
Key <px:PXUploadDialog ... Key="NewRevisionPanel" ... />	Use the key property, which is available in the config attribute of the qp-upload-dialog control. <qp-upload-dialog ... key="NewRevisionPanel" ...>
SessionKey <px:PXUploadDialog ... SessionKey="ImportStatementProtoFile" />	Use the sessionKey property, which is available in the config attribute of the qp-upload-dialog control. <qp-upload-dialog ... session-key="ImportStatementProtoFile">
Caption <px:PXUploadDialog ... Caption="Statement File Upload" />	Use the caption property, which is available in the config attribute of the qp-upload-dialog control. <qp-upload-dialog ... caption="Statement File Upload">

ASPX	HTML or TypeScript

`Width`
 	
`<px:PXUploadDialog ...`
`Width="560px" />`
	Use the `width` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	
	`<qp-upload-dialog ...`
	`width="560px">`

`AllowedFileTypes`
 	
`<px:PXUploadDialog ...`
`AllowedFileTypes=".als" />`
	Use the `allowedTypes` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	
	`<qp-upload-dialog ...`
	`allowed-types=".als">`

`IgnoreSize`
 	
`<px:PXUploadDialog ...`
`IgnoreSize="true"/>`
	Use the `ignoreSize` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	
	`<qp-upload-dialog ...`
	`ignore-size="true">`

`AutoSaveFile`
 	
`<px:PXUploadDialog ...`
`AutoSaveFile="false" />`
	Use the `autoSave` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	
	`<qp-upload-dialog ...`
	`auto-save="true">`

`NamePrefix`
 	
`<px:PXUploadDialog ...`
`NamePrefix="ar" />`
	Use the `namePrefix` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	
	`<qp-upload-dialog ...`
	`name-prefix="ar">`

`RenderLink`
 	
`<px:PXUploadDialog ...`
`RenderLink="true" />`
	Use the `renderLink` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	
	`<qp-upload-dialog ...`
	`render-link="true">`

`RenderComment`
 	
`<px:PXUploadDialog ...`
`RenderComment="true" />`
	Use the `renderComment` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	
	`<qp-upload-dialog ...`
	`render-comment="true">`

ASPX	HTML or TypeScript

`RenderImportOptions`
`<px:PXUploadDialog ...`
`RenderImportOptions="true" />`
	Use the `renderImportOptions` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	`<qp-upload-dialog ...`
	`render-import-options="true">`

`RenderCheckIn`
`<px:PXUploadDialog ...`
`RenderCheckIn="true" />`
	Use the `renderCheckIn` property, which is available in the `config` attribute of the `qp-upload-dialog` control.
	`<qp-upload-dialog ...`
	`render-check-in="true">`

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the upload dialog box. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXUploadDialog`	• `runat`

Upload Files Button

In this chapter, you will learn about the configuration of the upload files button control, which is used to select and upload any number of files to a record. You will learn when to use this control and how to use its properties.

Upload Files Button: General Information

The upload files button control provides users with the ability to upload any number of files to a record. This control functions as both a button you can click and an area where files can be dropped. A user can upload the files either by dragging them onto the button or via the file upload dialog box, which opens when the button is clicked.

The following screenshot shows an example of the upload files button control in the **Files** dialog box. This dialog box opens when a user clicks **Files** on the form title bar of a form. In this example, the control is assigned the **Upload Files** name.

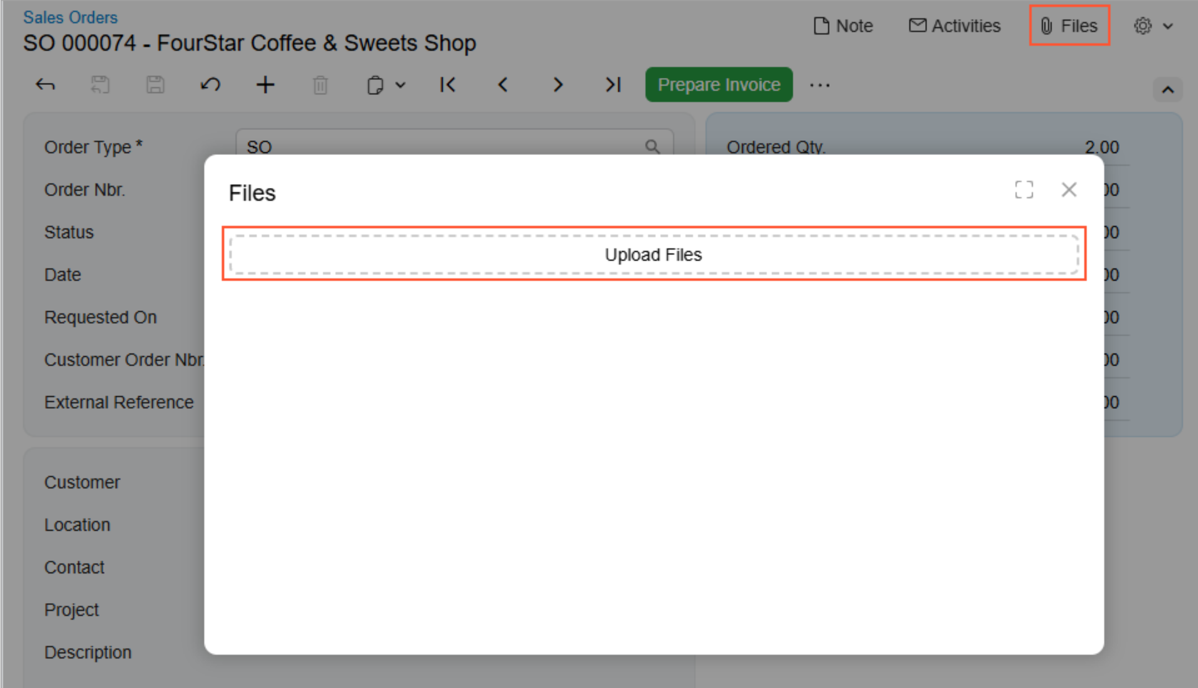


Figure: The Upload Files button in the Files dialog box

An upload files button control is defined by `PXFileUploader` in the Classic UI. In the Modern UI, an upload files button control is defined by the `qp-multi-upload` tag.

Learning Objectives

In this chapter, you will learn the following about an upload files button control:

- The proper configuration of the control
- How to convert the ASPX elements of this control to HTML or TypeScript

Applicable Scenarios

You configure the upload files button control when you want to give users the ability to upload any number of files to a record.

Configuration of an Upload Files Button

You configure the upload files button control in HTML by using the `qp-multi-upload` tag.

The configuration properties of the `qp-multi-upload` control are stored in the `IMultiUploaderConfig` interface.

The following code shows an example of defining an upload files button control in HTML.

```
<qp-multi-upload
   config.bind="{
        id: 'uploadFilesControl',
        graph: 'MyPhotoLogEntry',
        view: 'Photos',
        action: 'UploadFiles',
        dropTarget: '#gridPhotos',
        autoRepaint: true,
   }"></qp-multi-upload>
```

Upload Files Button: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the upload files button control to HTML or TypeScript elements.

PXFileUploader

The following table shows the correspondence between the `PXFileUploader` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

<table>
<tr><th>ASPX</th><th>HTML or TypeScript</th></tr>
<tr><td><code>PXFileUploader</code>
<pre><code><px:FileUploader ID="fileUploaderControl"
runat="server" DataMember="uploadFiles" /></code></pre></td><td>To attach files to a record, use the <code>qp-multi-upload</code> tag in the HTML template.
<pre><code><qp-multi-upload config.bind="{
  id: 'uploadFilesControl',
  action: 'attachFiles',
  accept: '.gif,.jpg'
}"></qp-multi-upload></code></pre></td></tr>
<tr><td><code>ID</code>
<pre><code><px:PXFileUploader
ID="uploadFilesDialog" ... /></code></pre></td><td>Use the <code>id</code> property, which is available in the <code>config</code> attribute of the <code>qp-multi-upload</code> control.
<pre><code><qp-multi-upload
id="uploadFilesControl" ...>
</qp-multi-upload></code></pre></td></tr>
</table>

ASPX	HTML or TypeScript

`AllowedTypes`
`<px:PXFileUploader  ...`
`AllowedTypes=".jpg" />`
	Use the `accept` property, which is available in the `config` attribute of the `qp-multi-upload` control.
	`<qp-multi-upload ...`
	`accept=".jpg">`
	`</qp-multi-upload>`

`SelectFileMessage`
`<px:PXFileUploader  ...`
`SelectFileMessage="Upload Files Here" />`
	Neither TypeScript nor HTML provide an equivalent property that can be used to customize the text that is displayed on the `qp-multi-upload` control. However, you can specify this text between the opening tag and the closing tag of the control, as shown in the following code example.
	`<qp-multi-upload ...>`
	`Upload Files Here`
	`</qp-multi-upload>`

Wizard

In this chapter, you will learn about the configuration of wizards. A wizard is a control that users can see on an Acumatica ERP form to be guided through a process, thus simplifying its completion. The user enters values and follows instructions on the steps (also referred to as a *page*) of the wizards. You will learn when to use wizards, how to name them, and how to organize their layout.

Wizard: General Information

You can configure a wizard control to divide multiple controls into several steps and make it easier for a user to provide values. In this topic, you will learn about the wizard control, its components, and its configuration.

Learning Objectives

In this chapter, you will learn the following about the wizard control:

- The design guidelines for the wizard control, including the naming conventions and layout recommendations
- The proper configuration of the wizard control for specific cases

Applicable Scenarios

You configure the wizard control when you want to a user to provide values for multiple fields and guide a user through a sequence of discrete steps.

Overview of the Wizard Control

A wizard is a control that can consist of one step or multiple steps, each of which focuses on a part of a larger process. Each step may include instructions or prompts and a set of controls, such as boxes and tables, where users can view and enter data. A user can navigate between steps by clicking buttons at the bottom of the wizard.

The wizard control includes the following components:

- A title at the top of the wizard (Item 1 in the screenshot)
- A set of steps; Item 2 in the screenshot below shows the first step of the wizard. The wizard can display only one step at a time.
- The bottom toolbar with buttons (Item 3).

  A toolbar contains the following buttons depending on the step of the wizard:

  - **Cancel**: This button is active on each step of the wizard.
  - **Prev**: This button is active on each step of the wizard except the first one.
  - **Next**: This button is active on each step of the wizard except the last one.
  - **Done**: This button is displayed only on the last step of the wizard.

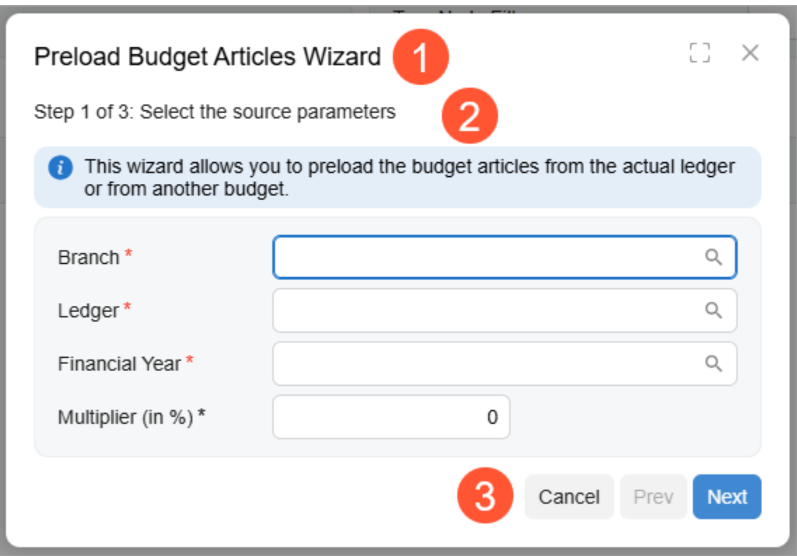


Figure: A wizard control

A wizard is defined by `PXWizard` in the Classic UI. In the Modern UI, a wizard can be defined by the *qp-wizard* control.

To define each step of the wizard, you should use the *qp-tab* control inside the *qp-wizard* control.

Configuration of the Wizard Control

A wizard control is typically displayed inside a dialog box, but it can also be displayed in a container control of a form, such as a tab or the Summary area

If you need to define a wizard inside a dialog box, you should first define the dialog box. You do this by defining the action that opens the dialog box and adding the *qp-panel* tag in HTML. For more details, see *Dialog Box: Opening a Dialog Box* and *Dialog Box: General Information*.

To define a wizard control, do the following:

1. In TypeScript, define the configuration of the wizard control by using the properties of the *IWizardConfig* interface.

   The following code shows a configuration that specifies the title of the wizard and an action for the **Next** button of the wizard. For details, see *Wizard: Configuration of Buttons*.

   ```
   export class GL302010 extends PXScreen {
     wizardConfig: IWizardConfig = {
       caption: ImportMessages.WizardCaption,
       nextCommand: "wizardNext"
     };
   }
   ```

   You can also define the properties from the `IWizardConfig` interface in HTML, as shown below.

   ```
   <qp-wizard ... caption="Preload Budget Articles">
   ```

2. In TypeScript, define views for the controls that are displayed on different steps of the wizard.

3. In HTML, add the `qp-wizard` tag. In the `config.bind` property, specify the configuration from Instruction 1.

   The following code shows how to specify the configuration that was defined in TypeScript.

```
<qp-wizard id="wizardBudgetArticles" config.bind="wizardConfig">
</qp-wizard>
```

If you need to define a wizard inside a dialog box, put the `qp-wizard` tag inside the `qp-panel` tag, which corresponds to a dialog box. The caption specified for the `qp-panel` tag is the title of the wizard.

4. In the `qp-wizard` tag, define the layout of each step in a *`qp-tab`* tag, as shown in the following code.

   To specify the title of the step, use the `caption` attribute of the `qp-tab` tag. The "Step X of Y:" prefix is calculated and added automatically to the caption string.

```
<qp-tab id="tabWizardBudgetArticles-SourceParameters"
  caption="Select the source parameters"">
  <qp-info-box caption="This wizard allows you to ..." type="info"></qp-info-box>
  <qp-template id="formWizardBudgetArticles-SourceParameters" name="1">
    <qp-fieldset slot="A" id="panelWizard-SourceParameters" view.bind="PreloadFilter">
      <field name="branchID"></field>
      ...
    </qp-fieldset>
  </qp-template>
</qp-tab>
```

   The overall structure of the `qp-wizard` control in HTML can look as shown in the following code.

```
<qp-wizard ...>
  <qp-tab ... > <!-- step 1 --> </qp-tab>
  <qp-tab ... > <!-- step 2 --> </qp-tab>
  <qp-tab ... > <!-- step 3 --> </qp-tab>
</qp-wizard>
```

Wizard and Step IDs

An ID of a wizard in HTML consists of two parts: the `wizard` prefix and the semantic name. The semantic name describes the purpose of the element. For example, a wizard that helps users preload budget articles may have the `wizardBudgetArticles` ID, as the following code shows.

```
<qp-wizard id="wizardBudgetArticles" config.bind="wizardConfig">
```

Each step of the wizard that you implement by using the *`qp-tab`* tag can also have an ID. An ID of a tab consists of the following parts:

1. The `tab` prefix.
2. The ID of the wizard.
3. A hyphen.
4. The semantic name of the step. This name describes the contents of the step.

For example, a tab that step with the source parameters of a budget article may have the `tabWizardBudgetArticles-SourceParameters` ID, as the following code shows.

```
<qp-wizard id="wizardBudgetArticles" config.bind="wizardConfig">
```

```
  <qp-tab id="tabWizardBudgetArticles-SourceParameters">
  </qp-tab>
</qp-wizard>
```

Related Links

- *Dialog Box*
- *Tab*
- *Class QpWizardCustomElement (qp-wizard)*
- *Interface IWizardConfig*

Wizard: Configuration of Buttons

A wizard has a footer with the following default buttons:

- **Cancel**
- **Prev**
- **Next**
- **Save**

You can configure these buttons in the following ways:

- Override the actions behind these buttons.

  For this purpose, use the `***Command` properties in the `config` attribute of the *`qp-wizard`* tag. In these properties, you specify the name of an action defined in a graph. The action specified in the `config` attribute will be performed before the default behavior for the button, such as switching to the next step of the wizard or saving the entered data.
- Override the default text displayed on the buttons.

  For this purpose, use the `buttons` array in the `config` attribute of the *`qp-wizard`* tag.
- Disable the **Next** button at runtime.

  For this purpose, use the `disableNext` property in the `config` attribute of the *`qp-wizard`* tag.

For reference of the `config` properties, see the *`IWizardConfig`* interface.

Overriding The Next Button

Suppose that you need to invoke an action that is performed when a user clicks **Next** in the wizard. For example, you need to perform additional checks when a user switches to the next step. Do the following:

1. In a graph, implement the action that should be performed when a user switches to the next step.

   An example is shown in the following code.

   ```
   public PXAction<BudgetFilter> WNext;
   [PXButton]
   [PXUIField(MapEnableRights = PXCacheRights.Update, Visible = false)]
   public virtual IEnumerable wnext(PXAdapter adapter)
   {
     bool errorHappened = false;
     if (PreloadFilter.Current.LedgerID == null)
     {
       PreloadFilter.Cache.
         RaiseExceptionHandling<BudgetPreloadFilter.ledgerID>(PreloadFilter.Current,
           PreloadFilter.Current.LedgerID,
           new PXSetPropertyException(ErrorMessages.FieldIsEmpty,
    PXErrorLevel.RowError));
   ```

```
      errorHappened = true;
    }
    if (errorHappened)
    {
      return adapter.Get();
    }
    return adapter.Get();
  }
```

2. In TypeScript, define the configuration of a wizard, and specify the name of the action from the graph.

   The following code shows how to specify the name of the action in the wizard configuration.

```
@graphInfo({ graphType: "PX.Objects.GL.GLBudgetEntry", primaryView: "Filter",
  pageLoadBehavior: PXPageLoadBehavior.PopulateSavedValues, bpEventsIndicator: true })
export class GL302010 extends PXScreen {
  wizardConfig: IWizardConfig = {
    nextCommand: "wNext"
  };
}
```

You do not need to map the action in the view or screen class in TypeScript.

Related Links

- *Class QpWizardCustomElement (qp-wizard)*
- *Interface IWizardConfig*

Wizard: Conversion from ASPX to HTML and TypeScript

The following tables will help you to convert the ASPX elements that are related to the wizard control to HTML or TypeScript elements.

PXWizard

The following table shows the correspondence between the `PXWizard` element and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript
`PXWizard` `<px:PXWizard ID="PXWizard1" runat="server" Width="400" Height="240" DataMember="PreloadFilter" SkinID="Flat">`	Use the *`qp-wizard`* tag in HTML. `<qp-wizard id="wizardBudgetArticles" config.bind="wizardConfig">`

ASPX	HTML or TypeScript

`ID`
 	
`<px:PXWizard ID="PXWizard1">`
	Use the `id` property of the *`qp-wizard`* tag.
	
	`<qp-wizard`
	`  id="wizardBudgetArticles">`

`DataMember`
 	
`<px:PXWizard`
`  DataMember="PreloadFilter">`
	Instead of specifying a single view for the whole wizard, specify views for the controls in each step of the wizard—that is, the `qp-tab` tags.
	
	`<qp-tab ...>`
	`  <qp-template>`
	`    <qp-fieldset ... view.bind="Preload-`
	`Filter">`
	`    </qp-fieldset>`
	`  </qp-tempalte>`
	`</qp-tab>`

NextCommand, PrevCommand, CancelCommand, and SaveCommand

The following table shows the correspondence between the actions associated with buttons (`NextCommand`, `PrevCommand`, `CancelCommand`, and `SaveCommand`) and the HTML or TypeScript elements. During the conversion of ASPX pages to HTML and TypeScript, you need to replace these ASPX elements with their analogs in HTML or TypeScript.

ASPX	HTML or TypeScript

• `NextCommand`
• `PrevCommand`
• `CancelCommand`
• `SaveCommand`
 	
`<NextCommand`
`  Target="ds" Command="wNext" />`
	To override the standard actions, do the following:
	1. In TypeScript, map the actions defined in the graph.
	2. Define a property that implements the *`IWizardConfig`* interface, and specify the action in the following properties of the `IWizardConfig` interface:
	  • `nextCommand`
	  • `prevCommand`
	  • `cancelCommand`
	  • `saveCommand`
	
	`wizardConfig: IWizardConfig = {`
	`  nextCommand: "wizardNext"`
	`};`
	
	3. In HTML, specify the configuration property in the `config.bind` attribute of the *`qp-wizard`* tag.
	
	`<qp-wizard`
	`  config.bind="wizardConfig">`

Obsolete ASPX Controls and Properties

The following table lists the obsolete ASPX elements that are related to the wizard control. You do not need to replace these ASPX elements with any HTML or TypeScript elements.

ASPX Control	Properties
`PXWizard`	• `runat` • `SkinID`
`Pages`	The entire control is obsolete. The steps of the wizard are defined by a set of `qp-tab` tags.
`PXWizardPage`	The entire control is obsolete. For each step of the wizard, use the `qp-tab` tag.
• `NextCommand` • `PrevCommand` • `CancelCommand` • `SaveCommand`	`Target`

Caption of the Wizard and Steps

The following table lists the correspondence between the ASPX elements that were used to specify the caption of the wizard and its step and the properties in the Modern UI.

ASPX Control	Properties

`PXSmarPanel` > `Caption`
`<px:PXSmartPanel`
`  Caption="Preload Budget Articles Wizard">`
	To specify the caption of the wizard, use the `caption` property of the *`IWizardConfig`* interface.
	`@localizable`
	`class WizardCaptions {`
	`  static Caption = "Preload Budget Articles";`
	`}`
	
	`wizardConfig: IWizardConfig = {`
	`  caption: WizardCaptions.Caption`
	`};`

`PXLabel`
`<px:PXWizardPage><Template>`
`  <px:PXFormView ...><Template>>`
`    <px:PXLabel runat="server" ID="lblStep1"`
`      Width="370px" Style="font-weight: bold;`
`      text-align: right">Step 1 of 3</px:PXLabel>`
	To specify the caption of a step, use the `caption` attribute of the `qp-tab` tag. Do not specify the *Step N of M* prefix; it is calculated and added automatically to the provided caption.
	`<qp-tab`
	`  id="tabWizardBudgetArticles-SourceParameters"`
	`  caption="Select the source parameters">`