PLUG-IN DEVELOPMENT GUIDE

DEVELOPER GUIDE

Acumatica ERP 2020 R2

Acumatica

Contents

Copyright **3**
Plug-In Development Guide **4**
Creating Widgets for Dashboards **5**
Widget Creation 5
Use of the Widgets 6
To Create a Simple Widget 9
To Create an Inquiry-Based Widget 11
To Load a Widget Synchronously or Asynchronously 13
To Add a Script to a Widget 14
To Add Custom Controls to the Widget Properties Dialog Box 15
Implementing Plug-Ins for Processing Credit Card Payments **17**
Interfaces for Processing Credit Card Payments 17
To Implement a Plug-In for Processing Credit Card Payments 18

Copyright

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: 2020 R2

Plug-In Development Guide

In this guide, you can find information about how to expand Acumatica ERP functionality by developing custom plug-ins.

In This Guide

- *Creating Widgets for Dashboards*
- *Implementing Plug-Ins for Processing Credit Card Payments*

Creating Widgets for Dashboards

In Acumatica ERP or an Acumatica Framework-based application, a *dashboard* is a collection of widgets that are displayed on a single page to give you information at a glance. A *widget* is a small component that delivers a particular type of information. Acumatica ERP and any Acumatica Framework-based application support a number of predefined types of widgets that you can add to dashboards. For more information on the supported widgets and their types, see *Dashboards and Widgets*.

If none of the predefined widget types suits your task, you can create custom widgets, as the topics in this chapter describe, and use these widgets in the dashboards of Acumatica ERP or an Acumatica Framework-based application.

In This Chapter

- *Widget Creation*
- *Use of the Widgets*
- *To Create a Simple Widget*
- *To Create an Inquiry-Based Widget*
- *To Load a Widget Synchronously or Asynchronously*
- *To Add a Script to a Widget*
- *To Add Custom Controls to the Widget Properties Dialog Box*

Widget Creation

When you create a custom widget to be used in Acumatica ERP or an Acumatica Framework-based application, you must implement at least the three classes described in the sections below. (For details on the implementation of the classes, see *To Create a Simple Widget* and *To Create an Inquiry-Based Widget*.)

Data Access Class with the Widget Parameters

The data access class (DAC) with the widget parameters is a general DAC that implements the `IBqlTable` interface.

You can use any DAC attributes with this DAC. However, only the fields with `PXDB` attributes, such as `PXDBString` and `PXDBInt`, are stored in the database. The values of these fields are stored in the database automatically; you do not need to create a database table for them.

The fields of the DAC are displayed as controls in the **Widget Properties** dialog box when a user adds a new widget instance to a dashboard or modifies the parameters of an existing widget instance. (The way the controls are displayed depends on *PXFieldState* of the corresponding fields.) For information on adding comprehensive controls to the dialog box, see *To Add Custom Controls to the Widget Properties Dialog Box*.

Graph for the Widget

The graph for the widget is used to manage the widget parameters and read data for the widget. The graph must be inherited from the `PX.Dashboards.WizardMaint` class. For the widget graph, you can manage widget parameters by defining new events and redefining the events that are available in the

base classes, such as the `RowSelected` and `FieldSelecting` events. (You work with the events of the widget graph in the same way as you work with the events of a general graph.)

Class of the Widget

The widget class is used by the system to work with the widget instances. The widget class must implement the `PX.Web.UI.IDashboardWidget` interface. The system treats as widgets the classes that implement this interface and are available in a library in the `Bin` folder of an instance of Acumatica ERP or Acumatica Framework-based application. The caption and description of the widget from these classes are displayed in the **Add Widget** dialog box automatically when a user adds a new widget to a dashboard. The methods of this class are used by the system to manage the parameters of a widget instance and display the widget on a dashboard page.

Related Links

- *To Create a Simple Widget*
- *To Create an Inquiry-Based Widget*

Use of the Widgets

In this topic, you will learn how a custom widget is used in Acumatica ERP or an Acumatica Framework-based application. For more information on how to work with widgets on a dashboard page, see *Personalizing Dashboards*.

How the Widget Is Detected in an Application

Once the library that contains the class of the widget is placed in the `Bin` folder of an instance of Acumatica ERP or an Acumatica Framework-based application, the **Add Widget** dialog box, which you open to add a new widget to a dashboard page, contains the caption and description of the new widget. The system takes the caption and description of the widget from the implementation of the `Caption` and `Description` properties of the `IDashboardWidget` interface.

The following diagram illustrates the relations of the class of the widget and the **Add Widget** dialog box. In the diagram, the items that you add for the custom widget are shown in rectangles with a red border.

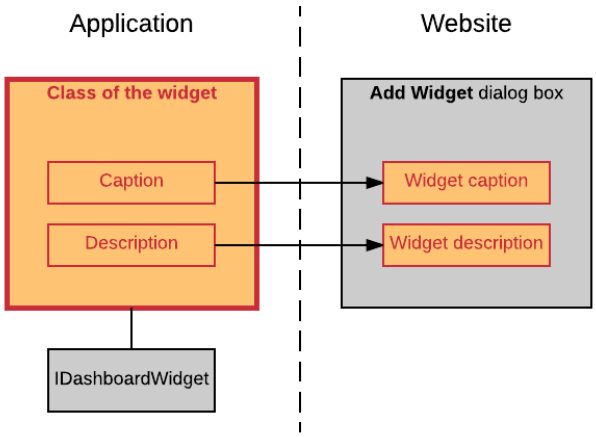

Figure: Widget detection

How a Widget Instance Is Configured

When you select the widget in the **Add Widget** dialog box and click **Next**, the controls for the parameters that should be configured for a new widget instance—that is, the predefined **Caption** box and the controls for the parameters, which are defined in the custom DAC—are displayed in the **Widget Properties** dialog box. These controls are maintained by using the graph for the widget, to which the reference is provided by the class of the widget. The graph uses the `Widget` DAC and the custom DAC with the widget parameters to save the parameters of a widget instance in the `Widget` table of the application database.

To display custom controls, such as buttons and pop-up panels, in the **Widget Properties** dialog box, the system uses the implementations of the `RenderSettings()` and `RenderSettingsComplete()` methods of the `IDashboardWidget` interface from the class of the widget. For more information on adding custom controls to the dialog box, see *To Add Custom Controls to the Widget Properties Dialog Box*.

The following diagram illustrates the interaction of the components of the widget with the **Widget Properties** dialog box. In the diagram, the items that you add for the custom widget are shown in rectangles with a red border.

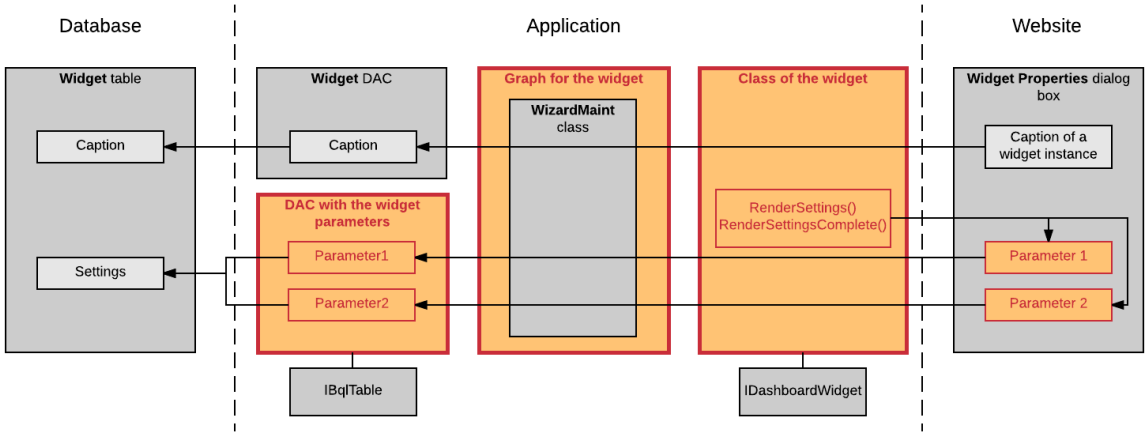

Figure: Configuration of a widget instance

How a Widget Instance Is Displayed on a Dashboard Page

After a widget instance is configured and saved in the **Widget Properties** dialog box, the widget instance is displayed on the dashboard page. To display the caption of the widget instance on the dashboad page, the widget graph retrieves the caption from the `Widget` table of the database by using the `Widget` DAC. To display the widget contents, the system uses the implementations of the `Render()` and `RenderComplete()` methods of the `IDashboardWidget` interface from the class of the widget. These implementations use the parameters of the widget, which the widget graph retrieves from the `Widget` table of the application database by using the DAC with the widget parameters.

You can specify how the widget should be loaded to a dashboard page and implement custom scripts to manage the way the widget is displayed on a dashboard page. For more information, see *To Load a Widget Synchronously or Asynchronously* and *To Add a Script to a Widget*.

The following diagram illustrates the interaction of the components of the widget with a dashboard page. In the diagram, the items that you add for the custom widget are shown in rectangles with a red border.

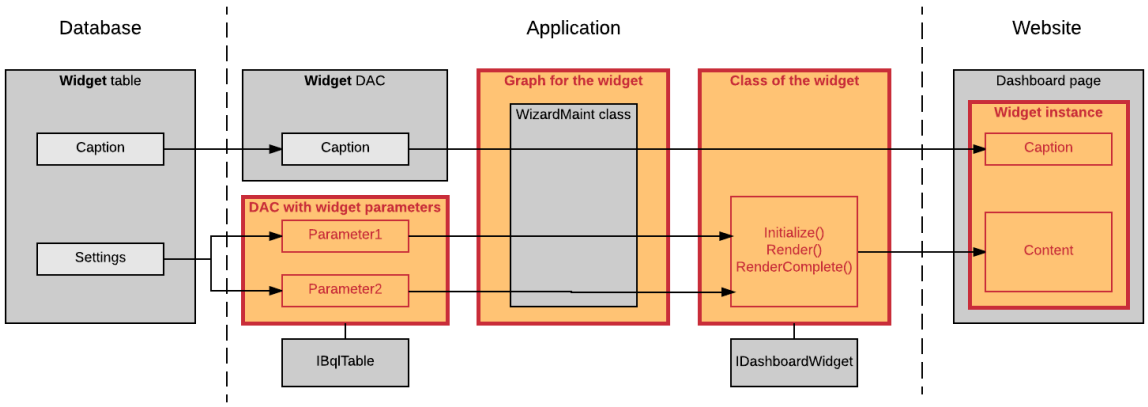

Figure: Widget instance on a dashboard page

Related Links

- *Personalizing Dashboards*

To Create a Simple Widget

A simple widget displays information from a single data source and does not require calculations or comprehensive data retrieval. Multiple simple widgets, such as wiki widgets or embedded page widgets, are available in Acumatica ERP or an Acumatica Framework-based application by default.

To create a simple widget, you need to perform the basic steps that are described in this topic.

To Create a Simple Widget

1. In the project of your Acumatica ERP extension library or your Acumatica Framework-based application, create a data access class (DAC), which stores the parameters of the widget. The DAC must implement the `IBqlTable` interface; you can use any DAC attributes with this DAC.

   The following code fragment shows an example of a DAC for a simple widget, which uses one parameter.

```
using PX.Data;

[PXHidden]
public class YFSettings : IBqlTable
{
    #region PagePath
    [PXDBString]
    [PXDefault]
    [PXUIField(DisplayName = "Help Article")]
    public string PagePath { get; set; }
    public abstract class pagePath : IBqlField { }
    #endregion
}
```

2. In the project, create a graph for working with the parameters of the widget and reading the data for the widget. The graph must be inherited from the `PX.Dashboards.WizardMaint` class.

   The following code fragment shows an example of a graph for a widget.

```
using PX.Dashboards;

public class YFSettingsMaint : WizardMaint<YFSettingsMaint, YFSettings>
{
}
```

3. In the project, create a widget class that implements the `PX.Web.UI.IDashboardWidget` interface. Use the following instructions for implementation:

   - Inherit the widget class from the `PX.Dashboards.Widgets.PXWidgetBase` abstract class. This class implements part of the required functionality of the `IDashboardWidget` interface, such as localization of the caption and the description of the widget (which are displayed in the **Add Widget** dialog box when a user adds a new widget to a dashboard). This class also

stores useful properties of the widget, such as `Settings`, `DataGraph`, `Page`, `DataSource`, and `WidgetID`.

- Store the values of the caption and the description of the widget in a `Messages` class that has the `PXLocalizable` attribute. This approach is required for localization functionality to work properly.
- Perform initialization of a widget class instance in the `Initialize()` method of the `IDashboardWidget` interface.
- Create the tree of controls of the widget in the `Render()` method of the `IDashboardWidget` interface.
- If you need to check the access rights of a user to the data displayed in the widget, implement the `IsAccessible()` method of the `IDashboardWidget` interface. If the user has access to the data in the widget, the method must return `true`; if the user has insufficient rights to access the data in the widget, the method must return `false`.
- If you want to specify the way the widget is loaded, override the `AsyncLoading()` method of the `PXWidgetBase` abstract class, as described in *To Load a Widget Synchronously or Asynchronously*.

The following code fragment gives an example of a widget class. This class is inherited from the `PXWidgetBase` class. The caption and description of the widget are specified in the `Messages` class, which has `PXLocalizable` attribute. The widget class implements the `Render()` method to create the control of the widget and performs the configuration of the control in the `RenderComplete()` method.

```
using PX.Web.UI;
using PX.Dashboards.Widgets;
using System.Web.UI;
using System.Web.UI.WebControls;

public class YFWidget: PXWidgetBase<YFSettingsMaint, YFSettings>
{
    public YFWidget()
        : base(Messages.YFWidgetCaption, Messages.YFWidgetDescription)
    {
    }

    protected override WebControl Render(PXDataSource ds, int height,
      bool designMode)
    {
        if (String.IsNullOrEmpty(Settings.PagePath)) return null;

        WebControl frame = _frame = new WebControl(HtmlTextWriterTag.Iframe)
          { CssClass = "iframe" };
        frame.Attributes["frameborder"] = "0";
        frame.Width = frame.Height = Unit.Percentage(100);
        frame.Attributes["src"] = "javascript:void 0";
        return frame;
    }

    public override void RenderComplete()
    {
        if (_frame != null)
```

```
        {
            var renderer = JSManager.GetRenderer(this.Page);
            renderer.RegisterStartupScript(this.GetType(),
                this.GenerateControlID(this.WidgetID),
                string.Format("px.elemByID('{0}').src = '{1}';",
                  _frame.ClientID, Settings.PagePath), true);
        }
        base.RenderComplete();
    }

    private WebControl _frame;
}

[PXLocalizable]
public static class Messages
{
    public const string YFWidgetCaption = "YogiFon Help Page";
    public const string YFWidgetDescription = "Displays a Help page.";
}
```

4. Compile your Acumatica ERP extension library or Acumatica Framework-based application.
5. Run the application and make sure that the new widget appears in the **Add Widget** dialog box. The widget class, which implements the `IDashboardWidget` interface, is detected by the system and automatically added to the list of widgets available for selection in the dialog box.

Related Links

- *To Load a Widget Synchronously or Asynchronously*

To Create an Inquiry-Based Widget

An inquiry widget retrieves data for the widget from an inquiry form, such as a generic inquiry form or a custom inquiry form. Inquiry-based widgets that are available in Acumatica ERP or an Acumatica Framework-based application by default include chart widgets, table widgets, and KPI widgets.

To create an inquiry-based widget, you need to perform the basic steps that are described in the section below. In these steps, you use the predefined classes, which provide the following functionality to simplify the development of an inquiry-based widget:

- Selection of the inquiry form in the widget settings
- Selection of a shared inquiry filter
- Selection of the parameters of the inquiry
- The ability to drill down in the inquiry form when a user clicks on the widget caption
- Verification of the user's access rights to the widget, which is performed based on the user's access rights to the inquiry form

To Create an Inquiry-Based Widget

1. In the project of your Acumatica ERP extension library or your Acumatica Framework-based application, create a data access class (DAC) that implements the `IBqlTable` interface and

stores the parameters of the widget. We recommend that you inherit the DAC from the `PX.Dashboards.Widgets.InquiryBasedWidgetSettings` class, which provides the following parameters for the widget:

- *InquiryScreenID*: Specifies the inquiry form on which the widget is based
- *FilterID*: Specifies one of the shared filters available for the specified inquiry form

The following code shows a fragment of the DAC for the predefined inquiry-based data table widget. The DAC is inherited from the `InquiryBasedWidgetSettings` class.

```
using PX.Data;
using PX.Dashboards.Widgets;

[PXHidden]
public class TableSettings : InquiryBasedWidgetSettings, IBqlTable
{
    #region AutoHeight
    [PXDBBool]
    [PXDefault(true)]
    [PXUIField(DisplayName = "Automatically Adjust Height")]
    public bool AutoHeight { get; set; }
    public abstract class autoHeight : IBqlField { }
    #endregion

    ...
}
```

2. In the project, create a graph for working with widget parameters and reading data for the widget. Use the following instructions when you implement the graph:
   - Inherit the graph from the `PX.Dashboards.Widgets.InquiryBasedWidgetMaint` abstract class, which is inherited from the `PXWidgetBase` abstract class.
   - Implement the `SettingsRowSelected()` event, which is the `RowSelected` event for the DAC with widget parameters; it contains the current values of the parameters of the widget instance and the list of available fields of the inquiry form. (For information on how to work with the fields of the inquiry form, see *To Use the Parameters and Fields of the Inquiry Form in the Widget*.) The signature of the method is shown below.

     ```
     protected virtual void SettingsRowSelected(PXCache cache,
       TPrimary settings, InqField[] inqFields)
     ```

3. In the project, create a widget class. We recommend that you inherit this class from the `PX.Dashboards.Widgets.InquiryBasedWidget` class.
4. Compile your Acumatica ERP extension library or Acumatica Framework-based application.
5. Run the application, and make sure that the new widget appears in the **Add Widget** dialog box. The widget class, which implements the `IDashboardWidget` interface, is detected by the system and automatically added to the list of widgets available for selection in the dialog box.

To Use the Parameters and Fields of the Inquiry Form in the Widget

You can access the parameters and fields of the inquiry from that is used in the widget by using the `DataScreenBase` class, which is available through the `DataScreen` property in the widget graph and in

the widget class. An instance of the `DataScreenBase` class, which is created based on the inquiry form selected by a user in the widget settings, contains the following properties:

- `ViewName`: Specifies the name of the data view from which the data for the widget is taken.
- `View`: Returns the data view from which the data for the widget is taken.
- `ParametersViewName`: Specifies the name of the data view with the parameters of the inquiry.
- `ParametersView`: Returns the data view with the parameters of the inquiry. It can be `null` if the inquiry has no parameters.
- `ScreenID`: Specifies the ID of the inquiry form.
- `DefaultAction`: Specifies the action that is performed when a user double-clicks on the row in the details table of the inquiry form.

To access the fields of the inquiry form in the widget, use the `GetFields()` method of the `DataScreenBase` class. This method returns the `InqField` class, which provides the following properties:

- `Name`: Specifies the internal name of the field
- `DisplayName`: Specifies the name of the field as it is displayed in the UI
- `FieldType`: Specifies the C# type of the field
- `Visible`: Specifies whether the field is visible in the UI
- `Enabled`: Specifies whether the field is enabled in the UI
- `LinkCommand`: Specifies the linked command of the field

To access the parameters of the inquiry form in the widget, use the `GetParameters()` method of the `DataScreenBase` class, which returns the `InqField` class.

To Load a Widget Synchronously or Asynchronously

By default, if a widget class is inherited from the `PX.Dashboards.Widgets.PXWidgetBase` abstract class, the widget is loaded asynchronously after the page has been loaded. You can change this behavior by using the `AsyncLoading()` method of the widget class, as described in the following sections.

To Load a Widget Synchronously

To load a widget synchronously along with the page, override the `AsyncLoading()` method, as the following code shows.

```
using PX.Web.UI;
using PX.Dashboards.Widgets;

public override AsyncLoadMode AsyncLoading
{
    get { return AsyncLoadMode.False; }
}
```

To Load a Widget Asynchronously

To load a widget asynchronously after the page has been loaded, you do not need to perform any actions, because the following implementation of the `AsyncLoading()` method is used by default.

```
using PX.Web.UI;
using PX.Dashboards.Widgets;

public override AsyncLoadMode AsyncLoading
{
    get { return AsyncLoadMode.True; }
}
```

To Load a Widget that Performs a Long-Running Operation

If a widget performs a long-running operation during loading, such as reading data that takes a long time, use the following approach to load this widget:

1. Override the `AsyncLoading()` method, as the following code shows. In this case, for processing the data of the widget, the system starts the long-running operation in a separate thread.

   ```
   using PX.Web.UI;
   using PX.Dashboards.Widgets;

   public override AsyncLoadMode AsyncLoading
   {
       get { return AsyncLoadMode.LongRun; }
   }
   ```

2. Override the `ProcessData()` method of the widget class so that it implements all the logic for the widget that consumes significant time while loading.

3. Override the `SetProcessResult()` method of the widget class so that it retrieves the result of the processing of the widget data.

If these methods are implemented, the system calls them automatically when it loads the widget on a dashboard page.

To Add a Script to a Widget

You can specify how a widget should be displayed on a dashboard page by using a custom script. For example, you can implement a script that handles the way the widget is resized.

To add a custom script to a widget, you override the `RegisterScriptModules()` method of the `PX.Dashboards.Widgets.PXWidgetBase` abstract class. The following code shows an example of the method implementation for a predefined data table widget.

```
using PS.Web.UI;
using PX.Dashboards.Widgets;

internal const string JSResource =
  "PX.Dashboards.Widgets.Table.px_dashboardGrid.js";
```

```
public override void RegisterScriptModules(Page page)
{
    var renderer = JSManager.GetRenderer(page);
    renderer.RegisterClientScriptResource(this.GetType(), JSResource);
    base.RegisterScriptModules(page);
}
```

To Add Custom Controls to the Widget Properties Dialog Box

The **Widget Properties** dialog box is displayed when a user creates or edits a widget. If you need to add custom controls, such as buttons or grids, to this dialog box, you need to create these controls in the `RenderSettings()` or `RenderSettingsComplete()` method of the widget class, as is described in the sections below.

To Add Buttons to the Widget Properties Dialog Box Dynamically

If you need to add buttons to the **Widget Properties** dialog box that appear based on a particular user action in the dialog box, override the `RenderSettings()` method of the widget class so that it dynamically adds the needed controls to the dialog box. The method must return `true` if all controls are created in the method implementation (that is, no automatic generation of controls is required). The default implementation of the `RenderSettings()` method of the `PXWidgetBase` class returns `false`.

The following example provides the implementation of this method in the predefined Power BI tile widget. When a user clicks the **Sign In** button in the **Widget Properties** dialog box and successfully logs in to Microsoft Azure Active Directory, the **Dashboard** and **Tile** boxes appear in the dialog box.

```
public override bool RenderSettings(PXDataSource ds, WebControl owner)
{
    var cc = owner.Controls;
    var btn = new PXButton() { ID = "btnAzureLogin", Text =
      PXLocalizer.Localize(Messages.PowerBISignIn, typeof(Messages).FullName),
      Width = Unit.Pixel(100) };
    btn.ClientEvents.Click = "PowerBIWidget.authorizeButtonClick";

    cc.Add(new PXLayoutRule() { StartColumn = true, ControlSize = "XM",
    LabelsWidth = "SM" });
    cc.Add(new PXTextEdit() { DataField = "ClientID", CommitChanges = true });
    cc.Add(new PXLayoutRule() { Merge = true });
    cc.Add(new PXTextEdit() { DataField = "ClientSecret",
      CommitChanges = true });
    cc.Add(btn);
    cc.Add(new PXLayoutRule() { });
    cc.Add(new PXDropDown() { DataField = "DashboardID",
      CommitChanges = true });
    cc.Add(new PXDropDown() { DataField = "TileID", CommitChanges = true });
    cc.Add(new PXTextEdit() { DataField = "AccessCode",
      CommitChanges = true });
    cc.Add(new PXTextEdit() { DataField = "RedirectUri" });
    cc.Add(new PXTextEdit() { DataField = "AccessToken" });
    cc.Add(new PXTextEdit() { DataField = "RefreshToken" });
```

```
    foreach (Control wc in cc)
    {
        IFieldEditor fe = wc as IFieldEditor;
        if (fe != null) wc.ID = fe.DataField;
        wc.ApplyStyleSheetSkin(ds.Page);

        PXTextEdit te = wc as PXTextEdit;
        if (te != null) switch (te.ID)
            {
                case "ClientID":
                    te.ClientEvents.Initialize =
                      "PowerBIWidget.initializeClientID";
                    break;
                case "ClientSecret":
                    te.ClientEvents.Initialize =
                      "PowerBIWidget.initializeClientSecret";
                    break;
                case "AccessCode":
                    te.ClientEvents.Initialize =
                      "PowerBIWidget.initializeAccessCode";
                    break;
                case "RedirectUri":
                    te.ClientEvents.Initialize =
                      "PowerBIWidget.initializeRedirectUri";
                    break;
            }
    }
    return true;
}
```

To Open a Pop-Up Panel in the Widget Properties Dialog Box

If you need to open a pop-up panel in the **Widget Properties** dialog box, override the `RenderSettingsComplete()` method of the widget class and create the panel within it.

The following code shows a sample implementation of the method in the predefined chart widget. The method adds the buttons to the dialog box and creates the pop-up panel after the standard controls of the dialog box have been created.

```
public override void RenderSettingsComplete(PXDataSource ds, WebControl owner)
{
    var btn = _btnConfig = new PXButton() {
      ID = "btnConfig", Width = Unit.Pixel(150),
      Text = PXLocalizer.Localize(Messages.ChartConfigure,
        typeof(Messages).FullName),
      PopupPanel = "pnlConfig", Enabled = false, CallbackUpdatable = true };
    owner.Controls.Add(btn);
    btn.ApplyStyleSheetSkin(owner.Page);

    owner.Controls.Add(CreateSettingsPanel(ds, ds.PrimaryView));
    (ds.DataGraph as ChartSettingsMaint).InquiryIDChanged += (s, e) =>
      _btnConfig.Enabled = !string.IsNullOrEmpty(e);
    base.RenderSettingsComplete(ds, owner);
}
```

Implementing Plug-Ins for Processing Credit Card Payments

Acumatica ERP processes credit card and debit card payments through the Authorize.Net payment gateway. To work with the Authorize.Net payment gateway, Acumatica ERP supports the *Authorize.Net API* payment plug-in. For more information on this plug-in, see *Means of Integration with Authorize.Net*.

In a customized Acumatica ERP application, you can implement payment plug-ins that work with credit card payment processing centers other than Authorize.Net. In this chapter, you can find information about how to develop custom plug-ins and use them in Acumatica ERP.

In This Chapter

- *Interfaces for Processing Credit Card Payments*
- *To Implement a Plug-In for Processing Credit Card Payments*

Interfaces for Processing Credit Card Payments

Acumatica ERP provides the interfaces for the implementation of plug-ins for credit card payment processing.

By using these interfaces, you can implement tokenized processing plug-ins. When a tokenized processing plug-in is used, the credit card information is not saved to the application database; this information is stored only at the processing center. The Acumatica ERP database stores only the identification tokens that link customers and payment methods in the application with the credit card data at the processing center.

For details on how to implement custom plug-ins, see *To Implement a Plug-In for Processing Credit Card Payments*.

In Acumatica ERP, the *Authorize.Net API* (*PX.CCProcessing.V2.AuthnetProcessingPlugin*) plug-in implements the interfaces described in the sections below. This plug-in works with the Authorize.Net processing center. For more information about the built-in plug-in, see *Integration with Authorize.Net Through the API Plug-in*.

Mandatory Interfaces

The root interface for implementation of custom plug-ins for credit card processing is `PX.CCProcessingBase.Interfaces.V2.ICCProcessingPlugin`. The system automatically discovers the class that implements the `ICCProcessingPlugin` interface in the `Bin` folder of the Acumatica ERP instance and includes it in the list in the **Payment Plug-In (Type)** box on the *Processing Centers* (CA205000) form. For creation of a custom plug-in, you also need to implement the `ICCTransactionProcessor` interface to process credit card transactions.

Additional Interfaces

You can implement the following additional functionality:

- Tokenized credit card processing: Implement the `ICCProfileProcessor` and `ICCHostedFormProcessor` interfaces.

- Processing of payments from new credit cards: To use this functionality, implement the `ICCProfileCreator`, `ICCHostedPaymentFormProcessor`, `ICCHostedPaymentFormResponseParser`, and `ICCTransactionGetter` interfaces. If this functionality is implemented in a custom plug-in, a user can select the **Accept Payment from New Card** check box on the *Processing Centers* form for the processing centers that use this custom plug-in.
- Synchronization of credit cards with the processing center: Implement the `ICCTransactionGetter` interface in a custom plug-in. If this functionality is implemented, on the *Synchronize Cards* (CA206000) form, users can work with the processing centers that use this custom plug-in.
- Retrieval of the information about suspicious credit card transactions (without the use of the hosted form that accepts payments): To use this functionality, implement the `ICCTranStatusGetter` interface.
- Webhooks as a way to obtain a response from the processing center: Implement the `ICCWebhookProcessor` and `ICCWebhookResolver` interfaces.

Related Links

- *To Implement a Plug-In for Processing Credit Card Payments*

To Implement a Plug-In for Processing Credit Card Payments

In Acumatica ERP, the *Authorize.Net API* built-in plug-in processes transactions in Authorize.Net. For more information on this plug-in, see *Integration with Authorize.Net Through the API Plug-in*. You can implement your own plug-in for working with processing centers other than Authorize.Net, as described in this topic.

To Implement a Credit Card Processing Plug-In

1. In a class library project, define a class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCTransactionProcessor` interface, which provides credit card processing functionality. In the class, implement the `DoTransaction` method, which processes the transaction in the payment gateway. For details on the implementation of the method, see *ICCTransactionProcessor Interface* in the API Reference.

   The following code shows the declaration of a class that implements the `ICCTransactionProcessor` interface.

```
using PX.CCProcessingBase.Interfaces.V2;

public class MyTransactionProcessor : ICCTransactionProcessor
{
  public ProcessingResult DoTransaction(ProcessingInput inputData)
  {
    ...
  }
}
```

2. If you need to implement tokenized processing, define the following classes:

- A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCProfileProcessor` interface (which provides methods to manage customer and payment profiles)
- A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCHostedFormProcessor` interface (which provides methods to work with hosted forms for the creation and management of payment profiles)

In the classes, implement the methods of the interfaces. For details on the implementation of the interfaces, see *ICCProfileProcessor Interface* and *ICCHostedFormProcessor Interface* in the API Reference.

3. If you need to implement payment processing without the preliminary creation of payment profiles, define the following classes:
   - A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCProfileCreator` interface (which provides the method to create the payment profile for a new credit card)
   - A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCHostedPaymentFormProcessor` interface (which provides methods to work with hosted forms for processing payments without preliminary creation of payment profiles)
   - A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCHostedPaymentFormResponseParser` interface (which provides a method to parse the response after a successful operation on a hosted form that processes payments without preliminary creation of payment profiles)
   - A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCTransactionGetter` interface (which provides the methods to obtain information about transactions by transaction ID)

   In the classes, implement the methods of the interfaces. For details on the implementation of the interfaces, see *ICCProfileCreator Interface*, *ICCHostedPaymentFormProcessor Interface*, *ICCHostedPaymentFormResponseParser Interface*, and *ICCTransactionGetter Interface* in the API Reference.
4. If you need to implement the retrieval of detailed information about transactions, which can be used for the synchronization of transactions with the processing center, define a class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCTransactionGetter` interface (which provides the methods to obtain information about transactions by transaction ID).

   In the class, implement the methods of the interface. For details on the implementation of the interface, see *ICCTransactionGetter Interface* in the API Reference.
5. If you need to retrieve the information about suspicious credit card transactions with the *Held for Review* status (without the use of the hosted form that accepts payments), define a class that implements the supplementary `PX.CCProcessingBase.Interfaces.V2.ICCTranStatusGetter` interface (which provides the method to obtain the transaction status after the execution of the `ICCTransactionProcessor.DoTransaction` method).

   In the class, implement the methods of the interface. For details on the implementation of the interface, see *ICCTranStatusGetter Interface* in the API Reference.
6. If you need to support webhooks as a way to obtain a response from the processing center, define the following classes:

- A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCWebhookProcessor` interface (which provides the methods to add, update, and delete webhooks and retrieve the list of webhooks)
- A class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCWebhookResolver` interface (which parses the information that comes from the processing center through webhooks)

In the classes, implement the methods of the interfaces. For details on the implementation of the interfaces, see *ICCWebhookProcessor Interface* and *ICCWebhookResolver Interface* in the API Reference.

7. Define a class that implements the `PX.CCProcessingBase.Interfaces.V2.ICCProcessingPlugin` interface, which is the root interface for credit card payment processing and is used by Acumatica ERP to find the plug-in in the application libraries. The class should have a public parameterless constructor (either explicit or default). In the class, implement the methods of the `ICCProcessingPlugin` interface (described in detail in *ICCProcessingPlugin Interface* in the API Reference) as follows:

- In the `ExportSettings` method, which exports the required settings from the plug-in to the *Processing Centers* (CA205000) form, return a collection that contains the settings that can be configured by the user on the form. The syntax of the method is shown in the following code.

```
IEnumerable<SettingsDetail> ExportSettings();
```

- In the `ValidateSettings` method, validate the settings modified on the *Processing Centers* form, which are passed as the parameter of the method, and return `null` if validation was successful or an error message if validation failed. The syntax of the method is shown in the following code.

```
string ValidateSettings(SettingsValue setting);
```

- In the `TestCredentials` method, check the connection to the payment gateway by using the credentials that are specified by the user on the *Processing Centers* form. The syntax of the method is shown in the following code.

```
void TestCredentials(IEnumerable<SettingsValue> settingValues);
```

- In the `CreateProcessor<T>` method, return a new object of the `T` type, and initialize the object with the settings passed to the method. The `T` type can be any of the following:
  - `ICCTransactionProcessor`
  - `ICCProfileProcessor`
  - `ICCHostedFormProcessor`
  - `ICCProfileCreator`
  - `ICCHostedPaymentFormProcessor`
  - `ICCTransactionGetter`
  - `ICCTranStatusGetter`
  - `ICCWebhookResolver`

- `ICCWebhookProcessor`

If a particular `T` type is not supported by your plug-in, return `null` for this type. The following code shows a sample implementation of the method.

```
public T CreateProcessor<T>(IEnumerable<SettingsValue> settingValues)
  where T : class
{
  if (typeof(T) == typeof(ICCProfileProcessor))
  {
    return new MyProfileProcessor(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCHostedFormProcessor))
  {
    return new MyHostedFormProcessor(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCHostedPaymentFormProcessor))
  {
    return new MyHostedPaymentFormProcessor(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCHostedPaymentFormResponseParser))
  {
    return new MyHostedPaymentFormResponseParser(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCTransactionProcessor))
  {
    return new MyTransactionProcessor(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCTransactionGetter))
  {
    return new MyTransactionGetter(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCProfileCreator))
  {
    return new MyProfileCreator(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCWebhookResolver))
  {
    return new MyWebhookResolver() as T;
  }
  if (typeof(T) == typeof(ICCWebhookProcessor))
  {
    return new MyWebhookProcessor(settingValues) as T;
  }
  if (typeof(T) == typeof(ICCTranStatusGetter))
  {
    return new MyTranStatusGetter() as T;
  }
  return null;
}
```

8. Build your project.

9. To add your plug-in to Acumatica ERP, include the assembly in the customization project. (When the customization project is published, the assembly is copied to the `Bin` folder of the Acumatica ERP website automatically.) During startup, the system automatically discovers the class that implements the `ICCProcessingPlugin` interface and includes it in the list in the **Payment Plug-In (Type)** box on the *Processing Centers* form.

Related Links

- *Interfaces for Processing Credit Card Payments*
- *PX.CCProcessingBase.Interfaces.V2 Namespace*