CentraSite

CentraSite Control Pluggable Architecture

Version 9.6

April 2014

This document applies to CentraSite Version 9.6.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IINM-OIINMDG-PLUG-96-20140318

Table of Contents

Preface

This document describes CentraSite Control's pluggable architecture. Using this architecture, you can extend CentraSite Control's functionality by adding your own features with appropriate graphical user interfaces and Java classes.

The document contains the following sections:

Introduction	This section gives an overview of the pluggable architecture and indicates the areas of functionality in which user-defined plug-ins can be added.
Customizing the Welcome Page	This section describes the available extension points for the CentraSite Control Welcome page, as well as the Java classes and methods that you need to implement in order to plug in to the extension points.
Customizing Content Pages	This section describes the available extension points for the CentraSite Control content pages, as well as the Java classes and methods that you need to implement in order to plug in to the extension points.
Setting the Preferred Plug-in and Order of Plug-ins	This section describes how to set a preferred extension point when you invoke the user interface.
Installing and Uninstalling Plug-ins	This section describes how to install and uninstall plug-ins.
Special and Advanced Topics	This section describes topics of particular interest to advanced users.
Javadoc Documentation of the APIs	This section summarizes the Javadoc documentation for the Java classes and methods that define the extension points.
Appendix A: Step-by-Step Guide	This section gives you step-by-step instructions on how to create, install and use sample plug-ins.

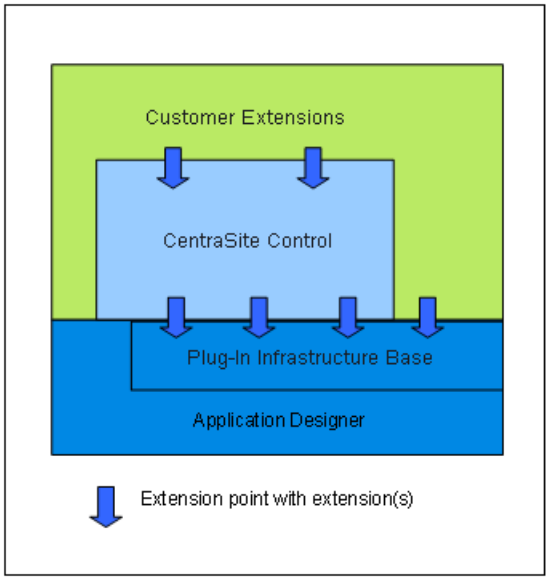
I Introduction

1 Introduction

CentraSite Control offers a pluggable architecture that allows you to extend the standard graphical interface by adding your own features.

The CentraSite Control user interface is itself a plug-in to a base infrastructure, in other words, the base infrastructure provides extension points where CentraSite Control is plugged in. The base infrastructure is composed of the Application Designer, which provides the basic graphical infrastructure of the GUI, and the plug-in infrastructure base, which allows plug-ins to communicate with the Application Designer.

The pluggable architecture is illustrated in the following diagram:



The plug-in infrastructure is inspired by Eclipse, which allows the user interface to be extended by domain-specific or customer-specific functionality.

Plug-ins are implemented as Java classes. The points in the code at which plug-ins can be added are called *extension points*. CentraSite Control offers extension points that allow you to implement or extend the following features:

- Provide an alternative login screen.
- Add a perspective.
- Add a topic to the navigation pane within any perspective.
- Support I18N (internationalization) for layouts contributed by a plug-in.
- Add a logo and links to the login dialog.
- Handle the creation and termination of the connection to a backend machine.
- Add a perspective contributing the following components: a toolbar, a logo, one or more topics and a background screen. A perspective allows you to group topics in the navigation view.
- Add a command to the context menu of a registry object or a repository object
- Add a property to an object. The property is then visible in detail views and under the **General** tab.
- Add a tab to the detail view of registry objects and repository objects.
- Add a source of notifications.
- Add secondary icons to nodes in the graphical impact analysis.
- Extend the **Summary** tab.
- Replace the standard detail view used as editor for registry / repository objects by an object type specific editor.
- Extend the set of available import commands.
- Extend the search dialog by additional conditions.

A plug-in can itself provide extension points for further plug-ins.

The available extension points are described in the section ***Extension Points.***

II Customizing the Welcome Page

2 Customizing the Welcome Page

The Welcome page that you see when you start CentraSite Control can be customized to suit your own requirements. You can change aspects such as icons used, colors, text, fonts and layouts. You can also define links that will take you straight to the pages of CentraSite Control that you use the most, and links to external web sites.

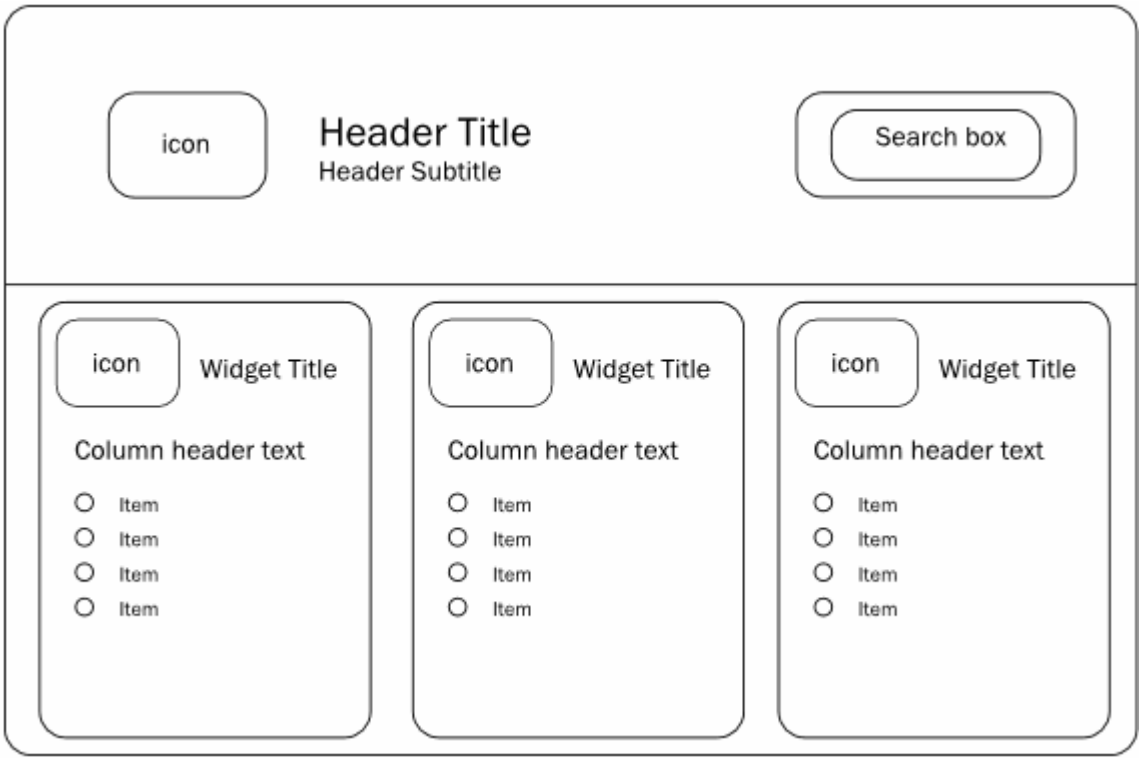
The information contained in the following sections describes how to customize your Welcome page.

Introduction

The standard Welcome page gives you quick links to the pages of CentraSite Control that you will probably use frequently during your day-to-day work with CentraSite. It also provides links to external web sites that provide useful information related to CentraSite. In the Welcome page you can specify the language you wish to use for your further work with CentraSite Control, and you can specify the date format to be used in the various displays.

A search box allows you perform a keyword search for registry assets and objects whose name or description contains the given keyword.

The Welcome page has the following schematic layout:



The header section at the top contains a title text and a subtitle text. You can change the texts, the fonts and colors used to display the texts. An icon can be displayed adjacent to the title and subtitle. A search box is displayed by default, which allows you to perform a keyword search for an asset. You can hide the search box as part of the page customization. You can change the background color for the whole header section, and you can change the background color of the search box.

Below the header section, there can be one or more so-called widgets. Each widget contains a title, with an icon adjacent to the text. Under the title, you can have a list of entries, each representing some executable action. Typically, an action contains a URL to either a page of your choice within CentraSite Control, or to an external web page that you regularly visit within the context of your work with CentraSite.

There are several kinds of widget:

- **Single-column widget**
  In this widget, the executable actions are displayed as a table consisting of a single column. Each table cell contains one executable action. Each cell can also have an icon beside it. There is a header text above the table.
- **Multi-column widget**
  In this widget, the executable actions are displayed as a table consisting of two or more columns. Each table cell contains one executable action. Each cell can also have an icon beside it. There is a header text above each column of the table.
- **HTML-style widget**
  In this widget, the contents are freely programmable as HTML code. The HTML statements you use must be valid within the context of an HTML table cell, i.e. there is an implicit HTML <td> element enclosing the HTML code you supply.

The Welcome page can contain up to 10 widgets. The widgets are displayed side by side in a single row.

In general, you can use CSS stylesheet statements to customize the appearance of text and colors in the Welcome page.

Technical Implementation of the Welcome Page

The Welcome page is implemented as a plug-in module within the CentraSite pluggable UI architecture. This means that all of the development aspects that are relevant for implementing CentraSite plug-in modules apply also to the Welcome page.

The following sections describe the technical implementation of the Welcome page.

- Java interface hierarchy

Overview of Java Methods used

The layout and contents of the Welcome page are implemented as Java code.

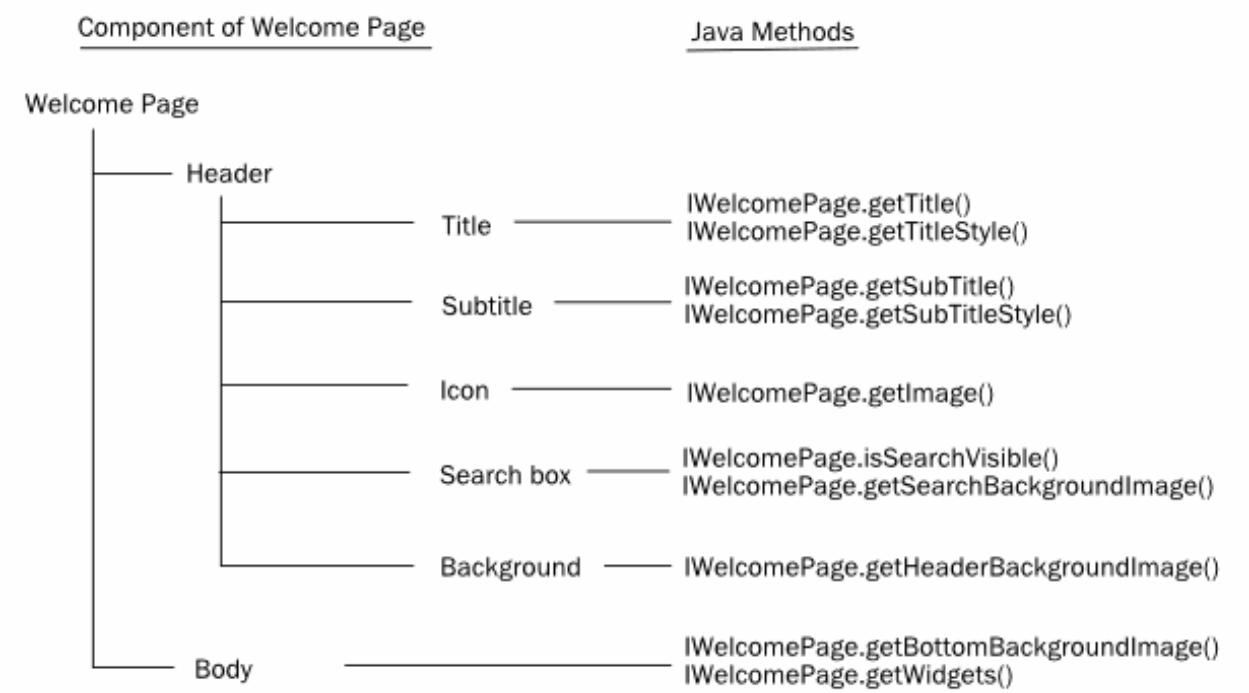
Each customizable part of the Welcome page requires a corresponding Java class. The Welcome screen can be defined as a combination of the following hierarchical structures:

- The header and body of the Welcome page.
- The widgets in the body of the Welcome page.
- The items in the columns of the widgets.

Screen Component: Welcome Page

The Welcome page is composed of a header and a body. The header contains a title, subtitle, icon, search box and background image. The body contains one or more widgets.

The content and appearance of these components are determined by the Java methods shown in the following diagram.



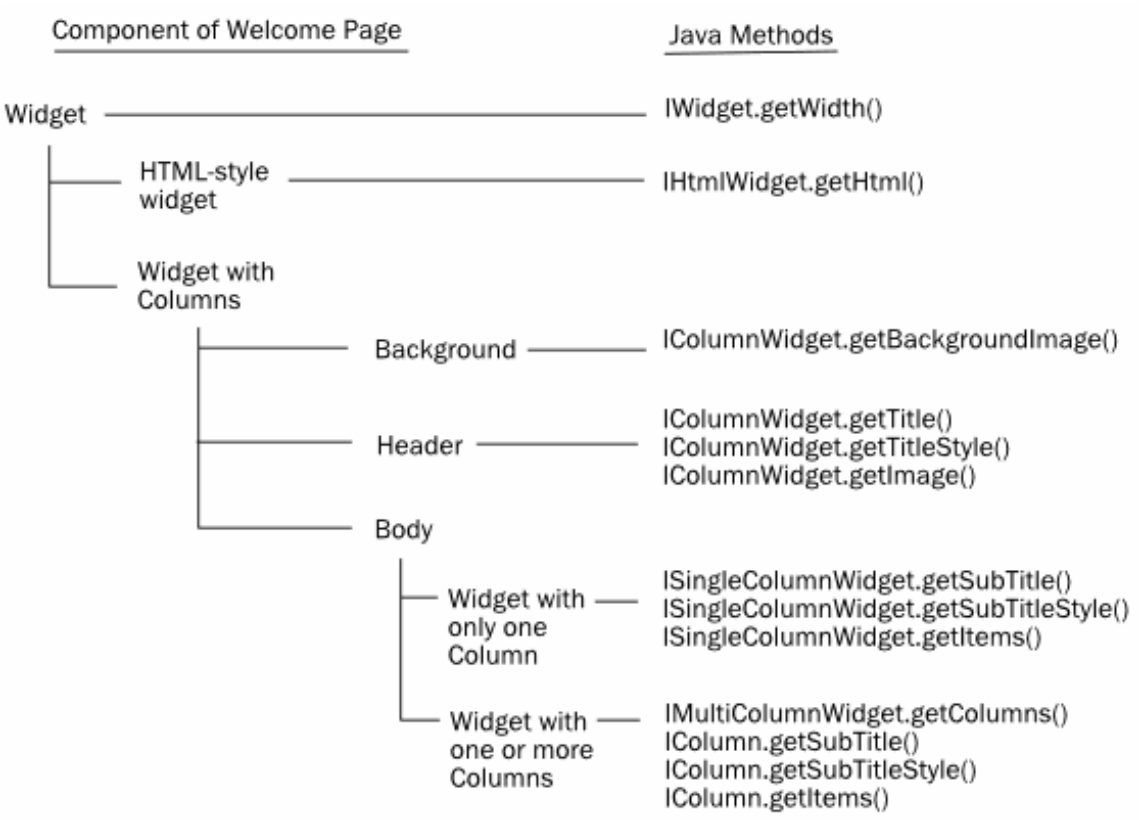
The following table describes the purpose of these Java methods:

Name of Java interface	Java Method	Description
IWelcomePage	getTitle();	Defines the header text to be used.
IWelcomePage	getTitleStyle();	Defines the CSS style information for the header.
IWelcomePage	getSubTitle();	Defines the header subtitle text to be used.
IWelcomePage	getSubTitleStyle();	Defines the CSS style information for the header subtitle.
IWelcomePage	getImage();	Defines the icon to be used in the header.
IWelcomePage	isSearchVisible();	Defines whether the search box in the header part is visible or invisible.
IWelcomePage	getSearchBackgroundImage();	Defines a background image to be used for the search box.
IWelcomePage	getHeaderBackgroundImage();	Defines a background image to be used for the header part.
IWelcomePage	getBottomBackgroundImage();	Defines a background image to be used for the body part.
IWelcomePage	getWidgets();	Defines the widgets that will be used in the body part.

Screen Component: Widget

The body part of the Welcome page is composed of one or more widgets. A widget can define just HTML code (an HTML-style widget) or can define content and layout, similar to the header part of the Welcome page. The content/layout components are: a background image, a header text, the definition of a single-column table of items, the definition of a multi-column table of items.

The content and appearance of these components are determined by the Java methods shown in the following diagram.



The following table describes the purpose of these Java methods:

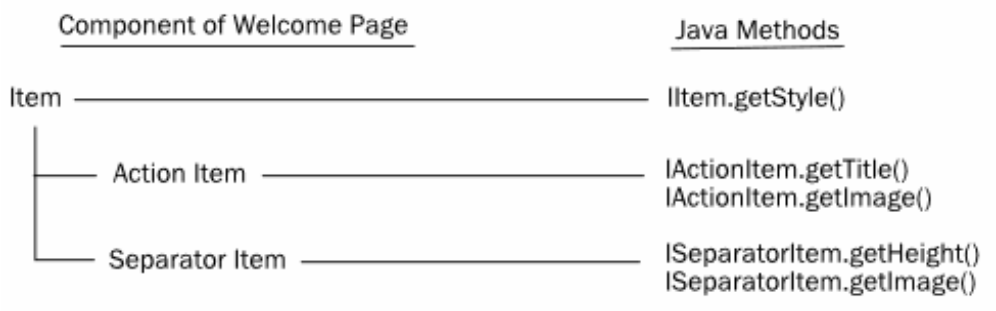
Name of Java interface	Java Method	Description
IWidget	getWidth();	Defines the screen width of the widget.
IHtmlWidget	getHtml();	Defines HTML code for an HTML-style widget.
IColumnWidget	getBackgroundImage();	Defines the background image to be used for a column of a widget.
IColumnWidget	getTitle();	Defines the header text of a column of a widget.
IColumnWidget	getTitleStyle();	Defines the CSS style for the header text of a column of a widget.
IColumnWidget	getImage();	Defines the background image to be used for the header part of the widget.
ISingleColumnWidget	getSubTitle();	Defines the subtitle text of a single-column widget.
ISingleColumnWidget	getSubTitleStyle();	Defines the CSS style for the subtitle header text of a single-column widget.
ISingleColumnWidget	getItems();	Defines the items contained in a single-column widget.
IMultiColumnWidget	getColumns();	Defines the columns used in a multi-column widget.
IColumn	getSubTitle();	Defines the subtitle text of a column of a multi-column widget.

Name of Java interface	Java Method	Description
IColumn	getSubTitleStyle();	Defines the CSS style for the subtitle text of a column of a multi-column widget.
IColumn	getItems();	Defines the items contained in a column of a multi-column widget.

Screen Component: Item

Each widget in the body part of the Welcome page can contain one or more items, arranged in one or more table columns. An item represents an executable action, which you can define freely; for example, the action could be the activation of a URL in order to reach a particular page within CentraSite Control or an external web site.

The content and appearance of these components are determined by the Java methods shown in the following diagram.



The following table describes the purpose of these Java methods:

Name of Java interface	Java Method	Description
IItem	getStyle();	Defines the CSS style for the item.
IActionItem	getTitle();	Defines the text to be displayed for the item.
IActionItem	getImage();	Defines the icon to be displayed adjacent to the descriptive text.
ISeparatorItem	getHeight();	Defines the height in pixels of the area that contains the separator image.
ISeparatorItem	getImage();	Defines the image be displayed as the separator item.

Methods Not Related To Screen Components

The following list shows the Java methods that are not related to a screen component, but which are required for the pluggable UI architecture of CentraSite Control.

Name of Java interface	Java Method	Description
(all interfaces)	setLocale();	This informs the widget or item about the CentraSite Control locale that is required to localize texts for the display. This method is called automatically before any other method that might depend on the locale.
(all interfaces)	setActionContext();	This informs the widget or item about the CentraSite Control context that is required for the processing to be done subsequently by the execute() method. This method is called automatically before any other method that might depend on the action context.
IWidget	invalidate();	This sets the status that the display of the current item must be refreshed (true) or does not need to be refreshed (false).
IWidget	isInvalidated();	This returns whether or not the display of the item needs to be refreshed.
IItem	getWidget();	This method gets the widget to which the current item belongs.
IItem	setWidget();	This method sets the widget to which the current item belongs.
IActionItem	execute();	This activates the action to be performed when you click on the current item.

Java interface hierarchy

The interface hierarchy is as follows:

```
IWelcomePage

IWidget
   IColumnWidget
      IMultiColumnWidget
      ISingleColumnWidget
   IHtmlWidget

IColumn

IItem
   IActionItem
   ISeparatorItem
```

Installing the Customized Welcome page

The Welcome page is implemented as a CentraSite Control extension point in the context of CentraSite's pluggable UI architecture. To install your customized Welcome page, you need to modify CentraSite Control's pluggable UI configuration in the Software AG Runtime environment.

The required deployment steps are describe in the following sections:

- Stop Software AG Runtime
- Updating the plugin.xml configuration file
- Deploying the new Java classes to the PluggableUI environment
- Start Software AG Runtime

Stop Software AG Runtime

Before you make any changes to the Software AG Runtime environment, stop the Software AG Runtime process.

Updating the plugin.xml configuration file

The standard *plugin.xml* configuration file delivered with the CentraSite kit contains all of the names of the CentraSite Control extension points, including the extension point for the Welcome page. You must update this file to contain the definition of the customized Welcome page. The configuration file is located in the folder `<RuntimeDir>\workspace\webapps\PluggableUI\CentraSiteControl` .

There are two elements in the standard *plugin.xml* file that refer to the Welcome page. The first part defines the name of the extension point to be used for the Welcome page, and looks like this:

```
<extension-point id="welcomePage">
</extension-point>
```

The second part defines the Java class that implements the Welcome page, and looks like this:

```
<!-- Welcome Page -->
<extension
   point="com.centrasite.control.welcomePage"
   id="welcomePage"
   class="com.centrasite.control.ext.welcome.standard.WelcomePage">
</extension>
```

The `point` attribute of the `extension` element in the second part must match the name given by the `id` attribute of the `plugin` element (usually in the first line in the *plugin.xml* file) concatenated

with a dot and the `id` attribute of the `extension-point` element in the first part. For example, if the `id` attribute of the `plugin` element is "`com.centrasite.control`" and the `id` attribute of the `extension-point` element is "`welcomePage`", then the value of `point` attribute of the `extension` element must be "`com.centrasite.control.welcomePage`".

The Java class identified by the `class` attribute of the `extension` element must implement the interface *IWelcomePage*.

To use your customized Welcome page instead of the standard Welcome page, set the `class` attribute to your customized Java class that implements the interface *IWelcomePage*.

For general information about *plugin.xml*, see the section **Installing and Uninstalling Plug-ins**.

The changes that you make in *plugin.xml* take effect the next time Software AG Runtime is started.

Note: Instead of overwriting the standard element in *plugin.xml*, you might want to retain a copy of the original element and comment it out. This means that you can revert easily to the original Welcome page if required, by commenting out your customized element and uncommenting the original element.

Deploying the new Java classes to the PluggableUI environment

In addition to modifying the *plugin.xml* file, as described above, you need to copy the Java classes for your customized Welcome page to the CentraSite Control location in Software AG Runtime.

There are two ways of doing this:

- Create a jar file containing the class files for your customized Welcome page, and copy the jar file to the *CentraSiteControl\lib* folder in Software AG Runtime.
- Copy the class files to the *CentraSiteControl\classes* folder and its subfolders, according to the naming convention of the Java package that contains the classes. If, for example, your package name is *com.centrasite.control.ext.welcome.sample*, then copy the classes to the *CentraSiteControl\classes\com\centrasite\control\ext\welcome\sample* folder.

You can also combine these methods, and define some classes via a jar file in the *lib* folder and some classes as class files in the appropriate subfolder of the *classes* folder. If you have defined a class in both *lib* and a subfolder of *classes*, the class in the *CentraSiteControl\classes* subfolder will be used.

If you have defined new icons for the customized Welcome page, you need to copy the icons to the appropriate location under the *CentraSiteControl* folder. If, for example, your code contains the definition `public String getImage() { return "images/my_welcome_icon.png"; }`, ensure that the icon *my_welcome_icon.png* is copied to *CentraSiteControl\images*.

Start Software AG Runtime

After you have made the changes to the Software AG Runtime environment, restart the Software AG Runtime process. The changes you have made should now be visible when you view the Welcome Page.

Example of a customized Welcome Page

This section describes the customized welcome page that is provided as a demo in the product distribution.

Location of demo files

All of the required files for the demo are contained in the folder *demos\WelcomePage* under the CentraSite installation location. The following files are available at this location:

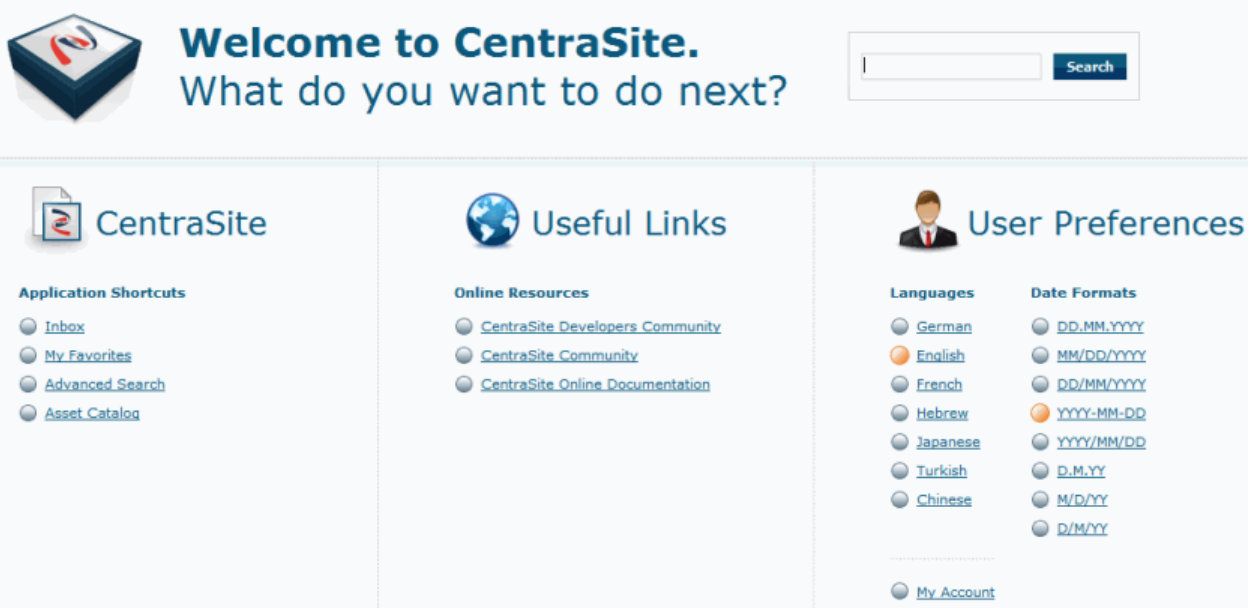
- The Java source files. These are located in the subfolder *src*.
- Icons to be displayed in the Welcome page. These are located in the subfolder *resources*.
- Updates for the Software AG Runtime configuration. These are in the file *resources\plugin.xml*.
- Eclipse project files *.classpath* and *.project*.
- Apache Ant files *build.properties* and *build.xml* for building the files that will be deployed to Software AG Runtime.

The following sections describe how to use these files to build and deploy the demo Welcome page.

Differences between standard Welcome Page and demo Welcome Page

This section shows the differences between the standard welcome page and the demo welcome page. Based on this you should be able to quickly evaluate the usefulness of this feature for your own business requirements.

The standard welcome page has the following appearance:



The demo welcome page used as an example in this section has the following appearance:

The main changes between the standard welcome page and the customized welcome page that Software AG supplies as a demo can be summarized as follows:

- The text in the title of the header section has changed. Also the color of this text has changed.
- The background color in the customized welcome page has changed.
- The large icons in the titles of the header part and of the widgets have changed.
- The small icons in the CentraSite widget have changed.
- The widgets have 3-D effect shadowed borders.
- The search box in the header section has been removed.

Implementation of Welcome Page layout

This section lists the layout possibilities of the welcome page and specifies the Java methods where the layout is defined.

Note: If any background image that is defined for an area of the display is not as wide as the area, the image is repeated horizontally until the whole width of the area is covered.

- Header area
- Separator between header part and widget part
- Widget CentraSite
- Widget Useful Links
- Widget User Preferences
- Default Settings for Widgets

Header area

Layout component	Source file	Java Method
Icon	WelcomePage.java	getImage();
Background image	WelcomePage.java	getHeaderBackgroundImage();
Title text	WelcomePage.java	getTitle();
CSS style of title text	WelcomePage.java	getTitleStyle();
Subtitle text	WelcomePage.java	getSubTitle();
CSS style of subtitle text	WelcomePage.java	getSubTitleStyle();
Background image of the Search box	WelcomePage.java	getSearchBackgroundImage();
Make the Search box visible/invisible	WelcomePage.java	isSearchVisible();

Separator between header part and widget part

Layout component	Source file	Java Method
Image	SeparatorItem.java	getImage();
Height in pixels	SeparatorItem.java	getHeight();

Widget CentraSite

Layout component	Source file	Java Method
Width of widget	CentraSiteWidget.java	getWidth();
Title text	CentraSiteWidget.java	getTitle();
CSS style of title text	CentraSiteWidget.java	getTitleStyle();
Subtitle text	CentraSiteWidget.java	getSubTitle();
CSS style of subtitle text	CentraSiteWidget.java	getSubTitleStyle();
Header icon	CentraSiteWidget.java	getImage();
Background image	CentraSiteWidget.java	getBackgroundImage();
CentraSite widget: define the items to be included in the bullet list	CentraSiteWidget.java	getItems();
Icon for item "Asset Catalog"	KeywordSearchItem.java	getImage();
Text for item "Asset Catalog"	KeywordSearchItem.java	getTitle();
Icon for item "Advanced Search"	AdvancedSearchItem.java	getImage();
Text for item "Advanced Search"	AdvancedSearchItem.java	getTitle();
Icon for item "Inbox"	InboxItem.java	getImage();
Text for item "Inbox"	InboxItem.java	getTitle();
Icon for item "My Favorites"	MyFavoriteItem.java	getImage();
Text for item "My Favorites"	MyFavoriteItem.java	getTitle();

Widget Useful Links

Layout component	Source file	Java Method	Comment
Header icon	UsefulLinksWidget.java	getImage();	
Width of widget	UsefulLinksWidget.java	getWidth();	
Title text	UsefulLinksWidget.java	getTitle();	
CSS style of title text	UsefulLinksWidget.java	getTitleStyle();	
Subtitle text	UsefulLinksWidget.java	getSubTitle();	
CSS style of subtitle text	UsefulLinksWidget.java	getSubTitleStyle();	
Background image	UsefulLinksWidget.java	getBackgroundImage();	

Layout component	Source file	Java Method	Comment
Text for item "CentraSite Developers Community"	DeveloperCommunityItem.java	getTitle();	
URL for item "CentraSite Developers Community"	DeveloperCommunityItem.java	execute();	The creation of a hyperlink that opens a new browser page is implemented by a call of the `openPageInNewWindow` method of the `getDisplayAdapter()` class that is available in the *CentraSiteControlUI.jar* file in Software AG Runtime.
Text for item "CentraSite Community"	CentraSiteCommunityItem.java	getTitle();	
URL for item "CentraSite Community"	CentraSiteCommunityItem.java	execute();	See the comment for "CentraSite Developers Community".
Text for item "CentraSite Online Documentation"	OnlineDocumentationItem.java	getTitle();	
URL for item "CentraSite Online Documentation"	OnlineDocumentationItem.java	execute();	See the comment for "CentraSite Developers Community".
Define the items to be included in the bullet list	UsefulLinksWidget.java	getItems();	

Widget User Preferences

Layout component	Source file	Java Method
Header icon	UserPreferencesWidget.java	getImage();
Width of widget	UserPreferencesWidget.java	getWidth();
Title text	UserPreferencesWidget.java	getTitle();
CSS style of title text	UserPreferencesWidget.java	getTitleStyle();
Background image	UserPreferencesWidget.java	getBackgroundImage();
Text of the "Languages" subtitle	LanguagesColumn.java	getSubTitle();
CSS style of the "Languages" subtitle	LanguagesColumn.java	getSubTitleStyle();

Layout component	Source file	Java Method
Width of "Languages" column in pixels	LanguagesColumn.java	getWidth();
Text of the "Date Formats" subtitle	DateFormatsColumn.java	getSubTitle();
CSS style of the "Date Formats" subtitle	DateFormatsColumn.java	getSubTitleStyle();
Width of "Date Formats" column in pixels	DateFormatsColumn.java	getWidth();
Languages column: define the items to be included in the bullet list	LanguagesColumn.java	getItems();
Date Formats column: define the items to be included in the bullet list	DateFormatsColumn.java	getItems();

Default Settings for Widgets

Layout component	Source file	Constant
Widgets: default "blue circle" icon to mark individual entries in a widget	WelcomePage.java	BLUE_CIRCLE_ICON
Widgets: default "orange circle" icon to mark individual entries in a widget	WelcomePage.java	ORANGE_CIRCLE_ICON
Widgets: default CSS style of the title text of a widget	WelcomePage.java	WIDGET_TITLE_STYLE
Widgets: default CSS style of the subtitle text of a widget	WelcomePage.java	WIDGET_SUBTITLE_STYLE
Widgets: default CSS style of the text for each item of a widget	WelcomePage.java	ACTION_ITEM_STYLE

Implementing the Demo as an Eclipse Java Project

If you wish to use Eclipse as your development environment for updating the Java sources of the customized welcome page, the *demos\WelcomePage* folder contains the Eclipse project files *.classpath* and *.project*. You can use these files to create an Eclipse Java project for managing your Java sources. To create and use the Eclipse Java project, proceed as follows:

To create and use the Eclipse Java project

1 Start Eclipse.

2 Select **File > New > Project > Java Project**.

This opens the wizard for creating a new Java project.

3 Select **Create project from existing source**.

4 Specify the path *demos\WelcomePage* as the location of the existing project files.

When you build the project in Eclipse (using for example **Project > Build Project**), there should be no errors reported.

Building the deployment files for Software AG Runtime

To deploy the demo welcome page to Software AG Runtime, you need to create a jar file containing the Java classes of your Java sources, then copy the jar file and any required graphic icons to the Software AG Runtime environment.

You can build the jar file by using Apache Ant with the build file *build.xml* provided in the *demos\WelcomePage* folder. The file *build.xml* uses a properties file *build.properties* to define some customer-specific files names and folder locations.

The build file *build.xml* also builds a zip file that contains the jar file and all required graphical icons. To deploy the demo welcome page, you can unzip the contents of the zip file directly into your Software AG Runtime location, as described below.

The build.properties file

The file *build.properties* contains the following properties that you should tailor to your working environment before you run *build.xml*.

Property	Description
projectName	This is the name that will be used for the jar file and zip file that are created by the Ant task. The jar file will be copied to the *CentraSite\lib* folder in the Software AG Runtime environment, so choose a name that will easily distinguish the jar file from other jar files at the Software AG Runtime location.
pluggableLocation	This is the location of the folder*PluggableUI* in the *<RuntimeDir>/workspace/webapps/* directory. In a Windows environment, you should use forward slashes instead of backward slashes in the path name.
centraSiteLocation	This is the path where your CentraSite installation is located. In a Windows environment, you should use forward slashes instead of backward slashes in the path name.

Building the deployment files

The *build.xml* file contains the definition of the tasks to be performed by Ant. The tasks defined in the delivered demo version are:

- Compile the Java sources that are located in the folder *src* and store the Java classes in the folder *classes*.
- Create a jar file containing all of the class files, and store the jar file in the folder *lib*.
- Create a zip file that contains the jar file and all icons associated with the customized welcome page, and store the zip file in the folder *lib*.

The *build.xml* file is an XML file that contains element definitions such as:

```
<zipfileset dir="resources" prefix="images">
   <include name="*.png" />
</zipfileset>
```

In such cases, the `dir` attribute indicates the name of the folder in the build environment where Ant can locate the required files, and the *prefix* attribute indicates the folder in the Software AG Runtime environment where the files will be copied to. In the extract shown above, Ant will search for all PNG graphics files ("*.png") in the *resources* folder in the build environment and add them to the zip file so that they can be unzipped into the *images* folder in the Software AG Runtime environment.

To build the deployment files, you can use either of the following methods:

To build the deployment files (method 1)

1 In Eclipse, select the *build.xml* file in the Package Explorer view.

2 In the context menu, click **Run As > Ant Build...**.

3 Ensure that the options are set for "Clear Environment", "Compile Sources", "Create JAR file", "Create ZIP file".

4 Click **Run**.

To build the deployment files (method 2)

This method assumes that you have Apache Ant installed as an executable program on your computer.

1 Open a command prompt window.

2 Go to the *demos\WelcomePage* folder.

3 Enter the command `ant clean`.

4 Enter the command `ant`.

In both methods, the Ant tasks defined in *build.xml* are processed. Ant builds a new jar file *demos\WelcomePage\lib\SagBlueWelcomePage.jar*, containing all of the Java classes required for the Software AG Runtime environment. It also build a zip file *demos\WelcomePage\lib\SagBlueWelcomePage.zip*, containing the jar file and all required PNG graphics. The name "SagBlueWelcomePage" comes from the definition of the property `projectName` in the file *build.properties*.

Deploying the demo to Software AG Runtime

To deploy the demo Welcome page to Software AG Runtime, you need to copy the Java classes and icons of the demo Welcome page to the Software AG Runtimet environment, and update the Software AG Runtime *plugin.xml* file. To do this, proceed as follows:

To deploy the demo Welcome page to Software AG Runtime

1 Stop Software AG Runtime.

2 Unzip the zip file created by the Ant build into `<RuntimeDir>`*workspace**webapps**PluggableUI**CentraSiteControl* directory.

This will copy the jar file created by Ant into the folder `<RuntimeDir>`*workspace**webapps**PluggableUI**CentraSiteControl**lib* and the PNG files into the folder `<RuntimeDir>`*workspace**webapps**PluggableUI**CentraSiteControl**images*.

Or:

As an alternative to using the zip file, you can just copy the jar file from the Ant build into `<RuntimeDir>`*workspace**webapps**PluggableUI**CentraSiteControl**lib* and the PNG files into `<RuntimeDir>`*workspace**webapps**PluggableUI**CentraSiteControl**images*.

3 As indicated in the section **Updating the *plugin.xml* configuration file**, the *plugin.xml* file in the Software AG Runtime environment must be updated to point to the Java classes of the customized Welcome page. The file *plugin.xml* in the folder *demos**WelcomePage**resources* contains the elements that must be updated in the *plugin.xml* file for Software AG Runtime.

Copy the entries manually from *demos**WelcomePage**resources**plugin.xml* to the *plugin.xml* file under Software AG Runtime. Remember to comment out the original entries for the standard Welcome page when you copy in the new entries.

4 Restart Software AG Runtime.

Displaying the demo Welcome page

After you have deployed the demo to the Software AG Runtime environment, as described in the previous section **Deploying the demo to Software AG Runtime**, and restarted Software AG Runtime, the demo Welcome page will be visible when you start CentraSite Control.

Special Programming Techniques

This section summarizes some of the techniques you might find useful when creating your own customized welcome page. You can find code examples of the techniques in the *demos\WelcomePage* folder.

Technique	Code Example in *demos\WelcomePage* folder
Activate the "Advanced Search" page.	AdvancedSearchItem.java
Activate the "Keyword Search" page.	KeywordSearchItem.java
Start the "My Account" dialog.	UserPreferencesItem.java
Start the "Add Asset" dialog.	CreateAssetItem.java
Start the "Import" dialog.	ImportWsdlFileItem.java
Activate "My CentraSite" and show "Assets I Provide".	MyFavoriteItem.java
Open the external website "http://communities.softwareag.com/centrasite".	CentraSiteCommunityItem.java
Open the external website "http://www.centrasite.com ".	DeveloperCommunityItem.java
Open the external website "http://documentation.softwareag.com/default.htm".	OnlineDocumentationItem.java
Create an empty line in a widget.	EmptyItem.java
Create a dotted dividing line.	SeparatorItem.java
Create a column (list) with all available date formats.	DateFormatsColumn.java
Select a specific date format from a list.	DateFormatItem.java
Create a column (list) with all available languages.	LanguagesColumn.java
Select a specific language from a list.	LanguageItem.java

III Customizing Content Pages

3 Customizing Content Pages

Extension Points

An extension point is characterized by the following properties:

- An ID by which it can be referenced.
- An interface to be implemented by plug-ins. In most cases there is also an abstract base class available that implements the interface. It is recommended to extend this class for your own extensions.
- Names of properties to be provided by a plug-in.
- Optionally, it may be related / compared to a corresponding extension point offered in an Eclipse environment.

An extension point provides the name of a class that implements the interface and property values. In general, if there is an abstract base class, its usage is strongly encouraged.

The available extension points are described in the following sections.

I18N for Layouts

Usage	Use this when the layout of a plug-in needs to be localized.
Attributes	■ point="com.softwareag.cis.plugin.i18n" ■ id ■ class ■ project (name of the plug-in directory) ■ prefix (as used by messages) ■ file (name of the property file to be used)
Interface	I18NHandler
Standard class	Common18NHandler
Processing	Class `I18NManager` inside *PluggableUI* handles this extension point. If an I18Message or a text ID in a layout definition (as created using the Application Designer) refers to a source ID that starts with the given prefix, the I18Manager will attempt to resolve this reference using the given property file. In the case of a text ID, the corresponding layout must be part of the plug-in whose directory is indicated by the `project` attribute.
Provided by	PluggableUI

Example	`<extension point="com.softwareag.cis.plugin.i18n"`
	`  id="CentraSiteControl"`
	`  class="com.softwareag.cis.plugin.ext.Common18NHandler"`
	`  project="CentraSiteControl"`
	`  prefix="INMCS"`
	`  file="com.centrasite.control.adapters.util.INMMessages"`
	`  >`
	`</extension>`

Parameters for Plug-ins

Usage	■ Use this to get parameters for a plug-in.
Attributes	■ point="com.softwareag.cis.plugin.parameter" ■ id ■ value
Interface	No interface to be implemented.
Processing	Use the plug-in call `ApplicationContext.getParameter()` to obtain value.
Provided by	*PluggableUI*

Example	`<extension point="com.softwareag.cis.plugin.parameter"`
	`id="welcomePageDefault"`
	`value="true">`
	`</extension>`

ConnectionHandler - Logon and Logoff / Exit

Usage	■ Use at the start or end of a session of CentraSite Control .
Attributes	■ point="com.softwareag.cis.plugin.connectionHandler" ■ id ■ value
Interface	ConnectionHandler ■ void init (CommonAdapter ca) ■ void connect (Credentials c, CommonAdapter ca) throws Exception ■ void notifyConnected (CommonAdapter ca) ■ boolean isConnected() ■ void prepareDisconnect (CommonAdapter ca) throws Exception ■ void disconnect (CommonAdapter ca);
Processing	■ Logon: ■ Obtain credentials from the login screen ■ Call connect(Credentials) for each extension ■ If an exception occurs: ■ Show a popup with the exception ■ Disconnect each extension which is already connected ■ Restart ■ If all successful: start the workplace ■ Logoff: ■ Call `prepareDisconnect()` for each extension ■ If an exception occurs: ■ Show a popup with the exception ■ Done ■ Disconnect each extension which is already connected by calling the `disconnect()` method
Provided by	*PluggableUI*

Example	<extension point="com.softwareag.cis.plugin.connectionHandler" id="login" ↩ class="com.centrasite.control.ext.CentraSiteConnectionHandler"> </extension>

Perspectives

Perspectives allow certain predefined screen layouts to be stored. When several perspectives are defined, it is possible to switch from one to the other easily.

The **Perspective** button will only be shown if more than one perspective is available. When you click the button, a popup dialog appears, which allows you to select the required perspective.

The perspective can be switched in two ways:

- Select one or more rows (perspectives) and click **OK**.
- Double click a single row.

Multiple perspectives will be represented in a way that the union of the corresponding topics is displayed on the right hand side. The header will be changed depending on the perspective to which the currently selected topic belongs.

The following features are provided for perspectives:

- A fixed set of perspectives as configured via extensions. You can switch a perspective via the **Select Perspective** dialog.
- A fixed set of topics per perspective. The association between topics and the corresponding perspective is established via the plug-in configuration file. Each declaration of a topic extension must contain a reference to the ID of the associated perspective extension.
- A perspective may contribute the following components:
  - A name and an icon being used to represent the perspective in the **Select Perspective** dialog.
  - An `ICONLISTInfo` object used to create a toolbar in the header frame. This can be suppressed if the perspective's `supportsViews()` method returns "false".
  - The label and valid values for the **View** list box. This can be suppressed.
  - A tailored layout to be used as the workplace background. This will only be used if the perspective is used as the initial perspective (see the section *Setting the Preferred Plug-in and Order of Plug-ins* for more information).

Usage	■ Each plug-in may contribute a perspective to contain its own topics or the topics of other plug-ins (see next section)
Attributes	■ point="com.softwareag.cis.plugin.perspective" ■ id ■ class
Interface	Perspective ■ String getTitle()

	(used in dynamically generated **Select Perspective** dialog) ■ String getImageURL() (used to represent a perspective by an icon in the **Select Perspective** dialog) ■ Toolbar: ■ ICONLISTInfo getToolbar() ■ Logo ■ String getLogoImageURL () (used for header frame) ■ Handling of the **View** listbox: ■ String getViewLabel() ■ String[] getViewValues() ■ String getView() (returns the currently selected view) ■ void setView(String view) (called when the user changes the view selection) ■ Default layout used for perspective background ■ String getWorkplaceDefaultLayout(); (used for background of workplace if no activity is opened)
Abstract base class	AbstractPerspective
Provided by	*PluggableUI*

Example	(*CentraSite Control/plugin.xml*)
	`<extension point="com.softwareag.cis.plugin.perspective"`
	`id="controlPerspective"`
	`class="com.centrasite.control.ext.ControlPerspective">`
	`</extension>`
	`<extension point="com.softwareag.cis.plugin.topic"`
	`id="registry"`
	`perspective="com.centrasite.control.controlPerspective"`
	`class="com.centrasite.control.ext.ImportantTypesTopic">`
	`</extension>`

If a perspective is selected for display, all topics belonging to that perspective become visible. If one of these perspectives is selected in the navigation pane, the content of the header frame is adjusted with respect to the toolbar, the View listbox and the visible logo.

Topic

Usage	■ Add a topic in the navigation view.
Attributes	■ point="com.softwareag.cis.plugin.topic" ■ id ■ perspective (see the section *Perspectives*) ■ class
Interface	Topic ■ String getImageURL() ■ boolean isVisible() (used when switching views)
Abstract base class	AbstractTopic
Processing	■ When starting the user interface, a topic is added to the active perspective for each known extension that refers to the perspective. ■ The first topic is selected. ■ When switching to a different topic, replace the content of the HEADER frame according to the data provided by the corresponding perspective.
Provided by	*PluggableUI*

Example	`<extension point="com.softwareag.cis.plugin.topic"`
	`id="registry"`
	`perspective="com.centrasite.control.perspective"`
	`class="com.centrasite.control.ext.ImportantTypesTopic">`
	`</extension>`

Command for Item

Usage	■ Add a command to a menu.
Attributes	■ point="com.centrasite.control.itemCommand" ■ id (default: name of implementing class) ■ class
Interface	ExtensionCommand ■ boolean appliesTo (Item) ■ String getName() ■ String getImageURL() ■ int getCategory() (used for grouping of commands) ■ abstract void execute(ActionContext actionContext)
Abstract base class	AbstractExtensionCommand
Processing	■ When a list of commands for a menu item is retrieved (e.g. for context menu or toolbar), the following steps are performed for each known extension: ■ create an instance of class and invoke `appliesTo(Item)`. ■ If "true" is returned, the command is added to the list.
Provided by	*CentraSite Control*

Example	`<extension point="com.centrasite.control.itemCommand"`
	`id="test"`
	`↩`
	`class="com.centrasite.control.extpt.junit.DisplayRegObjKeyCommand">`
	`</extension>`

Bulk Command for Items

Usage	■ Add a command to a menu in which bulk actions are permitted.
Attributes	■ point="com.centrasite.control.itemBulkCommand" ■ id (default: name of implementing class) ■ class
Interface	ExtensionCommand ■ boolean appliesTo (Item) ■ String getName() ■ String getImageURL() ■ int getCategory() (used for grouping of commands) ■ abstract void execute(ActionContext actionContext)
Abstract base class	AbstractExtensionCommand
Processing	■ When a list of commands for a menu item is retrieved (e.g. for context menu or toolbar), the following steps are performed for each known extension: ■ create an instance of class and invoke `appliesTo(Item)`. ■ If "true" is returned, the command is added to the list.
Provided by	*CentraSite Control*

Example	`<extension point="com.centrasite.control.itemBulkCommand"`
	`id="test"`
	`↩`
	`class="com.centrasite.control.extpt.junit.DisplayRegObjKeyCommand">`
	`</extension>`

Add Property

Usage	■ Add a property to a registry object.
Attributes	■ point="com.centrasite.control.registryObjectProperty" ■ d ■ class ■ (boolean) visible by default
Interface	ExtensionPropertyAccessor ■ boolean appliesTo (String objectTypeQName, Connector con) ■ String getDisplayName(Locale locale) ■ String getDescription(Locale locale) ■ String getInternalName() ■ boolean getVisibleByDefault() ■ String getValue(Item item) throws Exception ■ void setValue(Item item, String value) throws Exception
Abstract base class	AbstractPropertyAccessor (must be explicitly implemented)
Processing	■ When opening a report, all extensions are checked whether they want to contribute. ■ The corresponding accessors are added to the report.
Provided by	CentraSite Control

Example	`<extension point="com.centrasite.control.registryObjectProperty"`
	`id="test"`
	`↩`
	`class="com.centrasite.control.extpt.junit.LastModifiedPropertyAccessor">`
	`</extension>`

Note: Additional columns might also show up in upper table of the **General** tab.

Tab in Detail View

Usage	■ Add a tab in the detail view of an object.
Attributes	■ point="com.centrasite.control.detailViewTab" ■ id ■ class
Interface	DetailViewTab ■ String getTitle() ■ String getImageURL() ■ String getLayout() ■ void initAdapterFor (Item, DetailViewTabAdapter) ■ protected String getAdapterClass(); ■ boolean appliesTo (Item) (If this is returned, the tab will be displayed for the corresponding Item if `isVisible()` returns true as well, otherwise the tab will not be displayed) ■ void setDetailsTabContext (DetailTabContext) ■ boolean isVisible(Item)
Abstract base class	AbstractDetailViewTab
Processing	■ If the detail view for an Item is opened, it is checked for each known extension. ■ Create an instance of class and invoke `appliesTo(Item)`. If "true" is returned, `getLayout()` is invoked and the layout is added as a tab. The respective adapter will be created implicitly by the Application Designer when processing the layout. ■ The title of the tab is set with the result from calling `getTitle()`. ■ Currently, items on tabs are not supported. Hence, the result from `getImageURL()` is ignored.
Provided by	*CentraSite Control*

Example	`<extension point="com.centrasite.control.detailViewTab"`
	`id="lifecycle"`
	`class="com.centrasite.control.lifecycle.LifeCycleDetails"`
	`>`
	`</extension>`

Add Source of Notification

Usage	■ Add a source of a notification.
Attributes	■ point="com.centrasite.control.addRowToMyNotifications" ■ id (default: name of implementing class) ■ class
Interface	ReportExtensionItemsProvider ■ Collection getItems() throws Exception; ■ void setConnector(Connector connector); ■ boolean isContributedItem(Item item); ■ String getChangedImageURL (Item item);
Abstract base class	AbstractReportExtensionItemsProvider
Processing	■ The extension is initialized via the `setConnector()` method. ■ Obtain all items to be added to the list of items with pending notification via the `getItems()` method. ■ `isContributedItem()` can be used to check whether this extension has contributed the given item via `getItems()`. ■ `getChangedImageURL()` is used to control the icon representing the reason for the notification.
Provided by	*CentraSite Control*

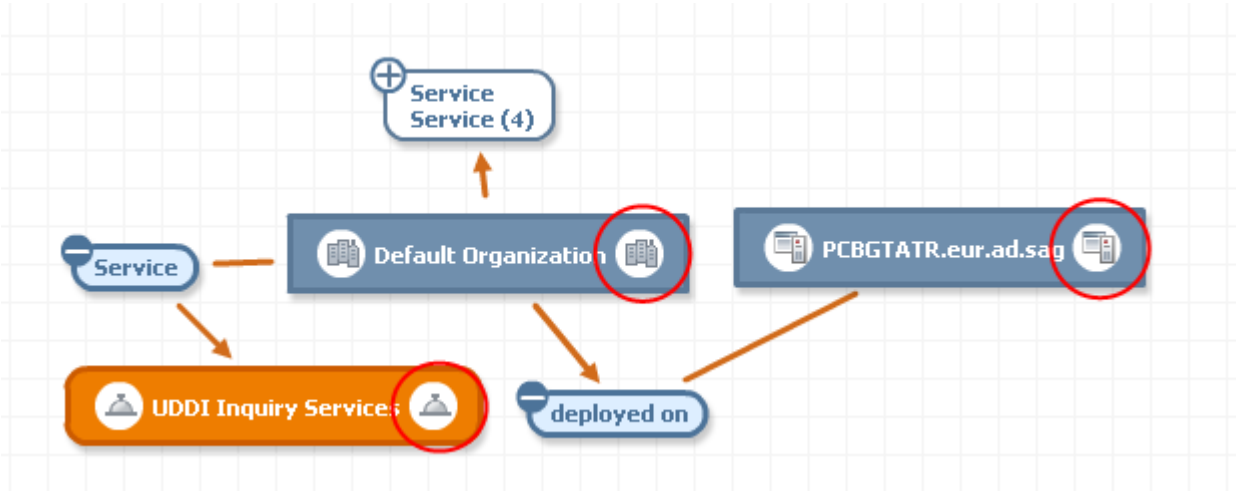
Example	`<extension point="com.centrasite.control.addRowToMyNotifications"`
	`id="MyNotificationsApprovalItemsProvider"`
	`class="com.softwareag.centrasite.control.lms.ext.`
	`MyNotificationsApprovalItemsProvider">`
	`</extension>`

Impact Analysis: NodeDecorator

Usage	■ Change the visual representation of registry objects
Attributes	■ point="com.centrasite.control.assocNavigatorNodeDecorator" ■ id (default: name of implementing class) ■ class
Interface	NodeDecorator ■ String getImageURL(Item)
Abstract base class	(none)
Processing	■ If the item is to be rendered in Impact Analysis, check all known extensions to determine if they contribute to the item's visualization; ■ If `getImageURL(item)` returns null: check for the next extension ■ otherwise: use the URL being returned for secondary icon within visualization of node in graphical impact analysis.
Provided by	CentraSite Control

Example	`<extension ↩`
	`point="com.centrasite.control.assocNavigatorNodeDecorator"`
	`id="ExternalLinkNodeDecorator"`
	`class="com.centrasite.control.ext.ExternalLinkNodeDecorator">`
	`</extension>`

The following picture illustrates how ExternalLinks objects are decorated with icons representing the type of the object they are referencing:



Append Root Node to Topic

Usage	■ Append a root node to an existing topic.
Attributes	■ point="com.centrasite.control.topicItems" ■ id ■ class
Interface	TopicItems ■ boolean appliesTo (Topic) ■ Collection getItems()
Abstract base class	AbstractTopicItems
Processing	■ For each topic whose implementation class is derived from a class named `BaseTopic` (true for all topics contributed by CentraSite Control) it is checked whether there are any extension for the topicItems extension point. Each extension whose `appliesTo()` method returns "true", the collection of *Item* objects returned by `getItems()` is appended to the set of root nodes for the corresponding topic.
Provided by	CentraSite Control

Example	`<extension point="com.centrasite.control.topicItems"`
	`id="filesystem"`
	`topic="com.centrasite.control.administration"`
	`class="com.centrasite.control.ext.junit.FileSystemTopicItems">`
	`</extension>`
	Note: Here, the *FileSystemTopicItems* extension is an extension of the base class `AbstractTopicItems` whose `appliesTo()` method returns "true" if the value of the `topic` attribute matches the ID of the topic being passed.

Replace Standard Detail View by another Editor

Usage	■ Add an editor that can be configured per object type, even per object instance.
Attributes	■ point="com.centrasite.control.itemEditor" ■ id (default: name of implementing class) ■ class
Interface	ItemEditor ■ public boolean appliesTo (Item item, Connector connector); ■ public String getLayout(); ■ public String getTitle(Item item); ■ public String getAdapterClass(); (must return a class implementing the *ItemEditorAdapter* interface)
Abstract base class	AbstractItemEditor
Processing	■ If `appliesTo()` returns true, the editor will be used when opening the detail for the item being passed: ■ The given adapter class will be instantiated and initialized. ■ The given layout (=pageURL) is opened in the CONTENT frame on the right hand side. ■ The title returned by `getTitle()` is used as the label for the activity.
Provided by	CentraSite Control

Example	`<extension point="com.centrasite.control.itemEditor"`
	`id="DataType"`
	`class="com.softwareag.centrasite.ext.DataTypeEditor">`
	`</extension>`

Extend Search Dialog by Additional Conditions

Usage	■ Extend the search dialog by additional conditions, for example, you can add specific search predicates for your own object types.
Attributes	■ point="com.centrasite.control.searchPredicate" ■ id ■ class
Interface	PredicateEditor ■ Predicate getPredicate() (Get predicate to be added by this editor) ■ String getLayout() (Get URL of layout to be rendered) ■ String getAdapterClass() (Get name of adapter class to be used for rendering, must be a subclass of AbstractPredicateAdapter) ■ String getPredicateClass() (Get name of predicate class to be used for rendering, must be a subclass of AbstractPredicate) The interface Predicate (many implementing classes are already available in *CentraSiteUtils.jar*) with its abstract subclass AbstractPredicate has the following methods ■ boolean appliesTo(String objectTypeValue, CentraSiteQueryManager qm) (Check whether this predicate applies to objects of given object type) ■ String getInternalType () (Get unique internal string representation of type of predicate; not to be localized. You may use a namespace-like notation for your own.) ■ String getDisplayType () (Get human readable localized representation of type of predicate; CentraSite Control will // display it on the left hand side in the **Add Condition** dialog) ■ void validate() throws InvalidPredicateException (Validate parameters set for this predicate. The InvalidPredicateException should contain a localized message text) ■ String getDisplayString () throws Exception

	(Get human readable localized string representation of predicate including values predicate; CentraSite Control will display it in the condition table in the header section of the **Search Registry** dialog) ■ boolean requiresEnterpriseLicense() (Check whether this predicate requires an Enterprise license) ■ void addTo (BusinessQuery bq) throws JAXRException (Add contribution of predicate to given BusinessQuery. This is the worker method applying the predicate to the search result.) AbstractPredicate also provides implementations for the following methods ■ Used for I18N support ■ Locale getLocale() ■ Void setLocale(Locale) ■ used for persisting predicates as part of queries. ■ String toXML (); ■ void setFromDom(Element predicateEement, Connection connection) ■ used to initialize the search dialog with readonly predicates / conditions which can neither modified or removed: ■ void setReadOnly(boolean readOnly); ■ boolean isReadOnly();
Abstract base class	AbstractPredicateEditor
Processing	■ When you click the appropriate button, this invokes the user-defined Adapter (layout) screen for entering custom search related settings. ■ Create an instance of the class ■ Execute
Provided by	CentraSite Control

Example	`<extension point="com.centrasite.control.searchPredicate"`
	`id="ObjectTypePredicateEditor"`
	`class="com.centrasite.control...ObjectTypePredicateEditor">`
	`</extension>`

Download Documents

There is a menu entry in each asset's context menu that allows you to create a zipped archive of the asset and optionally any attached documents, and to download the zipped archive to the file system. You can customize the way in which the download feature behaves:

- You can make the download entry in the context menu visible or invisible for users with the Guest role.
- You can change the text string displayed in the context menu.
- You can change the format of the zipped archive.

Making the download menu entry visible/invisible for guest users

If a user with the Guest role can access an asset and view its context menu, the context menu entry **Download Document** is visible by default. You can specify whether this entry is visible or invisible for such users as follows:

To make the download menu entry visible/invisible for guest users

1 Locate the configuration file *plugin.xml* in the `<RuntimeDir>\workspace\webapps\PluggableUI\CentraSiteControl` directory.

2 Open the file and locate the entry:

```
<extension point="com.softwareag.cis.plugin.parameter"
 id="guestCanDownloadDocuments" value="true" />
```

3 To make the context menu entry invisible for guest users, change "true" to "false" and restart Software AG Runtime. Similarly, if the context menu entry is already invisible and you want to make it visible for guest users, set the value to "true" and restart Software AG Runtime.

Changing the text string displayed in the context menu

The text string displayed in the context menu is by default "Download Document". If you wish to change this, you can do so by extending the CentraSite Control functionality via the extension point `downloadDocumentCommand`. This extension point has the following definition:

Usage	■ Change the text string displayed in the context menu for downloading an asset.
Attributes	■ com.centrasite.control.downloadDocumentCommand ■ id (default: name of implementing class) ■ class
Interface	See the sample code.

Abstract base class	AbstractExtensionCommand
Provided by	*CentraSite Control*
Example	See the sample code.

To change the text displayed for the context menu, your implementation of the extension point must define a method `getName()` of type `String`. The return value of this method is the text that will be displayed in the context menu.

You can find sample code for defining the extension point in the file *DownloadDocumentCustomCommand.java* that is provided in the *demo* folder under the CentraSite installation folder.

Changing the format of the zipped archive

By default, the zipped archive contains the folder structure of the asset and its attached documents. If you wish to change this, you can do so by extending the CentraSite Control functionality via the extension point `downloadDocumentCommand`. The definition of the extension point is given above.

To change the format of the zipped archive, your implementation of the extension point must define a method that extends the base class `DownloadOperation`.

You can find sample code for defining the extension point in the files *DownloadDocumentCustomCommand.java* and *DownloadCustomOperation.java* that are provided in the *demo* folder under the CentraSite installation folder.

Attach Documents

Some assets include file-related attributes that allow you to attach supporting document(s) such as programming guides, sample code and script files with the asset. When trying to attach a supporting document with an asset, CentraSite Control displays the available documents underneath their respective organization directory by default. If you wish to change this (that is, simply display the documents by the side of its organization directory), you can do so by extending the CentraSite Control functionality via the extension point `attachDocumentCommand`.

To customize the document layout:

1 Locate the configuration file *plugin.xml* in the *<RuntimeDir>\workspace\webapps\PluggableUI\CentraSiteControl* directory.

2 Open the file and locate the entry:

```
<extension point="com.softwareag.cis.plugin.parameter" ↩
id="isCustomAttachDocument" value="false" />
```

```
<extension point="com.centrasite.control.attachDocumentCommand" ↩
id="AttachDocumentCustomCommand" ↩
class="com.centrasite.control.extpt.AttachDocumentCustomCommand" />
```

Where `com.centrasite.control.extpt.AttachDocumentCustomCommand` is the name of the abstract base class that implements the interface.

3 To define your custom document layout, change "false" to "true" and restart Software AG Runtime. Similarly, if the document layout is already customized and you want to revert back to the standard layout, set the value to "false" and restart Software AG Runtime.

Usage	■ Use this to define a custom layout of the documents while attaching to an asset via the **Attach Document** dialog.
Attributes	■ com.centrasite.control.attachDocumentCommand ■ id (default: name of implementing class) ■ class
Interface	See the sample code.
Abstract base class	AbstractExtensionCommand
Processing	■ When you click the appropriate button, this invokes the user-defined Adapter (layout) screen displaying all documents that are available for attaching to an asset. ■ Create an instance of the class ■ Execute
Provided by	*CentraSite Control*

Example	`<extension point="com.centrasite.control.attachDocumentCommand" ↩`
	`id="AttachDocumentCustomCommand" ↩`
	`class="com.centrasite.control.extpt.AttachDocumentCustomCommand" />`

You can find sample code for defining the extension point in the files *AttachDocumentCustomCommand.java* , *AttachFile.xml* and *AttachFileAdapter.java* that are provided in the *demo* folder under the CentraSite installation folder.

Activating the IDE

The CentraSite distribution kit contains an IDE (integrated development environment) that you can use to create and design a layout page. The IDE is a web application whose clients run inside a web browser. The URL (assuming installation defaults) to start the IDE on a machine where CentraSite is installed is:

http://localhost:53307/PluggableUI/HTMLBasedGUI/workplace/ide.html

The IDE is deactivated by default. In order to activate the IDE, set the attribute `plugindevelopment` in the file *cisconfig.xml* to "true". This file is located in the CentraSite Control web application (in the Application Server or Software AG Runtime location) in the folder *cis/cisconfig*.

The following example illustrates the required configuration setting:

```
<cisconfig plugindevelopment="true" ...>
 ...
</cisconfig>
```

Security Considerations

When activated, the IDE and included development tools do not require further authentication. The following example illustrates the `security-constraint` and `login-config` elements to protect the IDE and development tools with the HTML basic authentication method.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Plugin Development</web-resource-name>
        <url-pattern>/HTMLBasedGUI/workplace/*</url-pattern>
        <url-pattern>/servlet/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>developer</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Plugin Development</realm-name>
</login-config>
```

In order to protect passwords transmitted in clear text between a browser and development tools running on the application server, it is recommended to protect the communication through the use of SSL. For information about SSL, refer to the section *Configuring Secure Communication between CentraSite Components* in the document *Basic Operations*.

Step-by-Step Guide

A step-by-step guide of how to create customized plug-ins for the CentraSite Control content pages is provided in **Appendix A: Step-by-Step Guide**.

IV Setting the Preferred Plug-in and Order of Plug-ins

4 Setting the Preferred Plug-in and Order of Plug-ins

You can adapt the URL used to invoke the pluggable user interface with a preferred plug-in by appending a query parameter such as

```
PLUGIN=com.centrasite.control
```

So the modified URL would be:

http://localhost:53307/PluggableUI/servlet/StartCISPage?PAGEURL=/PluggableUI/Login.html&PLUGIN=com.centrasite.control&LOCALE=en

The value of the `PLUGIN` parameter must match the value of the `id` parameter of a `<plugin>` root element in one of the plug-ins. This sets the preferred plug-in.

This implies that for all extensions for a specific extension point, the extensions belonging to the referenced plug-in will be first in order (normally the order is determined by the `order` attribute in the plug-in configuration file).

For any extension point, the order of the associated extensions is determined by the following properties:

- The processing order of the plug-ins is controlled by the value of the `order` attribute of `<plugin>` in the *plugin.xml* file. Plug-ins with a smaller value of the `order` attribute are processed first. The preferred plug-in is always processed first.
- The order of extensions, as configured in *plugin.xml*, for the associated extension point.

Depending on the extension point, the order of the extensions has a specific impact, for example:

- The login screen displayed when the user interface is started in the browser.
- The initial perspective shown after login.

V Installing and Uninstalling Plug-ins

5 Installing and Uninstalling Plug-ins

Directory Structure

The plug-in environment is contained in a directory structure under the installation directory *<RuntimeWebAppsDir>* of the Software AG Runtime. The document *Basic Operations* describes the location of this directory.

Under *<RuntimeWebAppsDir>**PluggableUI* we have the following structure:

```
WEB-INF/
    classes/
        log4j.xml
    lib/          //JARs

cis/

HTMLBasedGUI/

PluggableUI
     plugin.xml
     *.html

     accesspath/
     xml/         // layout definitions
     images/

CentraSiteControl
     plugin.xml
     *.html
     *_SWT.xml

     accesspath/
     xml/         // layout definitions
     images/      // icons
     lib/         // JARs
     classes/     // class files

MyPlugIn
     plugin.xml
     *.html
     *_SWT.xml

     accesspath/
     xml/         // layout definitions
     images/      // icons
     lib/         // JARs
     classes/     // class files
```

The structure includes a sample user-written plug-in *MyPlugIn* for illustration purposes.

Installing a Plug-in

A plug-in should be provided as a ZIP archive with the directory structure as given in the section *Directory Structure*.

The following actions need to be performed when installing a plug-in manually:

- Check for availability of other plug-ins being a prerequisite.
- Copy files (except the *plugin.xml* configuration file) into the directory structure shown in the section *Directory Structure*.
- Compile layout definitions.

Note: Using the plug-in may require a restart of Software AG Runtime.

Uninstalling a Plug-in

The following actions need to be performed to uninstall a plug-in manually:

- Before you uninstall a plug-in, ensure that it is not required by another plug-in.
- Remove the plug-in configuration file *plugin.xml*.
- Remove the plug-in directory, for example *MyPlugIn* as shown in the section *Directory Structure*.

Note: It might not be possible to remove files if they are in use, for example, while the application server is running.

The Plug-In Management Perspective

A separate *Plug-In Management* perspective offers the following functions:

Function	Description	Invoke via...
Install Plug-In	Import a ZIP-file containing all required files for a plug-in	Button in toolbar
Table of Plug-ins	Similar to the **About** dialog	Select a node in the **Plug-Ins** topic
Uninstall Plug-In	Check which other plug-ins rely on the plug-in to be uninstalled. If there are no dependencies, the plug-in is uninstalled.	Select the plug-in in the table and select the command from the context menu

Function	Description	Invoke via...
Compile Layouts	Required when the underlying Application Designer runtime is upgraded	Select plug-in in table and select command from context menu
Start the Application Designer layout editor		Button in toolbar

The Plug-In Management perspective is not visible by default. It is only visible if you set the preferred plug-in using `PLUGIN=com.softwareag.cis.plugin` in the URL that is used to start the GUI.

Example:

http://localhost:53307/PluggableUI/servlet/StartCISPage?PAGEURL=/PluggableUI/Login.html&PLUGIN=com.softwareag.cis.plugin&LOCALE=en

VI Special and Advanced Topics

6 Special and Advanced Topics

Icons

There are various optional references to icons which may be contributed by a plug-in. Most icons should be transparent GIFs unless stated otherwise in the table below. Here is a set of potential locations for contributing icons:

Context	Recommended Size in Pixels	Remarks
Plug-in icon appearing in the common **About** dialog	16x16	Transparent GIF
Bitmap for plug-in specific 2nd-level **About** dialog	None	May be also JPG or PNG file
Perspective icon in **Select Perspective** dialog	16x16	Transparent GIF
Header icon contributed by perspective	Height: 35. Width: depends on space required for toolbar and view listbox.	May be also JPG or PNG file
Icon representing an item in tree or table	16x16	Transparent GIF
Icon for command for an item (context menu or toolbar in detail view)	16x16	Transparent GIF

Class Loading

The Pluggable UI relies heavily on dedicated class loaders. Whereas the code for a normal web application is only loaded via the basic *WebappClassLoader* provided by Software AG Runtime, this class loader is only used for loading the classes resembling the PluggableUI base with the underlying Application Designer. The respective classes are loaded from the following directories below *<RuntimeWebAppsDir>\PluggableUI*:

```
WEB-INF/classes
WEB-INF/lib/*.jar
```

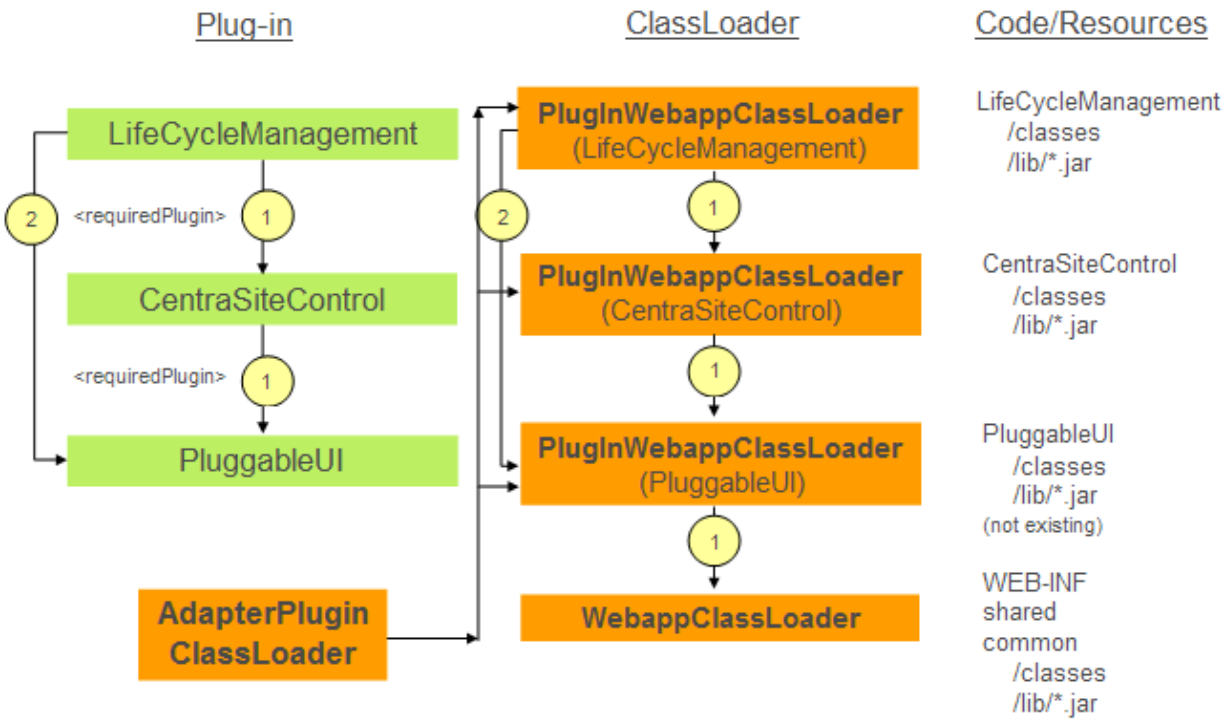
In addition, locations holding common or shared class or jar files are searched by the *WebappClassLoader* when attempting to resolve references to required classes.

Any code contributed by a plug-in is loaded by a corresponding instance of the *PlugInWebappClassLoader* from the following directories below *<RuntimeWebAppsDir>\PluggableUI*

```
plugInDir/classes
plugInDir/lib/*.jar
```

If resolution fails, the *PlugInWebappClassLoader* for the current plug-in will delegate class loading to the *PlugInWebappClassLoaders* for other plug-ins in the order as listed as `<requiredPlugin>` in the *plugin.xml* of the current plug-in recursively. Finally, if resolution via required plug-ins fails, the *PlugInWebappClassLoader* will delegate class loading to the *WebappClassLoader*.

The following picture illustrates the scenario of the LifeCycleManagement plug-in (representing any other 3rd party plug-in) that depends on the CentraSiteControl plug-in and the PluggableUI base infrastructure.



The *AdapterPlugInClassLoader* is used by ApplicationDesigner when resolving references to adapter classes found in layout definitions. The *AdapterPlugInClassLoader* will never load classes itself. Instead, it will ask all known *PlugInWebappClassLoaders* in an unspecified order whether they can load a required class.

Caution: You must avoid having the same adapter classes in more than one plug-in! Otherwise, various classloading related issues (ClassCastException, ClassNotFoundException, …) will result. Under normal conditions, fulfilling this restriction should not cause any problems.

In general, you should avoid having multiple locations contributing the same classes within the graph of locations spanned by the required plug-ins.

Multithreading and Synchronization

Normally, when executing the HTTP requests on behalf of a single Application Designer session, there is no parallel execution in multiple threads (unless the code contributed by a plug-in starts a thread on its own). Hence, access to objects or properties having a scope restricted to the session context does not require any synchronization.

However, when using global / static variables, this is no longer true: multiple active user sessions may be processed in parallel.

Warning

Avoid usage of global variables. They may lead to the following issues:

- Synchronization is required otherwise non-reproducible race conditions will result.
- Memory leakages: if global collections that grow for each session are used.
- Global references to JAXR-based RegistryObjects: a RegistryObject contains a reference to the JAXR-based Connection (including the underlying credentials) which had been used to load it. When resolving a secondary reference either of the following may happen:
  - If the connection is still open: credentials of another user will be used, thus causing a security hole.
  - If the connection is no longer open: a corresponding exception will be thrown ("trying to use a closed connection").

Nested Layouts

All adapter classes of a plug-in should not be just subclasses of *com.softwareag.cis.server.Adapter*. Instead, they should be derived from one of the following classes:

- *com.softwareag.cis.plugin.adapter.util.CommonAdapter* - for a plug-in that does not depend on CentraSite Control
- *com.centrasite.control.adapters.BaseAdapter* - for a plug-in that depends on CentraSite Control

`BaseAdapter` is a subclass of `CommonAdapter` and thus inherits certain properties. Among those is the implicit registration of all adapters as "known adapters" in the current session context. However, under certain circumstances it may happen that adapters for nested pages displayed using a *SUBCISPAGE* or *ROWTABSUBPAGES* control are not automatically deregistered when closing e.g. an activity displayed in the *CONTENT* frame on the right hand side of the workplace. This may lead to subsequent NullPointerExceptions.

Normally, de-registration is accomplished within the `destroy()` method in `CommonAdapter`. Hence, be careful when overriding this method in a subclass to call `super.destroy()`. In addition, you should override the `endProcess()` method in the adapter for the container layout, which should perform at least the following actions:

- call `super.endProcess()`
- call `CommonAdapter.removeKnownAdapter (subPageAdapter)` for each subPageAdapter

VII Javadoc Documentation of the APIs

7 Javadoc Documentation of the APIs

The HTML-based Javadoc documentation of the APIs provides details of the classes and methods described here. Please refer to the HTML Javadoc documentation for reference material.

You should only use the packages and classes that are explicitly mentioned in the documentation.

There are three sets of Javadoc documentation:

- PluggableUI (base architecture): *com.softwareag.cis.plugin*
- CentraSite Control User Interface: *com.centrasite.control...*
- CentraSite Control Backend: *com.centrasite.control...*

VIII Appendix A: Step-by-Step Guide

This section gives examples of how you can use the pluggable architecture to define and install your own plug-ins.

Section	Description
Appendix A1: Eclipse Prerequisites	Describes how to set up the Eclipse development environment in which you will define the sample plug-ins.
Appendix A2: Setting up the plug-in project	Describes how to set up the DemoPlugIn01 sample project.
Appendix A3: Plugging into CentraSite Control	Describes how to make the sample project known to CentraSite Control.
Appendix A4: Bring your own layouts to the screen	Describes how to design your own layout pages (detail views)

8 Appendix A1: Eclipse Prerequisites

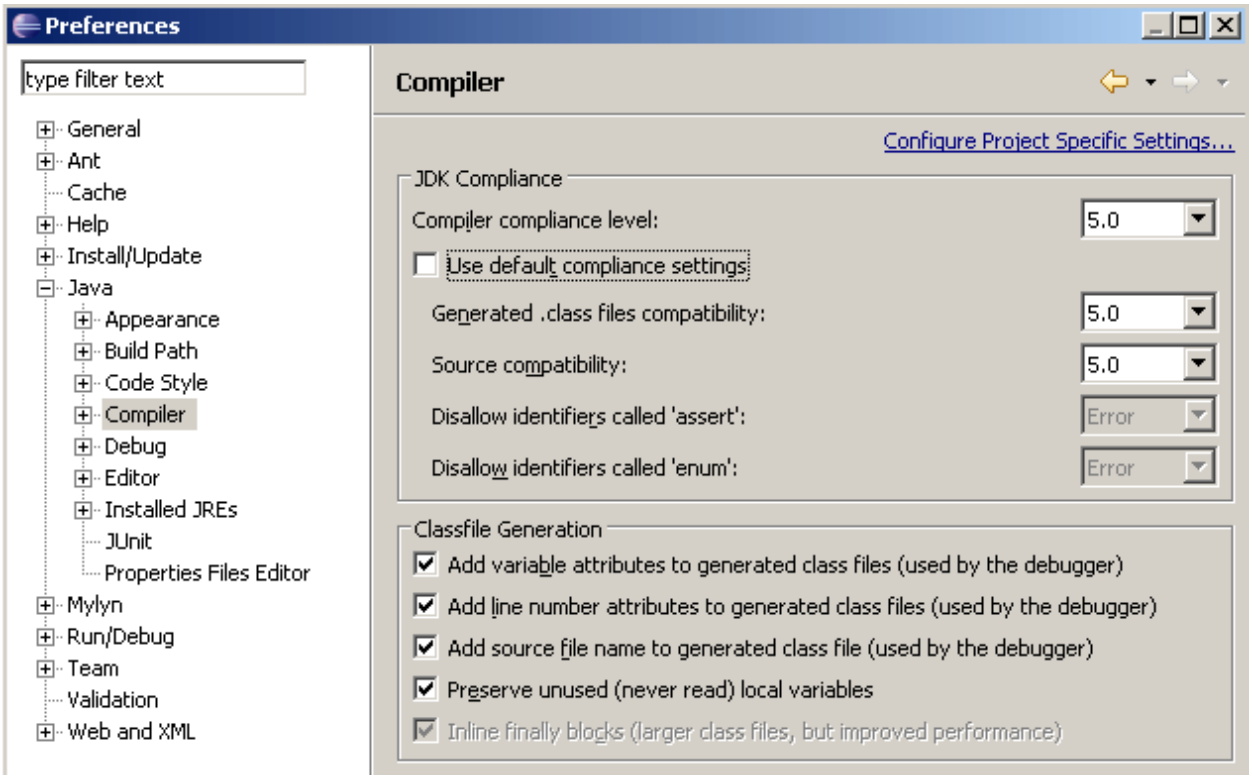
The descriptions in this chapter are based on a sample plug-in named *DemoPlugIn01*. We will show how to use Eclipse and standard CentraSite features in order to add the plug-in to CentraSite.

Before you start, ensure that you have a recent Eclipse version installed on your machine. Eclipse is available as a download from **http://www.eclipse.org/**.

In Eclipse, select **Window > Preferences > Java > Compiler** in order to configure usage of / compliance with the Java version currently supported by CentraSite.

The system requirements can be checked at **http://documentation.softwareag.com/**.

Example:

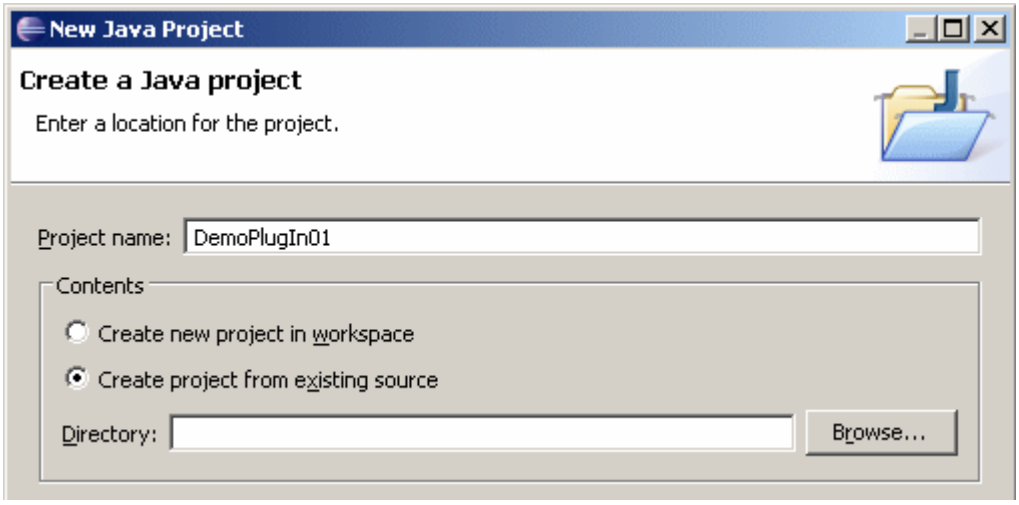


Click **Apply** to activate the settings. Eclipse will ask you to confirm the change, indicating that an internal rebuild is required. Reply **Yes**. The rebuild takes only a few seconds.

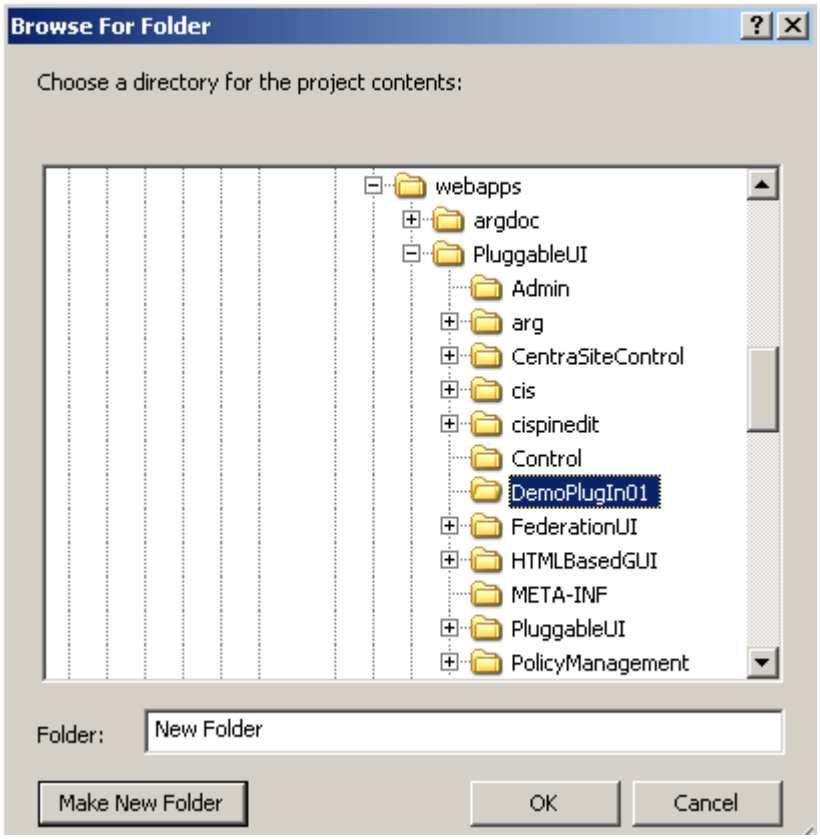
9 Appendix A2: Setting up the Plug-in project

Follow the steps below to set up the Java project for DemoPlugIn01:

1. Create a new Java project in Eclipse using **File > New > Project > Java Project**.
2. Specify "DemoPlugIn01" as the project name and check radio button labeled **Create project from existing source**.



3. Click the **Browse** button that is located right to the input field labeled **Directory**. The **Browse For Folder** dialog is displayed.
4. Within the **Browse For Folder** dialog, navigate to and click on the *PluggableUI* web application folder of the Software AG Runtime application. In the remainder of this document, this folder is indicated by `<PluggableUIFolder>`.
5. Click **Make New Folder**. This causes an entry *New Folder* to be created under *PluggableUI*. Select the entry *New Folder*, then from its context menu choose **Rename**, then enter the name *DemoPlugIn01*.



6. Click **OK**.
7. In the **New Java Project** dialog that becomes visible again, click **Finish**.

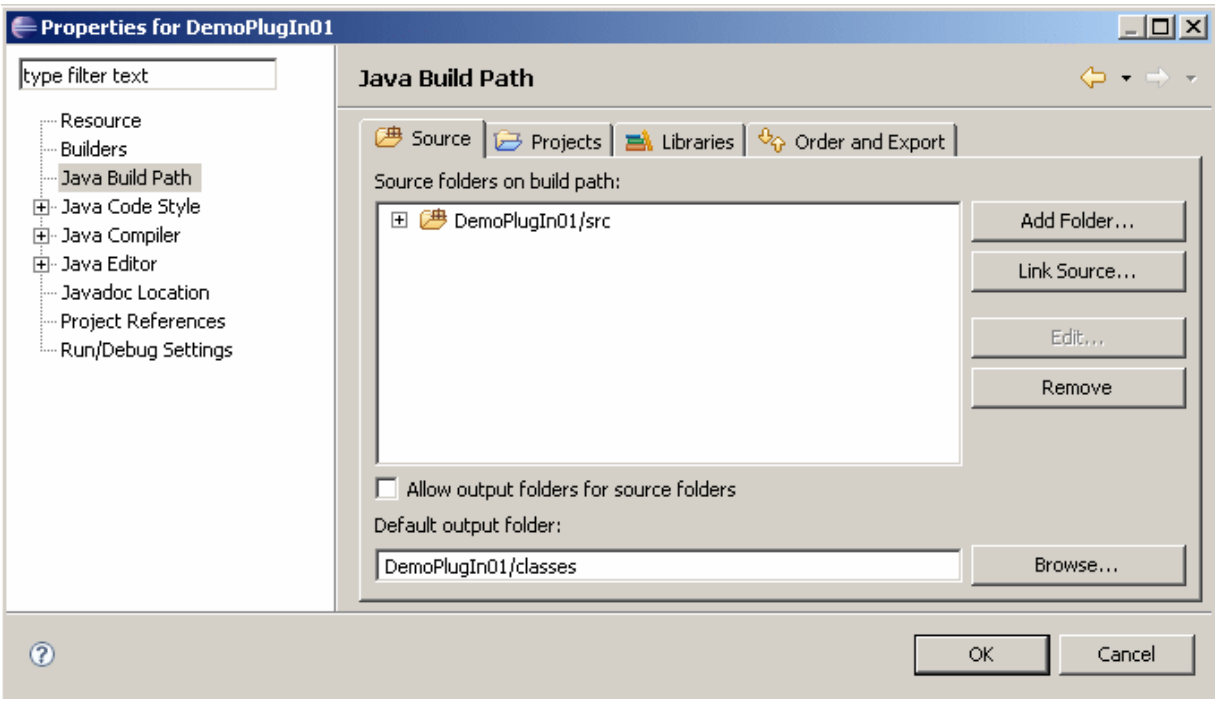
A new Java project called *DemoPlugIn01* has been created due to the previous actions. This project is now visible in the **Package Explorer** view in Eclipse.

This project needs to be adapted.

1. Create the following four subfolders of *DemoPlugIn01*:
   - accesspath
   - classes
   - images
   - xml

   To create each of these subfolders, choose **New > Folder** from the context menu of *DemoPlugIn01* in the package explorer, then type in the name in the **New Folder** dialog.
2. Create a source folder called *src*. You can create the source folder by choosing **New > Source Folder** from the context menu of *DemoPlugIn01*.
3. In the context menu of *DemoPlugIn01*, choose **Properties**.

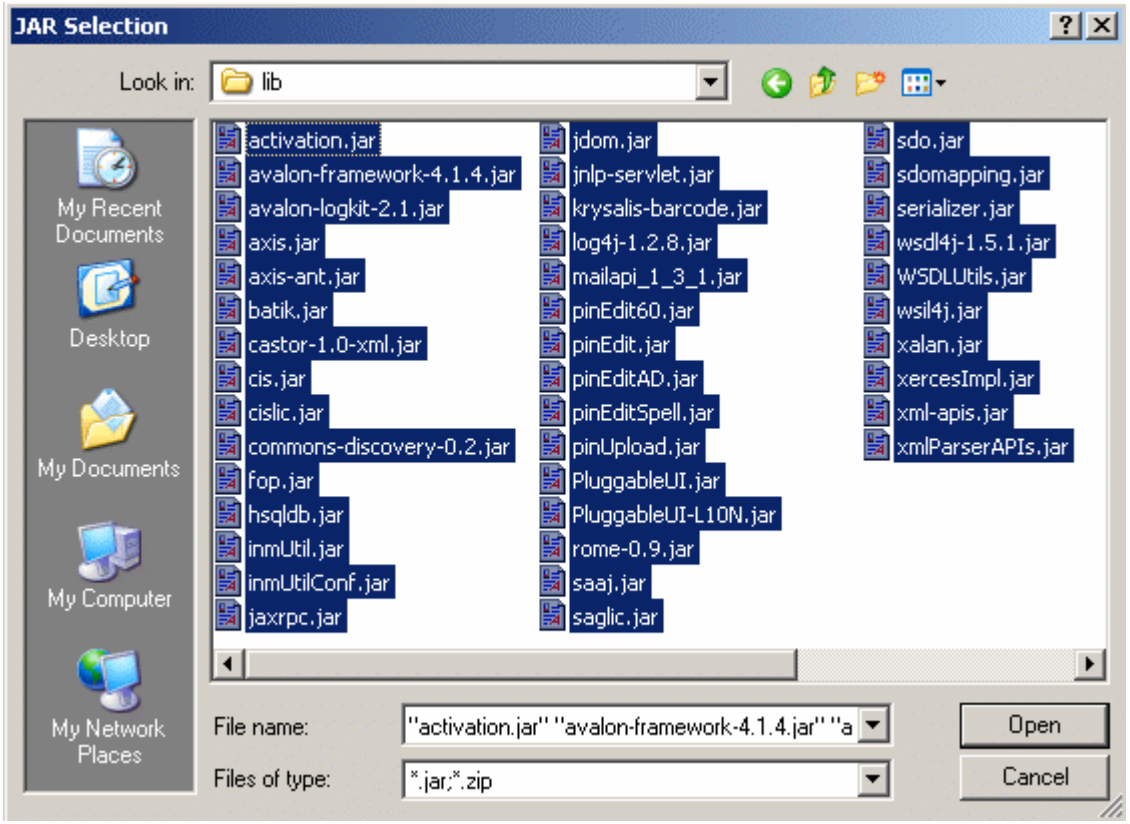
4. Select **Java Build Path** from the tree on the left.
5. Select the **Source** tab and enter the value "DemoPlugIn01/classes" in the field **Default output folder**.



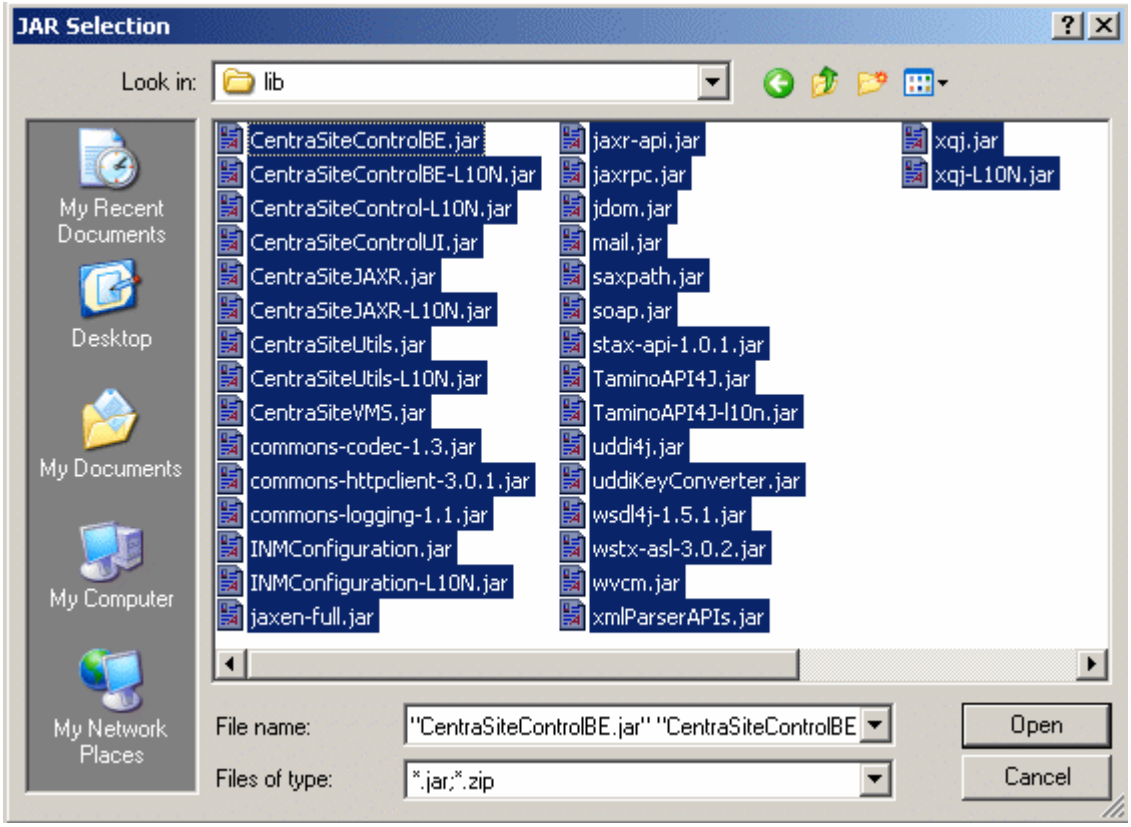
6. Switch to the **Libraries** tab.

   An entry for the JRE library should be visible. If you do not see this entry, click **Add Library** and select the JRE system library from the displayed list, then click **Finish**.
7. Click **Add External JARs**

   In the resulting **JAR Selection** dialog, navigate to `<PluggableUIFolder>`/*WEB-INF/lib* and open this folder.
8. Select all files, using for example the key combination Control-A, and click **Open**.

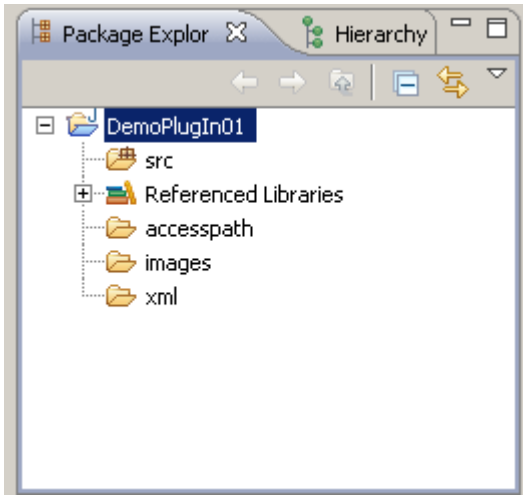


9. Click **Add External JARs** again.
10. In the **JAR Selection** dialog, navigate to `<PluggableUIFolder>`/*CentraSiteControl/lib*.
11. Again, select all files and click **Open**.



12. Click the **OK** button of the **Properties for DemoPlugIn01** dialog.

Your project should now look like this:

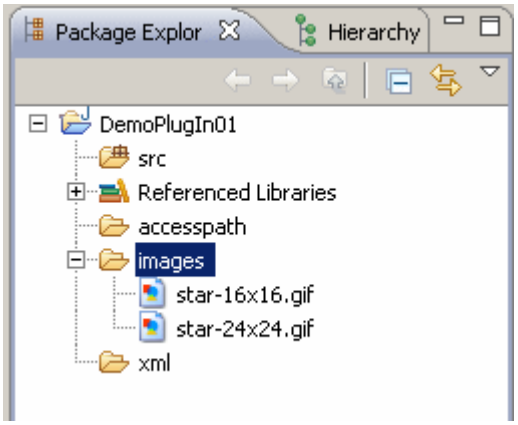


Perhaps you have noticed that the *classes* subfolder of the project *DemoPlugIn01* has disappeared from the display. This is normal because the Java Development Tools (JDT) of Eclipse suppress output folders from displaying by default (but they still exist on your hard disk).

Furthermore, the old output folder *bin* that has been created by the JDT when creating the Java project is not of any use for us, so you can delete it.

Later on we will need some icons for our plug-in. For now, let's just copy and rename some already existing icons from the CentraSite Control plug-in and use them instead:

1. Using the Windows Explorer, navigate to `<PluggableUIFolder>`/*CentraSiteControl/images*.
2. Copy the files *myFavorites.gif* and *myFavorites24x24.gif* to the *images* subfolder of our Java project *DemoPlugIn01*.
3. In *DemoPlugIn01/images*, rename the file *myFavorites.gif* to *star-16x16.gif* and rename *myFavorites24x24.gif* to *star-24x24.gif*. Use the command **File > Rename** in the Eclipse menu to do this.
4. In Eclipse, refresh the display of the package explorer. The names of the two images should now be visible.



10 Appendix A3: Plugging into CentraSite Control

We have now created a Java project inside the PluggableUI web application. However, there is one missing piece that tells CentraSite Control that this folder contains a plug-in: the plug-in configuration file. Amongst other things, the plug-in configuration file contains the information about where a plug-in plugs into in CentraSite Control.

The idea of using plug-ins to extend an application's functionality is quite simple and meanwhile well established by the Eclipse platform. The CentraSite Control software provides so-called extension points. These are positions in the program logic of the CentraSite Control program where functionality can be added by a plug-in. Every time the program flow comes to such an extension point, a search for plug-ins that extend CentraSite Control at this point takes place and the code provided by the plug-ins is invoked.

Let's convert our arbitrary Java project to a CentraSite Control plug-in folder by providing a plug-in configuration file. To do so, follow the steps below:

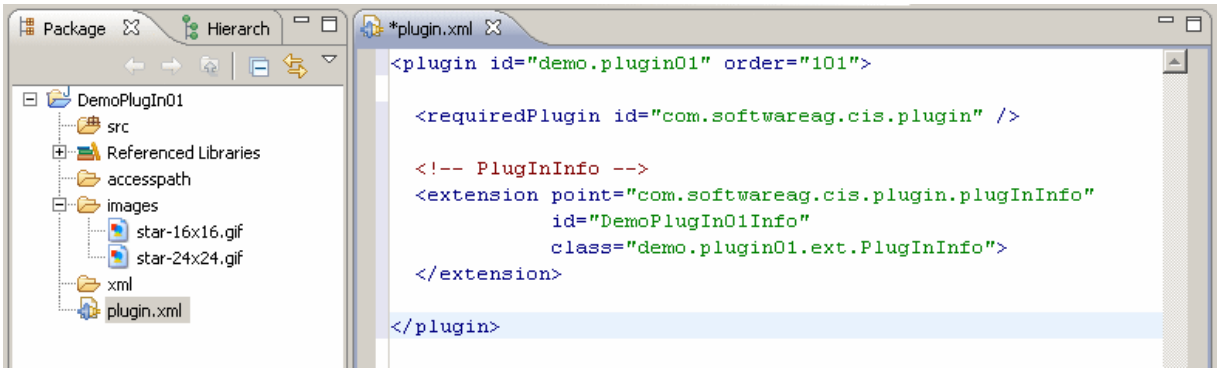
1. In the context menu of *DemoPlugIn01* in the package explorer, choose **New > File**.
2. Type *plugin.xml* as the file name and click **Finish**.
3. Enter the following XML code:

```
<plugin id="demo.plugin01" order="101">

  <requiredPlugin id="com.softwareag.cis.plugin" />

  <!-- PlugInInfo -->
  <extension point="com.softwareag.cis.plugin.plugInInfo"
             id="DemoPlugIn01Info"
             class="demo.plugin01.ext.PlugInInfo">
  </extension>

</plugin>
```



4. Save the file using <Ctrl>+S.

First of all a plug-in must have an identifier (here "demo.plugin01") which has to be unique among all plug-ins. We recommend you to use naming conventions similar to Java package names.

The order number of a plug-in (here "101") gives CentraSite Control a priority for the sequence in which the plug-ins have to be loaded at startup. The higher the number, the later a plug-in is loaded.

We need to declare our plug-in as being dependent on the plug-in *com.softwareag.cis.plugin* because we use an extension point provided by this plug-in. This dependency is indicated through the `requiredPlugin` XML element.

For a list of all supported extension points, see the section ***Extension Points***.

The `extension` XML element in our file *DemoPlugIn01/plugin.xml* denotes that our plug-in extends the user interface at a point where information about a plug-in can be contributed. The string that looks like a Java package name is the name of the extension point (*com.softwareag.cis.plugin.plugInInfo*).

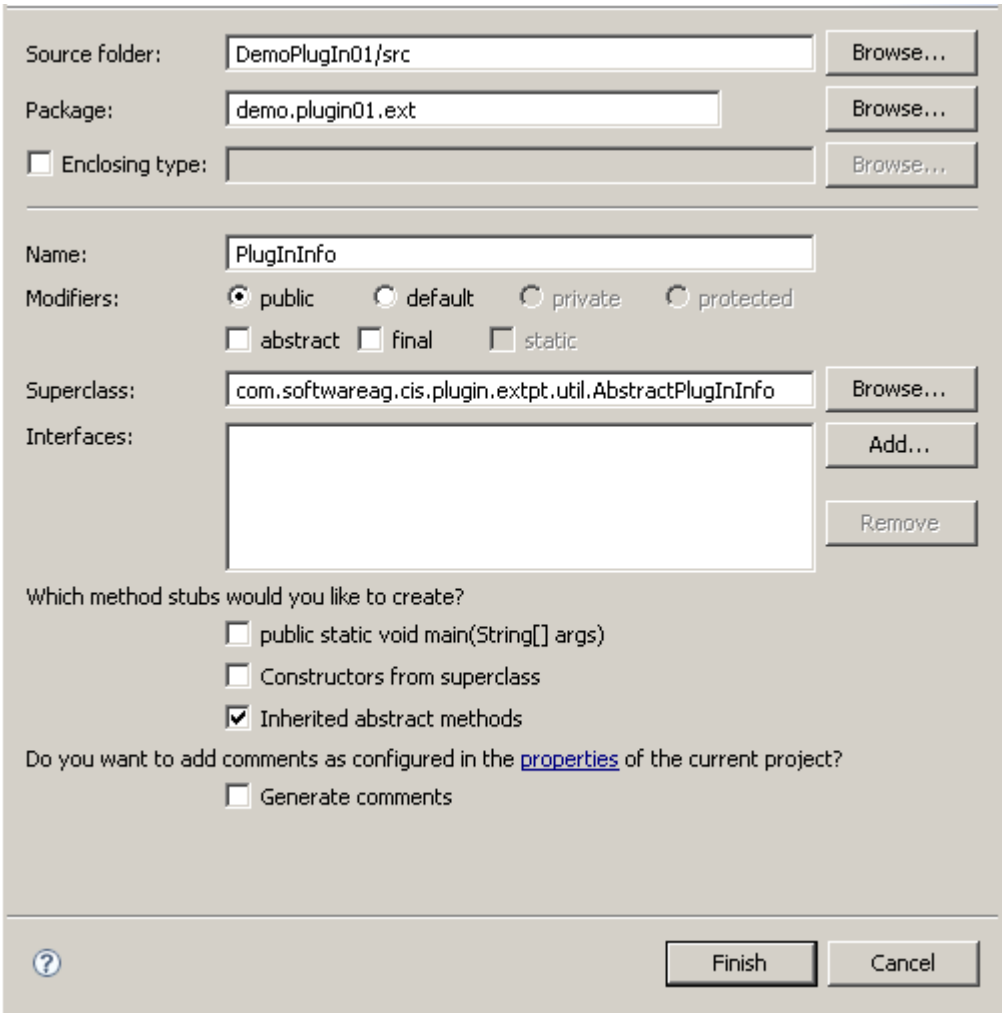
The extension identifier (here "DemoPlugIn01Info") must be unique among all extension identifiers of a plug-in.

The `class` attribute specifies the fully qualified name of the class that implements the extension (here *demo.plugin01.ext.PlugInInfo*). The top level package name for all of our Java code will be *demo.plugin01*. We choose *ext* as the subpackage name for the implementing class to denote that code that extends CentraSite Control resides here.

Now we have to implement the extension, i.e. we have to provide a Java class called *demo.plugin01.ext.PlugInInfo* which implements a specific interface required by the extension point.

1. In the context menu of *DemoPlugIn01/src* in the package explorer, choose **New > Class**.
2. Specify *demo.plugin01.ext* for the package name, *PlugInInfo* for the class name and *com.softwareag.cis.plugin.extpt.util.AbstractPlugInInfo* for the superclass (you may uses the **Browse** button to save some typing).

3. Make sure that the check box labeled with **Inherited abstract methods** is checked and click **Finish**.



Eclipse now opens the file *PlugInInfo.java* in the Java editor.

Modify *PlugInInfo.java* in the Java editor as follows:

```
package demo.plugin01.ext;

import com.softwareag.cis.plugin.extpt.util.AbstractPlugInInfo;

public class PlugInInfo extends AbstractPlugInInfo {

    public String getImageURL() {
        return "../DemoPlugIn01/images/star-16x16.gif";
    }

    public String getLayout() {
```

```
        return null;
    }

    public String getTitle() {
        return "DemoPlugIn01";
    }

    public String getVendor() {
        return "Software AG";
    }

    public String getVersion() {
        return "0.0.0.1";
    }
}
```

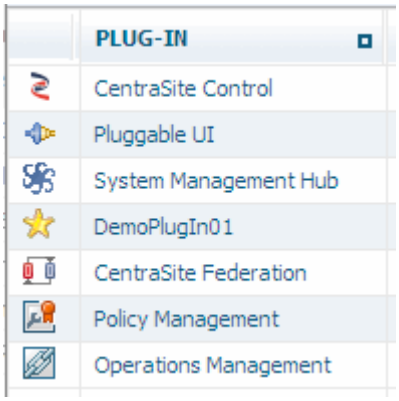
Save the modified file and make sure that no compile errors occur.

When you save the file, the Java file is automatically compiled into the folder *classes* of the project *DemoPlugIn01*. (Remember: the *classes* subfolder of our project is suppressed from displaying). The resulting class file is now accessible for the pluggable user interface of CentraSite Control.

Finally, let's check if CentraSite Control is aware of our minimalist plug-in:

1. Restart the Windows service "Software AG Runtime".
2. Start the CentraSite Control application from the Windows Start menu, using **Start > All Programs > Software AG > CentraSite > CentraSite Control**, and log in using your usual ID and password.
3. Click the **About** button at the top of the page. In the subsequent dialog, click **Plug-Ins**.

If everything works fine, you should see a dialog box whose contents look quite similar to the following screenshot. In particular, the line that represents our sample plug-in *DemoPlugIn01* should be visible.

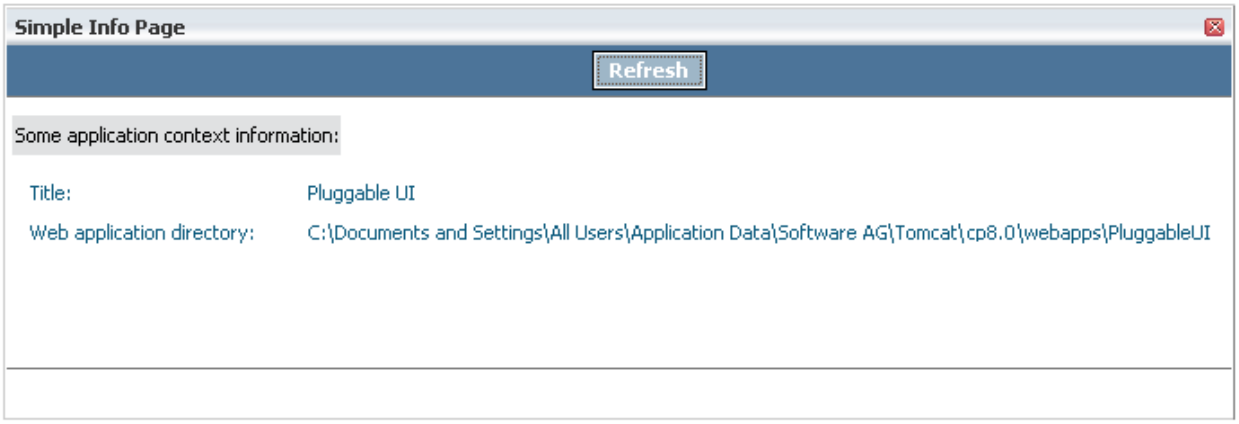


11 Appendix A4: Bring Your Own Layouts to the Screen

You can extend CentraSite Control by embedding your own layout pages. In this step you will learn how to create a layout page with by using the Application Designer IDE. Furthermore you will learn a very simple way to bring your own layout onto the screen.

Before we start, let's preview what the result of this step will be. We will create a simple layout page that presents some information to the user. This page can be requested by the user by clicking on an icon in the tool bar of CentraSite Control. Feel free to extend the layout page and enhance it with more information after you have worked through this step. In subsequent steps we will use this page many times when we extend CentraSite Control at more and more extension points. So please make sure that this page and the code behind it is running properly.

Here is a screenshot of the final result of this step:



In order to create and design a layout page we need to have the right tool. The CentraSite distribution kit contains an IDE (integrated development environment) that you can use for this purpose. There is currently no shortcut created by the installation to start this IDE. Therefore we have to create one manually. The IDE is a web application whose clients run inside a web browser. The

URL (assuming installation defaults) to start the IDE on a machine where CentraSite is installed is *http://localhost:53307/PluggableUI/HTMLBasedGUI/workplace/ide.html*.

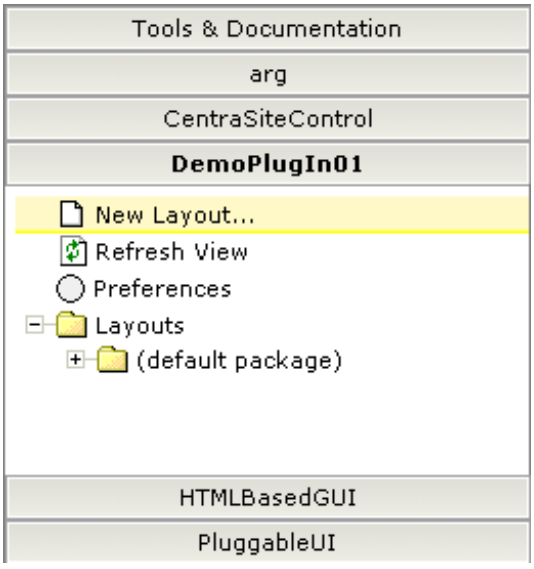
The IDE delivered in the distribution kit needs to be activated before you can use it. Refer to the section **Activating the IDE** for information on how to do this.

To access the documentation for the IDE, open the Software AG documentation website at **http://documentation.softwareag.com/**, click **webMethods** to display the list of documentation components for the webMethods suite, then click **Application Designer**.

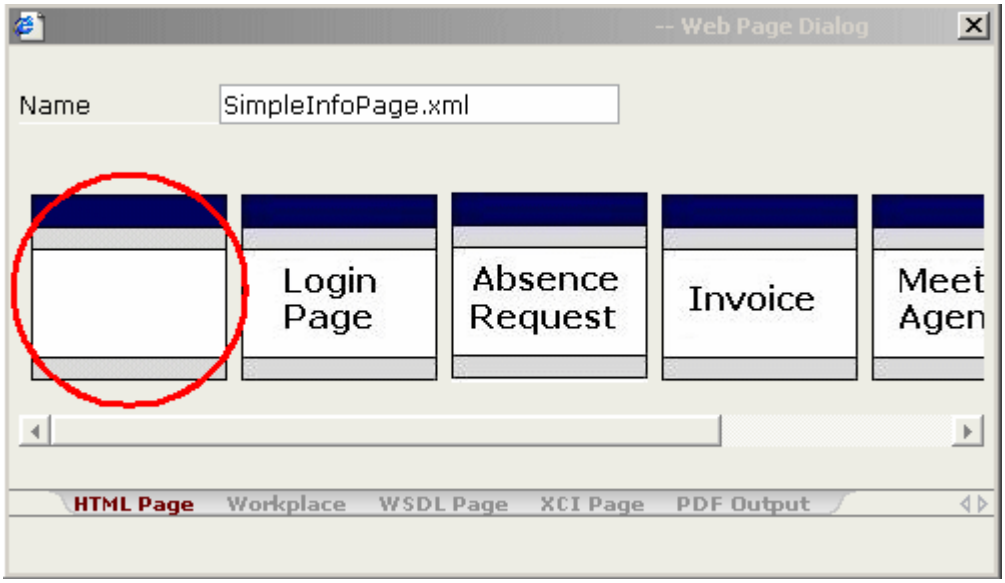
We start with the creation and design of a layout page. So please start the IDE now (leave your Eclipse instance running).

After the IDE has started, perform the following steps:

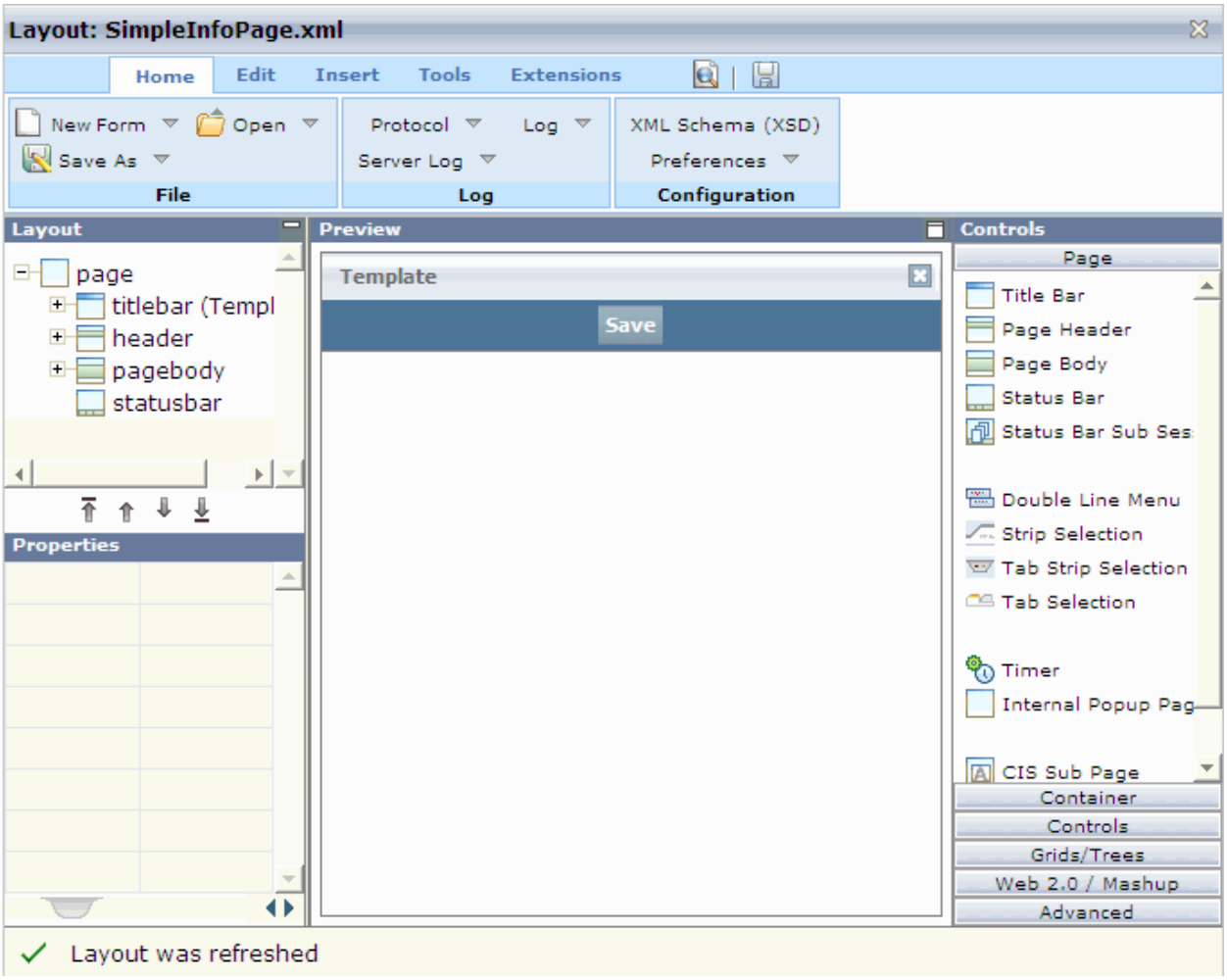
1. In the button list on the left, click the button labeled with the name of our plug-in **DemoPlugIn01**.



2. Click **New Layout** on the left side directly below the **DemoPlugIn01** button.
3. In the resulting dialog window enter "SimpleInfoPage.xml" in the input field labeled **Name**.
4. Click the leftmost image below the input field (see screenshot) to create an HTML page.

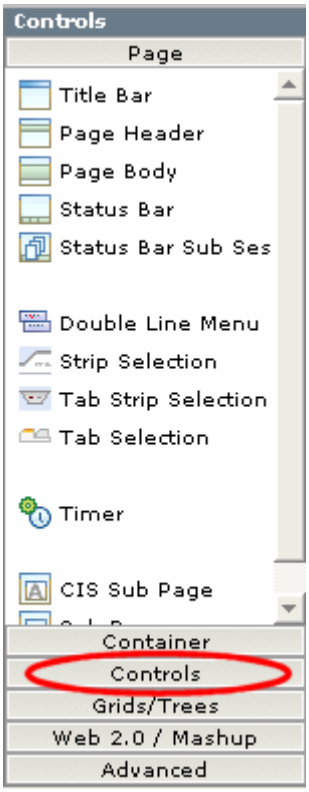


The IDE presents a standard HTML page in the preview area of the layout painter (in the center of the right side). To get an idea about how our newly created layout looks initially, we should request a preview of it from the layout painter. To do so, select the **Preview** icon from the toolbar of the layout painter (located beside the diskette symbol). The current look of layout *SimpleInfoPage.xml* is presented in the preview area.

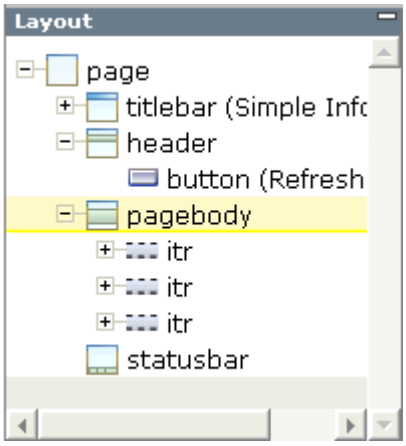


Starting from this layout, we will follow the steps below to create a layout that looks like the one that is shown at the beginning of this step:

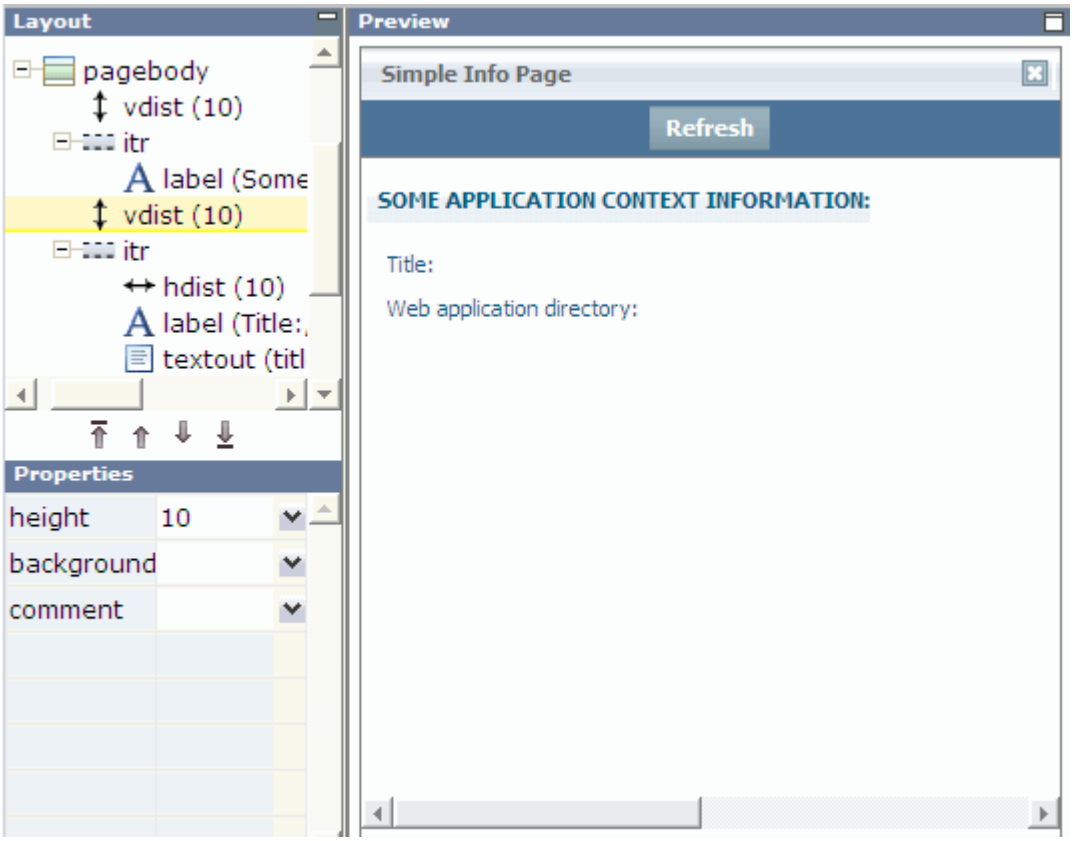
1. Click on the title bar of our layout (the bar above the **Save** button where the word **Template** is visible).
2. In the **Properties** view for the title bar, located at the lower left corner of the layout painter, change the `name` property from "Template" to "Simple Info Page".
3. Click the **Save** button of our layout (the content of the **Properties** view changes and the current properties for the selected button become visible). Change the `name` property from "Save" to "Refresh" and set the `method` property to "onRefresh" by just typing it in.
4. Save the layout by clicking on the diskette symbol in the tool bar of the layout painter. The preview of our layout changes and now reflects the properties we changed.
5. Click the **Controls** button of the button list in the **Controls** view (located right to the preview area).



6. Add three **Independent Row** controls to the body of our page:
   a. Click **Independent Row** and hold the left mouse button down.
   b. Drag the **Independent Row** icon to the page body of the layout (the white area below the button that is now labeled **Refresh**) and release the left mouse button.
   c. Click **Add as Subnode** from the popup menu that appears.
   d. Perform the same action to add a second and a third **Independent Row** control to the page body and click **Add as last Subnode** from the popup menu that appears after releasing the mouse button.
7. Please notice that the `pagebody` node of the **Layout** view (located above the **Properties** view) now contains three `itr` subnodes, representing the three **Independent Row** controls.



8. From the **Controls** view, drag and drop a **Label** control onto the first `itr` subnode of the `pagebody` node in the **Layout** view.
9. In the **Properties** view (which now presents the properties for the **Label** control we just added to the layout) set the `name` property to "Some application context information:".
10. Set the `asheadline` property of the label to "true". To access this property, you have to select the **Appearance** tab at the bottom of the **Property** view. You can select the value "true" using the combo box to the right of the property name.
11. Drag and drop a **Horizontal Distance** control onto the second `itr` subnode of the `pagebody` node in the **Layout** view.
12. In the **Properties** view, set the `width` property for the **Horizontal Distance** control to "10" by just typing it in.
13. Drag and drop a **Label** control onto the second `itr` subnode of the `pagebody` node in the **Layout** view. From the popup menu that appears after you release the mouse button, click **Add as last Subnode**.
14. Set the `name` property of the newly added label to "Title:" and the `width` property to "200".
15. Add a **Dynamic Text** control as the last subnode to the second `itr` subnode of `pagebody`. Set the `valueprop` property to "title" and the `width` property to "500".
16. Execute the last five steps again for the third `itr` subnode of the `pagebody` node in the **Layout** view. Set the `name` property of the **Label** control to "Web application directory:" and the `valueprop` property of the **Dynamic Text** control to "webAppDir". All `width` properties remain the same as for the children of the second `itr` subnode of `pagebody`.
17. Surround the first `itr` subnode with two vertical distances by dragging and dropping two **Vertical Distance** controls onto the first `itr` subnode of `pagebody` (one as a preceding node and one as a subsequent node of the `itr`). Set the `height` property for each **Vertical Distance** control to "10".
18. Save the layout by clicking again on the diskette symbol in the tool bar of the layout painter.



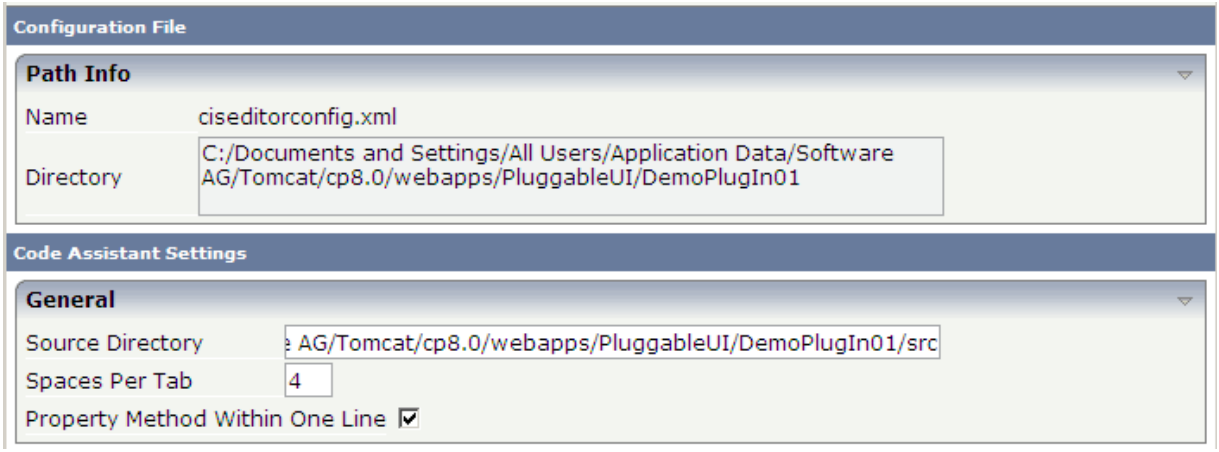
Now our layout looks like the one that is shown at the beginning of this step. But we are not finished yet. Each layout needs to have some code behind it (the so-called page adapter) which we did not provide yet. Among other things within a page adapter we can specify how to react on events that occur due to user interactions (the push of a button for example) or fill the controls with application specific values etc.

The code behind our layout at this stage is provided by a dummy adapter that comes with the IDE. But we need to provide our own, of course, so that the adapter knows what to do when a user presses the **Refresh** button, for example. So our next task is to create an adapter for our layout. Fortunately the IDE is equipped with tools that make life easy.

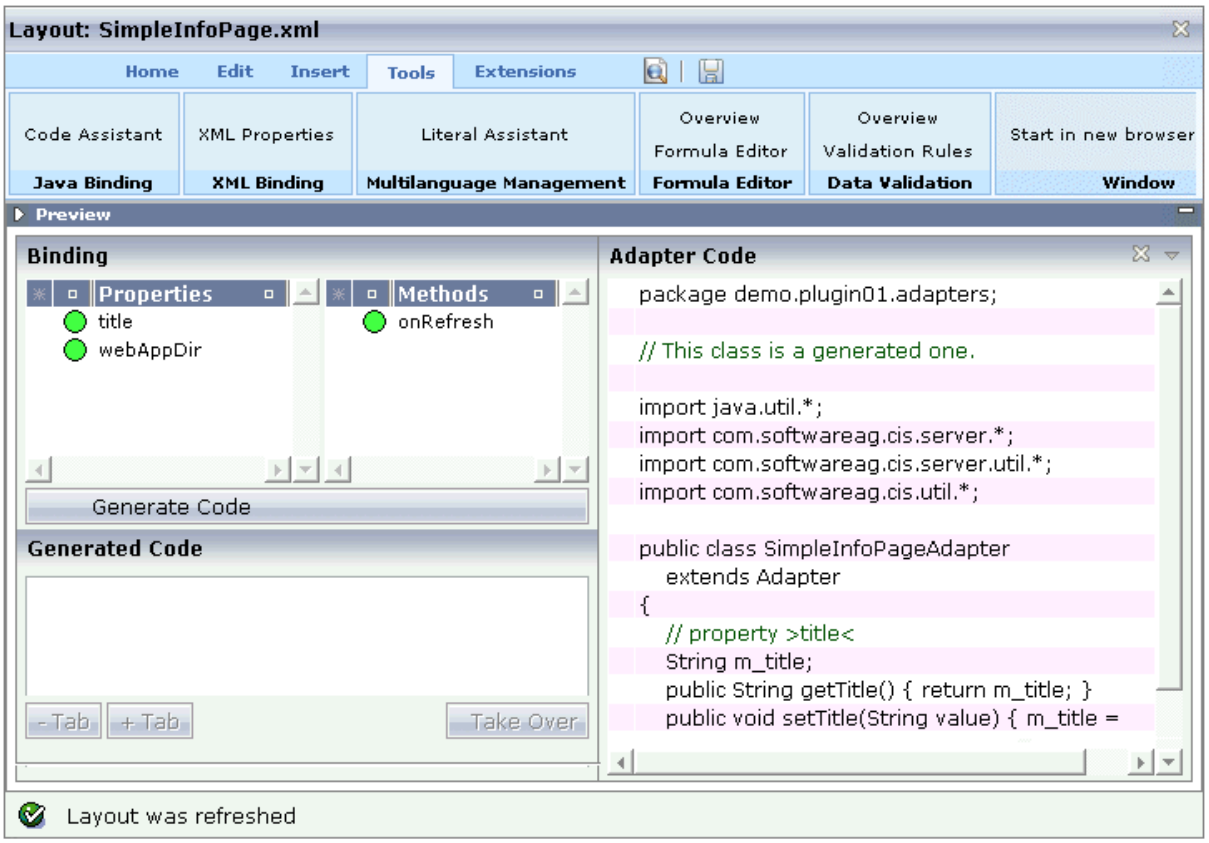
Follow the steps below to create an adapter for *SimpleInfoPage*:

1. If not already active, switch to the **Home** tab of the layout painter.
2. Select **Preferences** and type in the absolute path for the Java source directory of our CentraSite Control plug-in *DemoPlugIn01*.

Tip: Instead of typing in the complete path, you can copy/paste the content of the field labeled **Directory** into the input field for the source directory. Then append "/src" to the copied content (compare with following screenshot).



3. Click the **Save and Apply** button at the top of the dialog.
4. In the **Layout** view select the topmost tree node called **page** (you probably need to scroll up).
5. Change the `model` property of **page** from "DummyAdapter" to "demo.plugin01.adapters.SimpleInfoPageAdapter".
6. Save the layout (using the diskette symbol). The content of the **Preview** view changes and indicates an error now. This is normal and can be ignored at the moment.
7. Switch to the **Tools** tab of the IDE and select **Code Assistant**. The look of the IDE changes and the generated code for our page adapter is visible on the right side.

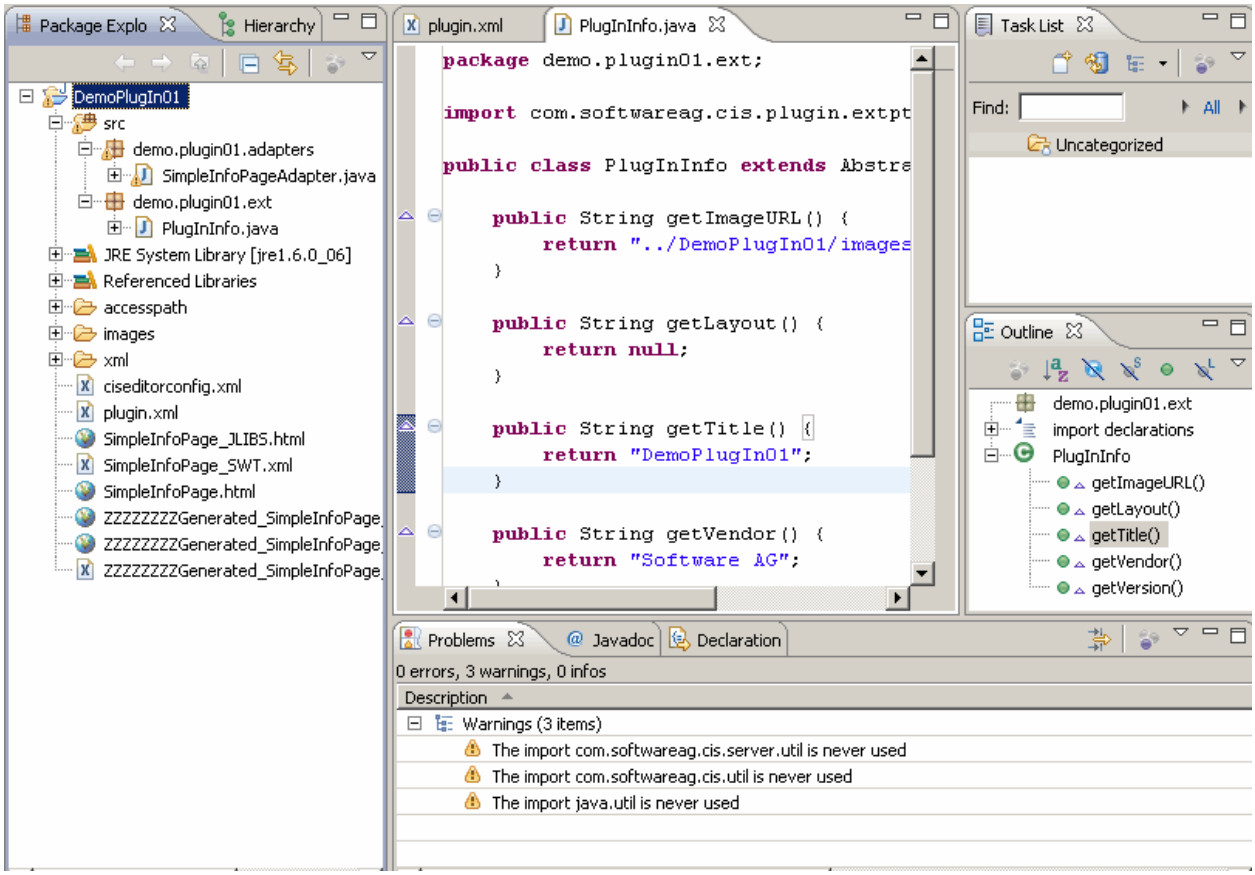


We could apply the necessary changes for the adapter class using the IDE. The more convenient way is doing this inside of our already existing eclipse project to which we will switch back soon. One more step inside the IDE is missing: the code is not yet stored in the file system. Hence, press the diskette icon again! Now the adapter source code is stored in the Java source directory of our CentraSite Control plug-in *DemoPlugIn01*.

You can close the IDE now.

Now return to your Eclipse environment. We need to refresh our plug-in project. To do so, select the folder *DemoPlugIn01* and choose **Refresh** from the context menu. After doing this and expanding all folders that relate to our plug-in, your eclipse project should look like the one below. Please note the contents of subfolders *accesspath* and *xml* which were formerly empty. The contents have been created by our IDE activities. Most importantly, you should notice that there is now a new package called *demo.plugin01.adapters* containing the class *SimpleInfoPageAdapter*.

Note that adapter classes used for plug-ins to CentraSite Control should be derived from the class `BaseAdapter` rather than from the class `Adapter` as provided by Application Designer.

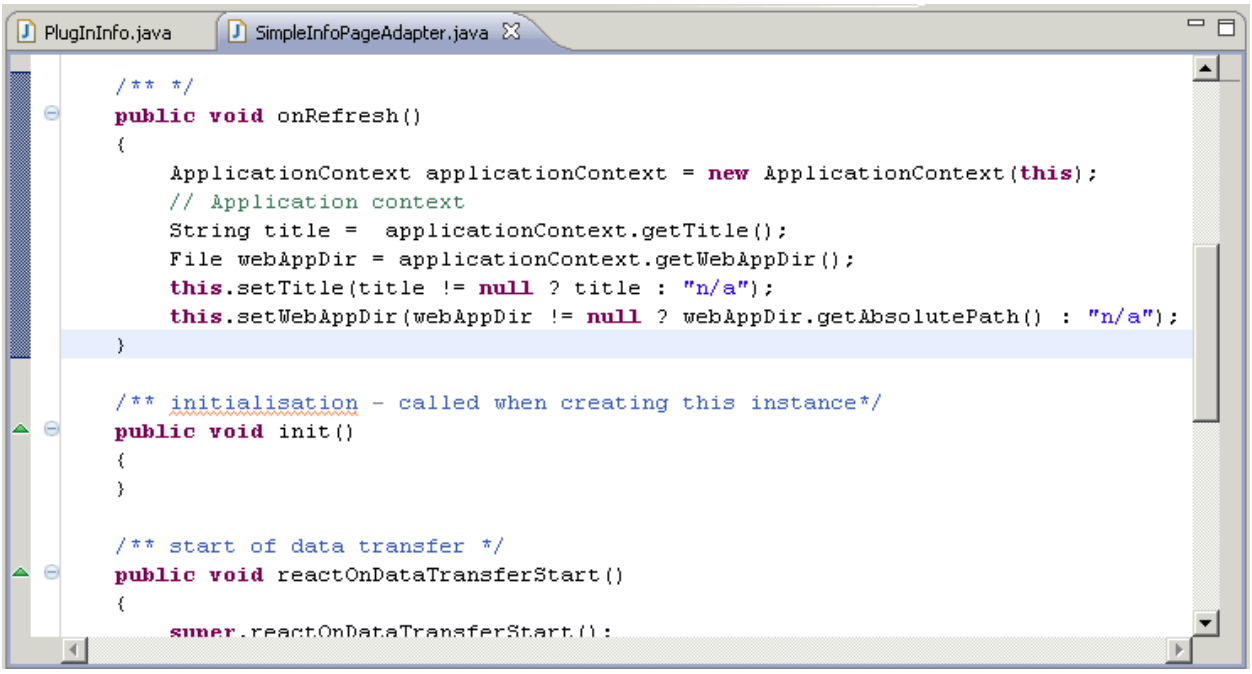


Let's apply application code to the adapter *SimpleInfoPageAdapter* now. To do so, open file *SimpleInfoPageAdapter.java* by double-clicking its node in the tree, and enter the following code inside the body of method `onRefresh`:

```
ApplicationContext applicationContext = new ApplicationContext(this);
// Application context
String title =  applicationContext.getTitle();
File webAppDir = applicationContext.getWebAppDir();
this.setTitle(title != null ? title : "n/a");
this.setWebAppDir(webAppDir != null ? webAppDir.getAbsolutePath() : "n/a");
```

Some types will be marked by the Java editor as unknown when you enter the code. So please press <Ctrl>+<Shift>+O to instruct the Java editor to add the necessary import statements automatically (for the missing class called *File* please choose *java.io.File* from the resulting dialog).

Now save the file. There should be no compilation errors.



Now the core of this step: we will bring our user-defined layout inside CentraSite Control to the screen. The questions here are when and how we do this. The first (when) is easy to answer: on user request. For the second (how) there are a lot of possibilities. Using an extension point defined by CentraSite Control suggests itself. But which one do we choose?

In this tutorial step we will extend CentraSite Control at another point in order to add a perspective. Perspectives are listed on the top of the workbench. Once again we have to inform the pluggable infrastructure that we are extending CentraSite Control at a new point. In order to do so, add the following XML element to the plug-in description file *plugin.xml* in your Eclipse environment and save it afterwards.

Add the following `requiredPlugin` XML element after the existing `requiredPlugin` XML element:

```
<requiredPlugin id="com.centrasite.control" />
```

Also add the following XML element to the already existing XML code:

```
<!-- Perspective -->
  <extension point="com.softwareag.cis.plugin.perspective"
             id="DemoPlugIn01Perspective"
             class="demo.plugin01.ext.PlugInPerspective" >
  </extension>
```

Save the file *plugin.xml*.

The implementation of our new perspective requires a new class which implements the necessary interface:

1. In the context of DemoPlugIn01/src, create a new Java class called `PlugInPerspective` in package *demo.plugin01.ext*. Use `class com.softwareag.cis.plugin.extpt.util.AbstractPerspective` as the superclass.
2. Let method `getTitle()` return the string "DemoPlugIn01".
3. Let the `getLogoImageURL()` method return the path to our 24x24 icon (*../DemoPlugIn01/images/star-24x24.gif*).
4. Insert the methods

```
public boolean hasTopicTree()
{
  return false;
}
```

and

```
public boolean supportsViews()
{
  return false;
}
```

The Java source should look exactly like this then:

```
package demo.plugin01.ext;

import java.util.List;

import com.softwareag.cis.plugin.extpt.util.AbstractPerspective;
import com.softwareag.cis.plugin.extpt.util.WorkplaceContext;
import com.softwareag.cis.server.util.ICONLISTInfo;

public class PlugInPerspective extends AbstractPerspective
{

  public String getTitle()
  {
    return "DemoPlugIn01";
  }

  public String getImageURL()
  {
    return null;
  }

  public boolean hasTopicTree()
  {
    return false;
  }
```

```
  public boolean supportsViews()
  {
    return false;
  }

  public String getLogoImageURL()
  {
    return "../DemoPlugIn01/images/star-24x24.gif";
  }

  public ICONLISTInfo getToolbar()
  {
    return null;
  }

  public String getView()
  {
    return null;
  }

  public String getViewLabel()
  {
    return null;
  }

  public List getViewValues()
  {
    return null;
  }

  public String getWorkplaceDefaultLayout()
  {
    return null;
  }

  public void setView(String arg0)
  {
  }

  public void setWorkplaceContext(WorkplaceContext arg0)
  {
  }

}
```

5. Save and close the Java source file.

We will now extend CentraSite Control with a new topic. In order to do so, add the following XML element to the plug-in description file *plugin.xml* in your Eclipse environment and save it afterwards.

```
<!-- Topic -->
  <extension point="com.softwareag.cis.plugin.topic"
             id="DemoPlugIn01Topic"
             perspective="demo.plugin01.DemoPlugIn01Perspective"
             class="demo.plugin01.ext.PlugInTopic" >
  </extension>
```

The implementation of our new topic requires two new classes: the topic class itself which implements the necessary interface for the extension point and an adapter class for the topic. Let's start with the implementation of the adapter class:

1. In the context of DemoPlugIn01/src, create a new Java class called `PlugInTopicAdapter` in package *demo.plugin01.ext.adapters*. Use class `com.centrasite.control.adapters.TopicAdapter` as the superclass. Do not inherit abstract classes here.
2. Add a public default constructor to the class. The Java source should look exactly like this then:

```
package demo.plugin01.ext.adapters;

import com.centrasite.control.adapters.TopicAdapter;

public class PlugInTopicAdapter extends TopicAdapter
{
  public PlugInTopicAdapter()
  {
  }

}
```

3. Save and close the Java source file.

And now the extending class:

1. In the context of DemoPlugIn01/src, create a Java class called *PlugInTopic*, in the package *demo.plugin01.ext*, using the superclass `com.centrasite.control.ext.util.BaseTopic`. Check the box labeled **Inherited abstract methods**.
2. Add a public default constructor which invokes the `super(int)` constructor to the source:

```
public PlugInTopic ()
{
  super(0);
}
```

3. Let the method `getTopicAdapterClass()` return `PlugInTopicAdapter.class`.
4. Let method `getTitle()` return the string "DemoPlugIn01".
5. Add the following code to method initTree:

```
String title = "Simple Info Page";
String pageUrl = "../DemoPlugIn01/SimpleInfoPage.html";
String adapterClass = SimpleInfoPageAdapter.class.getName();
ActionContext actionContext = getTopicAdapter().getActionContext();
actionContext.showPage(pageUrl, title, adapterClass);
```

After this change of the source code you should press <Ctrl>+<Shift>+O to resolve compilation problems.

6. Save the Java source file and make sure that no compilation errors occur. After applying the changes described above, the Java source code for class `PlugInTopic` should look like this:

```
package demo.plugin01.ext;

import com.centrasite.control.ActionContext;
import com.centrasite.control.Item;
import com.centrasite.control.ext.util.BaseTopic;

import demo.plugin01.adapters.SimpleInfoPageAdapter;
import demo.plugin01.ext.adapters.PlugInTopicAdapter;

public class PlugInTopic extends BaseTopic
{

  public PlugInTopic()
  {
    super(0);
  }

  protected Class getTopicAdapterClass()
  {
    return PlugInTopicAdapter.class;
  }

  protected void initTree() throws Exception
  {
    String title = "Simple Info Page";
    String pageUrl = "../DemoPlugIn01/SimpleInfoPage.html";
    String adapterClass = SimpleInfoPageAdapter.class.getName();
    ActionContext actionContext = getTopicAdapter().getActionContext();
    actionContext.showPage(pageUrl, title, adapterClass);
  }

  public void refresh(Item arg0, int arg1)
  {
  }

  public String getTitle()
  {
    return "DemoPlugIn01";
  }
```

```
  public String getImageURL()
  {
        return null;
  }

}
```

To see how our new extension affects CentraSite Control, restart the Software AG Runtime service and open CentraSite Control afterwards. The navigation pane shows the new perspective **DemoPlugIn01** which has 1 topic entry called **DemoPlugIn01**. Note also that the *star-24x24.gif* graphic is visible in the header bar.

When you click **Refresh** in the **Simple Info Page** display, the values for **Title** and **Web application directory** are updated: