Escenic Content Studio

Plug-in Guide

5.4.7.169266


vizrt

Disclaimer

Vizrt provides this publication “as is” without warranty of any kind, either expressed or implied.

This publication may contain technical inaccuracies or typographical errors. While every precaution has been taken in the preparation of this document to ensure that it contains accurate and up-to-date information, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained in this document.

Vizrt’s policy is one of continual development, so the content of this document is periodically subject to be modified without notice. These changes will be incorporated in new editions of the publication. Vizrt may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

Vizrt may have patents or pending patent applications covering subject matters in this document. The furnishing of this document does not give you any license to these patents.

Technical Support

For technical support and the latest news of upgrades, documentation, and related products, visit the Vizrt web site at www.vizrt.com.

Last Updated

29.06.2015


vizrt

Table of Contents


vizrt

1 Introduction

This manual describes the programming interfaces available for customizing and extending Escenic Content Studio. It describes three different extension techniques:

Creating Javascript extensions
This is the easiest way to extend Content Studio, and is the recommended method.

Creating Java plug-ins
This is a legacy method of extending Content Studio and requires Java programming skills.

Creating custom property editors
This is a legacy method of extending Content Studio and requires Java programming skills.

2 Javascript Extensions

Writing a Javascript extension is the newest and simplest method of extending Content Studio, and is also now the method we recommend. It requires far less specialized knowledge than the other methods, and any developer with some experience of Javascript programming for the web should be able to write extensions for Content Studio.

Development using the Javascript API is very straightforward. You can develop extensions locally on your own computer and test them in Content Studio without affecting other users in any way. Once you are satisfied with an extension, deployment is simply a matter of copying it to the correct location on the server. Content Studio also has a Javascript console: any code you enter in this console is executed immediately. You can use the console both as a quick means of trying out code and as a means of examining the system state for debugging purposes.

The Content Studio Javascript environment consists of:

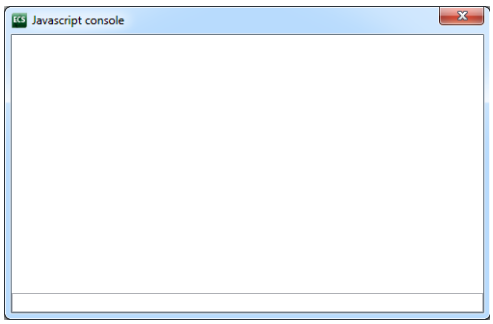
- The Javascript language itself
- A number of standard general-purpose Javascript libraries for purposes such as string handling, maths, XML manipulation, regular expressions and so on
- Content Studio-specific objects that provide access to the Content Studio object model

2.1 Hello World

To get an idea of how easy it can be to extend Content Studio using the Javascript API:

1. Start Content Studio in the usual way.
2. - On Windows, select **View** from the menu bar **while holding down the `Ctrl` key**. An extra **Debug** option should be displayed at the bottom of the **View** menu.
   - On Macs, right click between the Content Engine address field and the publication name field in the Content Studio footer:

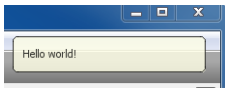
3. Select **Debug** > **Javascript console**. A dialog containing a Javascript console is now displayed:



4. Enter the following Javascript statement in the Javascript console:

```
require('notifier').notify('Hello world!');
```

and press **Enter**. A message bubble containing your "Hello world!" message should immediately appear near the top right corner of the Content Studio window, and disappear after a few seconds:



The **require** function used in this one-line "Hello World" program asks Content Studio to create an object - in this case a **notifier** object that can be used to display notification messages. Usually, of course, you would assign the object to a variable so you can use it more than once. Try entering the following three lines in the console:

```
no = require('notifier');
no.notify('Hello again world!');
no.notify('Goodbye world!');
```

2.2 Requiring Objects

You can use the **require** function to get various objects from Content Studio:

notifier
: You can use this object to display notification messages in Content Studio (see **section 2.1**).

actions
: You can use this object to create your own menu items and shortcuts, which you can then add to Content Studio menus.

content-studio
: This object represents the whole Content Studio application and you can use it to:

- Open and close content editors.
- Create and display your own side panels (which appear in the same area as the **Sections**, **Search** and **Clipboard** panels).

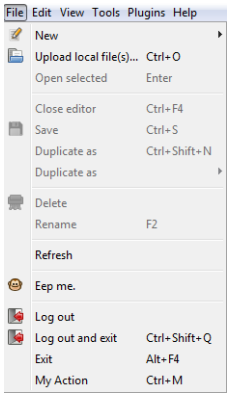
- Open browser windows.
- Listen for various events and do something when they occur.

Using the actions object

Try this:

```
actions = require('actions');
myaction = actions.createAction('My Action', 'shortcut M');
actions.addAction(myaction, 'file');
```

Now look at the **File** menu. Your action should appear at the bottom of the menu:



The **`Action`** object you have created also has methods - you can, for example, enable and disable the menu option using its **`enabled`** method:

```
myaction.enabled(false);
```

The menu entry should now be disabled. To re-enable it, enter:

```
myaction.enabled(true);
```

Using the content-studio object

You can get access to the **`content-studio`** object and use its methods in just the same way. This code, for example, opens your default browser and displays a web page:

```
cs = require('content-studio');
cs.browse('http://www.vizrt.com');
```

2.3 Handling Events

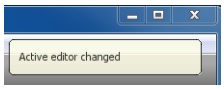
Content Studio, like most GUI-based applications is **event-based**: once it has been started, it listens for **events** such as user mouse clicks and responds to them. Your extensions need to work in the same way, and the objects in the Javascript API provide ways of both emitting and listening for events.

To listen for and respond to events, you use an object's **`on`** method. The **`content-studio`** object, for example, has an **`on`** method that you can use to listen for various events. Every time the user opens a new editor or switches

between editors by clicking on one of the editor tabs, an **editor-active** event is emitted. The **on** method lets you add a **listener** that associates a function with this event, so that every time an editor is activated, your function is executed. Try entering this code in the console:

```
no = require('notifier');
cs = require('content-studio');
cs.on('editor-active', function(editorcontrol) {no.notify('Active editor changed');});
```

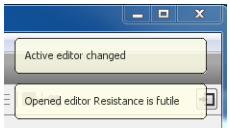
Now try opening a new editor or clicking on a editor tab to change the active editor. The notification message "Active editor changed" should appear in the top right corner of the window:



The **editor-active** event contains a parameter that the listener passes to the associated function. This parameter, **editorcontrol**, is an object representing the editor that has become active. You can access event parameters from inside listener functions. In this case you could use the **editorcontrol** object's **displayName** property to make a more informative message:

```
cs.on('editor-active', function(editorcontrol) {no.notify('Opened editor ' +
 editorcontrol.displayName);});
```

More than one function can be associated with the same event. You have now associated two functions with the **editor-active** event, so if you click on an editor tab they will both be executed and you will see two messages displayed:



You can clear all the functions associated with an event by calling the **removeAllListeners** method:

```
cs.removeAllListeners('editor-active');
```

No messages will be displayed if you click on an editor tab now.

The **content-studio** object can also emit other events:

- **editor-inactive**
- **panel-init**
- **panel-destroy**
- **panel-show**
- **panel-hide**
- **drop-listen**

For information about these events, see the **Content Studio Javascript API Reference** on documentation.vizrt.com.

Other objects that emit events (and therefore have **on** methods) include:

`Action` objects

An **`Action`** object emits an **`action`** event whenever the Content Studio user either clicks on its menu item or presses its keyboard shortcut. The **`action`** event has no parameters. You can therefore make a "Click me" menu item as follows:

```
no = require('notifier');
actions = require('actions');
myaction = actions.createAction('Click me', 'shortcut C');
myaction.on('action', function() {no.notify('I got clicked!');});
actions.addAction(myaction, 'file');
```

If you now click on **File** > **Click me**, you will see the notification message "I got clicked!".

`EventEmitter` objects

When you create a Content Studio side panel, an **`EventEmitter`** object is returned. This object emits an **`action`** event whenever the Content Studio user clicks inside the panel. For more information about this object, see the **Content Studio Javascript API Reference** on documentation.vizrt.com.

2.4 Making a Development Environment

The Javascript console is very useful for testing code and getting instant feedback, but for development purposes you need to be able to store your code somewhere and load your stored code into Content Studio. To set up a development environment on your computer:

1. Create a folder for your Content Studio extensions somewhere on your computer (for example **`C:\Users\`***your-name*`\cs-extensions`**).
2. Create one sub-folder for each extension you want to create (for example **`C:\Users\`***your-name*`\cs-extensions\hello-world`**).
3. In each extension sub-folder create a Javascript file (for example **`C:\Users\`***your-name*`\cs-extensions\hello-world\main.js`**) containing your extension code. For **`hello-world`** this could be:

   ```
   require('notifier').notify('Hello world!');
   ```

4. Add a file called **`package.json`** to each extension sub-folder. This must contain the following JSON data:

   ```
   {
     "main" : "js-file-name",
     "name": "extension-name",
     "version" : "extension-version"
   }
   ```

   where:

   - *js-file-name* is the name of your extension Javascript file (for example, **`main.js`**). Note that currently this file must contain **all** your extension code - it cannot load other **`.js`** files.
   - *extension-name* is the name of your extension (for example, **`hello-world`**). The name may contain alphanumeric characters and hyphens only.

- *version* is the version number of your extension. Currently this is not used for anything, but it must be supplied.

5. Open a command prompt window.
6. Start Content Studio by entering:

   ```
   javaws http://content-engine-host/studio/Studio.jnlp?com.escenic.script.root=dev-folder
   ```

   where:

   - *content-engine-host* is the name of your Content Engine host.
   - *dev-folder* is the absolute path of your extensions folder (**`C:\Users`**`\your-name`**`\cs-extensions`** in the example given above).

7. - On Windows, select **View** from the menu bar **while holding down the `Ctrl` key**. An extra **Debug** option should be displayed at the bottom of the **View** menu.
   - On Macs, right click between the Content Engine address field and the publication name field in the Content Studio footer:

8. Select **Debug** > **Developer Mode**. All the extensions in your extensions folder will now be loaded and executed. If you have included the hello-world extension described above then you should immediately see the "Hello World!" message appear in the top right corner of the Content Studio window.

2.5 Deploying Your Extensions

Deploying extensions is simply a matter of copying them to a fixed location on the Content Engine host. You can, however, use an **`autoinstall`** flag in the extensions' **`package.json`** files to determine whether or not they are automatically installed by Content Studio. This makes it possible to start out with a limited deployment for testers and/or advanced users before making an extension fully available. These stages are described below:

Limited deployment

Without making any changes to your **`package.json`** files, copy the contents of your local extensions folder to the **`/opt/escenic/studio/extensions`** folder on your Content Engine host.

The extensions are now available for use by Content Studio, but will not be automatically installed. Testers/advanced users can install specific extensions by starting Content Studio with the following command:

```
javaws http://content-engine-host/studio/Studio.jnlp?com.escenic.studio.extension.enable=extension-list
```

where:

- *content-engine-host* is the name of your Content Engine host.

- *extension-list* is a comma-separated list of the extensions to load.

Full deployment

When an extension is ready for full deployment, edit its **`package.json`** file and add an **`autoinstall`** flag. For our **`hello-world`** example the **`package.json`** file should look something like this:

```
{
  "main" : "main.js",
  "name": "hello-world",
  "version" : "1.0.0.0",
  "flags": ["autoinstall"]
}
```

Note that the **`flags`** property is an array, so the **`"autoinstall"`** value must be enclosed in braces.

Upload the edited file to the correct location. This extension will now be automatically installed when Content Studio is started.

Preventing auto-installation

You can prevent Content Studio from installing extensions even if the **`autoinstall`** flag is set by starting Content Studio with the following command:

```
javaws http://content-engine-host/studio/Studio.jnlp?com.escenic.studio.extension.disable=extension-list
```

where:

- *content-engine-host* is the name of your Content Engine host.
- *extension-list* is a comma-separated list of extensions that are not to be loaded. You can prevent installation of all extensions by specifying the special value **`*`**.

2.6 A Simple Extension

This simple extension adds bookmarking functionality to Content Studio. It adds two options, **Bookmark** and **Remove bookmark** to the **Tools** menu. Clicking on **Bookmark** adds the current editor to a list of bookmarks, clicking on **Remove bookmark** removes the current editor from the list of bookmarks. For each bookmarked editor, a menu item is added to the **Tools** menu. Clicking on a bookmark menu item opens the corresponding bookmarked editor.

Create a text file called **`main.js`** in **`C:\Users\`***your-name*`**\cs-extensions`** (for example) and open it for editing, then add the code described below.

A more complete version of this example can be downloaded from https://bitbucket.org/mogsie/studio-extension-tutorial-bookmarks. This mercurial repository takes the form of a tutorial. If you look at the "Commits" page (https://bitbucket.org/mogsie/studio-extension-tutorial-bookmarks/changesets) you will see that each revision is a "lesson" consisting of:

- The code so far
- The commit message, which contains a detailed explanation of the changes made in the step
- A **diff** showing exactly what was changed since the previous step

The lessons are listed in reverse order, so start at the bottom.

If you install mercurial then you can download the repository, read the commit messages and try to create your own "fork" of the extension.

The extension Javascript code

Use **`require`** to create the necessary Content Studio objects:

```
var notifier = require('notifier');
var actions = require('actions');
var CS = require('content-studio');
```

Create the **Bookmark** and **Remove bookmark** actions, and disable them - they will only be enabled when appropriate:

```
var bookmarkAction = actions.createAction('Bookmark', 'shortcut D');
var unbookmarkAction = actions.createAction('Remove bookmark', 'shortcut shift D');
bookmarkAction.enabled(false);
removeBookmarkAction.enabled(false);
```

Create the other variables you will need - an array to hold the bookmarks, two variables to keep track of the current editor's URI and title, and an array to keep track the number of bookmarks:

```
var currentURI = null;
var currentName = null;
var allBookmarks = {};
```

Add the actions you have created to the **Tools** menu:

```
actions.addAction(bookmarkAction, 'tools');
actions.addAction(removeBookmarkAction, 'tools');
```

The **Bookmark** and **Remove bookmark** actions should only be enabled if an editor is open. In addition, the **Bookmark** action should only be enabled if the current editor is **not** bookmarked, and the **Remove bookmark** action should only be enabled if the current editor **is** bookmarked. So you need add a call to a function that will perform these checks and initialize the menu items correctly:

```
enableDisableActions();
```

Then, of course, you need to actually define this **`enableDisableActions`** function. The following function first checks to see if an editor is actually open by checking the **`currentURI`** variable and disabling both actions if it is **`null`**. If not, it checks whether or not the current editor is in the **`allBookmarks`** array and enables/disables the actions accordingly.

```
function enableDisableActions() {
  if (currentURI == null) {
    bookmarkAction.enabled(false);
    removeBookmarkAction.enabled(false);
    return;
```

```
  }
  if (allBookmarks[currentURI]) {
    removeBookmarkAction.enabled(true);
    bookmarkAction.enabled(false);
  }
  else {
    removeBookmarkAction.enabled(false);
    bookmarkAction.enabled(true);
  }
}
```

Now you can add the actual bookmarking code that needs to be executed when the user clicks on one of your menu entries. Here is the code for the **Bookmark** action:

```
bookmarkAction.on('action', function() {
  if (currentURI == null) {
    return;
  }
  if (allBookmarks[currentURI]) {
    notifier.notify('Already bookmarked...');
    return;
  }
  var uri = currentURI;
  var bookmark = actions.createAction(currentName);
  actions.addAction(bookmark, 'tools');
  bookmark.on('action', function() {
    CS.openEditor(uri);
  });
  allBookmarks[uri] = bookmark;
  enableDisableActions();
});
```

What is happening here? First of all **`currentURI`** is checked: if there is no current editor to bookmark, then the function exits. Then if the current editor is already bookmarked, the function displays a message and exits. If these checks are passed, then the function:

- Creates a new action with the name of the current editor (**`currentName`**)
- Adds this action to the **Tools** menu
- Defines the event code for the new action
- Adds the editor's URI to the **`allBookmarks`** array
- Calls **`enableDisableActions`** to update the state of the **Bookmark** and **Remove Bookmark** menu items.

Here is the corresponding code for the **Remove Bookmark** action:

```
removeBookmarkAction.on('action', function() {
  if (currentURI == null) {
    return;
  }
  if (! allBookmarks[currentURI]) {
    return;
  }
  var bookmark = allBookmarks[currentURI];
  delete allBookmarks[currentURI];
  actions.removeAction(bookmark);
  enableDisableActions();
});
```

As you can see, it's more or less a mirror of the **Bookmark** action code.

Finally, you need to add some code to react to changes in the current editor. This code needs to set the **`currentURI`** and **`currentName`** variables used in the functions described above. Here is the missing event code:

```
function editorActive(editorControl) {
  console.log('Editor activated: ' + editorControl.uri);
  currentURI = editorControl.uri;
  currentName = editorControl.displayName;
  enableDisableActions();
}

function editorInactive(editorControl) {
  console.log('Editor deactivated: ' + editorControl.uri);
  currentURI = null;
  currentName = null;
  enableDisableActions();
}

CS.on('editor-active', editorActive);
CS.on('editor-inactive', editorInactive);
```

Save **`main.js`**.

The extension package file

In the same folder create a text file called **`package.json`**. Open it for editing and add the following code:

```
{
  "main" : "main.js",
  "name": "bookmarks",
  "version" : "1"
}
```

Save **`package.json`**.

Testing the extension

Open a command terminal and start Content Studio by entering the following command:

```
javaws http://content-engine-host/studio/Studio.jnlp?com.escenic.script.root=C:\Users\your-name\cs-extensions
```

Select **View** from the menu bar while holding down the **`Ctrl`** key, then select **Debug** > **Developer Mode**. Your **Tools** menu should now contain the **Bookmark** and **Remove bookmark** options you have defined. If you open a content item in an editor and click on the **Bookmark** option then a new option with the name of the content item should be added to the **Tools** menu. If you close the content item editor and click on this new option, then the content item should be reopened for editing.

3 Java Plug-ins

The extension method described in this chapter requires Java programming skills and knowledge of:

- The Swing GUI component kit
- The Escenic content model

The class **`StudioPlugin`** can be used to create standard plug-ins for Content Studio. You can use a plug-in based on **`StudioPlugin`** for a wide range of purposes. A **`StudioPlugin`** may be an "invisible" component that runs entirely in the background, but it may also add menu items to the Content Studio user interface and display its own user interface components (dialogs, messages and so on). The plug-in starts up and terminates together with Content Studio. During its life, it communicates with Content Studio via the **`StudioContext`** object.

Note that any **`StudioPlugin`**s you write must adhere strictly to Swing's **event dispatch thread (EDT)** conventions, as described here:http://java.sun.com/docs/books/tutorial/uiswing/concurrency/dispatch.html

Failure to do so may cause Content Studio to become unresponsive.

The main classes involved in writing a plug-in are:

`com.escenic.studio.plugins.StudioPlugin`
The plug-in class itself. You create a plug-in by extending this class.

`com.escenic.studio.plugins.spi.StudioPluginSpi`
The SPI class that creates **`StudioPlugin`** instances. You extend this class to create an SPI class that creates instances of your plug-in.

`com.escenic.studio.StudioContext`
All plug-ins communicate with Content Studio via this class. **`StudioPlugin`** has a **`getContext()`** method that returns a **`StudioContext`** object. **`StudioContext`** has methods the plug-in can use to subscribe to various events and to gain access to various components of the Content Studio session.

- **`addLifeCycleListener()`**: This method adds a **`StudioLifeCycleListener`** to the **`StudioContext`** object.
- **`getContentEditorContainer()`**: This method returns a **`ContentEditorContainer`** object.
- **`getContentService()`**: This method returns a **`ContentService`** object.
- **`getUser()`**: This method returns a **`Content`** object containing information about the current user.

- **`execute()`**: This method can be used to execute code encapsulated in **`PluginTask`** objects.

`com.escenic.studio.StudioLifeCycleListener`
This class provides plug-ins with a means of listening for and reacting to important events in a session's life cycle such as **`launched`**, **`userSessionStarted`**, **`userSessionEnded`** and **`exiting`**.

`com.escenic.studio.editors.ContentEditorContainer`
This object is a container for all the editors opened in a Content Studio session. You can use this object to get access to all the editors in a session. It also has an **`addEditorListener()`** method that adds an **`EditorListener`** to the **`ContentEditorContainer`**. Note that this object is not available until Content Studio is completely initialized, so you should always access it via the **`userSessionStarted`** event (see the example code in **section 3.1**).

`com.escenic.studio.editors.EditorListener`
This class provides plug-ins with a means of listening for and reacting to editor events such as **`editorOpened`** and **`editorClosed`**.

`com.escenic.client.content.ContentService`
This object provides access to content items. It also has an **`addContentListener()`** method that adds a **`ContentListener`** to the **`ContentService`**.

`com.escenic.client.content.ContentListener`
This class provides plug-ins with a means of listening for and reacting to content item events such as **`contentCreated`**, **`contentCreated`** and **`contentDeleted`**.

`com.escenic.studio.plugins.PluginTask`
This class is a container for any time-consuming tasks that a plug-in performs. You must use this class to wrap any actions that are potentially time-consuming, such as disk reads or accessing Internet resources. You create a task by extending **`PluginTask`** and execute it using **`StudioContext.execute()`**. For further information, see **section 3.2**.

For detailed information about the classes listed, see the javadoc documentation in your ***`engine-distribution`*`/apidocs`** folder.

3.1 Writing a Basic Plug-in

Follow these steps to write a basic plug-in for Content Studio:

1. **Set up the classpath**

   In order to write the plug-in you need **`studio-api.jar`** and **`client-core.jar`** in your classpath. You will find these in either ***`engine-distribution`*`/contrib/lib`** or ***`engine-distribution`*`/lib`**.

2. **Create a subclass of `StudioPlugin`**

   Use the following example as a basis. You must implement an **`initialize()`** method that uses the **`StudioContext`** object to retrieve the information it needs from Content Studio.

```
public class CustomStudioPlugin extends StudioPlugin {

  public CustomStudioPlugin(final StudioPluginSpi pPluginSpi, final StudioContext pContext) {
    super(pPluginSpi, pContext);
  }

  @Override
  public void initialize() {
    // You need to put your initialization code here.
    getContext().addLifeCycleListener(new StudioLifeCycleListener.Stub() {
      @Override
      public void userSessionStarted(final SessionEvent pEvent) {
        getContext().getContentEditorContainer().addEditorListener(new EditorListener.Stub() {
          @Override
          public void editorOpened(final EditorEvent pEvent) {
            // This example starts a background task when the editor is opened.
            // But you can do whatever you want.
            // You can also use other listener methods like editorClosed() etc.
            ResourceHandle handle = pEvent.getEditor().getResourceHandle();
            getContext().execute(new IconFetcherTask(getContext().getContentService(),
 handle));
          }
        });
      }
    });
  }
}
```

   Note that **`getContentEditorContainer()`** is called inside the **`userSessionStarted()`** method implementation. This ensures that it does not get called until Content Studio is fully initialized and the **`ContentEditorContainer`** is available.

3. **Create a subclass of `StudioPluginSpi`**

   Your subclass must override the **`createStudioPlugin()`** method and return an instance of the **`StudioPlugin`** subclass you created in the previous step.

```
public class CustomStudioPluginSpi extends StudioPluginSpi {
  public CustomStudioPluginSpi() {
    super("Custom Studio Plugin", "1.0", "Escenic");
  }

  @Override
  public StudioPlugin createStudioPlugin(final String pId, final StudioContext pContext) {
    return new CustomStudioPlugin(this, pContext);
  }
}
```

3.2 Making a Plug-in Task

In order to ensure that your plug-in does not make the Content Studio user interface freeze up, you must never directly include any code that executes potentially time-consuming operations. You must always create **plug-in tasks** for such operations, in order to ensure that they are handled correctly.

You can create a plug-in task by extending the **`com.escenic.studio.plugins.PluginTask`** class. The following example

shows the outline of an **`IconFetcherClass`** that is intended to retrieve an icon from the web or a file system and do something with it. Any code involving such a retrieval operation should always be implemented as a plug-in task.

```
private class IconFetcherTask extends PluginTask<Icon> {

  @Override
  public String getTitle() {
    return "Icon Fetcher";
  }

  @Override
  public Icon doInBackground() throws Exception {
    // In this method you can do some long running operations.
    Icon icon = fetchIconFromSomePlace();
    return icon;
  }

  @Override
  public void succeeded(Icon pIcon) {
    // Do something with the icon here
  }
}
```

Once you have defined a plug-in task in this way, you can call it from your plug-in class using **`StudioContext.execute()`**:

```
getContext().execute(new IconFetcherTask());
```

For further information see the **`PluginTask`** Javadoc.

3.3 Adding User Interface Components

Currently, you can extend the Content Studio user interface using plug-ins in the following ways:

- You can add options to the Content Studio menus
- You can display your own dialogs

You cannot, however, currently use plug-ins to modify the main window of the application in any way. (It is possible to create custom property editors for Content Studio, as described in **chapter 4**, but this is not done using the **`StudioPlugin`** class.)

3.3.1 Adding Custom Menu Items

The **`StudioPlugin`** class has a **`getDeclaredActions()`** method that returns a **`DeclaredActions`** object. This object contains all the menu items displayed by Content Studio. You can add menu items of your own by calling its **`addAction()`** method. **`addAction()`** has two signatures:

`addAction(`*`action`*`)`
: This form of **`addAction`** adds the menu item defined by the *action* parameter to Content Studio's **Plug-in** menu.

`addAction(`*`placement`*`,`*`action`*`)`
: This form of **`addAction`** adds the menu item defined by the *action* parameter to the menu specified with the *placement* parameter.

The following example shows how you can add a custom menu item to Content Studio's **View** menu:

```
private void createActions() {
  Action action = new MyCustomAction(name, icon);
  getDeclaredActions().addAction(DeclaredActions.Placement.VIEW_MENU, action);
}
```

This method can be called from inside your plug-in's **`initialize()`** method. An **`Action`** class contains both code to execute some action and the information needed to display a menu item. You can create your own **`Action`** classes by extending **`AbstractAction`** (see **section 3.3.2** for an example).

3.3.2 Displaying Custom Dialogs

You can display your own user interface components using the **`JOptionPane`** class. The following example shows the definition of a custom **`Action`** class that displays an information message:

```
private class MyCustomAction extends AbstractAction {
    public MyCustomAction(final String name, final Icon icon) {
      // give a name and icon to your action
    }

    @Override
    public void actionPerformed(final ActionEvent e) {
      JOptionPane.showMessageDialog(null,
          "Message from studio plugin",
          "Plugin dialog",
          JOptionPane.INFORMATION_MESSAGE
      );
    }
}
```

You can, of course display much more sophisticated user interface components using **`JOptionPane`**. For further information about JOptionPane and about Swing in general, see:

http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html
http://java.sun.com/docs/books/tutorial/uiswing/components/dialog.html

3.4 Packaging and Deploying a Plug-in

To make your plug-in available to Content Studio users, you must package it correctly in a JAR file, deploy it to the correct folder in your Content Engine installation, reassemble the Content Engine and redeploy the **`studio`** application that serves Content Studio to users.

You must package your plug-in in accordance with the standard for Java service provider interfaces. To do this you must create a file called **`com.escenic.studio.plugins.spi.StudioPluginSpi`** and add a line to the file that contains the fully-qualified name of your new SPI class. For example:

```
com.mycompany.contentstudio.plugin.CustomStudioPluginSpi
```

You must then make sure that this file is included in the **`META-INF/services`** folder of the JAR file you build, as specified in the standard for Java service provider interfaces. For further information about this, see:

http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider

Once you have created a correctly structured JAR file, you deploy it as follows:

1. Create a folder called **`plugins`** on the server in your *engine-home* folder (if it doe not already exist).
2. Create a subfolder for your plug-in (any name you choose) under *engine-home*`/plugins`.
3. Upload the JAR file to *engine-home*`/plugins/`*your-plugin*`/studio/lib`.
4. Use the assembly tool to assemble the Content Engine
5. Redeploy the **`studio`** application.

If you now start Content Studio, you should be able to see that your plug-in is running.

4 Property Editors

The extension method described in this chapter requires Java programming skills and knowledge of:

- The Swing GUI component kit
- The Escenic content model

When a Content Studio user opens a content item for editing, it is opened in a **content editor**: a tab displayed in the main editing area of the Content Studio window. Content editors are specialized for each content type: only the appropriate fields for each content type are displayed, and each field is displayed in a specialized control suited to the field's data type. A plain text field, for example, is displayed in a simple text field control, an HTML field is displayed in a rich text editing field, a boolean field is displayed as a check box, an enumeration field is displayed as a combo box and so on. These field editing controls are called **property editors**.

If the built-in property editors are not sufficient for your needs, you can implement more specialized property editors as plug-ins. You might, for example, want to build an editor composed of several inter-dependent controls to represent a complex field.

In order to make a custom property editor, you must be familiar with Swing. If you are familiar with Swing, making a property editor is quite straightforward. It involves the following tasks:

- Define the mark-up that will be used in the `content-type` resource to identify the fields for which the custom property editor is to be used.
- Add the defined markup to the required fields in the `content-type` resource.
- Implement a class to display the custom property editor. This class must implements the `PropertyEditorUI` interface.
- Implement a `PropertyEditorSpi` subclass that responds to property descriptors containing the markup values you have defined by creating an instance of your `PropertyEditorUI` implementation.
- Create a JAR file containing your property editor components and a service loader definition file.

The first of the above tasks is straightforward Swing programming. For detailed information about `PropertyEditorUI`, see the API Javadoc.

4.1 Defining Custom Property Editor Mark-up

You indicate that a custom editor is required for a field by adding special elements and attributes you have defined yourself to the field definition in the `content-type` resource. You need to determine the following values:

Namespace URI
Define a URI that you will use to identify the special elements you add to the **`content-type`** resource. For example **`"http://my-company.com/2008/studio-plugin"`**.

Property editor name
Define a name for your property editor. This name will also be the name of the element you add to the **`content-type`** resource, so it must not contain any spaces or special characters other than '-' and '_'. For example **`"my-custom-property-editor"`**.

Enabled attribute name
The name of the attribute that is to be used to determine whether or not the editor is enabled. This will usually be **`"enabled"`**.

Enabled attribute value
The value that is to indicate that the editor is enabled. This will usually be **`"true"`**.

Parameter names
If you want to allow parameters to be set in the **`content-type`** resource, then you should decide on the names of the parameters. These will be XML element names so they must not contain any spaces or special characters other than '-' and '_'. You might, for example, decide that you need a **`"min"`** and a **`"max"`** parameter.

4.2 Adding Mark-up to the content-type Resource

Once you have decided on the mark-up to use, you can add it to the required field definitions in your **`content-type`** resource. For example:

```
<field name="my-custom-field" type="number">
 <my-custom-property-editor xmlns="http://my-company.com/2008/studio-plugin" enabled="true">
   <min>2</min>
   <max>300</max>
 </my-custom-property-editor>
</field>
```

4.3 Implement PropertyEditorUI

The class that actually displays the custom property editor must implement the **`com.escenic.studio.editors.PropertyEditorUI`** interface. The main body of this class will be standard Swing programming and is not discussed here. In addition to ensuring that the class implements **`PropertyEditorUI`** correctly, however, you also need to be aware of the following:

- The **`PropertyEditorSpi`** class that creates instances of this class passes two parameters to the constructor: an **`AbstractPropertyBinding`** and a **`ResourceRecorder`**.
- The **`AbstractPropertyBinding`** object has a **`getPropertyDescriptor()`** method that gives you access to the contents of the field definition in the **`content-type`** resource.

- The **PropertyDescriptor** you obtain in this way has **getModule()** and **getModules()** methods that you can use to access any parameters specified in the field definition.
- Your class should include a **dispose()** method that calls the **ResourceRecorder**'s **disposeAll()** method for all the **AbstractPropertyBinding**'s **BindingListener**s. For example:

```
public void dispose() {
  List<BindingListener> listeners = new
ArrayList<BindingListener>(mBinding.getBindingListeners());
  for (BindingListener listener : listeners) {
    mBinding.removeBindingListener(listener);
  }
  mResourceRecorder.disposeAll();
}
```

4.4 Implement PropertyEditorSpi

When you have a class that implements **PropertyEditorUI**, you must create a subclass of **com.escenic.studio.editors.spi.PropertyEditorSpi** that will create instances of it. This class's **supports()** method must check for the markup values you have defined for your custom property editor, and its **createPropertyEditor()** must return an instance of your custom property editor class. For example:

```
public class MyCustomPropertyEditorSpi extends PropertyEditorSpi {
  public static final URI PLUGIN_URI = URI.create("http://xmlns.escenic.com/2008/studio-plugin");
  public static final String EDITOR = "my-custom-property-editor";
  public static final String ENABLED_ATTRIBUTE = "enabled";
  public static final String ENABLED_ATTRIBUTE_VALUE = "true";

  public boolean supports(final PropertyDescriptor pPropertyDescriptor) {
    return Number.class.isAssignableFrom(pPropertyDescriptor.getType()) &&
        pPropertyDescriptor.getModule(PLUGIN_URI, EDITOR, ENABLED_ATTRIBUTE,
 ENABLED_ATTRIBUTE_VALUE) != null;
  }

  public PropertyEditorUI createPropertyEditor(final AbstractPropertyBinding pBinding, final
 ResourceRecorder pResourceRecorder) {
    return new MyCustomPropertyEditorUI(pBinding, pResourceRecorder);
  }
}
```

4.5 Package The Property Editor

Create a JAR file containing all the components of your custom property editor. The JAR file must also contain a service loader definition file. This file must be located in the JAR file's **META-INF/services/** folder, and must be called: **com.escenic.studio.editors.spi.PropertyEditorSpi**. It must contain the name of your **PropertyEditorSpi** subclass. For example:

```
com.mycompany.studio.plugin.MyCustomPropertyEditorSpi
```

You must then deploy the JAR file as described in **section 3.4** and restart Content Studio in order to test your property editor.

4.6 Example Code

The following sections contain code for an extremely simple custom property editor called **`rating-editor`**. This editor displays a number field and prevents the user from entering values greater than 10.

This property editor can be enabled for a field by entering the following mark-up in the **`content-type`** resource:

```
<field name="rating" type="number">
 <rating-editor xmlns="http://xmlns.escenic.com/2008/studio-plugin" enabled="true">
   <amount xmlns="http://xmlns.escenic.com/2008/studio-plugin">4</amount>
 </rating-editor>
</field>
```

4.6.1 RatingPropertyEditorUI Example

```
package com.escenic.studio.plugin;

import com.escenic.domain.PropertyDescriptor;
import com.escenic.module.Module;
import com.escenic.studio.binding.AbstractPropertyBinding;
import com.escenic.studio.editors.PropertyEditorUI;
import com.escenic.studio.editors.ResourceRecorder;
import com.escenic.swing.DesaturatedIcon;
import com.escenic.swing.IterableButtonGroup;
import com.escenic.swing.SwingHelper;
import com.escenic.swing.binding.BindingEvent;
import com.escenic.swing.binding.BindingListener;
import com.escenic.swing.binding.BindingState;

import com.jgoodies.forms.factories.Borders;
import org.apache.commons.lang.Validate;
import org.apache.commons.lang.math.NumberUtils;
import org.jdesktop.application.Application;
import org.jdesktop.application.ResourceMap;

import javax.swing.AbstractButton;
import javax.swing.Icon;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JToggleButton;

import java.awt.FlowLayout;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

public class RatingPropertyEditorUI implements PropertyEditorUI {

  private static final int DEFAULT_VALUE = 10;

  private final AbstractPropertyBinding mBinding;
  private final List<RatingModel> mRatingModels = new ArrayList<RatingModel>();
  private final IterableButtonGroup mButtonGroup = new IterableButtonGroup();
  private final Icon mUnselectedIcon;
  private final Icon mSelectedIcon;
  private final JPanel mRootPanel;
  private boolean mIgnoreChangeEvents;
  private final ResourceRecorder mResourceRecorder;

  public RatingPropertyEditorUI(final AbstractPropertyBinding pBinding,
                                ResourceRecorder pResourceRecorder) {
    Validate.notNull(pBinding, "Binding may not be null");
    Validate.notNull(pResourceRecorder, "Resource recorder may not be null");
    mResourceRecorder = pResourceRecorder;
    mRootPanel = new JPanel();
    mBinding = pBinding;
    ResourceMap resourceMap = Application.getInstance().getContext().getResourceMap(getClass());
```

```
    mRootPanel.setName(getClass().getName());
    mSelectedIcon = resourceMap.getIcon("selected-icon");
    mUnselectedIcon = new DesaturatedIcon(mSelectedIcon);
    SwingHelper.initialize(this);
  }

  public PropertyDescriptor getPropertyDescriptor() {
    return mBinding.getPropertyDescriptor();
  }

  public AbstractPropertyBinding getBinding() {
    return mBinding;
  }

  public void dispose() {
    List<BindingListener> listeners = new ArrayList<BindingListener>(mBinding.getBindingListeners());
    for (BindingListener listener : listeners) {
      mBinding.removeBindingListener(listener);
    }
    mResourceRecorder.disposeAll();
  }

  public void initModels() {
    Module ratingEditor = getPropertyDescriptor().getModule(RatingPropertyEditorSpi.PLUGIN_URI,
                                                             RatingPropertyEditorSpi.RATING_EDITOR,

RatingPropertyEditorSpi.ENABLED_ATTRIBUTE,

RatingPropertyEditorSpi.ENABLED_ATTRIBUTE_VALUE);
    Module amountModule = ratingEditor.getModule(RatingPropertyEditorSpi.PLUGIN_URI, "amount");
    int amount = NumberUtils.toInt(amountModule.getContent(), DEFAULT_VALUE);
    if (amount > DEFAULT_VALUE) {
      amount = DEFAULT_VALUE;
    }
    for (int i = 1; i < amount + 1; i++) {
      mRatingModels.add(new RatingModel(i));
    }
  }

  public void initCommands() {
  }

  public void initComponents() {
    for (RatingModel ratingModel : mRatingModels) {
      JButton button = new JButton();
      button.setBorder(Borders.EMPTY_BORDER);
      button.setContentAreaFilled(false);
      button.setFocusable(false);
      button.setOpaque(false);
      button.setIcon(mUnselectedIcon);
      button.setDisabledIcon(mUnselectedIcon);
      button.setModel(ratingModel);
      mButtonGroup.add(button);
    }
  }

  public void initListeners() {
    mBinding.addPropertyChangeListener(
        "value",
        new PropertyChangeListener() {
          public void propertyChange(final PropertyChangeEvent pEvent) {
            setValueFromBinding();
          }
        }
    );
    mBinding.addPropertyChangeListener("readOnly", new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent pEvt) {
        setComponentState();
      }
    });

    mBinding.addBindingListener(new BindingListener<Object>() {
      public void bindingUpdated(BindingEvent<Object> pEvent) {
        setComponentState();
      }
    });
    mButtonGroup.addPropertyChangeListener(IterableButtonGroup.SELECTED_PROPERTY,
                                           new PropertyChangeListener() {
```

```
    public void propertyChange(PropertyChangeEvent pEvent) {
      if (!mIgnoreChangeEvents) {
        updateBindingValue((RatingModel) pEvent.getNewValue());
      }
    }
  });
}

public void initLayout() {
  mRootPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
  for (AbstractButton button : mButtonGroup) {
    mRootPanel.add(button);
  }
}

public void initState() {
  setValueFromBinding();
  setComponentState();
}

public JComponent getRootComponent() {
  return mRootPanel;
}

private void updateBindingValue(final RatingModel pValue) {
  mIgnoreChangeEvents = true;
  mButtonGroup.setSelected(pValue, true);
  mBinding.setValue(pValue == null ? null : new BigDecimal(pValue.getValue()));
  updateSelectedIcon(pValue);
  mIgnoreChangeEvents = false;
}

private void setValueFromBinding() {
  if (!mIgnoreChangeEvents) {
    BigDecimal value = (BigDecimal) mBinding.getValue();
    if (value != null) {
      for (RatingModel ratingModel : mRatingModels) {
        if (ratingModel.getValue() == value.intValueExact()) {
          mButtonGroup.setSelected(ratingModel, true);
          updateSelectedIcon(ratingModel);
          break;
        }
      }
    }
    else {
      updateSelectedIcon(null);
    }
  }
}

private void updateSelectedIcon(final RatingModel pRatingModel) {
  long value = pRatingModel == null ? -1 : pRatingModel.getValue();
  for (int i = 0; i < mButtonGroup.getButtonCount(); i++) {
    Icon icon = mUnselectedIcon;
    AbstractButton button = mButtonGroup.get(i);
    if (i <= value - 1) {
      icon = mSelectedIcon;
    }
    button.setIcon(icon);
  }
}

private void setComponentState() {
  if (mBinding.getState() == BindingState.UNBOUND) {
    for (AbstractButton button : mButtonGroup) {
      button.setEnabled(false);
      button.setSelected(false);
    }
  }
  for (AbstractButton button : mButtonGroup) {
    button.setEnabled(!mBinding.isReadOnly());
  }
}


private class RatingModel extends JToggleButton.ToggleButtonModel {
  private final int mValue;
```

```
    public RatingModel(final int pValue) {
      mValue = pValue;
      setRollover(false);
    }

    public int getValue() {
      return mValue;
    }
  }
}
```

4.6.2 RatingPropertyEditorSpi Example

```
package com.escenic.studio.plugin;

import com.escenic.domain.PropertyDescriptor;
import com.escenic.studio.binding.AbstractPropertyBinding;
import com.escenic.studio.editors.PropertyEditorUI;
import com.escenic.studio.editors.ResourceRecorder;
import com.escenic.studio.editors.spi.PropertyEditorSpi;

import java.net.URI;

public class RatingPropertyEditorSpi extends PropertyEditorSpi {
  public static final URI PLUGIN_URI = URI.create("http://xmlns.escenic.com/2008/studio-plugin");
  public static final String RATING_EDITOR = "rating-editor";
  public static final String ENABLED_ATTRIBUTE = "enabled";
  public static final String ENABLED_ATTRIBUTE_VALUE = "true";

  public boolean supports(final PropertyDescriptor pPropertyDescriptor) {
    return Number.class.isAssignableFrom(pPropertyDescriptor.getType()) &&
        pPropertyDescriptor.getModule(PLUGIN_URI,
                                      RATING_EDITOR,
                                      ENABLED_ATTRIBUTE,
                                      ENABLED_ATTRIBUTE_VALUE) != null;
  }

  public PropertyEditorUI createPropertyEditor(final AbstractPropertyBinding pBinding,
                                               final ResourceRecorder pResourceRecorder) {
    return new RatingPropertyEditorUI(pBinding, pResourceRecorder);
  }
}
```