Get Involved

java-net Project
Request a Project
Project Help Wanted Ads
Publicize your Project
Submit Content

Get Informed

About java.net
Articles
Weblogs
News
Events
Also in Java Today
java.net Online Books
java.net Archives

Get Connected

java.net Forums
Wiki and Javapedia
People, Partners, and Jobs
Java User Groups
RSS Feeds

Search

Web and Projects:
Online Books:
Advanced Search

Home | Changes | Index | Search | Go

Project Wonderland v0.5: Handling Content in Custom Cells

by Jordan Slott (jslott@dev.java.net)

Introduction

In the four-part tutorial series "Writing a New Wonderland Module" (start at Part 1) you learned how to extend the functionality of Project Wonderland by writing a custom Cell type and packaging it into a module. The module you created in this tutorial series simply displayed a shape, but did not otherwise display any form of user content. Many custom Cell types that you may wish to create involve displaying users' media content, for example, to display a picture, or a drawing, or a text file.

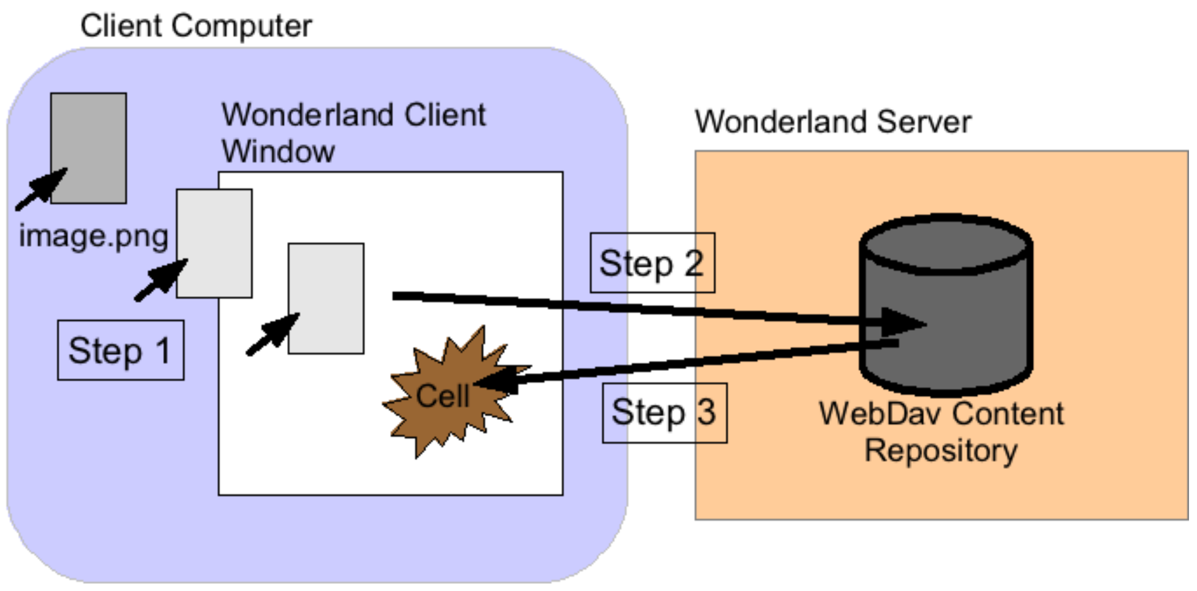
Developers of Cells can support handling content when dragged-and-dropped into the Project Wonderland client. For example, using **SVG Whiteboard** module (see Download, Build and Install Optional Project Wonderland Modules for Version 0.5 to learn how to download and install the add-on modules accompanying Project Wonderland), a user can drag-and-drop an SVG document from their computer's Desktop into the world: a new **SVG Whiteboard** is launched with that content.

This tutorial describes how your custom Cell type can handle imported user content, much like the **SVG Whiteboard** does.

Architectural Diagram

Before delving into code, let's review the process the user takes to import content into Wonderland and the general steps the system takes. Refer to the following Figure:

Figure 1. Three steps to import and display user content into Wonderland



- **Step 1:** The user imports a file from his/her computer Desktop by dragging and dropping the file into the Wonderland client. This kicks off the import process.
- **Step 2:** The system uploads the dropped file to the user's space in the WebDav-based content repository on the Wonderland server, so that all clients have access to it.
- **Step 3:** The system looks for a Cell type that supports the content type and creates a new Cell referring to the file uploaded to the content repository.

Changes to Implementing CellFactorySPI

When writing a new Cell type, it is relatively straightforward to support the scenario above. Most of the work is done automatically by the system, if you desire the default behavior. If you wish to override the default behavior, you may do so, as is described below (see "Technical Deep Dive"). Specifically, **Step 1** and **Step 2** are handled for you; you do not need to write any additional code to support your content type. You must write some code to handle **Step 3**, but all of these changes will occur in your implementation of the **CellFactorySPI** interface.

Here, we use a sample Cell type part of the Wonderland core software that displays either a PNG or JPEG image. You may find the source for this Cell under **modules/samples/image-viewer** directory. Please take a moment to browse its source code, although its operation is relatively simple: it uses the URI of an image from its server-state (see

ImageViewerCellServerState.java) and textures a jME Box primitive with it. You may wish to do something more complicated with the content in your custom Cell type.

We break **Step 3** above into two parts. Please refer to the **ImageViewerCellFactory.java** file in the **org.jdesktop.wonderland.modules.imageviewer.client.cell** package.

Step 3, Part A

After the system has imported the file and uploaded it to the WebDav-based content repository, it then looks through all registered Cell Factories for a Cell type that supports content with the given file extension. (Note: For now, only a single Cell type can be associated with a file extension; if more than one Cell type handles the same file extension, the system chooses the first Cell type it finds). It achieves this by querying the *getExtensions()* method for each Cell Factory. The *getExtensions()* method returns an array of (case-insensitive) String file extensions that the Cell type supports. Recall, in the four-part tutorial series, you returned an empty array for this method.

The *getExtensions()* method in **ImageViewerCellFactory.java** is implemented as follows:

```
    public String[] getExtensions() {
        return new String[] { "png", "jpg" };
    }
```

Step 3, Part B

Once the system identifies the Cell type to use to display the content, it creates a new Cell in the world, by first calling the **getDefaultCellServerState()** method on the **ImageViewerCellFactory** class. The system uses the **java.util.Properties** argument to communicate initial configuration information to the Cell. This class is merely a String key-value mapping; we define standard key-strings for common configuration attributes. In our case, the system passes the URI representing the content stored on the WebDav-based content repository using the **content-uri** attribute.

The implementation of the **getDefaultCellServerState()** for the **ImageViewerCellFactory** class is as follows. Note that the **java.util.Properties** argument may be null. We simply fetch the URI given in the property map and set it in the image-viewer's cell server state object. Please refer to the initial tutorial series on how the Cell server state object is used to initialize the state of your Cell.

```
    public <T extends CellServerState> T getDefaultCellServerState(Properties props) {
       ImageViewerCellServerState state = new ImageViewerCellServerState();

       // Look for the content-uri field and set if so
       if (props != null) {
           String uri = props.getProperty("content-uri");
           if (uri != null) {
               state.setImageURI(uri);
           }
       }
       return (T)state;
    }
```

Technical Deep Dive

While the default behavior of the drag-and-drop and content import system for Wonderland will suffice in most cases, this section covers the underlying technical details of the core APIs that expose this functionality. Developers are encouraged to experiment with these APIs if they so choose.

Drag-and-Drop Support

Drag-and-drop support for the virtual world is managed by the **DragAndDropManager** class in the **org.jdesktop.wonderland.client.jme.dnd** package. This class is a singleton, its instance accessed via the **getDragAndDropManager()** method. When an item is dropped into the Project Wonderland client window, the Java runtime environment generates a "drop" event that is handled by the **DragAndDropManager** class. The content type of each item is described by a collection of "data flavors" (see the **DataFlavor** class in the Java SE API).

Implementations of the Wonderland **DavaFlavorHandlerSPI** interface register themselves on the **DragAndDropManager** class; the **DragAndDropManager** class dispatches handling of the dropped item to the most appropriate data flavor handler. The **DataFlavorHandlerSPI** interface has three methods:

- **getDataFlavors():** Returns an array of DataFlavor objects the handler supports
- **accept():** Returns true if the handler really wants to accept a drop target, otherwise, the **DragAndDropManager** class looks for the next most appropriate handler
- **handleDrop():** Handles a dropped item

Examples of data flavors include "URL", "List of Files", and "InputStream". The core software defines handlers for the "URL" data flavor and "List of Files" data flavor. These two data flavor handlers should cover the vast majority of the needed handlers for drag and drop into the world.

- **FileListDataFlavorHandler:** This handler is typically used for files dragged from your local file system. This handler interacts with the **ContentImportManager** class (discussed below) to upload the file to the WebDav-based content repository and creates a new Cell to display the content.
- **URLDataFlavorHandler:** This handler is typically used for items and links dragged from a Web browser. Since the content is already available over the Internet, there is no need to first upload the content to the WebDav-based content repository,

so this handler simply creates a new Cell to display the content. Once more Cell types are written (e.g. to display HTML and other media), then you should be able to drag and drop a wide variety of content or links from a Web browser directly in world. (But you should be able to drag-and-drop images and links to images from Web browsers into the world: give it a try!)

Content Import Support

Content import support for the virtual world is managed by the **ContentImportManager** class in the **org.jdesktop.wonderland.client.content** package. This class is a singleton, its instance accessed via the **getContentImportManager()** method. This class maintains a collection of content import handlers, that implement the **ContentImporterSPI** interface, for a set of file extensions (e.g. jpg, png, etc). Users of the **ContentImportManager** class query it for a handler given a file extension using the **getContentImporter()** method and invoke the **importFile()** method on the returned object (that implements the **ContentImporterSPI** interface) to import the content. If a content import handler does not exist for a given file extension, then a "default" content handler is used (as set by the **setDefaultContentImporter()** method)

The **ContentImporterSPI** interface has the following two methods:

- **getExtensions():** Returns an array of String file extensions the handler supports
- **importFile():** Imports the file to an appropriate place so all clients may access it

Import handlers may import the content to anywhere they wish, however most often, it is most desired to import the content into the user's directory within the system's WebDav-based content repository. In most cases, you do not need to implement a **ContentImporterSPI** for your file extension; the default importer that comes with Wonderland will suffice. The **image-viewer** Cell type above depends upon both the drag-and-drop handlers and default content import handler defined by the system.

If you do wish to define your own **ContentImporterSPI** class, the **AbstractContentImporter** class (in package **org.jdesktop.wonderland.client.jme.content**) contains useful code; you may wish to implement the **ContentImporterSPI** interface by extending the **AbstractContentImporter** base class.

Content Browser Support

Although not explicitly related to **image-viewer** Cell type, Wonderland also lets you display a graphical browser of the WebDav-based content repository from your code. The **ContentBrowserManager** class maintains a single "default" content browser that can be displayed. This class is a singleton, its instance accessed via the **getContentBrowserManager()** method. To obtain the content browser, invoke the **getDefaultContentBrowser()** method; you are returned an implementation of the **ContentBrowserSPI** interface.

The **ContentBrowserSPI** interface has the following methods:

- **setActionName():** Sets the names of the two buttons on the GUI, roughly corresponding to an "OK" action and a "Cancel" action, as defined by the **BrowserAction** enumeration. By default, "OK" and "Cancel" are used as button names
- **addContentBrowserListener():** Registers a listener (the **ContentBrowserListener** interface) to receive notification when the "OK" or "Cancel" action has been selected. If an "OK" action is selected, the listener is given the URI of the content selected.
- **removeContentBrowserListener():** Removes a listener from the content browser
- **setVisible():** Sets the browser either to be visible or not visible
- **setModal():** Sets whether the browser is modal (not other GUI actions can happen while it is visible) or non-modal

The content browser can be used in several different situations. For example, a menu under the Tools menu lets you browse the content repository and create a Cell displaying the selected item. It may also be used on a Swing GUI to let you select an item from the content repository. You can even define your own GUI to act as the default content browser, although it is unlikely you need to do so; the GUI provided with the system should suffice.

Conclusion

In this tutorial, you learned how to interface your custom Cell type with content that is imported into Project Wonderland. This allows you to create advanced Cell types that display some form of content in the world.

Topic **ProjectWonderlandHandlingContent05** . { Edit | Ref-By | Printable | Diffs r1 | More }