IBM Cúram Social Program Management
Version 6.0.5

Cúram - Pod Developers Guide

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 35

Revised: March 2014

This edition applies to IBM Cúram Social Program Management v6.0.5 and to all subsequent releases unless otherwise indicated in new editions.

Contents

Figures

Tables

Developing pods

Use this information to develop Cúram pods. Pods are presented through a standard UIM Page. The UIM includes the PodContainer.vim that contains the predefined API for interacting with the pod. The PodContainer interface allows the client to interact with the server. A PodLoader is required for each pod.

Introduction

Purpose

This guide is a cookbook for Developers who want to create Pods. The guide will coach Developers through various scenarios beginning with the simplest implementation of a Pod, then adding content to Pods using tools provided and eventually introducing more advanced scenarios where the user will require knowledge of the widget development process.

Audience

This guide is aimed at Developers who want to create new Pods and new Pod Pages.

Prerequisites

Users of this guide will need basic Java™ , XML, HTML and CSS skills and a knowledge of the development environment. For the more advanced material the user will need to be familiar with the rendering framework which is covered in the Cúram Widget Development Guide.

Further Reading

Table 1. Further Reading

Guide	Description
Cúram Custom Widget Development Guide	A complete reference for developing custom widgets
Cúram Personal Page Configuration Guide	How to configure Personal Pages (Pod Pages)
Cúram User Experience Guidelines	Design guidelines for UI components including Pods. See Chapter 6: Home Pages and Pods

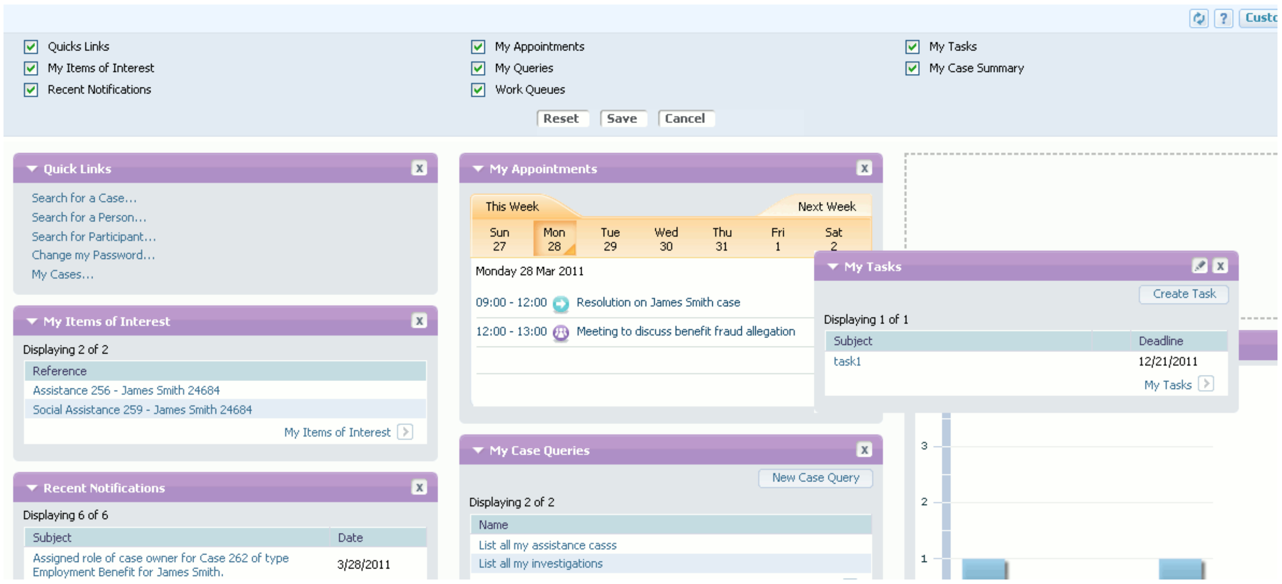
A Technical Overview

What is a Pod?

A Pod is a user interface widget that can be placed on a client page. In this respect it is no different to any other user interface widget that presents data such as a list or cluster. Where a Pod differs from other types of widgets is that it can be placed in a Pod-Container where a number of additional features are activated, such as the ability to be re-positioned in the container and the persistence of user settings such as whether the Pod is displayed and what filter settings are applied. A filter is an optional feature of a Pod that allows the content to be customized by the user, it can be accessed if available through the pen icon on the title bar of the Pod.

What is a Pod page?

A Pod page is a UIM page which contains a Pod-Container widget. The Pod-Container widget manages Pods. The widget is configured to present a selection of Pods that can be viewed in the container. The addition and removal of Pods from the container is managed through a *customization-console*. The Pod-Container widget manages the movement of Pods to different locations within the container. Where applicable it processes filters associated with Pods. In each case the last configuration of the Page is saved for the current user and retrieved the next time they load the page.



Displaying 2 of 2

Reference
Assistance 256 - James Smith 24684
Social Assistance 259 - James Smith 24684

My Items of Interest

Monday 28 Mar 2011

09:00 - 12:00 Resolution on James Smith case

12:00 - 13:00 Meeting to discuss benefit fraud allegation

Displaying 1 of 1

Subject	Deadline
task1	12/21/2011

My Tasks

Displaying 6 of 6

Subject	Date
Assigned role of case owner for Case 262 of type Employment Benefit for James Smith.	3/28/2011

Displaying 2 of 2

Name
List all my assistance casss
List all my investigations

Figure 1. Pod-Container

How does it work?

The next section provides an overview of the artefacts that work together to present a Pod page.

UIM Page

The Pods are presented through a standard UIM Page. The UIM must include the `PodContainer.vim` which contains the predefined API for interacting with the Pods infrastructure including the display of the page and saving of user preferences.

PodContainer

The PodContainer is the interface through which the client interacts with the server. At the display phase the server interface invokes the `loadData()` method on the `PodContainer` class. At action phase one of the save API's processes the data from the Pod-Container. The `PodContainer.vim` provides a reusable interface to the Pod infrastructure, simply add the `PodContainer.vim` to your UIM page and you will have a fully functioning interface.

PodLoader

A PodLoader must be written for each Pod. The PodLoader defines the Pod and its content. This book will mainly deal with the development of PodLoaders.

Database Tables

A number of tables are used to manage Pods.

Table 2. Database tables used to load Pods

Table	Description
PODTYPE	A list of all existing Pods
PODLOADERBINDINGS	A list of all existing PodLoaders mapped to a Pod type
PAGECONFIG	A list of configurations of Pod Pages
USERPAGECONFIG	A list of user customizations of Pod Pages

Loading the Page

At the display phase the server interface invokes the `loadData()` method on the `PodContainer` class. The `PodContainer` uses the `PodContainerManager` to identify all the Pods to be displayed on the page using the information in the PAGECONFIG and USERPAGECONFIG database tables. The PodContainerManager then identifies the PodLoader for each Pod to be displayed using the information in the PodType and PodLoaderBindings codetables. The PodContainer manager invokes the createPod() method on each PodLoader. The PodLoader supplies the data for a single Pod and the PodContainerManger builds up the cumulative data for all the Pods within the container.

Rendering the page

The Rendering of the page is handled by a collection of Renderers. The rendering begins with the PodContainerRenderer which recevies the document from the loading process and generates the PodContainer widget. It then delegates the rendering of Pods to a Pod renderer which in turn delegates to other renderers using markers in the data it receives. Each renderer returns its own content which it either generates itself or generates with the help of other renderers. This pattern of delegation is repeated unit all content is rendered.

The Curam Widget Development Guide discusses in detail the rendering framework and how renderers interact.

Saving the Page

At the action phase the server interface saves any changes the user made to the Pod selection and layout of the container back to the database again by way of the PodContainer API. The page is saved by any of the following actions, clicking the save button in the customization console, clicking the save button on a Pod filter, dragging and dropping a Pod (each time a Pod is dropped the save action is invoked to record the new layout of the page).

Configuring Pods

A Pod page can be configured through an Administration wizard which allows the layout and content of the Pod page to be defined. A full explanation of the Administration wizard is available in the Curam Personal Page Configuration guide.

Pod Dimensions: The dimensions of a Pod are not directly specified by a Pod. This allows Pods to dynamically resize to fit their environment and facilitates the re-use of Pods across Pod containers.

Pod Height
: The height of each Pod is determined by its content. A Pod's height will extend to display it's content.

Pod Width

The width of a Pod is determined by the container it is being displayed in. Each Pod container is configured with a number of equally sized columns. The Pod width will dynamically size to fill the width of the column it is placed in.

Tip: When deciding on a layout for your Pod page we recommend you consider the type of Pods you will be adding to the container and how they might be affected by resizing. Many of the predefined Pods are optimally sized for a 3 column layout. Using alternate layouts may distort the content of the Pods and visually this could detract from the page.

Product Pods

A collection of Pods are provided with the product. The Home section of each Application view is pre-configured with a set of Pods appropriate to that Application view (Pods can be shared across Pod pages). The configuration for each Application view can be updated by an administrator. See the Curam Personal Page Configuration guide for details of how to do this.

User configuration of Pod Pages

Each Pod page is pre-configured with a set of available Pods and a set of selected Pods which are visible in the container. An application user can further customize the workspace by...

- adding Pods from the available list using the customization-console
- removing Pods using the customization console
- removing Pods using the close button on the title bar of the Pod
- moving Pods by dragging to a new location in the container.
- filtering Pods by using the filter feature (where available).

Each time a user takes one of the actions listed above a record of the current configuration of the page is saved. When the page reloads this saved configuration will be re-displayed.

Developing new Pods

In addition to re-using the Pods provided in the product an Organization may want to create new Pods. The Pod framework provides the ability to create new Pods with custom content. This guide will present examples of how this can be done.

Getting Started

Introduction

Before creating a Pod we will need to create a Page to host it. The page that hosts our Pods needs a Pod container which manages the Pods allowing them to be added/removed/moved and updated.

Creating a page with a Pod container

Starting with a page that is mapped to a section and tab in the application (See WebClient reference guide for details of how to map pages in the application), add a Pod Container to the page by including the PodContainer.vim file as in the example below.

```
<PAGE PAGE_ID="MyPodContainer"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="file://Curam/UIMSchema.xsd"
>

  <CONNECT>
    <SOURCE NAME="CONSTANT" PROPERTY="MyPodContainer"/>
    <TARGET NAME="DISPLAY" PROPERTY="pageID$pageID"/>
  </CONNECT>

  <INCLUDE FILE_NAME="PodContainer.vim"/>

</PAGE>
```

Figure 2. MyPodContainer.uim

Identifying a Pod page

Add a `Constant.properties` file to the same folder as the UIM file. Add a property to the file that maps to the name of the constant used in the UIM to the *page-id* of the UIM page. When the server interface is called this value will be used to uniquely identify the Pod page.

```
MyPodContainer=MyPodContainer
```

Figure 3. Constant.properties

Configuring the database information about the page

The Pod page requires 2 database records to operate. The PAGECONFIG table stores information about which Pods are available on the page. The USERPAGECONFIG table stores the users customizations. Add the following DMX files to the component. Run the database build target to insert the records.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="PAGECONFIG">
  <column name="pageConfigID" type="id"/>
  <column name="userRoleName" type="text"/>
  <column name="pageID" type="text"/>
  <column name="config" type="text"/>
  <column name="versionNo" type="number"/>

  <row>
    <attribute name="pageConfigID">
      <value>9999</value>
    </attribute>
    <attribute name="userRoleName">
      <value></value>
    </attribute>
    <attribute name="pageID">
      <value>MyPodContainer</value>
    </attribute>
    <attribute name="config">
      <value>
       &lt;page-config&gt;
       &lt;contexts&gt;
       &lt;sequence domain="CURAM_CONTEXT"/&gt;
       &lt;/contexts&gt;
       &lt;availablePods&gt;
       &lt;sequence domain="POD_TYPE_SELECT"&gt;
       &lt;/sequence&gt;
       &lt;/availablePods&gt;
       &lt;layout&gt;
       &lt;sequence domain="COL_SIZE"&gt;
       &lt;value&gt;33&lt;/value&gt;
       &lt;value&gt;33&lt;/value&gt;
       &lt;value&gt;33&lt;/value&gt;
       &lt;/sequence&gt;
       &lt;/layout&gt;&lt;/page-config&gt;
      </value>
    </attribute>
    <attribute name="versionNo">
      <value>1</value>
    </attribute>
  </row>
</table>
```

Figure 4. PAGECONFIG.DMX

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="USERPAGECONFIG">
  <column name="userPageConfigID" type="id"/>
  <column name="userRoleName" type="text"/>
  <column name="userName" type="text"/>
  <column name="pageID" type="text"/>
  <column name="config" type="text"/>
  <column name="defaultInd" type="bool"/>
  <column name="versionNo" type="number"/>
  <row>
    <attribute name="userPageConfigID">
      <value>9999</value>
    </attribute>
    <attribute name="userRoleName">
      <value></value>
    </attribute>
    <attribute name="userName">
      <value/>
    </attribute>
    <attribute name="pageID">
      <value>MyPodContainer</value>
    </attribute>
    <attribute name="config">
      <value>
       &lt;user-page-config&gt;&lt;/user-page-config&gt;
      </value>
    </attribute>
    <attribute name="defaultInd">
      <value>1</value>
    </attribute>
    <attribute name="versionNo">
      <value>1</value>
    </attribute>
  </row>
</table>
```

Figure 5. USERPAGECONFIG.DMX

Testing the page

Build the application, launch it, login and go to the new Pod Page.

When our new Pod page loads it will be empty except for few buttons in the top right corner. The container is empty because we have not added any Pods to the page. Clicking the Customize button will open the customization-console. When the console opens it will be empty except for the action buttons. Again, because we have not assigned any Pods to the container there will be no Pods to select.

- The *Save* button will store the current users customizations.
- The *Reset* button will delete the current users customizations and revert to the default for this Page.
- The *Cancel* button resets the selection in the customization-console and closes it.

In the next chapter we will create a very simple Pod and add it to the container.

Hello World Pod

Introduction

In this chapter we are going to create a basic Pod with a title and some text. We will also use the Admin Wizard to add our new Pod to our Pod page.

There are 4 basic steps to get our Pod on a page...

1. Declaring a Pod
2. Declaring a PodLoader
3. Implementing a PodLoader
4. Adding the Pod to the Pod Container

Declaring a new Pod

The first step is to declare our new Pod. The `PodType` codetable is used for this purpose. Create a file CT_PodType.ctx in our component. Add a code and value for the new Pod like the example below. The convention is to use the prefix *PT* for the codetable value. The description field will be used by the Administration wizard to refer to our Pod.

```
<?xml version="1.0" encoding="UTF-8"?>
<codetables package="codetable">
  <codetable java_identifier="PODTYPE" name="PodType">
    <code
      default="false"
      java_identifier="HELLOWORLD"
      status="ENABLED"
      value="PT9001"
    >
      <locale language="en" sort_order="0">
        <description>Hello World!</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
</codetables>
```

Figure 6. CT_PodType.ctx, declaring a 'Pod-Type'

Declaring a new PodLoader

Next we need to declare our PodLoader. The PodLoader is the java class that will generate the fragment of XML that will populate our Pod. The `CT_PodLoaderBindings.ctx` codetable entry binds a Pod-Type to a PodLoader. When the infrastructure processes our Pod it will look up the PodLoader class in this codetable.

- The *value* field must match the value field on the PodType codetable. This is what binds the 2 codetable entries.
- The *description* field contains the fully qualified name of the PodLoader class.

```
<?xml version="1.0" encoding="UTF-8"?>
<codetables package="codetable">
  <codetable
    java_identifier="PODLOADERBINDINGS"
    name="PodLoaderBindings"
  >
    <code
      default="false"
      java_identifier="HELLOWORLD"
      status="ENABLED"
      value="PT9001"
    >
      <locale language="en" sort_order="0">
        <description>pods.podloaders.HelloWorld</description>
        <annotation/>
      </locale>
    </code>
  </codetable>
</codetables>
```

Figure 7. CT_PodLoaderBindings.ctx, declaring a 'PodLoader'

Now that we have added our codetable entries to the PodType and PodLoaderBindings files we will need to run the **`ctgen`** target to create the codetables and the **`database`** target to insert the codetable values into the database.

Creating a Pod using a PodLoader

The next step is to create our PodLoader class. The PodLoader extends the class `curam.cefwidgets.pods.pod.impl.PodLoader` and implements the `createPod` method. Create a new class on the Server by copying the example below into a class named `HelloWorld` in the package pods.loaders.

```
package pods.podloaders;

import java.util.Map;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import curam.cefwidgets.docbuilder.impl.PodBuilder;
import curam.cefwidgets.pods.pod.impl.PodLoader;
import curam.codetable.PODTYPE;

public class HelloWorld extends PodLoader {

  @Override
  public Node createPod(Document document, Map<String,Object> contexts) {
    try{
      PodBuilder helloWorld =
        PodBuilder.newPod(document, PODTYPE.HELLOWORLD);
      helloWorld.setTitle("Hello World");
      return helloWorld.getWidgetRootNode();
    }catch(Exception e){
      throw new RuntimeException(e);
    }
  }
}
```

Figure 8. A very simple PodLoader

Input:

The `createPod` method receives 2 parameters from the infrastructure that calls it.

Table 3. `createPod` method parameters

Parameter	Description
document	The Document parameter is an instance of a `org.w3c.Document` class. It is passed to the method by the infrastructure that calls it. The Document instance is used to create and append the 'pod' Node that describes the Pod.
context	The context parameter is used to pass page level paramtets to the Pods. Currently this is not support.

Output:

An instance of the `org.w3c.Node` object is returned by the createPod method.

Table 4. Return object from createPod

Return object	Description
org.w3cNode	The content of the Node returned must match a predefined schema. The PodBuilder class provides an API to create our 'pod' Node in the correct format.

In the example above, the simple Pod is created by creating a new instance of a PodBuilder class on line 16. The Document instance from the PodLoader and the codetable value from the PodType codetable are passed to the constructor. On line 17 we use the PodBuilder to set the title of the Pod. The PodBuilder builds a Node tree representing the Pod which is returned on line 18 as a Node object.

Adding a Pod to the Pod Container

The last piece of the jigsaw is adding the Pod to the Pod-Container. We will use the wizard provided in the Administrator application. We must login to the Administrator application, so we will need the username and password assigned to this application. When we have logged in we must open the admin wizard by...

1. Selecting the Administration Workspace section
2. Selecting the User Interface tab
3. Selecting Personalized Pod Pages

When the Personalized Pod Pages tab loads we can see the MyPodContainer page that we created in Chapter 2 in the list of Personal Pages. Selecting edit will open the wizard for maintaining our Personal Page. The first step lists all the Pods available for selection. In this list we find our Pod 'HelloWorld!'. Select the Pod and click next on the remaining steps finally saving the record. We have now added our Pod to our Pod Container. Log out of the Administrator application and log into the application that contains the Pod page.

Viewing the Pod

Now lets see our Pod in action. Login to the application and go to the Pod page. When the page loads it will be empty except for the buttons in the top right corner. Click the customize button to open the *customization-console*. We can see our Pod listed in the console. Select the checkbox beside the Pod and choose save. The page will reload with our Pod defaulted to the top right corner.

You will notice that the Pod contains some text *NO CONTENT* which is a place holder added by the infrastructure when the Pod contains no content. In the next chapter we will create another Pod with some content and take a closer look at the PodBuilder class.

Review

Lets take a look back on what we did.

- We started by adding our new Pod to the PodType and PodLoaderBindings codetables.
- We then created a PodLoader where we used the PodBuilder class to create our Pod and add the title.
- We used the wizard in the Administrator application to add our new Pod to the PodContainer.
- We used the customization-console to select and view our new Pod.

In the next chapter we will create a new Pod with some more interesting content.

Creating a Pod with a list

Introduction

In this chapter we will expand on what we did in the previous chapter by adding some content to a Pod and we will use the tools provided for creating the basic content types.

Lets begin with a new Pod which we will add to our Pod-Container in the same way we added the Hello World! Pod in the previous chapter. We will use a movies theme for our examples, so lets create a Pod with a short list of our favourite movies.

Creating a new list Pod

Register new Pod

In the same way we did in the previous chapter we are going to register a new Pod and bind it to a PodLoader by adding the codetable entries in the PodType and PodLoaderBindings tables using the examples below.

```
<code
    default="false"
    java_identifier="MYFAVMOVIES"
    status="ENABLED"
    value="PT9002"
  >
    <locale language="en" sort_order="0">
      <description>My Favourite Movies</description>
      <annotation/>
    </locale>
  </code>
```

Figure 9. Adding a new PodType to CT_PodType.ctx

```
<code
    default="false"
    java_identifier="MYFAVMOVIES"
    status="ENABLED"
    value="PT9002"
  >
    <locale language="en" sort_order="0" >
      <description>pods.podloaders.MyFavouriteMovies</description>
      <annotation/>
    </locale>
  </code>
```

Figure 10. Adding PodLoader binding

Create a new PodLoader

Next we add our PodLoader class to our loaders package remembering to reference the new codetable value we created in the *PodType* codetable when we construct our new Pod using the PodBuilder.

```
package pods.podloaders;

import java.util.Map;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import curam.cefwidgets.docbuilder.impl.PodBuilder;
import curam.cefwidgets.pods.pod.impl.PodLoader;
import curam.codetable.PODTYPE;

public class MyFavouriteMovies extends PodLoader {

  @Override
  public Node createPod(Document document, Map<String,Object> contexts) {
    try{
      PodBuilder moviesPod =
        PodBuilder.newPod(document, PODTYPE.MYFAVMOVIES);
      moviesPod.setTitle("My Favourite Movies");
      return moviesPod.getWidgetRootNode();
    }catch(Exception e){
      throw new RuntimeException(e);
    }
  }
}
```

Figure 11. Creating a PodLoader class

Log into the Administrator application and add the new Pod to the Pod-Container in the same way we did in the previous chapter.

Open the Pod page and ensure that our Pod is visible.

Create the list

Now that we have our Pod in place we can add content to it. The `PodBuilder` class provides an `addContent(...)` method to add the content to a Pod. In our movies example we are going to delegate to the list widget which can generate a HTML *table*.

To start we will need to provide the movies for our list. Appendix A.1 contains a full program-listing for a Java class that will act as a simple read-only DB of our favourite movies. Add this class to a package in the project where it can be accessed by our PodLoader.

Next we will create a list in our PodLoader and populate it with our favourite movies. In the PodLoader add the following code to the `createPod` method before the return statement.

```
001 public Node createPod(Document document, Map<String,Object> contexts) {
002   try{
003     PodBuilder moviesPod =
004       PodBuilder.newPod(document, PODTYPE.MYFAVMOVIES);
005     moviesPod.setTitle("My Favourite Movies");
006
007     MoviesDB moviesDB = new MoviesDB();
008
009     Collection<MoviesDB.Movie> favMovieCollection =
010       moviesDB.getAllMovies();
011     Iterator<MoviesDB.Movie> movieList =
012       favMovieCollection.iterator();
013
014     // Create the list
015     ListBuilder myFavouriteMovies =
016       ListBuilder.createList(1, document);
017
018     int row = 1;
019     while(movieList.hasNext()) {
020       Movie movie = movieList.next();
021       String movieName = movie.title;
022       myFavouriteMovies.addRow();
023       myFavouriteMovies.addEntry(1, row++, movieName);
024     }
025
026     RendererConfig contentRenderer = new RendererConfig(
027         RendererConfigType.STYLE, "single-list");
028     moviesPod.addContent(myFavouriteMovies, contentRenderer);
029
030     return moviesPod.getWidgetRootNode();
031   }catch(Exception e){
032     throw new RuntimeException(e);
033   }
```

Figure 12. Adding a list to a Pod

Compile your PodLoader class and reload the Pod page. The 'My Favourite Movies' Pod will be updated with the list of our favourite movies.

In the next section we will look in more detail at how the list was created.

Deconstructing the code

Constructing the list

A Pod does not need to know what its content will be. At runtime the Pod will delegate to other widgets to produce the HTML that will render the content. Our movies Pod is a list of movie names and it will re-use another widget to return a HTML table containing the list data. Like the PodBuilder the ListBuilder is an API for creating lists that conform to the schema for a renderer called `ListBodyRenderer`. The ListBuilder generates a fragment of XML that describes a list and at runtime the `ListBodyRenderer` will translate this XML into the HTML that can be added to the body of a Pod. To build the Pod content for our Pod the PodLoader will use the ListBuilder to produce the list of movies.

The first step in creating our list is to construct a new `ListBuilder` object. The constructor on line 16 accepts an *int* value which is the number of columns in the list. The second parameter is a `org.w3c.Document`. The document parameter represents the overall PodContainer to which our Pod will be added. The document object is used to create the new Nodes that represent our Pod and its content. Those Nodes will be appended to the some part of the document object.

```
015      ListBuilder myFavouriteMovies =
016        ListBuilder.createList(1, document);
```

Adding rows

Next we iterate over our movies. For each movie we add a new row (line 22).

```
019      while(movieList.hasNext()) {
020        Movie movie = movieList.next();
021        String movieName = movie.title;
022        myFavouriteMovies.addRow();
023        myFavouriteMovies.addEntry(1, row++, movieName);
024      }
```

Creating content in the cells

We use the `addEntry(...)` method to add content to cells. This method accepts a column, a row and a Java Object which represents the content to be added to the cell.

Table 5. ListBuilder.addEntry(...) parameters

Parameter	Type	Descrption
col	int	The column index, offset 1.
row	int	The row index, offset 1.
content	Object<?>	A Java Object that represents the content. The List Renderer can accept a number of different types including CodetableItems and LocalizedString objects which it will process for display. (See Javadoc for ListBuilder)

In our movies Pod we want to add a list of movie names so we pass a Java String in the *content* parameter. On lines 19 to 24 we iterate over the collection of movies.

```
023        myFavouriteMovies.addEntry(1, row++, movieName);
```

Adding the list to a Pod

Now that the list is populated we insert it into the body of our Pod.

The `addContent(...)` method provides the mechanism for adding our Pod content. The method accepts as it's first parameter either a `org.w3c.Node` or a `WidgetDocumentBuilder` object (which internally is converted to a Node using the `getWidgetRootNode` operator of the WidgetDocumentBuilder object).

The second parameter is a configuration for a Renderer that will create the HTML for our Pod content. The `RendererConfig` object specifies the type of configuration (Style or Domain) and name of a renderer configuration entry. Configuring renderers is covered in detail in the Curam Widget Development Guide.

Table 6. PodBuilder.addContent(...) parameters

param	type	descrption
content	Node | WidgetDocumentBuilder	The Node object is appended to the instance of org.w3c.Document that was passed to the constructor of the PodBuilder.
rendererConfig	RendererConfig	The RendererConfig object nominates the Renderer that will process the *content* parameter.

```
026     RendererConfig contentRenderer = new RendererConfig(
027         RendererConfigType.STYLE, "single-list");
028     moviesPod.addContent(myFavouriteMovies, contentRenderer);
```

The movies Pod uses the `ListBodyRenderer` which is invoked using a Style configuration called "single-list". On line 28 we add the list widget with the renderer configuration for a list to the body of the Pod.

The Pod is now complete. The content of our movies list is defined in the ListBuilder object which is added to our Pod. The ListBodyRenderer will generate the HTML table which will be appended to out Pod body.

Adding a Pod filter

Introduction

In this chapter we will explore Pod filters. We will look at the existing filters available and we will use one to add a filter to our movies Pod.

What is a Pod filter?

A Pod can optionally include a Pod filter. The filter allows a user to refine the information presented in the Pod. For example, some Pods display reports as charts that are based on periods of time. A Pod filter may present a selection of time periods which the user can select to re-draw the Pod with a different chart representing the selected time period.

Types of filter

The ChoiceRenderer is a generic renderer for a number of filter style renderers, such as checkboxes, radiobuttons, and dropdowns. The ChoiceRenderer delegates to a specific renderer depending on what displayType is selected by the ChoiceBuilder.

The table below lists the existing filter renderers. The type and displayType combine to select a specific renderer.

Table 7. Filter Types

Filter	CT*	Type	Display Type	Renderer
Checkbox	Y	multiple	n/a	CTCheckboxSelectRenderer
Radiobutton	Y	single	n/a	CTRadiobuttonSelectRenderer
Radiobutton	N	db-single	n/a	RadiobuttonSelectRenderer
Dropdown	Y	single	dropdown	CTDropdownSelectRenderer
Dropdown	N	single	listdropdown	ListDropDownSelectRenderer

Note: CT *, Denotes a filter based on the values in a specific codetable file.

Adding a Drop Down Filter

To demonstrate the use of filters we are going to create a filter for our movies Pod. The filter will select movies by genre. As we did in the last chapter we will insert the complete code sample first to see the Pod in action, then we will step through the code to see what we did to create the filter.

Appendix A.4 contains a program listing, replace the createPod method in the `MyFavouriteMovies` PodLoader with the version from the Appendix. Compile the PodLoader and launch the Application.

When the page loads the Pod will be updated to include a filter feature denoted by the pen icon on the title bar.

Open the filter by clicking on the pen icon. Select a genre from the drop-down. Use the Save button to save the selection and reload the list. The list will only return movies that match the selected genre in the dropdown.

Lets look at the steps we took to create the filter.

Creating the Pod Filter

To add a filter to the Pod we need to use the `PodBuilder.addFilter(...)` method which accepts a parameter of type `PodFilter`. The `PodFilter` object specifies the id of the filter and the renderer configuration that will be used to invoke the render that will create the filter. In our example we are creating a filter with the id "genre" and we will be using a renderer called the ChoiceRenderer to render the content of the filter.

```
010      RendererConfig filterRenderer =
011        new RendererConfig(RendererConfigType.DOMAIN, "CT_CHOICE");
012
013      // Create the PodFitler
014      PodFilter genreFilter =
015        new PodFilter("genre", document, filterRenderer);
```

Figure 13. Creating the Pod Filter

On line 10-11 we create a renderer configuration which is mapped to a domain 'CT_CHOICE'. This configuration will invoke a renderer called `ChoiceRenderer`. We then create a PodFilter object passing an id, the document instance of the PodLoader and the renderer configuration.

Creating the options

Now that we have the basic framework of our filter we need to add our choices. The filter can be described as a set of options and a set of selections, which are a subset of the options. Collectively we refer to these as the 'choices'. As we are using the ChoiceRenderer to create the drop-down list, so we can use the ChoiceBuilder to create the content that we pass to the ChoiceRenderer. The ChoiceBuilder accepts a HashMap which is the set of id's and values. In our case the values will be the list of genres that we display in the drop down. In this simple example we use the lower case version of the value as the id.

```
018      HashMap<String, String> genres = new HashMap<String, String>();
019      genres.put("all", "- All -");
020      genres.put("horror", "Horror");
021      genres.put("drama", "Drama");
022      genres.put("romance", "Romance");
023      genres.put("comedy", "Comedy");
024      genres.put("action", "Action");
025
026      // Create the options and selections using the ChoiceBuilder.
027      ChoiceBuilder choices =
028        ChoiceBuilder.newInstance(genres, document);
```

Figure 14. Creating the set of Choices for the genre drop-down

Creating the selections

The next step is adding the selected values. In most cases you will want this to be the last saved selections. We can retrieve these values because they are saved for each filter every time a save action occurs on the container. The `PodLoader` class provides a `getPodFilterById(...)` which will return the selected values for each Pod filter.

```
031      Node genreSelectionNode =
032        getPodFilterById(PODTYPE.MYFAVMOVIES, "genre", document);
033
034      // Convert the Node to an ArrayList.
035      ArrayList<String> selectedGenres =
036       PodFilter.convertSelectionsNodeToArrayList(genreSelectionNode);
037
038      // Create a default genre selection.
039      if (selectedGenres.isEmpty()){
040        selectedGenres.add("all");
041      }
042      choices.addSelection(selectedGenres.get(0));
```

Figure 15. Retrieving the saved selections and adding them to the Pod filter

On line 32 we use the `getPodFiltersById(...)` method to return the saved selections for the 'genre' filter on the 'MYFAVMOVIES' Pod. The values are returned as a Node object in the raw format that they were encoded and stored as. The `PodFilter.converSelectionsNodeToArrayList(Node)` utility is used to convert the values into a list of String values. On line 42 we add the selected value, in this case it is the only value returned in the array.

Setting the type of filter

In our example we are using the ChoiceRenderer to create a dropdown list. The ChoiceRenderer delegates to a specific renderer depending on what displayType is selected by the ChoiceBuilder. We are creating a drop down list which is not based on a codetable, so we selected "listdropdown" for the the display type.

```
choices.setTypeOfDisplay("listdropdown");
```

Figure 16. Setting the type of filter

Adding a label and CSS styling

Optionally we can add a label to the filter by passing a String * to the `addFitlerLabel(...)` method. Custom styling can also be applied to the filter by passing CSS class names to the `addCSSClasses(...)`

Note: * The filter label is configured for localization. The String passed to the addFilterLabel method is assumed to be a key in a properties file associated with the Pod. If no property value is returned by the key the key is used as the label. Localization is covered in Chapter 8: Localization in Pods

```
genreFilter.addFilterLabel("Genre");
genreFilter.addCSSClasses("genre-filter");
```

Figure 17. Adding a PodFilter to a Pod

Add the Filter to the Pod

Next we add the filter to the Pod by passing it to the `PodFilter.addFilter(...)` method.

```
moviesPod.addFilter(genreFilter);
```

Figure 18. Adding a PodFilter to a Pod

Filtering your Pod

The final task is to filter the content of the Pod. In the movies example we will want to filter out all movies where the genre does not match the currently selected one.

```
  if (selectedGenre.equals(movie.genre)
      || selectedGenre.equals("all")){

    myFavouriteMovies.addRow();
    myFavouriteMovies.addEntry(1, row++, movieName);
  }
```

Figure 19. Filter the movies by genre

So that completes our filter. When the Pod is loaded for the first time no value will be stored for the filter. Every subsequent save will store the filter value, even if that is an empty String. When the Pod reloads it will use the saved value to filter the list of Movies, and it will also pass the stored value back to the filter for display so that we can see what filter is being applied.

Using the `PodBuilder`, `PodFilter` and `ChoiceBuilder` has meant that there was no requirement to create Renderers. The builder classes allow us to reuse existing renderers. There will however be occasions where you want to create a custom filter type. In the next chapter we will demonstrate how to create a new filter renderer.

Creating new Pod filters

Introduction

In this section we are going to create a new filter for a Pod to demonstrate how to add form items to Pods.

To complete this section you will need to create a Renderer so you will need to be familiar with building Renderers and topics such as source paths, target paths and marshallers. These are covered in the Curam Widget Development Guide. This chapter will assume you have a good working knowledge of the renderers.

We will start with some simple definitions which you should already be familiar with from the Curam Widget Development Guide

Renderer
A Java class that generates HTML markup.

Marshaller
A Java class used to access properties of a server interface and pre-processes data retrieved from a field

Source Path
A pointer used when accessing server interface properties.

Target Path
A pointer used for accessing the content of form fields.

In this example we will create a simple text filter that filters by movie title. To create our new filter we are going to...

- Create a Pod filter Renderer
  - Map the source path
  - Map the target path
  - Create the text box
- Create a configuration for the Pod filter Renderer.
- Update our movies PodLoader.
  - Create a new PodFilter that uses our new filter Renderer.

Create a Pod filter Renderer

Appendix A.3 contains a program listing for a `PodTextFilterRenderer`. Add this class to your component in the webclient project in a package named `sample` under the `javasource` folder. Below we will step through the important code

Preparing to delegate

We start our Renderer by creating a `FieldBuilder`. We do this because our Renderer isn't going to do all the work. It will delegate the task of rendering the input box to an existing Renderer. The `FieldBuilder` will store up the settings that we pass to that Renderer.

```
    Field field = ((Field)component);

    final FieldBuilder fieldBuilder =
      ComponentBuilderFactory.createFieldBuilder();
    fieldBuilder.copy(field);
```

Figure 20. Setting up a FieldBuilder

Setting a source path

In the code extract in Example 7.2 we have extended the source path received to access the text for the filter. The text will be stored in a Document Node named *text-filter* (we will create that later in the PodLoader). We use the data accessor to retrieve the text that will be added to the input box.

```
    String sourcePathExt = "text-filter";
    Path sourcePath =
      field.getBinding().getSourcePath().extendPath(sourcePathExt);
    fieldBuilder.setSourcePath(sourcePath);
```

Figure 21. Setting the source path

Setting a target path

Next we extend the target path. We need to extend the target path to ensure the form item value is processed by the Marshaller attached to the Pod-Container. The Marshaller is configured to process a number of specific target paths. Example 7.3 shows how to extend the target path in the correct format.

```
    String targetPathExt =
      "choice/" + field.getID() + "/selected-options";
    Path targetPath =
      field.getBinding().getTargetPath().extendPath(targetPathExt);
    fieldBuilder.setTargetPath(targetPath);
```

Figure 22. Setting the target path

Note: The `PodFiltersRenderer` passes an Id value to the Renderer it invokes. The Id is the concatenation of a *podID* and *filterID* in the format *podID/filterID. The Id value is retrieved by the called renderer using the* `getID()` *method. That renderer will use the Id to uniquely identify itself.*

```
choice/ podId
        /
        filterId
          /selected-options
        /option-value
         |--1--|
        --2--|
        ---3----|
         --------4-------|
        -----5------|
```

Figure 23. Format of a Pod filter target path

The extended target path is broken in to what are known as steps which are divided by the '/' character. Each step in our target path is defined below.

Table 8. Target Path break down

Step	Description
1	This acts as the marker for the marshal. The *'choice'* text indicates that this field is to be processed by the Pod-Container.
2	Contains the unique identifier (as specified in the PodType codetable) for the Pod to which the filter is attached. E.g. PT9001
3	Contains the unique identifier for the filter attached to the Pod. This Id is created when the `PodFilter` is constructed in the PodLoader.
4	The *selected-options* step indicates that this is a filter. Knowing this, the infrastructure will process the form values as a Pod filter.
5	The *option-value* step is optional and is used to uniquely identify selections in multi-select filters. E.g a checkbox filter can select more than 1 value, so each option gets an *option-value* step to distinguish it from it's siblings.

In our code extract in Example 7.3 we have extended the target path using the id passed from the `PodFiltersRenderer` to map our text input form item. At runtime its value will be...

```
choice/PT9001/title/selections
```

Figure 24. Format of a target path for My Favourite Movies Pod text filter

Creating the input field

The last section of our renderer creates the input field. It actually delegates the task to an existing Renderer which can create the input field for us. The `TextRenderer` is mapped to the *TEXT_NODE* Domain, so we simply set the Domain on our `FieldBuilder` instance and call the `render` function on that. The `TextRenderer` will create the form item and return the input box which is appended to the HTML document.

```
      fieldBuilder.setDomain(context.getDomain("TEXT_NODE"));
      DocumentFragment textFilter =
        fragment.getOwnerDocument().createDocumentFragment();
      context.render(fieldBuilder.getComponent(), fragment, contract);

      fragment.appendChild(textFilter);
```

Figure 25. Rendering the input box

Create a configuration for the Pod filter Renderer

In the StylesConfig.xml in your component add the following entry. The 'style' name will be used in our PodLoader to configure the `PodFilter` to use our new `PodTextFilterRenderer`. We will need to execute the build target for the client to add this configuration.

```
<sc:style name="pod-text-filter">
    <sc:plug-in
      class="sample.PodTextFilterRenderer"
      name="component-renderer"
    />
  </sc:style>
```

Figure 26. Style configuration for Pod filter Renderer

Create a new PodFilter in the PodLoader

Now that we have created our filter all that remains is to invoke it in our PodLoader and use the saved value to filter our list of Movies. Appendix A.6 contains the updated version of the `createPod(...)` method. The code extract below shows the specific code that creates our text filter and adds it to the Pod.

```
      // Create the configuration for the filter renderer.
      RendererConfig titleFilterRenderer =
        new RendererConfig(RendererConfigType.STYLE, "pod-text-filter");

      // Create the filter.
      PodFilter titleFilter =
        new PodFilter("title", document, titleFilterRenderer);
      titleFilter.addFilterLabel("Title");

      // Retrieve the saved filter value and extract to an array
      Node titleTextNode =
        getPodFilterById(PODTYPE.MYFAVMOVIES, "title", document);
      ArrayList<String> titleTextArray =
        PodFilter.convertSelectionsNodeToArrayList(titleTextNode);

      // Create the Node that the filter Renderer expects and add the
      // saved filter text to it.
      String titleFilterText = "";
      if (!titleTextArray.isEmpty()) {
        titleFilterText = titleTextArray.get(0);
      }
      Element titleFilterNode = document.createElement("text-filter");
      titleTextNode = document.createTextNode(titleFilterText);
      titleFilterNode.appendChild(titleTextNode);
      titleFilter.addFilter(titleFilterNode);

      // Add the title filter to the Pod
      moviesPod.addFilter(titleFilter);
```

Figure 27. Adding the Pod Text filter

Create a new filter:

In lines 10-11 we create the configuration for our new filter by referencing the style we created in the StylesConfig.xml. We pass this to the PodFilter constuctor along with the id of our filter, 'title' in this case.

Retrieve saved filter values:

In lines 19-22 we use the utility functions to return the saved values for our 'title' filter and convert them to an array for ease of use.

Create input to Renderer:

In lines 19-33 we create the text Node that will passed to our Renderer. The Renderer is expecting a Node named "text-filter" so we create this and add the filter text to it. We add the Node to our `PodFilter` object using the `addFilter(...)` method.

Add the filter to the Pod:

Finally we pass our `PodFilter` object to the `addFilter(...)` method of our `PodBuilder` object.

When we iterate over the movies we only select movies whose title contains the substring that was returned from the filter. When we put it all together we can load our Pod, select the pen icon to open the filter, choose a genre and click save. The page will re-draw with the new filtered list.

Localization in Pods

Introduction

In this chapter we are going to look at building Pods in a localizable manner. The examples provided use non-locale-specific properties file, these can be supplemented with locale specific versions to return translated text if required. The Curam Widget Development Guide has a Chapter on Internationalization and Localization for widgets which covers this topic in more detail and the Curam Regionalization Guide discusses building a locale aware product.

To demonstrate the features built into the framework of Pods to support localization we will update our movies Pod to source various fields from property resources.

The textresource property

For each of the existing renderers used with Pods a 'textresource' attribute can be set that defines a resource property file. The code extract in Example 8.1 shows a renderer reading a property from a text resource file. The file name is passed in the XML received by the renderer (See example 8.2)

```
private static final String RESOURCE_FILE_PATH = "@textresource";
  ...
  ...
  String textResource = context.getDataAccessor().get(
    field.getBinding().getSourcePath().extendPath(
      RESOURCE_FILE_PATH));
  Path textPath =
    ClientPaths.GENERAL_RESOURCES_PATH.extendPath(textResource);
  ...
  ...
  final String saveButtonText =
    context.getDataAccessor().get(
      textPath.extendPath("button.save.text"));
```

Figure 28. A Renderer reading a property from a text resource file

In the example above the Renderer is expecting to receive the name of the text resource file in the 'textresource' attribute of the document Node it receives.

```
<pod textresource="sample.i18n.MyFavouriteMovies" ...>
    <config>
      ...
    </config>
    <data>
      ...
    </data>
  </pod>
```

Figure 29. Example of a document Node input to a Pod renderer

The Renderer uses the `ClientPaths` class to create a pointer to the text resource file. The value of the property is retrieved by extending the path into the file to point at the specific property. The path extension is the property key. The value returned is the property value. If the request is made for a specific locale, and the resource file for that locale has been provided then ClientPaths class will access the property in the appropriate resource file.

```
pod.title=My Favourite Movies
  pod.filter.genre.label=Genre
  ....
```

Figure 30. MyFavouriteMovies.properties

The location of the properties file must be on the classpath of the client project. Adding the properties file to the javasource folder will achieve this. The convention is to add property files to a folder called i18n to differentiate them.

Setting the text resource

A number of Renderers for producing standard content types in Pods are provided. Each of these Renderers has an associated Builder class which acts as an API for the Renderer to simplify the task of generating content to pass to the Renderer.

Table 9. Builders & Renderers

Builder	**Renderer**
PodBuilder	PodBodyRenderer
ListBuilder	ListBodyRenderer
PodListBuilder	PodListBodyRenderer
LinkBuilder	LinkRenderer

Table 9. Builders & Renderers (continued)

Builder	Renderer
PodBuilder	PodBodyRenderer

The builder classes provide a `setTextResource(String)` method. At runtime each instance of the Renderer uses the properties file received in the 'textresource' attribute to retrieve values that can be localized. We'll see this in action in the next section.

Localizing the My Favourite Movies Pod

In this section we will update our Movies Pod to read the values from properties files instead of using hardcoded Strings. Lets start with a simple example, localizing the Pod title. We will create a properties file with a title property and then update our PodBuilder to reference this property.

Note: The full listing for the createPod method for all examples that follow can be found in Appendix A.8.

Localizing the Pod

Create a new file called 'MyFavouriteMovies.properties' in a folder called 'i18n' under the javasource/sample folder in the webclient project (If you haven't already created that folder you can do so now). In the file add the key *pod.title* with the value 'My Top Movies' which will distinguish it from the current title.

```
pod.title=My Top Movies
```

Figure 31. MyFavouriteMovie.properties

Update the code used to construct our Pod by setting a text resource and use the property key for the title of the Pod.

```
005      moviesPod.setTextResource("sample.i18n.MyFavouriteMovies");
006      moviesPod.setTitle("pod.title");
```

Figure 32. MyFavouriteMovies.java, sourcing the Pod title from a properties file

Compile the PodLoader class, build the client target and launch the application. When the Pod is loaded you will see the new title "My Top Movied" which has been read from the properties file.

Now we have a localizable Pod title! Lets do some more...

Localizing the filter

Next we will add localizable text to our filter labels. The Pod filter is tied to the Pod so it inherits the same resource file that we have given the Pod. In the same way that we did for the Pod title, we use a property key for the labels and add the property value to the properties file.

```
pod.title=My Top Movies
  pod.filter.title.label=Movie Title:
  pod.filter.genre.label=Select Genre:
```

Figure 33. MyFavouriteMovies.properties

```
...
        titleFilter.addFilterLabel("pod.filter.title.label");
  ...
        genreFilter.addFilterLabel("pod.filter.genre.label");
  ...
```

Figure 34. MyFavouriteMovies.java, using the properties file for labels

When we load our Pod we will see that the label on the filter has changed to the value specified in the properties file.

Localizing the movie list

Lets take one more example. This time we are going to use a properties file with our list of movies. To do this we are going to add a title to the list which will be sourced from a properties file.

- Create a new properties file MoviesList.properties and add it to the i18n folder.
- Build the client to publish the properties.
- Update the list to use the properties file and add a column title as a property key. (See Example 8.8)

```
        myFavouriteMovies.setTextResource("sample.i18n.MoviesList");
        myFavouriteMovies.addColumnTitle(1, "list.col1.title");
```

Figure 35. Adding a column title

Sharing properties files

The last example of localizing the list illustrates the value of sharing properties files. If we think about how a Pod is made up of various widgets, the complexity of which could extend to any number of widgets, then having 1 property file per widget would be difficult to maintain. For this reason it makes sense to share the same properties files for aggregated widgets such as Pods even though it is not technically necessary to do this.

In our example above, instead of creating a new properties file for the movies list widget, we can re-use the MyFavouriteMovies.properties file. Using this technique we have a single resource for all properties associated with the 'MyFavouriteMovies' Pod.

Progam Listings

The Movies DB: A Java class serving our favourite movies

```
package pods.podloaders;

  import java.util.Collection;
  import java.util.TreeMap;

  /** Simple read-only Java DB for a movie collection */
  public class MoviesDB {

    private TreeMap<Integer, Movie> allMovies;

    /** Constructor */
    public MoviesDB() {

      allMovies = new TreeMap<Integer, Movie>();
      allMovies.put(1, new MoviesDB.Movie(1,"The Dark Knight", "action",
        2008, "Christopher Nolan", "Christian Bale", 1));
      allMovies.put(2, new MoviesDB.Movie(2,"Casablanca", "romance",
        1942, "Michael Curtiz", "Humphrey Bogart", 3));
      allMovies.put(3, new MoviesDB.Movie(3,"Schindler's List", "drama",
        1993, "Steven Spielberg", "Liam Neeson", 7));
      allMovies.put(4, new MoviesDB.Movie(4,"Alien", "horror",
        1979, "Ridley Scott", "Sigourney Weaver", 1));
      allMovies.put(5, new MoviesDB.Movie(1, "The GodFather, Part II",
        "drama", 1974, "Francis Ford Coppola", "Marlon Brando", 6));
      allMovies.put(5, new MoviesDB.Movie(1, "Toy Story 3",
       "comedy", 2010, "Lee Unkrich", "Tom Hanks", 2));
      allMovies.put(6, new MoviesDB.Movie(6, "Toy Story 2",
          "comedy", 1999, "John Lasseter", "Tom Hanks", 0));

    }

    /** Return all movies as a Collection */
    public Collection<MoviesDB.Movie> getAllMovies(){
      Collection<MoviesDB.Movie> movieCollection =
        this.allMovies.values();
      return movieCollection;
    }
    /** Return a movie by its Id */
    public Movie getMovieById(Integer id) {
      return allMovies.get(id);
    }

    class Movie {

      public int id,year,oscars;
      public String title, genre, director, leadrole, url;

      public Movie(int id,String title,String genre,
        int year,String director,String leadrole, int oscars){
       this.id = id;
       this.title = title;
       this.genre = genre;
       this.year = year;
       this.director = director;
       this.leadrole = leadrole;
       this.oscars = oscars;

      }
    }
  }
```

Figure 36. This class is the helper for our examples. It is a simple read-only Java DB for our favourite Movies. Feel free to add your personal favourites!

Chapter 4: Hello World Pod-Loader

```
package pods.podloaders;

import java.util.Map;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import curam.cefwidgets.docbuilder.impl.PodBuilder;
import curam.cefwidgets.pods.pod.impl.PodLoader;
import curam.codetable.PODTYPE;

public class HelloWorld extends PodLoader {

  @Override
  public Node createPod(Document document, Map<String,Object> contexts) {
    try{
      PodBuilder helloWorld =
        PodBuilder.newPod(document, PODTYPE.HELLOWORLD);
      helloWorld.setTitle("Hello World");
      return helloWorld.getWidgetRootNode();
    }catch(Exception e){
      throw new RuntimeException(e);
    }
  }
}
```

Figure 37. This is the simplest Pod-Loader you can have

Chapter 5: My Favourite Movies Pod-Loader

```
public Node createPod(Document document, Map<String,Object> contexts) {
  try{
    PodBuilder moviesPod =
      PodBuilder.newPod(document, PODTYPE.MYFAVMOVIES);
    moviesPod.setTitle("My Favourite Movies");

    MoviesDB moviesDB = new MoviesDB();

    Collection<MoviesDB.Movie> favMovieCollection =
      moviesDB.getAllMovies();
    Iterator<MoviesDB.Movie> movieList =
      favMovieCollection.iterator();

    // Create the list
    ListBuilder myFavouriteMovies =
      ListBuilder.createList(1, document);

    int row = 1;
    while(movieList.hasNext()) {
      Movie movie = movieList.next();
      String movieName = movie.title;
      myFavouriteMovies.addRow();
      myFavouriteMovies.addEntry(1, row++, movieName);
    }

    RendererConfig contentRenderer = new RendererConfig(
        RendererConfigType.STYLE, "single-list");
    moviesPod.addContent(myFavouriteMovies, contentRenderer);

    return moviesPod.getWidgetRootNode();
  }catch(Exception e){
    throw new RuntimeException(e);
  }
```

Figure 38. This version of the createPod method creates a list of movies using the MoviesDB class

Chapter 6: My Favourite Movies Pod-Loader for Pod filter example

```
public Node createPod(Document document, Map<String,Object> contexts) {
  try{
    PodBuilder moviesPod =
      PodBuilder.newPod(document, PODTYPE.MYFAVMOVIES);
    moviesPod.setTitle("My Favourite Movies");

    MoviesDB moviesDB = new MoviesDB();

    // Create the configuration for the drop down filter.
    RendererConfig filterRenderer =
      new RendererConfig(RendererConfigType.DOMAIN, "CT_CHOICE");

    // Create the PodFitler
    PodFilter genreFilter =
      new PodFilter("genre", document, filterRenderer);

    // Create genre list
    HashMap<String, String> genres = new HashMap<String, String>();
    genres.put("all", "- All -");
    genres.put("horror", "Horror");
    genres.put("drama", "Drama");
    genres.put("romance", "Romance");
    genres.put("comedy", "Comedy");
    genres.put("action", "Action");

    // Create the options and selections using the ChoiceBuilder.
    ChoiceBuilder choices =
      ChoiceBuilder.newInstance(genres, document);

    // Return the last saved selection for the filter with id "genre".
    Node genreSelectionNode =
      getPodFilterById(PODTYPE.MYFAVMOVIES, "genre", document);

    // Convert the Node to an ArrayList.
    ArrayList<String> selectedGenres =
     PodFilter.convertSelectionsNodeToArrayList(genreSelectionNode);

    // Create a default genre selection.
    if (selectedGenres.isEmpty()){
      selectedGenres.add("all");
    }
    choices.addSelection(selectedGenres.get(0));
    choices.setTypeOfDisplay("listdropdown");

    genreFilter.addFilter(choices.getWidgetRootNode());

    // Add a filter label
    genreFilter.addFilterLabel("Genre");
    genreFilter.addCSSClasses("genre-filter");
    moviesPod.addFilter(genreFilter);


    Collection<MoviesDB.Movie> favMovieCollection =
      moviesDB.getAllMovies();
    Iterator<MoviesDB.Movie> movieList =
      favMovieCollection.iterator();

    // Create the list
    ListBuilder myFavouriteMovies =
      ListBuilder.createList(1, document);

    int row = 1;
    while(movieList.hasNext()) {
      Movie movie = movieList.next();
      String movieName = movie.title;
      String selectedGenre = selectedGenres.get(0);
      if (selectedGenre.equals(movie.genre)
          || selectedGenre.equals("all")){

        myFavouriteMovies.addRow();
        myFavouriteMovies.addEntry(1, row++, movieName);
      }
    }
```

Chapter 7: PodTextFilterRenderer for new Pod filter example

```java
package sample;

import org.w3c.dom.DocumentFragment;
import curam.util.client.ClientException;
import curam.util.client.model.Component;
import curam.util.client.model.ComponentBuilderFactory;
import curam.util.client.model.Field;
import curam.util.client.model.FieldBuilder;
import curam.util.client.view.RendererContext;
import curam.util.client.view.RendererContract;
import curam.util.common.path.DataAccessException;
import curam.util.common.path.Path;
import curam.util.common.plugin.PlugInException;
import curam.widget.render.infrastructure.AbstractComponentRenderer;

/**
 * Creates a text input for use with a Pod Filter
 */
public class PodTextFilterRenderer extends AbstractComponentRenderer {

  public void render(Component component, DocumentFragment fragment,
    RendererContext context, RendererContract contract)
    throws ClientException, DataAccessException, PlugInException {

    Field field = ((Field)component);

    final FieldBuilder fieldBuilder =
      ComponentBuilderFactory.createFieldBuilder();
    fieldBuilder.copy(field);

    // Update the source path to point at the text node
    String sourcePathExt = "text-filter";
    Path sourcePath =
      field.getBinding().getSourcePath().extendPath(sourcePathExt);
    fieldBuilder.setSourcePath(sourcePath);

    // Update the target path to use the Pod filter id
    String targetPathExt =
      "choice/" + field.getID() + "/selected-options";
    Path targetPath =
      field.getBinding().getTargetPath().extendPath(targetPathExt);
    fieldBuilder.setTargetPath(targetPath);

    // Use TextRenderer to create input box
    fieldBuilder.setDomain(context.getDomain("TEXT_NODE"));
    DocumentFragment textFilter =
      fragment.getOwnerDocument().createDocumentFragment();
    context.render(fieldBuilder.getComponent(), fragment, contract);

    fragment.appendChild(textFilter);
  }
}
```

Figure 40. This renderer will create the text filter that we use in Chapter 7 when we demonstrate how to create new filters for Pods

Chapter 7: My Favourite Movies Pod-Loader for new Pod filter example

```java
public Node createPod(Document document, Map<String,Object> contexts) {
  try{
    PodBuilder moviesPod =
      PodBuilder.newPod(document, PODTYPE.MYFAVMOVIES);
    moviesPod.setTitle("My Favourite Movies");

    MoviesDB moviesDB = new MoviesDB();

    // Create the configuration for the filter renderer.
    RendererConfig titleFilterRenderer =
      new RendererConfig(RendererConfigType.STYLE, "pod-text-filter");

    // Create the filter.
    PodFilter titleFilter =
      new PodFilter("title", document, titleFilterRenderer);
    titleFilter.addFilterLabel("Title");

    // Retrieve the saved filter value and extract to an array
    Node titleTextNode =
      getPodFilterById(PODTYPE.MYFAVMOVIES, "title", document);
    ArrayList<String> titleTextArray =
      PodFilter.convertSelectionsNodeToArrayList(titleTextNode);

    // Create the Node that the filter Renderer expects and add the
    // saved filter text to it.
    String titleFilterText = "";
    if (!titleTextArray.isEmpty()) {
      titleFilterText = titleTextArray.get(0);
    }
    Element titleFilterNode = document.createElement("text-filter");
    titleTextNode = document.createTextNode(titleFilterText);
    titleFilterNode.appendChild(titleTextNode);
    titleFilter.addFilter(titleFilterNode);

    // Add the title filter to the Pod
    moviesPod.addFilter(titleFilter);

    // Create the configuration for the drop down filter.
    RendererConfig filterRenderer =
      new RendererConfig(RendererConfigType.DOMAIN, "CT_CHOICE");

    // Create the PodFitler
    PodFilter genreFilter =
      new PodFilter("genre", document, filterRenderer);

    // Create genre list
    HashMap<String, String> genres = new HashMap<String, String>();
    genres.put("all", "- All -");
    genres.put("horror", "Horror");
    genres.put("drama", "Drama");
    genres.put("romance", "Romance");
    genres.put("comedy", "Comedy");
    genres.put("action", "Action");

    // Create the options and selections using the ChoiceBuilder.
    ChoiceBuilder choices =
      ChoiceBuilder.newInstance(genres, document);

    // Return the last saved selection for the filter with id "genre".
    Node genreSelectionNode =
      getPodFilterById(PODTYPE.MYFAVMOVIES, "genre", document);

    // Convert the Node to an ArrayList.
    ArrayList<String> selectedGenres =
     PodFilter.convertSelectionsNodeToArrayList(genreSelectionNode);

    // Create a default genre selection.
    if (selectedGenres.isEmpty()){
      selectedGenres.add("all");
    }
    choices.addSelection(selectedGenres.get(0));
    choices.setTypeOfDisplay("listdropdown");

```

Chapter 8: My Favourite Movies Pod-Loader for localization example

```
public Node createPod(Document document, Map<String,Object> contexts) {
  try{
    PodBuilder moviesPod =
      PodBuilder.newPod(document, PODTYPE.MYFAVMOVIES);
    moviesPod.setTextResource("sample.i18n.MyFavouriteMovies");
    moviesPod.setTitle("pod.title");

    MoviesDB moviesDB = new MoviesDB();

    // Create the configuration for the filter renderer.
    RendererConfig titleFilterRenderer =
      new RendererConfig(RendererConfigType.STYLE, "pod-text-filter");

    // Create the filter.
    PodFilter titleFilter =
      new PodFilter("title", document, titleFilterRenderer);
    titleFilter.addFilterLabel("pod.filter.title.label");

    // Retrieve the saved filter value and extract to an array
    Node titleTextNode =
      getPodFilterById(PODTYPE.MYFAVMOVIES, "title", document);
    ArrayList<String> titleTextArray =
      PodFilter.convertSelectionsNodeToArrayList(titleTextNode);

    // Create the Node that the filter Renderer expects and add the
    // saved filter text to it.
    String titleFilterText = "";
    if (!titleTextArray.isEmpty()) {
      titleFilterText = titleTextArray.get(0);
    }
    Element titleFilterNode = document.createElement("text-filter");
    titleTextNode = document.createTextNode(titleFilterText);
    titleFilterNode.appendChild(titleTextNode);
    titleFilter.addFilter(titleFilterNode);

    // Add the title filter to the Pod
    moviesPod.addFilter(titleFilter);

    // Create the configuration for the drop down filter.
    RendererConfig filterRenderer =
      new RendererConfig(RendererConfigType.DOMAIN, "CT_CHOICE");

    // Create the PodFitler
    PodFilter genreFilter =
      new PodFilter("genre", document, filterRenderer);

    // Create genre list
    HashMap<String, String> genres = new HashMap<String, String>();
    genres.put("all", "- All -");
    genres.put("horror", "Horror");
    genres.put("drama", "Drama");
    genres.put("romance", "Romance");
    genres.put("comedy", "Comedy");
    genres.put("action", "Action");

    // Create the options and selections using the ChoiceBuilder.
    ChoiceBuilder choices =
      ChoiceBuilder.newInstance(genres, document);

    // Return the last saved selection for the filter with id "genre".
    Node genreSelectionNode =
      getPodFilterById(PODTYPE.MYFAVMOVIES, "genre", document);

    // Convert the Node to an ArrayList.
    ArrayList<String> selectedGenres =
     PodFilter.convertSelectionsNodeToArrayList(genreSelectionNode);

    // Create a default genre selection.
    if (selectedGenres.isEmpty()){
      selectedGenres.add("all");
    }
    choices.addSelection(selectedGenres.get(0));
    choices.setTypeOfDisplay("listdropdown");

    genreFilter.addFilter(choices.getWidgetRootNode());

    // Add a filter label
    genreFilter.addFilterLabel("pod.filter.genre.label");
    genreFilter.addCSSClasses("genre-filter");
    moviesPod.addFilter(genreFilter);


    Collection<MoviesDB.Movie> favMovieCollection =
      moviesDB.getAllMovies();
    Iterator<MoviesDB.Movie> movieList =
      favMovieCollection.iterator();

    // Create the list
    ListBuilder myFavouriteMovies =
      ListBuilder.createList(1, document);
    myFavouriteMovies.setTextResource("sample.i18n.MoviesList");
    myFavouriteMovies.addColumnTitle(1, "list.col1.title");

    int row = 1;
    while(movieList.hasNext()) {
      Movie movie = movieList.next();
      String movieName = movie.title;
      String selectedGenre = selectedGenres.get(0);
      if (selectedGenre.equals(movie.genre)
          || selectedGenre.equals("all")){

        if (movieName.toUpperCase().indexOf(
            titleFilterText.toUpperCase()) != -1) {
          myFavouriteMovies.addRow();
          myFavouriteMovies.addEntry(1, row++, movieName);
        }
      }
    }

    RendererConfig contentRenderer = new RendererConfig(
        RendererConfigType.STYLE, "single-list");
    moviesPod.addContent(myFavouriteMovies, contentRenderer);

    return moviesPod.getWidgetRootNode();

  }catch(Exception e){
    throw new RuntimeException(e);
  }
}
```

Notices

Privacy Policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user’s name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and

Trademarks

Printed in USA