Building composite applications and templates in WebSphere Portal V6

Empower users to create their own well- structured, highly functional applications

Peter Fischer, Hendrik Haddorp, Thomas Stober
Websphere Portal Development
IBM Germany
July, 2006

Composite applications are key to implementing meaningful business value with SOA. WebSphere Portal V6 provides tools and infrastructure for creating and running composite applications. This article explains how business analysts and application designers can assemble applications that implement complex business logic using individual components, such as portlets, processes, or other code artifacts. These applications define the configuration and interaction of these components, as well as how the user interface should be arranged on the glass. You see how these new capabilities fit into SOA, and you see a detailed example involving a company we call "Funny Bikes" who creates and deploys a composite application.

This article is intended for multiple audiences, including:

- **Business analysts** and **application designers**, who want to easily implement their individual business logic without first having to develop programming skills.
- **Java developers**, who want to provide portlets and components leveraging the new application capabilities.
- **Software architects** and **technical decision makers**, who need an oerview of the concepts and capabilities of composite applications.

Table of contents

Introduction

This summer (northern hemisphere), IBM releases Version 6 of its award winning WebSphere Portal product. WebSphere Portal V6 is part of IBM's service-oriented architecture (SOA) architectural approach, and its composite applications is one of the key SOA concepts.

Composite applications aggregate a set of components into a single, coherent entity. The individual components and the interactions among these components can work together to implement complex business logic as defined by a business user. Components of a composite application can be virtually any code artifacts including Java™ classes, portlets, EJBs, or plain old Java objects (POJOs). Meta data describes the behavior and configuration for each involved component such as a reference to a document library or a workflow definition.

Composite applications involve two fundamental aspects: templates and applications. A *template* describes a composite application in an abstract way, including information which defines how complex business logic is assembled out of a given set of components. The template is an XML file which references all applicable components, (such as portlets or Java code artifacts), and it specifies meta information (such as specific configuration settings) for each component. You describe the composed

application behaviour in the template, and you define the desired interaction between the components, such as wires between portlets. You also define the desired access control logic to be enforced, such as application specific user roles. You can use the template to exchange application definitions between different systems. After creating a template, you can store it in a template library and make it available to your user community. So, to summarize, a template is an XML file which represents the abstract definition for an application; that is, it is the "blue print" for a composite application.

You create composite *applications (instances)* from a defined template. You can pick a template from the template library, and create new instances of the composite application based on the template definition. The instances run within an application runtime, which is referred to as the *composite application infrastructure* (CAI). End users can manage their own application instances, without having the need of administrative authority.

You can use a set of *tools* to create your own composite applications quickly. The tools are tailored for business users. These tools enable end-users to assemble and manage business logic from individual components, such as portlets, processes, or other code artifacts. By empowering users to define, create, and manage their own composite applications, WebSphere Portal V6 helps facilitate a strong business-driven usage model with fewer dependencies on support by system administrators.

Figure 1 shows how business components, templates, and the application instances relate to each other. The template is assembled from business components and stored as an XML description in the template catalog. The template instantiation service creates the component instances in the runtime environment.

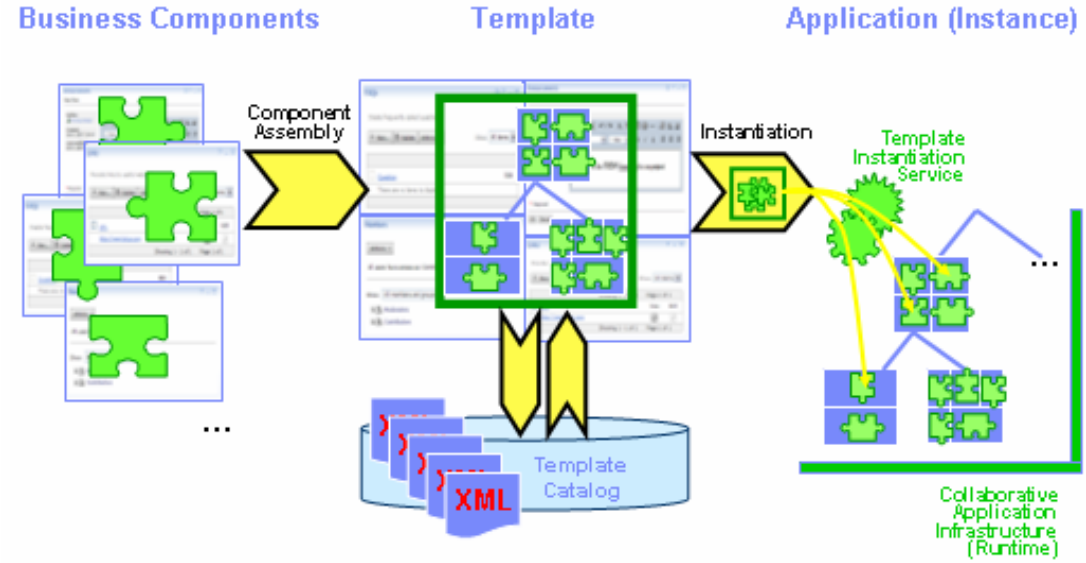


Figure 1. Composite applications and templates

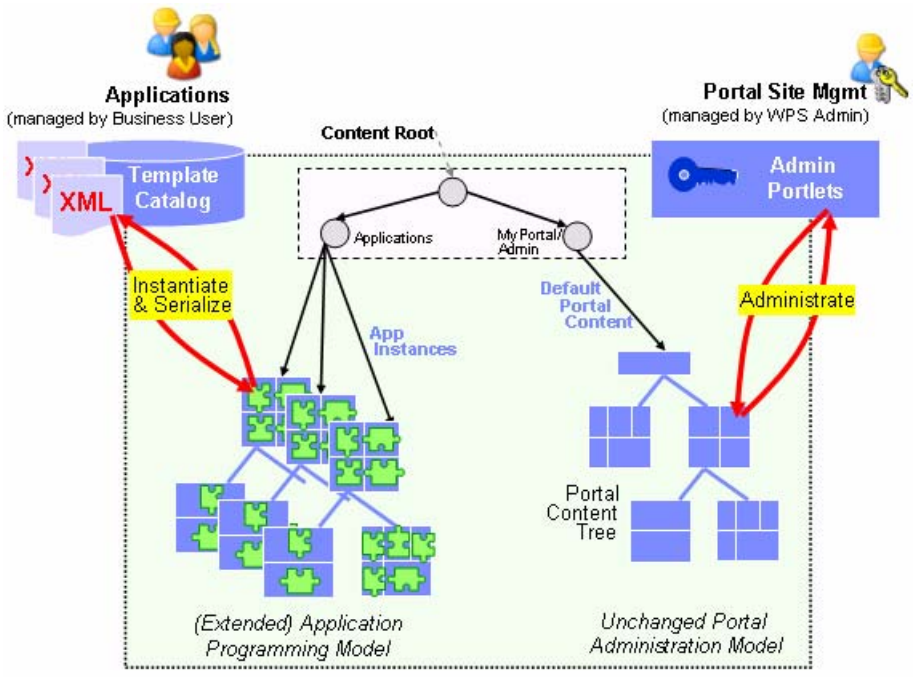
IBM's Workplace products, such as Workplace Collaboration Services, include similar capabilities. Workplace users can build templates and instantiate them to create *collaborative applications*. This technology has been incorporated into WebSphere Portal so that the large community of WebSphere Portal customers can use it as well.

You might ask: How will composite applications look to portal users? Will the architecture of existing customer sites need to change in order to use applications actively? What is the impact of this new feature on existing WebSphere Portal capabilities?

The answer, as illustrated in Figure 2 is this: Support for composite applications is strictly an additional capability that runs as an extension on top of the WebSphere Portal foundation layer. While the "classic" WebSphere Portal navigation and **MyPortal** content tree underneath remain untouched, a new parallel **label** has been introduced as an **application root node**. All instances of composite applications are placed underneath that node. The composite application infrastructure adds a valuable set of runtime enhancements to the sub tree (exclusively) underneath the application root node. You manage and use the instances, as well as the corresponding templates, through a set of administration portlets that are provide by the product. You learn more about these enhancements and portlets in the sections below.

Because these features are additions, there is no impact on the existing WebSphere Portal capabilities, neither in terms of performance nor in terms of usability.

Figure 2. Extended WebSphere Portal navigation



A sample scenario

Now that you have some background on the key concepts and enhancements related to composite applications, the rest of this article explains these features in more detail. First, you see a real-life example which walks through the steps to develop components, to

assemble an application template, and to create application instances. Then, you see how to run an instance within the application runtime infrastructure.

In this example, the application provider "Useful Templates Inc." is developing a generic "online shop" application. They want to provide a template, which describes the application's business logic and a corresponding user interface, and then sell this template. They already have a buyer, a dealer called Funny Bikes, who plans to use the template to create his own online shop. The dealer can easily customize his own application instance, which is based on the template, and then operate the online shop "out-of-the-box" on his own portal.

These Useful Templates Inc. employees are involved in this project:

1. **A component developer**, who will implement these application specific components:

    - A `catalog` portlet to display the list of articles available for online purchase. The portlet allows shop owners to manage their items and to categorize them. Customers can use the portlet to buy articles and place them into a shopping cart
    - A `business` component to retrieve the list of available categories with their articles from a database. This *service component* supports the catalog portlet, and involves writing custom code in Java.

2. A **user interface designer**, who will provide some elegant JSPs.
3. An **application designer**, who will assemble the composite application by dragging and dropping generic and custom code components into the composite application assembly. In addition to the two components created by the component developer, the application designer will add some "as-is" components to the application, such as an `FAQ` and a `newsgroup` portlet. He will also pre-define the application layout and pre-define the access control information.

The Funny Bikes dealer assigns employees to these roles to operate his shop:

1. A **portal administrator**, who has the necessary privileges to deploy the code artifacts, which are needed for the application execution.
2. A **business user**, who understands the business needs and can perform the instantiation of the template. He is responsible for the customization for this individual shop and will manage the user community who is using the shop.

Step 1: Develop the components

You can often create composite applications based on existing, ready-to-use common components, such as portlets from the WebSphere Portal catalog. In those cases, you can skip this step, and proceed to Step 2.

However, in some cases, you might need or want to develop your own components. In this example, let's assume you are the component developer in Useful Templates Inc., and you will create the `catalog` portlet and a supporting business component. The UI Designer will provide the necessary JSPs and add some impressive graphics. You will concentrate on the programming task.

Use cases for your `catalog` portlet

First, think about what you want to support with your `catalog` portlet. Of course, you want users to be able to list items. You also probably want to add and remove items. To create some structure, you introduce categories; for example, for a bike shop, you could have bikes, forks, and tires as categories with the corresponding items. In addition to listing the items in the categories, you want users to be able to add and delete categories as well.

So, you decide to support listing, adding, and deleting items and categories. All catalog users can list items and categories; however, regular users will not be able to add or delete items or categories; only administrators of the catalog will be permitted to do that.

Last but not least, your `catalog` portlet should have a name or title to identify the specific instance of the `catalog`.

Enable customization using parameters

The component you develop can be used within any composite application. Good programming practice is to make components as generic and flexible as possible so that you create a powerful library of building bricks for your applications.

One way to increase the flexibility of a component is to introduce parameters. Parameters in a component are “points of variability” (PoV), which can remain as placeholders during application assembly and template definition. The actual values are filled in by the end-user during the instantiation of the application. End-users can assign different values for different instances. That is, they can customize a generic template to create unique behavior for each instance. In this example, you set up parameters for the `catalog` portlet and for the corresponding business component.

Because every instance of the catalog component can be used for different kinds of content, it makes sense to turn the catalog’s title into a parameter. Whenever a template with a catalog component gets instantiated, the end-user will be asked for the title of this specific version of the catalog. Later in this article, you see how to programmatically declare parameters for business components and how to get the values entered by the user.

Because the portlet provides the user interface for your component, you want to define some parameters that influence the look-and-feel. To keep the coding simple, you choose the font’s size and colour to practice using the parameterization concept for portlets. Of

course later, you could choose more sophisticated parameters such as a selectable image or the name of a complete skin or theme.

Design the business component

Business components encapsulate the business logic and they take part in the application's lifecycle and community. The community includes the application's members, the application roles the members are in, and the mappings to the component specific roles. For example, the owner of a bike shop could be the admin of the catalog. You need to consider three things when you design a business component:

1. What business logic should the component provide through a corresponding custom business interface?
2. Upon which technology base or library should the business logic be implemented?
3. About which application infrastructure events should the component get notified?

After you decide on and implement these considerations, you need to create a deployment descriptor to register your business component in the system.

Design the business interface

Each business component instance is not a separate instance of a Java object. To design a business component interface, you use the *memento* pattern. You need to pass in the business component's instance ID to every call. How the instance ID gets created and how you can get it to work with your component is covered later in this article in Implementing the Lifecycle Interface.

Derive the methods for a suitable business interface from the use cases. In this example:

1. Choose `catalog` as the name for the business interface of your catalog component.
2. Next, define a method to get the name (title) of a catalog instance. To set the title for the catalog you use the application infrastructure's parameterization mechanism.
3. Then you need methods to get, add, and remove categories. To keep it simple, each category only has a name, so that you can use a single `String` to model the category.
4. The next functional area of your interface covers the management of articles in the shop. Again, you need methods to retrieve, add, and remove items; this time, the items are scoped to a corresponding category. To keep it as simple as possible, assume an item consists only of a `String` as the item's name. You can later extend the sample to cover additional data such as a localized title and description, article number, prize, and so on.

If you look at the use cases again, you might wonder what happened to the information about the current user so that you can check whether he or she is allowed to only read from or can also write to the catalog. Because the component runs in the context of portal requests, you can use the portal user management APIs (PUMA) to get the current user from within the implementation's code. Therefore, you can be sure that some user ID was not just passed in, but that this user has been authenticated by the portal.

Important: This article covers exception handling only where it is of special meaning in the context of the composite application infrastructure. In all other places, any references to exceptions are omitted from the code snippets to concentrate on the essentials.

Listing 1 The catalog's business interface

```
public interface Catalog
{
    String getTitle(String catalogID);

    List getCategories(String catalogID);
    void removeCategory(String catalogID, String category);
    void addCategory(String catalogID, String category);

    List getItems(String catalogID, String category);
    void removeItem(String catalogID, String category, String item);
    void addItem(String catalogID, String category, String item);
}
```

Choose the back-end storage to be used

After you know *what* you want to do, you need to decide *how* you want to do it. You need to persist your items and categories scoped by the business component instance, and you need to select the appropriate backend storage. Because you do not want to spend lots of time thinking about deployment, clustering, staging, or data structures, you choose the easy (and recommended) approach:

WebSphere Portal V6.0, which comes with a Java Content Repository (JCR - see JSR 170 for more information) and a document-oriented Content Management API (CMAPI) that handles almost everything for you.

What you do is:
1. Create one document library per catalog component instance.
2. Store the catalog title as title of the document library.
3. Create one folder per category, named like the category.
4. Create one document per item, named like the item.

Tip: If you want to extend the sample to cover more data for the items, you can easily store this data as properties of the document, and/or as the document's content.

Select the application callback interfaces

After you decide what to provide from a business perspective, you determine how to make your component an application infrastructure-compliant business component. A business component must implement various interfaces to become part of an application. To choose the correct interfaces, you first need to know what interfaces you can choose from.

The business component SPIs and APIs
In WebSphere Portal 6.0 the list of public interfaces for business components is still quite short and easy to grasp. So let's take a look at each of them.

- **Lifecycle**
  Contains callback methods for initialization and destruction of instances of a business component, and a method to provide supported parameters during creation.
- **Templatable**
  Contains callback methods for the serialization of a business component into a template and for the re-creation of a new instance based on the data within a template. You can use this interface to add business component-specific data to a template, and to instantiate your component again based on this data.
- **Membership**
  Enables a component to declare its supported roles, and contains callback methods to inform the business component whether users where added to or removed from a role that the business component supports. You use this interface to do business component-specific access control handling.
- **DisplayInfo**
  Use to provide localized title and description for your business component to be displayed whenever your component is listed in a UI (for example, the Manage Roles UI of WebSphere Portal V6.0).

Implementing the Lifecycle Interface
All business components must implement this interface to get instantiated (and destroyed) by the composite application infrastructure:

Listing 2: The Lifecycle interface

```
Lifecycle
{
   ListModel getCreateParameters();
   ListModel createInstance(ListModel params);
   void removeInstance(String instanceID);
}
```

The sample scenario uses the Lifecycle callback methods as follows. You can take a look at com/ibm/wps/app/bc/impl/CatalogImpl.java in the sample code for further details.

- **`getCreateParameters()`**
  The sample `catalog` component returns a variable with the name "title" as the only parameter that is supported during creation of the component.
- **`createInstance()`**
  In the createInstance method, it calls your component to create a new JCR document library using the CMAPI, stores the title (if passed in as a parameter), and returns the document library's ID as the business component ID. Because this ID will be passed in to any additional calls to the component, it is easy to find the corresponding document library with the correct content for every call.

- **`removeInstance()`**
  When the component is deleted again, it deletes the document library during the removeInstance call to clean up any resources of this instance.

Implementing the DisplayInfo Interface
If you don't want the user interface to display some cryptic, technical information about your `catalog` component, you can use the DisplayInfo interface to provide a localized title and a description for your component.

Listing 3: The DisplayInfo interface

```
DisplayInfo
{
   Localized getDisplayInfo(String id)
}
```

Implementing the Templatable Interface
You use the Templatable Interface to serialize a business component into a template, and to re-create a new instance from the data in a template.

Listing 4: The Templatable interface

```
Templatable
{
   ListModel serializeToTemplate(String id, Writer objectData);
   ListModel createFromTemplate(Reader objectData,
                                ListModel variables);
}
```

The sample implements the corresponding methods as follows:

- `serializeToTemplate()`
  You want to be able to create templates that can contain topic-oriented as well as generic data catalogs. For example, if you create a catalog for a bike shop, it would have categories with their category-specific items, such as a “bikes” category with different bikes the shop sells. If you create another bike shop (instance), you will probably have different bikes for sale, but you still need a “bikes” category.

  To enable such re-use, your `catalog` component stores the categories as part of the template − not the items − by writing them to objectData in a simple XML schema:

  ```
  <categories>
     <category name='categoryA'/>
     <category name='categoryB'/>
  </categories>
  ```

  Also, using the `serializeToTemplate` method , you can declare points of variability (PoV). You can return a ListModel which consists of objects implementing

the Variable interface. Each Variable will become a PoV for this component in the template. The end-user instantiating the template will be asked for a value using the Variables' title and description for the prompt. WebSphere Portal uses the value of the returned Variable to pre-fill the default value.

- `createFromTemplate`
  With a `createFromTemplate` a new instance of the component gets created – this time based on a template. There is no separate `Lifecycle.createInstance()` call for the template case. So, the first thing your business component has to do is to create a new document library.

  In that step you also cover the PoVs. Any PoVs for this component have already been prompted for at this point of time. The corresponding Variable objects with the newly entered values are the parameters passed in to that method.

  Your component now simply takes the value for the `title` Variable and stores it as the title of the document library.

  Before you can return the document library's ID as the business component ID (just as for Lifecyle) the passed in objectData must be handled. The corresponding Reader object lets you access the XML containing the categories that were added to the template during `serializeToTemplate()`. A simple SAX parser retrieves the category names from the XML and, for each one, creates a folder with that name in the newly created document library.

Implementing the Membership Interface
You use the Membership Interface to declare component-specific roles. It contains callback methods to inform the business component when users are added to or removed from a role that the business component supports.

Listing 2: The Membership interface

```
Membership
{
   ListModel getComponentRoles(String componentID);
   void memberAdded(String componentID, ObjectID principalOID,
                    String roleID);
   void memberRemoved(String componentID, ObjectID principalOID,
                      String roleID);
}
```

The sample uses the Membership's callback methods like this:

- `getComponentRoles()`
  The `catalog` component offers two roles.
  - Users in the **Admin** role can manage categories and items.
  - Users in the **User** role can only read categories and items from the catalog.

- `memberAdded()`
  In WebSphere Portal V6.0, there are no public APIs to store and access the component role mappings. Therefore, you need to store the component-specific role mappings yourself. Because you are already have a component instance-specific document library available, you can simply use a JCR document to store the role maps as a XML document like this:

  ```
  <maps>
     <map name='user_object_id_1' role='User'/>
     <map name='user_object_id_2' role='Admin'/>
  </maps>
  ```

  The role mappings are read in again for every access to a method of the business interface to check whether the current user is in a role that allows for the corresponding access. You use a simple SAX parser to extract the name and role combinations from the XML. The sample code does this for every check; you might consider adding caching of that type of information in a production environment.

  Important: Because the role mapping document is stored in the JCR, it could be accessed through other applications as well. In a production implementation you should make sure that only authorized people can modify this document.

- `memberRemoved()`
  Whenever a member is removed from one of the component's roles, the corresponding entry is removed from the document. (To be precise, a new XML document with the latest role mappings is stored in the document.)

Write the deployment descriptor

Let's assume you have implemented all of the interfaces above and are finished with the Java coding of the business component. Now you need to make your component available in the system.

To deploy a business component:

1. Create a plugin.xml deployment descriptor to register the business component at the system as an (eclipse) extension.

   The plugin.xml must contain the namespace for the business component (the plugin `id` value), the ID of the business component (the extension `id` value), and the name of the business component's implementation class. These three attributes are shown in **bold** in this example listing from the sample portal.xml file:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <plugin id="com.ibm.wps.shop.app" version="1.0.0">
      <extension
          point="com_ibm_portal_app.BusinessComponents“
   ```

```
        id="Catalog">
         <provider class="com.ibm.wps.app.bc.impl.CatalogImpl">
         </provider>
   </extension>
</plugin>
```

Important: The value for the extension's `point` attribute must always be `com_ibm_portal_app.BusinessComponents`; otherwise the component will not be found. If a corresponding plugin.xml file is added to the classpath, the extension will automatically get added to the extension point.

2. Add the binaries and the plugin.xml to the classpath of the system.
   While you could do this any way you want, we recommend that you package the component's classes or JAR and the plugin.xml in the corresponding portlet's WAR file. Then, you can easily deploy, update, and remove the code; it is automatically distributed in the cluster; you can use the existing staging mechanisms; and so on.

Lookup of a business component
Using the extension registry directly can be complicated. WebSphere Portal V6.0 lets you lookup business components registered as extensions using JNDI.

You need to get an instance of the component for the portlet that provides the view for your catalog; so, the portlet needs to create the correct name for the lookup. It consists of the prefix `portal:extreg/` , followed by the namespace of your component (which is the plugin ID in the XML), a dot `"."` , and the ID of your component (which is the extension ID in the XML).

For example, the JNDI name for the `catalog` component would look like this:

```
portal:extreg/com.ibm.wps.shop.app.Catalog
```

Create a portlet for the catalog business component

The purpose of this portlet is to demonstrate how to use your new business component together with a corresponding user interface component. You decide to limit the complexity of the portlet to a very simple UI.

You start by thinking about what to display in each of the modes. Here is a list of things the sample portlet will display in the different modes:

- View mode:
  - Display some information about the runtime to better understand how things work
  - Display a list of all categories and items of your catalog
  - Place items in a virtual shopping cart
  - Display the contents of your shopping cart
- Edit mode:
  - Edit the catalog

- Config mode:
    - Modify some display properties

To display some information about the runtime, you need to get access to the new catalog service. As shown earlier, the business components work by leveraging the memento pattern. There is one service instance, and you need to pass it the correct business component ID to get access to your data. This ID is placed into the portlet preferences by the application infrastructure when the component is created and connected to the portlet − either when the portlet referencing the business component is added to an existing application or during template instantiation.
The name of this preference is defined to be `com.ibm.portal.bc.instance.id`. So the method to fetch it looks like this:

```
private String getInstanceId(PortletRequest request) {
    return request.getPreferences().getValue(
            "com.ibm.portal.bc.instance.id","null");
}
```

You can fetch the value just as easily from inside a JSP. The example code leverages both options.

You also need to get access to the catalog service. Use the JNDI lookup and the construction of the name that was discussed in the last section. The lookup code looks like this:

```
private Catalog getCatalog() throws NamingException {
    Context context = new InitialContext();
    Object component = context.lookup(
            "portal:extreg/com.ibm.wps.shop.app.Catalog");
    if (component instanceof Catalog) {
        return (Catalog)component;
    }
    return null;
}
```

Important: You can easily run into a pitfall here if you try this in a more complex setup. The `instanceof` check and the following cast will only work if the `Catalog` interface was loaded from the same classloader as used by the business component. Otherwise, the two classes are incompatible. In the example case, you deploy everything in a single WAR, therefore you are fine. If you tried to do this from a different WAR file, which contains a copy of the interface class, it would fail. In case you need those setups, you should use an EAR file, place the interface on the EAR level, and use it from the different web modules. Consult the InfoCenter for details on how to deploy an EAR file in WebSphere Portal.

In the `render` method of the portlet, you need to store the catalog object as a request attribute.

```
request.setAttribute("catalog", getCatalog());
```

Now that you have made all required data available, you only need to do a redirect call to the JSP. The final code looks like this:

```
private static final PortletMode CONFIG_MODE = new
PortletMode("config");

public void render(RenderRequest request, RenderResponse response)
    throws PortletException, java.io.IOException {
  WindowState state = request.getWindowState();

  if (!state.equals(WindowState.MINIMIZED)) {
    PortletMode mode = request.getPortletMode();

    response.setContentType("text/html");
    PortletRequestDispatcher dispatcher = null;
    try {
      if (mode.equals(PortletMode.VIEW)) {
        request.setAttribute("catalog", getCatalog());
        dispatcher = getPortletContext().getRequestDispatcher(
                     "/WEB-INF/view/view.jsp");
      } else if (mode.equals(PortletMode.EDIT)) {
        request.setAttribute("catalog", getCatalog());
        dispatcher = getPortletContext().getRequestDispatcher(
                     "/WEB-INF/view/edit.jsp");
      } else if (mode.equals(CONFIG_MODE)) {
        dispatcher = getPortletContext().getRequestDispatcher(
                     "/WEB-INF/view/config.jsp");
      }
    } catch (NamingException e) {
      e.printStackTrace(response.getWriter());
    }
    if (dispatcher != null) {
      dispatcher.include(request, response);
    }
  }
}
```

This code dispatches to a special JSP for each mode. For the view and edit modes, you store the catalog object as an attribute and redirect to the appropriate JSP (`view.jsp` and `edit.jsp`). When the portlet is in config mode, you simply redirect to the JSP (`config.jsp`), without forwarding any data.

Let's take a look at the `view.jsp`. First, you get the attribute and retrieve the `instanceId` from the portlet preferences. This is done by the following fragment:

```
<%@ page session="false" %>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
<%@ page import="java.util.Enumeration" %>
<%@ page import="java.util.Iterator" %>
<%@ page import="javax.portlet.PortletMode" %>
<%@ page import="com.ibm.wps.app.bc.Catalog" %>

<portlet:defineObjects/>
```

```
<% Catalog catalog    = (Catalog)request.getAttribute("catalog");
%>
<% String  instanceId =
renderRequest.getPreferences().getValue("com.ibm.portal.bc.instance.id"
, "");%>
```

You invoke the `defineObjects` TAG to get access to the `renderRequest` object, which lets you access the preferences.

Next, you write out some debug information:

```
<br>font_size  = <%=
renderRequest.getPreferences().getValue("font_size" , "") %>
<br>font_color = <%=
renderRequest.getPreferences().getValue("font_color", "") %>
<br><font size="<%= renderRequest.getPreferences().getValue("font_size"
, "-1") %>"
          color="<%=
renderRequest.getPreferences().getValue("font_color" , "#000000")
%>">SOME DEBUG INFO:
<br>id = <%=
renderRequest.getPreferences().getValue("com.ibm.portal.bc.ref","null")
%>
<br>instanceId = <%= instanceId %>
<br>catalog = <%= catalog %>
```

Important: The above example is vulnerable for active content insertion attacks. You can prevent that by using the JSTL out tag.

This code first fetches two additional preferences, which control the font. You can use these preferences as a simple example to demonstrate the use of PoVs and how they can modify the UI. Start by simply printing out `font_size` and `font_color`. Then, change the font to use those values and then display the business component reference, your `instanceId`, and the `catalog` object.

The business component reference (`com.ibm.portal.bc.ref`) is a special preference and is defined in the API for the composite application infrastructure. You set it in the portlet.xml. The value is a JNDI name of the business component that will be connected with the portlet. Then, the infrastructure can instantiate an instance whenever the portlet is added to an application. Once you generate a template for your application, this preference is not required anymore because the value from the template will be used. The output of your JSP so far looks like this:

Figure 3. Displaying some debug information

The most important parts in the output are the instanceId and the catalog; both of which are set. Now, you have all the data you need to write the real user interface! This part is nothing special and works just like writing a normal portlet. Therefore, writing a view for a business component is as easy as writing a simple portlet.

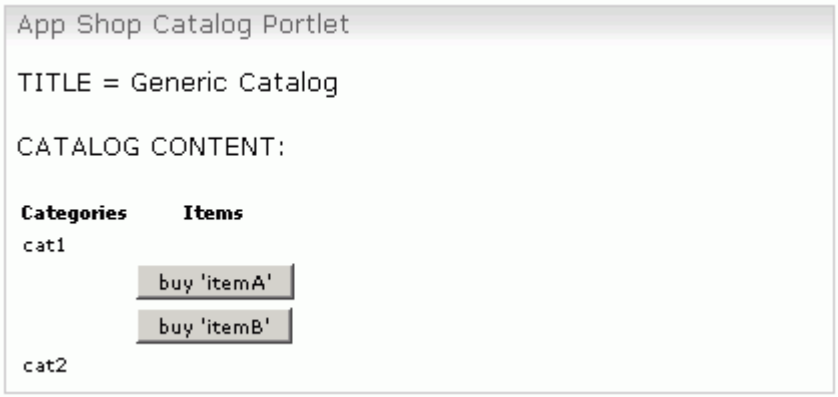
In the view mode, display the contents of your catalog and let the user buy items. Your simple UI for this functionality creates a small table with the categories and items in those categories. You fetch the information from the `catalog` business component. Every item has a small button which the user clicks to add to the shopping cart. The button invokes a portlet action that simply adds the item to the session. Take a look at the JSP code:

```
<portlet:actionURL var="actionURI"/>

<br><br><br>TITLE = <%= catalog.getTitle(instanceId) %>

<br><br>CATALOG CONTENT: <br><br>
<table>
  <tr>
    <th>
      <b>Categories</b>
    </th>
    <th>
      <b>Items</b>
    </th>
  </tr>
<% for (Iterator categoryIter =
catalog.getCategories(instanceId).iterator(); categoryIter.hasNext();)
{ %>
<%     String category = (String)categoryIter.next();
%>
  <tr>
      <td>
        <%=category%>
      </td>
  </tr>
```

```
    <% for (Iterator itemIter = catalog.getItems(instanceId,
category).iterator(); itemIter.hasNext();) { %>
    <%     String item = (String)itemIter.next();
%>
  <tr>
    <td></td>
    <td align="center">
      <form action="<%=actionURI%>" method="POST">
        <input type="hidden" name="buyItemCategory"
value="<%=category%>">
        <input type="hidden" name="buyItemItem"     value="<%=item%>">
        <input type="submit" value="buy '<%=item%>'">
      </form>
    </td>
  </tr>
      <% }
%>
<% }
%>
</table>
```

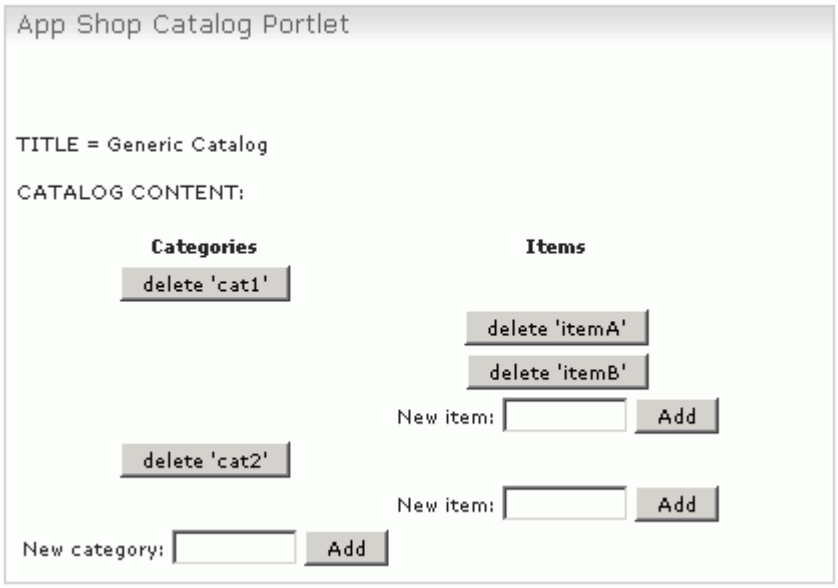
This code produces a page like this:

Figure 4. The view mode for the portlet, allowing to buy items



Create your edit view similarly. Instead of writing out the category and item names, you use buttons (similar to the above JSP fragment) to remove the entries. After that, you only need to add some more forms to create categories and items. The final result could look like this:

Figure 5. Edit mode, which enables management of the catalog

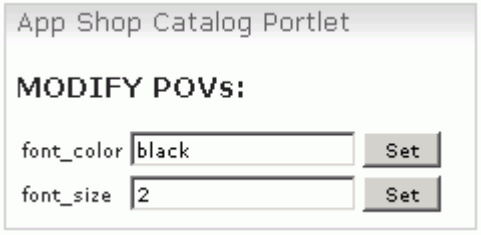


This is only a simple example UI. For a production level implementation, you would want a UI designer to create something more elegant, which is usable for large catalogs.

All forms use action URLs and result in a `processAction` call on your portlet. The code for those is straight-forward. Basically all actions map directly to the API of your catalog business component. The only exception is the **Buy** button, which is handled by placing the item into the portlet session. Your `view.jsp` can then simply list the session content to display a shopping cart.

The last part to create is the `config` mode. You want to enable the admin of your application to modify the UI by using the two font attributes described earlier. The required code is similar to what you have written so far. Because the data is stored in portlet preferences you can easily read it from the JSP code. The portlet code stores the new value during the action processing.

Figure 6. Config mode display



Step 2: Assemble the application

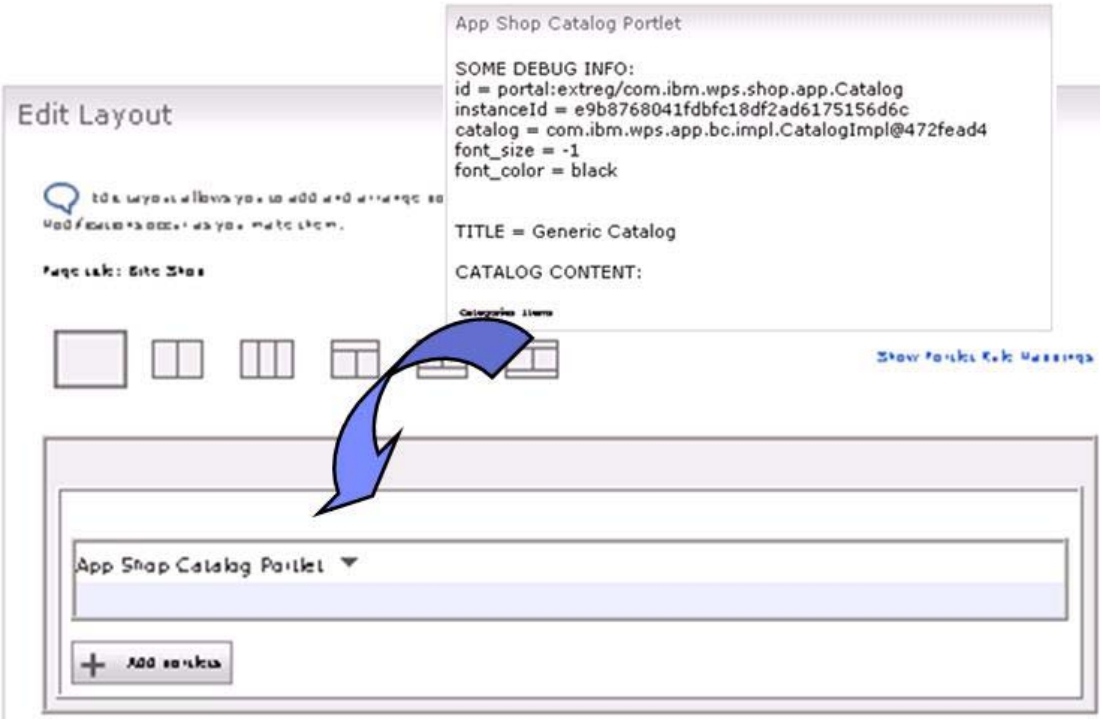
After all the necessary components are ready, the application designer of Useful Templates Inc. can start the assembly step to build the application template for the online shop.

Tools

For this purpose, the designer (you, for now) can use the Template Builder, an intuitive portlet that comes with WebSphere Portal. You do not need to have any programming skills to use the Template Builder. Its graphical user interface is very similar to the page customizer. During the application assembly, you can create pages and add portlets to a template, exactly the same way that you would define classical portal page content.

The Builder is not a separate development tool; it is part of the regular WebSphere Portal V6 installation. You can access it through the template administration pages. However, the pages of the template will not be part of the regular portal content tree. Instead, the application designer works in a separate, temporary Workspace. If you want to use components, such as portlets, you need to make sure that these are deployed onto the WebSphere Portal installation first. Similar to page customizer, the Template Builder can only deal with properly deployed artifacts.

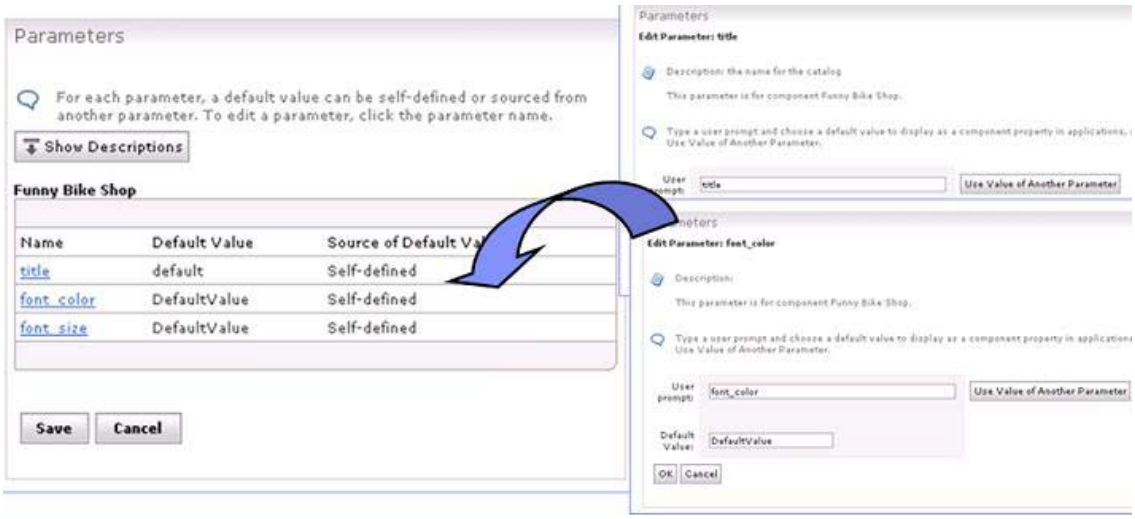
Figure 7. Builder screenshots



Work with parameters

After you have turned your application into a template, you can fine tune the template's parameters. Click **Edit Template Parameters** in the context menu for your template in the Template Library you can get to the Parameters portlet. Here you can select the parameters and modify the prompt information or the default values for your template's Parameters.

Figure 8. Parameter screenshots



Add wires

If you want to avoid redundancy of your parameters or you want values to be automatically filled into appropriate fields, you can wire parameters together. Use the Parameters portlet.

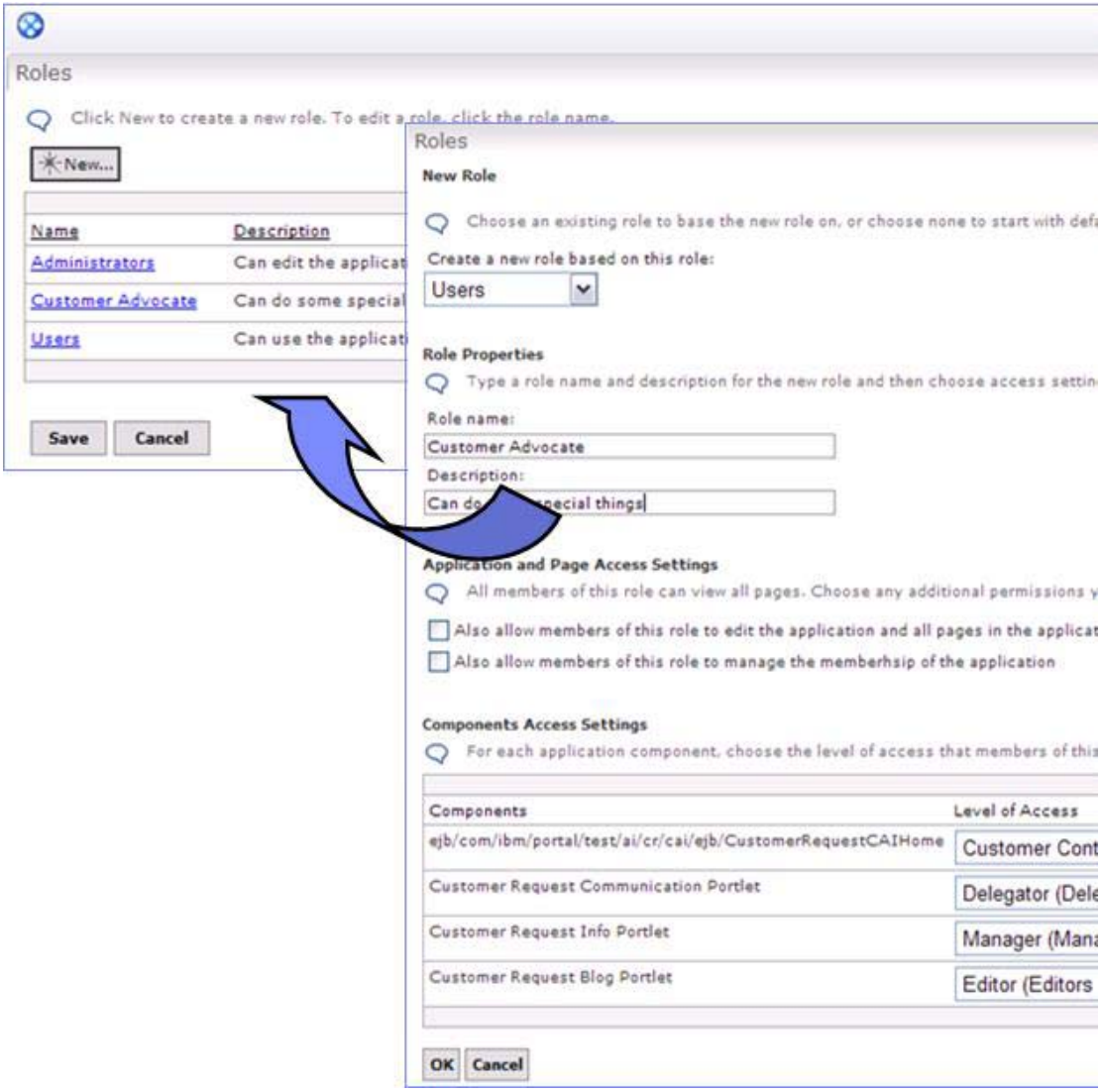
1. Select the parameter that should get its value from another parameter.
2. In the Edit Parameter panel, select **Use Value of another Parameter** .
3. Select the corresponding parameter from which the value should be passed into your parameter.

Add application roles

You can define access control settings through templates very efficiently. Rather than exposing all the very detailed and fine grained access control settings for each individual resource, you can group related access control roles to larger "application level" roles, which are meaningful in the context of this particular composite application. This feature hides the complexity of too many access control roles. Instead, your application can expose a few application level roles with well-understood and easy to use names.
For example, your online shop template would have the application roles "shop owner", "customer", and "guest". Each of these roles aggregates a set of specific, fine-grained portal access control roles. A "shop owner" role will have manager access to the catalog

portlet to be able to change the article description. This role will also have manager access to the FAQ and the Newsgroup portlets. A “guest” role will have no access to the catalog, but reader access to the Newsgroup and FAQ portlets. A “customer” will be a user of all 3 portlets.

Figure 9. Application roles screenshots



Step 3: Deploy the templates and application

Let's asume the application designer of Useful Templates Inc. has completed the development work. The Template Builder tool translates the assembled content from the designer's temporary workspace into a XML file which gets stored in the template catalog.

Now the template is available within the WebSphere Portal installation of Useful Templates Inc. To deliver the composite application to the Funny Bikes dealer, you need

to export the XML description of the template. The result is an XML file, which can be easily transferred.

In addition to the template, you must provide the components that are to used as well. In the example, Useful Templates Inc. has implemented the custom catalog, FAQ, and newsgroup portlets, as well as the business component. Typically, the dealer will receive these components as standard J2EE WAR files which can be exported by the development tools that the component developers used to implement them.
On the dealer side, the WebSphere Portal administrator of Funny Bikes imports the WAR file into his WebSphere Portal production system, and then imports the template XML file into the template catalog. The template is now ready for instantiation.

For staging to production the transfer and deployment of the WAR files and the transfer of the template(s) can be handled in bulk through the portal staging tools. For further details, see WebSphere Portal 6.0 InfoCenter.

Step 4: Create and manage application instances

At this point of time the business user gets involved. Harry, from the Funny Bikes sales team, picks the template from the template catalog and create a real instance of the Funny Bikes online shop on the production system. During the instantiation process, WebSphere Portal creates the necessary pages and portlet instances exactly as specified in the template. Harry is prompted to enter values for the parameters so that he can customize this particular instance of the generic application template in a very unique way.
The runtime environment for the newly created application is the “composite application infrastructure”. This infrastructure offers Harry a set of very powerful options:

- The template includes the specification of application level roles. Now the time has come to map actual users or groups to these predefined roles. Harry reviews the list of registered users and categorizes them into “shop owner”, “customer”, and “guest”. He does not need to deal with the low level WebSphere Portal access control at all. And, he does not need the help of the system administrator to do this. He can manage the *membership* of users to his application instance on his own!

- Harry has some really good ideas to improve the online shop. The runtime offers pretty much the same capabilities as the Template Builder tool. He starts editing the page layout of his application instance. He adds a document viewer portlet which displays a document library featuring recent test reviews of bicycles. He defines another application role called “gold customer” and he specifies that only gold customers can access the document library. All these changes apply directly to his application instance; they do not change the generic template of Useful Templates Inc.! If Harry wants to share his improvements with others, he can save his changed application instance as a new template and then that new template can be stored in the template catalog as well.

Figure 10. Template instantiation

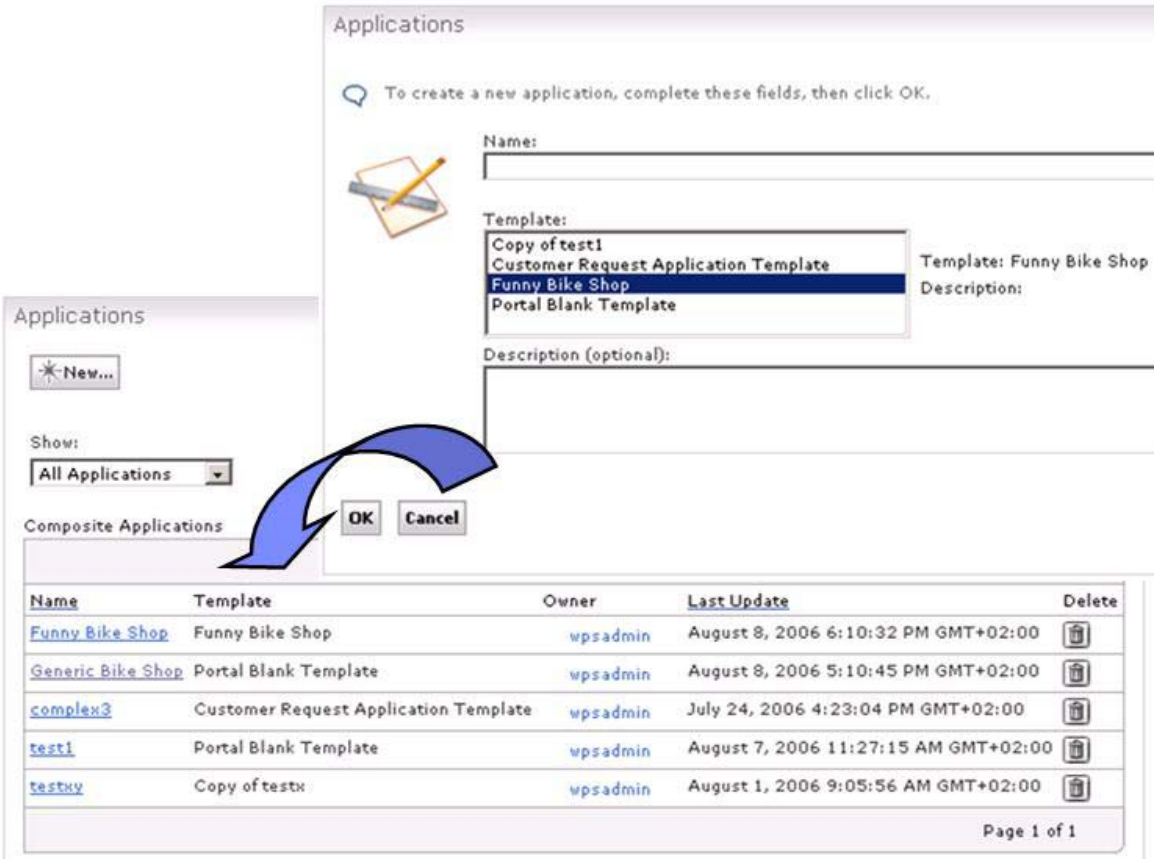
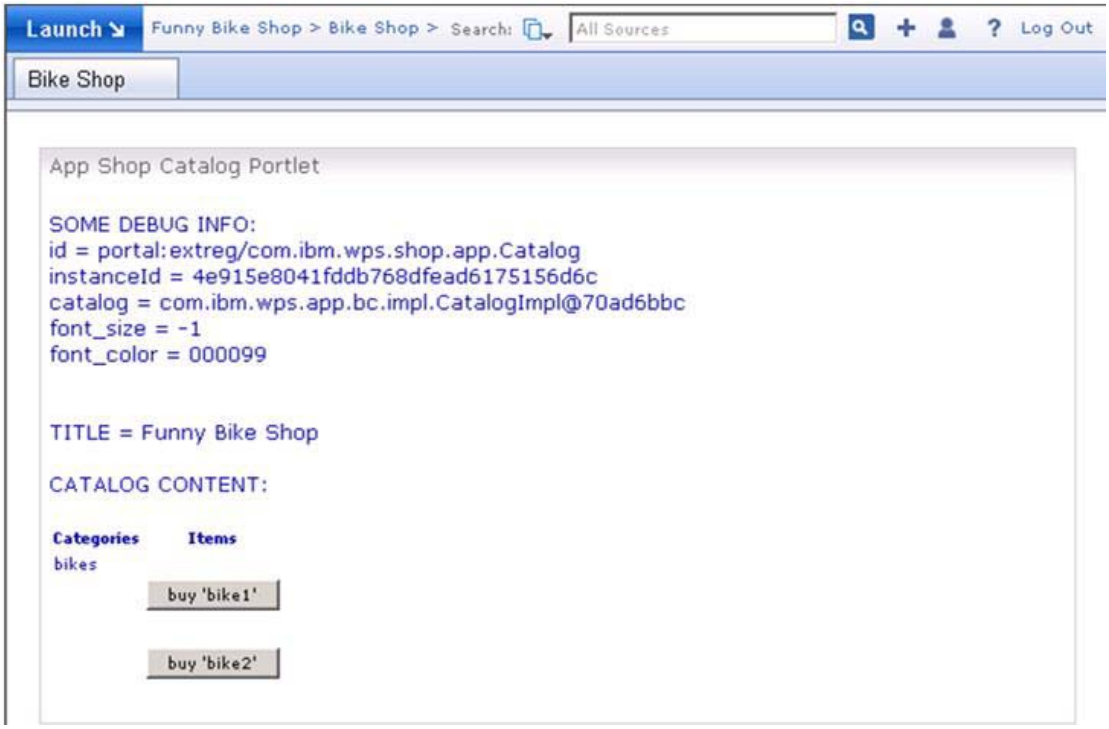


Figure 11. Funny Bike Shop application



Mary is another member of the sales team of Funny Bikes. Her job is to establish a new brand of an online shop which will focus exclusively on mountain biking equipment. For her online sales, she decides to take Harry's updated template as a basis. All she needs to do is to pick Harry's template from the catalog and to create a new instance for her own shop. WebSphere Portal creates the necessary pages and portlet instances for her. Mary is prompted to specify values for the parameters used by the template.

The two application instances reside in parallel and operate independently of each other. An administration portlet can be used to manage all instances which have been created on this production WebSphere Portal system.

Conclusion

WebSphere Portal V6 provides enhancements in many technical areas to help your company improve productivity, accelerate application and content deployment, and to increase responsiveness and reliability. One of the major features it provides is composite applications which enable end users to translate their knowledge and skills into business value, by assembling complex applications out of a set of components.

This article provides a starting point to understand how templates and applications play together. You learned about the key features which are provided by the composite application infrastructure. These features include:

- Parameterization of components, by defining points of variability within template.
- Application level roles, which aggregate fine-grained portal access control roles to simplified roles representing application specific semantics.
- Membership, which lets the owner of an application instance assign individuals to application level roles.
- A set of public APIs, which component developers can use to leverage the capabilities of the application infrastructure.

To keep learning, see the WebSphere Portal V6 InfoCenter.

Download

Get the download from the cover page for this article:
http://www.ibm.com/developerworks/websphere/library/techarticles/0608_stober/0608_stober.html

Resources

IBM WebSphere Portal V6 announcment
http://www.ibm.com/isource/cgi-bin/goto?it=usa_annred&on=206-163

Introducing the Java Content Repository API
http://www.ibm.com/developerworks/java/library/j-jcr/

Introducing the WebSphere Portal V6.0 Content API
http://www.ibm.com/developerworks/websphere/techjournal/0607_kubik/0607_kubik.html

Java Content Repository (JCR), JSR 170
http://www.jcp.org/en/jsr/detail?id=170

WebSphere Portal catalog
http://www.ibm.com/websphere/portal/catalog

WebSphere Portal V6 demo
https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=sw-pprod05&S_PKG=SW-yourworldyourwaydemo&S_TACT=105AGX10&S_CMP=WPZN

WebSphere Portal product documentation, including the InfoCenter
http://www.ibm.com/developerworks/websphere/zones/portal/proddoc.html

WebSphere Portal V6 product information
http://www.ibm.com/websphere/portal

WebSphere Portal zone
http://www.ibm.com/developerworks/websphere/zones/portal

What's new in WebSphere Portal V6?
http://www.ibm.com/developerworks/websphere/library/techarticles/0607_hepper/0607_hepper.html

About the authors

Dr. Thomas Stober is release architect for WebSphere Portal and owns the technical responsibility for the new Portal 6.0 release. Thomas is also a key player in IBM's composite application effort. In the past, Thomas has been focusing on virtual portal, software componentization as well as on mobile computing and data synchronization. Thomas is co-author of the book *Pervasive Computing Handbook.*

Peter Fischer is the lead for the application infrastructure in the WebSphere Portal foundation, and is responsible for composite applications. Previously he was the lead on the ISC 6.0 effort in the foundation and the implementation of the WSRP standard in portal. He also has been part of the development team of WebSphere Application Server for zSeries Peter is co-author of book *Portlets and Apache Portals*.

Hendrik Haddorp works on the application infrastructure in WebSphere Portal. He is responsible for the implementation and maintenance of the templating and instantiation component. Since joining WebSphere Portal in 2003, he has gained experience in many different areas of the WebSphere Portal foundation and in portlet deployment. He also worked on the relocation of the portlet container from WebSphere Portal into WebSphere Application Server.

- Other company, product, and service names may be trademarks or service marks of others.

IBM copyright and trademark information: http://www.ibm.com/legal/copytrade.phtml