



SAS Publishing



SAS[®] Web Infrastructure Kit 1.0

Developer's Guide

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2004. *SAS® Web Infrastructure Kit 1.0: Developer's Guide*. Cary, NC: SAS Institute Inc.

SAS Web Infrastructure Kit 1.0: Developer's Guide

Copyright © 2002-2004, SAS Institute Inc., Cary, NC, USA

All rights reserved. Produced in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

April 2004

SAS Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at support.sas.com/pubs or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

Table of Contents

SAS® Web Infrastructure Kit 1.0: Developer's Guide.....	1
Developing Custom Portlets.....	2
Integrating Web Applications With the Portal.....	4
Tasks.....	6
Creating a Portlet Deployment Descriptor.....	7
Example Deployment Descriptor for a Local Portlet.....	8
Example Deployment Descriptor for a Remote Portlet.....	9
Detailed Element Descriptions for Portlet Deployment Descriptor DTD.....	10
Creating Display Resources Files.....	11
Developing the Presentation JSP.....	12
Creating Action Classes.....	13
Creating an Initializer Action Class.....	14
Creating a Portlet Action Class.....	15
Creating a Postprocessing Action Class.....	16
Creating an Error Handling Action.....	17
Creating a PAR File for Deployment in Your Application.....	19
Use Cases.....	20
Creating a Simple Display Portlet.....	21
Creating a Localized Portlet.....	24
Creating a Portlet Template (Editable Portlet).....	27
Creating a Remote Portlet.....	30
Samples.....	32
Sample: Localized Display Portlet (Welcome Portlet).....	33

Table of Contents

Step 1: Create the Directory Structure.....	34
Step 2: Create the Portlet Deployment Descriptor.....	36
Step 3: Create the Display Page.....	38
Step 4: Create the Action Class.....	39
Step 5: Create the Resource Bundles.....	41
Step 6: Create Translated Titles and Descriptions.....	42
Step 7: Create the PAR File, and Deploy and Test the Portlet.....	43
Sample: Interactive Form Portlet (FormExample).....	44
Step 1: Create the Directory Structure.....	45
Step 2: Create the Portlet Deployment Descriptor.....	47
Step 3: Create the Display Page.....	49
Step 4: Create the Action Class.....	50
Step 5: Create the JavaBean.....	52
Step 6: Create the PAR File, and Deploy and Test the Portlet.....	53
Sample: Portlet Template, or Editable Portlet (DisplayURL).....	54
Step 1: Create the Directory Structure.....	55
Step 2: Create the Portlet Deployment Descriptor.....	57
Step 3: Create the Display Pages for the Portlet and the Editor.....	59
Viewer.jsp.....	60
Editor.jsp.....	61
Error.jsp.....	63
Step 4: Create the Action Classes.....	64
Initializer Action.....	65

Table of Contents

Base Action.....	67
Display Action.....	69
Editor Action.....	70
OK and Cancel Actions.....	72
Error Handler Action.....	74
Step 5: Create the Resource Bundle.....	76
Step 6: Create the Display Resources File.....	77
Step 7: Create the PAR File, and Deploy and Test the Portlet.....	78
Sample: Web Application (HelloUserWikExample).....	79
Step 1: Create the Directory Structure.....	80
Step 2: Create the Web Application Deployment Descriptor.....	81
Step 3: Create Deployment Definitions and Properties Files for Local and Remote Services.....	83
Step 4: Create the Display Page (JSP).....	86
Step 5: Create the WAR File, and Deploy and Test the Application.....	88
Sample: Remote Portlet (HelloUserRemotePortlet).....	89
Step 1: Create the Directory Structure.....	90
Step 2: Create the Portlet Deployment Descriptor.....	91
Step 3: Create the Display Resources File.....	92
Step 4: Create the Web Application.....	93
Step 5: Create the PAR File, and Deploy and Test the Portlet.....	94
Themes.....	95
Defining and Deploying New Themes.....	96
Styles in Portal.css.....	99

Table of Contents

ExampleTheme.xml File.....	102
Element Descriptions for Themes DTD.....	105
Changing the Application Name.....	106
Resources.....	107
Using the Portlet API.....	108
Using SAS Foundation Services With the Portal.....	110

SAS® Web Infrastructure Kit 1.0: Developer's Guide

The SAS Web Infrastructure Kit: Developer's Guide provides information to help you use the SAS Web Infrastructure Kit to develop your own custom applications. It also explains how to use the SAS Web Infrastructure Kit to customize and extend the SAS Information Delivery Portal 2.0 to meet the unique requirements of your organization.

Note: In this guide, "portal Web application" is a generic term that refers to either of the following:

- the SAS Portal Web Application Shell, which is a portal-like Web application shell that is included in the SAS Web Infrastructure Kit and is used by other SAS Web applications, or
- the SAS Information Delivery Portal, which (when installed with the SAS Web Infrastructure Kit) fully implements the capabilities of the SAS Portal Web Application Shell.

The guide includes the following chapters:

- **Tasks** provides step-by-step instructions for performing important application development tasks, which include creating deployment descriptor files for portlets, creating resource files for portlet metadata, developing JavaServer Page (JSP) pages for portlets, creating custom action classes for portlets, and creating portlet archive (PAR) files to use in deploying portlets in the portal Web application.
- **Use Cases** provides development steps and best practices for common use cases. The use cases include creation of simple display portlets, localized portlets, portlet templates (also called editable portlets), and remote portlets (Web applications that are deployed and executed outside of the portal Web application).
- **Samples** provides fully developed samples of custom portlets, including all of the portlet deployment descriptors, JSPs, resource files, and action classes that are required to implement each portlet. The samples include a localized display portlet, an interactive form portlet, a Web application, a remote portlet, and a portlet template (editable portlet).
- **Themes** describes how to customize the appearance of the portal Web application by developing new themes, including text attributes, backgrounds, logos, and other graphical elements. This section also describes how to change the application name that appears in the banner.
- **Resources** describes the application programming interfaces (APIs) that are provided with the SAS Web Infrastructure Kit, including the Portlet API and the SAS Foundation Services Facade. Links to the detailed class documentation are provided.

For details about administrative tasks that are required to support development activities, including the deployment of new portlets, themes, and content items, refer to the [SAS Web Infrastructure Kit Administrator's Guide](#).

Developing Custom Portlets

A portlet is a Web component that is managed by a Web application and aggregated with other portlets to form a page within the application. A portlet processes requests from the user and generates dynamic content such as report lists, alerts, workflow notifications, or performance metrics. The components that are needed to implement a portlet may include JSP pages, custom classes, and associated resources.

Portlets that are created with the SAS Web Infrastructure Kit have a standard appearance, which includes a title bar that contains links or icons to portlet actions, as shown in this example:

Organization	Status	Trend	Performance Rating	Contact
Finance	▲▲	▲▲	<div></div>	Jane Dollar
Operations	▲▲	▲▲	<div></div>	Susan Day
Sales	▲	▲	<div></div>	Sam Sellers
Marketing	▲	▲▲	<div></div>	Brandy Hip
Manufacturing	▲	▲	<div></div>	May Kerr
RnD	▲	▲	<div></div>	Dr. Brian Sm

The framework of the SAS Web Infrastructure Kit makes it easy for you to quickly develop and deploy custom portlets that meet your organization's needs. This framework, which is based on the Struts architecture and conforms to industry-standard Model-Viewer-Controller (Model 2) design patterns, provides

- **an execution environment** that allows portlets to execute in the portlet container, in the same way that servlets execute in the servlet container. The portal Web application processes all HTTP requests for portlets, while the portal Web application's session and state information are maintained and shared among portlet actions and across requests.
- **support for portlets running remotely** in other Web technology frameworks, with the option to pass the portal Web application's session and state information to these portlets.
- **simplified portlet deployment** through the use of the following:
 - ◆ **a portlet deployment descriptor**, which is an XML file that specifies the portlet's actions as well as initialization, path, and access control information.
 - ◆ **a portlet archive (PAR) file**, which includes all of the elements needed to deploy a portlet or series of portlets, including the portlet deployment descriptor, JSP pages, custom Java classes, and associated resources (such as images, resource bundles, HTML files, and style sheets).
- **a set of action and initializer classes**, which reduce the need for developing custom programs. These classes perform the most commonly used functions, such as displaying the JSP page that is specified in the portlet deployment descriptor.
- **access to SAS custom tags and to tags in the Struts development framework** to simplify development of JSP pages for your portlets.
- **a dynamic (or "hot") deployment mechanism** that enables new portlets to be deployed without the need to restart the Web server.

Options for Implementing Portlets

The action and initializer classes included in the SAS Web Infrastructure Kit are designed to handle a portlet's basic function of displaying a single JSP page. However, to meet specialized needs you can

- write one or more Java classes that implement the `com.sas.portal.portlet.PortletActionInterface`. Alternatively, you can extend `com.sas.portal.portlet.HTMLPortletAction` to obtain a basic implementation of the interface.
- write Java classes that implement the `PortletInitializerInterface`, `ErrorHandlerInterface`, or `PostProcessorInterface` in the `com.sas.portal.portlet` package in order to meet more specialized requirements.

For more information, see the [Portlet API](#) and the [sample portlets](#).

Best Practices

The following is a summary of best practices for developing portlets for deployment in the portal Web application:

- To avoid namespace problems, use a standard naming convention for portlet paths. The portlet namespace is comprised of the path (with leading underscores in place of slashes) and the portlet's name. For example, a portlet with the name `simpleJSP` and a path of `/sas/portlets` would be deployed as `_sas_portlets_simpleJSP`.
- Deploy new portlets into a staging area (that is, a test installation of the portal Web application) for verification and testing before deploying them into the production environment.
- For remote portlets, use the portlet's direct URL to test and debug the portlet before you deploy it into the portal Web application.

Integrating Web Applications With the Portal

The SAS Web Infrastructure Kit enables you to easily integrate other applications with the portal Web application. To make a Web application available in the portal Web application, you can

- **implement a remote portlet and a corresponding Web application.** A remote portlet looks like any other portlet, but it calls a remote Web application. The remote Web application returns an HTML fragment to the portal Web application to be displayed within the portlet's borders. This approach is useful when you want to incorporate a portion of the output from your application into the portal Web application.
- **implement a stand-alone application that is invoked from the portal Web application but executed remotely.** The stand-alone Web application returns a complete HTML page that is displayed in a separate browser window. This approach is useful when you want to enable users to invoke your application from the portal Web application, but the application output needs to appear separately.

Using SAS Foundation Services to Integrate Applications and Enable Single Signon

Whether your application is called by a remote portlet or is invoked on a stand-alone basis, the SAS Web Infrastructure Kit provides tools to facilitate secure information sharing between the portal Web application and the remote application. One type of information sharing is the single signon feature, which enables other applications to be invoked from the portal Web application without requiring the user to re-enter a user name and password. Other information related to a portal Web application session can be shared as well.

To incorporate the single signon feature or other information sharing into a remote portlet application or a stand-alone Web application, you must

- use classes from SAS Foundation Services in your Web application. SAS Foundation Services is a set of infrastructure and extension services that support the development of integrated, scalable, and secure Java-based applications. For convenient access to the most common access patterns, you will probably want to use the foundation services facade classes, which are part of the [Portlet API](#). For more information, see [Using SAS Foundation Services with the Portal](#).
- use these classes to access SAS Foundation Services that have been deployed remotely. The SAS Services application (SASServices), which is provided with the SAS Web Infrastructure Kit, provides the remote service deployment. This application, which can be installed anywhere on your network, provides a secure mechanism for the portal Web application to share information with remote Web applications.

For more information about developing Web applications that are enabled by SAS Foundation Services, see:

- [Sample Web Application \(HelloUserWikExample\)](#)
- [Using SAS Foundation Services with the Portal](#)

For more information about SAS Foundation Services, see [SAS Foundation Services](#) in the *SAS Integration Technologies Developer's Guide*.

Making Web Applications Available in the Portal Web Application

If your application is to be called by a remote portlet, you must create a portlet deployment descriptor for the portlet and package it in a portal archive (PAR) file. When you deploy the portlet, its metadata is registered automatically with the portal Web application. For more information, see:

- [Creating a Remote Portlet](#)
- [Sample Remote Portlet \(HelloUserRemotePortlet\)](#)
- [Deploying Portlets](#) in the SAS Web Infrastructure Kit Administrator's Guide.

If you want to add a stand-alone application to the portal Web application, then you must use a SAS program to register the application's metadata. For instructions, see [Adding Applications](#) in the *SAS Web Infrastructure Kit Administrator's Guide*.

Tasks

Tasks

This chapter provides step-by-step instructions for tasks that developers can or must perform when creating custom portlets. The tasks include:

- Creating a portlet deployment descriptor (`portlet.xml`). Each portlet that you deploy must be defined in a portlet deployment descriptor. A portlet deployment descriptor is an XML file that provides all of the information that the portal Web application requires in order to deploy one or more portlets. The file includes information about the portlet's initialization, actions, security settings, and resource paths.
- Creating display resources files (`portletDisplayResources_xx.properties`). The display resources file contains text strings for the portlet's title and description for use in the portlet's metadata. If you create multiple display resources files for different locales, the portal Web application uses these files to localize the portlet title and description at the time of deployment, according to the portal Web application's default locale.
- Developing the presentation JavaServer Pages (JSP). Each portlet must have a JSP page to serve as the presentation component.
- Creating action classes. You can use the resources of the Portlet Development Kit to develop the following types of action classes for your portlets: initializer classes, portlet actions, postprocessing classes, and error handling classes.
- Creating a portlet archive (PAR) file for deployment in your application. To enable automatic deployment of a portlet into the portal Web application, you must provide a PAR file which contains all of the needed files. A PAR file can contain files for one portlet or for multiple related portlets.

For practical applications of these tasks, see Use Cases. For examples of fully developed portlet code, see the Samples.

Tasks

Creating a Portlet Deployment Descriptor

For each portal archive (PAR) file that you create for deployment in the portal Web application, you must create a portlet deployment descriptor. The portlet deployment descriptor is an XML file that provides all of the information that the portal Web application needs to deploy the portlets that are contained in the PAR file.

A PAR file, and its associated portlet deployment descriptor, can contain one portlet or it can contain multiple related portlets; there is no limit to the number of portlets that a PAR file and its associated descriptor can contain.

In addition, a PAR file (and its associated portlet deployment descriptor) can contain local portlets, remote portlets, or a combination of local and remote portlets.

To create a portlet deployment descriptor, use element tags as defined in the portlet deployment descriptor document type definition (DTD). The following information is provided for your reference:

- An [example portlet deployment descriptor for a local portlet](#). You can use this example as a template for creating deployment descriptors for your own local portlets. Simply replace the element values in the example with the appropriate values.
- An [example portlet deployment descriptor for a remote portlet](#). You can use this example as a template for creating deployment descriptors for your own remote portlets. Simply replace the element values in the example with the appropriate values.
- A [detailed description of elements](#) in the portlet deployment descriptor DTD.

After creating the deployment descriptor file, assign it the name `portlet.xml`. Then include it in the portal archive (PAR) file that you create for your portlet or group of portlets. For more information, see [Creating a PAR File for Deployment in Your Application](#).

Tasks

Example Deployment Descriptor for a Local Portlet

A local portlet is a portlet that

- is deployed within the portal Web application
- executes inside the portlet container
- consumes the computing resources (for example, CPU, memory, and disk storage) of the server machine on which the portal container runs
- can include resources such as Web pages, style sheets, images, resource bundles, and Java classes which are deployed inside the portal Web application.

An example of a portlet deployment descriptor for a local portlet follows. You can use this example as a template for creating portlet deployment descriptors for your own local portlets. Simply replace the element values with your own values as appropriate.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE portlets SYSTEM "http://www.sas.com/idp/portlet.dtd">
<portlets>
  <local-portlet name="simplejsp" title="SimpleJspPortlet"
    icon="images/ndd.jpg">
    <localized-resources locales="en" />
    <deployment scope="user" autoDeploy="true"
      userCanCreateMore="false">
    </deployment>
    <initializer-type>
      com.sas.portal.portlets.JspPortlet.JspPortletInitializer
    </initializer-type>
    <init-param>
      <param-name>display-page</param-name>
      <param-value>simpleJspTest.jsp</param-value>
    </init-param>
    <portlet-path>/sas/portlets</portlet-path>
    <portlet-actions>
      <portlet-action name="display" default="true">
        <type>com.sas.portal.portlets.JspPortlet.JspPortlet</type>
      </portlet-action>
    </portlet-actions>
  </local-portlet>
</portlets>
```

Tasks

Example Deployment Descriptor for a Remote Portlet

Remote portlets are portlets that execute outside of the portal container. You can use remote portlets to incorporate data from external applications into the portal Web application. When a user interacts with a remote portlet, the remote portlet appears to be the same as a local portlet.

Many of the elements in the portlet deployment descriptor DTD relate only to local portlets. Therefore, a portlet deployment descriptor for a remote portlet requires fewer elements than a descriptor for a local portlet.

An example of a portlet deployment descriptor for a remote portlet follows. You can use this example as a template for creating portlet deployment descriptors for your own remote portlets. Simply replace the element values with your own values as appropriate.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE portlets SYSTEM "http://www.sas.com/idp/portlet.dtd">

<portlets>
  <remote-portlet name="MyRemotePortlet" title="MyRemotePortlet" >
    <localized-resources locales="en" />
    <deployment scope="group" autoDeploy="true"
      userCanCreateMore="false">
      <group>Public</group>
    </deployment>
    <portlet-path>/sas/portlets/remote</portlet-path>
    <portlet-actions>
      <portlet-action name="display" default="true">
        <url>http://d9999.mycompany.com:8080/test.html</url>
      </portlet-action>
    </portlet-actions>
  </remote-portlet>
</portlets>
```

Tasks

Detailed Element Descriptions for Portlet Deployment Descriptor DTD

For detailed descriptions of the elements in `portlet.dtd`, see the following:

- **[Top Elements](#)** contains a list of the top–most elements of the DTD, with links to pages describing each top–most element. From these pages, you can access descriptions of individual child elements.
- **[All Elements](#)** contains a list of all elements defined in the DTD and provides quick access to the description of any individual element.
- **[Top Element Trees](#)** displays the content hierarchy trees of the top–most elements in the DTD, with links to detailed descriptions of the top–most elements and all child elements.

Document generated by  [dtd2html](#) 1.5.1.

portlet DTD*Tasks*

Creating Display Resources Files

A display resources file is a file that contains `<key>=<value>` statements to define text strings for a portlet's title and description. You can create display resources files for the following purposes:

- To specify a description for your portlet. If you do not provide a display resources file, the portal Web application will use the portlet's name to create a default description.

Note: The `local-portlet` and `remote-portlet` elements of the portlet deployment descriptor contain a `description` attribute. However, this description is only for internal documentation purposes. It is not displayed to users.

- To enable the portal Web application to localize the portlet title and description at the time of deployment, according to the portal Web application's default locale. When the portlet is first deployed, the deployment process checks to see which default locale was specified when the SAS Web Infrastructure Kit was installed. Based on this locale, the deployment process uses the title and description from the appropriate display resources file to create metadata and register the portlet in the SAS Metadata Repository.

If your portlet will be deployed in only one locale, then the display resources files can be omitted.

Note: The SAS Metadata Repository cannot store multiple, localized values for metadata. Therefore, the portlet title and description are translated only into the portal Web application's default locale. They cannot be translated based on the user's locale preference.

If your portlet does not include any display resources files, the portlet deployment mechanism will send a warning message to the server log. The message will indicate that no localized title or description can be found.

To create display resources files:

1. Create a separate file for each language (or each country and language combination) that you need to support. In each file, use `<key>=<value>` statements to define text strings for `portlet.title` and `portlet.description`, as in the following examples:

```
portlet.title=Welcome Portlet
portlet.description=Welcome Portlet

portlet.title=Portlet de bienvenida
portlet.description=Portlet de bienvenida
```

2. Name the files as follows:

- ◆ Use the base name `portletDisplayResources.properties`.
- ◆ If you are creating files for multiple locales, append each file's name with the appropriate locale identifier (for example, `portletDisplayResources_en_US.properties` for U.S. English, `portletDisplayResources_fr_CA.properties` for Canadian French, etc.). The file for the default locale does not need to have a locale identifier.

3. Place the files in the `/portletname/classes` directory of the PAR file.

Tasks

Developing the Presentation JSP

JavaServer Page (JSP) pages are the presentation components of local portlets. Because you can define a local portlet's initialization, actions, security settings, and resource paths in the portlet deployment descriptor, the JSP page does not need to contain this information.

In developing the JSP page, you can use the following tags:

- tags from the JSP Standard Tag Libraries (JSTL)
- tags from the Struts tag libraries
- SAS custom tags, which are available with SAS AppDev Studio (WebAF). For information about these tags, see [SAS Custom Tags](#) on the AppDev Studio Developer's Site.

When you create a JSP page for a portlet, the only requirements are the following:

- the JSP page must be an HTML fragment. This means it must
 - ◆ not contain starting and ending <HTML>, <HEAD>, or <BODY> tags
 - ◆ be able to be displayed inside a table cell in an HTML document.
- taglib directive. If the JSP page includes custom tags from a tag library, you must include a taglib directive before the first use of a tag from that library. For the JSTL format tag library, use this taglib directive:

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
```

- UTF-8 directive. You must include a UTF-8 directive if you want the JSP page to provide full support for internationalization. This directive causes all user input to be encoded in the 8-bit Unicode Transformation Format, which supports all of the world's languages, including those that use non-Latin1 characters.

Note: The portal Web application supplies this directive when it displays portlets on a portal page. However, you must supply the directive to ensure correct internationalization when your portlet is displayed from the Search Results panel. You should consider making your portlet actions extend `HTMLPortletAction`, because this class supplies the directive.

The syntax for the UTF-8 directive is:

```
<%@ page contentType= "text/html; charset=UTF-8"%>
```

The SAS Web Infrastructure Kit includes a number of sample JSP pages that you can use as templates for creating your own custom JSP pages. The samples can be found as follows:

1. Go to the `DeployedPortlets` directory of the portal setup directory. For example, if you used the default install location on a Windows system, then you would go to `c:\Program Files\SAS\Web\Portal2.0.1\DeployedPortlets`.
2. Use a utility such as WinZip to open the portlet archive (PAR) file for the portlet whose JSP page you want to access. You can then extract the JSP file from the archive.

Tasks

Creating Action Classes

You can use the resources of the SAS Web Infrastructure Kit to develop the following types of action classes for your local portlets:

- [Initializer classes](#)
- [Portlet action classes](#)
- [Postprocessing classes](#)
- [Error handling classes](#)

The Portlet API includes classes that you can use to create your own action classes for custom portlets. For a summary of these classes, see [Using the Portlet API](#).

For detailed information about the Portlet API, see the [class documentation](#).

Thread Safety

Portlet actions, like Struts actions, are multithreaded. There will be only a single instance of your `PortletAction` subclass, and you must make your actions thread-safe. This means that

- you cannot use class properties to share values between member methods.
- if you use member methods, be sure to pass all values through the method's signature. The signature passes all values through the thread-safe stack.

Tasks

Creating an Initializer Action Class

When you develop a local portlet, you can implement an initializer class that runs before the portlet is displayed for the first time on a portal Web application page. The initializer does not execute again if the user interacts with your portlet or with other portlets on the same page. It also does not execute again if the user navigates to another page and then back again. However, the initializer does run again if the user logs off, logs on again, and displays the page that contains the portlet.

Uses for an initializer might include reading initial parameters that are specified in your portlet's deployment descriptor file (`portlet.xml`), or connecting to an external resource such as a database.

The portal Web application is delivered with a default initializer class called `JspPortletInitializer`, which requires a parameter called `display-page`. The initializer places the value of this parameter in the `PortletContext` object so that it can be used by the portlet's action class. To pass additional parameters, you would need to create your own initializer class.

If you create an initializer class, the class must

- be specified in the `initializer-type` element of the portlet's deployment descriptor file (`portlet.xml`)
- implement `com.sas.portlet.portlet.PortletInitializerInterface`.

The `com.sas.portlet.portlet.PortletInitializerInterface` class includes one method called `initialize()`. The following objects are passed to this method:

- `java.util.Properties`, which contains all of the initial parameters that are specified in your portlet's deployment descriptor. If your portlet's action class or JSP page requires access to these parameters, you should place them in the portlet context object using its `setAttribute()` method.
- `com.sas.portal.portlet.PortletContext`, which provides a getter method for the `HttpSession` object so that you can access or set session attributes.

Here is an example of an `initialize()` method that places initial parameters into the portlet context.

```
/**Puts initial properties into the PortletContext object.
 * These come from the portlet.xml.
 * @param initProperties a Properties object
 * @param context the PortletContext for this portlet
 */
public void initialize(Properties initProperties,
    PortletContext context) {
    context.setAttribute("display-page",
        initProperties.getProperty("display-page"));
    context.setAttribute("image-location",
        initProperties.getProperty("image-location"));
}
```

Tasks

Creating a Portlet Action Class

When developing a local portlet, you can implement one or more action classes for the portlet. The portlet action class

- must be specified in your portlet deployment descriptor file (`portlet.xml`)
- must implement `com.sas.portal.portlet.PortletActionInterface`
- can extend `DefaultPortletAction` or `HTMLPortletAction` in `com.sas.portal.portlet`.

The `DefaultPortletAction` and `HTMLPortletAction` contain two simple methods for setting and getting an instance of `com.sas.portal.portlet.PortletActionInfoInterface`, as shown in this example:

```
public void setInfo(PortletActionInfoInterface pai) {
    _actionInfo = pai;
}
public PortletActionInfoInterface getInfo() {
    return _actionInfo;
}
```

The primary method, called `service()`, runs before the portlet is displayed and every time the portlet is redisplayed. For example, it runs after a user interacts with the portlet or with a different portlet on the same page.

The `service()` method is provided with the `HttpServletRequest`, `HttpServletResponse`, and `PortletContext` objects. From the `PortletContext` object, you can obtain the `HttpSession` object, which provides access to the most important servlet objects.

Your `service()` method must return a string representing a valid URL for the portlet. Typically, the URL is the name of the portlet's JSP page. If your initializer places the `display-page` property of the `portlet.xml` into the `PortletContext`, you can obtain the URL as in this example:

```
String url = (String) context.getAttribute("display-page");
```

If user interaction with your portlet requires a different URL string, you can return that URL instead.

The `service()` method can handle any kind of exception subclass that is thrown by code within your action. If your portlet action needs to throw an exception, you can use the portlet error handler. For more information, see [Error Handling](#).

Tasks

Creating a Postprocessing Action Class

The `com.sas.portal.portlet.PostProcessorInterface` is available for implementing activity that should occur when a local portlet is no longer on display. Like other parts of the portlet architecture, it must be defined in your XML file. You can use the post-processor phase to free resources that you attached to in the portlet initializer. You could also remove `HttpSession` attributes that were set in the initializer or action. This is especially important to consider because multiple copies of your portlet could exist on other portal Web application pages or even on the same page.

Tasks

Creating an Error Handling Action

The `com.sas.portal.portlet.ErrorHandlerInterface` is available for gracefully handling any errors that your local portlets encounter. This interface has one method, which is called `service()`. This method has the same arguments as the `service()` method of the `PortletActionInterface`, plus an additional object called `Exception`.

If you specify an error handler in your portlet deployment descriptor file (`portlet.xml`), the error handler will be called if the portlet action throws an exception. You can direct your error handler to send messages to the server log and to return a URL string representing an error page for the user to view.

If your portlet initializer encounters an exception, the error handler will not be called. If you want to ensure that the error handler will execute, you can store the exception object in the portlet context. Then, in your action class's `service()` method, you can get the exception object out of the context and re-throw it. In the following example, this code is put into a method that should be called at the start of the action's `service()` method:

```
/**Check the PortletContext for an exception object. If
 * present, throw it so that the error handler will kick in.
 * @param context the PortletContext
 */
private static void errorCheck(PortletContext context)
throws Exception {
Exception e = (Exception) context.getAttribute("PORTLET_EXCEPTION");
    if (e != null)
    {
        throw e;
    }
}
```

Here is a simple error handler that logs the exception and calls a static error page. The error page supplies a general error message from the portlet's localized resource bundles.

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.sas.portal.portlet.PortletContext;
import com.sas.services.logging.LoggingServiceInterface;
import com.sas.portal.portlet.ErrorHandlerInterface;
import com.sas.portal.portlet.NavigationUtil;

/**Error handler for some portlets.
 * It logs the exception and returns ErrorPage.jsp
 * for the portlet to display.
 */
public class MyErrorHandler implements ErrorHandlerInterface {

    private final String _loggingContext = this.getClass().getName();

    /**Returns the URL for the portlet controller to call. This is the
     * name of the error page JSP.
     * @param request the HttpServletRequest
     * @param response the HttpServletResponse
     * @param context the PortletContext
     * @param exception the exception thrown by a portlet action
     * @return the URL to call
     */
    public String service(HttpServletRequest request, HttpServletResponse
```

```
response, PortletContext context, Exception thrownException)
{
[code omitted for obtaining a LoggerInterface]
//prepare the localized resources for use by the jsp.
try {
    NavigationUtil.prepareLocalizedResources(
        "com.mycompany.portlets.Resources", request, context);
}
catch (java.io.IOException ioe) {
    logger.error(ioe.getMessage(), _loggingContext, ioe);
}

logger.error(thrownException.getMessage(), _loggingContext,
    thrownException);
return "ErrorPage.jsp";
}
}
```

Tasks

Creating a PAR File for Deployment in Your Application

A portlet archive (PAR) file is an archive file that contains all of the files needed in order to deploy a portlet or a group of portlets into the portal Web application.

A PAR file can contain files for one portlet, or files for multiple related portlets. There is no limit to the number of portlets that a PAR file can contain. In addition, a PAR file can contain local portlets, remote portlets, or a combination of local and remote portlets.

To create a PAR file:

1. Create the directory structure for the portlet(s) on your local machine, and place the required files in the appropriate directory location.
2. Use the JAR utility to compress the directories and files into an archive.
3. Rename the archive with a unique name and the extension `.par`.

For information about how to deploy a PAR file into the portal Web application, refer to [Portlet Deployment](#) in the *SAS Web Infrastructure Kit Administrator's Guide*.

PAR File Directory Structure

For correct deployment, you must organize the files in a PAR file using the following directory structure:

Directory	Contents	Notes
(root)	Portlet deployment descriptor file	The name of the deployment descriptor file must be <code>portlet.xml</code> .
<code>/portletname</code>	None	Include one <code>portletname</code> directory (and associated subdirectories) for each portlet that is defined in <code>portlet.xml</code> . The directory name must match the name of the portlet as specified in the <code>name</code> attribute of the <code><local-portlet></code> or <code><remote-portlet></code> element in <code>portlet.xml</code> .
<code>/portletname/classes</code>	Class files and any display resources files that are used by the portlet	Replicate any package structure as subdirectories of <code>/portletname/classes</code> . The display resources files must be in <code>/portletname/classes</code> .
<code>/portletname/content</code>	Web resources used by the portlet, including JSPs, HTML files, CSS files, and images	The directory name must be <code>content</code> . Each portlet can have only one content location directory; however, the content location directory can have an unlimited number of subdirectories.
<code>/portletname/lib</code>	JAR files used by the portlet	

Use Cases

Use Cases

This chapter describes the steps that a portlet developer would need to perform in four common uses cases:

- Creating a simple display portlet. A simple display portlet is one that displays text, data, and/or images, with no localization and no interactive capabilities.
- Creating a localized portlet. A localized portlet, also referred to as an internationalized portlet, displays its text, numbers, and dates in the correct language and format for the locale (country and language) that the user has selected.
- Creating a portlet template (editable portlet). A portlet template is a portlet from which users can create their own portlet instances. By clicking the portlet's Edit icon, the user can change the portlet's behavior as enabled by the editor action that is associated with the portlet.
- Creating a remote portlet. A remote portlet calls a Web application which is deployed and executed outside of the portal Web application.

For detailed information about a specific portlet development task, refer to the Tasks chapter. For examples of fully developed portlet code, see the Samples chapter.

Use Cases

Creating a Simple Display Portlet

The simplest kind of portlet is a JSP page that displays text, data, and images, with no interactive capabilities.

You might have existing display JSP pages that you would like to deploy in the portal Web application. For example, you might have custom JSP pages (called widgets) that you created for a previous version of the SAS Information Delivery Portal Web application. Or you might have created JSP pages using another SAS product such as SAS Enterprise Guide. It is easy to implement these JSP pages as portlets.

To implement a simple JSP portlet:

1. Create the JSP page, if it is not already created.
 2. Create a portlet deployment descriptor file.
 3. Create a display resources file containing the portlet title and description.
 4. Pack the files into a PAR file.
-

Step 1: Create the JSP Page

A JSP page that you deploy as a local portlet can be as simple as the following example:

```
This is a simple JSP portlet.
```

The only requirements are that

- the JSP page must be an HTML fragment. This means it must
 - ◆ not contain starting and ending <HTML>, <HEAD>, or <BODY> tags
 - ◆ be able to be displayed inside a table cell in an HTML document
- the file name must have the extension .jsp.

For more information, see [Developing the Presentation JSP](#).

Step 2: Create a Portlet Deployment Descriptor File

To create a portlet deployment descriptor file for a simple display JSP portlet:

- a. Specify the portlet name and title using the attributes of the <local-portlet> element. The name cannot contain spaces. The portlet identifier, which consists of the portlet path (defined in the portlet-path element) together with the portlet name, must be unique within the portal Web application.
- b. Optionally, specify key words for use in searching in the <keyword> element.
- c. Specify the portal Web application's default initializer, which is called JspPortletInitializer, in the initializer-type element. For this initializer, you must specify the following in the <init-param> element:

- i. display-page in the param-name sub-element
- ii. the name of your JSP page in the param-value sub-element.

Note: The default initializer passes only the display-page parameter. To pass additional parameters, you would need to create your own initializer class (see [Creating an Initializer Action Class](#)).

In most cases, you can use the default settings for the remainder of the elements. For more information, see [Creating a Portlet Deployment Descriptor](#).

Here is an example of a `portal.xml` file for a simple display JSP page that will run as a local portlet (that is, inside the portlet container):

```
<?xml version="1.0" ?>
<!DOCTYPE portlets SYSTEM "http://localhost:9090/portlet.dtd">
<portlets>
  <local-portlet name="SimpleJsp" title="SIMPLE_JSP_PORTLET">
    <keywords>
      <keyword>example</keyword>
    </keywords>
    <initializer-type>com.sas.portal.portlet.JspPortletInitializer
    </initializer-type>
    <init-param>
      <param-name>display-page</param-name>
      <param-value>simpleJspTest.jsp</param-value>
    </init-param>
    <content-location>content</content-location>
    <portlet-path>/sas/portlets</portlet-path>
    <portlet-actions>
      <portlet-action name="display">
        <type>com.sas.portal.portlet.JspPortlet</type>
      </portlet-action>
    </portlet-actions>
  </local-portlet>
</portlets>
```

Step 3: Create a Display Resources File Containing the Portlet Title and Description

Create the display resources file as follows:

- a. Use `<key>=<value>` statements to define text strings for `portlet.title` and `portlet.description`, as in the following example:

```
portlet.title=Welcome Portlet
portlet.description=Welcome Portlet
```

- b. Name the file `portletDisplayResources.properties`.
- c. Place the file in the `/portletname/classes` directory of the PAR file.

If you want the portlet title and description to be localized at the time of deployment according to the portal Web application's default locale, you can create multiple display resources files for various locales. For details, see [Creating Display Resources Files](#).

Note: If you omit this step, the portal Web application will use the portlet's name to create a default description. In addition, the portlet deployment mechanism will send a warning message to the server log that no localized title or description can be found.

Step 4: Pack the Files into a PAR File

Use the JAR utility to compress the portlet deployment descriptor, the JSP page, and any needed support files into an archive. For information about the required directory structure, see [Creating a PAR File for Deployment in Your Web Application](#).

Use Cases

Creating a Localized Portlet

A localized portlet, also referred to as an internationalized portlet, is one that displays its text, numbers, and dates in the correct language and format for the locale (country and language) setting that the user has selected. Users can specify their locale preference in the Web browser software or in the User Preferences option of the portal Web application.

To create a localized portlet, use the followings steps :

1. Create files containing translated messages.
 2. Create display resources files containing translated titles and descriptions (optional).
 3. Create an action class for the portlet.
 4. Use internationalization tags from the JSP Standard Tag Library (JSTL).
-

Step 1: Create Files Containing Translated Messages

Place translations of your portlet's messages in resource bundles, which are files suitable for use by the Java `ResourceBundle` class. These files must be named with the extension `.properties`.

Create these files as follows:

- a. Create a separate file for each language (or each country and language combination) that you need to support.
- b. In each file, use `<key>=<value>` statements to define the text strings. Use the same key names in each file.

If a message does not require translation, you can omit it. For example, if the default locale is U.S. English, some messages might not require translation to British English. These messages can be omitted from the British English resource bundle, and the default U.S. English translation will be used.

- c. Name the files as follows:

- ◆ Use the same base name for each file (for example, `Resources`).
- ◆ Append each file's base name with the appropriate locale identifier (for example, `_en_US` for U.S. English, `_fr_CA` for Canadian French, and so on). The file for the default locale does not need to have a locale identifier.
- ◆ Give each file name the extension `.properties`.

For example, if the default language and country is U.S. English, you could create

- ◆ a file called `Resources.properties` that contains U.S. English messages
 - ◆ a file called `Resources_fr_CA.properties` that contains Canadian French messages
 - ◆ a file called `Resources_uk_EN.properties` that contains British English messages
- d. You can place all of the files in the package of your choice. Since the files must be included in the PAR file, it might be convenient to place them in the package where the portlet's action class will be located.

For more information on resource bundles, refer to the internationalization information on the Sun Web site at <http://java.sun.com>.

Step 2: Create Display Resources Files Containing Translated Titles and Descriptions (Optional)

The resource files created in Step 1 affect only the text that is displayed inside of the portlet. They do not affect the metadata (including the title and description) that describes the portlet. The SAS Metadata Repository cannot store multiple, localized values for metadata. Therefore, the portlet title and description cannot be translated based on the user's locale preference.

However, the portlet title and description can be translated into the portal Web application's default locale at the time that the portlet is deployed. To make this happen, you must place translated titles and descriptions for your portlet in `portletDisplayResources.properties` files.

When the portlet is first deployed, the deployment process will check to see which default locale was specified when the portal Web application was installed. Based on this locale, the deployment process uses the title and description from the appropriate display resource file to create metadata and register the portlet in the SAS Metadata Repository.

Note: If your portlet will be deployed in only one locale (language), this step can be omitted. If you do omit this step, the portlet deployer mechanism will send a warning message to the server log that no localized title or description can be found. In addition, you will not be able to specify a description for your portlet. Instead, the portal Web application will use the portlet's name to create a default description.

Create the display resources files as follows:

- a. Create a separate file for each language (or each country and language combination) that you need to support. In each file, use `<key>=<value>` statements to define text strings for `portlet.title` and `portlet.description`, as in the following examples:

```
portlet.title=Welcome Portlet
portlet.description=Welcome Portlet

portlet.title=Portlet de bienvenida
portlet.description=Portlet de bienvenida
```

- b. To name each file, use the base name `portletDisplayResources`. Append each file's base name with the appropriate locale identifier (as described in Step 1–c previously) and then with the extension `.properties`.
- c. Place the files in the `/portletname/classes` directory of the PAR file.

Step 3: Create an Action Class for the Portlet

Create an action class for your portlet, as follows:

- a. Import the `com.sas.portal.portlet.NavigationUtil` class.
- b. Add the following code to the `service()` method of your action class, with the correct package and name for your resources:

```
NavigationUtil.prepareLocalizedResources(
    "com.mycompany.portlets.Resources", request, context);
```

This method uses the portlet classloader to obtain the resource bundle from the portlet's PAR file. It then uses the bundle and the locale of the current user to make a new JSTL localization context. The localization

context will be available to your portlet's JSP page.

Step 4: Use the Internationalization Tags from the JSP Standard Tag Library (JSTL)

When you create the portlet's JSP page, use tags from the JSP Standard Tag Library to display text, as follows:

- a. First, include the taglib directive for the JSTL formatting tags:

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
```

The directive must appear before the first use of these tags.

- b. To display a text message from the resource bundles, use the message tag with the key attribute, as in the following example:

```
<fmt:message key="welcome.msg1.txt"/>
```

This example will obtain text associated with the key `welcome.msg1.txt` from the resource bundle for the locale that most closely matches the user's locale preference.

You can also use the other JSTL tags for formatting such locale-sensitive items as dates and currency. The portal Web application makes the user's locale available to those tags.

For more information about creating JSP pages for portlets, see [Developing the Presentation JSP](#). For information about JSTL, refer to the Sun Web site at <http://java.sun.com/products/jsp/jstl>. The JSTL jar files are provided with the SAS Web Infrastructure Kit.

Use Cases

Creating a Portlet Template (Editable Portlet)

A portlet template is a portlet from which users can create their own portlet instances. When created, the new portlet instance belongs to the user. By clicking the portlet's Edit icon, the user can change the portlet's behavior as enabled by the editor action that is associated with the portlet.

The portal Web application is delivered with three portlet templates: the Collection Portlet template, the WebDAV Navigator Portlet template, and the URL Display Portlet template. The Add a Portlet function of the portal Web application enables users who have the appropriate permissions to create new instances of these portlets. Users can then edit the portlets that they have created.

The SAS Web Infrastructure Kit enables you to develop and deploy additional portlet templates. After you deploy a new portlet template, the template name appears in the drop-down box on the Create a New Portlet dialog box along with the Collection Portlet and the WebDAV Navigator Portlet.

To create a new portlet template, use the following steps:

1. Code the portlet deployment descriptor file to support replication and editing capabilities.
 2. Code the editor action class.
 3. Code the editor JSP page.
 4. Create resource bundles to support localized text.
-

Step 1: Code the Portlet Deployment Descriptor File

To support replication and editing capabilities, code the portlet deployment descriptor file (`portlet.xml`) as follows:

- a. In the `local-portlet` element, specify the attribute value `editorType="portlet"`. This value indicates that portlets created from the template are to be editable.

Note: If you specify this attribute value, you must also specify a portlet action with the attribute `editor="true"`, as described in Step 1–c, which follows. Otherwise, the portlet deployer will send a warning to the server log and will not deploy the portlet.

- b. In the `deployment` element, specify the attribute value `userCanCreateMore="true"`. This value indicates that the portlet is a template and can be replicated by portal users.
- c. Code the action elements:

- ◆ Code a default action to specify processing that is to occur before the portlet's JSP page renders, as in this example:

```
<portlet-action name="display" default="true">
  <type>com.mycompany.portlets.NavigatorAction</type>
</portlet-action>
```

- ◆ You must code an `<action>` element to specify the editor action which is to be invoked when a user clicks the portlet's Edit icon. Use the attribute value `editor="true"` to indicate that the action is for this purpose, as in the following example:

```
<portlet-action name="editor" editor="true" >
  <type>com.mycompany.portlets.EditorAction</type>
</portlet-action>
```

- ◆ Code `<action>` elements to specify the actions that are to be invoked when the user clicks buttons on the editor JSP page, as in the following example:

```
<portlet-action name="ok" default="false" >
  <type>com.mycompany.portlets.ReturnAction</type>
</portlet-action>
<portlet-action name="cancel" default="false" >
  <type>com.mycompany.portlets.CancelAction</type>
</portlet-action>
```

Step 2: Code the Editor Action Class

Code the editor action class, which is the class that is to be invoked when a user clicks the portlet's Edit icon. In the portlet deployment descriptor file, the name of this action class must be specified in the `<action>` element with the attribute value `editor="true"`, as described previously in Step 1–c. The editor action class should

- place the portlet's navigation path into the `PortletContext` object so that it can be used by the portlet's JSP page.
- use the `com.sas.portal.portlet.NavigationUtil` class to create URLs for buttons on the JSP page (for example, **OK** and **Cancel**). You can place the URLs into either the `HttpServletRequest` object or in the `PortletContext` object.
- When editing is complete, display the portlet again using the default display action. (This is necessary because the portlet takes over the entire page as its display mode when it is in edit mode.) To reset the display
 - check to see that edit mode is active. This is necessary because the editor might have delegated control to other portlet actions that use the default display.
 - if edit mode is active, call the `PortletContext`'s `resetMode()` method to reset the display mode from full page to the default display action.

A complete example of an editor action class follows:

```
public String service(HttpServletRequest request,
    HttpServletResponse response, PortletContext context) throws Exception
{
    InformationServicesSelector infoSelector = (InformationServicesSelector)
        context.getAttribute("myISSKey");
    String path = infoSelector.getPath();
    //put the path into the portlet's context object
        context.setAttribute("myISS_InitialPath", path);

    //create the URLs for the OK and Cancel buttons.
    request.setAttribute("Return_URL",
        NavigationUtil.buildBaseURL(context, request, "ok"));

    request.setAttribute("Cancel_URL",
        NavigationUtil.buildBaseURL(context, request, "cancel"));

    //prepare the localized resources for use by the jsp.
    try {
        NavigationUtil.prepareLocalizedResources(
            "com.mycompany.portlets.NavigatorResources",
            request, context);
    }
    catch (java.io.IOException ioe) {
        Logger.error(ioe.getMessage(), _loggingContext, ioe);
    }
}
```

```

}
// be sure we are in edit mode, then call reset.
if (context.getMode().equals(PortletContext.EDIT_MODE)) {
context.resetMode();
}
}
}

```

Step 3: Code the Editor JSP Page

When a portlet goes into edit mode, it takes over the entire page as its display area. Code the JSP page for this area as follows:

1. Obtain the portlet's navigation path from the portlet context so that the page can be displayed to the user, as in the following scriptlet code example:

```

PortletContext context = (PortletContext) request.getAttribute (
com.sas.portal.portlets.PortletConstants.CURRENT_PORTLET_CONTEXT );
String path = (String) request.getAttribute ( "myISS_InitialPath" );

```

2. Create buttons (for example, **OK** and **Cancel**) with the appropriate form actions. Obtain URLs for these buttons from the `HttpServletRequest` or the `PortletContext` object (as described previously in Step 2–b), and assign them to the action attribute. To localize the button text, you can use the message tag from the JSTL tag library, as in the following scriptlet code example:

```

<form method="post"
  action = "<%= (String)
    request.getAttribute("myISS_InitialPath") %>">
  <input class="button" type="submit" value= "<fmt:message
    key="action.ok.txt"/>" name="submit" >
</form>

```

3. Code the visual elements of the page as desired.

For a complete example of an editor JSP page, see [Sample: Portlet Template, or Editable Portlet \(DisplayURL\)](#).

Step 4: Create Resource Bundles to Support Localized Text

To ensure that the portlet can display localized text, create a `NavigatorResources.properties` file, and then create localized versions of the same file. These files should contain the appropriate key–value pairs that are needed to obtain localized text. For more information, see [Creating a Localized Portlet](#).

Use Cases

Creating a Remote Portlet

A remote portlet is a portlet that calls a remote Web application. A remote Web application is an application that runs outside of the portal Web application.

To create a remote portlet, use the following steps:

1. Create the Web application.
 2. Create a portlet deployment descriptor file.
 3. Create display resources files containing the portlet title and description.
 4. Create WAR and PAR files.
-

Step 1: Create the Web Application

The Web application for a remote portlet can be developed using any Web technology, and can be as simple as a single JSP page. The Web application

- must be located anywhere outside of the portal Web application
- has access to the portal Web application's context information, including session and user identity information, through the use of SAS Foundation Services (this is the only point at which remote portlets are coupled with the portal Web application)
- must display an HTML fragment when a request is received from the portal Web application
- must rewrite URLs so that requests are routed through the portal Web application before they are passed to the remote portlet.

For an example of a JSP page for a remote portlet's Web application, see [Sample: Web Application \(HelloUserWikExample\)](#).

Step 2: Create a Portlet Deployment Descriptor File

To create a portlet deployment descriptor file for a remote portlet:

- a. In the `<remote-portlet>` element
 - i. specify the portlet's name and title. The name cannot contain spaces. The portlet identifier, which consists of the portlet path (defined in the `portlet-path` element) together with the portlet name, must be unique within the portal Web application.
 - ii. specify the value "true" for the `passContextId` attribute. This value makes the portal Web application's session information, including user identity, available to the remote portlet.
- b. Optionally, use the `<keyword>` element to specify key words for use in searching.
- c. Specify the URL for the remote portlet's Web application in the `url` subelement of the `portlet-action` element. This subelement must contain a fully qualified URL, and the URL must contain a fully qualified host domain name.

For more information, see [Creating a Portlet Deployment Descriptor](#). For an example of a deployment descriptor for a remote portlet, see [Sample: Remote Portlet \(HelloUserRemotePortlet\)](#).

Step 3: Create Display Resources Files Containing the Portlet Title and Description

To specify the title and description for the portlet's metadata, create a `portletDisplayResources.properties` file. If you want the portlet title and description to be localized at the time of deployment according to the portal Web application's default locale, then create a separate `portletDisplayResources.properties` for each locale. For details, see [Creating Display Resources Files](#).

Note: If you omit this step, the portal Web application will use the portlet's name to create a default description. In addition, the portlet deployment mechanism will send a warning message to the server log that no localized title or description can be found.

Step 4: Create the PAR File and WAR File

Use the JAR utility to place the portlet deployment descriptor into an archive. For information about the required directory structure, see [Creating a PAR File for Deployment in Your Web Application](#).

You must also create a WAR file to use in deploying the remote portlet's Web application.

Samples

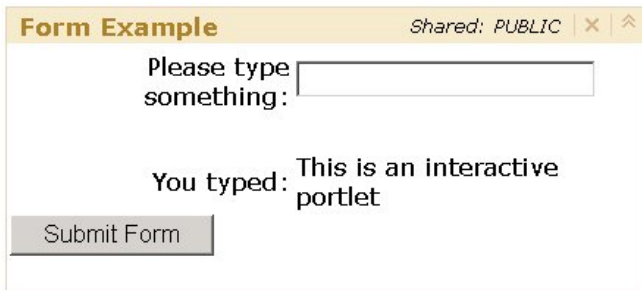
Samples

This chapter provides fully developed samples of custom portlets. The chapter includes complete code for portlet deployment descriptors, JSP pages, resource files, and action classes as applicable for each portlet. The following sample portlets are provided:

- Welcome is a simple display portlet which has no interactive capabilities. Because it is internationalized, it displays text in the user's locale (language and country) preference.



- FormExample is an interactive form portlet that accepts free-form input and displays it back to the user.



- DisplayURL is a portlet template (also referred to as an *editable portlet*), which is a portlet from which users can create their own portlet instances. The DisplayURL portlet template, which is delivered with the portal Web application, enables users to create portlets that return HTML content from any URL.



- HelloUserWikExample is a Web application that is enabled by SAS Foundation Services. The application displays the string Hello *user*, where *user* is the name of the user who is logged on to the portal Web application.
- HelloUserRemotePortlet is a remote portlet that executes the sample Web application HelloUserWikExample and displays the name of the user who is logged on to the portal Web application.



For detailed information about a specific portlet development task, see Tasks. For general information about creating various types of portlets, see Use Cases.

Sample: Localized Display Portlet (Welcome Portlet)

Sample: Localized Display Portlet (Welcome Portlet)



The sample portlet called Welcome is

- a simple display portlet that has no interactive capabilities
- a local portlet that runs inside the portlet container
- an internationalized portlet that displays text in the user's locale (language and country) preference.

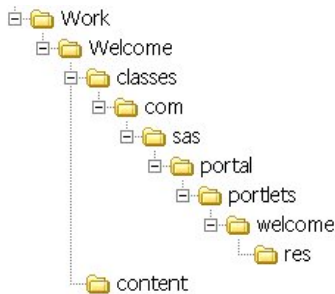
The following steps were used to create the Welcome portlet. Click on each step to display details.

1. [Create a directory structure](#) for the portlet.
2. [Create the portlet deployment descriptor](#) (portlet.xml).
3. [Create the display page](#) (Welcome.jsp).
4. [Create the action class](#) (WelcomeAction.class).
5. [Create resource bundles](#) to support eleven different locales.
6. [Create translated titles and descriptions](#) in the portletDisplayResources.properties files to support eleven different locales.
7. [Create the PAR file, and deploy and test the portlet.](#)

Sample: Localized Display Portlet (Welcome Portlet)

Step 1: Create the Directory Structure

The following directory structure was used to create the Welcome portlet:



This structure includes the following directories and subdirectories:

Directory	Contents
Work (root) This directory serves as a development area for the portlet.	<u>Portlet deployment descriptor file</u> <code>portlet.xml</code> The name of the portlet deployment descriptor file must be <code>portlet.xml</code> .
/Welcome	This is the main portlet directory. It does not contain any files. The directory name must not have any spaces, and it must match the name of the portlet as specified in the name attribute of the <code><local-portlet></code> element in <code>portlet.xml</code> .
/Welcome/classes	<u>Display resource files</u> : <code>portletDisplayResources_de.properties</code> , <code>portletDisplayResources_en.properties</code> , <code>portletDisplayResources_es.properties</code> , and so on.
/Welcome/classes/com/sas/portal/portlets/welcome	The <u>action class</u> called <code>WelcomeAction.class</code> .
/Welcome/classes/com/sas/portal/portlets/welcome/res	<u>Resource bundles</u> : <code>Resources.properties</code> , <code>Resources_de.properties</code> , <code>Resources_es.properties</code> , <code>Resources_fr.properties</code> , and so on. Note: Alternatively, the resource bundles could be placed in the same directory as the action class.
/Welcome/content	This directory contains the <u>display page</u> called <code>Welcome.jsp</code> . Each portlet can have only one content location directory, and the directory name must be <code>content</code> . However, the content location directory can have an unlimited number of subdirectories.

The following rules apply when you set up the directory structure:

- Neither portlet names nor their paths can contain spaces.
- The portlet identifier (which consists of the name and the path) must be unique. Developers should devise a convention to ensure unique name–spaces, similar to the conventions used for naming Java packages.

For example, the Sales division of a company named ABCD could create portlets in the path `ABCD/Sales`, and the Purchasing division could create portlets in the path `ABCD/Purchasing`. Then, both Sales and Purchasing could have different portlets named `Welcome`.

Sample: Localized Display Portlet (Welcome Portlet)

Step 2: Create the Portlet Deployment Descriptor

The portlet deployment descriptor is an XML file that provides all of the information that the portal Web application needs to deploy one or more portlets. Here is the portlet deployment descriptor for the Welcome portlet. The boxes contain explanatory comments. For more information, see [Creating a Portlet Deployment Descriptor](#).

```
<?xml version="1.0" encoding="UTF-8"?>
```

The DOCTYPE statement must be present in the descriptor file in order for the portlet to run. However, the document type definition (DTD) does not need to be accessible at the URL that the statement specifies.

If you want to look at the `portlet.dtd` file, you can find it in the portal setup directory in the path `Portal\WEB-INF`. For example, if you used the default install location on a Windows system, then the DTD is located under the following path: `c:\Program Files\SAS\Web\Portal2.0.1\Portal\WEB-INF`.

```
<!DOCTYPE portlets SYSTEM "http://www.sas.com/idp/portlet.dtd">
```

```
<portlets>
```

The `local-portlet` element assigns the name `Welcome` to the portlet. The name cannot contain spaces. The portlet identifier, which consists of the portlet path (defined in the `portlet-path` element) together with the portlet name, must be unique within the portal Web application.

The `icon` attribute is optional. It specifies the icon that is to appear with the portlet's name in the portal Web application's search results. In the portlet's directory structure, the icon's image file must be placed in or under the `content` directory. If it is in a subdirectory of `content`, you must specify the subdirectory with the image name. For example, if the image `greeting.gif` is in the path `content\icons`, then you would specify `icon="icons/greeting.gif"`.

If an icon is not specified, then the default icon for portlets will be used.

```
<local-portlet name="Welcome" title="Welcome" icon="Portlet.gif">
```

The `localized-resources` element lists the locales that the portlet supports. [Display resource files](#) must be provided for each of these locales.

```
<localized-resources locales="de,en,es,fr,it,ja,pl,ru,sv,zh_CN" />
```

The `deployment` element specifies that the portlet is to be available to the Public group. This means that all users will be able to search for this portlet and add it to their pages.

```
<deployment scope="group" autoDeploy="true" userCanCreateMore="false">
  <group>Public</group>
</deployment>
```

Since the `Welcome` portlet does not need its own initializer class, the portal Web application's default portlet initializer (`JspPortletInitializer`) is specified. This class requires a parameter called `display-page`. The initializer places the value of this parameter in the `PortletContext` object so that it can be used by the portlet's action class. The value of the parameter is the name of the `Welcome` portlet's JSP page, called [Welcome.jsp](#).

Note: The default initializer passes only the `display-page` parameter. To pass additional parameters, you would need to create your own initializer class (see [Creating an Initializer Action Class](#)).

```
<initializer-type>
  com.sas.portal.portlets.JspPortlet.JspPortletInitializer
</initializer-type>
<init-param>
  <param-name>display-page</param-name>
  <param-value>Welcome.jsp</param-value>
</init-param>
```

The `portlet-path` element specifies the directory location where the portlet is to be deployed. The portlet identifier, which consists of the portlet path together with the portlet name (defined in the `local-portlet` element), must be unique within the portal Web application. For example, Orion Star Sports & Outdoors could have two Welcome portlets if different paths are specified for each (as in `OrionStar/Sales/Welcome` and `OrionStar/Purchasing/Welcome`).

```
<portlet-path>/sas/portlets</portlet-path>
<portlet-actions>
```

To provide for internationalization of the text that appears inside the portlet border, the Welcome portlet has its own action class, called [WelcomeAction](#). The name of the class is specified in the `type` subelement of the `portlet-action` element.

```
<portlet-action name="display" default="true">
  <type>com.sas.portal.portlets.welcome.WelcomeAction</type>
</portlet-action>
</portlet-actions>
</local-portlet>
</portlets>
```

Sample: Localized Display Portlet (Welcome Portlet)

Step 3: Create the Display Page

JSP pages are the presentation components of portlets. This is the source code for the Welcome portlet's JSP page, called `Welcome.jsp`. The boxes contain explanatory comments. For more information, see [Creating the Presentation JSP](#).

```
<!-- Copyright (c) 2001 by SAS Institute Inc., Cary, NC 27513 -->
```

The following line contains the UTF-8 directive, which is required for internationalization. This directive causes all user input to be encoded in the 8-bit Unicode Transformation Format, which supports all of the world's languages including those that use non-Latin1 characters.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
```

The following line contains the taglib directive for the JSP Standard Tag Library (JSTL) formatting tags. The directive must appear before the first use of these tags.

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<!--This portlet provides the welcome greeting on the Public Kiosk. -->
```

The following lines use JSTL formatting tags to display text. The `key` attribute is used to obtain the appropriate text from the resource bundle that most closely matches the user's locale preference. The SAS Web Infrastructure Kit makes the user's locale available to these tags.

```
<fmt:message key="welcome.msg1.txt" />
<br>
<fmt:message key="welcome.msg2.txt" />
```

Sample: Localized Display Portlet (Welcome Portlet)

Step 4: Create the Action Class

The Welcome portlet has its own action class, `WelcomeAction`, which provides support for localizing messages. This class extends `com.sas.portal.portlet.HTMLPortletAction`, which contains code to correctly display non-Latin1 character sets when the portal Web application displays the portlet in preview mode.

The source code for `WelcomeAction` follows. The boxes contain explanatory comments. For more information, see [Creating Action Classes](#).

```
/** Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.welcome;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sas.portal.portlet.HTMLPortletAction;
import com.sas.portal.portlet.NavigationUtil;
import com.sas.portal.portlet.PortletContext;

/**Action for the Welcome Portlet. This prepares the localized resource
 * bundles for use by the JSTL tags within the portlet's JSP.
 * @version 1
 */
public final class WelcomeAction extends HTMLPortletAction {

    /**
     * Configure the JSTL localization context for use in the Welcome
     * portlet. Returns the value of "display-page" from the portlet's
     * XML descriptor.
     *
     * @param request The HttpServletRequest associated with the
     *                 method invocation
     * @param response HttpServletResponse associated with the
     *                 method invocation
     * @param context PortletContext mapped to the request path
     *
     * @return java.lang.String - representing a valid URL.
     */
    public String service(HttpServletRequest request,
        HttpServletResponse response, PortletContext context)
        throws Exception{

        super.service(request, response, context);

        NavigationUtil.prepareLocalizedResources(
            "com.sas.portal.portlets.welcome.res.Resources", request, context);

        //This comes from the portlet.xml.
        String url = (String) context.getAttribute("display-page");
```

In the following code, the `NavigationUtil` method uses the portlet's classloader to obtain the portlet's resource bundle. Using this bundle and the locale of the current user, it creates a new JSTL localization context. The localization context is made available to the portlet's JSP page with request scope.

```
        super.service(request, response, context);

        NavigationUtil.prepareLocalizedResources(
            "com.sas.portal.portlets.welcome.res.Resources", request, context);

        //This comes from the portlet.xml.
        String url = (String) context.getAttribute("display-page");
```

```
        return url;
    }
}
```

Sample: Localized Display Portlet (Welcome Portlet)

Step 5: Create the Resource Bundles

The resource bundles provide translated text to be displayed inside the Welcome portlet. The portlet's `WelcomeAction.class` calls the `NavigationUtil.prepareLocalizedResources()` method to create a JSTL localization context based on the user's locale preference. This context enables the JSTL tags in `Welcome.jsp` to use the appropriate resource bundle to display the text. For more information about creating resource bundles, see [Creating a Localized Portlet](#).

Note: For information about localizing the portlet's title and description, see [Step 6: Create Translations of the Title and Description](#).

A number of resource bundles are provided with the Welcome portlet. The contents of three of the files (`Resources.properties` for U.S. English, `Resources_de.properties` for German, and `Resources_es.properties` for Spanish) follow.

```
#This is where you put key/value pairs for message strings that need to
#be localized.
##These are the messages for the Welcome portlet
#This is for the Public Kiosk Welcome portlet
welcome.msg1.txt=Welcome to Version 2 of the Information Delivery Portal.
welcome.msg2.txt=Please look around!
```

```
#This is where you put key/value pairs for message strings that need to
#be localized.
##These are the messages for the Welcome portlet
#This is for the Public Kiosk Welcome portlet
welcome.msg1.txt=Willkommen bei Information Delivery Portal Version 2.
welcome.msg2.txt=Schauen Sie sich um!
```

```
#This is where you put key/value pairs for message strings that need to
#be localized.
##These are the messages for the Welcome portlet
#This is for the Public Kiosk Welcome portlet
welcome.msg1.txt=Bienvenido a la versión 2 de Information Delivery
Portal.
welcome.msg2.txt=Puede continuar
```

Sample: Localized Display Portlet (Welcome Portlet)

Step 6: Create Translated Titles and Descriptions

Display resource files called `portletDisplayResources.properties` contain translated titles and descriptions for the Welcome portlet. These files contain text to be used in creating the metadata that describes the portlet.

When the portlet is first deployed, the deployment process checks to see which default locale was specified when the portal Web application was installed. Based on this locale, it uses the title and description from the appropriate display resource file to create metadata and register the portlet in the SAS Metadata Repository.

Note: The SAS Metadata Repository cannot store multiple, localized values for metadata. Therefore, the title and description are translated only into the portal Web application's default locale. They cannot be translated based on the user's locale preference.

A number of display resource files are provided for the Welcome portlet. The contents of three of the files (`portletDisplayResources_de.properties`, `portletDisplayResources_en.properties`, and `portletDisplayResources_es.properties`) follow.

```
portlet.title=Begrüßungs-Portlet
portlet.description=Begrüßungs-Portlet
```

```
portlet.title=Welcome Portlet
portlet.description=Welcome Portlet
```

```
portlet.title=Portlet de bienvenida
portlet.description=Portlet de bienvenida
```

Sample: Localized Display Portlet (Welcome Portlet)

Step 7: Create the PAR File, and Deploy and Test the Portlet

The last step in developing the Welcome portlet was to zip its files into a PAR file. The PAR file includes:

- Appropriately named and organized directories and subdirectories, as described in [Step 1: Create the Directory Structure](#).
- All of the portlet's supporting files, including the files created in Steps 2 through 6. The files must be placed in the appropriate directories as described in [Step 1: Create the Directory Structure](#) and [Creating a PAR File for Deployment in Your Application](#).

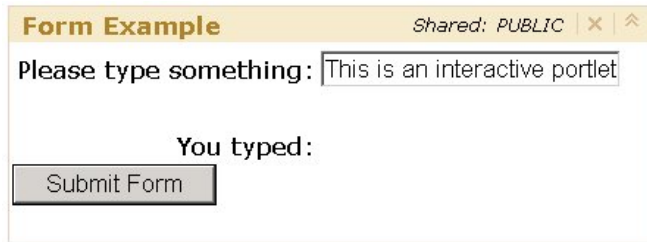
The JAR utility was used to compress the directories and files into an archive, and the archive was given the name `Welcome.par`.

It is a good practice to deploy new portlets into a staging area (that is, a test installation of the portal Web application) for verification and testing before deploying them into the production environment. For information about how to deploy a PAR file into the portal Web application, refer to [Portlet Deployment](#) in the *SAS Web Infrastructure Kit Administrator's Guide*.

Sample: Interactive Form Portlet (FormExample)

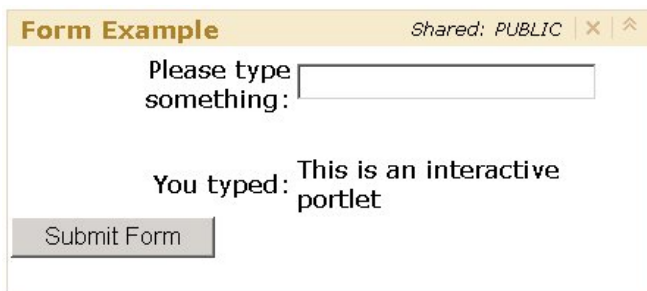
Sample: Interactive Form Portlet (FormExample)

The sample portlet called FormExample is an interactive portlet. It accepts free-form input from the user.



The screenshot shows a portlet window titled "Form Example" with a "Shared: PUBLIC" status bar. Inside, there is a label "Please type something:" followed by a text input field containing the text "This is an interactive portlet". Below the input field is the label "You typed:". At the bottom left is a button labeled "Submit Form".

When the user clicks **Submit Form**, the portlet displays the entered text back to the user.



The screenshot shows the same portlet window after the user has submitted the form. The input field is now empty. The label "You typed:" is now followed by the text "This is an interactive portlet", which is the text that was entered in the input field. The "Submit Form" button remains at the bottom left.

FormExample is a local portlet that runs inside the portlet container. It was developed using a JavaBean, which places values in the `PortletContext` object so that the values are available to the JSP page. The `HttpServletRequest` or `HttpSession` object could be used for this purpose. However, the `PortletContext` object is unique to the portlet and is not shared with other processes. Therefore, using it avoids collisions that could cause attribute values to be overwritten.

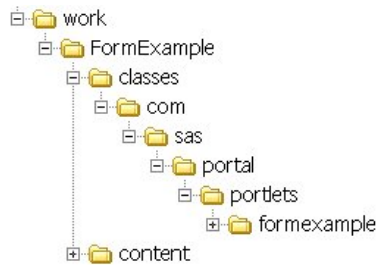
The following steps were used to create the FormExample portlet. Click on each step to display details.

1. [Create a directory structure](#) for the portlet.
2. [Create the portlet deployment descriptor](#) (`portlet.xml`).
3. [Create the display page](#) (`FormExample.jsp`).
4. [Create the action class](#) (`DisplayAction.class`).
5. [Create the bean](#).
6. [Zip the portlet into a PAR file, and deploy and test the portlet](#).

Sample: Interactive Form Portlet (FormExample)

Step 1: Create the Directory Structure

The following directory structure was used to create the FormExample portlet:



This structure includes the following directories and subdirectories:

Directory	Contents
work (root) This directory serves as a development area for the portlet.	<u>Portlet deployment descriptor file</u> portlet.xml The name of the deployment descriptor file must be portlet.xml.
/FormExample	This is the main portlet directory. It does not contain any files. The directory name must not have any spaces, and it must match the name of the portlet as specified in the name attribute of the <local-portlet> element in portlet.xml.
/FormExample/classes/com/sas/portal/portlets/formexample	The <u>action class</u> called DisplayAction.class and the JavaBean called ExampleBean.class.
/FormExample/content	This directory contains the <u>display page</u> called FormExample.jsp. Each portlet can have only one content location directory, and the directory name must be content. However, the content location directory can have an unlimited number of

The following rules apply when you set up the directory structure:

- Neither portlet names nor their paths can contain spaces.
- The portlet identifier (which consists of the name and the path) must be unique. Developers should devise a convention to ensure unique name-spaces, similar to the conventions used for naming Java packages.

For example, the Sales division of a company named ABCD could create portlets in the path ABCD/*Sales*, and the Purchasing division could create portlets in the path ABCD/*Purchasing*. Then both Sales and Purchasing could have different portlets named *FormExample*.

Sample: Interactive Form Portlet (FormExample)

Step 2: Create the Portlet Deployment Descriptor

The portlet deployment descriptor is an XML file that provides all of the information that the portal Web application needs to deploy one or more portlets. Here is the portlet deployment descriptor for the FormExample portlet. The boxes contain explanatory comments. For more information, see [Creating a Portlet Deployment Descriptor](#).

```
<?xml version="1.0" encoding="UTF-8"?>
```

The DOCTYPE statement must be present in the descriptor file in order for the portlet to run. However, the document type definition (DTD) does not need to be accessible at the URL that the statement specifies.

If you want to look at the `portlet.dtd` file, you can find it in the portal setup directory in the path `Portal\WEB-INF`. For example, if you used the default install location on a Windows system, then the DTD is located under the following path: `c:\Program Files\SAS\Web\Portal2.0.1\Portal\WEB-INF`.

```
<!DOCTYPE portlets SYSTEM "http://www.sas.com/idp/portlet.dtd">
```

```
<portlets>
```

The `local-portlet` element assigns the name `FormExample` to the portlet. The name cannot contain spaces. The portlet identifier, which consists of the portlet path (defined in the `portlet-path` element) together with the portlet name, must be unique within the portal Web application.

```
<local-portlet name="FormExample" title="Form Example" >
  <localized-resources locales="en" />
  <keywords>
    <keyword></keyword>
  </keywords>
```

The deployment element specifies that the portlet is to be available to the Public group. This means that all users will be able to search for this portlet and add it to their pages.

```
<deployment scope="group" autoDeploy="true"
  userCanCreateMore="false">
  <group>Public</group>
</deployment>
```

Since no initializer class is specified, the `FormExample` portlet will use the default initializer `JspPortletInitializer`. This initializer requires a page name as a parameter. The `FormExample` has its own JSP page, called [FormExample.jsp](#).

```
<init-param>
  <param-name>display-page</param-name>
  <param-value>FormExample.jsp</param-value>
</init-param>
```

The `portlet-path` element specifies the directory location where the portlet is to be deployed. The portlet identifier, which consists of the portlet path together with the portlet name (defined in the `local-portlet` element), must be unique within the portal Web application.

```
<portlet-path>/sas/portlets</portlet-path>
```

To provide its interactive functionality, the FormExample portlet has its own action class, called DisplayAction. The name of the class is specified in the type subelement of the `portlet-action` element.

```
<portlet-actions>
  <portlet-action name="display" default="true">
    <type>com.sas.portal.portlets.formexample.DisplayAction</type>
  </portlet-action>
</portlet-actions>
</local-portlet>
</portlets>
```

Sample: Interactive Form Portlet (FormExample)

Step 3: Create the Display Page

JSP pages are the presentation components of portlets. This is the source code for the FormExample portlet's JSP page, called `FormExample.jsp`.

This JSP page uses SAS custom tags, which are available with SAS AppDev Studio. For more information, see [Creating the Presentation JSP](#).

```
<!-- Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513 --%>
<%@ page language="java"
import="com.sas.portal.portlet.PortletContext,
com.sas.portal.portlet.PortletConstants"
contentType= "text/html; charset=UTF-8" %>

<%@taglib uri="http://www.sas.com/taglib/sas" prefix="sas"%>

<!--This portlet provides for echoing the user input back to the
portlet display. --%>

<% PortletContext context = (PortletContext)
request.getAttribute(PortletConstants.CURRENT_PORTLET_CONTEXT );

%>

<sas:Form id="form" name="form" method="POST"
action="<%= (String) context.getAttribute(\"formExample_baseURL\")%>">

<table border="0">
<tr>
<td align="right">Please type something:</td>
<td><sas:TextEntry id="userInput" /></td>
</tr>
<tr>
<td> </td>
</tr>

<tr>
<td align="right">You typed:</td>
<td><%= (String) context.getAttribute("formExample_userInput") %></td>
</tr>

</table>

<sas:PushButton id="submit"
text="Submit Form" type="submit" />

</sas:Form>
```

Sample: Interactive Form Portlet (FormExample)

Step 4: Create the Action Class

The FormExample portlet has its own action class, DisplayAction. The source code for DisplayAction follows. For more information, see [Creating Action Classes](#).

```
/**Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.formexample;

import java.util.Enumeration;
import java.util.HashMap;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.beanutils.BeanUtils;

import com.sas.portal.portlet.HTMLPortletAction;
import com.sas.portal.portlet.NavigationUtil;
import com.sas.portal.portlet.PortletContext;

/**Action for the Form Example Portlet. This prepares the URL
 * that will be assigned to the form's action within the portlet's
 * JSP. It also populates a bean with the parameters from the JSP
 * form.
 *
 * @author Todd.Folsom@sas.com
 * @version 1
 */
public final class DisplayAction extends HTMLPortletAction {

    /**
     * Prepare the URL for the form used in the portlet.
     * Returns the value of "FormExample.jsp".
     *
     * @param request The HttpServletRequest associated with the
     *                method invocation
     * @param response HttpServletResponse associated with the
     *                method invocation
     * @param context PortletContext mapped to the request path
     *
     * @return java.lang.String - representing a valid
     *                URL.
     */
    public String service(HttpServletRequest request,
        HttpServletResponse response, PortletContext context)
        throws Exception{

        super.service(request, response, context);

        //prepare the base URL for setting on the form in the JSP.
        // The "display" is the value used in portlet.xml for this
        // action.
        String baseURL = NavigationUtil.buildBaseURL(context, request,
            "display");
        context.setAttribute("formExample_baseURL", baseURL);
    }
}
```

```

//Make a new ExampleBean. Alternatively, this could be made
// once in the portlet initializer class, then you manage
// its properties in the action.
ExampleBean bean = new ExampleBean();

//The BeanUtils class will populate any bean with all the
// parameters from the form.
HashMap map = new HashMap();
Enumeration names = request.getParameterNames();
while (names.hasMoreElements()) {
    String name = (String) names.nextElement();
    map.put(name, request.getParameterValues(name));
}
BeanUtils.populate(bean, map);

//put the userInput into the portlet context so we can get it out
// in the JSP.
context.setAttribute("formExample_userInput", bean.getUserInput());

return "FormExample.jsp";
}

```

Sample: Interactive Form Portlet (FormExample)

Step 5: Create the JavaBean

The FormExample portlet was developed using a JavaBean called `ExampleBean.java`. This JavaBean places values in the `PortletContext` object so that the values are available to the JSP page.

The `HttpServletRequest` or `HttpSession` object could be used for this purpose. However, the `PortletContext` object is unique to the portlet and is not shared with other processes. Therefore, using it avoids collisions that could cause attribute values to be overwritten.

The source code for `ExampleBean.java` follows.

```
/**Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.formexample;

public final class ExampleBean {

    /**Sets the user's input to this property. If input
     * is null, it will be changed to "" so that getUserInput()
     * never returns null.
     *
     * @param input
     */
    public void setUserInput(String input) {
        if (input == null) {
            input = "";
        }
        this.input = input;
    }

    /**Returns the user's input or "".
     *
     * @return String
     */
    public String getUserInput() {
        return input;
    }

    private String input = "";
}
```

Sample: Interactive Form Portlet (FormExample)

Step 6: Create the PAR File, and Deploy and Test the Portlet

The last step in developing the FormExample portlet was to archive its files into a PAR file. The PAR file includes

- appropriately named and organized directories and subdirectories, as described in [Step 1: Create the Directory Structure](#).
- all of the portlet's supporting files, including the files created in Steps 2 through 5. The files must be placed in the appropriate directories as described in [Step 1: Create the Directory Structure](#) and [Creating a PAR File for Deployment in Your Application](#).

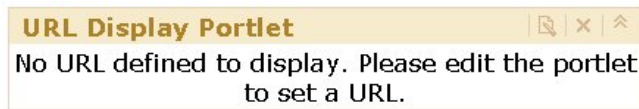
The JAR utility was used to compress the directories and files into an archive, and the archive was given the name `FormExample.par`.

It is a good practice to deploy new portlets into a staging area (that is, a test installation of the portal Web application) for verification and testing before deploying them into the production environment. For information about how to deploy a PAR file into the portal Web application, see [Portlet Deployment](#) in the SAS Web Infrastructure Kit Administrator's Guide.

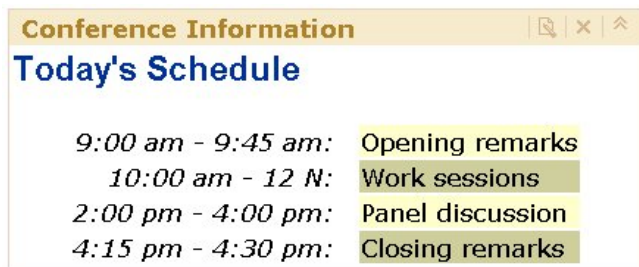
Sample: Portlet Template, or Editable Portlet (DisplayURL)

Sample: Portlet Template, or Editable Portlet (DisplayURL)

A *portlet template*, also referred to as an *editable portlet*, is a portlet from which users can create their own portlet instances. The portlet called DisplayURL, which is delivered with the portal Web application, is one example of a portlet template.



Users can create new instances of this portlet by choosing **Add a portlet** on the Options menu and then choosing **URL Display Portlet** as the portlet type in the Create a New Portlet dialog box. The DisplayUrl portlet includes classes that enable the user to edit the new portlet instance to point to any URL which returns an HTML fragment. For example, the user could edit the portlet instance to point to a URL which returns this HTML fragment.



If you create a portlet template, the title of the portlet template will appear as a portlet type that users can create their own instances of. Action classes that you provide with the portlet template will then enable users to edit the portlet instances that they have created.

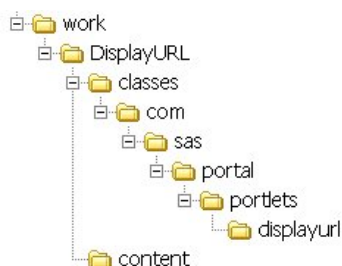
The following steps were used to create the DisplayURL portlet and the URL Display Portlet portlet type. Click on each step to display details.

1. [Create a directory structure](#) for the portlet.
2. [Create the portlet deployment descriptor](#) (portlet.xml).
3. [Create display pages for the portlet and the editor](#) (FormExample.jsp).
4. [Create the action classes.](#)
5. [Create the resource bundle.](#)
6. [Create the display resources file.](#)
7. [Create the PAR file, and deploy and test the portlet.](#)

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Step 1: Create the Directory Structure

The following directory structure was used to create the DisplayURL portlet:



This structure includes the following directories and subdirectories:

Directory	Contents
Work (root) This directory serves as a development area for the portlet.	<u>Portlet deployment descriptor file</u> <code>portlet.xml</code> The name of the deployment descriptor file must be <code>portlet.xml</code> .
/DisplayURL	This is the main portlet directory. It does not contain any files. The directory name must not have any spaces, and it must match the name of the portlet as specified in the name attribute of the <code><local-portlet></code> element in <code>portlet.xml</code> .
/DisplayURL/classes	This directory contains the <u>display resources file</u> file.
/DisplayURL/classes/com/SAS/portal/portlets/displayurl	This directory contains the <u>resources.properties</u> file and the following action classes: <ul style="list-style-type: none"> • <code>BaseAction.class</code> • <code>CancelAction.class</code> • <code>DisplayAction.class</code> • <code>EditorAction.class</code> • <code>ErrorHandler.class</code> • <code>Initializer.class</code> • <code>OkAction.class</code>
/DisplayURL/content	This directory contains the display pages called <code>Editor.jsp</code> , <code>Error.jsp</code> , and <code>Viewer.jsp</code> . Each portlet can have only one content location directory, and the directory name must be <code>content</code> . However, the content location directory can have an unlimited number of subdirectories.

The following rules apply when you set up the directory structure:

- Neither portlet names nor their paths can contain spaces.
- The portlet identifier (which consists of the name and the path) must be unique. Developers should devise a convention to ensure unique name–spaces, similar to the conventions used for naming Java packages.

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Step 2: Create the Portlet Deployment Descriptor

The portlet deployment descriptor is an XML file that provides all of the information that the portal Web application needs to deploy one or more portlets. Here is the portlet deployment descriptor for the DisplayURL portlet. The boxes contain explanatory comments. For more information, see [Creating a Portlet Deployment Descriptor](#).

```
<?xml version="1.0" encoding="UTF-8"?>
```

The DOCTYPE statement must be present in the descriptor file in order for the portlet to run. However, the document type definition (DTD) does not need to be accessible at the URL that the statement specifies.

If you want to look at the `portlet.dtd` file, you can find it in the portal setup directory in the path `Portal\WEB-INF`. For example, if you used the default install location on a Windows system, then the DTD is located under the following path: `c:\Program Files\SAS\Web\Portal2.0.1\Portal\WEB-INF`.

```
<!DOCTYPE portlets SYSTEM "http://www.sas.com/ldap/portlet.dtd">
```

```
<portlets>
```

In the `local-portlet` element, the value of the `title` attribute specifies the new portlet type that will be displayed to users in the Create a New Portlet dialog box.

The attribute `editorType="portlet"` indicates that portlets created from the template are to be editable. When this attribute value is specified, then a portlet action with the attribute `editor="true"` must also be specified. Otherwise, the portlet deployer will send a warning to the server log and will not deploy the portlet.

```
<local-portlet name="DisplayURL" title="URL Display Portlet"
  editorType="portlet">
  <localized-resources locales="en,de,es,fr,it,ja,pl,ru,sv,zh_CN" />
```

The deployment element includes the attribute value `userCanCreateMore="true"`. This value indicates that the portlet is a template and can be replicated by portal users.

```
    <deployment scope="user" autoDeploy="false"
      userCanCreateMore="true" />
    <initializer-type>
      com.sas.portal.portlets.displayurl.Initializer
    </initializer-type>
    <init-param>
      <param-name>error-page</param-name>
      <param-value>Error.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>display-page</param-name>
      <param-value>Viewer.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>edit-page</param-name>
      <param-value>Editor.jsp</param-value>
    </init-param>
    <error-handler>
      <type>com.sas.portal.portlets.displayurl.ErrorHandler</type>
    </error-handler>
```

```
<portlet-path>/sas/portlets</portlet-path>
<portlet-actions>
```

This action element specifies the action class `DisplayAction`. The attribute `default="true"` indicates that this is the default action class, which means that the class is to be invoked before the portlet's JSP renders.

```
<portlet-action name="display" default="true" >
  <type>
    com.sas.portal.portlets.displayurl.DisplayAction
  </type>
</portlet-action>
```

This action element specifies the action class `EditorAction`. The attribute `editor="true"` indicates that this action is to be invoked when a user clicks the portlet's Edit icon.

```
<portlet-action name="editor" editor="true" >
  <type>
    com.sas.portal.portlets.displayurl.EditorAction
  </type>
</portlet-action>
```

The following action elements specify the action classes that will be invoked when the user clicks the **OK** button and the **Cancel** button on the editor display page.

```
<portlet-action name="ok" default="false" >
  <type>com.sas.portal.portlets.displayurl.OKAction</type>
</portlet-action>
<portlet-action name="cancel" default="false" >
  <type>com.sas.portal.portlets.displayurl.CancelAction</type>
</portlet-action>
</portlet-actions>
</local-portlet>
</portlets>
```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Step 3: Create the Display Pages for the Portlet and the Editor

The DisplayURL portlet has three JSP pages:

- Viewer.jsp, which is the presentation component of the portlet
- Editor.jsp, which is the presentation component of the editor action
- Error.jsp, which displays messages for errors that occur during the editing process

To see the code for these JSP pages, click on the links shown previously.

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Viewer.jsp

The code for `Viewer.jsp`, which is the presentation component of the `DisplayURL` portlet, follows.

```
<!-- Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513 -->
<%@ page language="java" contentType= "text/html; charset=UTF-8" %>
<%@ page import="com.sas.portal.portlet.PortletContext" %>
<%@ page import="com.sas.portal.portlet.PortletConstants" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c_rt" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt_rt" prefix="fmt_rt" %>

<%
    PortletContext context = (PortletContext) request.getAttribute(
        PortletConstants.CURRENT_PORTLET_CONTEXT );
    String url = (String) context.getAttribute("sas_DisplayURL_DisplayURL");

    if ((url == null) || (url.length() == 0))
    {
%>
<p style="text-align: center;"><fmt:message key="viewer.nourl.txt" /></p>
<%
    } else {
        try {
%>
<c_rt:import charEncoding="UTF-8" url="<%= url %%" />
<%
        } catch (Exception ex) {
%>
<p style="text-align: center;">
    <fmt_rt:message key="viewer.badurl.fmt">
        <fmt_rt:param value="<%= url %%" />
        <fmt_rt:param value="<%= ex.getMessage() %%" />
    </fmt_rt:message>
</p>
<%
    }
}
%>
```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Editor.jsp

The code for `Editor.jsp`, which is the presentation component of the editor for the `DisplayURL` portlet, follows.

```
<!-- Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513 -->
<%@ page language="java" contentType= "text/html; charset=UTF-8" %>
<%@ page import="com.sas.portal.portlet.PortletContext" %>
<%@ page import="com.sas.portal.portlet.PortletConstants" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>

<% PortletContext context = (PortletContext) request.getAttribute(
    PortletConstants.CURRENT_PORTLET_CONTEXT ); %>

<table border="0" cellpadding="2" cellspacing="0" align="center"
width="100%">

<form method="post" action="<%= context.getAttribute(
"sas_DisplayURL_EditOkURL") %>">






</td>


```

</table>

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Error.jsp

The code for `Error.jsp`, which displays messages for any errors that occur during the editing process, follows.

```
<!-- Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513 --%>
<%@ page language="java" contentType= "text/html; charset=UTF-8" %>
<%@ page import="com.sas.portal.portlet.PortletContext" %>
<%@ page import="com.sas.portal.portlet.PortletConstants" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>

<% PortletContext context = (PortletContext)request.getAttribute(
    PortletConstants.CURRENT_PORTLET_CONTEXT ); %>

<fmt:message key="error.msg1.txt"/>
<br>
<%= context.getAttribute( "Exception_message" ) %>
```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Step 4: Create the Action Classes

The DisplayURL portlet has the following action classes:

- [Initializer](#)
- [BaseAction](#)
- [Display](#)
- [EditorAction](#)
- [OkAction and CancelAction](#)
- [ErrorHandler](#)

To see the source code for these action classes, click on the links shown previously.

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Initializer Action

The DisplayURL portlet's `Initializer` action class initializes properties that are used by the other action classes and puts the properties into a `PortletContext` object. The source code follows.

```
/**Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.displayurl;

import java.rmi.RemoteException;
import java.util.Properties;

import com.sas.portal.Logger;
import com.sas.portal.portlet.PortletContext;
import com.sas.portal.portlet.PortletInitializerInterface;
import com.sas.portal.portlet.configuration.Attribute;
import com.sas.portal.portlet.configuration.Configuration;
import com.sas.portal.portlet.configuration.ConfigurationFactory;

/**
 * This initializes common properties by putting them into a
 * PortletContext object.
 */
public class Initializer implements PortletInitializerInterface
{
    private final String _loggingContext = this.getClass().getName();

    /** Key for the URL String in the PortletContext.*/
    public static final String DISPLAY_URL_KEY =
        "sas_DisplayURL_DisplayURL";

    /** PortletContext key for the edit screen Ok button URL */
    public static final String EDIT_OK_URL_KEY =
        "sas_DisplayURL_EditOkURL";

    /** PortletContext key for the edit screen Cancel button URL */
    public static final String EDIT_CANCEL_URL_KEY =
        "sas_DisplayURL_EditCancelURL";

    /** Key for the PortletException object in the PortletContext.*/
    public static final String PORTLET_EXCEPTION_KEY =
        "sasPortletException";

    /**
     * Puts initial properties into the PortletContext object. These
     * come from the portlet.xml.
     * @param initProperties a Properties object
     * @param context the PortletContext for this portlet
     */
    public void initialize(Properties initProperties,
        PortletContext context)
    {
        try {
            // get the initial url from the portlet configuration object
            Configuration config = ConfigurationFactory.getConfiguration(
                context);
            Attribute attr = config.getAttribute(
```

```

        Initializer.DISPLAY_URL_KEY);
String url = (attr == null) ? "" : attr.getValue();

context.setAttribute("error-page",
    initProperties.getProperty("error-page"));
context.setAttribute("display-page",
    initProperties.getProperty("display-page"));
context.setAttribute("edit-page",
    initProperties.getProperty("edit-page"));
context.setAttribute(Initializer.DISPLAY_URL_KEY, url);

if (Logger.isDebugEnabled(_loggingContext)){
    Logger.debug("Display portlet URL: " +
        url, _loggingContext);
}
    } catch ( RemoteException e) {
context.setAttribute(Initializer.PORTLET_EXCEPTION_KEY, e);
    }
}
}

```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Base Action

The DisplayURL portlet's BaseAction class is a superclass which is extended by the DisplayAction, EditorAction, OkAction, and CancelAction classes. The source code is shown below.

```
/* Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513. All Rights Reserved. */
package com.sas.portal.portlets.displayurl;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sas.portal.Logger;
import com.sas.portal.container.deployment.PortletActionInfoInterface;
import com.sas.portal.portlet.NavigationUtil;
import com.sas.portal.portlet.PortletActionInterface;
import com.sas.portal.portlet.PortletContext;
import com.sas.portal.portlets.displayurl.Initializer;

public abstract class BaseAction implements PortletActionInterface
{
    private final String _loggingContext = this.getClass().getName();

    private PortletActionInfoInterface _actionInfo = null;

    /**
     * This method must be overridden in subclasses. They must call super and
     * supply a return value.
     * In this class, the method returns null.
     *
     * @see com.sas.portal.portlet.PortletActionInterface#service(HttpServletRequest,
     *      HttpServletResponse, PortletContext)
     */
    public String service(HttpServletRequest request,
                        HttpServletResponse response,
                        PortletContext context) throws Exception
    {
        Logger.debug("started..", _loggingContext);
        response.setContentType("text/html; charset=UTF-8");

        // prepare the localized resources for use by the jsp.
        try {
            NavigationUtil.prepareLocalizedResources(
                "com.sas.portal.portlets.displayurl.Resources", request,
                context);
        }
        catch (java.io.IOException ioe) {
            Logger.error(ioe.getMessage(), _loggingContext, ioe);
        }

        return null;
    }

    /**
     * @see com.sas.portal.portlet.PortletActionInterface#setInfo(PortletActionInfoInterface)
     */
    public final void setInfo(PortletActionInfoInterface info)
    {
        _actionInfo = info;
    }
}
```

```

/**
 * @see com.sas.portal.portlet.PortletActionInterface#getInfo()
 */
public final PortletActionInfoInterface getInfo()
{
    return _actionInfo;
}

/**
 * Check the PortletContext for an exception object. If present, throw it so
 * that the error handler will kick in.
 * @param context the PortletContext
 */
protected static final void errorCheck(PortletContext context) throws Exception
{
    Exception e = (Exception)context.getAttribute(Initializer.PORTLET_EXCEPTION_KEY);
    if (e != null)
        throw e;
}
}

```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Display Action

The `DisplayAction` class is the default action class for the `DisplayURL` portlet. This means that the class is to be invoked before the portlet's JSP page renders. The source code follows.

```
/**Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.displayurl;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
//import com.sas.portal.Logger;
import com.sas.portal.portlet.PortletContext;

/**
 * Action class that presents the display page. It sets up the display
 * model then instructs the portlet container to present the display
 * page.
 */
public final class DisplayAction extends BaseAction
{
    // private final String _loggingContext = this.getClass().getName();

    /**
     * Service the portlet request.
     *
     * @param request the HttpServletRequest
     * @param response the HttpServletResponse
     * @param context the PortletContext
     * @return the URL to call
     */
    public String service(HttpServletRequest request,
                        HttpServletResponse response,
                        PortletContext context) throws Exception
    {
        super.service(request, response, context);

        // see if there is an initialization error
        errorCheck(context);
        return (String)context.getAttribute("display-page");
    }
}
```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Editor Action

The DisplayURL portlet's EditorAction class is invoked when a user clicks the portlet's Edit icon. The source code follows.

```
/**Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.displayurl;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

//import com.sas.portal.Logger;
import com.sas.portal.portlet.PortletContext;
import com.sas.portal.portlet.NavigationUtil;

/**
 * Action class that presents the edit page. It sets up the edit model
 * then instructs the portlet container to present the edit page.
 */
public final class EditorAction extends BaseAction
{
    // private final String _loggingContext = this.getClass().getName();

    /**
     * Service the portlet request.
     *
     * @param request the HttpServletRequest
     * @param response the HttpServletResponse
     * @param context the PortletContext
     * @return the URL to call
     */
    public String service(HttpServletRequest request,
                        HttpServletResponse response,
                        PortletContext context) throws Exception
    {
        super.service(request, response, context);

        //create the URLs for the OK and Cancel buttons.
        String url;

        url = NavigationUtil.buildBaseURL(context, request,
            "ok");
        context.setAttribute(Initializer.EDIT_OK_URL_KEY, url);

        url = NavigationUtil.buildBaseURL(context, request,
            "cancel");
        context.setAttribute(Initializer.EDIT_CANCEL_URL_KEY, url);

        // the following call resets the mode back to display for the
        // next call
        context.resetMode();

        return (String) context.getAttribute("edit-page");
    }
}
```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

OK and Cancel Actions

The DisplayURL portlet's `OkAction` class is invoked when a user clicks the **OK** button on the editor display page. The `CancelAction` class is invoked when a user clicks the **Cancel** button on the editor display page. The source code for both classes follows.

```
/**Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.displayurl;

import com.sas.portal.portlet.configuration.ConfigurationFactory;
import com.sas.portal.portlet.configuration.Configuration;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.sas.portal.Logger;
import com.sas.portal.portlet.PortletContext;

/**
 * Action class that processes the Ok action from the editor. It persists
 * the user-specified URL, sets up the display model, then instructs the
 * portlet container to present the display page.
 */
public final class OKAction extends BaseAction
{
    private final String _loggingContext = this.getClass().getName();

    /**
     * Service the portlet request.
     *
     * @param request the HttpServletRequest
     * @param response the HttpServletResponse
     * @param context the PortletContext
     * @return the URL to call
     */
    public String service (HttpServletRequest request,
                          HttpServletResponse response,
                          PortletContext context) throws Exception
    {
        super.service(request, response, context);

        String url = request.getParameter(Initializer.DISPLAY_URL_KEY);
        context.setAttribute(Initializer.DISPLAY_URL_KEY, url);

        // save the URL parameter
        Configuration config = ConfigurationFactory.getConfiguration(
            context);
        config.setAttribute(Initializer.DISPLAY_URL_KEY, url);
        ConfigurationFactory.storeConfiguration(context, config);

        if (Logger.isDebugEnabled(_loggingContext)){
            Logger.debug("Display portlet URL: " + url, _loggingContext);
        }

        // back to the default, display, mode
        context.resetMode();

        return (String)context.getAttribute("display-page");
    }
}
```

```

    }
}



---


/** Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.displayurl;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
//import com.sas.portal.Logger;
import com.sas.portal.portlet.PortletContext;

/**
 * Action class that processes the Cancel action from the editor. It sets
 * up the display model then instructs the portlet container to present
 * the display page.
 */
public final class CancelAction extends BaseAction
{
    // private final String _loggingContext = this.getClass().getName();

    /**
     * Service the portlet request.
     *
     * @param request the HttpServletRequest
     * @param response the HttpServletResponse
     * @param context the PortletContext
     * @return the URL to call
     */
    public String service(HttpServletRequest request,
                        HttpServletResponse response,
                        PortletContext context) throws Exception
    {
        super.service(request, response, context);

        // back to the default, display, mode
        // context.resetMode();
        return (String)context.getAttribute("display-page");
    }
}

```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Error Handler Action

The source code for the DisplayURL portlet's ErrorHandler action class follows.

```
/** Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513.
 * All Rights Reserved.
 */
package com.sas.portal.portlets.displayurl;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sas.apps.portal.PortalException;
import com.sas.portal.Logger;
import com.sas.portal.portlet.ErrorHandlerInterface;
import com.sas.portal.portlet.NavigationUtil;
import com.sas.portal.portlet.PortletContext;
import com.sas.portal.preferences.PortalPreferences;

/**
 * Error handler for this portlet. It logs the exception and returns
 * ErrorPage.jsp for the portlet to display.
 */
public class ErrorHandler implements ErrorHandlerInterface
{
    private final String _loggingContext = this.getClass().getName();

    /**
     * Returns the URL for the portlet controller to call. This is
     * the name of
     * the error page JSP.
     * @param request the HttpServletRequest
     * @param response the HttpServletResponse
     * @param context the PortletContext
     * @param exception the exception thrown by a portlet action
     * @return the URL to call
     */
    public String service(HttpServletRequest request,
                        HttpServletResponse response,
                        PortletContext context,
                        Exception thrownException)
    {
        //prepare the localized resources for use by the jsp.
        try {
            NavigationUtil.prepareLocalizedResources(
                "com.sas.portal.portlets.displayurl.Resources",
                request, context);
        }
        catch (java.io.IOException ioe) {
            Logger.error(ioe.getMessage(), _loggingContext, ioe);
        }

        //send error to server log in default locale.
        Logger.error(thrownException.getMessage(), _loggingContext,
            thrownException);

        //get msg in user's locale.
        String msg = null;
        try {
```

```

        PortalException ourException = (PortalException)
            thrownException;
        msg = ourException.getMessage(PortalPreferences.getLocale(
            context.getHttpSession()));
    }
    catch (ClassCastException cce){
        msg= "";
    }

    if (msg == null) {
        //prevent showing the word null in a JSP
        msg = "";
    }

    //make msg available for display on error jsp.
    context.setAttribute("Exception_message", msg);

    return (String)context.getAttribute("error-page");
}
}

```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Step 5: Create the Resource Bundle

The resource bundles provide translated text to be displayed inside the DisplayURL portlet. The portlet's [BaseAction](#), [EditorAction](#), and [ErrorHandler](#) classes call the `NavigationUtil.prepareLocalizedResources()` method to create a JSTL localization context based on the user's locale preference. This context enables the JSTL tags in the portlet's JSP pages to use the appropriate resource bundle to display text.

For more information about creating resource bundles, see [Creating a Localized Portlet](#).

Note: For information about localizing a portlet's title and description, see [Creating Display Resources Files](#).

One resource bundle is provided with the DisplayURL portlet, as follows.

Note: If you copy and paste this code, then you must remove the line breaks in the message strings for `error.msg1.txt` and `viewer.nourl.txt`.

```
# Messages for the DisplayURL portlet

# NOTE: this is the same message text as found in
#       com.sas.portal.res.Resources.properties. The localized versions
#       from there can be used here.
error.msg1.txt=A serious error occurred. Contact the Portal
              administrator.

# {0} will be a URL. {1} will be an exception message.
viewer.badurl.fmt=Unable to display ''{0}'' because ''{1}''
viewer.nourl.txt=No URL has been specified. Please edit the portlet to
                set a URL.

editor.task.txt=Enter the URL of the HTML fragment to display.
editor.url.txt=URL:

# NOTE: these are the same messages as found in
#       com.sas.portal.res.Resources.properties. The localized versions
#       from there can be used here.
editor.action.cancel.txt=Cancel
editor.action.ok.txt=OK
```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Step 6: Create the Display Resources File

The sample remote portlet DisplayURL uses a display resources file to provide a description to be placed in the portlet's metadata for display to users.

You can supply multiple display resources files if you want the portal Web application to localize the portlet title and description at the time of deployment, according to the portal Web application's default locale. For more information, see [Creating Display Resources Files](#).

The DisplayURL portlet has one display resources file, which is named `portletDisplayResources.properties`. The contents of this file follow.

```
portlet.title=URL Display Portlet
portlet.description=Portlet that displays the contents of a URL
```

Sample: Portlet Template, or Editable Portlet (DisplayURL)

Step 7: Create the PAR File, and Deploy and Test the Portlet

The last step in developing the DisplayURL portlet was to archive its files into a PAR file. The PAR file includes

- appropriately named and organized directories and subdirectories, as described in [Step 1: Create the Directory Structure](#).
- all of the portlet's supporting files, including the files created in Steps 2 through 6. The files must be placed in the appropriate directories as described in [Step 1: Create the Directory Structure](#) and [Creating a PAR File for Deployment in Your Application](#).

The JAR utility was used to compress the directories and files into an archive, and the archive was given the name SAS_DisplayURL.par.

It is a good practice to deploy new portlets into a staging area (that is, a test installation of the portal Web application) for verification and testing before deploying them into the production environment. For information about how to deploy a PAR file into the portal Web application, see [Portlet Deployment](#) in the SAS Web Infrastructure Kit Administrator's Guide.

Sample: Web Application (HelloUserWikExample)

Sample: Web Application (HelloUserWikExample)

The HelloUserWikExample application is a Web application that displays the string **Hello 'user'**, where *user* is the name of the user who is logged on to the portal Web application, as shown in this example:

```
Hello 'Portal Demo'
```

The HelloUserWikExample application is

- **enabled by SAS Foundation Services.** The application uses SAS Foundation Services to access session information created by the portal Web application, extracts the user's name from the session information, inserts the name in a message, and displays the message to the user.
- **deployable as either a remote portlet or a stand-alone application.** The [HelloUserRemotePortlet](#) sample shows how this application can be deployed in a remote portlet.

If an application is called from a remote portlet, then it must generate an HTML fragment to be displayed within the portlet borders. If an application is to be invoked as a stand-alone application, then its JSP page must generate a complete HTML file (with starting and ending <HTML> tags.) The JSP page for the HelloUserWikExample contains conditional code that determines whether the application request was generated by a portlet. The JSP page then generates the appropriate type of HTML.

Note: The purpose of the HelloUserWikExample application is to show how you can use the APIs to access the shared session context. It is not intended to illustrate best programming practices.

The following steps were used to create the application:

1. [Create a directory structure](#) for the application.
2. [Create the Web application deployment descriptor](#) (web.xml).
3. [Create deployment definitions and properties files for local and remote services.](#)
4. [Create the display page](#) (app.jsp).
5. [Create the WAR file, and deploy and test the application.](#)

Sample: Web Application (HelloUserWikExample)

Step 1: Create the Directory Structure

The following directory structure was used to create the HelloUserWikExample application:



This directory structure will be used in [Step 5](#) to create a WAR file for the application. The structure includes the following directories and subdirectories:

Directory	Contents
app_work (root)	This directory serves as a development area for the Web application.
/jsp	This directory contains the <u>display page</u> called <code>app.jsp</code> .
/WEB-INF	This directory contains the <u>Web application deployment descriptor</u> file. The name of this file must be <code>web.xml</code> .
/WEB-INF/conf	This directory contains the <u>services properties files</u> , which point to the locations of definitions for locally and remotely deployed SAS Foundation Services.
/WEB-INF/lib	<p>This directory contains the JAR files that are used by the JSP page. The following JAR files must be provided at a minimum:</p> <ul style="list-style-type: none"> • <code>log4j-1.2.3.jar</code> • <code>mail.jar</code> • <code>sas.core.jar</code> • <code>sas.entities.jar</code> • <code>sas.iquery.metadata.jar</code> • <code>sas.oma.joma.jar</code> • <code>sas.oma.joma.rmt.jar</code> • <code>sas.oma.omi.jar</code> • <code>sas.portal.metadata.jar</code> • <code>sas.report.repository.jar</code> • <code>sas.svc.bootstrap.jar</code> • <code>sas.svc.connection.jar</code> • <code>sas.svc.connection.platform.jar</code> • <code>sas.svc.core.jar</code> • <code>sas.svc.events.jar</code> • <code>sas.svc.publish.jar</code> • <code>sas.svc.sec.login.jar</code> • <code>sas.svc.storedprocess.jar</code> • <code>sas.svc.webdav.jar</code> • <code>sas.text.jar</code> • <code>sas.web.framework.jar</code>

Sample: Web Application (HelloUserWikExample)

Step 2: Create the Web Application Deployment Descriptor

The Web application deployment descriptor is an XML file that describes the Web application's initialization parameters, servlets, and other components. Here is the Web application deployment descriptor for the HelloUserWikExample application. The boxes contain explanatory comments.

For more information about creating Web application deployment descriptors, see the documentation for your servlet container.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
  Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
```

The servlet entry specifies the SAS Foundation Services Bootstrap servlet. This servlet provides a convenient way to deploy local services for the Web application to use. For more information, see `com.sas.services.webapp` in the Portlet API [class documentation](#).

The `localPropsfile` parameter specifies the name of the properties file that is to be used for local services deployment, and the `remotePropsFile` parameter specifies the name of the properties file that is to be used to access remote services. The referenced files must be present in the path `/WEB-INF/conf` in the Web application's [directory structure](#). For more information, see [Step 3: Create Deployment Definitions and Properties Files for Local and Remote Services](#).

```
<!-- BEGIN BootstrapServlet -->
<servlet>
  <servlet-name>BootstrapServlet</servlet-name>
  <servlet-class>
    com.sas.services.webapp.BootstrapServlet
  </servlet-class>
  <init-param>
    <!-- metadata source for local deployable services -->
    <param-name>localPropsFile</param-name>
    <param-value>
      sas_metadata_source_client_omr.properties
    </param-value>
  </init-param>
  <init-param>
    <!-- metadata source to retrieve remotely deployable services -->
    <param-name>remotePropsFile</param-name>
    <param-value>
      sas_metadata_source_server_omr.properties
    </param-value>
  </init-param>
  <init-param>
    <param-name>loggingURL</param-name>
    <param-value>
      file:///C:\Program Files\SAS\SAS Foundation
        Services\Deployments\Portal\logging_config_idp.xml
    </param-value>
  </init-param>
  <init-param>
    <param-name>SystemPropsFile</param-name>
    <param-value>
```

SAS® Web Infrastructure Kit 1.0: Developer's Guide

```
C:\Program Files\SAS\SAS Foundation
  Services\Deployments\Portal\system_properties.config
</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<!-- END BootstrapServlet -->
<servlet-mapping>
  <servlet-name>BootstrapServlet</servlet-name>
  <url-pattern>/Bootstrap</url-pattern>
</servlet-mapping>
</web-app>
```

Sample: Web Application (HelloUserWikExample)

Step 3: Create Deployment Definitions and Properties Files for Local and Remote Services

The HelloUserWikExample application requires access to both local and remote services. The following files are required to enable access to the services: a local services deployment definition, a local services properties file, a remote services deployment definition, and a remote services properties file, as described below.

Local Services Deployment Definition

The HelloUserWikExample application requires a definition file for local services deployment. This file must be installed in the SAS Foundation Services directory on the machine where the HelloUserWikExample application will be installed.

The contents of the HelloUserWikExample application's local services deployment definition can be exactly the same as the definition that is used by the portal Web application. The portal Web application's local deployment definition is called `sas_services_idp_local_omr.xml`, and it generally can be found in the path `C:\Program Files\SAS\SASFoundationServices\Deployments\Portal\`.

Follow these steps to create the local services deployment definition:

- a. On the machine where the portal Web application is installed, copy the portal Web application's local deployment definition file (`sas_services_idp_local_omr.xml`) to a separate file. Give the new file a different name (for example, `sas_services_hellouser_local_omr.xml`).
- b. On the machine where the HelloUserWikExample application will be installed:
 - i. If you are installing the HelloUserWikExample application on a different machine than the portal Web application, and if SAS Foundation Services is not installed on this machine, then create the following directory structure: `C:\Program Files\SAS\SASFoundationServices\Deployments\`. (**Note:** It is not necessary to install SAS Foundation Services on this machine.)
 - ii. Create a new directory under `C:\Program Files\SAS\SASFoundationServices\Deployments\`. For example:

```
C:\Program
Files\SAS\SASFoundationServices\Deployments\HelloUserWikExample\
```

Then place the new deployment definition file in the new directory.

Note: As a best practice, Web applications should not share the same local services deployment definition. Therefore, you should follow these steps even if the HelloUserWikExample application will be installed on the same machine as the portal Web application.

Local Services Properties File

The local services properties file (`sas_metadata_source_client_omr.properties`) tells the HelloUserWikExample application where to find the definition file for local service deployment. This file must be placed in the path `/WEB-INF/conf` in the Web application's directory structure.

To create this file, you can do the following:

- a. Copy the portal Web application's local services properties file (`sas_metadata_source_client_omr.properties`) to a separate file with the same name.
- b. Edit the new file to specify a URL with the actual location and file name of the local services deployment definition that you created for the HelloUserWikExample application.
- c. Place the new file in the path `/WEB-INF/conf` in the HelloUserWikExample application's directory structure.

Here is a sample of a local services properties file:

```
software_component=ID Portal Local Services
deployment_group_1=BIP Local Services OMR

type=URL
url=file:///C:/Program Files/SAS/SASFoundationServices/Deployments/
    HelloUserWikExample/sas_services_hellouser_local_omr.xml
```

Note: The line break in the `url` property statement is for display purposes only. If you copy this example into your properties file, you must put the entire statement on one line. In addition, be sure to update the URL with the actual location and file name of the local services deployment definition that you created for the HelloUserWikExample application.

Remote Services Deployment Definition

The Web application also requires a definition file for remote services deployment. This file must be installed in the SAS Foundation Services directory on the machine where the HelloUserWikExample application will be installed.

If the HelloUserWikExample application will be installed on the same machine as the portal Web application, then it can use the same remote services deployment definition (`sas_services_idp_remote_omr.xml`) that the portal Web application uses. It is not necessary, nor is it recommended, to make another copy of this file for the HelloUserWikExample application.

If the HelloUserWikExample application will be installed on a different machine from the portal Web application, then follow these steps to create the remote services deployment definition:

- a. On the machine where the portal Web application is installed, copy the portal Web application's remote deployment definition file (`sas_services_idp_remote_omr.xml`) to a separate file with the same name.
- b. On the machine where the HelloUserWikExample application will be installed, place the new deployment definition file in the directory where the new local deployment definition file is located. For example:

```
C:/Program
    Files/SAS/SASFoundationServices/Deployments/HelloUserWikExample/
```

Remote Services Properties File

The remote services properties file (`sas_metadata_source_server_omr.properties`) tells the HelloUserWikExample application where to find the definition file for the remote services. The file must be placed in the path `/WEB-INF/conf` in the HelloUserWikExample application's directory structure.

To create this file, you can do the following:

- a. Copy the portal Web application's remote services properties file (`sas_metadata_source_server_omr.properties`) to a separate file with the same name.
- b. If necessary, edit the new file to specify a URL with the actual location and file name of the remote services deployment definition that you created for the HelloUserWikExample application. If these are the same as that used by the portal Web application, then you can skip this step.
- c. Place the new file in the path `/WEB-INF/conf` in the HelloUserWikExample application's directory structure.

Here is a sample of a remote services properties file:

```
software_component=Remote Services
deployment_group_1=BIP Remote Services OMR

type=URL
url=file:///C:/Program Files/SAS/SASFoundationServices/Deployments/
    RemoteServices/sas_services_idp_remote_omr.xml
```

Note: The line break in the `url` property statement is for display purposes only. If you copy this example into your properties file, you must put the entire statement on one line. In addition, be sure that the URL contains the actual location of your remote services deployment file.

Sample: Web Application (HelloUserWikExample)

Step 4: Create the Display Page (JSP)

The JSP page for the HelloUserWikExample application is called `app.jsp`. The scriptlet code in this JSP page uses methods from SAS Foundation Services to obtain the user's name from the portal Web application session. The user's name is then inserted into the text on the display page.

The code for `app.jsp` follows. The boxes contain explanatory comments.

```
<!-- Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513 --%>
<%@ page language="java" contentType= "text/html; charset=UTF-8" %>
<%@ page import="com.sas.webapp.contextsharing.WebappContextParams" %>
<%@ page import="com.sas.services.session.SessionContextInterface" %>
<%@ page import="com.sas.services.user.UserContextInterface" %>
<%@ page import="java.util.Enumeration" %>
<%@ page import="java.util.HashMap" %>

<%
try {
```

In the following code, a new `WebappContextParams` object is created to obtain the session key from the portal Web application request and to obtain a reference to the portal Web application's remotely deployed services.

```
// Use WebappContextParams to obtain information passed from the
// calling BIA application
WebappContextParams params = new WebappContextParams(request, true);

// See if we are being requested to display as a portlet
```

The following code determines whether a portlet ID is available. The application uses this information to determine whether to return a complete HTML page (for display alone in a browser window) or an HTML fragment (for display within a portlet).

```
boolean displayAsPortlet = (params.getPortletid() == null) ? false :
    true;

SessionContextInterface sharedSession = null;
Object sessionLock = null;
String user = null;

// Get the name of the user from the BIA shared session context
try {
```

The following code obtains access to the portal Web application's remotely deployed services. The session is protected with a lock.

```
sharedSession = params.getSessionContext();
if (sharedSession != null) {
    sessionLock = sharedSession.lock("com.sas.HelloUserWikExample");
```

The following code obtains the user's name from the user context object in the remote services session.

```
UserContextInterface userContext = sharedSession.getUserContext();
user = userContext.getName();
} else {
    user = "unknown";
```

```

    }
} catch (Throwable thr2) {
    thr2.printStackTrace();
} finally {

```

The following code removes the lock.

```

// Unlock the shared BIA session

try {
    if ((sharedSession != null) &(sessionLock != null))
        sharedSession.unlock(sessionLock);
} catch (Throwable thr3) {
    // Non-fatal
}
}

```

The following code determines how to display the application output. If the application was called by a portlet, then it generates an HTML fragment for display within the portlet. If the application was called as a stand-alone application, then it generates a complete HTML page to be displayed alone in a browser window.

```

if (displayAsPortlet == false) {
    // Called as a web application
%>

<html>
<head>
    <title>Hello User Remote Application</title>
</head>
<body>
<p>Hello '<%= user %>'.</p>
</body>
</html>

<%
    } else {
        // Called as either a web application or portlet
%>
<p>Hello '<%= user %>'.</p>
<%
    }
} catch (Throwable thr1) {
    thr1.printStackTrace();
}
%>

```

Sample: Web Application (HelloUserWikExample)

Step 5: Create the WAR File, and Deploy and Test the Application

The last step in developing the HelloUserWikExample application was to archive its files into a WAR file. The WAR file includes

- appropriately named and organized directories and subdirectories, as described in [Step 1: Create the Directory Structure](#).
- all of the application's supporting files, including the files created in Steps 2 through 4 and the JAR files required for SAS Foundation Services. The files must be placed in the appropriate directories as described in [Step 1: Create the Directory Structure](#).

The JAR utility was used to compress the directories and files into an archive, and the archive was given the name `HelloUserWikExample.war`.

Before deploying the HelloUserWikExample application in the portal Web application, you should test it using its direct URL. You can then deploy it in either of the following ways:

- as an application that is called by a remote portlet. The sample [HelloUserRemotePortlet](#) describes the steps for implementing this application as a remote portlet.
- as a stand-alone application, which appears in the portal Web application to be the same as a link. For information about how to add the metadata for a stand-alone application that is enabled by SAS Foundation Services, see [Adding Web Applications](#) in the *SAS Web Infrastructure Kit Administrator's Guide*.

Sample: Remote Portlet (HelloUserRemotePortlet)

Sample: Remote Portlet (HelloUserRemotePortlet)

The following sample portlet, called HelloUserRemotePortlet is a remote portlet:



This portlet calls the Web application HelloUserWikExample. The HelloUserRemotePortlet application displays the string `Hello user`, where *user* is the name of the user who is logged on to the portal Web application. For details about the HelloUserRemotePortlet application, see [Sample Web Application \(HelloUserWikExample\)](#).

The following steps were used to create the HelloUserRemotePortlet. Click on each step to display details.

1. [Create a directory structure](#) for the portlet.
2. [Create the portlet deployment descriptor](#) (portlet.xml).
3. [Create the display resources file.](#)
4. [Create the Web application.](#)
5. [Create the PAR file, and deploy and test the portlet.](#)

Sample: Remote Portlet (HelloUserRemotePortlet)

Step 1: Create the Directory Structure

The following directory structure was used to create the portlet called HelloUserRemotePortlet.



This structure includes the following directories and subdirectories:

Directory	Contents
portlet_work (root) This directory serves as a development area for the portlet.	<u>Portlet deployment descriptor file</u> <code>portlet.xml</code> The name of the deployment descriptor file must be <code>portlet.xml</code> .
/HelloUserRemotePortlet	This is the main portlet directory. It does not contain any files. The directory name must not have any spaces, and it must match the name of the portlet as specified in the name attribute of the <code><remote-portlet></code> element in <code>portlet.xml</code> .
/FormExample/classes	The <u>display resources file</u> called <code>portletDisplayResources.properties</code> .

The following rules apply when you set up the directory structure:

- Neither portlet names nor their paths can contain spaces.
- The portlet identifier (which consists of the name and the path) must be unique.

Note: Developers should devise a convention to ensure unique name–spaces, similar to the conventions used for naming Java packages. For example, the Sales division of a company named ABCD could create portlets in the path `ABCD/Sales`, and the Purchasing division could create portlets in the path `ABCD/Purchasing`. Then both Sales and Purchasing could have different portlets named `HelloUserRemotePortlet`.

Note: You must create a separate directory structure for the Web application that is called by the remote portlet, as described in the [Sample Web Application \(HelloUserWikExample\)](#). *Sample: Remote Portlet (HelloUserRemotePortlet)*

Step 2: Create the Portlet Deployment Descriptor

The portlet deployment descriptor is an XML file that provides all of the information that the portal Web application needs to deploy one or more portlets. Here is the portlet deployment descriptor for the HelloUserRemotePortlet. The boxes contain explanatory comments. For more information, see [Creating a Portlet Deployment Descriptor](#).

```
<?xml version="1.0" encoding="UTF-8"?>
```

The DOCTYPE statement must be present in the descriptor file in order for the portlet to run. However, the document type definition (DTD) does not need to be accessible at the URL that the statement specifies.

If you want to look at the portlet.dtd file, you can find it in the portal setup directory in the path Portal\WEB-INF. For example, if you used the default install location on a Windows system, then the DTD is located under the following path: c:\Program Files\SAS\Web\Portal2.0.1\Portal\WEB-INF.

```
<DOCTYPE portlets SYSTEM "http://www.sas.com/idp/portlet.dtd">
```

```
<portlets>
```

The remote-portlet element assigns the name HelloUserRemotePortlet to the portlet. The name cannot contain spaces. The portlet identifier, which consists of the portlet path (defined in the portlet-path element) together with the portlet name, must be unique within the portal Web application.

The "true" setting for the passContextId attribute makes the portal Web application session information, including user identity, available to the remote portlet.

```
<remote-portlet name="HelloUserRemotePortlet"
  title="HelloUserRemotePortlet" passContextId="true">
  <localized-resources locales="en" />
  <deployment scope="user" autoDeploy="true"
    userCanCreateMore="false" />
  <portlet-path>/sas/portlets/remote</portlet-path>
  <portlet-actions>
```

The URL for the remote portlet's Web application, called HelloUserWikExample, is specified in the url subelement of the portlet-action element. This subelement must contain a fully qualified URL, including a fully qualified host domain name.

For details about the Web application, see [Sample Web Application \(HelloUserWikExample\)](#).

```
    <portlet-action name="display" default="true">
      <url>
        http://d9999.mycompany.com:8080/HelloUserWikExample/jsp/app.jsp
      </url>
    </portlet-action>
  </portlet-actions>
</remote-portlet>
</portlets>
```

Sample: Remote Portlet (HelloUserRemotePortlet)

Step 3: Create the Display Resources File

The sample remote portlet `HelloUserRemotePortlet` uses a display resources file to provide a description to be placed in the portlet's metadata for display to users. (If this file is not provided, the portal creates a description based on the portlet's name.)

The contents of the file, which is named `portletDisplayResources.properties`, follow.

```
portlet.title=Hello User Remote Portlet
portlet.description=Example remote portlet
```

For more information, see [Creating Display Resources Files](#).

Sample: Remote Portlet (`HelloUserRemotePortlet`)

Step 4: Create the Web Application

The sample remote portlet (`HelloUserRemotePortlet`) calls the Web application `HelloUserWikExample`. This application is enabled by SAS Foundation Services and runs outside of the portal Web application, either on the same machine or on another machine.

If an application is called from a remote portlet, then it must generate an HTML fragment to be displayed within the portlet borders. The JSP page for the `HelloUserWikExample` contains conditional code that determines whether the application request was generated by a portlet. The JSP page then generates the appropriate type of HTML.

The steps required to create the application are described in [Sample Web Application \(HelloUserWikExample\)](#).

Sample: Remote Portlet (`HelloUserRemotePortlet`)

Step 5: Create the PAR File, and Deploy and Test the Portlet

The last step in developing the HelloUserRemotePortlet was to create the files that were used to deploy the portlet. The portlet files were compressed into a portlet archive (PAR) file that contains

- appropriately named and organized directories and subdirectories, as described in [Step 1: Create the Directory Structure](#).
- all of the supporting files, including the files created in Steps 2 through 4. The files must be placed in the appropriate directories as described in [Step 1: Create the Directory Structure](#) and [Creating a PAR File for Deployment in Your Application](#).

The JAR utility was used to compress the directories and files into archives, and the archives were named HelloUserRemotePortlet.par and HelloUserWikExample.war.

Before deploying a remote portlet into the production environment, you should

1. deploy the associated Web application and test it using its direct URL. The steps required to create the application are described in [Sample Web Application \(HelloUserWikExample\)](#).
2. deploy the portlet into a staging area (that is, a test installation of the portal Web application) for verification and testing.

For information about how to deploy a PAR file into the portal Web application, see [Portlet Deployment](#) in the *SAS Web Infrastructure Kit Administrator's Guide*.

Themes

To customize the appearance of the portal Web application to meet the requirements of your organization, you can implement one or more new *themes*. A theme consists of

- **Cascading style sheets**, which determine the attributes and backgrounds for text in the portal Web application. A cascading style sheet is a standard mechanism for defining consistent and reusable formatting instructions for Web-based content. You can create your own customized styles as part of a new theme.
- **Graphical elements**, including images for the company logo and the banner. You can incorporate your own customized graphics files as part of a new theme.
- **HTML documents**, which serve as page templates for the portal Web application. For best results, you should not modify these documents when creating a new theme.
- **A theme descriptor**, which is an XML file that describes the elements of a theme. You must create a theme descriptor for any new themes that you create.

Note: The application name, "SAS Portal," which appears in the banner of the portal Web application, is not part of the theme. However, you can change it. For details, see [Changing the Application Name](#).

The SAS Web Infrastructure Kit is delivered with a theme called `default`, which is automatically invoked when a user brings up the portal Web application. You can create as many additional themes as you want. Users who have the appropriate permissions will then be able to invoke one of the new themes by using the **Set user preferences** option on the Options menu.

To provide a starting point for defining new themes, the SAS Web Infrastructure Kit includes an archive named `example.zip`, which contains all of the elements needed to create a new theme. See [Defining and Deploying New Themes](#) for instructions on using this archive to create your own themes.

The `example.zip` file contains a theme descriptor called [ExampleTheme.xml](#), which you can use as a basis for creating your own theme descriptor. For more information about the theme descriptor's elements and attributes, see the [DTD description](#) and the comments at the top of the `ExampleTheme.xml` file.

Note: For best results when creating a new theme, you should modify the files in `example.zip` rather than modifying the portal Web application's default themes. This will ensure that your new theme will not be lost if a new version of the application is installed.

Defining and Deploying New Themes

Use these steps to create a new theme for your portal Web application:

1. Extract the files from `example.zip`.
2. Make desired changes to the cascading style sheets and graphics.
3. Update and rename the theme descriptor (`ExampleTheme.xml`).
4. Deploy the new theme.

Note: In addition to the style sheets, the graphics, and the theme descriptor, the archive file also contains a number of HTML documents which serve as page templates. For best results, do not modify these HTML files.

Step 1: Extract the Files from `example.zip`

The SAS Web Infrastructure Kit includes an archive named `example.zip` which contains all of the elements needed to create a new theme. You can find this archive in the path `Samples\Themes` within the SAS Web Infrastructure Kit's install location.

For example, if you used the default install location on a Windows system, then the `example.zip` file would be located in the following path:

```
c:\Program Files\SAS\Web\Portal2.0.1\Samples\Themes
```

Locate this file, and extract its contents to a working directory. The following directory structure is created:

Directory	Contents
(root)	This directory contains the theme descriptor file (<code>ExampleTheme.xml</code>).
<code>/images</code>	This directory contains the graphics files.
<code>/styles</code>	This directory contains the following Cascading Style Sheets: <ul style="list-style-type: none">• <code>Portal.css</code>• <code>sasStyle.css</code>• <code>sasComponents.css</code>
<code>/templates</code>	This directory contains HTML document templates. Note: These files should not be changed.

Step 2: Make Desired Changes to the Cascading Style Sheets and Graphics

Determine which aspects of the portal Web application's appearance you would like to change. For assistance in identifying the specific style sheet classes that you need to modify in order to implement these changes, see [Styles in Portal.css](#).

Make the desired changes to the style sheets and graphics files. Then assign a new name to any of the CSS files and graphics files that you have changed.

Note: You must update the theme descriptor file (as described in [Step 3](#)) with any new file names that you have used for CSS and graphics files. When updating the elements for graphics files, you can update only the `file` attributes; the `name` attributes for these elements must remain the same.

Step 3: Update and Rename the Theme Descriptor (ExampleTheme.xml)

The theme descriptor is an XML file that describes the elements of the new theme. The `example.zip` file contains a theme descriptor called `ExampleTheme.xml`.

Open the `ExampleTheme.xml` in an editor, and make the following changes. For more information about the elements and attributes, see the [Element Descriptions](#) and the comments at the top of the `ExampleTheme.xml` file.

1. In the `theme` element, edit the attribute values as follows:

name

Replace `Example` with a unique name for your theme. The name cannot contain spaces.

label

Replace `An Example` with a descriptive label for your theme. When users choose **Set user preferences** on the Options menu in the portal Web application, this label will appear as a selection in the Theme field.

description

Replace `Example Theme for SAS applications` with a free-form description.

URIPath

In the path `themes/example`, replace `example` with the name of the directory where the new theme is to be deployed. The directory name must be the same as the theme name, as defined in the `name` attribute.

2. In the `Style` elements, update the `file` attribute to reflect any new names that you have assigned to CSS files. **Note:** Do not change the value of the `media` attributes. This attribute must equal `SCREEN` in order for the style to be loaded.
3. In the `Image` elements, update the `file` attribute to reflect any new names that you have assigned to graphics files. **Note:** Do not change the `name` attributes, since the portal Web application uses these names to determine which graphics to display. For example, the `name` attribute for the logo image must retain the value `logo`. However, you can change the `file` attribute from `logo.gif` to `MyCompanyLogo.gif`.
4. Make any other necessary changes to the example file. For example, if you are not going to use the directory names `images` and `styles`, you must change the `directory` attribute in the `Styles` and `Images` elements. However, it is not advisable to change these directory names.
5. Save the theme descriptor with a new name.

Step 4: Deploy the New Theme

Use the following steps to deploy the new theme into the portal Web application:

1. In the path `Portal\themes` within the SAS Web Infrastructure Kit's install location, create a directory with the same name that you defined in the `URIPath` attribute of the `Theme` element. For example, if you used the default install location on a Windows system, then you would create your new directory under the

following path:

`c:\Program Files\SAS\Web\Portal2.0.1\Portal\Themes`

2. Move all of the theme elements into the new directory. Be sure to retain the directory structure that is described above, and be sure to use the directory names that you defined in the theme descriptor.
3. Use the `configure_wik.bat` utility to create a new `Portal.war` file that incorporates the new theme directory.

Note: Instead of creating a new WAR file, you can place the new theme directly into the servlet container. However, the new theme will be destroyed if you deploy a new WAR file in the future that does not contain the new theme.

4. Deploy the new WAR file by using the appropriate procedures for your servlet container.

When authorized users click **Change user preferences** on the Options menu, they will now see the new theme as a selection in the User Preferences dialog box.

If you want to make the new theme the default, then the administrator must log on as `sasguest`, click **Change user preferences** on the Options menu, and then select the new theme.




















Themes

Styles in Portal.css

Portal.css is a style sheet that was created specifically for the portal Web application. You can use it as is, or modify it as needed, for any application that you develop using the SAS Web Infrastructure Kit.

The following table lists some specific classes in Portal.css, which you might want to modify when creating a new theme. These styles will continue to evolve as the SAS Web Infrastructure Kit evolves.

Presentation Component Affected	Class	Elements Affected	Default Color Setting	
Banner	apptitle	Application name (SAS Portal in the default installation)	White	
	utilmenu	Banner links (Options, Search, Log On, Log Off, Help , etc.)	White	
	utilmenuBG	Background for banner links	Grayish blue (#3872AC)	
Navigation bar	div.portalTabArea	Area to the right of the tabs in the navigation bar	Dark blue (#003399)	
	ltBeige	Horizontal divider between navigation bar and page area	Light beige (#F1EACB)	
	tabDivider	Horizontal divider between the banner and the navigation bar	Grayish blue (#3872AC)	
	td.portalActiveTabHoriz td.portalActiveTabVert	Tab background for the active page	Light beige (#F1EACB)	
	td.portalActiveTabHoriz A td.portalActiveTabVert A	Tab text for the active page	Grayish blue (#3872AC)	
	td.portalTabHoriz td.portalTabVert	Tab background for inactive pages	Dark blue (#003399)	
	td.portalTabHoriz A td.portalTabVert A	Tab text for inactive pages	White	
Options menu	utilmenudropdown:link	Menu items	Grayish blue (#3872AC)	
	utilmenudropdown:visited	Visited menu items	Grayish blue (#3872AC)	
	utilmenuTable	Background	White	

	utilmenuTable	Border	Grayish blue (#3872AC)	
Pages	body	Default for text messages	Black	
	workarea	Background for pages and portlets	White	
Portlets	portletTableBorder	Border for portlets	Beige (#E4D2BA)	
	portletTableHeader	Background for portlet headers	Light beige (#F1EACB)	
	portletTableHeaderLeft	Text for portlet headers	Brown (#AB8029)	
	treeText	Item labels in collection portlets	Blue (#0000CC)	
	treeDescription	Item descriptions in collection portlets	Dark gray/brown (#555555)	
Search results	searchTableBorder	Search results border	Beige (#E4D2BA)	
	searchTableHeader	Search results heading background	Light beige (#F1EACB)	
	secondTableRow	Background color for alternate rows in search results	Very light beige (#F8F5E6)	
Task wizards (for adding and editing pages, adding and editing portlets, managing subscriptions, etc.)	Button	Button text	Grayish blue (#3872AC)	
	Button	Button border	Grayish blue (#3872AC)	
	TableBorder	Border for tables appearing in wizards	Gray (#B2B2B2)	
	TableHeader	Heading text for tables appearing in wizards	Black	
	TableHeader	Heading background for tables appearing in wizards	Light gray (#DDDDDD)	
	wizardHeader	Text for wizard headers	Brown (#AB8029)	
	wizardHeader	Background for wizard headers	Light beige (#F1EACB)	
	wizardTableBorder	Borders for wizards	Beige	

			(#E4D2BA)	
Viewers (for content items)	actionsMenuBg	Background for toolbar	Light beige (#F1EACB)	
	topRightMenuItem:link topRightMenuItem:visited topRightMenuItem:active	Text for links in toolbar	Grayish blue (#3872AC)	

ExampleTheme.xml File

The contents of the ExampleTheme.xml file, which you should use as a basis for creating your own theme descriptor, follow.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Themes SYSTEM "http://www.sas.com/webapp/themes.dtd">

<!--
  Themes consist of images, templates and stylesheets. Templates are
  typically HTML files. Themes must have a name, which will not be
  localized. Themes must have a label for display to users.
  All file paths must be relative to the URI path of the theme. For themes
  used within a web application, we recommend creating new directories
  under context root/themes whose names match the theme names. Within the
  theme-name directory, create subdirectories named templates, images, and
  styles. Put cascading style sheets into the styles folder, image files
  into images, and all template files into templates.

  <Themes>
  <Theme name="" // the name of the theme (without blanks). Not localized.
    //Required.
    label="" // a label for display to users. Required.
    description="" // a description for display to users. Optional.
    device="" // name suitable display device, e.g. desktop browser, PDA,
    // WAP phone. Required.
    URIPath="" //the path to the theme's directory. If it does not include
    // a final "/", one will be appended when constructing paths
    // to subdirectories. Required.
  />

  <Images directory=""> // the name of directory where the images are
    // located (without backslash). Required
  <Image name="" // Required.
    description=""
    altTextKey="" // Optional. Resource key for alt= text on img tag
    path="" // Optional. Used if path to image is not URIPath + images
    appliesTo="ALL | className(s)" // indicates that ALL class will make
    // use of this value, or if there is
    // a list of classes, only apply to
    // those. 'ALL' is a reserved word.
    // Required.
    width="" //Optional. Value for width= attr on img tag
    height="" //Optional. Value for height= attr on img tag
    file="" // the name of image file, e.g. logo.gif Required
  />

  <Styles directory="" // the name of directory where the stylesheets are
    // located (without backslash).
  <StyleSheet name="" // descriptive name of the cascading style sheet.
    description="" // description of the style's purpose
    media="" // valid values are: screen|print Required.
    order="n" // for each media type, the order in which to load the style
    // sheet. Required. The value for order must be an ASCII
    // integer (no Unicode-escaped characters).
    file="" // The file name, e.g. sasStyle.css Required.
  />

  <Templates directory=""> // the name of directory where the templates
    // are located (without backslash). Required.
  <Template name="" // name of the template. Required
```

```

description="" // description of the template's purpose
markup="" // markup language used in the template Required
file="" // name of the template file Required
/>
</Theme>
</Themes>
path=Theme.URIPath + Images.dir + asset.value OR
path=Theme.URIPath + asset.path + asset.value

```

The actual path to an asset is the concatenation of the URIPath attribute on the Theme element, the dir attribute on the Asset type (style, image, or template), and, of course, the value of the asset itself. For example, for the logo, it would be themes/default/images/logo.gif

Alternatively, if an asset includes a "path=" attribute, then the actual path to the asset is the concatenation of the URIPath attribute on the Theme element, the path attribute on the asset, and the value of the asset itself.

```
-->
```

```

<Themes>
  <Theme name="Example" label="An Example" description="Example Theme
for SAS applications"
    URIPath="themes/example" device="DESKTOP">
    <Styles directory="styles">
      <StyleSheet name="SAS Style" description="Default SAS style"
        order="1" media="SCREEN" file="sasStyle.css"/>
      <StyleSheet name="AppDev Studio" order="2" media="SCREEN"
        file="sasComponents.css"/>
      <StyleSheet name="Example Style" order="3" media="SCREEN"
        file="Portal.css"/>
    </Styles>
    <Images directory="images">
      <Image name="logo" description="SAS Logo"
        altTextKey="desktop.logo.txt" appliesTo="ALL" width="67"
        height="30" file="logo.gif"/>
      <Image name="banner_swirl_bg" description="SAS banner swirl
background" altTextKey="" appliesTo="ALL" height="40"
        file="banner_swirl_bg.gif"/>
      <Image name="banner_first_bg" description="SAS banner, first
background" altTextKey="" appliesTo="ALL" height="40"
        file="banner_first_bg.gif"/>
      <Image name="banner_top" description="SAS banner top"
        altTextKey="" appliesTo="ALL" height="5" width="100%"
        file="banner_top.gif"/>
    </Images>
    <Templates directory="templates">
      <Template name="navigation_action_menu" description="Navigation
Menu with actions" markup="HTML"
        file="navigation_action_menu.html"/>
      <Template name="navigation_action_nologoff_menu"
        description="Navigation Menu with actions but no logoff option"
        markup="HTML" file="navigation_action_nologoff_menu.html"/>
      <Template name="navigation_menu" description="Navigation Menu"
        markup="HTML" file="navigation_menu.html"/>
      <Template name="viewer_title" description="Viewer Title Bar"
        markup="HTML" file="viewer_title.html"/>
      <Template name="task_action" description="Task Action Menu"
        markup="HTML" file="task_action.html"/>
      <Template name="logon_action" description="Logon Action Menu"

```

```

        markup="HTML" file="logon_action.html"/>
<Template name="viewer_action_menu" description="Viewer Action
    Menu" markup="HTML" file="viewer_action_menu.html"/>
<Template name="utilmenu" description="Options Menu" markup="HTML"
    file="utilmenu.html"/>
<Template name="helpmenu" description="Help
    Menu" markup="HTML" file="helpmenu.html"/>
<Template name="banner" description="Banner" markup="HTML"
    file="banner.html"/>
<Template name="banner_product" description="Banner with product
    name" markup="HTML" file="banner_product.html"/>
<Template name="banner_logo" description="Banner with logo"
    markup="HTML" file="banner_logo.html"/>
</Templates>
</Theme>
</Themes>

```

Themes

Element Descriptions for Themes DTD

Use any of the following links to view detailed descriptions of the elements in `themes.dtd`:

- ***Top Elements*** is a list of the top–most elements of the DTD, with links to pages describing each top–most element. From these pages, you can link to descriptions of individual child elements.
- ***All Elements*** is a list of all elements defined in the DTD. The links on the page provide quick access to the description of any individual element.

Document generated by  [dtd2html](#) 1.5.1.

themes DTD

Changing the Application Name

The application name, "SAS Portal," which appears in the banner of the portal Web application, is not part of the theme. However, you can specify a different application name to appear in the banner.

Use the following steps to specify a different application name:

1. In the `install.properties` file (which is located in `PortalConfigure` folder of the setup directory), add the following property:

```
$NAME_IN_BANNER$=Application Name
```

where *Application Name* is the name that you want to display in the banner.

2. Use the `configure_wik.bat` utility to create a new `Portal.WAR` file that incorporates the new application name.
3. Deploy the new WAR file by using the appropriate procedures for your servlet container.

Note: If you do not want to create a new WAR file, then you can edit the `NameInBanner=` parameter of the `Portal.config` file to specify the new name. The `Portal.config` file is located in the `Portal/WEB-INF/` directory of your servlet container.

If you use this method, then you should also add the `$NAME_IN_BANNER$=` parameter to the `install.properties` file. Otherwise, the new application name will be destroyed if you run the `configure_wik` script again in the future.

Resources

This section provides information about the application programming interfaces (APIs) that are provided with the SAS Web Infrastructure Kit. The following topics are discussed:

- the Portlet API, which gives you access to classes that provide the portal Web application's navigation and request processing functions.
- SAS Foundation Services, which is a set of infrastructure and extension services that support the development of integrated, scalable, and secure Java–based applications. The SAS Web Infrastructure Kit includes the SAS Foundation Services Facade API, which is a set of convenience classes that developers can use to obtain references to the most commonly used foundation services.

Links to the detailed class documentation are also provided.

Using the Portlet API

The Portlet API provides access to classes that provide the portal Web application's navigation and request processing functions. For detailed information about the API, see the [class documentation](#).

The following classes are of particular usefulness in creating custom portlets:

com.sas.portal.portlet.DefaultPortletAction

You can extend this class in order to create your own portlet actions. For more information, see [Creating a Portlet Action Class](#).

com.sas.portal.portlet.ErrorHandlerInterface

You can use this interface to handle errors that your portlet encounters. For more information, see [Error Handling Actions](#).

com.sas.portal.portlet.HTMLPortletAction

You can extend this class in order to create your own portlet actions. For more information, see [Creating a Portlet Action Class](#). Possible uses include

- ◇ correctly displaying non-Latin1 character sets when a portlet is displayed in preview mode. For an example of this use, see [Step 4: Create the Action Class](#) in the [Sample Localized Display Portlet \(Welcome Portlet\)](#).
- ◇ preparing URLs for actions within an interactive form JSP, and to populate a JavaBean with parameters from a JSP form. For an example of this use, see [Step 4: Create the Action Class](#) in the [Sample Interactive Form Portlet \(FormExample\)](#).

com.sas.portal.portlet.PortletContextInterface

You can use this interface to obtain the user ID of the person who is logged on to the portal Web application. Use the `getSessionContext()` method to retrieve the session context, use the `getUserContext()` method to retrieve the user context from the session context, and then use the `getPerson()` method to obtain the user ID from the user context.

com.sas.portal.portlet.NavigationUtil

You can use this class to

- ◇ create URLs for buttons on your JSP pages (for example, **OK** and **Cancel** buttons). For an example of this use, see [Creating a Portlet Template](#).
- ◇ obtain a portlet's resource bundles in order to create a localization context for your portlet's JSP page. For an example of this use, see [Creating a Localized Portlet](#).

com.sas.portal.portlet.PortletActionInterface

You can use this interface to develop an action class for your portlet. For more information, see [Creating a Portlet Action](#), and [Step 4: Create the Action Class](#) in the [Sample Localized Display Portlet \(Welcome Portlet\)](#).

com.sas.portal.portlet.PortletInitializerInterface

You can use this interface to develop an initializer class, which runs before your portlet is displayed for the first time on a portal page. Possible uses include

- ◇ reading initial parameters that are specified in your portlet's deployment descriptor file (`portlet.xml`)
- ◇ connecting to an external resource such as a database.

For more information, see [Creating an Initializer Action](#).

com.sas.portal.portlet.PostProcessorInterface

You can use this interface to develop a postprocessor class that runs when your portlet is no longer on display. Possible uses include

- ◇ freeing resources that were used in the portlet initializer.
- ◇ removing HttpSession attributes that were set in the portlet initializer or portlet action. This is especially important to consider since multiple copies of your portlet could exist on other portal pages or even on the same page.

For more information, see [Creating a Postprocessor Action](#).

For detailed information, see the [class documentation](#).

Using SAS Foundation Services With the Portal

SAS Foundation Services is a set of infrastructure and extension services that support the development of integrated, scalable, and secure applications that are developed using Java. The design model for SAS Foundation Services supports both local and remote resource deployment and promotes resource sharing among applications. Sharing can occur for a specific session, for a specific user, or globally, as appropriate. At the same time, the model controls access to protected resources based on privileged–user status and group membership.

For a list and description of each service, refer to [SAS Foundation Services](#) in the *SAS Integration Technologies Developer's Guide*.

The SAS Web Infrastructure Kit includes the SAS Foundation Services Facade API, which is a set of convenience classes that developers can use to obtain references to the most commonly used foundation services. The foundation services facade should provide all of the necessary functionality to integrate your applications with the portal Web application. The foundation services facade is part of the Portlet API. For detailed information about the facade classes, select `com.sas.services.webapp` in the Portlet API [class documentation](#).

You can also choose to use the SAS Foundation Services classes directly.

Using Locally Deployed SAS Foundation Services

When SAS Foundation Services is deployed locally, the core local services stack is used, as follows:

1. `com.sas.services.security.AuthenticationService`
2. `com.sas.services.user.UserService`
3. `com.sas.services.logging.LoggingService`
4. `com.sas.services.information.InformationService`
5. `com.sas.services.session.SessionService`

Using Remotely Deployed SAS Foundation Services

In the portal Web application architecture, an application called SAS Services (SASServices) is used to remotely deploy SAS Foundation Services. This application enables secure information sharing among applications. Through SAS Services, you can implement remotely deployed content viewers, remote portlets, and stand-alone Web applications that are called by the portal Web application and invoked with the portal Web application user's credentials. The user does not need to log on again, and user and session information can be shared as needed.

The SAS Services application makes the following stack available to applications that are enabled by SAS Foundation Services:

1. `com.sas.services.security.AuthenticationService`
2. `com.sas.services.user.UserService`
3. `com.sas.services.logging.LoggingService`
4. `com.sas.services.information.InformationService`
5. `com.sas.services.session.SessionService`

The SAS Services application must be up and running on a machine that is accessible to the remote Web application. In addition, the remote Web application must include a properties file that points to the definition of the remote services.

The portal Web application provides the SAS Services application with session context information for each authenticated user who is logged on. If the portal Web application passes a unique session ID to a remote Web application, then the remote Web application can obtain the appropriate user's session context information from the SAS Services session. The remote Web application should use the following steps to accomplish this:

- retrieve the session ID from the portal Web application request and use it to obtain a reference to the SAS Services session. This is done by creating a new `WebappContextParams` object, as follows:

```
WebappContextParams params = new WebappContextParams(request);
```

- use the `SessionContextInterface` to get the remote session, as follows:

```
sharedSession = params.getSessionContext();
```

- protect the session with a lock object, as follows:

```
sessionLock = SharedSession.lock("com.sas.MySessionName");
```

- call methods from the `WebappContextParams` class to retrieve data, as in this example:

```
UserContextInterface userContext = sharedSession.getUserContext();
```

- unlock the session, as follows:

```
sharedSession.unlock(sessionLock);
```

For an illustration of the use of SAS Foundation Services to access a SAS Services session, see the [Sample Web Application \(HelloUserWikExample\)](#).