# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Chapter 1</td>
<td><strong>Tutorial Preparation and Basics</strong></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Typographical Conventions Used in the Tutorials</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Setting Up the Environment for the WDK Tutorials</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Tutorial Basics</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Clearing Caches and Refreshing</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Overview of the Custom Layer</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Custom Layer Definition Inheritances and Overrides</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Setting Up an IDE</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Creating a Project in NetBeans</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Mounting the WDK Directories</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Mounting the DFC JAR Files</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Stopping the Internal Tomcat Server</td>
<td>13</td>
</tr>
<tr>
<td>Chapter 2</td>
<td><strong>Creating a Start Page for a New WDK Application</strong></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Variations on This Tutorial</td>
<td>16</td>
</tr>
<tr>
<td>Chapter 3</td>
<td><strong>Adding a Logout Link</strong></td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Task Objective</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Technical Overview</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Creating the Custom Layer Files</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Extending the Drilldown Component</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Modifying the Generic Actions XML File</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Creating a Custom Resource File</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Testing the Logout Link</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Extending the Logout Action</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Variations on This Tutorial</td>
<td>22</td>
</tr>
<tr>
<td>Chapter 4</td>
<td><strong>Hiding Attributes from a Data Dictionary Display</strong></td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Task Objective</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Technical Overview</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Overview of Content Server Display Configuration</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>How the docbaseattributelist Control Works</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Procedures Required for This Tutorial</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Creating the Custom Layer Files</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Modifying the docbaseattributelist Control Definition</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Testing the Configuration</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Variations on This Tutorial</td>
<td>30</td>
</tr>
<tr>
<td>Chapter 5</td>
<td><strong>Adding an Attribute Column to Webtop Classic View</strong></td>
<td>33</td>
</tr>
</tbody>
</table>
### Table of Contents

- Task Objective ........................................................................................................... 33
- Technical Overview .................................................................................................... 35
  - Defining the Default View in Webtop ................................................................. 35
  - Anatomy of Webtop Classic View ........................................................................ 35
  - The Custom Layer for WDK Applications .......................................................... 36
- Creating the Custom Layer Files ............................................................................. 36
- Changing the Default View in Webtop ..................................................................... 37
- Displaying the Creator Attribute ............................................................................ 37
- Variations on This Tutorial ..................................................................................... 38

**Chapter 6 Creating an Action List Component** .......................................................... 41
- Task Objective .......................................................................................................... 41
- Technical Overview ................................................................................................... 42
- Creating the Component Definition ......................................................................... 42
- Creating the Component Layout .............................................................................. 43
- Creating the Component Class .................................................................................. 45
- Supporting Navigation Between Pages ....................................................................... 47
- Creating the Action List Datagrid ........................................................................... 47
- Creating the Action Info Datagrid ........................................................................... 50
- Creating the Resource Bundle .................................................................................. 54
- Testing the Customization ........................................................................................ 55
- Variations on This Tutorial ..................................................................................... 55

**Chapter 7 Using Tracing and Logging in a Component** .......................................... 57
- Task Objective .......................................................................................................... 57
- Technical Overview ................................................................................................... 57
- Creating the Tracing Class ....................................................................................... 58
- Adding Tracing to the Component Class ................................................................... 59
- Adding Custom Tracing to the List .......................................................................... 60
- Turning on Tracing .................................................................................................... 60
- Testing the Customization ....................................................................................... 60
- Examining the Trace Log ......................................................................................... 61
- Variations on This Tutorial ..................................................................................... 61

**Chapter 8 Troubleshooting** ...................................................................................... 63
- Did you precompile your Java class? ....................................................................... 63
- Did you refresh the Configuration Service? ............................................................. 63
- Did you remove generated files? .............................................................................. 63
- Did you clear the browser cache? ............................................................................ 64
- Did you check the name and location of the XML resource file? ......................... 64
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Custom Directory Contents after Installation</td>
<td>11</td>
</tr>
<tr>
<td>2-1</td>
<td>Post-Login Drilldown Page</td>
<td>16</td>
</tr>
<tr>
<td>3-1</td>
<td>Drilldown Page with Logout Link</td>
<td>17</td>
</tr>
<tr>
<td>4-1</td>
<td>Typical Drilldown Page</td>
<td>24</td>
</tr>
<tr>
<td>4-2</td>
<td>Default dm_document Info page</td>
<td>24</td>
</tr>
<tr>
<td>4-3</td>
<td>Display Configuration Tab in DAB</td>
<td>26</td>
</tr>
<tr>
<td>4-4</td>
<td>Scope Configuration Dialog Box in DAB</td>
<td>26</td>
</tr>
<tr>
<td>5-1</td>
<td>The Doclist Component in WDK</td>
<td>34</td>
</tr>
<tr>
<td>5-2</td>
<td>The Objectlist Component in Webtop</td>
<td>34</td>
</tr>
<tr>
<td>5-3</td>
<td>Classic View with Creator Attribute Displayed</td>
<td>34</td>
</tr>
<tr>
<td>5-4</td>
<td>Framesets in Webtop Classic View</td>
<td>35</td>
</tr>
<tr>
<td>6-1</td>
<td>Action List Page</td>
<td>42</td>
</tr>
<tr>
<td>6-2</td>
<td>Action Info Page</td>
<td>42</td>
</tr>
<tr>
<td>7-1</td>
<td>Tracing JSP Page</td>
<td>60</td>
</tr>
<tr>
<td>7-2</td>
<td>Tracing Output</td>
<td>61</td>
</tr>
<tr>
<td>7-3</td>
<td>JavaScript Tracing</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Tables

Table 1–1. Directory Structure of the Custom Layer .................................................. 11
Table 4–1. Docbaseattributelist Control XML Resource Files .................................. 27
Table 5–1. Frameset Contents of Classic View JSPs ...................................................... 36
This manual contains a set of tutorials that show how to perform some common configuration and customization tasks in Web Development Kit (WDK) and WDK applications. While the tutorials and other WDK documentation do not cover every kind of configuration or customization, they will teach you the basics of working with WDK configuration files, UI pages (JSP), and classes.

The tutorials are based on WDK and include information for configuring and customizing some WDK client applications.

For more information on installing and using WDK and WDK applications, see the release notes, installation guide, and other WDK manuals installed with the product.

**Intended Audience**

This manual is intended for Java developers who are developing JSP-based Web applications that incorporate Documentum functionality. To use all WDK customization capabilities, you must be familiar with the following technologies and standards:

- Java 2
- Servlet 2.3
- JSP 1.1 including tag libraries
- Cascading style sheets (CSS)
- HTML, particularly forms, tables, and framesets
- JavaScript, including client events and event handling, frame referencing, and form action methods
- XML

**Revision History**

The following changes have been made to this document:
# Revision History

<table>
<thead>
<tr>
<th>Revision Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 2005</td>
<td>Updated for 5.3 release. Added tracing tutorial. Moved tutorial source code from this document to developer Web site (<a href="http://developer.documentum.com">developer.documentum.com</a>).</td>
</tr>
</tbody>
</table>
Chapter 1

Tutorial Preparation and Basics

This chapter contains information that will help you complete the tutorials in the chapters that follow:
The chapter contains the following sections:
• Typographical Conventions Used in the Tutorials, page 9
• Setting Up the Environment for the WDK Tutorials, page 9
• Tutorial Basics, page 10

Typographical Conventions Used in the Tutorials

This guide uses italics for replaceable parts of a path or URL. For example:
http://server_name:port_number/virtual_dir/wdk/samples/dumpRequest.jsp

In the above URL, server_name should be replaced with the host name, the port_number should be replaced with the port number defined for your application server (the default port number for a Tomcat application server is 8080), and virtual_dir should be replaced with the virtual directory that was created during installation of WDK or a WDK application. If you are accessing the application through a browser running on the server machine, substitute localhost for server_name.

Setting Up the Environment for the WDK Tutorials

The procedures in this guide are based on the following environment:
• Sun Java JDK 1.4.2, available from http://java.sun.com/
• Tomcat 5.0.28, available from http://jakarta.apache.org/tomcat/

Note: The tutorial procedures are based on the Tomcat J2EE application server, selected because it is freely available for download. You may use any application server that is certified for WDK.
• WDK 5.3
See the WDK release notes for hardware requirements and software certifications. See Web Development Kit and Applications Installation Guide for information on installing WDK and verifying your installation on an application server.

- (Optional) Webtop or another WDK client application, with WDK installed to customize that application (See Web Development Kit and Applications Installation Guide for the procedure on installing WDK to customize a WDK client application.)
- NetBeans IDE 3.6 or another IDE

Note: You can use any J2EE-compliant IDE to compile the Java code for the customization tasks. The instructions for the customization tutorials in this manual are based on NetBeans IDE because it is freely available for download.

## Tutorial Basics

The following sections provide some basic information that will help you with the tutorials.

### Clearing Caches and Refreshing

If you add or change XML resource files, you should refresh the configuration service. An easy way to do this is to open a browser, log into the WDK application, and navigate to http://server_name:port_number/virtual_dir/wdk/refresh.jsp. (You must have a session before refreshing the configuration definitions in memory.)

If you add a new JSP file, the page will be compiled automatically by the application server. If you change a JSP file that contains server-side generated content or included JSP files, or if you modify any JavaScript function, you should clear the java and class files by deleting the folder `tomcat_home/work/Standalone/localhost/virtual_dir` or navigating within the folder to the files themselves and deleting them. Repeat the process if necessary for `tomcat_home/work/catalina/localhost/virtual_dir`.

If you change a .properties file containing an externalized string that appears in your JSP page or Java class, you must restart the application server to pick up the change.

If you are working with the data dictionary and you make a change to a custom type, you can clear the data dictionary cache by opening a browser and navigating to http://server_name:port_number/virtual_dir/wdk/refresh.jsp. You must wait until the dictionary dictionary change has been published to the Content Server, which is usually done by a job running at regular intervals on the Server.

If you add or customize a Java file, you should pre-compile it, using an IDE or other compiler.
Overview of the Custom Layer

WDK and WDK applications make use of a customization layer, which allows you to keep configurations and customizations in a location separate from the installed product. This allows the custom layer to be easily migrated after an upgrade or reinstallation.

A default custom directory is installed with the product in the following location:

```
virtual_dir/custom
```

This directory is installed with the subdirectories and files shown in the following figure.

![Custom Directory Contents after Installation](image)

The custom layer must contain the directories shown above. The directory contents, and additional directories for your custom components, are described in the table below.

<table>
<thead>
<tr>
<th>Directory of File Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>custom/config</td>
<td>XML resource files</td>
</tr>
<tr>
<td>custom/component_name</td>
<td>JSP pages. Recommended one folder per component.</td>
</tr>
<tr>
<td>custom/strings</td>
<td>Custom resource strings</td>
</tr>
<tr>
<td>custom/theme</td>
<td>Custom themes (to change the look and feel of the UI)</td>
</tr>
<tr>
<td>app.xml</td>
<td>File for changes to app.xml settings</td>
</tr>
</tbody>
</table>

Custom Layer Definition Inheritances and Overrides

When you extend a definition in a configuration file in WDK or a WDK application, any element that is defined in the custom definition will override that same element in the definition that has been extended. If the element is not specified in the custom definition, that element will be inherited from the component that has been extended in the XML resource file.

For example, suppose you want to extend the docbaseattributelist control definition, which includes the section below:

```xml
<config version='1.0'>
    <scope>...
    <category>
        <name><nlsid>MSG_INFO</nlsid></name>
```

In this example, if you include the category element in your extended definition, this will overwrite all <category> elements. This means that you must repeat any <attribute> elements that you want to include in your override. If you want to add a single element, in your extended definition you must repeat the entire list of attributes plus add the one additional attribute.

Setting Up an IDE

When you create a custom class, you compile it using either an IDE, such as the NetBeans IDE, or an external compiler. This section contains information on creating a project in the NetBeans IDE. If you use another IDE, consult the IDE documentation for the proper way to set the classpath and reference the external libraries.

This guide uses the NetBeans IDE version 3.6 for the customization tutorials that require compiling Java classes. You can use it as your text editor for the configuration tutorials as well.

Creating a Project in NetBeans

To create a new project in NetBeans

1. Open the NetBeans IDE.
2. Choose Project→Project Manager.
3. Click New.
4. Create a name for your project, such as wdk53, and click OK.
Mounting the WDK Directories

Mounting the WDK Java archives effectively sets the classpath for your project.

To mount the WDK directories
1. Select File→Mount Filesystem from the NetBeans IDE menu bar.
2. Select Local Directory as the filesystem type, then click Next.
3. Navigate to the directory in which you have installed the Tomcat application server, and open the /webapps subdirectory. Highlight the virtual directory in which you installed WDK (for example, wdk53), then click Finish.
   **Note:** If you see a message that an alternate view is available, you can safely ignore it.
4. Select File→Mount Filesystem from the NetBeans menu bar.

Mounting the DFC JAR Files

Use the following procedure to mount the DFC JAR files necessary for the project.

To mount the DFC JAR files:
1. Select File→Mount Filesystem from the NetBeans IDE menu bar.
2. Select Archive Files as the filesystem type, then click Next to move to the next page.
3. Navigate to the Shared subdirectory of the DFC program root. (The default location on Microsoft Windows is C:\Program Files\Documentum\Shared.) Highlight all the JAR files listed, and then click Finish.

Stopping the Internal Tomcat Server

If you are running NetBeans, stop the internal Tomcat server instance before starting an external Tomcat server. Use the following procedure.

To stop the NetBeans IDE server instance:
1. Click the Runtime tab in the Explorer frame.
3. Right-click on http://localhost:8084 to see whether the server has been started. If the status is running, select Stop Server.
Caution: Do not start an internal or external Tomcat server from within the IDE using the default command arguments. The default start script will not use the environment settings required by WDK.
Chapter 2

Creating a Start Page for a New WDK Application

Most Web applications are accessed through an index.html or default.html page. This is the starting point of your application. WDK does not require an index page, but it is a best practice to have one. If you are customizing Webtop or another WDK-based application that already has an index.html page, you do not need to do this tutorial.

In this tutorial, you define a welcome page with a JavaScript function that redirects to the drilldown component. The drilldown component is a navigation component that will be the starting point of the new application. If the user has not logged in to the application, then the login component will be invoked from the drilldown component class, and the user will be prompted to log in.

**Note:** Any component that requires a session will launch the login dialog if the user does not have a session. You do not need to call the login component explicitly.

Use the following procedure to create an index.html start page for your application.

**To create a start page:**

1. Create a new file called index.html in the `tomcat_home/webapps/virtual_dir` directory, where `tomcat_home` is the directory in which Tomcat is installed, and `virtual_dir` is the virtual directory into which you installed WDK.

2. Add the following text:

   ```html
   <html>
   <head>
   <script>
   function redirect()
   {
   var strPath = window.location.pathname;
   var nIndex1 = strPath.indexOf("/");
   var nIndex2 = strPath.indexOf("/", nIndex1 + 1);
   var strVirtualDir = strPath.substring(nIndex1 + 1, nIndex2);
   window.location.replace("/" + strVirtualDir + "/component/drilldown?Reload=" + new Date().getTime());
   }
   </script>
   </head>
   ```

   This JavaScript defines a function called “redirect,” which gets the root URL for the Web application, adds to it the drilldown component URL, and adds an argument of the current date and time, which will force a reload of the page every time it is accessed.
3. Add the following `<body>` tag:
```html
<body onload="redirect()">
</body>
</html>
```
This tag contains a call to the JavaScript function that you just created.

4. Save the file.

5. Test your page by starting a browser and entering the following address (after ensuring that Tomcat is running):
   ```
   http://server_name:port_number/virtual_dir
   ```
   The address should immediately change to
   ```
   http://server_name:port_number/virtual_dir/component/drilldown
   ```
If you have not previously logged in, the login page will be displayed. After log in, you see the drilldown page, shown in the following figure.

**Figure 2-1. Post-Login Drilldown Page**

If you see a blank page instead of the login screen, check for typos in the index.html file.

**Variations on This Tutorial**

This tutorial applies to WDK only, since Webtop and other WDK client applications come with their own start pages.

Instead of redirecting to the drilldown component, you could redirect to the doclist component, which combines folders and their contents in a single view.

Instead of redirecting directly to doclist or drilldown, you could have the start page redirect to the main component. Since the WDK main component is a blank page, you would need to configure or customize the main component to display the views that you choose. For example, the Webtop main component extends the WDK main component to display the streamline view after login.
Chapter 3

Adding a Logout Link

The following tutorial shows you how to add a Logout link to the drilldown page. The tutorial contains the following sections:

• Task Objective, page 17
• Technical Overview, page 18
• Creating the Custom Layer Files, page 19
• Extending the Drilldown Component, page 19
• Modifying the Generic Actions XML File, page 20
• Creating a Custom Resource File, page 21
• Testing the Logout Link, page 21
• Extending the Logout Action, page 22
• Variations on This Tutorial, page 22

Task Objective

Webtop and other WDK client applications have Logout buttons, but WDK has no logout link or button in the UI out of the box. Suppose that you want to add a Logout link to the drilldown pages in a WDK application, which will return the user to the start page with a login screen. The Logout link should appear as in the upper right of the following figure.

Figure 3-1. Drilldown Page with Logout Link
Technical Overview

The drilldown component (used by default in WDK) is defined in the XML resource file drilldown_component.xml, which points to the start page drilldown.jsp. The drilldown.jsp itself points to drilldown_body.jsp, and drilldown_body.jsp contains an actionlinklist control to which you will add the Logout link. First, you must change the logout action.

The logout action is defined in session_actions.xml in the WDK layer. It includes a <homeurl> element, which points to a post-logout location. In Webtop and Web Publisher, this <homeurl> element calls the logoff component, a sessionless component in the WDK layer that invalidates the current session and calls the drilldown component, with the login screen appearing first. The logoff component is important for controlling the number of open sessions.

Once the user clicks the Logout link, the LogoutAction class calls /wdk/logout.jsp, and logout.jsp redirects to dologout.jsp, which contains a client-side initialization containing an onLogout client event. This onLogout client event fires only if an onLogout event handler is registered. If none is registered, then the onInit() JavaScript function in logout.jsp changes the location of the top frame to whatever is configured in the <homeurl> parameter of the logout action:

```javascript
/** Client-side Initialization */
function onInit()
{
    var homeUrl = "<%=session.getAttribute("_logoutactionurl")%>";
    if (isClientEventHandlerRegistered("onLogout"))
    {
        fireClientEvent("onLogout", homeUrl);
    }
    else
    {
        getTopLevelWnd().location.replace(homeUrl);
    }
}
```

(The script that tests whether a handler for a given client event has been registered is in virtual_dir/wdk/include/events.js.)

In Webtop and in the Webtop layer of applications that are based on Webtop, the client-side event handler, onLogout, is defined in main.js:

```javascript
function onLogout(homeURL)
{
    if (homeURL != null && homeURL != "")
    {
        document.location.replace(homeURL);
    }
}
```

In this tutorial, the <homeurl> element will be configured so that it calls the logoff component, similar to Webtop and other client applications.

This tutorial involves the following procedures:

- Creating the files in the custom layer
- Extending the drilldown component XML definition to point to a custom properties file that contains text for the Logout link
Adding a Logout Link

- Modifying the list of actions displayed by the actionlinklist control to include a Logout link
- Extending the properties file that contains the resource strings for the drilldown component so that it contains the Logout link text
- Extending the logout action to call the logoff component

Creating the Custom Layer Files

Use the following procedure to set up the custom layer.

**To create the files for the customization:**

1. Copy the following file:
   
   ```
   virtual_dir/wdk/config/session_actions.xml
   ```

   to the following location:

   ```
   virtual_dir/custom/config
   ```

2. Copy the following file:

   ```
   virtual_dir/webcomponent/config/navigation/drilldown/
   drilldown_component.xml
   ```

   to the following location:

   ```
   virtual_dir/custom/config
   ```

3. Copy the following file:

   ```
   virtual_dir/webcomponent/config/actions/generic_actions.xml
   ```

   to the following location:

   ```
   virtual_dir/custom/config
   ```

4. Create a new text file named MyDrillDownNlsProp.properties in the following location (creating the directory structure if necessary):

   ```
   virtual_dir/WEB-INF/classes/com/mycompany/drilldown
   ```

Extending the Drilldown Component

The following procedure extends the drilldown component to point to the custom properties file.

**To modify the XML resource file:**

1. Open the following file:

   ```
   virtual_dir/custom/config/drilldown_component.xml
   ```

2. Change the `<component>` start tag as follows:

   ```
   <component id="drilldown" extends="drilldown:webcomponent/config/navigation/drilldown/drilldown_component.xml">
   ```
This will extend the drilldown component that is defined in the default resource file.

3. Change the <nlsbundle> element as follows:
   
   \[
   \text{<nlsbundle>com.mycompany.drilldown.MyDrillDownNlsProp</nlsbundle>}
   \]
   
   This points to the empty properties file you created in the custom layer.

4. Save the file and close it.

   Tip: When you make modifications to an XML file, check to see that the XML is well formed so it does not cause problems later. One easy way to do this is to open the XML file in Microsoft Internet Explorer.

### Modifying the Generic Actions XML File

There are two JSP pages involved with the drilldown component, which is displayed after login: drilldown.jsp calls drilldown_body.jsp. The drilldown_body.jsp file includes an actionlinklist control that displays the available actions:

\[
\text{<dmfx:actionlinklist listid='container-actions' name='container_action_list' cssclass='actions' separatorhtml='</nobr><nobr>'>}
\]

The listid attribute of the actionlinklist (id="container-actions") corresponds to the name of an action list in generic_actions.xml. To add the Logout action, update the container-actions actionlist in a custom version of generic_actions.xml.

To modify generic_actions.xml:

1. Open the following file:
   
   `virtual_dir/custom/config/generic_actions.xml`

2. Locate the beginning of the container-actions actionlist element and change it as follows:

   \[
   \text{<actionlist id="container-actions" extends="container-actions:webcomponent/config/actions/generic_actions.xml"/>}
   \]

3. Add the Logout action immediately before the closing tag for the <actionlist> element:

   \[
   \text{<action id='logout' nlsid='MSG_LOGOUT'/>}
   \]

4. Save the file and close it.
Creating a Custom Resource File

The resource string that you specified in the actionlink, namely MSG_LOGOUT, is defined in some WDK resource files, but not in DrillDownNlsProp. You must define this string in the custom properties file you created in the custom layer and pointed to from the XML resource file.

To define a custom resource string:

1. Open the following file:

   virtual_dir/WEB-INF/classes/com/mycompany/drilldown/MyDrillDownNlsProp.properties

2. Add the following two lines. (The first two lines shown here should be on one line in your file.)

   NLS_INCLUDES=com.documentum.webcomponent.navigation.drilldown.DrillDownNlsProp
   MSG_LOGOUT=Logout

   These lines include the DrillDownNlsProp.properties file in the default location and define one additional resource string, which is referenced in the nlsid attribute MSG_LOGOUT for the action that you added to the actionlist definition.

3. Save the file and close it.

Testing the Logout Link

The Logout link should now be functional, and you can test it before making other changes.

To test the Logout link:

1. Stop and restart the application server. (You must restart the application server to pick up changes to .properties files.)

2. Enter the following address:

   http://http://server_name:port_number/virtual_dir

   After you log in, the drilldown page should appear, with the new Logout link in the upper right-hand corner as shown below.

3. Click the Logout link. For now, this should result in the display of a generic Action canceled browser page.
Extending the Logout Action

The following procedure extends the logout action to call the logoff component.

**To extend the logout action:**

1. Open the following file:
   
   \`\`\virtual_dir/custom/config/session_actions.xml\`

2. Change the logout `<action>` start tag as follows:
   
   ```xml
   <action id="logout" extends="wdk/config/session_actions.xml">
   ```

3. Change the `<homeurl>` element as follows:
   
   ```xml
   <homeurl>/component/logoff</homeurl>
   ```
   
   This will call the logoff component when the logout action is invoked.

4. Save the file and close it.

5. Refresh the configuration service and test the Logout link as in the previous procedure.
   
   This time when you click the Logout button, you should see a logout message with a link to return to the login page.

Variations on This Tutorial

Adding a Logout link is not necessary in Webtop or Web Publisher, since the default applications use Logout buttons. However, you could use this tutorial as the basis for modifying other action controls. For example, you could change any actionbutton to an actionlink or vice versa. You could also change the actionlinklist to an actionbuttonlist in drilldown_body.jsp The actionbutton control requires additional attributes related to the button image and the placement of text on the button. For more information, see the Web Development Kit Reference Guide.
Chapter 4

Hiding Attributes from a Data Dictionary Display

WDK, Webtop, and other WDK client applications enable the display of attributes to be controlled by the data dictionary. Using Documentum Application Builder, you can create scopes that set distinct display characteristics for attributes on the Properties page, the Checkin page, and the Import page when each scope is active. Scopes can be based on object type, user role, or other qualifier that is defined for the application. (For more information on qualifiers and scope, see Web Development Kit and Client Applications Development Guide.)

By default, the data dictionary is enabled as the source for attribute display in a WDK-based application. The advantages of enabling the data dictionary are the following.

- A single control called docbaseattributelist controls the entire display of attributes on a page.
- Custom attributes are automatically displayed on the Properties, Import, and Checkin pages, as well as other components that use a docbaseattributelist control, with the ability to hide attributes from display.
- You can override the default and configure the display of attributes separately for each page that uses a docbaseattributelist control.
- If you are using Documentum Application Builder (DAB), you do not need to manually configure the attribute display configuration in the WDK-based application. The order of attributes can be easily set or modified through Documentum Application Builder by modifying the display configuration for any combination of scope values.

If you are upgrading a customized version of Webtop version 5.1, or a custom WDK 5.1 application, and you want to use data dictionary attribute display, you must replace your docbaseattribute controls with the docbaseattributelist control.

This chapter contains a tutorial on using the docbaseattributelist control and configuring attribute display when the data dictionary is enabled. It shows how to hide attributes displayed by the data dictionary for an object of type dm_document in WDK, Webtop, or a WDK-based application.

The tutorial consists of the following sections:

- Task Objective, page 24
- Technical Overview, page 25
- Creating the Custom Layer Files, page 29
- Modifying the docbaseattributelist Control Definition, page 29
- Variations on This Tutorial, page 30
Task Objective

The following figure is typical of a drilldown page in WDK, which displays the contents of a cabinet or a folder:

**Figure 4-1. Typical Drilldown Page**

If you click the Info icon for an object of type dm_document (the default object type for documents), you see a short list of its attributes, as in the following figure:

**Figure 4-2. Default dm_document Info page**

If you click the [+]show more link, some additional attributes are displayed, and the link changes to [-] hide more. If you are an administrator, the Show all properties control at
the top of the page will display the entire set of attributes that are stored for documents of type dm_document.

Suppose you are building a WDK application or customizing Webtop or another WDK client, and you want to remove the Subject attribute from display on the Properties, Import, and Checkin pages for objects of type dm_document.

**Technical Overview**

This task requires an understanding of the display configuration capabilities available with Content Server 5.2 and also of the docbaseattributelist control in WDK and WDK applications.

**Note:** If you are using Content Server 5.1, you can still use the docbaseattributelist control to have the data dictionary display attributes, but there is a merging process between the data dictionary category information and attributes defined in the XML resource file for the docbaseattributelist control. For more information, see the *Web Development Kit Development Guide* and *Web Development Kit and Applications Reference Guide*. Since this tutorial does not involve category information, it works with all versions of Content Server.

**Overview of Content Server Display Configuration**

Content Server 5.2 and higher includes two objects, dm_scope_config and dm_display_config, which clients can utilize to order attributes and to control which attributes are displayed. The easiest way to see the display configuration capability is to view a dm_document or dm_folder object type or subtype on a 5.2 Content Server or higher or create a new object in DAB on a 5.2 Content Server or higher. Then choose the Display Configuration tab, as in the following figure.
Figure 4-3. Display Configuration Tab in DAB

If you select Webtop in the Scope field, dm_document, dm_folder, or any subtype will display an Info category in the Display Configuration List pane. Categories can be created, renamed, reordered, or removed in this pane.

The drop-down list at the top of this tab shows all existing scope configuration definitions. When you choose a scope, the display configuration list changes to reflect the display configuration for the selected scope. You add new scopes in DAB using the Scope Configurations dialog box.

Figure 4-4. Scope Configuration Dialog Box in DAB
How the Docbaseattributelist Control Works

In WDK or a WDK application, a single docbaseattributelist control is used to display the category tabs and attributes that are defined for a particular scope context, such as an object type or user role.

The docbaseattributelist control has several XML resource files. Their use depends on the object type and the page on which the attributes are displayed. These XML resource files allow you to enable or disable data dictionary validation and to configure the display of attributes differently according to a qualifier value such as object type or the page on which the attributes appear.

The following table provides more information about each resource file.

Table 4-1. Docbaseattributelist Control XML Resource Files

<table>
<thead>
<tr>
<th>File Name</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes_docbaseattributelist.xml</td>
<td>attributes component</td>
<td>Base definition for display of attributes for sysobject subtypes</td>
</tr>
<tr>
<td>attributes_dm_document_docbaseattributelist.xml</td>
<td>attributes component (Properties page)</td>
<td>Extends base definition for object type dm_document; overrides &lt;category&gt; element</td>
</tr>
<tr>
<td>attributes_dm_folder_docbaseattributelist.xml</td>
<td>attributes component (Properties page)</td>
<td>Extends base definition for object type dm_folder; overrides &lt;category&gt; element</td>
</tr>
<tr>
<td>attributes_dm_smartlist_docbaseattributelist.xml</td>
<td>attributes component (Properties page)</td>
<td>Extends base definition for object type dm_smartlist; overrides &lt;category&gt; element</td>
</tr>
<tr>
<td>checkin_docbaseattributelist.xml</td>
<td>checkin component</td>
<td>Extends base definition and dm_document extension; overrides &lt;data_dictionary_population&gt; and &lt;category&gt; elements</td>
</tr>
<tr>
<td>import_docbaseattributelist.xml</td>
<td>import component</td>
<td>Extends base definition and dm_document extension; overrides &lt;data_dictionary_population&gt; and &lt;category&gt; elements</td>
</tr>
</tbody>
</table>

In this tutorial we are concerned with the base definition (contained in attributes_docbaseattributelist.xml) and the dm_document extension of that (contained in attributes_dm_document_docbaseattributelist.xml). The base definition contains two main configuration elements:

- `<data_dictionary_population>`: controls enabling or disabling of the data dictionary to display attributes and allows specification of attributes to be hidden from the display
- `<category>`: controls the display of attributes if the data dictionary is disabled.
The `<data_dictionary_population>` element in the base definition looks as follows:

```xml
<data_dictionary_population>
  <enable>true</enable>
  <!-- ddscope specifies scope class and scope value for retrieving -->
  <!-- the right data dictionary information from the content -->
  <!-- server. Currently, only application is valid in the name field -->
  <ddscopes>
    <ddscope name="application">webtop</ddscope>
  </ddscopes>
  <!-- add any attributes that shouldn't be shown in the UI here -->
  <!-- attribute name="attribute2"/ -->
</ignore_attributes>
</data_dictionary_population>
```

The data dictionary is enabled because the value of `<enable>` is “true”. Any attributes that should be hidden from display can be added to the `<ignore_attributes>` display in an extension that overrides this definition. The `<ddscope>` element’s value is used to configure the Application Name field.

The `<category>` element in the base definition looks as follows:

```xml
<category id="info">
  <name>nlslid>MSG_INFO</nlslid></name>
  <attributes>
    <attribute name="object_name"/>
    <attribute name="owner_name"/>
  </attributes>
  <moreattributes>
    <attribute name="r_creation_date"/>
    <attribute name="r_creator_name"/>
    <attribute name="r_modify_date"/>
    <attribute name="r_modifier"/>
  </moreattributes>
</category>
```

The information in the `<category>` element is ignored unless the data dictionary is disabled. When it is disabled, the attributes listed in the `<attributes>` element will be displayed as primary attributes, and the attributes listed in the `<moreattributes>` element will be displayed as secondary attributes, in the `[+]show more` section on the Properties page.

The dm_document extension in attributes_dm_document_docbaseattributelist.xml extends the base definition in order to override the `<category>` element for the purpose of adding some additional attributes for display. Since the `<data_dictionary_population>` element does not appear in this extension, it is inherited from the base definition. This means that the data dictionary will be enabled and the `<category>` element in the extension will be nonfunctional with a 5.2 Content Server or higher.

### Procedures Required for This Tutorial

To hide the subject attribute from the display of attributes in dm_document, you need to override the `<data_dictionary_population>` element in the base definition and add the subject attribute to the `<ignore_attributes>` element.

The following procedures are required to complete this tutorial.

- Create the appropriate structure in the custom layer.
• Extend the dm_document extension in the custom layer.
• Add the <data_dictionary_population> element from the base definition to this new extension.

Creating the Custom Layer Files

Use the following procedure to set up the custom layer.

To create the custom layer:
1. In WDK, copy the following file:
   
   virtual_dir/webcomponent/config/library/
   attributes_dm_document_docbaseattributelist.xml
   
   to the following location, with a new file name:
   
   virtual_dir/custom/config/
   attributes_dm_document_docbaseattributelist.xml

   Note: The exact name of the XML file is not important, because it is not used by the framework. You can assign any file name that will be meaningful to you.

Modifying the docbaseattributelist Control Definition

Use the following procedure to hide the subject attribute from the display.

To modify the docbaseattributelist definition:
1. Open virtual_dir/custom/config/attributes_dm_document_docbaseattributelist.xml.
2. Extend the definition by modifying the existing <attributelist> tag so it looks like the following:
   
   <attributelist id="attributes" extends="attributes:webcomponent/config/library/
   attributes_dm_document_docbaseattributelist.xml">

   In the <attributelist> element, the id attribute must have the same value as that used in the JSP page that displays the attribute list. In this case, we use the same ID that we are overriding from the dm_document attributelist definition.

3. Copy the following <data_dictionary_population> element in bold type (or copy it from the file virtual_dir/webcomponent/config/library/attributes_docbaseattributelist.xml) and insert it into the following location:
   
   <attributelist id="attributes" extends="attributes:custom/config/
   attributes_dm_document_docbaseattributelist_extended.xml">

   <data_dictionary_population>
   <enable>true</enable>
   <ddscopes>
     <ddscope name="application">webtop</ddscope>
   </ddscopes>
<ignore_attributes>
    <!-- specifies attributes that shouldn't be shown in the UI -->
</ignore_attributes>
</data_dictionary_population>

This overrides the <data_dictionary_population> element in the base definition.

4. Add the subject attribute to the <ignore_attributes> section:

   <ignore_attributes>
     <attribute name="subject"/>
   </ignore_attributes>

5. Save and close the file.

Testing the Configuration

 Whenever you change an XML resource file, you should reload the configuration service, described in the first step of the procedure below.

To test the new configuration:

1. Open a browser and refresh the configuration service by navigating to http://server_name:port_number/virtual_dir/wdk/refresh.jsp.
   If this page is already loaded, refresh the page.

2. Go to http://server_name:port_number/virtual_dir/.
   Note: If you have not logged in yet or your session has timed out, you will be presented with the login page before the drilldown page appears.

3. Make sure the default drilldown page displays, as in Figure 4-1, page 24.

4. Navigate to any Docbase folder containing a document of type dm_document and click the Info icon. The main list of attributes should appear without the Subject attribute.

Variations on This Tutorial

The Import and Checkin pages are not affected by the change made in this tutorial. This is because the XML resource file for the Checkin page (checkin_docbaseattributelist.xml) and the Import page (import_docbaseattributelist.xml) already list the subject attribute, as well as a number of other attributes, within the <ignore_attributes> element.
You can use these procedures to hide attributes when data dictionary validation is enabled in Webtop and Web Publisher.
Chapter 5

Adding an Attribute Column to Webtop Classic View

The following tutorial shows you one way to change the default view in Webtop from Streamline to Classic, and how to add the Title attribute to the display when viewing the contents of cabinets and folders. It also provides an example of how to create a custom layer in Webtop. The tutorial contains the following sections:

• Task Objective, page 33
• Technical Overview, page 35
• Creating the Custom Layer Files, page 36
• Changing the Default View in Webtop, page 37
• Displaying the Creator Attribute, page 37
• Variations on This Tutorial, page 38

Note: You must have Webtop installed in order to complete this tutorial.

Task Objective

There are two WDK components that can be used for viewing the contents of cabinets and folders. The drilldown component separates folders and their objects into two separate viewing areas. The WDK application start page that was created in Chapter 2, Creating a Start Page for a New WDK Application called the drilldown component, which displayed the page shown in Figure 4–1, page 24. It is also used for the Streamline view in Webtop and Web Publisher.

The doclist component displays folders and their objects with checkboxes to support multiple selection, shown in the following figure.
Adding an Attribute Column to Webtop Classic View

Figure 5-1. The Doclist Component in WDK

The doclist component is extended in Webtop and is called the objectlist component. This component is used for the classic view, as shown in the following figure.

Figure 5-2. The Objectlist Component in Webtop

The objectlist component provides additional functionality to update the location in the tree while navigating into a folder.

Suppose that you want to configure Webtop in the following way:

- Make the classic view the default for all users (before they select a preferred view)
- Add a column to the classic view for the attribute Creator

After logging in, the user should see a UI as in the following figure.

Figure 5-3. Classic View with Creator Attribute Displayed
Technical Overview

The following sections describe some of the technical details underlying the configuration.

Defining the Default View in Webtop

The Webtop start page (index.html or default.html) redirects users to the main component, which initiates a login sequence, then goes to the view specified in the main definition or by a user’s preferences.

The default view after login or timeout is specified in the main component definition by the following lines:

```xml
<pages>
  <start>/webtop/main/streamlineview.jsp</start>
  <classic>/webtop/main/classicview.jsp</classic>
  <streamline>/webtop/main/streamlineview.jsp</streamline>
</pages>

<entrypage>streamline</entrypage>
```

The default view can be changed by changing the <entrypage> value.

Note: There are several different ways to change the default view, including user preferences. For more information on views in Webtop, see Webtop Development Guide.

Anatomy of Webtop Classic View

The Webtop classic view contains a number of nested framesets, as shown in the following figure.

Figure 5-4. Framesets in Webtop Classic View
These framesets are controlled by several different JSP files, as shown in the following table.

<table>
<thead>
<tr>
<th>JSP</th>
<th>frameset</th>
</tr>
</thead>
<tbody>
<tr>
<td>classicview.jsp</td>
<td>titlebar + classic.jsp frameset + status</td>
</tr>
<tr>
<td>classic.jsp</td>
<td>toolbar + nested frameset containing browser + workarea.jsp</td>
</tr>
<tr>
<td>workarea.jsp</td>
<td>menubar + current content page</td>
</tr>
</tbody>
</table>

The following components can serve as content pages:
- objectlist (Cabinets icon in the classic view)
- homecabinet
- inbox
- myfiles
- subscriptions
- resultlist (results of a search query)
- objectgrid (history, relationships, renditions, viewassemblies, versions)

These components can have two definitions in Webtop, divided according to the view, for example myfiles_classic_component.xml and myfiles_streamline_component.xml.

When the classic view is displayed after login, the objectlist content page will be displayed. The objectlist component definition (objectlist_component.xml) is an extension of the doclist component in the webcomponent layer and contains a list of default Docbase attributes. The Title attribute already exists as a column in doclist_component.xml, but it is hidden from display. All you need to do is to set up the correct custom layer and unhide this attribute.

---

The Custom Layer for WDK Applications

Webtop is built on the WDK and webcomponent layers. The Webtop-specific configurations and source files that are installed with Webtop are stored in Webtop’s own configuration layer, virtual_dir/webtop.

When you create a custom layer for a product like Webtop, you use the same customization model as for WDK, placing files to be configured into the virtual_dir/custom folder.

---

Creating the Custom Layer Files

Use the following procedure to set up the custom layer to change the default view in the main component and unhide the Title attribute in the doclist component.
To create the files for the customization:

1. Copy the following two files:
   - virtual_dir/webtop/config/main_component.xml
   - virtual_dir/webtop/config/objectlist_component.xml

   to the following location:
   - virtual_dir/custom/config

   These files will be used to change the default view and display the Title attributes, respectively.

Changing the Default View in Webtop

The following procedure changes the default view from Streamline to Classic by changing the <entrypage> definition in the main component.

To change the default view in Webtop:

1. Open the following file:
   - virtual_dir/custom/config/main_component.xml

2. Extend the Webtop main component by modifying the <component> element in the following way:
   
   `<component id="main" extends="main:webtop/config/
    main_component.xml">

3. Add an <entrypage> element (from the wdk main component definition) as follows:
   
   `<entrypage>classic</entrypage>

4. Refresh the configuration service by logging out of Webtop and logging back in. Then enter the following URL:
   
   http://server_name:port_number/virtual_dir/wdk/refresh.jsp

   **Note:** This exercise assumes that you have not previously set a preference for the classic or streamline view to be displayed.

5. The Classic view is displayed after login.

Displaying the Creator Attribute

The attribute settings for the objectlist component are specified in the <columns> element of the component’s configuration file. Since the objectlist component is an extension of the doclist component, it uses the <columns> from the doclist by default. To unhide the Creator attribute, you will copy the <columns> element from the doclist component into your custom objectlist component and modify it, using the following procedure.

**Note:** Users can set a preference for the attributes that are displayed. This exercise sets the default attributes to be displayed. Make sure that you have set your preferences to the default list in order to see your results.
To display the Title attribute in Webtop:

1. Open the following file:
   
   ```
   virtual_dir/webcomponent/config/navigation/doclist/
   doclist_component.xml
   ```

2. Copy the entire `<columns>` element.

3. Open the following file:
   
   ```
   virtual_dir/custom/config/objectlist_component.xml
   ```

4. Position the cursor between the `<class>` and the `<helpcontextid>` elements and paste in the `<columns>` element from the doclist component definition.

5. In your same custom doclist component definition, extend the Webtop doclist component by modifying the `<component>` element in the following way:

   ```
   <component id="objectlist" extends="objectlist:webtop/config/
   objectlist_component.xml">
   ```

6. Locate the MSG_CREATOR_NAME line in the `<columns>` element, and change the `<visible>` element from false to true in the line beneath it:

   ```
   <!-- visibility and order of attribute columns -->
   <columns>
   <column>
     <attribute>object_name</attribute>
     <label><nlsid>MSG_NAME</nlsid></label>
     <visible>true</visible>
   </column>
   ... 
   <column>
     <attribute>r_creator_name</attribute>
     <label><nlsid>MSG_CREATOR_NAME</nlsid></label>
     <visible>true</visible>
   </column>
   ... 
   </columns>
   ```

   This unhides the display of the Creator attribute.

7. Save the file and close it.

8. Refresh the configuration service and test the page by navigating to a cabinet or folder in the classic view.

9. Click a Docbase cabinet or folder and verify that a Creator column is visible.

   **Note:** The Creator column will not be visible in the Home Cabinet, because that uses a different component for display.

Variations on This Tutorial

To make the Creator column appear by default in all of the content pages in the classic view, you need to make the same change to homecabinet_classic_component.xml (an extension of homecabinet_list_component.xml), inboxclassic_component.xml (an extension of inboxlist_component.xml), and so on.
You can make the same change to the streamline view. The definition for streamlineview in streamlineview_component.xml is an extension of the drilldown component in the webcomponent layer. You would copy streamlineview_component.xml into the custom layer, paste in the <columns> element from drilldown_component.xml, and change the <visible> element to true for the Title attribute.

WDK also has two views, displayed through the drilldown and doclist components. To add a Creator column to the doclist component, you would copy doclist_component.xml to the custom layer and make the same change for the Creator attribute in the <columns> element. To add the Creator attribute to the drilldown component, you would copy drilldown_component.xml to the custom layer and change the visibility of the Creator attribute.
Chapter 6

Creating an Action List Component

This tutorial requires both configuration and customization to display a list of all actions configured in your Web application. Each action has a link to an information page about the action that displays the action class, parent action definition, container and component launched, if any, parameters, and other configurable elements.

The full source code for this customization is located on the Documentum developer site, developer.documentum.com.

The tutorial contains the following sections:

- Task Objective, page 41
- Technical Overview, page 42
- Creating the Component Definition, page 42
- Creating the Component Layout, page 43
- Creating the Component Class, page 45
- Supporting Navigation Between Pages, page 47
- Creating the Action List Datagrid, page 47
- Creating the Action Info Datagrid, page 50
- Creating the Resource Bundle, page 54
- Testing the Customization, page 55
- Variations on This Tutorial, page 55

Task Objective

In this tutorial, you will create a simple action list component that reads information from all of the action definition files in your application and displays them in two UI pages: a list of actions, and a drilldown page that displays info for an action.

After customization, the action list component start page will look similar to the one shown in the following figure:
Creating an Action List Component

Figure 6-1. Action List Page

Number of Action Definitions: 237

Creating the Component Definition

To create a new WDK actionlist component, you must create a component definition. The best way to create a component definition is to copy the configuration file for a similar component, then make any necessary changes.
To create the actionlist definition:

1. Copy the componentlist component configuration file from `/webcomponent/config/ componentList/component_list.xml` to the following location: `/custom/config`.
2. Rename the file to `actionlist_component.xml`.
3. Open the file and locate the `<desc>` element. Change it to the following content:
   
   Action list: Displays outline information on all of the defined actions in the application

4. Locate the `<component>` element. Change it to the following content:
   
   `<component id="actionlist"/>

5. Change the value for the `<pages><start>` element to the following:

   `/custom/actionlist/actionlist.jsp`

6. Locate the `<pages>` element and replace the `<componentinfo>` element and contents with the following element and contents:

   `<actioninfo>/custom/actionlist/actionInfo.jsp</actioninfo>`

7. Locate the `<class>` element and change the value to the following class that you will create:

   `com.mycompany.actionlist.ActionList`

8. Locate the `<nlsbundle>` element and change the value to the following NLS resource that you will create:

   `com.mycompany.actionlist.ActionListNlsProp`

9. Save and close the file.

Creating the Component Layout

You can modify the componentlist layout pages to provide a similar layout for the action list.

To create the layout for the action list:

1. Copy the componentlist JSP page from `/webcomponent/componentList/ componentlist.jsp` to `/custom/actionlist`. Create a `/custom/actionlist` directory if one does not exist.
2. Rename the file to `actionlist.jsp`, and open the file.
3. Locate the following file import tag:

   `<%@ page import="com.documentum.webcomponent.componentList.ComponentList" %>`

   Change the imported class to the following:

   `com.mycompany.actionlist.ActionList`

4. Locate the following tag:

   `<dmf:label nlsid='MSG_NUM_COMPONENTS'/>: <%= "" + ComponentList.getNumberOfComponents() %>

   Change the tag contents to the following:
Creating an Action List Component

<dmf:label nlsid='MSG_NUM_ACTIONS'/>: <%= "" + ActionList.getNumberofActions()%>

5. Locate following tag:

<dmf:datagrid name=' <%=ComponentList.COMPLIST_GRID%>' ...

Change the name of the datagrid from ComponentList.COMPLIST_GRID to ActionList.ACTIONLIST_GRID

6. Remove the table header rows (<th ...> ... </th>) that contain the following labels (for example, dmf:label nlsid=):

MSG_ISCONTAINER
MSG_ISCONFIGURABLE
MSG_DESCRIPTION

Reason: These columns are not relevant for actions.

7. Locate the label tag (<dmf:label...>) with the following nlsid attribute value:

MSG_COMPONENT_ID

Change the nlsid value to the following:

MSG_ACTION_ID

8. Locate the <dmf:link> tag with the onclick event handler value onClickCompId.

Change the onclick event handler value to onClickActionId. Change the datafield value from compid to actionid. The action ID will be displayed for the link.

Change the inner <dmf:argument> name from compcfgid to actioncfgid. Change the datafield from compcfgid to actioncfgid. The indexed action ID will be passed to the event handler.

9. Locate the <dmf:label> tag whose name attribute value is compscope.

Change the name attribute value to actionscope and the datafield value to actionscope.

10. Locate the <dmf:link> tag whose onclick event handler is onClickXML.

Change the datafield value from compxml to actionxml, which is the base URL for the action. The link will display the base URL.

Change the inner <dmf:argument> name from compxmlurl to actionxmlurl and the datafield from compxmlurl to actionxmlurl. This inner argument will pass the full URL to the event handler.

11. Remove the table cells (<td>... </td>) for isContainer, isConfigurable, and description. These labels are not applicable to an action definition. Do not delete the cells between the label isconfigurable and the label desc. In other words, delete the following lines only:

  <td nowrap=""/>
  <dmf:label name="iscontainer" datafield="iscontainer"/>
  <td nowrap=""/>
  <dmf:label name="isconfigurable" datafield="isconfigurable"/>
  ...

  //leave these lines in
  <td nowrap=""/>
  <dmf:label name="desc" datafield="desc"/>

12. Save and close the JSP file.
To create the layout for action info:

1. Copy the componentinfo JSP page from /webcomponent/componentList/componentinfo.jsp to the following location: /custom/actionlist
2. Rename the file actioninfo.jsp and open the file in the editor.
3. Locate the following file import tag:
   `<%@ page import="com.documentum.webcomponent.componentList.ComponentList" %>`
   Change the imported class to the following:
   `com.mycompany.actionlist.ActionList`
4. Locate following tag:
   `<dmf:datagrid name='<%=ComponentList.COMPINFO_GRID%>'`...
   Change the name of the datagrid from ComponentList.COMPINFO_GRID to ActionList.ACTIONINFO_GRID.
5. Locate the `<dmf:link>` tag whose name is lnkback.
   Change the value of the onclick attribute from onClickCompList to onClickActionList
6. Save and close the file.

Creating the Component Class

The following procedure uses the ComponentList class as a model for the actionlist component behavior class.

Caution: You can copy and paste code from the PDF tutorial, but certain symbols may be changed when you paste them into your IDE. For example, the symbol “&” may be changed to &amp; and the “<” symbol may be changed to &lt;. The single quote (&apos;) may be changed to a curly quote. The NetBeans IDE compiler will show the line containing the error. You must ensure that these symbols are correct in your IDE before you compile.

You can get the full source code for this tutorial on the Documentum developer site, [developer.documentum.com](http://developer.documentum.com).

To create the component class:

1. In the NetBeans IDE, create a directory /WEB-INF/classes/com/mycompany/actionlist. .
2. Create a new Java class by right-clicking on the directory WEB-INF/classes/com/mycompany/actionlist and selecting New⇒Java Class.
3. Name the object ActionList and click Finish.
   The file will be created with the following lines:
   ```java
   package com.mycompany.actionlist;
   public class ActionList
   {
   public ActionList()
   {
   }
   ```
Note: If the package line appears different from that shown above, then either the WEB-INF classes directory was not mounted correctly, or you created the file within the wrong directory.

4. In the Source Editor window, following the package declaration, add the following import statements:

```java
import com.documentum.web.common.ArgumentList;
import com.documentum.web.form.control.Link;
import com.documentum.web.form.control.databound.DataTableSet;
import com.documentum.web.formext.component.Component;
import com.documentum.web.formext.config.ConfigFile;
import com.documentum.web.formext.config.ConfigService;
import com.documentum.web.formext.config.IConfigElement;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Iterator;
```

5. Next, edit the class definition that was created by NetBeans, adding an extends clause and some class variables:

```java
public class ActionList extends Component {
    public static final String ACTIONLIST_GRID = "_actionlist_grid";
    public static final String ACTIONINFO_GRID = "_actioninfo_grid";
    private HashMap m_hashCfgElementLookup;
    private HashMap m_hashCfgScopeLookup;
    public ActionList() {
    }
}
```

6. Within the class constructor that was provided by NetBeans IDE when you created the class, instantiate hash maps to hold the action IDs and scopes:

```java
public ActionList() {
    m_hashCfgElementLookup = new HashMap(23);
    m_hashCfgScopeLookup = new HashMap(23);
}
```

7. After the constructor, add an onInit() implementation that gets or creates the datagrids on both of the JSP pages:

```java
public void onInit(ArgumentList args) {
    super.onInit(args);
    getControl("_actionlist_grid",
        com.documentum.web.form.control.databound.DataTableSet.class);
    getControl("_actioninfo_grid",
        com.documentum.web.form.control.databound.DataTableSet.class);
    //updateActionList();
}
```

The last call is to the updateActionList() method, which has not yet been created. This call must be commented out for now. See Creating the Action List Datagrid, page 47 for information on the updateActionList() method, which creates or updates the actionlist datagrid.

8. Add a method that counts the number of actions that are defined in the application:
public static int getNumberOfActions()
{
    return ConfigService.getPrimaryElementIds("action").length;
}

9. Compile your class file. Note any errors in the output window and resolve them. The exact location of the error is noted by the compiler.

Supporting Navigation Between Pages

The actionlist JSP page lists all actions, and each action contains a link to the info page. The link requires an event handler, onClickActionId().

The actioninfo page displays information for an action and contains a link to return to the actionlist page. This link requires an event handler, onClickActionList().

To add navigation support between layout pages:

1. Within the class implementation, add an event handler for the onClickActionId event that is launched by a link on the actionlist.jsp page. This event handler gets the selected action ID, calls a method that gets the info for the selected action, and navigates to the actioninfo JSP page:

   public void onClickActionId(Link link, ArgumentList arg)
   {
       String strCfgid = arg.getString("actioncfgid");
       //updateActionInfo(strCfgid);
       setComponentPage("actioninfo");
   }

   Note: You will uncomment the call to the updateActionInfo() method after you have created that method and its helper methods.

2. Add an event handler for the onClickActionList event that is launched by a link on the actioninfo.jsp page. This event handler navigates back to the actionlist JSP page:

   public void onClickActionList(Link link, ArgumentList arg)
   {
       setComponentPage("start");
   }

3. Right-click and compile your class file. Note any errors in the output window and resolve them. The exact location of the error is noted by the compiler.

Creating the Action List Datagrid

In the ActionList class, you must add several methods that populate or update the action list datagrid. (The datagrid displays a list of all actions.) The main method, updateActionList(), gets the data, and several helper methods get information to provide to the datagrid.
Creating an Action List Component

To update the action list:

1. Uncomment the following line from the onInit() method by removing the "//" symbols:

   //updateActionList();

2. Create the update method signature and get all action IDs from the ConfigService in-memory tables:

   protected void updateActionList()
   {
   String strActionIds[] = ConfigService.getPrimaryElementIds("action");
   Arrays.sort(strActionIds);
   }

3. Create datafields for values of elements in the action definitions: a datafield for indexed action ID, action ID, scope, base action definition URL, and full path to the action definition URL. Add the following lines to the unfinished method updateActionList():

   String strDataFields[] = {
   "actioncfgid", "actionid", "actionscope", "actionxml", "actionxmlurl"
   };

4. Create a TableResultSet to provide the data to the datagrid. Add the following line to the unfinished method updateActionList():

   TableResultSet rsList = new TableResultSet(strDataFields);

5. Iterate through the array of action IDs. Put the definition for each action ID into an array and put the scope for each definition into another array. Add the following lines to the unfinished method updateActionList():

   for(int idxActionId = 0; idxActionId < strActionIds.length; idxActionId++)
   {
   String strActionId = strActionIds[idxActionId];
   IConfigElement icfgEls[] = ConfigService.getPrimaryElements("action[id=" + strActionId + "]");
   String strScopes[] = ConfigService.getPrimaryElementScopes("action[id=" + strActionId + "]");
   }

6. Iterate through the array of configuration definitions and put each action ID and scope into a hash map. Add the following lines to the unfinished method updateActionList():

   for(int idxEl = 0; idxEl < icfgEls.length; idxEl++)
   {
   ArrayList listRow = new ArrayList(strDataFields.length);
   String strCfgElId = strActionId + "." + idxEl;
   m_hashCfgElementLookup.put(strCfgElId, icfgEls[idxEl]);
   m_hashCfgScopeLookup.put(strCfgElId, strScopes[idxEl]);
   }

7. Add to the unfinished updateActionList() method list rows for the following data, as shown in the code snippet below:

   - Indexed action ID (strCfgElId), because there are duplicate action IDs when they have multiple scopes
   - Action ID (for display; uses indexed ID for retrieval)
   - Scope
Creating an Action List Component

- Configuration file link text (for display; uses full path to configuration file for retrieval)
- Full path to configuration file

```java
listRow.add(strCfgElId);
listRow.add(strActionId);
listRow.add(strScopes[idxEl]);
listRow.add(getConfigFilePath(icfgEls[idxEl]));
listRow.add(getConfigFileUrl(icfgEls[idxEl]));
```

You will see red error symbols next to the lines that call getConfigFilePath() and getConfigFileUrl(). You have not yet created these helper methods.

8. Add the row of data for the unique action definition to the result set, and after iteration set the result set on the datagrid. Add the following lines to the unfinished method updateActionList():

```java
rsList.add(listRow);
}
}

{{Datagrid}getControl("_actionlist_grid", com.documentum.web.form.control.databound.Datagrid.class).getDataProvider().setResultSet(rsList, null);
}

Note: Do not compile your class file. You must add some supporting methods before your class will compile properly.

Three methods are used to generate the configuration file path for display and the configuration file URL for the link target. Add these methods after the closing bracket of the updateActionList() method:

**To support the action list update:**

1. getConfigFile() returns the configuration file for the primary element. Add the following method after the updateActionList() method:

```java
private ConfigFile getConfigFile(IConfigElement icfgEl)
{
    ConfigFile file;
    for(file = null; file == null && icfgEl != null;
        icfgEl = icfgEl.getParent())
    {
        if(icfgEl instanceof ConfigFile)
        {
            file = (ConfigFile)icfgEl;
        }
    }
    return file;
}
```

2. getConfigFilePath() uses ConfigFile.getPathName() to read the path to the configuration file on the file system:

```java
private String getConfigFilePath(IConfigElement icfgEl)
{
    ConfigFile file = getConfigFile(icfgEl);
    String strXMLFileName = file.getPathName().replace('\\', '/');
    return strXMLFileName;
}
3. `getConfigFileUrl` returns the URL that will launch the configuration file in the browser when the user clicks the config file link (browser must be IE to display an XML file)

```java
private String getConfigFileUrl(IConfigElement icfgEl) {
    String strBaseAppDir = getPageContext().getServletContext().getRealPath("");  
    if(strBaseAppDir != null) { 
        strBaseAppDir = strBaseAppDir.replace('\', '/');   
    } else { 
        strBaseAppDir = "";
    }
    String strXMLFileName = getConfigFilePath(icfgEl);  
    int idxBaseAppDir = strXMLFileName.indexOf(strBaseAppDir); 
    String strUrl = strXMLFileName;
    if(idxBaseAppDir == 0) { 
        strUrl = strXMLFileName.substring(strBaseAppDir.length());
        strUrl = getBaseUrl() + strUrl;
    }
    return strUrl;
}
```

4. At this point, you should have no red markers for errors in your class file. If you do, check your code against the class file on the Developer Web site and correct your errors. Right-click and compile your class file. Note any errors in the output window and resolve them. The exact location of the error is noted by the compiler.

---

**Creating the Action Info Datagrid**

Continuing the same action class from the previous section, you must add several methods that populate or update the action info datagrid. The main method, `updateActionInfo()`, gets the data, and several helper methods get information to provide to the datagrid.

**To update the action info page:**

1. Uncomment the following line from the method `onClickActionId()`:

   ```java
   updateActionInfo(strCfgId)
   ```

2. Create the update method signature and get the selected action ID from the hash table that was populated by `updateActionList()`:

   ```java
   protected void updateActionInfo(String strCfgId) {
       IConfigElement cfgel = {
           IConfigElement)m_hashCfgElementLookup.get(strCfgId);
   ```

3. Create a datafield string array and a `TableResultSet` to provide the data to the datagrid. Add the following lines to the unfinished `updateActionInfo()` method:

   ```java
   String strDataFields[] = {
       "propname", "propvalue"};
   TableResultSet rsList = new TableResultSet(strDataFields);
   ```
4. Add list rows for the following data, as shown in the code snippet below. (Helper methods that provide the data are described in the next procedure.). Add the following lines to the unfinished updateActionInfo() method:

- Action ID
- Scope
- Action execution class
- Component launched by the action, if any
- Container launched by the action, if any
- XML file that contains the action definition
- List of parameters, with optional or required label

```java
addRowFromAttribute(rsList, cfgel, "MSG ACTION_ID", "id");
rsList.add(new String[] {getString("MSG_SCOPES"), (String)m_hashCfgScopeLookup.get(strCfgId)});
rsList.add(new String[] {getString("MSG_EXTENDS"), getExtends(cfgel)});
rsList.add(new String[] {getString("MSG_CLASS"), getActionClass(cfgel)});
rsList.add(new String[] {getString("MSG_EXECCOMP"), getLaunchedComponent(cfgel)});
rsList.add(new String[] {getString("MSG_EXECCONTAINER"), getLaunchedContainer(cfgel)});
rsList.add(new String[] {getString("MSG_XML_FILE"),getConfigFilePath(cfgel)});
addParameterRows(rsList, cfgel);
```

You will see red error symbols next to the lines containing calls to nonexistent methods. You will add those methods later. The getConfigFilePath() method was created in an earlier procedure.

5. Set the result set on thedatagrid by adding the following lines to finish the method:

```java
((Datagrid)getControl("_actioninfo_grid", com.documentum.web.form.control.databound.Datagrid.class)).
getDataProvider().setResultSet(rsList, null);
```

**Note:** Do not compile your class file. You must add some supporting methods before the class will compile properly

Several methods are used to generate row data for the infodatagrid. Add the following methods after the closing bracket of the updateActionInfo() method:

**To support the action info update**

1. Add the method addRowFromAttribute(), which adds a row containing the value of the supplied attribute.

   ```java
   private void addRowFromAttribute(
       TableResultSet rsList, IConfigElement cfgAction, String strTitleNlsId, 
       String strAttributeName)
   {
       String strValue = cfgAction.getAttributeValue(strAttributeName);
       if(strValue != null &\& strValue.length() > 0) 
       {
           rsList.add(new String[] {getString(strTitleNlsId), strValue});
       }
   }
   ```
2. Ad the method `getExtends()`, which gets the value of the extends attribute. This will display the name of the action that the current action extends.

```java
private String getExtends(IConfigElement icfgEl) {
    String strExtends = icfgEl.getAttributeValue("extends");
    if(strExtends != null) {
        return strExtends;
    } else {
        return getString("MSG_NONE");
    }
}
```

3. Add the method `getActionClass()`, which gets the fully qualified action class name.

```java
private String getActionClass(IConfigElement icfgEl) {
    IConfigElement cfgExec = icfgEl.getChildElement("execution");
    if(cfgExec != null) {
        String strClass = cfgExec.getAttributeValue("class");
        if(strClass != null) {
            return strClass;
        } else {
            return getString("MSG_NONE");
        }
    } else {
        return getString("MSG_NONE");
    }
}
```

4. Add the method `getLaunchedComponent()`, which gets the name of the component that is launched by the action, if any.

```java
private String getLaunchedComponent(IConfigElement icfgEl) {
    IConfigElement cfgExec = icfgEl.getChildElement("execution");
    if(cfgExec != null) {
        String strComp = cfgExec.getChildValue("component");
        if(strComp != null) {
            return strComp;
        } else {
            return getString("MSG_NONE");
        }
    } else {
        return getString("MSG_NONE");
    }
}
```

5. Add the method `getLaunchedContainer()`, which gets the name of the container that is launched by the action, if any.
private String getLaunchedContainer(IConfigElement icfgEl) {
    IConfigElement cfgExec = icfgEl.getChildElement("execution");
    if(cfgExec != null) {
        String strContnr = cfgExec.getChildValue("container");
        if(strContnr != null) {
            return strContnr;
        } else {  
            return getString("MSG_NONE");
        }
    } else {
        return getString("MSG_NONE");
    }
}

6. Add the method addParameterRows(), which reads the parameters from the XML file.

private void addParameterRows(
    TableResultSet rsList, IConfigElement cfgAction) {
    IConfigElement iParams = getApplicableChildElement(cfgAction, "params");
    String strNlsParams = getString("MSG_PARAMS");
    if(iParams != null) {
        int nParams = 0;
        for(Iterator itParams = iParams.getChildElements("param"); itParams.hasNext();)
            {
            IConfigElement iParamEl = (IConfigElement)itParams.next();
            String strParam = iParamEl.getAttributeValue("name");
            String strRequired = iParamEl.getAttributeValue("required");
            if(strRequired != null) {
                strParam = strParam + (strRequired.equalsIgnoreCase("true") ? " (required)" : " (optional)");
                rsList.add(new String[] { strNlsParams, strParam });
            }
            nParams++;
        }
    }
}

7. Add the method getApplicableChildElement(), which is a helper method that gets multiple parameter values:

private IConfigElement getApplicableChildElement(IConfigElement cfgAction,
    String strCfgElementName) {
    IConfigElement el;
    for(el = null; cfgAction != null && el == null;
        cfgAction = ConfigService.getExtendedElement(cfgAction)) {
        el = cfgAction.getChildElement(strCfgElementName); 
    } return el;
}
8. At this point, you should have no red markers for errors in your class file. If you do, check your code against the class file on the Developer Web site and correct your errors. Right-click and compile your class file. Note any errors in the output window and resolve them. The exact location of the error is noted by the compiler.

**Creating the Resource Bundle**

In *Creating the Component Definition*, page 42 you specified an NLS resource bundle that will contain the strings for your component: com.mycompany.actionlist.ActionListNlsProp.

**To create a properties file in the new directory**

1. Open the NetBeans explorer Filesystems tab.
2. Create the directory /custom/strings/com/mycompany/actionlist.
3. Right-click the mycompany directory, and select New→All Templates.
4. Open the Other templates group and click Text File.
5. Click Next and name the file ActionListNlsProp.
6. Click Next and add the properties extension.
7. Click Finish.
   Because you added the properties extension, NetBeans displays a page for editing property values. You can also open this page as a text file by right-clicking the file in the tree and selecting Edit.
8. Add each of the following key and value pairs by clicking the New Property button and filling in the dialog box that appears:

   MSG_NUM_ACTIONS = Number of Action Definitions
   MSG_ACTION_ID=Action ID
   MSG_SCOPES=Scopes
   MSG_ISCONFIGURABLE=Is configurable
   MSG_XML_FILE=XML file
   MSG_DESCRIPTION=Description
   MSG_EXECCOMP=Execution Component
   MSG_YE=YES = Yes
   MSG_NO = No
   MSG_NONE = None
   MSG_BACKTO_LIST = Back To Action List
   MSG_PARAMS = Parameters
   MSG_EXTENDS = Extends
   MSG_CLASS=Class
   MSG_EXECCONTAINER=Execution Container
   MSG_NLSCLASS=NLS Class
   MSG_NLSBUNDLE=NLS Bundle
   MSG_NONE=No

9. Save and close the file.
Creating an Action List Component

Testing the Customization

The full source code for this customization is located on the Documentum developer site, developer.documentum.com.

To refresh files and test the actionlist component:

1. If you have changed only XML files, refresh the configuration service by navigating to http://server_name:8080/virtual_dir/wdk/refresh.jsp.

2. If you have created and compiled (or edited and recompiled) the ActionList class, or if you have made changes to the properties file, stop and restart the Tomcat server.

3. Log in and navigate to the actionlist component with the following URL:
   http://localhost:8080/virtual_dir/component/actionlist

During your development, changes to a JSP page or changes to a Java class that provides values to the JSP page, require that you delete the generated Java class file that represents the compiled JSP page in order to see your changes. Changes to an NLS resource bundle require that you reload the Web application in the server.

If the strings do not display properly, for example, xxMSG_NUM_ACTIONSxx instead of Number of Actions, check the path to the properties file that you created. It must match the fully qualified NLS property resource bundle in the actionlist component definition.

Variations on This Tutorial

You could simplify the string that is displayed for the action definition file on the actionlist page. Because the link passes the full URL, the full path to the file does not need to be displayed.

In a more extensive customization, you could extend this component to test action definitions by passing required and optional parameters. You could add a text input tag for each parameter and add a test action button whose action is the name of the action itself. Because many action IDs have variations based on scope, you could add a scope field to the test, but this can present problems if the user enters a scope that is not compatible with one of the input parameters.
Creating an Action List Component
Chapter 7

Using Tracing and Logging in a Component

This tutorial demonstrates how to add tracing and logging to your component to make it easier to detect runtime errors.

The full source code for this customization is located on the Documentum developer site, developer.documentum.com.

The tutorial contains the following sections:
- Task Objective, page 57
- Technical Overview, page 57
- Creating the Tracing Class, page 58
- Adding Tracing to the Component Class, page 59
- Adding Custom Tracing to the List, page 60
- Turning on Tracing, page 60
- Examining the Trace Log, page 61
- Testing the Customization, page 60
- Variations on This Tutorial, page 61

Task Objective

In this tutorial, you will create a tracing class for your custom code and add a tracing flag to your code, then turn on tracing and inspect the log.

Technical Overview

WDK tracing flags trace the following types of operations or content manipulation: sessions, JSP requests, locales, actions, configuration, roles, preferences, resources, clipboard, controls, control tags, form navigation and history, validation, repository attributes, content transfer, components, containers, object caching, virtual links, and failover. For the full list of tracing flags and their usage, see Web Development Kit and Client Applications Development Guide.
You can add your own tracing flags to trace operations that are used in more than one class. After you enable tracing, tracing statements will be written to the wdk.log file in your DOCUMENTUM_HOME/config directory. (You selected a DOCUMENTUM_HOME directory when you first installed a Documentum product.)

WDK tracing flags are enumerated in the WDK resource file TraceProp.properties located in /WEB-INF/classes/com/documentum/debug. This file contains all tracing flags that are defined in your application. If there is an unknown flag in this file, the Trace class initialization will generate a warning message but will continue.

**Note:** You must enable tracing for the current session using one of the following methods:
- Set the SESSION flag (mandatory) and another other flags you require in TraceProp.properties and then restart the application server.
- Use a browser to navigate to /wdk/tracing.jsp and check the boxes that enable tracing.

You can enable tracing for all sessions for setting SESSIONENABLEDBYDEFAULT to true in /WEB-INF/classes/com/documentum/debug/TraceProp.properties.

The open source Apache logging library log4j is installed by the DFC installer. This package allows you to enable logging at runtime without modifying the application library or incurring a significant performance impact. The Apache log4j library is used by the DFC logger class DfLogger. Each log4j Logger class method such as debug() and warn() is wrapped by a DfLogger method. The WDK Trace class uses DfLogger to write the log file for all enabled traces.

The log file location is specified in a log4j.properties file, which is installed by the WDK and client applications installers to DOCUMENTUM_HOME/config. By default, the log file name (not path) is specified in log4j.properties as the value of the key log4j.appender.file.File, for example, C:\\Document\\logs\\trace.log (the double backslashes indicates an escaped backward slash).

**Creating the Tracing Class**

**Caution:** You can copy and paste code from the PDF tutorial, but certain symbols may be changed when you paste them into your IDE. For example, the symbol “&” may be changed to &amp; and the “<” symbol may be changed to &lt;. The single quote (&apos;) may be changed to a curly quote. TheNetBeans compiler will show the line containing the error. You must ensure that these symbols are correct in your IDE before you compile. Instead of copying and pasting, type in the content or get it from the code zip file on the developer Web site.

**To create the component class:**

1. Create a new Java class by right-clicking on the directory WEB-INF/classes/ mycompany and selecting New⇒Java Class.
2. Name the class “Trace” and click **Finish**.
   The file will be created with the following line:
   ```java
   package com.mycompany;
   ```
Using Tracing and Logging in a Component

Note: If the package line appears different from that shown above, then either the WEB-INF classes directory was not mounted correctly, or you created the file within the wrong directory.

3. Edit the class definition that was created by NetBeans, adding an extends clause:
   ```java
   public class Trace extends com.documentum.web.common.Trace
   ```

4. After the constructor, add the following tracing flag:
   ```java
   public static boolean ACTIONLIST;
   ```

5. Compile your class file. Note any errors in the output window and resolve them. The exact location of the error is noted by the compiler.

Adding Tracing to the Component Class

Add tracing statements for the main methods in your component class or for methods that may throw exceptions. In this tutorial you will add tracing statements to the ActionList class that you created in the previous tutorial. You will add tracing statements to the update method and to the methods that handle links and navigation.

Note: For this example to work, you must create all of the files required for the tutorial actionlist component. Alternatively, you can create a tracing flag that is appropriate for one of your custom classes, and add it to your class.

To Add Tracing to the Update Method:

1. Open ActionList.java in your IDE and locate the method updateActionList().
2. Import your Trace class by adding the following import statement:
   ```java
   import com.mycompany.Trace;
   ```
3. Locate the following line in the method:
   ```java
   TableResultSet rsList = new TableResultSet(strDataFields);
   ```
4. Add the following lines:
   ```java
   if (Trace.ACTIONLIST) {
     if (rsList != null) {
       Trace.println("Result set successfully created");
     } else {
       Trace.println("Result set not created");
     }
   }
   ```

To Add Tracing to the Navigation Event Handler:

1. Locate the method onClickActionList(). This event handler handles the onclick event for the Inlinkback link on actionInfo.jsp.
2. Before the first line in the method, add the following lines:
   ```java
   if (Trace.ACTIONLIST) {
   ```
Adding Custom Tracing to the List

To make your tracing flags accessible, so that you can turn them on or off, you must add them to a tracing properties file. Open the properties file TraceProp.properties in /WEB-INF/classes/com/documentum/debug

Add your flag on the last line, as shown in the following example:

```
com.mycompany.Trace.ACTIONLIST=true
```

Turning on Tracing

You must restart your application server to apply changes to a properties file.

There are two ways to turn on tracing in your application:

- Set the appropriate tracing flags to true in /WEB-INF/classes/com/documentum/debug/TraceProp.properties.
- Navigate to /wdk/tracing.jsp and select the first checkbox, **Tracing is enabled for current session**. Then check relevant tracing flags, such as ACTIONLIST. See the example below.

**Figure 7-1. Tracing JSP Page**

```
Tracing flag settings

☐ Tracing is enabled for current session

... 

☐ MESSAGING

☐ ACTIONLIST

com.mycompany.webshop.Trace

com.mycompany.Trace
```

Testing the Customization

The full source code for this customization is located on the Documentum developer site, [developer.documentum.com](http://developer.documentum.com).

**To refresh files and test tracing:**

1. Because you have added a new class, stop and restart the Tomcat server.
2. Log in and navigate to the actionlist component:

```
http://localhost:8080/virtual_dir/component/actionlist
```
3. Exercise the actionlist component by clicking on an action, returning to the start page, and then clicking on the XML file link.

If you add tracing statements to you behavior class, you must recompile and stop and restart the server to see your tracing statements in the log.

**Examining the Trace Log**

Trace statements are written to the Tomcat console, so you can see them as you exercise your component. After you stop the application server, you can examine the log file for your trace statements.

Open the tracing log in the location specified in your DOCUMENTUM_HOME/config/log4j.properties file. By default, this location is the value of log4j.appender.file.File, for example:

```java
log4j.appender.file.File=C:\/Documentum/logs/wdk.log
```

The following except from wdk.log shows the statements added in this tutorial:

```
```

The resulting trace statements are also written to the Tomcat console, as shown below.

**Variations on This Tutorial**

You can add JavaScript tracing for client-side functions. The trace JavaScript file trace.js is referenced in every HTML page that is output by the WebformTag class.

To output client-side tracing messages to a popup browser window, call the Trace_println function, passing in the message as the sole parameter. The following example is added to the actionList.jsp page to output JavaScript tracing when the user clicks the link to the action configuration file:

```javascript
function onClickXML(obj, xmlurl)
{
  Trace_println ("You are now viewing the XML configuration file");
  window.open(xmlurl);
}
```

A popup browser window is opened with the tracing statement:
Figure 7-3. JavaScript Tracing

You are now viewing the XML configuration file
Chapter 8

Troubleshooting

If your customization does not appear as you intended, you can try some of the procedures below to troubleshoot:

• Did you precompile your Java class?, page 63
• Did you refresh the Configuration Service?, page 63
• Did you remove generated files?, page 63
• Did you clear the browser cache?, page 64
• Did you check the name and location of the XML resource file?, page 64

For more troubleshooting tips, see the PDF manual Troubleshooting WDK 5 Applications.

Did you precompile your Java class?

If you create a new Java file, you should remember to compile it in Forte for Java. (Make sure it is compiled into the right directory) On the other hand, you should allow Tomcat 4 to compile your JSP pages automatically, because Forte for Java supports Tomcat 3.

Did you refresh the Configuration Service?

If you make changes to an XML resource file, you need to restart the configuration service, either by navigating to http://server_name:port_number/virtual_dir/wdk/refresh.jsp or by stopping and restarting the application server.

Did you remove generated files?

Normally, the application server picks up changes to JSP files. However, JSP pages may not be updated when they are called from a container. If you edit JSP files and the changes are not reflected in the display, you may need to clear the cache by deleting the directory tomcat_home_directory/work/Standalone/localhost/virtual_dir.
Did you clear the browser cache?

If you make changes to JavaScript, you should try clearing your browser cache, as described in the following procedure.

To clear the browser cache in Internet Explorer:
1. In Internet Explorer, choose Tools→Internet Options.
2. In the Temporary Internet files pane, click Delete Files.
3. Check the box to delete offline content and click OK.

Did you check the name and location of the XML resource file?

All customized XML resource files should be located in the `tomcat_home_directory/webapps/virtual_dir/custom/config` directory, where `tomcat_home_directory` is the directory in which Tomcat is installed, and `virtual_dir` is the name of the virtual directory that was provided when WDK was installed.

One way to test whether this resource file is being recognized is to make a copy of it in the same directory, then refresh the configuration service by navigating to `http://server_name:port_number/virtual_dir/wdk/refresh.jsp`. You should get an error message that there is a duplicate element, and the name of this copied file will be mentioned near the end of the error message. This shows that your XML resource file is being read. Delete the copy and continue troubleshooting.

If your XML resource file is being read, then you can assume that you will get a 404 error if the browser does not find the JSP file in the specified location, and a Java error if it does not find the class file in the specified location.
A
application start page, 15
attribute column
adding in Webtop, 33
attributes
hiding, 23

B
browser cache
clearing, 64

C
caches
clearing, 10
clearing caches, 10
Configuration Service
refreshing, 63
Content Server Display
Configuration, 25
conventions
typographical, 9
custom layer, 11
in WDK applications, 36
inheritances and overrides, 11

D
data dictionary display, 23
docbaseattributelist control, 27
modifying, 29
Docbases
limiting selection, 41
doclist component, 33
drilldown component
extending, 19
drilldown page
typical, 24
drilldown, changing generic_actions.xml, 20

E
environment, 9

F
Forte for Java
setting up, 12

G
generated files
removing, 63

I
IDE
setting up, 12
Info page
dm_document, 24

J
Java class files
precompiling, 63

L
logging, 57
logout action
extending, 22
logout link
adding, 17

O
objectlist component, 34

P
preparing for the tutorials, 9
project
creating, 12
R
refreshing the Configuration Service, 63

troubleshooting, 63
typographical conventions, 9

S
start page, 15
Sun ONE Studio
setting up, 12

W
Webtop
adding attribute column, 33
Classic view, 33
custom layer, 36

T
Tomcat application server
stopping, 13
tracing, 57

X
XML resource file
checking name and location, 64