Customizing the WebSphere Portal login and logout commands

Abstract

This technical note provides detailed information about how the WebSphere Portal login or logout flow can be extended or customized by using the login and logout commands. Focused on versions 5.1.0.x and 6.0 of Portal, the technote describes the configuration and the interfaces of the commands, followed by two code samples that are also available for download.

Introduction

The article "Leveraging WebSphere Portal V6 programming model: Part 4. Understanding and configuring WebSphere Portal login and logout" [1] mentions several ways to customize Portal's login or logout behavior by plugging custom code. As the officially recommended way to plug a JAAS login module (see [3] and [3]) does not work for all scenarios, Portal additionally provides the command based plug points for login and logout. Although these APIs are not public and partially deprecated, there is currently in some cases no alternative in using them if the scenario can not be realized using JAAS modules. While IBM will support you using the APIs in the current version of Portal, we cannot provide any migration support to future public APIs for custom code that is written against the APIs specified in this document.

The remainder of this document is structured as follows: First the role of the commands is recapitulated and a description of their configuration is given. In the second chapter, the command interfaces and related classes are listed and explained in detail. Finally, two short samples for the login and logout case show how to apply the customization in practice.

Configuration of the commands

A short recapitulation of the commands

As mentioned in the article "Leveraging WebSphere Portal V6 programming model: Part 4. Understanding and configuring WebSphere Portal login and logout" [1], the login command called `LoginUser` is executed for the explicit Portal login by user id and password using the login portlet / login screen or login URL, and the implicit login when being already authenticated by the WebSphere Application Server. It is not executed when a login is triggered by the XML configuration interface or the scripting interface. The logout command (`LogoutUser`) is executed for all of the four possible occasions for a logout, i.e. the explicit logout triggered by the user, the implicit logout when an authenticated user accesses a public page, the implicit logout after a session idle timeout, and the implicit logout if the current user does not belong to the realm associated to the virtual portal addressed in the request.
Both commands are designed to be extended and replaced by custom implementations and therefore provide dedicated methods to be overwritten by subclasses.

Plugging a custom implementation of a command

The actual implementations of the login and logout commands are determined each by a combination of two configuration settings. First, the class name of the respective command can be specified as a property of the Portal Configuration Service (see the Portal Information Center [4] on how to view or modify service properties):

```
command.login  = LoginUserAuth
```

```
command.logout = LogoutUserAuth
```

Second, you can specify the search path for the package name of all commands in a comma-separated list as a property of the Portal Loader Service:

```
command.path = com.mycompany.commands,com.ibm.wps.engine.commands,
com.ibm.wps.dynamicui.commands
```

In the sample above the classes `LoginUserAuth` and `LogoutUserAuth` will first be looked up in the package `com.mycompany.commands`, then if not found, in the Portal packages `com.ibm.wps.engine.commands` and if still not found in `com.ibm.wps.dynamicui.commands`.
You should not remove the default packages `com.ibm.wps.engine.commands` and `com.ibm.wps.dynamicui.commands` as these are not only needed for the login and logout commands.

Interfaces of the commands

Extending a command

In order to implement your own version of a login or logout command, extend the `com.ibm.wps.engine.LoginUserAuth` or `com.ibm.wps.engine.LogoutUserAuth` class and overwrite the respective methods where you need to add custom logic. Always call the parent's method with a `super` call either before or after your code. As the commands are instantiated as singletons, make sure that each method is thread-safe. If a checked exception is to be thrown in one of the methods that define to throw a `WpsException` in their signature, an instance of the `com.ibm.portal.CustomWpsException` or a custom child class of this exception should be used.

Classes used by the command

Nearly all methods in the commands get passed in an instance of a Portal internal helper class that wraps the request and response. Manipulating this so-called `RunData` object can have an unexpected impact on Portal's behavior. Therefore, in the context of a custom command, only the methods listed in the following should be used.

```
package com.ibm.wps.engine;

public class RunData
{
   // returns the http request object
   public HttpServletRequest getRequest();

   // returns the http response object
   public HttpServletResponse getResponse();

   // returns the session object, same semantics as HttpServletRequest.getSession
   public HttpSession getSession(boolean create);

   // sets an http redirect target URL. The redirect will be triggered after
   // execution of the command
   // for redirects this method has to be used, you must not use the
   // httpResponse.sendRedirect method.
   public void setRedirectURL(String aRedirectURL);

   // sets an http status code for the response. You should set this to 301 or
   // 302 to make sure a redirect is sent after executing the command (although this
   // may not be necessary in all cases)
   public void setStatusCode(int aStatusCode);
}
```

Exceptions thrown by custom command implementations should subclass the following class:

```
package com.ibm.portal;

public class CustomWpsException extends WpsException
{
    /**
    * Construct a new WpsException without parameters - the cause field is not initialized
    * The cause may subsequently be initialized via calling the {@link #initCause} method.
    */
    public CustomWpsException();

    /**
    * Construct a new WpsException with the cause field
    * set to the specified Throwable. Subsequent calls to the {@link #initCause}
    * method on this instance will result in an exception. The value of the cause
    * field may be retrieved at any time via the {@link #getCause()} method.
    * @param cause the Throwable that was caught and is considered the root cause
    * of this exception; may be null.
    */
    public CustomWpsException(Throwable t);

    /**
    * Construct a new WpsException with the specified detail message
    * - the cause field is not initialized
    * The cause may subsequently be initialized via calling the {@link #initCause} method.
    * @param message detailed message in form of a String.
    */
    public CustomWpsException(String message);

    /**
    * Construct a new WpsException with the specified detail message and a cause field
    * set to the specified Throwable. Subsequent calls to the {@link #initCause}
    * method on this instance will result in an exception. The value of the cause
    * field may be retrieved at any time via the {@link #getCause()} method.
    * @param message detailed message in form of a String.
    * @param cause the Throwable that was caught and is considered the root cause
    * of this exception; may be null.
    */
    public CustomWpsException(Throwable t, String message);
}
```

Specification of the login command

The following listing gives details about the `com.ibm.wps.engine.LoginUserAuth` class and the methods that can be overwritten by custom implementations.

```
public class LoginUserAuth extends com.ibm.wps.engine.commands.LoginUser
{
    /**
     ** @param   com.ibm.wps.engine.RunData
     **          the RunData object
     **
     ** @param   String
     **          the userid, "null" for the case being already authenticated by WAS
     **
     ** @param   String
     **          the user password, "null" for the case being already authenticated by WAS
     **
     ** @throws  WpsException, a generic exception.
     **          If an exception is thrown, the login command will
     **          fail to execute successfully.
     **/
    protected void doPreLogin (RunData aRunData, String aUserID, String aPassword)
                               throws WpsException
    {}

    /**
     ** Constants for error codes returned by the doAuthenticate() method
     **
     **/
    protected final static int NO_ERROR                    = 0;
```

```
    protected final static int OTHER_ERROR                 = 1;
    protected final static int USER_RETRIEVE_ERROR         = 2;
    protected final static int USERID_INVALID_ERROR        = 3;
    protected final static int PASSWORD_INVALID_ERROR      = 4;
    protected final static int AUTHENTICATION_FAILED_ERROR = 5;
    protected final static int JAAS_LOGIN_FAILED_ERROR     = 6;
    protected final static int RESERVED                    = 7;
    protected final static int USER_SESSION_TIMEOUT_ERROR  = 8;

    protected final static int USER_DEFINED_ERROR          = 1000;

    /**
     ** This method does the user authentication. The return value is an error code.
     **
     ** @param   com.ibm.wps.engine.RunData
     **          the RunData object
     **
     ** @param   String
     **          the userid, "null" for the case being already authenticated by WAS
     **
     ** @param   String
     **          the user password, "null" for the case being already authenticated by WAS
     **
     ** @return  ErrorBean
     **          The error state. Error code is NO_ERROR if successful. NO_ERROR to
     **          USER_DEFINED_ERROR are reserved codes, custom LoginUser classes must
     **          define error codes higher than USER_DEFINED_ERROR.
     **/
    protected ErrorBean doAuthenticate(RunData aRunData, String aUserID, String aPassword)
    {
        <perform the default Portal authentication>
    }

    /**
     ** This method is called if the doAuthenticate() return code is other than NO_ERROR
     **
     ** @param   com.ibm.wps.engine.RunData
     **          the RunData object
     **
     ** @param   ErrorBean
     **          error state as returned by the doAuthenticate() method
     **/
    protected void onAuthenticationError (RunData aRunData, ErrorBean aErrorBean)
    {
        <handle authentication error>
    }

    /**
     ** This method is called after a successful user login/authentication,
     ** (i.e. only if the doAuthenticate() return code is NO_ERROR).
     **
     ** @param   com.ibm.wps.engine.RunData
     **          the RunData object
     **
     ** @param   String
     **          the userid, "null" for the case being already authenticated by WAS
     **
     ** @param   String
     **          the user password, "null" for the case being already authenticated by WAS
     **
     ** @throws  WpsException, a generic exception.
     **/
    protected void doPostLogin (RunData aRunData, String aUserID, String aPassword)
                                throws WpsException
    {}
}
```

The main use case to customize the login command is to add additional checking of the credentials or entitlements of the user or a custom redirect after the login. To prevent the login from succeeding either an exception should be thrown or an according `ErrorBean` be returned. Exceptions thrown in the `doPostLogin` method will not prevent the login any more. Redirects should be set by calling the `setRedirectURL` method on the `RunData` object.

See the paper [1] for the properties that allow modifying the login behavior without the need to write your own login command.

The `ErrorBean` object returned by the `doAuthenticate` method is a simple bean for the error code and exception:

```
package com.ibm.wps.auth;

public class ErrorBean
{
    public ErrorBean(int ErrorCode, Exception exception);

    public int getErrorCode();
    public Exception getException();
}
```

Specification of the logout command

The logout case is less complex than the login, which results in only two methods dedicated to be overwritten. Additionally, the command also contains a method you can implement to listen for the session timeout event, as the following listing of the `com.ibm.wps.engine.LogoutUserAuth` class shows.

```
public class LogoutUserAuth extends com.ibm.wps.engine.commands.LogoutUser
{
    /**
     ** This method is called before the Portal logout is performed.
     **
     ** @param   com.ibm.wps.engine.RunData
     **          the RunData object
     **
     ** @throws  WpsException, a generic exception.
     **          If an exception is thrown, only the Portal session will be invalidated.
     **/
    protected void doPreLogout (RunData aRunData) throws WpsException
    {
        <perform Websphere Application Server logout>
    }

    /**
     ** This method is called after the Portal logout is performed.
     **
     ** @param   com.ibm.wps.engine.RunData
     **          the RunData object
     **
     ** @throws  WpsException, a generic exception.
     **/
    protected void doPostLogout (RunData aRunData) throws WpsException
    {}

    /**
     ** This method is called when a user's session times out and the user has not
     ** previously logged out.
     **
     ** @param   Session
     **          the user's session
     **/
    public void onUserSessionTimeout (HttpSession aSession)
    {}
}
```

The main use case to customize the logout command is to add additional logouts of custom applications or a custom redirect after the login. To prevent the logout from succeeding, an exception should be thrown in the `doPreLogout` method. Redirects should be set by calling the `setRedirectURL` method on the `RunData` object.

See the paper [1] for the properties that allow modifying the logout behavior without the need to write your own logout command.

Samples

The following samples for a custom login and a logout command show how a typical use case, namely the dynamic redirect after login or logout depending on the current virtual portal, can be realized. First, some helper methods are described. The actual implementation of the commands then is basically a sequence of calls to those helper methods. The complete java source files of the samples are also attached to this technote.

Service Lookup

As we access some services over JNDI, the necessary lookups for these should be done only once:

```
// PortalStateManagerServiceHome object to retrieve the service from
private static PortalStateManagerServiceHome psmServiceHome;
private static final String PSM_JNDI_NAME =
"portal:service/state/PortalStateManager";

private synchronized static PortalStateManagerServiceHome getPSMServiceHome() {
       if (psmServiceHome == null) {
              try {
                     final Context ctx = new InitialContext();
                     psmServiceHome =
                            (PortalStateManagerServiceHome) ctx.lookup(PSM_JNDI_NAME);
              } catch (Exception e) {
                     // error handling
              }
       }
       return psmServiceHome;
}
```

Determining the Virtual Portal for the login case

When logging in, there is no direct way for a login command to find out the virtual portal the user is logging in to. However, the `URLAccessorFactory` may be employed to query the `PortalStateManager` for a URL to an empty selection. The returned URL will contain the virtual portal.

```
private String getLoginRedirectTarget(RunData aRunData) {
       URLAccessorFactory urlAccFac = null;
       try {
              urlAccFac =
       (URLAccessorFactory) getPSMServiceHome().
getPortalStateManagerService(aRunData.getRequest(), aRunData.getResponse())
              .getAccessorFactory(URLAccessorFactory.class);
       } catch (UnknownAccessorTypeException e) {
              ...
       } catch (CannotInstantiateAccessorException e) {
              ...
       } catch (StateManagerException e) {
              ...
       }
       EngineURL engineURL = null;
       try {
              // create a new URL with an empty selection
              engineURL = urlAccFac.newURL(aRunData.getRequest(),
aRunData.getResponse(), null);
```

```
        } catch (InvalidConstantException e) {
                ...
        } catch (CannotCloneDocumentModelException e) {
                ...
        } catch (CannotCreateDocumentException e) {
                ...
        } catch (StateNotInRequestException e) {
                ...
        }
        StringWriter strWrt = new StringWriter();
        try {
                engineURL.writeDispose(strWrt);
        } catch (OutputMediatorException e) {
                ...
        } catch (PostProcessorException e) {
                ...
        } catch (IOException e) {
                ...
        }

        return strWrt.toString();
}
```

As we know that the URL contains the virtual portal information after the `/context path/uri home/` element, we can then parse the URL to determine the virtual portal name.

```
private String getVirtualPortalName(RunData aRunData) {
        // parse request url: /context path/uri home/virtual portal
        String contextPath = "/wps";
        String uriHome = "portal";
        Pattern vpRE = Pattern.compile("^" + contextPath + "/" + uriHome +
"/([^/]+).*?");

        Matcher m = vpRE.matcher(getLoginRedirectTarget(aRunData));

        String virtualPortalName = null;
        if (m.matches() ){
                virtualPortalName = m.group(1);
        }

        return virtualPortalName;
}
```

Determining the Virtual Portal for the logout case

When logging out, the virtual portal name is contained in the request URI of the `RunData` object and we can extract it directly from there:

```
private String getVirtualPortalName(RunData aRunData) {
        // parse request url: /context path/uri home/virtual portal
        String contextPath = "/wps";
        String uriHome = "portal";

        Pattern vpRE = Pattern.compile("^" + contextPath + "/" + uriHome +
"/([^/]+).*?");
        Matcher m = vpRE.matcher(aRunData.getRequest().getRequestURI());

        String virtualPortalName = null;
        if (m.matches() ){
                virtualPortalName = m.group(1);
        }
}
```

Setting the redirect target

As indicated above, we should use the `RunData.setRedirect` method to trigger a redirect. You can redirect the user to a specific page in Portal, but also use an absolute URL to another site. The snippet shown below generates a valid portal URL from the unique name of a page.

```
private EngineURL getRedirectURL(RunData aRunData, final String pageID) {
	try {
		// get the request-specific service from our home interface
		final StateManagerService service = getPSMServiceHome().
getPortalStateManagerService(aRunData.getRequest(), aRunData.getResponse());

		// get the needed factories
		final URLFactory urlFactory = service.getURLFactory();
		final SelectionAccessorFactory selFct = (SelectionAccessorFactory)
		service.getAccessorFactory(SelectionAccessorFactory.class);
		try {
			// get a new URL from the URL factory
			final EngineURL url = urlFactory.newURL(null);
			url.setProtected(Boolean.FALSE);
			// get a selection controller which operates on the
			// URL-specific state
			final SelectionAccessorController selCtrl =
				selFct.getSelectionAccessorController(url.getState());
			try {
				// change the selection
				selCtrl.setSelection(pageID);

				return url;
			} finally {
				selCtrl.dispose();
			}
		} finally {
			urlFactory.dispose();
			// indicate that we do not need it any longer
			service.dispose();
		}
	} catch (Exception e) {
		// error handling
		return null;
	}
}
```

Customized login command

Our custom login command extends the `com.ibm.wps.engine.commands.LoginUserAuth` class and overwrites the `doPostLogin` method to add the custom redirect handling.

```
public class LoginUserAuth extends com.ibm.wps.engine.commands.LoginUserAuth {
	protected void doPostLogin(RunData aRunData, String aUserID, String aPassword)
throws WpsException {
		String virtualPortalName = getVirtualPortalName(aRunData);
		if (null != virtualPortalName) {
			if (virtualPortalName.equals("foo") ) {
				// unique name of the page we want to redirect to
				String pageID = "mypage";
				EngineURL redirectUrl = getRedirectURL(aRunData, pageID);

				if (null != redirectUrl) {
					aRunData.setRedirectURL(redirectUrl.toString() );
				}
			} else if (virtualPortalName.equals("bar") ) {
				// another virtual portal, another page
				String pageID = "anotherpage";
				EngineURL redirectUrl = getRedirectURL(aRunData, pageID);
```

```
                        if (null != redirectUrl) {
                                aRunData.setRedirectURL(redirectUrl.toString() );
                        }
                }
        }
    }
}
```

Customized logout command

The custom logout command works almost the same way, extending the `com.ibm.wps.engine.commands.LogoutUserAuth` class and overwriting the `doPostLogout` method.

```
public class LogoutUserAuth extends com.ibm.wps.engine.commands.LogoutUserAuth {
       protected void doPostLogout(RunData aRunData) throws WpsException {
              String virtualPortalName = getVirtualPortalName(aRunData);
              if (null != virtualPortalName) {
                     if (virtualPortalName.equals("foo") ) {
                            // unique name of the page we want to redirect to
                            String pageID = "mypage";
                            EngineURL redirectUrl = getRedirectURL(aRunData, pageID);

                            if (null != redirectUrl) {
                                   aRunData.setRedirectURL(redirectUrl.toString() );
                            }
                     } else if (virtualPortalName.equals("bar") ) {
                            // another virtual portal, another page
                            String pageID = "anotherpage";
                            EngineURL redirectUrl = getRedirectURL(aRunData, pageID);

                            if (null != redirectUrl) {
                                   aRunData.setRedirectURL(redirectUrl.toString() );
                            }
                     }
              }
       }
}
```

Using the sample code

Attached to the technote is a zip archive containing the java source files of the two sample commands. To compile the commands, you can import them to a Portal project using one of the assembly tools that integrate with WebSphere Portal (e.g. IBM Rational Application Developer) and thus provide the build path of the respective Portal version out of the box. However, it may be necessary to additionally add the `wp.auth.cmd.jar` from the `<Portal server>/shared/app` directory to the build path.
To test the sample scenario, you first need to create a virtual portal *foo* that contains a page with the unique name *mypage* and a virtual portal *bar* with a page having the unique name *anotherpage*. Then add `vpr.cmd` as the first package name to the property `command.path` in the Loader service as described in the first section. Finally, you have to deploy the compiled commands to the Portal class path, which is done best by exporting the class files to a jar file and copying this file to the `<Portal server>/shared/app` directory. After restarting Portal, you should then be redirected to *mypage* when logging in to virtual portal *foo*, and to *anotherpage* when logging in to virtual portal *bar*.

Resources

[1] Leveraging WebSphere Portal V6 programming model: Part 4. Understanding and configuring WebSphere Portal login and logout

http://www.ibm.com/developerworks/websphere/library/techarticles/0706_buchwald/0706_buchwald.html
[2] JAAS Documentation and Specification
http://java.sun.com/products/jaas/
[3] Exploiting the WebSphere Portal V5.1.0.1 programming model: Part 3: Integrating WebSphere Portal into your security environment and user management system
http://www-128.ibm.com/developerworks/websphere/library/techarticles/0606_buehler/0606_buehler.html
[4] WebSphere Portal product documentation, including the Information Center
http://www.software.ibm.com/wsdd/zones/portal/proddoc.html