BEA WebLogic Portal™®

White Paper: Content Personalization

Version 8.1
Document Revised: May 2004
By: Greg Smith

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

Content Personalization

Content Personalization

What is Content?

Content is information. Most people think of things like documents, images, audio, and video as being content, which is true. Content also generally entails an amount of information that describes the content in more concrete terms. In fact, content can be just information, or metadata, with no associated media. Metadata is typically parametric and/or structured data about the content itself.

For example, an advertisement image for a car might have metadata parameters that describe its make, model, size, color, and price. Content can usually be retrieved by searching through the metadata and, sometimes, the media. Content is often managed by a Content Management System, which provides a variety of services to help in the creation, editing, and publishing of content.

BEA WebLogic Portal provides content management services through a Virtual Content Repository. The Virtual Content Repository supports "plugging" in access to multiple content management systems and providing a single point of access to all of them. Additionally, BEA WebLogic Portal provides a full-featured content repository that can be used either instead of or in conjunction with 3rd party content management systems.

What is Personalized Content?

Personalized content is content which matches a particular context, generally around a user. It takes into account information contained in the context to correctly generate search queries which will retrieve the content most appropriate to the context. For example, if you have red, green, and blue images, and you have determined that the user prefers green, you would probably want to

display green images to the user (assuming you want to have happy users).
In BEA WebLogic Portal, the context to match against includes at least the user's profile, the user's current request, the user's current session, and current date and time. Additionally, Portal supports writing business rules which can classify users into various segments; these segments support the same contextual information. For example, you could define a business rule which defines who your Premier Users are. In some cases, you can also use these segments when personalizing content.

Users

Users use your application. In Portal, you will need to have a user, or a simulation of a user, to retrieve personalized content. Users have 2 aspects in Portal: their identity and their profile.

Authentication Identification

This is a Subject containing multiple Principals, including the user's. This is used by the security subsystems to identify the user and their capabilities.

Listing 1 Getting a principal

```
<%@ page import="com.bea.p13n.security.Authentication" %>
<%-- Current Subject, which can comprise multiple Principals.
     This method is usable anywhere, not just from webapps.
--%>
Authentication Utility class:<br>
<pre><%= Authentication.getCurrentSubject() %></pre><br>

Servlet Request: [<code><%= request.getUserPrincipal() %></code>]<br>
```

There are several ways to authenticate users with the system, including:

- `<security-constraint>` in `WEB-INF/web.xml`: This can be used for both HTTP BASIC Auth and form-based authentication in a standard, j2ee-compliant way.
- `com.bea.p13n.security.Authentication.login()`, generally from a servlet, servlet filter, portlet BackingFile, or a PageFlow.

- `com.bea.p13n.security.Authentication.authenticate()`.
- `<um:login>` JSP tag.
- the `UserLoginControl's login()` method, from a PageFlow or JSP.

Profile

The profile is metadata about the user. It can be separate from a user identity. In fact, in Portal, there are three kinds of user profiles:

- Registered: This is the profile for a fully-registered user who can be authenticated with the system.
- Anonymous: This is the profile for a user who does not have an identity with the system.
- Tracked Anonymous: This is the profile for a user who Portal recognizes but is not registered; it, therefore, does not have an identity which can authenticate with the system.

The profile is kept in the session by default. The user profile is usually initialized by the `PortalServletFilter` in the webapp. This filter will initialize either an Anonymous or Tracked Anonymous profile in the session on first access, depending upon whether user tracking is enabled and the request has a valid tracking cookie. If the user authenticates (logs in), the filter will switch the profile with the user's registered profile. If the user registers with the system, the filter will create a registered profile, initialized with any values in their (tracked) anonymous profile, and put that in the session.

User profiles can store properties specified via User Profile Property Sets defined in the data project of a portal application in WebLogic Workshop and those that are not. However, the Administration Portal and the other WebLogic Workshop editors can only operate against properties defined in a User Profile Property Set.

Example User Profiles

The user's profile object can be retrieved from the session in several ways:

Listing 2 Retrieving a user profile from the session

```
<%@ page import="com.bea.p13n.usermgmt.SessionHelper"%>
<%@ taglib uri="http://www.bea.com/servers/p13n/tags/usermanagement"
prefix="um"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
```

```
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
Profile is: [<code><%= SessionHelper.getProfile(request) %></code>]<br>

<%-- This tag works for authenticated users. --%>
<um:getProfile profileKey="<%=request.getUserPrincipal().getName()%>"
profileId="profile"/>
Profile is: [<code><%= profile %></code>]<br>

<%-- You would generally want to do this in your PageFlow, not your JSP. --%>
<netui-data:declareControl controlId="profileControl"
  type="com.bea.p13n.controls.profile.UserProfileControl"/>
<netui-data:callControl resultId="getProfileFromRequestResult"
  controlId="profileControl" method="getProfileFromRequest">
<netui-data:methodParameter value="{request}"></netui-data:methodParameter>
</netui-data:callControl>
Profile is: [<code><netui:label
value="{pageContext.getProfileFromRequestResult}"></netui:label></code>]<br>
```

If the user is registered, then the user's profile can be retrieved without a reference to the session:

Listing 3 Retrieving a user profile without a session reference

```
import com.bea.p13n.usermgmt.profile.ProfileFactory;
import com.bea.p13n.usermgmt.profile.ProfileNotFoundException;
import com.bea.p13n.usermgmt.profile.ProfileWrapper;
import java.rmi.RemoteException;
public class MyHelper
{
    public static String helperMethod(String username)
    {
        try
        {
            ProfileWrapper profile = ProfileFactory.getProfile(username, null);
            // do something helpful here.
            return profile.toString();
        }
        catch (RemoteException ex)
        {
        }
        catch (ProfileNotFoundException ex)
        {
        }
        return null;
```

```
    }
}
```

Additionally, the `UserProfileControl` has methods for retrieving a user's profile based upon the username.

Of course, for anonymous and tracked anonymous profiles, you have to retrieve the profile from the session. Anonymous profiles have no identity whatsoever. Tracked anonymous profile have an identity which is not valid for authentication. A safe way to retrieve the identity for a user, based upon whatever type of profile they have, is:

Listing 4 Sample Code for Retreiving a User's Identity

```
<%@ page import="com.bea.p13n.usermgmt.SessionHelper"%>
Profile Id is: [<code><%= SessionHelper.getUserId(request) %></code>]<br>
```

Note: The returned value will be null for anonymous profiles, the tracking id (which can not be used for authentication) for tracked anonymous profiles, and the user principal name for authenticated and registered profiles.

Manipulating User Profiles

To manipulate the user's profile, you can either:

- use the `ProfileWrapper` object directly, or
- use the `<um:getProperty>` and `<um:setProperty>` tags, or
- use a `UserProfileControl` to manipulate a `ProfileWrapper`.

Here's an example page flow (and associated JSP) that use controls to offer a form for the user to set their Favorite Color. This examples requires that a `Generalnfo.usr` User Profile Property Set file exist in the userprofiles/ folder of the data project, with a single-valued, restricted, text FavoriteColor property.

Note: See the sample application for more details.

Listing 5 Letting a user change a user profile property

```
package users.setcolor;

import com.bea.p13n.controls.exceptions.P13nControlException;
import com.bea.p13n.property.PropertyDefinition;
import com.bea.p13n.property.PropertySet;
import com.bea.p13n.usermgmt.profile.ProfileWrapper;
import com.bea.wlw.netui.pageflow.FormData;
import com.bea.wlw.netui.pageflow.Forward;
import com.bea.wlw.netui.pageflow.PageFlowController;
import java.util.Collection;
import java.util.Iterator;

/**
 * @jpf:controller
 */
public class SetColorController extends PageFlowController
{
    /**
     * @common:control
     */
    private com.bea.p13n.controls.ejb.property.PropertySetManager propSetMgr;

    /**
     * @common:control
     */
    private com.bea.p13n.controls.profile.UserProfileControl profileControl;

    /** Cached possible colors from the User Profile Property Set definition.
     */
    private String[] possibleColors = null;

    /** Get the possible colors, based upon the User Profile Property Set.
     */
    public String[] getPossibleColors()
    {
        if (possibleColors != null)
            return possibleColors;
        try
        {
            PropertySet ps = propSetMgr.getPropertySet("USER", "GeneralInfo");
            PropertyDefinition pd = ps.getPropertyDefinition("FavoriteColor");
            Collection l = pd.getRestrictedValues();
            String[] s = new String[l.size()];
            Iterator it = l.iterator();
            for (int i = 0; it.hasNext(); i++)
                s[i] = it.next().toString();
```

```
            possibleColors = s;
        }
        catch (P13nControlException ex)
        {
            ex.printStackTrace();
            possibleColors = new String[0];
        }
        return possibleColors;
    }

    /** Get the user's favorite color from their profile.
     */
    public String getUsersColor()
    {
        try
        {
            ProfileWrapper profile =
profileControl.getProfileFromRequest(getRequest());
            return profileControl.getProperty(profile, "GeneralInfo",
"FavoriteColor").toString();
        }
        catch (P13nControlException ex)
        {
            ex.printStackTrace();
        }
        return null;
    }

    // Uncomment this declaration to access Global.app.
    //
    //     protected global.Global globalApp;
    //

    // For an example of page flow exception handling see the example "catch"
and "exception-handler"
    // annotations in {project}/WEB-INF/src/global/Global.app

    /**
     * This method represents the point of entry into the pageflow
     * @jpf:action
     * @jpf:forward name="success" path="index.jsp"
     */
    protected Forward begin()
    {
        return new Forward("success");
    }

    /**
     * @jpf:action
```

```
     * @jpf:forward name="success" path="begin.do"
     */
    protected Forward setColor(ColorFormBean form)
    {
        // set the color in the user's profile
        try
        {
            ProfileWrapper profile =
profileControl.getProfileFromRequest(getRequest());
          profileControl.setProperty(profile, "GeneralInfo", "FavoriteColor",
form.getColor());
        }
        catch (P13nControlException ex)
        {
            ex.printStackTrace();
        }
        return new Forward("success");
    }

    /**
     * FormData get and set methods may be overwritten by the Form Bean editor.
     */
    public static class ColorFormBean extends FormData
    {
        private String color;

        public void setColor(String color)
        {
            this.color = color;
        }

        public String getColor()
        {
            return this.color;
        }
    }
}
```

Listing 6 Example of the Associated index.jsp

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib uri="netui-tags-databinding.tld" prefix="netui-data"%>
<%@ taglib uri="netui-tags-html.tld" prefix="netui"%>
<%@ taglib uri="netui-tags-template.tld" prefix="netui-template"%>
<netui:html>
    <body>
```

```
        <netui:form action="setColor">
            <table>
                <tr valign="top">
                    <td>Favorite Color:</td>
                    <td>
                    <netui:select dataSource="{actionForm.color}"
                      defaultValue="{pageFlow.usersColor}"
                  optionsDataSource="{pageFlow.possibleColors}"></netui:select>
                    </td>
                </tr>
            </table>
            <br/> 
            <netui:button value="Set Color" type="submit"/>
        </netui:form>
    </body>
</netui:html>
```

Note: Much of this sample was generated via drag-and-drop and the wizards in WebLogic Workshop.

Content

In Portal, content exists in a hierarchy. At the top are the configured repositories for the application. Under each repository is the tree of content Nodes. Each Node can be either a hierarchy or content Node. Hierarchy Nodes are analogous to folders; content Nodes are analogous to files. Both can have metadata properties bound to a content type, which is internally called an ObjectClass. ObjectClasses are like Property Sets for content. They define the available metadata properties, their data types, and their default and possible values.

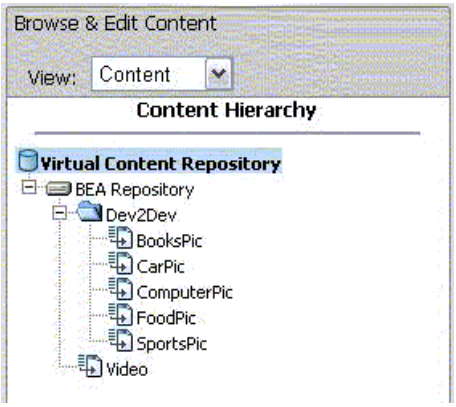
There are APIs in the com.bea.content and com.bea.content.manager packages are creating, editing, deleting, and retrieving ObjectClasses and Nodes. For this article, however, it will suffice to use the Administration Portal to manipulate the content tree.

To access the Administration Portal, use the PortalIPortal Administration... option in the menu in WebLogic Workshop; alternatively, the Administration Portal is a webapp deployed to the `<appname>`Admin URL on your server (e.g. `contentAppAdmin/`). Once it's open in your browser, use an admin-capable username to login, such as the one you used to create the domain; weblogic/weblogic and portaladmin/portaladmin are the defaults. Select the Content link in the header.

Retrieving Content Nodes

Content is retrieved generally in one of 2 ways: by Node path or by a search query. All Nodes are addressable by a unique path. This path is visible in the Administration Portal as you create folder Nodes (via the Add Node button) and content Nodes. Then, you can use the `<cm:getNode>` JSP tag retrieve the Node. For example, if your content hierarchy appears in the Administration Portal as shown in Figure 1, you could use the code in Listing 7 to retrieve the `CarPic` node.

Figure 1 Content hierarchy



Listing 7 Sample Code for Retrieving the CarPic Node

```
<%@ taglib uri="content.tld" prefix="cm"%>
<cm:getNode path="/BEA Repository/Dev2Dev/CarPic" id="carpic" />
```

Searching for Content Nodes with Content Query Expressions

To search for Nodes, you can construct a content query expression. Content query expressions can be constructed either in object form or via the query syntax. Full details can be found in the JavaDoc for `com.bea.content.expression.ExpressionHelper.parse()`, including on the format of the query syntax.

There is a JSP tag available to search for Nodes based upon a search expression. For example, to find all Nodes whose name (which is the last part of the Node path) includes 'Pic', you could use the following:

Listing 8 Sample Code for Searching Content with Expressions

```
<%@ taglib uri="content.tld" prefix="cm"%>
<%@ taglib uri="http://www.bea.com/servers/p13n/tags/utility"
prefix="utility"%>
<cm:search id="nodes" query=" cm_nodeName like '*Pic' " sortBy="cm_nodeName
desc"/>
Found <%=nodes.length%> Node(s):
<ol>
<utility:forEachInArray array="<%=nodes%>" id="node"
type="com.bea.content.Node">
<li><cm:getProperty id="node" name="cm_nodeName"
conversionType="html"/></li>
</utility:forEachInArray>
</ol>
```

This will print out the names of the `Node`s that end in '`Pic`' in reverse order, so, given the previous `Node` hierarchy, it would be:

1. `SportsPic`
2. `FoodPic`
3. `ComputerPic`
4. `CarPic`
5. `BooksPic`

For most portal tags, the Property Editor in WebLogic Workshop gives descriptions of the tag and attributes. Additionally, F1 help exists for most portal tags; just put the cursor on the tag (in Source or Design view) and press F1 (or Help|Context Help in the menu).

Searching for Content Nodes Via the APIs

Alternatively, you could use the search APIs directly, particularly if you're not in a JSP or servlet context:

Listing 9 Sample Code for Searching Content Via the APIs

```
<%@ page import="com.bea.content.Node"%>
<%@ page import="com.bea.content.expression.ExpressionHelper"%>
<%@ page import="com.bea.content.expression.Search"%>
<%@ page import="com.bea.content.manager.RepositoryManager"%>
<%@ page import="com.bea.content.manager.RepositoryManagerFactory"%>
<%@ page import="com.bea.content.manager.SearchOps"%>
<%
// Construct the search object from a content query and a sorting clause
Search search = new Search();
search.setExpression(ExpressionHelper.parse(" cm_nodeName like '*Pic' "));
search.setSortCriteria("cm_nodeName desc");
// connect to the repositories
RepositoryManager mgr = RepositoryManagerFactory.connect(session);
// search and fetches Nodes
Node[] nodes =
mgr.getNodeOps().getNodes(mgr.getSearchOps().search(search));
%>
Found <%=nodes.length%> Node(s):
<ol>
<% for (int i = 0; i < nodes.length; i++) { %>
<li><%=nodes[i].getName()%></li>
<% } %>
</ol>
```

Personalization

So, now we know about users and profiles, and about content and searching. We just need to put these together to get personalized content. There are many ways to do this, depending upon your needs and what technologies are being used.

Notes on the Following Samples

The following samples, and the sample application, rely upon certain data being initialized in the content repository. I've created a Dev2Dev content type which appears in the Administration Portal, shown in Listing 2.

Figure 2 Properties in the Dev2Dev content type

Property Name (Primary Property is **Bold**)	Property Value(s) (* = Default Value)	Choice Type	Data Type
adWeight		Single Unrestricted	Integer
color	* Blue Green Red	Single Restricted	String
media		Single Unrestricted	Binary
title		Single Unrestricted	String
topic	Books Cars Computers Cooking Sports	Single Restricted	String

Add Property

Note: The media property is designated as the primary property.

From that content type, Nodes were created for each topic, with each color being represented in at least one Node. For the media attribute for each, I uploaded an image file appropriate to the node's topic and color. This made sure I had enough content to try out various personalizations.

In a real application, especially against an enterprise content management system, it will be typical to have significantly more data. Often, the developer does not generate the data but must instead use the tools to discover what's available.

Also, the sample application and this sample code was written with a BEA WebLogic Platform 8.1.2 installation. You might see different behavior with other versions.

Personalized Searching

The content query syntax supports the ability to refer to user, request, and session properties. This is done by using special keywords in the query syntax. For example, to refer to user's

FavoriteColor property from the GeneralInfo User Profile PropertySet, your query could look like:

```
color == userProperty('GeneralInfo', 'FavoriteColor')
```

Similarly, `requestProperty` and `sessionProperty` will refer to request and session properties, respectively.

Of course, you could manually build the query in a `StringBuffer` by fetching the user, request, or session property and doing appends. In some cases, this might be required, but the above syntax is often easier to use and understand.

The `SearchOps` API from above does not recognize the `userProperty` syntax; it expects fully complete (realized) expressions. An expression can be realized via the `ExpressionHelper.realize()` method, which takes a `PropertyProvider` implementation.

The `DefaultPropertyProvider` implementation supports a context with a `com.bea.p13n.usermgmt.profile.ProfileWrapper`, a `com.bea.p13n.http.Request`, a `com.bea.p13n.http.Session`, and a `com.bea.p13n.events.Event. ProfileWrapper` is the base interface for all implementations of a user profile.

Request is a serializable copy of the `HttpServletRequest` object; similarly, Session is a serializable copy of the `HttpSession` object. Event is used in conjunction with campaign-based personalized content queries, which will be covered later in this article.

Listing 10 Manually building a query

```
// this could be retrieved instead of hard-coded (e.g. from a ResourceBundle)
String query = "color == userProperty('GeneralInfo', 'FavoriteColor')";
Expression expr = ExpressionHelper.parse(query);
// this assumes you have access to the request and session.
// if you don't, use the no-args constructors for empty copies.
PropertyProvider provider = new DefaultPropertyProvider(
    SessionHelper.getProfile(request),
    new Request(request),
    new Session(session),
    null); // the event can be null
expr = ExpressionHelper.realize(expr, provider);
// Construct the search object from a content query and a sorting clause
Search search = new Search();
search.setExpression(expr);
search.setSortCriteria("cm_nodeName desc");
// connect to the repositories
RepositoryManager mgr = RepositoryManagerFactory.connect(session);
```

```
// search and fetches Nodes
Node[] nodes = mgr.getNodeOps().getNodes(mgr.getSearchOps().search(search));
```

If you're in a JSP, you can just use the same query with the `<cm:search>` tag:

Listing 11 Building a query with the <cm:search> JSP tag

```
<%@ taglib uri="content.tld" prefix="cm"%>
<%@ taglib uri="http://www.bea.com/servers/p13n/tags/utility"
prefix="utility"%>
<cm:search id="nodes" query=" color == userProperty('GeneralInfo',
'FavoriteColor') "
    sortBy="cm_nodeName desc"/>
Found <%=nodes.length%> Node(s):
<ol>
<utility:forEachInArray array="<%=nodes%>" id="node"
type="com.bea.content.Node">
<li><cm:getProperty id="node" name="title" conversionType="html"/></li>
</utility:forEachInArray>
</ol>
```

The tag will take care of constructing and realizing the query. When these are used in conjunction with the code to update the user's profile, the display for the user will dynamically change to match their settings.

Content Rendering

The next questions is what to do with each Node that comes back. In the previous examples, we've just printed out a metadata property of each Node. While that's useful and common, we probably want to show the media related to the Node.

The easiest way to do this is to use the `<ad:render>` JSP tag. This tag will call the `com.bea.p13n.ad.AdService.renderContent()` method. This will use the MIME type of the primary property's `BinaryValue.getContentType()` to pick an `AdContentProvider` to generate the appropriate HTML.

The `AdContentProviders` are configured in `META-INF/application-config.xml` in the application directory, in an `<AdContentProvider>` block under the `<AdService>` tag. The

Name attribute of the <AdContentProvider> should be the major/minor MIME type to match on, with the minor portion being optional.

The Provider attribute should be the fully qualified class name of the AdContentProvider implementation; you can create these in Java projects in your applications. So, if you created an implementation of AdContentProvider that handles generating appropriate HTML for video media, you would add the code shown in Listing 12 in the <AdService> tag in META-INF/application-config.xml:

Listing 12 Sample Code for <AdContentProvider>

```
<AdContentProvider Name="video"
   Notes="A content render that handles video."
   Provider="examples.contentp13n.VideoContentProvider"
   Properties=""/>
```

In your AdContentProvider, you need to implement the renderContent() method to return a block of XHTML that can display the Node correctly. You can reference the com.bea.content.manager.servlets.ShowPropertyServlet registered under the servletBase URI to return the bytes of the Node; this is useful in conjunction with <IMG>, <EMBED>, and <OBJECT> HTML tags. An example VideoContentProvider is in the sample application in the utils Java project; it generates an <EMBED> HTML statement.

Note: <EMBED> tags don't always work correctly in all browsers, depending upon what plugins you have configured).

BEA WebLogic Portal includes AdContentProviders to handle images, text, and shockwave. Additionally, the default AdContentProvider will just print out a link to any Node of an unknown MIME type.

Now that you have your content renderers configured, you can use the <ad:render> JSP tag to display your content:

Listing 13 Displaying content with the <ad:render> JSP tag

```
<%@ taglib uri="content.tld" prefix="cm"%>
<%@ taglib uri="http://www.bea.com/servers/p13n/tags/utility"
```

```
prefix="utility"%>
<%@ taglib uri="http://www.bea.com/servers/portal/tags/ad" prefix="ad"%>
<cm:search id="nodes" query=" color == userProperty('GeneralInfo',
'FavoriteColor') "
  sortBy="cm_nodeName desc"/>
Found <%=nodes.length%> Node(s):
<dl>
<utility:forEachInArray array="<%=nodes%>" id="node"
type="com.bea.content.Node">
<dt><cm:getProperty id="node" name="title" conversionType="html"/></dt>
<dd><ad:render id="node" /></dd>
</utility:forEachInArray>
</dl>
```

Another option is pick one Node and to cycle through the matching Nodes on subsequent requests. You can use the `<ad:adTarget>` JSP tag to do this. It will use the `AdConflictResolver` to pick which Node to show. That will get each Node's `adWeight` property (converted to a number) as the relative weight of each Node and then use a random number to pick which Node to use; the higher the weight, the more likely the Node is to be displayed. If the Node doesn't have an `adWeight` property, it assumes a value of 1. This is an easy way to get rotating banner-style content on your website:

```
<%@ taglib uri="http://www.bea.com/servers/portal/tags/ad" prefix="ad"%>
<ad:adTarget query=" color == userProperty('GeneralInfo', 'FavoriteColor') "/>
```

Content Selectors

The previous methods give you a large amount of control over what gets displayed and how. However, it's all contained in code, which is compiled and deployed. This makes it difficult to modify the queries in a live, production server. Additionally, you have to have do much of the work.

Content selectors are a rules-based mechanism to define both who gets to see content and what content they get to see. The rules are created as files in WebLogic Workshop. During development, the files reload when they change, just like JSPs, so you can quickly develop with content selectors. However, when the server's in production mode, content selectors are loaded into the database (from the file-based definitions in the application) where they can be modified in the Administration Portal, without redeploying the application or restarting the server.

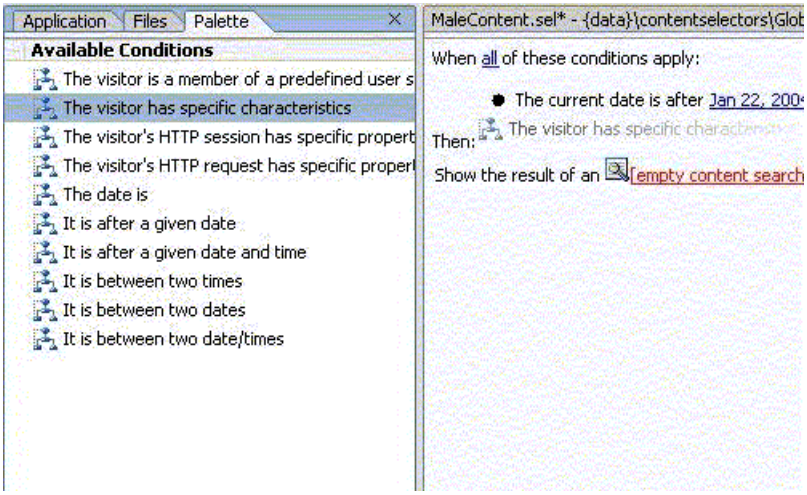
Note: For more details, see the Datasync Documentation on edocs.bea.com.

Additionally, the tools in the Administration Portal can be used by non-developers, making it possible for analysts to make changes to the content displayed on your website without requiring developer time.

Content selectors are created in the contentselectors/GlobalContentSelectors folder of the data project. In it, you define the conditions under which a user will match the content selector by dragging conditions from the Available Conditions Palette. You can delete conditions by right-mouse-clicking on one and selecting Delete.

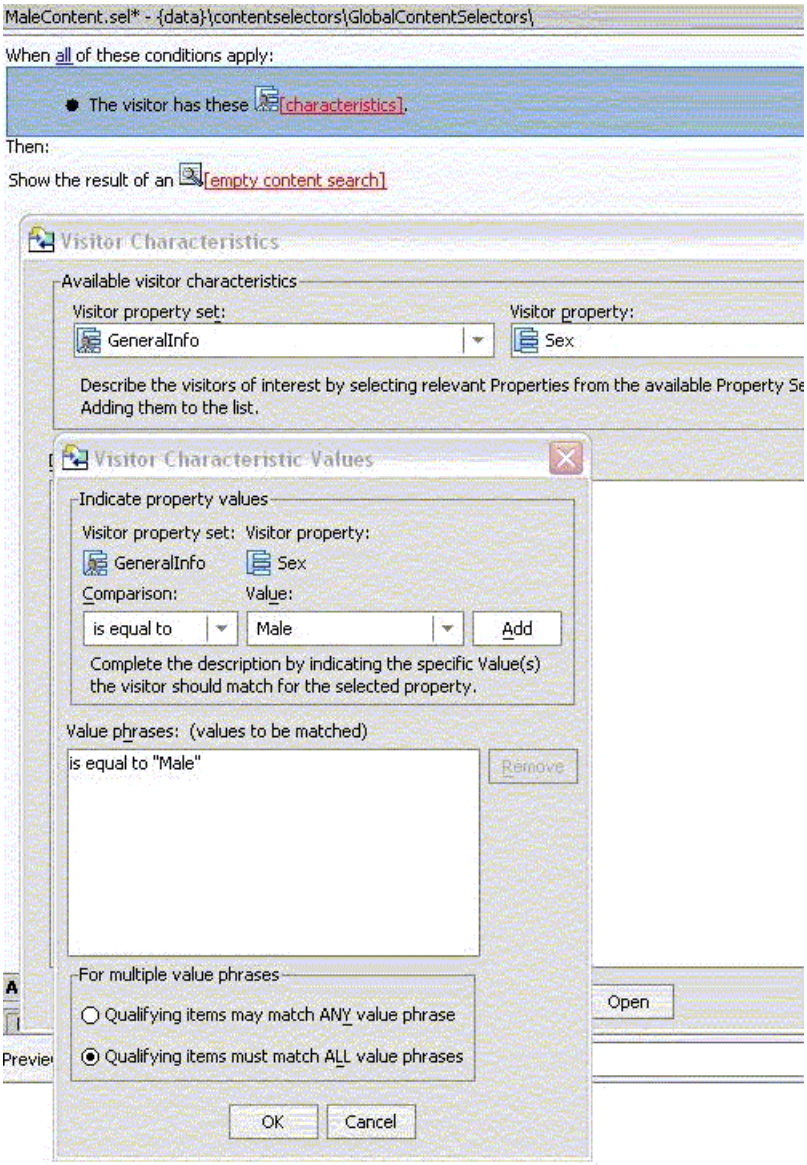
For example, let's make a content selector which causes Male visitors to see content that matches their favorite color. First, you drag the condition from the palette to the content selector, as shown in Figure 3.

Figure 3 Adding a condition to a content selector definition



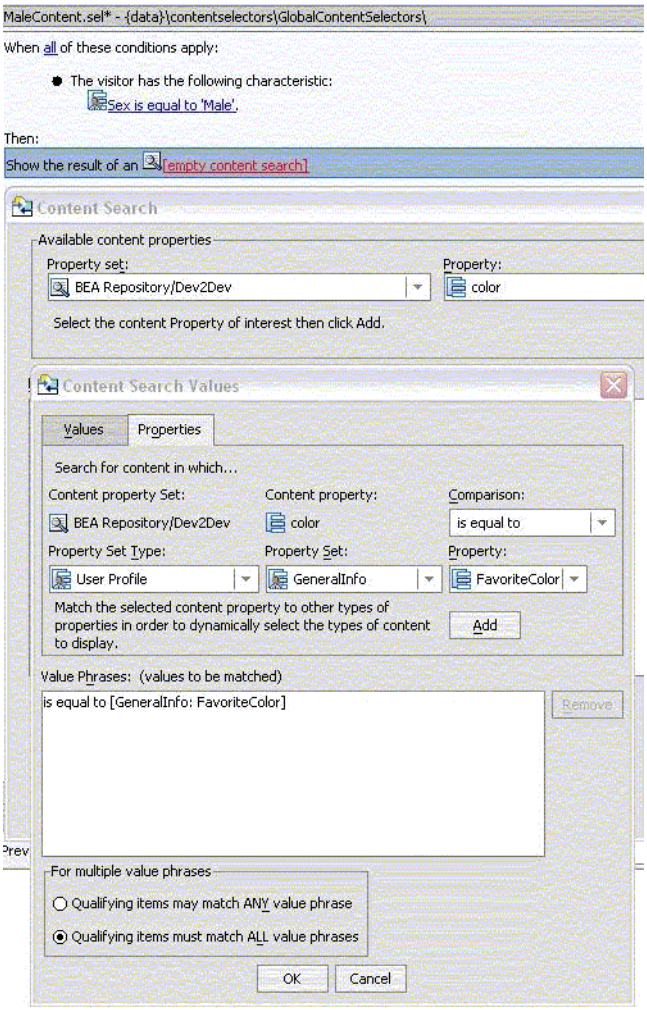
You can delete the date checking condition since it's not needed for this example. In the rule editor, red portions are considered unfinished and will prevent the content selector from operating. You can also see this with the icons in the Document Structure window. Next, you would define the condition by clicking on the underlined part, as shown in Figure 4.

Figure 4 Setting up the condition you added to the content selector



Next, you would define the content query, as shown in Figure 5.

Figure 5 Defining the query for the content selector

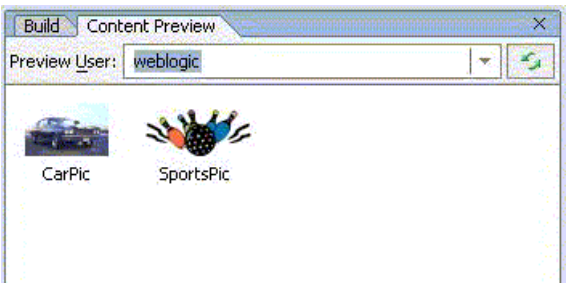


Now, the content selector definition is complete, as shown in Figure 6.

Figure 6 The completed content selector definition with a condition and a query

You can see what content would return from the content query in the Content Preview window, as shown in Figure 7. In this case, since the content query refers to a user's profile, you must provide a username to the Content Preview for it to preview. For queries that don't refer to user properties, you don't need a preview username.

Figure 7 Previewing retrieved content



As you change the query, or switch between queries, the Content Preview pane will show what content Nodes match the query. For Nodes whose primary property is an image, it will attempt to show a thumbnail; for others, it will show an icon based upon the content type.

The next thing is to put the content selector somewhere. You can use the `<pz:contentSelector>` JSP tag do run the content selector and possibly get the results, as shown in Listing 14. We can actually just replace the `<cm:search>` in our previous example and have the rule define what content we see.

Listing 14 Using the <pz:contentSelector> JSP tag to display content from the content selector

```
<%@ taglib uri="content.tld" prefix="cm"%>
<%@ taglib uri="http://www.bea.com/servers/p13n/tags/utility"
prefix="utility"%>
<%@ taglib uri="http://www.bea.com/servers/portal/tags/ad" prefix="ad"%>
<%@ taglib uri="http://www.bea.com/servers/portal/tags/personalization"
prefix="pz"%>
<pz:contentSelector rule="MaleContent" id="nodes" sortBy="cm_nodeName desc"/>
Found <%=nodes.length%> Node(s):
<dl>
<utility:forEachInArray array="<%=nodes%>" id="node"
type="com.bea.content.Node">
<dt><cm:getProperty id="node" name="title" conversionType="html"/></dt>
<dd><ad:render id="node" /></dd>
</utility:forEachInArray>
</dl>
```

You can also drag the `.sel` file on to a JSP, or from the Data Palette window, and it will generate the appropriate tag. So, if the user matches the conditions, it will return the results of the content query. For this example, you can use the Administration Portal's Users & Groups page to set values in a user's profile. Just find the user and choose the Edit User Profile Values tab. In your application, you will want to have places that manipulate the user's profile, such as demographics forms, preferences, and questionnaires, which can affect the personalized content.

You can use multiple content selectors with conditional logic to get hierarchical personalized content, where you try to match the most specific to the least specific, or for mutually exclusive content selectors.

Listing 15 Using multiple content selectors

```
<%@ taglib uri="content.tld" prefix="cm"%>
<%@ taglib uri="http://www.bea.com/servers/p13n/tags/utility"
prefix="utility"%>
<%@ taglib uri="http://www.bea.com/servers/portal/tags/ad" prefix="ad"%>
<%@ taglib uri="http://www.bea.com/servers/portal/tags/personalization"
prefix="pz"%>
<pz:contentSelector rule="FemaleContent" id="nodes" sortBy="cm_nodeName desc"/>
<% if (nodes == null || nodes.length <= 0) { %>
<pz:contentSelector rule="MaleContent" id="nodes" sortBy="cm_nodeName desc"/>
<% }%>
```

```
<% if (nodes == null || nodes.length <= 0) { %>
Sorry, you don't get anything today.
<% }%>
Found <%=nodes.length%> Node(s):
<dl>
<utility:forEachInArray array="<%=nodes%>" id="node"
type="com.bea.content.Node">
<dt><cm:getProperty id="node" name="title" conversionType="html"/></dt>
<dd><ad:render id="node" /></dd>
</utility:forEachInArray>
</dl>
```

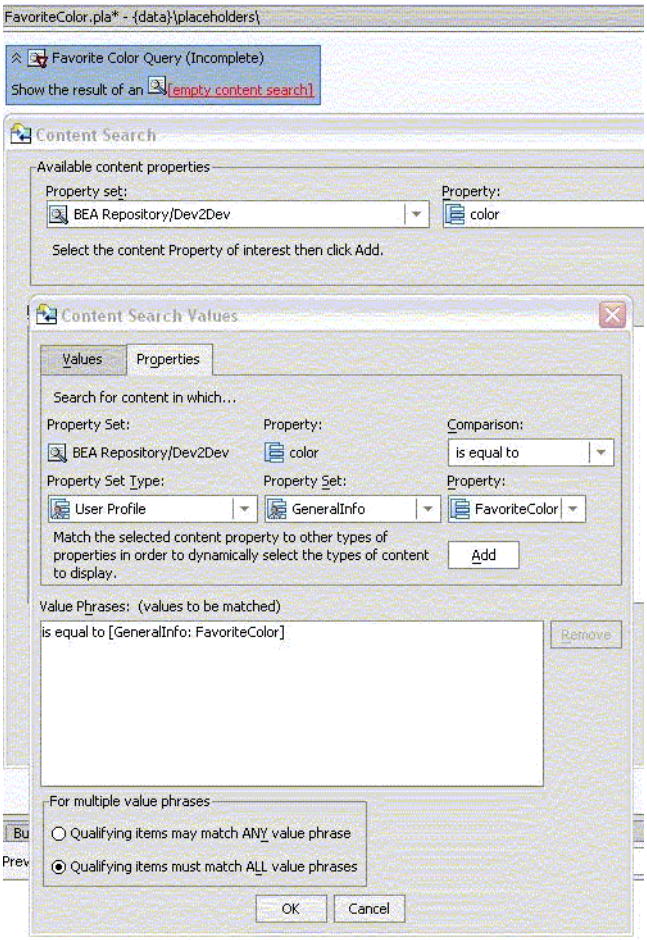
Content Placeholders

To mix automatic content rendering with content selection (like from `<ad:adTarget>`) with externalized definitions, you can use content placeholders. Content placeholder contains multiple queries, but no conditions. Each query has a priority, which is basically a weight.

When a content placeholder is supposed to display content, it uses the `AdConflictResolver` which uses the weight of each query to randomly pick a query; once a query is picked it's like the `<ad:adTarget>` tag for how the content is displayed (it uses the same `AdConflictResolver` and `AdContentProviders`). Content placeholder are managed similarly to content selectors, so they can edited in the Administration Portal as well. This allows you to have externalized rotating banner-style content display.

Content placeholder are created in the placeholders folder of the data project. You can either drag the New Query from the palette on to the placeholder or right-mouse-click and select New Query to add a query. You can use the Property Editor window to give each query a name (which is only used for display purposes) and to specify the query's priority relative to other queries. Once a query is added to the placeholder, it's edited just like in a content selector, as shown in Figure 8.

Figure 8 Defining a placeholder query



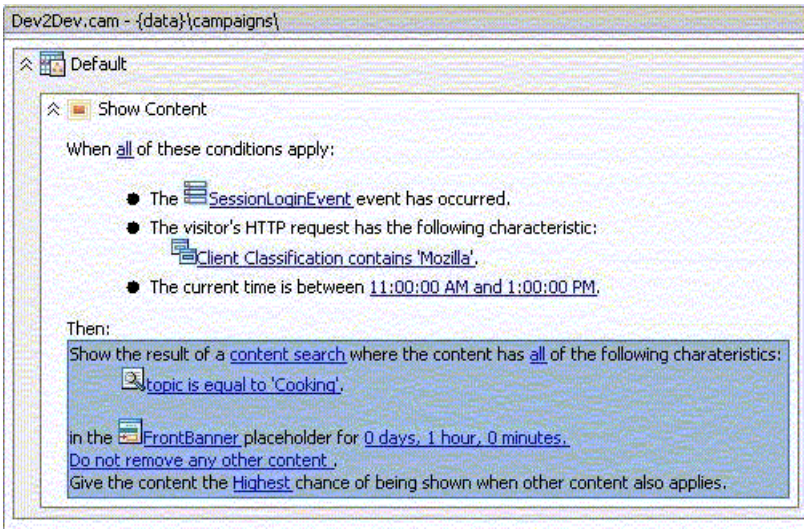
To display the content of the placeholder, you use the `<ph:placeholder>` JSP tag, or you can just drag the `.pla` file on to a JSP and it will generate the tag, or you can drag the placeholder from the Data Palette to your JSP.

```
<%@ taglib uri="http://www.bea.com/servers/portal/tags/placeholder"
prefix="ph"%>
<ph:placeholder name="/placeholders/FavoriteColor.pla"/>
```

Campaigns

Content placeholders can also be supplied queries for particular users by Interaction Management Campaigns. Campaigns allow you to run certain behaviors when Events occurs on the website and specified conditions are true. For example, you might want people who login with a certain browser type at lunchtime to get some special content displayed in a content placeholder. To do that, you would create a campaign (in the campaigns folder of the data project), like that shown in Figure 9.

Figure 9 Creating a campaign



Then, when the FrontBanner content placeholder picks a query for the user, if its `Mix Globals` property it false, it will check if it has campaign-based queries; if so, it choose from amongst only those, otherwise it will fall back to the queries defined in the placeholder. If Mix Globals is true, it will choose from the queries in the placeholder and those from campaigns.

The Events that campaigns respond to can include arbitrary metadata properties, which you can use to further customize the content query, as well as the conditions.

Conclusion

BEA WebLogic Portal provides many features to help you show the right content to the right people at the right time. These technologies can be utilized from within many parts of the BEA

WebLogic Platform to help you succeed in making your application be personalized to your customers.