Oracle® Enterprise Performance Management System

Developer's Guide

Release 11.2.x

F81224-01

May 2023

Using the Financial Management SDK

Related Topics

Financial Management SDK

Financial Management SDK is an object-oriented Java interface that allows consumer programs to access Oracle Hyperion Financial Management Services. It enables consumers to create and open an Financial Management application and to perform various operations such as load, extract and query, and update metadata and data programmatically.

Financial Management SDK is completely written in Java and communicates with HFM servers using TCP/IP protocol. An example developed using Oracle JDeveloper is provided.

Oracle Financial Management users can develop custom pages to solve use cases using the SDK. This can be useful in scenarios where Financial Management desired functionality is not directly offered in Financial Management. Custom pages can be developed in any J2EE technology available to users. An example of a customized page developed in Oracle ADF is also provided in this document. See Oracle® Fusion Middleware Fusion Developer's Guide for Oracle. Also consult JDeveloper and ADF Tutorials

Prerequisites

To develop custom pages for Oracle Hyperion Financial Management using the SDK, users should be familiar with Java and J2EE technologies. Prerequisites:

1. Familiarity with Java and J2EE technology.

2. Oracle JDeveloper 12c (download from Oracle JDeveloper Software site). Apply Patch 34944256, which you can download from Oracle Support. Alternate IDEs can be used. Samples and examples are provided for JDeveloper only.

SDK Reference

The complete SDK reference is available on Oracle Help Center. See HFM Javadoc.

> **Note:**
>
> The SDK packages and classes also contain public classes and methods defined for Oracle Hyperion Financial Management internal use. They are documented accordingly in the reference guide. It is not advised to use these APIs and they may not be supported in future versions.

SDK DEMO, the sample application, creates a Financial Management application called SDKDEMO, opens the application, loads metadata, and then displays the Scenario and Entity members.

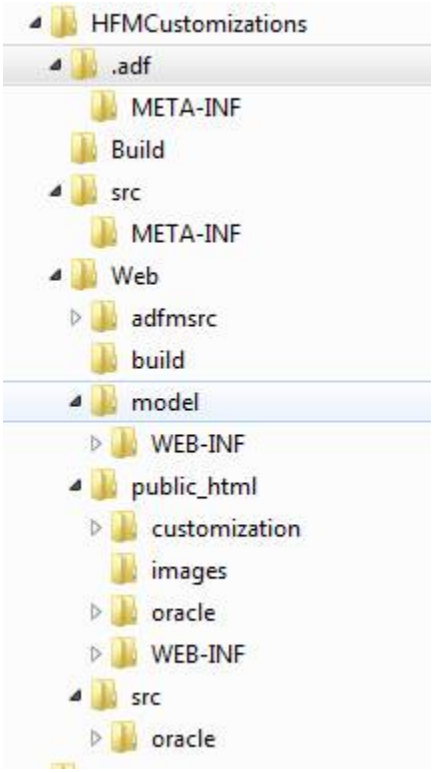
Sample Application Structure

Customization sample files are installed with the Oracle Hyperion Financial Management Web Tier component in this location:

EPM_ORACLE_HOME\products\FinancialManagement\CustomizationSample.

Files distributed: `HFMCustomizations.zip`

File structure within the zip:



High-level directories:

- `.adf`—Contains adf-config.xml inside META-INF directory
- `Build`—Contains build.xml (ant file to build the application)
- `src`—Contains weblogic-application.xml in META-INF directory
- `Web`—Directory with web application files
  High-level directories in the web application:
  - `adfmsrc`—Contains adfm.xml (config file for adf and DataBindings.cpx)
  - `build`—build.xml (ant file for building the project)
  - `public_html`—Web application content
  - `src`—Java source code for beans and business logic layer

Shared Libraries

The Oracle Hyperion Financial Management SDK demo application requires that these shared libraries be referenced:

1. epm-shared-libraries—Deployed by EPM System Configurator and targeted for all HFM servers. Required for HFM.
2. epm-hfm-libraries—Deployed by EPM System Configurator and targeted for all Financial Management servers. Required for accessing Financial Management. Primary API that gives access to Financial Management.

3. epm-thrift-libraries - Deployed by EPM System Configurator and targeted for all Financial Management servers. Required for accessing Financial Management. Provides TCP/IP communication to Financial Management Servers.

Custom Pages

Related Topics

- Sample Application Structure
- Framework for Custom Pages
- Build Project
- Sample Application
- Installation and Deployment
- Steps for Custom Page Developers

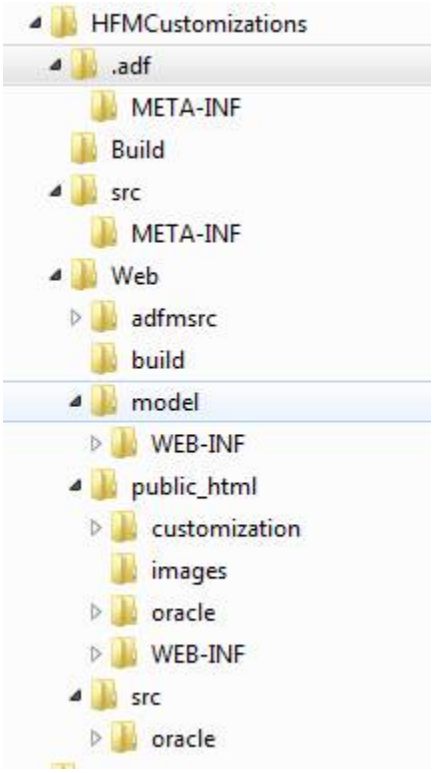
Sample Application Structure

Customization sample files are installed with the Oracle Hyperion Financial Management Web Tier component in this location:

EPM_ORACLE_HOME\products\FinancialManagement\CustomizationSample.

Files distributed: `HFMCustomizations.zip`

File structure within the zip:



High-level directories:

- `.adf`—Contains adf-config.xml inside META-INF directory
- `Build`—Contains build.xml (ant file to build the application)
- `src`—Contains weblogic-application.xml in META-INF directory
- `Web`—Directory with web application files
  High-level directories in the web application:
  - `adfmsrc`—Contains adfm.xml (config file for adf and DataBindings.cpx)
  - `build`—build.xml (ant file for building the project)
  - `public_html`—Web application content
  - `src`—Java source code for beans and business logic layer

Framework for Custom Pages

Related Topics

- Links
- Shared Libraries
- Servlet Filters
- Resources
- Logging

- POV Bar and member selector
- Displaying the POV Bar
- Message and Error Dialog Box
- Images
- Exceptions
- Cleanup

Links

Oracle Hyperion Financial Management Custom pages framework leverages Links in Financial Management. Links can be displayed on a tab or in a browser window. The Links module displays the user's application in an inline frame. Financial Management instantiates the frame and posts a request to the user's external customized deployed page.

In the post request, Financial Management determines the calling application name, the server/cluster on which the application is run, the SSO token, the request type, and the URL for callback to the External Member Selector. The request type indicates whether users are initializing the application or closing the application (when the tab is closed). The custom application uses the information inside the post request from Financial Management and creates an application session with it. After the session is established, users can make Web Service calls back to Financial Management using the Financial Management Java Object Model.

Shared Libraries

Oracle Hyperion Financial Management Custom pages require that these shared libraries be referenced:

1. `adf.oracle.domain`—Deployed by EPM System Configurator and targeted for all Financial Management Servers. Required for common `adf` files.
2. epm-shared-libraries—Deployed by the EPM System Configurator and targeted for all Financial Management Servers. Required for Financial Management.
3. epm-hfm-libraries—Deployed by the EPM System Configurator and targeted for all Financial Management Servers. Required for accessing Financial Management. Primary API that gives access to Financial Management.
4. epm-thrift-libraries - Deployed bythe EPM System Configurator and targeted for all Financial Management Servers. Required for accessing Financial Management. Provides TCP/IP communication to Financial Management Servers.

Library references are available in `weblogic-application.xml in ear\META-INF\weblogic-application.xml`.

To run the custom web application in JDeveloper or integrated Weblogic Server, you must specify an additional parameter `-Depm.oracle.instance`, which should be the same as the env instance home. Typically this variable is: `C:\Oracle\Middleware\user_projects\epmsystem1`.

Servlet Filters

The sample web application also has a BaseServletFilter implementation. This filter caches the post parameters in the user session. This caching is required as the initial requests are subjected to HTTP redirects while adding the window ID and control state parameters to the URL. During this redirect, the POST parameters are lost.

Filters and mappings should be specified in web.xml.

```
<filter>
    <filter-name>BaseServletFilter</filter-name>
    <filter-class>
       oracle.epm.fm.ui.customization.servlets.BaseServletFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>BaseServletFilter</filter-name>
    <url-pattern>/faces/customization/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Update mappings according to the structure of the UI pages developed.

Resources

Messages and captions can be displayed in the user language according to the browser preferences. The Oracle Hyperion Financial Management custom UI can use XLF files to handle resources. The locale is retrieved for the ADF Faces Context in the`ADFUtil::getLocale` method. To retrieve the correct message, users can access the `ADFUtil::getMessage(String msg, String... params)` method. The `msg` parameter is the name of the key to `WebBundle_XX.xlf` (where XX is the locale such as es, en, and so on). A key in the file can be defined as follows:

```
<trans-unit id="ERROR_USER_PARAM">
  <source>
     There was an error getting the user parameter '{0}'.
     Please inspect the logs for more details.
  </source>
  <target/>
</trans-unit>
```

The key can be accessed on the Java (server) side:

```
String msg = ADFUtil.getMessage("ERROR_USER_PARAM ", "SSOTOKEN");
```

It can be accessed on the client side jspx/jsff files as follows:

```
<c:set var="webBundle"
       value="#{adfBundle

['oracle.epm.fm.ui.customization.resources.WebBundle']}"/>
<af:outputText id="ot1" value="#{webBundle.REQUESTED_APPLICATION}/>
```

If the key is not declared in the XLF of the bundle for the locale, the value defaults to the English bundle. This behavior is stipulated in the `Web.jpr` project in the Resource Bundle section.

Logging

The sample application leverages ODL logging. The following parameters must be specified as startup parameters when the web application is started in JDeveloper (in Integrated WebLogic Server):

```
        -
Djava.util.logging.config.class=oracle.core.ojdl.logging.LoggingConfigura
tion
-Doracle.core.ojdl.logging.config.file=logging.xml
```

The logging configuration (`logging.xml`) is in the Build project, where it is stipulated that ALL ODL logging levels are logged for all loggers programmatically established with the `oracle.FMCUSTADF` package name. In addition, all logging is sent to `${domain.home}/servers/${weblogic.Name}/logs/hfm/oracle-epm-fm-cust.log`.

Users must declare a logger within each Java class as follows:

```
private static ODLLogger logger = ODLLogger.getODLLogger(
Constants.ADF_LOGGER_NAME + "." +
Constants.<SOME_UNIQUE_DEFINED_VARIABLE> + "." +
<JAVA_CLASS>.class.getName());
```

The logger can then be used as follows:

```
logger.log(Level.FINEST, "ENTER");
```

POV Bar and member selector

The sample customized application includes a commonly used component for displaying POV information. A taskflow is implemented as part of the sample application. The taskflow requires a POVBarData structure to indicate the look and functionality of the POV Bar. The application in turn leverages the Member Selector URL passed through the initial post request to display the Member Selector when a dimension is selected in the POV Bar. Because the Member Selector also uses a post

request to an inlineframe, the custom application is responsible for populating the post request parameters expected by the HFM Member Selector.

Parameters required for the member selector:

- SSO Token
- Application name
- Cluster
- Slice string for currently selected members in the format A#Sales.S#Actual
- Dimension xml for controlling the behavior of the selections and visibility of dimensions. The Dimension xml is in the format:

```
<Dimensions>
   <Dimension
     name='Account'
     type='Account'
     listSelectable='false'
     listSelected=''/>
   <Dimension
     name='Custom1'
     type='Custom'
     listSelectable='false'
     listSelected=''/>
</Dimensions>
```

In addition, the custom application is responsible for implementing the callback javascript methods from the member selector to determine whether anything was selected or changed.

Displaying the POV Bar

The POV Bar can be displayed as a taskflow on custom pages. On the jspx page, you must add the following:

```
<af:panelGroupLayout id="pgpov" layout="horizontal">
<af:region value="#{bindings.POVBartaskflow1.regionModel}"
           id="regpov"
           clientComponent="true"/>
</af:panelGroupLayout>
```

The `af:region`, which is the ADF UI component for a taskflow, must be embedded in a panelGroupLayout whose name is *pgpov*. This is required to do a partial page refresh of the POVBar. You must rename the region *regpov* and create an associated clientComponent. Define the bindings in the jspx page definition for POVBartaskflow1 as follows:

```
<taskFlow id="POVBartaskflow1"
          taskFlowId="/WEB-INF/oracle/epm/fm/ui/customization/
taskflows/povbar/POVBar-
```

```
                    task-flow.xml#POVBar-task-flow"
          activation="deferred"
          xmlns="http://xmlns.oracle.com/adf/controller/binding">
 <parameters>
  <parameter id="CallbackBean" value="#{viewScope.SampleBean}"/>
 </parameters>
</taskFlow>
```

This code indicates that a taskflow named POVBartaskflow1 can be used as an af:region in the jspx file. The taskflow called is POVBar-task-flow.xml. It also indicates that a CallbackBean named SampleBean must be indicated. This is usually the root Managed Bean view scoped in the adfc-config.xml file. This bean is notified when a change has been made to the POVBar using the Member Selector. The bean's BaseBean::handleCallbackFromPOVBar is called with the new data. The bean can then react to that; however, the PPR of the POVBar has already been done by that point.

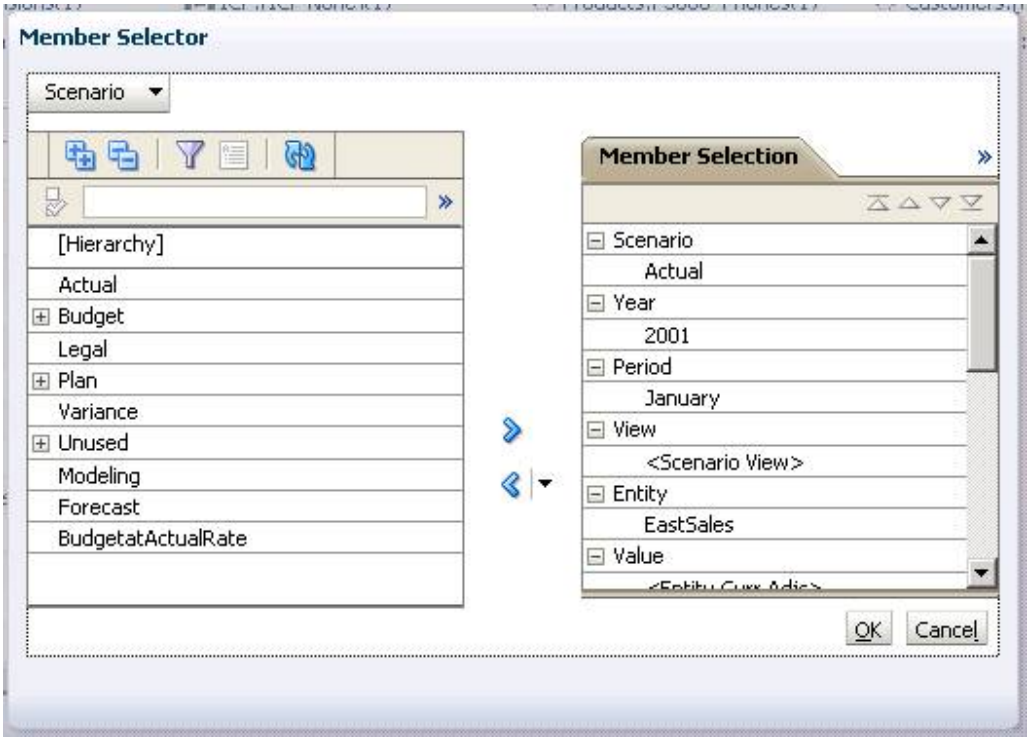
The POVBar taskflow also defines a pageFlowScope bean of type POVBarBean. The taskflow only opens the POVBar.jsff page.

The POVBar.jsff page, leveraging an af:iterator, iterates through the POVBarData to display the POVBar as a group (panelGroupLayout) of af:commandImageLinks. This implies that each component in the POVBar can be clicked on to bring up the member selector. The actual data structure used to iterate over is defined as a List<POVBarItem>. The POVBarItems are extracted from the POVBarData in the POVBarBean using the POVBarBean::getPovBarItems() method. A POVBarItem defines the name of the dimension to be displayed, the name of the member of that dimension to be displayed, and the associated dimension type's icon.

Upon clicking a POVBarItem/commandImageLink, the POVBarData::launchMemberSelector method is called. This method is essentially responsible for launching the External Member Selector via an af:inlineFrame, inside of an af:popup dialog box. This is achieved as follows: The method first searches the POVBar.jsff for the af:popup component by leveraging the ADFUtil::findComponentNear method and displays the popup. The popup contains a dialog, with a single af:inlineFrame component. Similarly to how HFM calls into the Custom Application, the inlineFrame leverages an internal jspx page to POST a request back to HFM for the External Member Selector. This is done to avoid Posting request parameters as URL query parameters, which is the only native way inlineFrames make such requests. The inline frame makes an internal request to /faces/oracle/epm/fm/ui/customization/memberselector/memberSelector.jspx. Similarly again to how HFM calls into its own internal jspx page, a randomly generated UUID is used as a key to the Session map to store all the POST parameters that need to be sent by the memberSelector.jspx. This UUID value is stored as the HFMCUSTKEY URL query parameter. The Session contains a HashMap value corresponding to the UUID. This HashMap contains the POST parameters. They include the ssoToken, appName, cluster, paramValue (slice string), request-parameters (dimension n xml from POVBarData), and the URL to make the POST request to the External Member Selector. This URL was passed in from HFM in the initial call to the Custom Application and cached accordingly in the entry Filter class. This portion of the code is done in POVBarBean::getQueryParams() method.

The memberSelector jspx file initially makes a call to the MemberSelectorBean::init method as part of the f:view beforePhase attribute. The MemberSelectorBean is defined in the adfc-config xml as a request scope bean. The init method simply extracts the POST parameter map from the Session using the UUID key specified on the request parameter HFMCUSTKEY.

It then makes a call to the javascript to programmatically POST to the ExternalMemberSelector with the POST parameters specified above, with an additional parameter (extcaller=fcm) to indicate callbacks from the Member Selector will be done through direct javascript back into an iframe. The member selector should look as follows:



If the user clicks Cancel, HFM expects the parent frame to implement a javascript method named memberSelectorCanceled. This method is implemented in POVBar.jsff. An ADF event is queued in the javascript to 'handleMemberSelectorCancel' within the popup. The popup implements an af:serverListener for this, which in turn calls into the POVBarBean::handleMemberSelectorCancel() method. This method closes the af:popup.

If the user clicks OK, HFM expects the parent iframe to implement a javascript method named hfmSelectorReturn that takes a string XML as a parameter which contains the new values inside the POVBar. The XML was defined above as follows:

```
<Dimensions>
 <Dimension name='Account' val='Expense;Margin'/>
 <Dimension name='Scenario' val='Actual'/>
```

```
 <Dimension name='Entity' val='Stamford'/>
</Dimensions>
```

This in turn calls the hfmSelectorReturn method in POVBar,jsff. The javascript method queues an ADF event for 'handleMemberSelectorOK' on the af:popup and sets a client attribute for the xml to memberSelectorXml. The af:popup implements an af:serverListener for this, that in turn calls into the POVBarBean::handleMemberSelectorOk. This method extracts the client attribute xml and updates the POVBarData with it. The POVBarData::update method does this by parsing the xml using the DOM parser. Then the POVBar is PPR'd, and the callback BaseBean's handleCallbackFromPOVBar method is called. Finally, the popup is closed.

Message and Error Dialog Box

After a message is obtained from the locale resource bundle, displaying a dialog box with this information is sometimes necessary. By default, the `ADFUtil` class provides a way to display these messages, under the assumption that the messages are error messages. The method for this is `ADFUtil::showErrorDialog(String msg)`. Essentially a `FacesMessage` is leveraged to do this. Only the first error message is displayed. If a current error message is to be displayed, another attempt to add a message using this message is ignored.

Images

Images are stored in the ADF default Web Content JDeveloper directory, in Web\public_html\images directory. Thus, if an image must be displayed, it can be done as follows:

```
<af:image id="imgscen"
          source="/images/scenario.png"
          shortDesc="Scenario Dimension"/>
```

Exceptions

All exceptions should derive from the `HFMCustomizationException`. This will indicate that an exception was caught or is handled by the application. This will imply that the exception has already been logged, because it is customary to log the exception only once at the point of occurrence. However, at all root access points in a Managed bean, checking for exceptions is required. If an exception is of the type `HFMCustomizationException`, it will only be required to display an error dialog box. Otherwise, if an unhandled exception occurs, it would be best to log the exception first and then display the error dialog box.

As an example, if a Web Service call fails, the following should be done:

```
try {
      // Code for business logic
} catch(HFMException ex) {
      Log exception
```

```
      Throw new HFMCustomizationException(ex);
}
```

And in the public methods of the Managed Bean, which are accessed in the client side jspx/jsff files – this should occur:

```
try {
      Code for business logic
} catch(HFMCustomizationException ex) {
      Display error dialog
} catch (Throwable ex) {
      Log the exception
      Display error dialog
}
```

Cleanup

The Custom UI page is also responsible for cleanup of resources on the server related to the session. When the Oracle Hyperion Financial Management tab that contains the custom module is closed, an `OnClose` event is triggered from the browser.

The custom application is responsible for handling this event and cleanup of resources both in the Java WebLogic as well as the session on the Financial Management App server, including closing the created application session. If the application does not do this, then the application session will be closed when it has timed out.

When the Financial Management tab in which the inlineFrame resides displaying the custom application is closed, the custom application's `window.onunload` event is triggered. It would be required at this point, at the very least, to close the application session. To catch the event, it would be required of the main jspx page of the custom application to implement the `window.onunload` method. The `common.js` file has a working implementation of this method. The jspx would need to include this file as follows:

```
<af:resource type="javascript" source="/oracle/epm/fm/ui/
customization/js/common.js"/>
```

The javascript method makes an internal AJAX POST request to an internal URL customization/cleanUp.jsp and passes the application session ID as a post parameter to it. The way the application session is retrieved is that the javascript assumes that there is a commandButton by the name *clnbtn1*, which has a client attribute called *appSessionId* containing this information. Thus, it is the responsibility of the root jspx page to include the following:

```
<af:commandButton id="clnbtn1"text="test" visible="false"
clientComponent="true">
<af:clientAttribute name="appSessionId"
     value="#{viewScope.SampleBean.appSessionId}"/>
</af:commandButton>
```

Since the SampleBean should derive from the BaseBean, the appSessionId should be available, having been extracted from the Session Map where it was stored in the Filter.

The AJAX POST request is made using the *XmlHttpRequest* object, which is compatible with IE6+ and FireFox2+. The actual request is not handled by a cleanUp.jsp file. Instead it is handled by `CleanUpServlet.java`, which derives from HttpServlet. This is indicated in the web.xml file by indicating a servlet which handles a request to that URL as follows:

```
<servlet>
    <servlet-name>CleanUpServlet</servlet-name>
    <servlet-
class>oracle.epm.fm.ui.customization.servlets.CleanUpServlet</servlet-
class>
</servlet>
<servlet-mapping>
    <servlet-name>CleanUpServlet</servlet-name>
    <url-pattern>/customization/cleanUp.jsp</url-pattern>
</servlet-mapping>
```

The servlet's `doPost(HttpServletRequest req, HttpServletResponse resp)` method is what is called when a POST request is made to the servlet. It extracts the application session ID from a post parameter named *appSessionId*, and makes a webservice call to close the appsession calling `ApplicationOM::closeWebSession`.

Build Project

The build project is used to compile the Customization Application code and create the `HFMCustomizationApplication.ear` file in the root dist directory to deploy to Oracle Hyperion Financial Management. The ear file uses OJDeploy to create the ear file leveraging the deployment configuration inside the `HFMCustomization.jws` file. The `build_all.xml` file is used to build the project. The following targets are available:

1. *clean*—Removes the root dist directory and the logging.xml file in DOMAIN_HOME
2. *init*—Creates the root dist directory and copies the logging.xml to DOMAIN_HOME
3. *web*—Calls the web's build.xml 'all' target. This in turn calls the following internal targets:
    a. *clean*—Deletes the `Web/classes` and `Web/dist` directory
    b. *init*—Creates the `Web/classes` and `Web/dist` directory
    c. *deploy*—Compiles the code and creates the `HFMCustomizationWeb.war` using ojdeploy and the deployment profiles in `Web.jpr`.
4. *Deploy*—Compiles and creates the `HFMCustomizationApplication.ear` file using ojdeploy and the application deployment configuration in the `HFMCustomization.jws` file.
5. *All* – calls all the other targets in order.

The `build.xml` file relies on two properties files to define all environment variables.

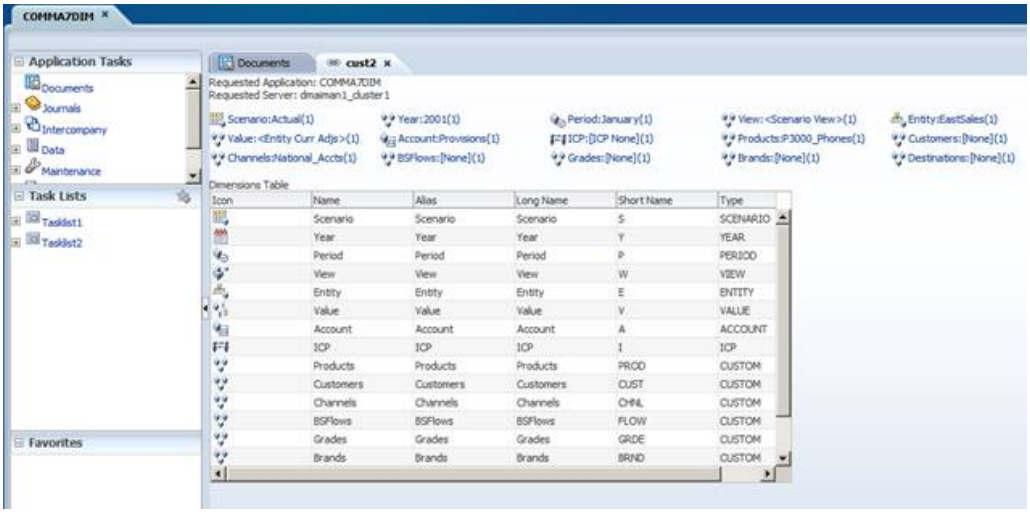
The `build_all.properties` file is used to determine the location of the Oracle installation, the WebLogic URL, username and password.

The `${jdev.middleware}/wlserver_10.3/.product.properties` file is used to indicate the DOMAIN-specific environment variables such as `${DOMAIN_HOME}` to copy `logging.xml`.

The `logging.xml` file is also in this location, and can be modified.

Sample Application

The Sample Application (`sample.jspx`) is used to demonstrate how the Custom Application can be designed. It leverages showing a sample POV Bar and displays a simple table of dimensions and their information. Developers of the Custom Application can leverage the entire flow to start from for a new module. The following is a snapshot of what to expect when running the `sample.jspx`. To see this, the user must create a link to `http://<hostname>:19000/hfmcustadf/faces/customization/sample/sample.jspx`.



Installation and Deployment

Deploy to WebLogic Server

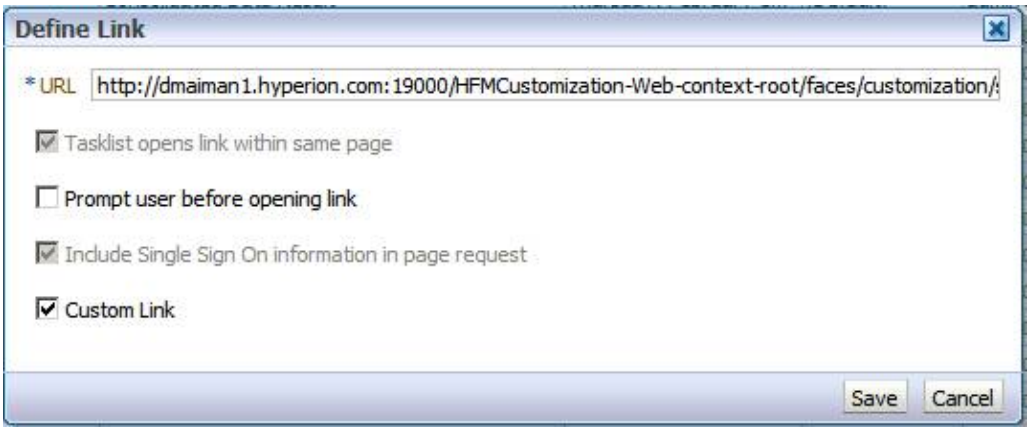
The web application that has the custom UI must be deployed to a Oracle WebLogic Server server as a separate application. JDeveloper generates an ear file. You can use an .ear file or a .war file.

Set up the Web Server

After the application is deployed, users must indicate to the proxy web server (OHS/IIS) that all URLs attempting to access the application must be routed through the web server.

Set up Links in Financial Management

To create the link, users must navigate to the Document Manager and create a document. The document should be a link type. When the document is presented, there is an option to set a custom module. Users should select this option. After this option is selected, it will force the system to open the link in a tab and send an SSO Token:



Steps for Custom Page Developers

1. From JDeveloper, open `HFMCustomization.jws`. This file should be located in *`EPM_ORACLE_HOME`* .
2. Ensure that you are developing on the same machine on which the Oracle Hyperion Financial Management is installed.
3. Right-click the web project and navigate to Libraries and Classpath. Confirm that the `Epm_hfm_web.jar`, `Epm.jar` and `Epm_pdf.jar` files are found. If not, you will need to point to the correct location of these jar files in your *`EPM_ORACLE_HOME`* . Correct paths for the above jars relative to *`EPM_ORACLE_HOME`* : `common\jlib\11.1.2.0`.
4. In the Build project, open `build_all.properties`. Ensure that all property keys point to your locations. You might need to change the following:
   - dev.middleware, which is the location of your JDeveloper installation
   - oracle.middleware, which is the root of your *`EPM_ORACLE_HOME`*
   - weblogic user name and password (required only if you want to deploy the custom web app to an integrated domain created by WebLogic)
   - adminurl, hfm.webservice location
   - hfmcustomization.homedir, which is the location on your jws file.

5. Open `logging.xml` in the build project and confirm that the location of the log is correct. You can use the default location.
6. Create a new main jspx file that will be the entry point to your custom page. Use the `sample.jspx` file as your guide. The file should be located within the customization context directory. As an example, the `sample.jspx` file is located in the `customization/sample` directory. You can create another directory underneath the customization directory and put the jspx file there. Ensure that it is within the customization directory, because the BaseServletFilter mapping in `web.xml` handles all requests pertaining to `/faces/customization/*`. Otherwise, you must create another mapping in `web.xml` and derive your own `BaseServletFilter` class to handle the incoming requests to your discretion.
7. The jspx file probably requires an associated Managed bean in order to do processing, such as making the Web Service calls. Create the Managed Bean using a standard Java class file and have it derive from the BaseBean class. Try to incorporate the bean under the `oracle.epm.fm.customization.beans` package by creating a new package within it. Remember to incorporate this bean in the `adfc-config.xml` file (which is the default unbounded taskflow). It is suggested to make the bean view scope. You can use `SampleBean` as your guide.
8. On the java side, you can use the `ADFUtil` file to display error dialog boxes, get messages from resource strings, and so on. If you need to add a string, you must add it to `WebBundle_XX.xlf` files. You can leverage the `HFMCustomizationException` class to throw valid exceptions.
9. In the root jspx file, include the correct cleanup code. By default, the application session is removed. You do this by including the following:

   ```
   <af:resource type="javascript" source="/oracle/epm/fm/ui/
   customization/js/common.js"/>
   ```

   This will include the `common.js` file which overwrites the default window.onunload event. This implies that when the tab is closed in Financial Management, this method gets called. The method then makes an AJAX call to the CleanUpServlet.java, which will close the created application session that was created in BaseServletFilter. In order for this AJAX POST request to happen, it will require the following to be added to the root jspx as well:

   ```
   <af:commandButton id="clnbtn1" text="test" visible="false"
   clientComponent="true">
   <af:clientAttribute name="appSessionId"
           value="#{viewScope.SampleBean.appSessionId}"/>
   </af:commandButton>
   ```

   The reason for this is that the JavaScript looks for a component named `clnbtn1` and attempts to get the client attribute for the session. If it is desired to have a more comprehensive cleanup code, it will first be required to create a new servlet which will derive from the CleanUpServlet.java class. Users must then overwrite

the doPost method. This class must be registered as a servlet in `web.xml` similarly to how the CleanUpServlet is done. This is done by adding the following:

```
<servlet>
    <servlet-name>CleanUpServlet</servlet-name>
    <servlet-
class>oracle.epm.fm.ui.customization.servlets.CleanUpServlet</
servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>CleanUpServlet</servlet-name>
    <url-pattern>/customization/cleanUp.jsp</url-pattern>
</servlet-mapping>
```

The servlet-mapping indicates the relative URL to which the AJAX call should be made, in order to redirect to the servlet. Thus, the user will need to tweak the window.onunload method to route calls to the new jsp. If there is more information it wishes to post to the request jsp, then that information must be incorporated as an addition clientAttribute on the clnbtn1. The information must be a clientAttribute because by the time the onunload method gets called, the ADF context has been destroyed, and there would be no way to access the information in the bean.

10. If the entry/root jspx requires the ability to display the POVBar and have the ability to change members using the member selector, then the user will be required to incorporate the POVBar task flow into their page as follows:
First include the following in the jspx page:

```
<af:panelGroupLayout id="pgpov" layout="horizontal">
  <af:region value="#{bindings.POVBartaskflow1.regionModel}"
id="regpov"
       clientComponent="true"/>
</af:panelGroupLayout>
```

In the jspx pageDef file, the binding for POVBartaskflow1 should be defined as follows:

```
<taskFlow id="POVBartaskflow1"
      taskFlowId="/WEB-INF/oracle/epm/fm/ui/customization/taskflows/
povbar/POVBar-
             task-flow.xml#POVBar-task-flow" activation="deferred"
       xmlns="http://xmlns.oracle.com/adf/controller/binding">
  <parameters>
     <parameter id="CallbackBean" value="#{viewScope.SampleBean}"/>
  </parameters>
</taskFlow>
```

This can be accomplished by dragging `POVBar-taskflow.xml` into the location in your jspx file. Ensure that you name the panelGroupLayout pgpov and the af:region regpov, because the taskflow makes assumptions about component names to PPR. Note that there is an optional parameter that can be passed into

the taskflow. This parameter should be called `POVBarData`, and it describes the contents of the POVBar using a POVBarData type. The POVBarData type can limit which dimensions to display, as well as stipulate their initial values. If this is not specified, the POVBar taskflow will by default be set to the user's background POV. The parameter can be added in the jspx pageDef file as follows:

```
<parameter id="POVBarData"
value="#{viewScope.SampleBean.povBarData}"/>
```

This example assumes that there is a viewScope bean called SampleBean defined in the adfc-config.xml, and a method called getPOVBarData in the bean, which returns POVBarData.

11. After the application is completed, you can test it by debugging the root jspx page. Right-click the jspx and choose debug. This process compiles the code and deploys an extracted ear to the default Oracle WebLogic Server for JDeveloper.
12. Alternatively the user can deploy the application to the production WebLogic Server where Financial Management resides. This is done by first running the `build_all.xml` target `all`. The `HFMCustomizationApplication.ear` file will be deployed to the dist directory. The user will need to deploy the ear file manually through the WebLogic admin console. It is recommended that this be deployed to the same WebLogic Server to which Financial Management has been deployed, and to which all of the shared libraries that Financial Management uses are deployed. In addition, it would be best to mirror the deployment configuration of Financial Management.
13. After the application is deployed, the user must indicate to the proxy web server (OHS/IIS) that all URLs attempting to access the application will be routed through the web server.
    a. Navigate to the OHS configuration directory: *`$EPM_ORACLE_INSTANCE`*`\httpConfig\ohs\config\OHS\ohs_component`. For example:

       Add entries to `mod_wl_ohs.conf` file. First you add the following:

       ```
       RedirectMatch 301 ^/hfmcustadf$ /hfmcustadf/
       ```

       Then add the following entry:

       ```
       <LocationMatch ^/hfmcustadf/>
           SetHandler weblogic-handler
           WeblogicCluster hostname:7363
       </LocationMatch>
       ```

       Let's make the assumption that the OHS port is 19000.

    b. Steps to configure IIS if using a web server:
       i. On the Foundation server, open the directory: `EOI\httpConfig\VirtualHosts`.
       ii. Copy an HFM directory and paste it.

iii. Rename the new directory to match the context name (for example, hfmcustadf).

iv. In the directory, edit `iisproxy.ini` and modify these properties:

- WLForward path - to match the context path, for example: hfmcustadf
- WebLogicHost/WeblogicCluster—Hosts on which hfmcustadf will be deployed. In most cases you will not have to change this property, because the custom web application runs on all servers on which HFMADF is running.
- WebLogicPort – Port number on which WebLogic is running. If WebLogicCluster is being used, the port is not necessary, because it is part of the cluster.

c. Open the IIS Administration page and edit the application with these details:

i. Alias—Same name as the custom adf context, for example: hfmcustadf

ii. Physical Path—Point to the new directory you created in step b

iii. ApplicationPool—Use hfmadfAppPool

14. From the HFM Document Manager Module, create a link and select the Custom Link option. By default, this selects that the Link opens within the same page, and includes the SSO token option. For the link, use a relative path, such as: /hfmcustadf /faces/customization/newsampledir/MyNewSample.jspx. You may choose to use an absolute path, however, that may lead to browser security checks and the page may not be displayed inline. To use an absolute URL, deselect the option to open the link inline.

15. After you double-click the application, the inline frame should display your application.

> **Note:**
>
> Because inline frames are used, the security to the URL should not be restricted in the browser.

Useful References

1. Oracle® Fusion Middleware Fusion Developer's Guide for Oracle
2. JDeveloper and ADF Tutorials
3. Javadoc - Oracle Hyperion Financial Management Java Object Model