An Oracle White Paper
June 2010

Building Customizable Oracle ADF Business Applications with Oracle Metadata Services (MDS)

ORACLE

Executive Overview

A quote by Charles Darwin [1809-1882] is that “It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change”. This statement also holds true in information technology (IT) where developers need to build long living business applications that are able to adapt to organizational changes, differences in end user preferences and changes in the business they support. Developers who create software for commercial or in house use, have the requirement to tailor their application to specific customer needs in a way that survives the installation of maintenance and error correction patches to the application over time. The ability of applications to adapt to changes is not a luxury in Enterprise 2.0. It is a necessity that needs to be considered in the application design and that should the drive selection of the development platform and architecure.

Oracle Fusion Middleware (FMW) and the Oracle Fusion development platform combine proven technologies for application developers to build, deploy and run robust Java EE and SOA applications. With metadata on all tiers, the Oracle Fusion development platform, including the Oracle Application Development Framework (ADF), supports organizations in building cutting edge rich enterprise business applications (REA) that are customizable and personalizable in all dimensions.

Oracle Metadata Services (MDS) is the customization and personalization framework integral to Oracle Fusion Middleware. It is a key differentiator of the Oracle development platform compared to other Rich Internet Application (RIA) and SOA solutions on the market. MDS seamlessly integrates with state of the art Java EE and Ajax technologies used in Oracle Fusion application development and the declarative programming paradigm promoted by the Oracle JDeveloper IDE and the Oracle ADF framework.

This paper focuses on how-to build customizable business applications with the Oracle Fusion development stack, including ADF. It steps you through the basics of the Metadata Services architecture and demonstrates how you might integrate it into your custom applications. In addition, this paper covers advanced topics such as the integration of ADF Security, and how to programmatically persist component and attribute changes.

Introduction

A good investment is one with longevity. This is also true for custom applications you build. Worldly wisdom says that nothing is as certain as change. Therefore, to protect your investment in enterprise application development, the need to react to change should be considered a part of the application design and not as an optional nice-to-have. The ability of an application to change in response to external conditions, like the needs of the individual user working with it, also helps to ensure good usability and leads to higher rates of end-user adoption.

"It is not the strongest of the species that survives, nor the most intelligent that survives.
It is the one that is the most adaptable to change" - Charles Darwin [1809-1882]

Before metadata customization, software developers used triggers and events built into an application to ensure that at runtime the application user interface (UI) tailors itself to the needs of the authenticated user, his or her responsibilities and organisational role and vertical. The areas in which customization is used include

- Rebranding
- Personalization
- Customization

Rebranding

The Brand is the visual identity of an application that associates it with a company, industry or market. The visual style of an application may change for various reasons, including acquisition, regional differences, modernization or other, for example seasonal, reasons. Look and feel definitions, like images and cascading style sheets (CSS) that have hard coded references in the application program code make it diffcult to change the look and feel dynamically and therefore should be avoided.

Personalization

Every end user is unique.. Even when sharing the same education, skills and job description, application users tend to have different preferences when it comes to how they like the user interface to be rendered. For example, agents that help customers online may have individual preferences of how the customer call history is displayed. Some agents may prefer a summary table to show the problem statements first, while others want to see the date of previous calls to be shown first. Similar, some agents may prefer list of values when querying or inputing data, while others don't. Personalization, or user customization as it is also referred to allows users to make themselves at home within an application. Throughout this paper, we use the terms personalization and user customization interchangeably.

Customization

Application developers define customizations in cumulative layers that are applied at runtime to change a base application to the needs of a specific installation or user group. The customization layer itself is pre-configured and referred to as "seeded". Customization may adapt an application to regional or industry differences, or just change its behavior and UI according to the responsibility of a user or group.

Figure 1 shows a user interface that is customizable for normal users and power users. Compared to normal mode, the power user mode replaces list of values in the table filters with input fields and wizards with a single input form. A similar, very popular requirement expressed by customers is to show more or fewer attributes on an input form based on the configured user roles within an application. This requirement can be achieved easily using MDS customization.

Another aspect of customization is security, which is not an obvious use case. However, using seeded customization to tailor the UI and navigation for the indvidual user helps to suppress the display of sensitive information if a user is not authorized to see it.

Though all the above can be achieved using conditional statements in Java, such a practice would become unmanageable quickly. Source code, added declaratively or in Java, that does not add to the application core business functionality should not go into an application as it makes maintenance harder and more expensive. To address the

requirement for rebranding, personalization and customization efficiently, applications should be designed such that differences between the versions are kept external to the base application.

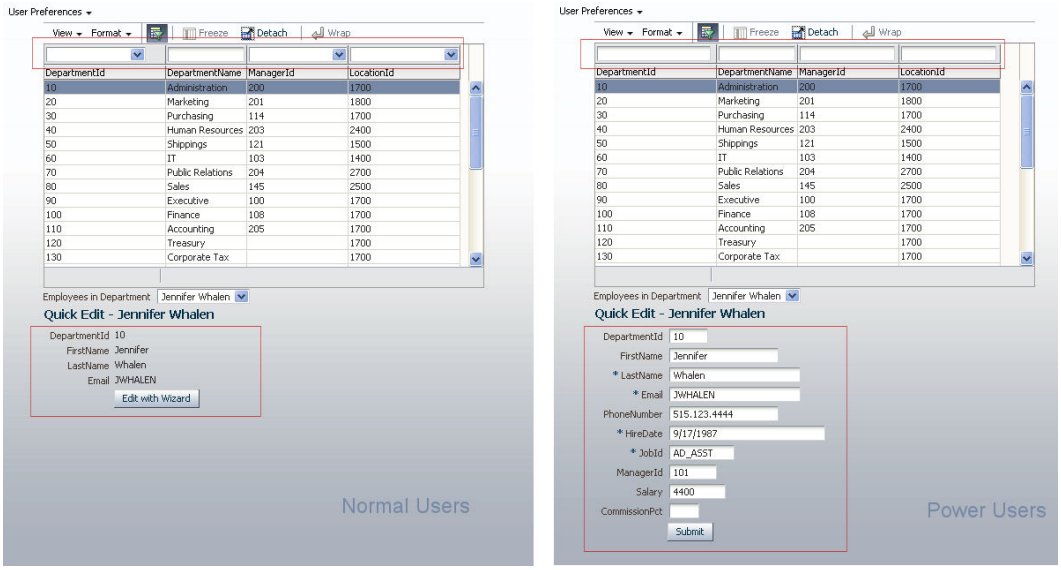

Figure 1: Customized user web interface, adapting to the user role as "normal" and "power" users. The difference shows in the select options in the table filter, as well as in the input form, which for normal users is read only as they are supposed to use a multi step wizard.

About Oracle ADF

Oracle ADF is anchored in the UI binding work Oracle proposed for standardization in Java EE as JSR-227. ADF is actually the family name for a whole set of integrated technologies that help developers of different programming backgrounds to productively and quickly build leading edge Java EE web applications. These applications leverage standards such as JavaServer Faces, XML, CSS, a selection of Java EE business services, elements of Service Oriented Architecture (SOA) and the aforementioned binding layer. ADF uses XML where ever possible, to configure and drive these technologies. This use of metadata is a key component of the overall ADF architecture.

Unlike other meta-information technologies like Java annotations, XML enables out-of-the box customization and personalization as it does not require application code to be compiled to apply changes to the application behavior or user interface.

Oracle ADF is a core technology component within Oracle Fusion Middleware used by Oracle for its own in-house product development teams and customers alike. Because Oracle uses the same version of ADF that customers do, customers get a mature and future safe development stack that is already proven for building large scale business applications for the enterprise.

About Oracle Metadata Services

Oracle Metadata Services (MDS) is the personalization and customization engine within Oracle Fusion Middleware that manages all of this XML metadata of behalf of components such as JDeveloper and ADF.

Metadata is used by the following components

- ADF Faces rich client Java Server Faces components
- ADF Faces Data Visualization Tools (DVT)
- ADF Task Flow
- ADF Binding layer
- ADF Business Components
- Oracle WebCenter

How MDS Manages Customization and Personalization

Customization and personalization are dynamic structure and property changes to the metadata of application documents like views, bindings and task flow definition files. Modifications that, for example, are applied as customization to a page or a page fragment include the addition, removal or property changes of UI components.

An application that is customized with MDS consists of a base application and one or many customization layers that hold the modifications that are applied at runtime. A customization layer is defined by a set of metadata documents that are stored in a metadata store on the file system or the MDS database repository, and a customization layer object, a Java class that determines when to apply the changes. This customization layer class determines the conditions under which a specific customization is applied to the application at runtime. As you will see in this paper, using Java gives you great flexibility and control over how sophisticated customization use cases can be implemented.

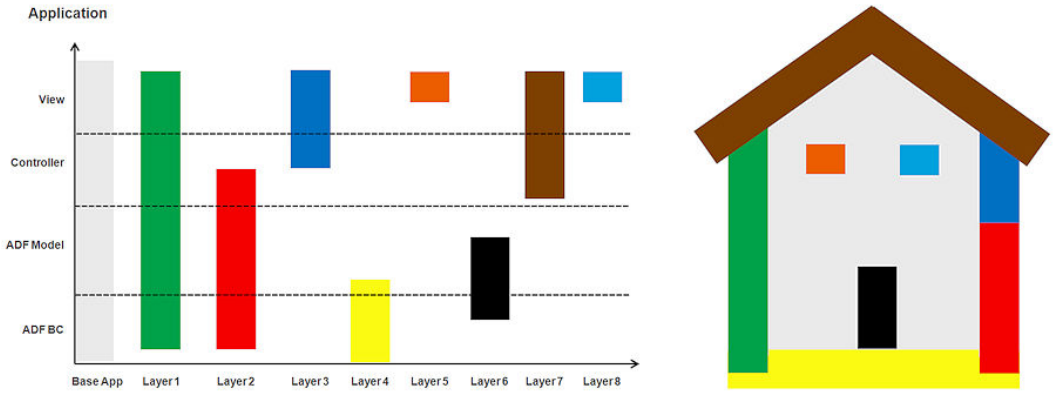

Figure 2: Customization layers are applied to construct the final user view of the application

As shown in figure 2, each document in a customization layer defines changes for a single base application document, like for a page or page fragment. The runtime appearance and behavior of an application then is the outcome of the merged content of a base application with its document change definitions stored in the MDS repository.

Customization in MDS is applied hierarchically in the order customization layers are defined for an application. This way, possible conflicts that may occur when different layers change the same setting of a component, are handled in a predictable way. As you will see later, the order of the layers is not necessarily statically defined but can also be influenced by the customization class object and thus the developer.

MDS stores its customization definitions separately from the base application so that changes applied to the base application, such as a patch, have no impact on customization unless components are removed, in which case no customizations for this component are applied at runtime.

If modifications of a base document remove a metadata element that has pending customizations, then no customization is applied for the missing artifacts, and he application continues running with no errors.

Figure 3 shows another view of the layered customization architecture, including a personalization layer, in which customizations defined by the end user users at runtime is always added as the topmost layer. Note however that availability of user personalization points is also under the control of the application developer. The developer can define which elements of the UI can be both changed and persisted in response to user interaction.

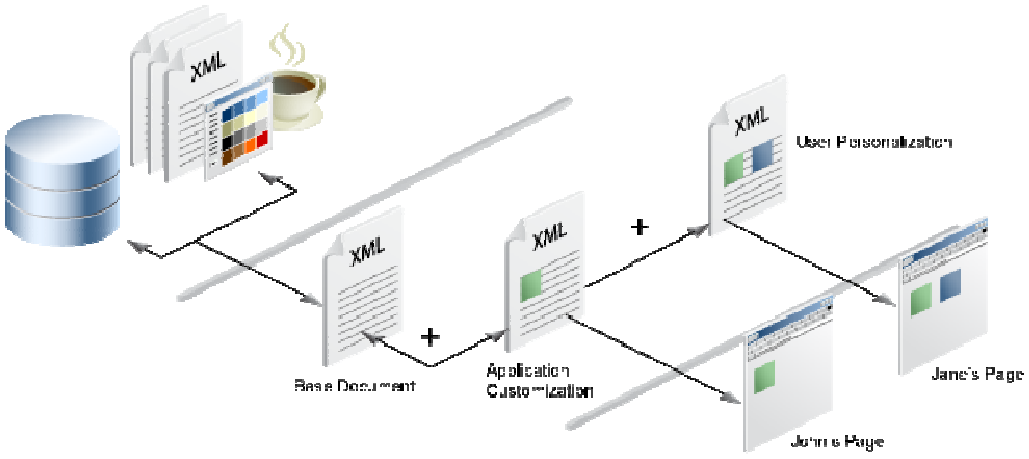

Figure 3: Layered customization and personalization in MDS

Note: Technology alone does not make a successful MDS customization. It is the understanding developers have about the business modeled by an application that allows them to identify layers that effectively tailor the software to individual user and site needs. This is why customization must be considered an essential part of core application design and not as something to apply retrospectively.

Runtime

The MDS runtime engine applies the document changes defined in the customization layer for the requested application when the view of a page renders. Because customization definitions may be used with multiple instances of an application, customization layers use configurable caching to determine when customization should be dynamically applied and when it can be read from the cache. Customization layers are applied in the order they are configured for an application with the personalization layer added on top. No additional coding is required by the application developer to enable end user personalization of an application, just a configuration of which options should be allowed. Personalizations, such as storing changes of the table column display index, is implemented and managed automatically by the ADF Faces components.

Design Time

To pre-seed customizations into an ADF application, Oracle JDeveloper must be started in the Customization Developer Role. This is an option displayed during the IDE startup, and also accessible from the tools preferences dialog. When the Customization Role in the IDE is enabled, the application opens in such a way that only customizable metadata files are editable. Application developers select the customization layer and layer value for which they want to define changes and start editing the application using the same design time editors and tools that they used for building the base application. Any change that is made, however, does not touch

the application definition but rather is stored in a separate metadata file specific to this layer and laver value.

Enabling Customization and Personalization

By default, Oracle Metadata Services is not enabled for Oracle Fusion web applications and all runtime changes of a component that are performed by the application user, such as reordering of table columns are transient and only last until the page is next refreshed.

To configure component changes to last longer, application developers need to enable change persistence in Oracle ADF Faces. Change persistence is a component level framework in ADF Faces that tracks UI changes by the application user or programmatically applied by the application developer in response to a user action. This allows these changes to persist for the duration of the application user session. Oracle MDS can then be used to further extend the change persistence behavior to persist changes across application restarts. This enhanced behavior is enabled through a simple options dialog shown in Figure 4. Under the covers, these options make changes to the web.xml and the adf-config.xml configuration files for the application.

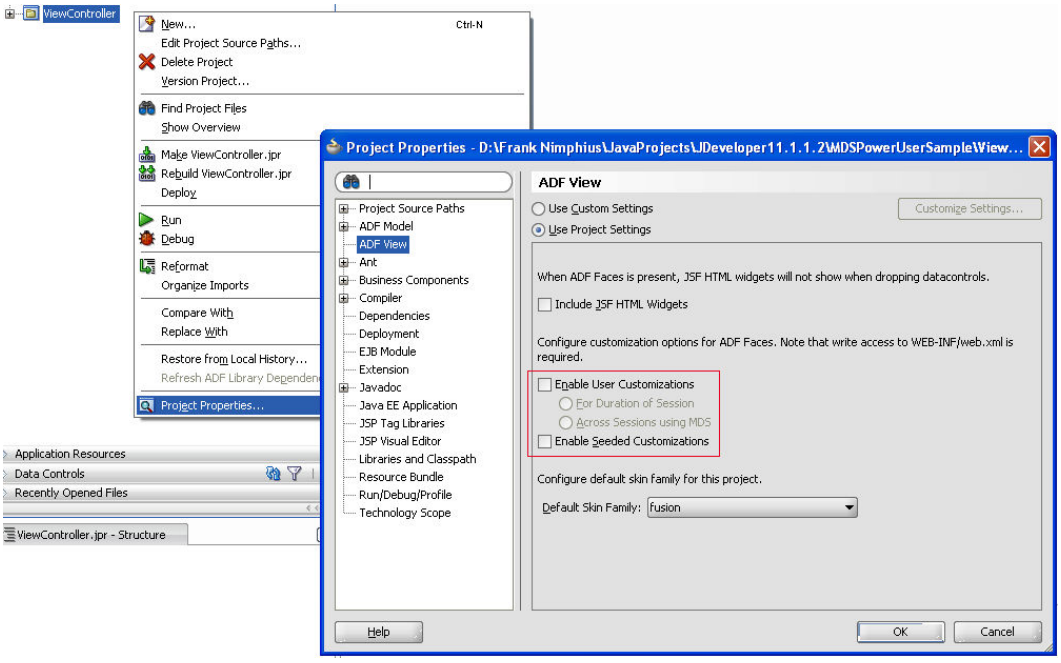

Figure 4: View layer project properties to enable Change Persistence and MDS

Figure 4 shows the two check boxes provided by the view layer projects ADF View settings. These are used to enable user runtime customization and seeded (design time) customization.

User customization, or personalization, saves UI changed applied declaratively by the user or programmatically by the application developer in response to a user action. Personalization can be saved for the duration of the user session or across session re-starts using MDS. The following context parameter gets added to the web.xml configuration file if session scope persistence was chosen.

```
<context-param>
  <param-name>
    org.apache.myfaces.trinidad.CHANGE_PERSISTENCE
  </param-name>
  <param-value>session</param-value>
</context-param>
```

Saving changes in the user session is an all or nothing choice. The developer cannot refine which components and which changes should be persisted. If there is a change on a component, for example the collapse of a particular panel box that the developer wanted to exclude, then the only option would be to override the change using Java.

Choosing MDS persistence, the following entry is added to the web.xml file

```
<context-param>
  <param-name>
    org.apache.myfaces.trinidad.CHANGE_PERSISTENCE
  </param-name>
  <param-value>
    oracle.adf.view.rich.change.FilteredPersistenceChangeManager
  </param-value>
</context-param>
<context-param>
  <param-name>
    oracle.adf.jsp.provider.0
  </param-name>
  <param-value>
    oracle.mds.jsp.MDSJSPProviderHelper
  </param-value>
</context-param>
```

The `FilteredPersistenceChangeManager` class extends the default Apache Trinidad persistence and, for each component change request, verifies that no restrictions have been applied by the developers to individual component instances, using the component *persist* and *dontPersist* attributes, or to a component type as a whole.

The following ADF Faces components expose the *persist* and *dontPersist* properties in the Oracle JDeveloper Property Inspector

- af:panelBox
- af:showDetail

- af:showDetailHeader
- af:showDetailItem
- af:column
- af:tree
- af:treeTable
- af:panelSplitter
- af:calendar

If no instance specific restrictions exist for a component, then restrictions are validated on the component type level. This is handled by a global configuration in the adf-config.xml file. The document change is passed to the document change manager configured in the adf-config.xml file to write it to MDS if no restriction is found. Document change requests that fail are still persisted for the current Session.

Note: Figure 6 depicts the change request evaluation in MDS

Note: To personalize ADF applications using MDS, you need to have at least one customization layer defined in form of a customization class specified in the adf-config.xml file. MDS provides default customization classes, for example `UserCC` that can be used to handle changes specific to a user. How to configure customization classes in adf-config.xml and how to enable changes persistence for components in MDS is covered later in this paper.

If the view layer project does not already contain ADF bound components when enabling customization with MDS, two additional components are added to the web.xml file

```
<filter>
  <filter-name>ADFLibraryFilter</filter-name>
  <filter-class>
    oracle.adf.library.webapp.LibraryFilter
  </filter-class>
</filter>
…
<servlet>
  <servlet-name>adflibResources</servlet-name>
  <servlet-class>
    oracle.adf.library.webapp.ResourceServlet
  </servlet-class>
</servlet>
```

The ADFLibraryFilter and the ResourceServlet are used by ADF to load documents from ADF library archive files (JAR) if they are not found in the document root of the class path. All static

content found in ADF Libraries are either copied to the document root or directly added to the servlet response.

The view layer project setting "Enable Seeded Customization" updates the web.xml with the configuration required for MDS-JSP integration.

```
<context-param>
  <param-name>oracle.adf.jsp.provider.0</param-name>
  <param-value>oracle.mds.jsp.MDSJSPProviderHelper</param-value>
</context-param>
```

The configured `MDSJSPProviderHelper` class returns the requested page definition as merged XML document loaded from MDS. When enabling MDS, another ADF configuration file, adf-config.xml in the .adf/META-INF directory of the application, is updated as shown below

```
<persistent-change-manager>
  <persistent-change-manager-class>
    oracle.adf.view.rich.change.MDSDocumentChangeManager
  </persistent-change-manager-class>
</persistent-change-manager>
```

In Oracle JDeveloper, the adf-config.xml file is accessible from the Application Navigator where it is located under Application Resources | Descriptors | ADF META-INF. You can open the configuration overview editor with a double click on the adf-config.xml entry.

The adf-config.xml overview editor contains a "MDS Configuration" category and a "View" category, which both are used to configuring MDS for an application.

The "MDS Configuration" category allows you to define customization layers for an application, for example the UserCC customization layer that is located in the oracle.adf.share.config package and that ship with ADF.

The `MDSDocumentChangeManager` extends the `SessionChangeManager` class, the Apache Trinidad base handler for persisting component changes, and updates the MDS metadata document associated with a page. The `MDSDocumentChangeManager` is referenced by the `FilteredPersistenceManager` configured in the web.xml file after the document change request has been validated against restrictions defined in the adf-config.xml file.

Note: Editing the web.xml file to replace the `FilteredPersistenceChangeManager` class with the `MDSDocumentChangeManager` class bypasses the change restrictions. While this may sound like a good idea, beware that doing so can reduce performance because of an increase in IO traffic between the application and MDS. In addition, you lose control over which component and component instances to persist changes.

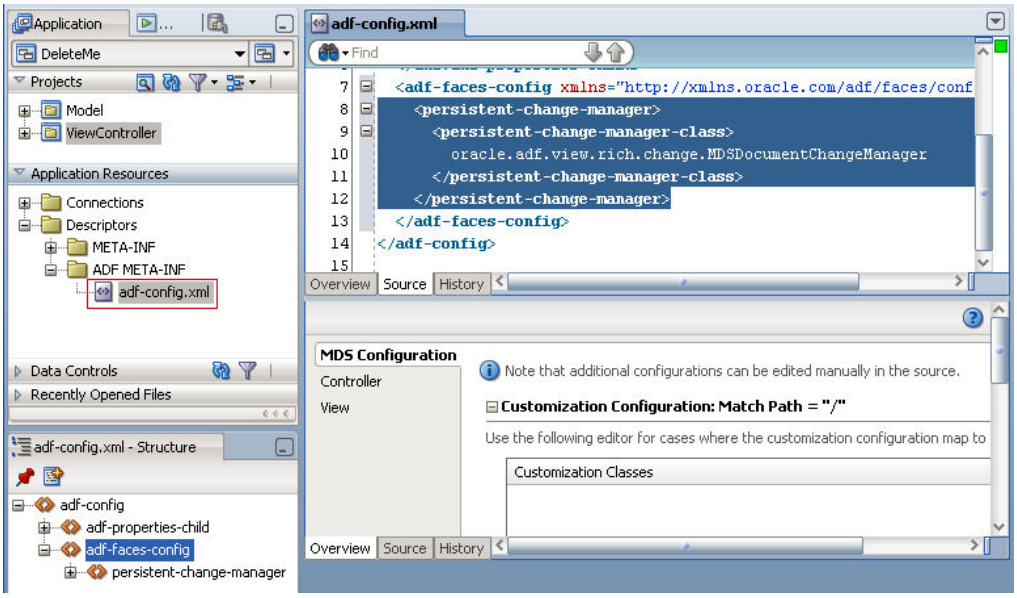

Figure 5: adf-config.xml file in Oracle JDeveloper

ADF Faces Change Persistence Framework

The change persistence framework has its origins in the Apache Trinidad open source project that provides the foundation for the ADF Faces rich client framework. The framework supports implicit and explicit persistence and restoration of component changes.

Implicit component changes are performed by the ADF Faces component renderers, for example; saving the changed column order index of a table component or the disclose state of a ShowDetailHeader.

Explicit component changes, on the other hand, are initiated by the application developer in Java. For example, developers may use Java to persist the changed order of panel boxes in a PanelDashboard, or the order of input text fields in a PanelFormLayout.

Implicit and explicit component changes both usually occur in response to a user-UI interaction and can include

- Change to an attribute value
- Addition or removal of a child component
- Re-ordering of children within the same parent
- Movement of a child component into a different parent
- Addition or removal of a facet

Based on the configuration, the change persistence framework saves changes as temporary component changes in the session or as permanent document changes in the MDS repository. Developers that use the change persistence Java API for implementing explicit component changes don't need to worry about handling this distinction as the programming interface – the `ChangeManager` API – automatically handles the distinction between component and document changes based on the configuration.

Change Manager

As mentioned, the change persistence framework is used for implicit and explicit component changes. Implicit component changes are automatically applied by the component renderer, whereas explicit changes require the developer to execute Java code in a managed bean method in response to events such as a button press. Both, implicit and explicit changes are controlled by an instance of the `ChangeManager` class.

A single instance of the `ChangeManager` class exists in ADF Faces applications to persist changes that are applied during a request. Application developers access the `ChangeManager` from the `AdfFacesContext` to programmatically apply component changes. These changes can be applied during any phase of the JSF lifecycle once the server side in memory component tree is available.

But wait – let's not jump the gun and get into coding too quickly. `ChangeManager` coding examples will come later in this paper. For now it is important to understand that customization and personalization is a framework feature of ADF Faces that you can extend with seeded customization defined by MDS.

Application Personalization Flow with MDS

Personalization is the process in which application users make themselves at home within an application. As shown, the duration of a change is configured at the application level and may last for the length of a user session or survive application restart. Changes applied by these personalizations are added on top of other (seeded) customization layers defined by an administrator or the developer.

As shown in the diagram below, in personalization the component change is initiated by a user action, like collapsing a panel header. With the next request, the change is passed to the server. If user customization is configured to use MDS, the MDS mode, whether customization is through the ADF view layer or Web Center Composer, determines how the change request continues.

Without MDS, component changes are only persisted for the duration of the session, and are stored in a session variable, indexed by their viewId and componentId. They can then be applied upon subsequent page requests.

Note: The runtime personalization of components contained in a page template are applied to the instance of the current view only. This is because the changes are indexed by the combination of viewId and the component Id and the viewId in this is the viewId of the page or page fragment that consumes the page template, not the template itself

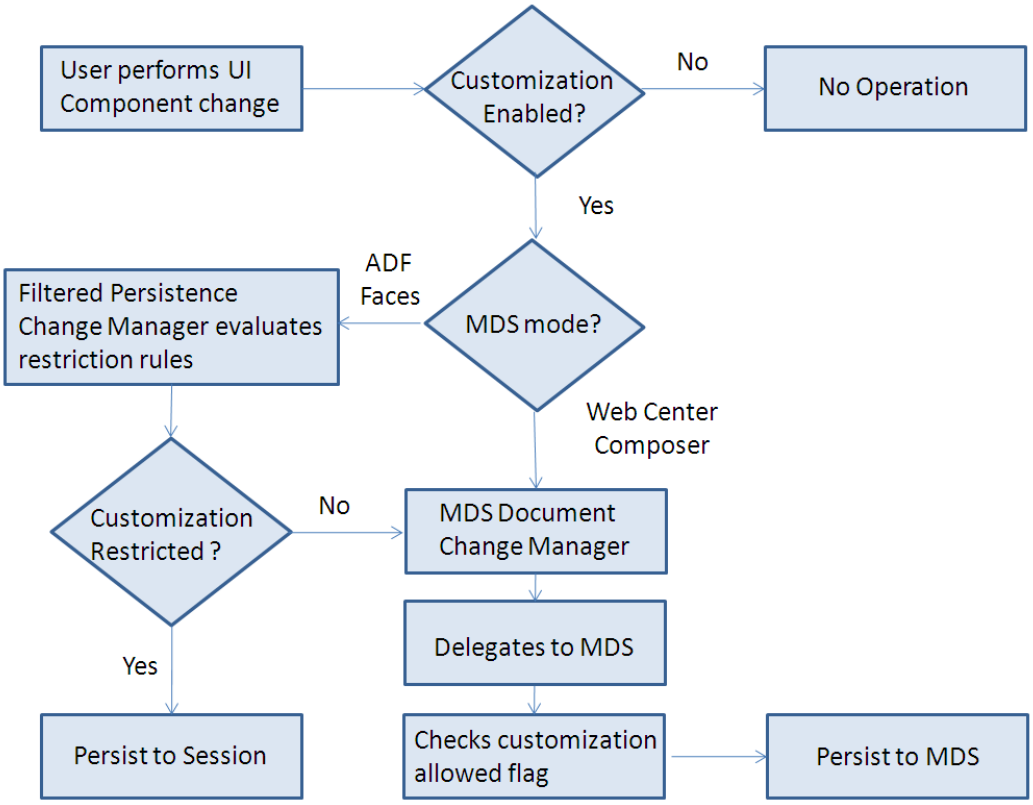

Figure 6: Personalization Flowchart

If MDS is configured as a persistence store and the change request comes from the ADF view layer, then the configured `FilteredPersistenceChangeManager` checks if the requested customization is restricted on the component instance or in general for the component type. If customization is restricted in MDS then the component change is written to the user session only and will be forgotten when the session is abandoned. In addition, messages similar to the one shown below will appear in the log output

<FilteredPersistenceChangeManager><_addDocumentChangeImpl> The DocumentChange is not configured to be allowed for the component: RichColumn[UIXFacesBeanImpl, id=c4]

To persist changes for a specific component in MDS the adf-config.xml file needs to be edited to add the component and the attributes to store changes for. For this, open the adf-config.xml file

that is located in the Application Resources | Descriptors | ADF META-INF node of the Oracle JDeveloper Application Navigator.

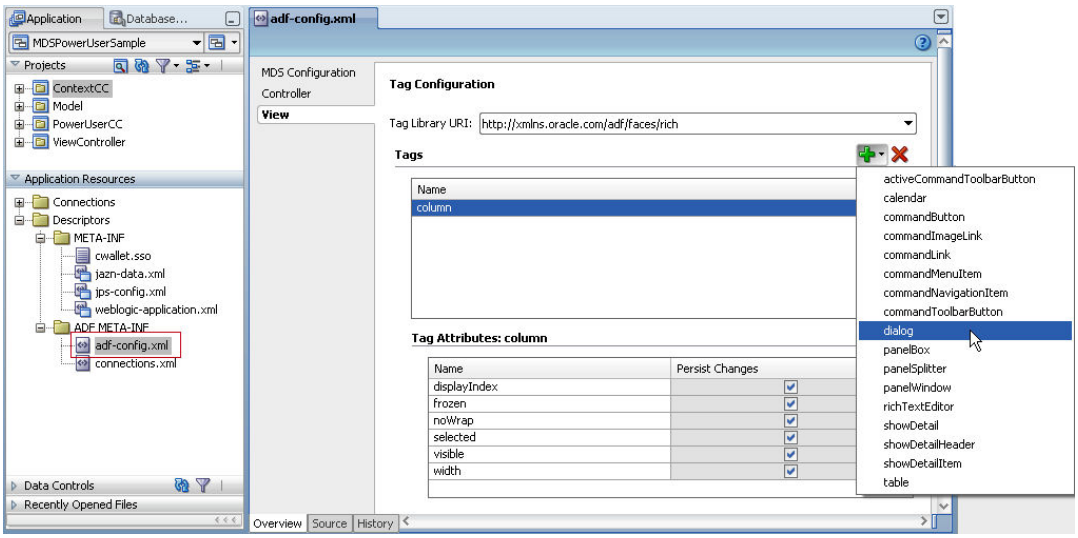

Figure 7: Configure ADF Faces components and attributes for global component type persistence in MDS

Type persistence shown in figure 7 applies to all instances of a component within an application unless a component instance *dontPersist* attribute setting disallows persistence for a particular usage of the component. On the other hand, if a component type as a whole is not selected for runtime customization, the *persist* attribute can be used on a component instance level to enable personalization for this particular usage of the component.

If the customization is not restricted, then as a last check, authorization is verified using the *customization allowed* property to ensure that only authorized users perform changes that are persisted to MDS.

As mentioned earlier, to make user customization work, you need to configure a customization layer that identifies the current user. The supplied user customization layer in ADF is `UserCC`, which returns the name of the Java EE authenticated user for MDS to save changes. In the following section, we explain how to configure the `UserCC` customization class and what it does for you.

When MDS customization changes are persisted

Application UI changes that are applied by the user, for example when collapsing a PanelHeader component or PanelBox are persisted with the next request. For this, MDS is integrated into the ADF lifecycle as shown in figure 8.

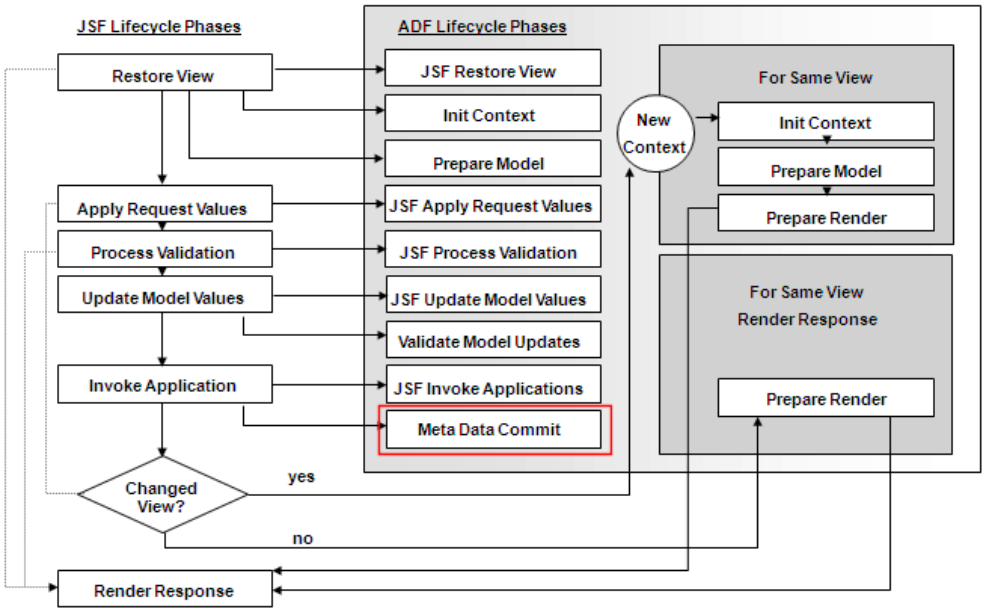

Figure 8: Metadata commit operation in the ADF page request lifecycle

As shown in figure 8 above, the metadata changes are written to the MDS repository after the JSF Invoke Application phase.

Note: When adding a component customization from Java, which we explain in the "How-to apply component changes in Java", you don't need to wait for a subsequent request to persist a change to MDS.

How-to manually configure components for MDS persistence

The "View" category in the adf-config.xml editor that is shown in figure 7 lists components that are configurable to use the `ChangeManager` API of the change persistence framework. The editor lists only components and attributes that are modifiable through the UI, for example when the user uses the mouse to rearrange the size of position of a component or component area.

However, only because attributes are not listed to be modified through the UI doesn't mean they cannot be changed and those alterations cannot be persisted in MDS.

Later in this paper we illustrate explicit runtime customization in Java using the example of disabling the column reorder functionality of a table.

This is accomplished by setting the "disableColumnReordering" attribute to true. In the example we use the `AttributeComponentChange` class to persist the attribute state change – true or false – to the session. To persist this change into MDS, in addition to persisting it to the user session, you must manually configure the adf-config.xml file as shown below.

```
<tag name="table">
…
</attribute>
  <attribute name="disableColumnReordering">
    <persist-changes>
      true
    </persist-changes>
  </attribute>
</tag>
```

This configuration in adf-config.xml can be created for all components that you intend to change attribute state for programmatically. The syntax for this metadata is always the same and uses the tag element to specify the component, followed by the one or more attribute elements for which attribute states should be saved.

Note: To enable all attributes of a component as candidates for MDS persistence, an asterisk '*' can be used as a wildcard attribute name in the component tag configuration. This way you don't need to list all possible attributes explicitly.

Another example, explained later in this paper, when manual configuration of the adf-config.xml file is required is when child components are re-ordered, added or removed.

For example, if child components of an af:panelFormLayout component are changed, for example a new field added, using drag and drop, then the following configuration must be added to the to the taglib-config | taglib section of the adf-config.xml file

```
<tag name="panelFormLayout">
  <persist-operations>
    all
  </persist-operations>
  … optionally add attributes to persist …
</tag>
```

Identifying the user

If changes should be persisted beyond session scope, then MDS needs to know about the user identity to persist the changes for. This requires Java EE authentication and the `UserCC` customization layer to be configured for an application.

To implement Java EE authentication, you can configure ADF Security to enforce authentication for an application. To use ADF Security, select Application | Secure | Configure ADF Security from the JDeveloper menu. In the first wizard screen, select the authentication only option and continue, accepting all the default settings. You can later re-run this wizard to enable

authorization or change the authentication from the default basic authentication to form based or other.

Choose Application | Secure | Users from the Oracle JDeveloper menu to create user accounts to use during application testing with MDS.

Note: Please also read "Enabling ADF Security in a Fusion Web Application" of the Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11g Release 1, available online[1].

The next step is to configure a customization layer that persists the component changes for a user. If you run the MDS enabled application without configuring a customization layer the following error message is displayed in the log when you perform a component change.

<MDSDocumentChangeManager><getDocument> ADFv: Trouble getting the mutable document from MDS. oracle.mds.exception.ReadOnlyStoreException: MDS-01273: The operation on the resource /<page name> failed because source metadata store mapped to the namespace / BASE DEFAULT is read only.

As explained in the customization flow shown in figure 6, if MDS change persistence fails, the change is only saved for the user session. Without a valid customization layer configured in the adf-config.xml file, the metadata store is read only and no changes can be persisted – hence the log message.

UserCC customization class

Customization layers are represented by Java objects, the customization classes, which determine when a specific customization should be applied at runtime. One of the default customization class provided by ADF is `UserCC`, which identifies the Java EE authenticated user to MDS.

To configure `UserCC` for an application, open the adf-config.xml file in Oracle JDeveloper and select the "MDS Configuration" category. Click the green plus icon to search the class path for the `UserCC` class, as shown in the image below. The username is accessed from the ADF security context, which reads the name from the security principal of the authenticated session. Though the `UserCC` layer accesses the ADF security context, it is not mandatory to use ADF Security for user authentication and plain container managed security configured in the web.xml file of a web application is sufficient.

[1] http://download.oracle.com/docs/cd/E15523_01/dev.htm

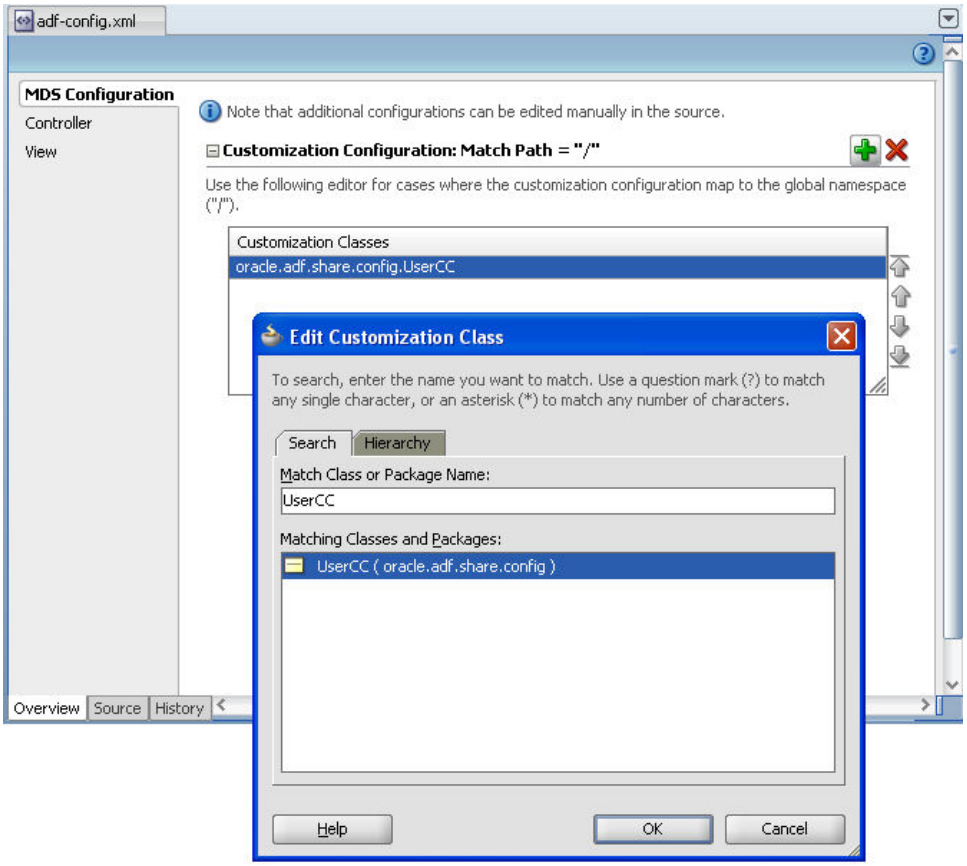

Figure 9: Configuring UserCC customization class for an application

Searching the view layer project class path, you'll find more predefined customization classes provided by ADF.

Note: The `AdfUserCC` and `SiteUserCC` classes are deprecated and should no longer be used in custom applications.

Note: Creating and configuring your own MDS customization is simple and is covered later in this paper. For example, applications that handle user authentication within the program code, without using Java EE container authentication, may extend the `UserCC` class and override the *getValue* method to return the user name authenticated by their application login handling routine.

The example we'll show later in this paper to explain customization class development illustrates the use case of switching between power user and normal user modes for an application.

Configuring change persistence for a single component instance

Using MDS, change persistence that is saved across the individual user session can be configured for single component uses in a view, overriding the global component type definition in the adf-config.xml file. This provides developers with fine grained control over changes saved in MDS. To do this, developers set a value to the *persist* and *dontPersist* properties available on many ADF Faces components. Allowed values are space delimited names of component attributes that hold a component state, as well as the keyword "all" to persist or don't persist all attributes.

How-to apply component changes in Java

So far, this paper has covered declarative user customization, which allows developers to implicitly save component changes performed by the application user. In the following, we explain how application developers can apply and persist component changes from Java using the `ChangeManager` API that is implemented by the `MDSDocumentChangeManager`, the `FilteredPersistenceChangeManager` and the `SessionChangeManager` classes introduced earlier.

The `ChangeManager` change API is accessible from the `AdfFacesContext` object as shown below

```
AdfFacesContext adfFacesContext = null;
adfFacesContext = AdfFacesContext.getCurrentInstance();
ChamgeManager cm = adfFacesContext.getChangeManager();
```

Using the `ChangeManager` instance, component changes can be applied during any phase of the JavaServer Faces lifecycle in which the server side in memory component tree exists, i.e. after the JSF RestoreView phase.

How-to persist attribute changes in Java

In JavaServer Faces, component properties are defined by developers entering static values to the page source, dynamically using Expression Language, or even by using Java.

Programmatic changes to component properties are temporary change and are reset when the component is removed from the JSF server side memory tree. This means that when the user navigates away from the current view, the change will be forgotten when they return. For a property change to survive navigation, a further step is required, in which the `ChangeManager` API is called to stored the change in the user session or, as we discuss later across application restarts in MDS.

For example, the following menu item calls a managed bean to enable or disable the column reordering functionality of a table.

```
<af:menuBar id="mb1">
  <af:menu text="User Preferences" id="m2">
```

```
  <af:commandMenuItem text="Toggle Column Reodering" id="cmi3"
    actionListener="#{allEmployeesBean.onToggleColumnOrdering}"/>
  </af:menu>
</af:menuBar>
```

The `onToggleColumnOrdering` managed bean method is shown below. The current reorder behavior is read from the table component and then reversed.

The persistAttributeChange helper method shown here is generic and can be used in your custom application developments as well.

```
public void onToggleColumnOrdering(ActionEvent event){
  //get access to the table instance handle, which is created
  //in the managed bean using the component "binding" property
  RichTable table = getEmployeeTableInstance();
  boolean reorderingDisabledVal =
   (Boolean)table.getAttributes().get("disableColumnReordering");
  //toggle component attribute value
  table.getAttributes().put("disableColumnReordering",
                            !reorderingDisabledVal);
  //persist the change for the duration of the user session or
  //beyond using MDS
  this.persistAttributeChange("disableColumnReordering",
                              table, !reorderingDisabledVal)
}

//private helper method to persist attribute changes
private void persistAttributeChange(String attribute,
                                    UIComponent component,
                                    Object value){
  AttributeComponentChange attributeChange = null;
  attributeChange =
       new AttributeComponentChange(attribute, value);
  FacesContext facesCtx = FacesContext.getCurrentInstance();
  AdfFacesContext adfFacesCtx = null;
  adfFacesCtx = AdfFacesContext.getCurrentInstance();
  //ChangeManager is of type oracle.adf.view.rich.change.
  //ChangeManager.
  ChangeManager manager = adfFacesCtx.getChangeManager();
  manager.addComponentChange(facesCtx,
                             component, attributeChange);
}
```

As you can see from the code lines above, the use of the `ChangeManager` is independent of the type of persistence – session or MDS – that is used.

To configure MDS for this change, edit the adf-config.xml file using the Oracle JDeveloper source view and add the following content

```
<tag name="table">
…
</attribute>
  <attribute name="disableColumnReordering">
    <persist-changes>
      true
    </persist-changes>
  </attribute>
</tag>
```

Note: The `oracle.adf.view.rich.change.ChangeManager` class is deprecated and the Apache MyFaces Trinidad `ChangeManager` class should be used instead. However, due to a class incompatibility this is not yet possible. Until a fix is provided, please continue using the deprecated Oracle `ChangeManager`.

Note: The Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11g product documentation provides an additional example that shows how to save a component's disclosure state.

How-to persist re-ordering of child components in Java

Another popular personalization use case is to change the order in which components appear on a view. For example, an application may allow users to change the order of input components in an af:panelFormLayout using drag and drop.

The following example code shows the page and managed bean source of an input form that allows users to move af:inputText components within the af:panelFormLayout, as shown in figure 10. These changes are then persisted.

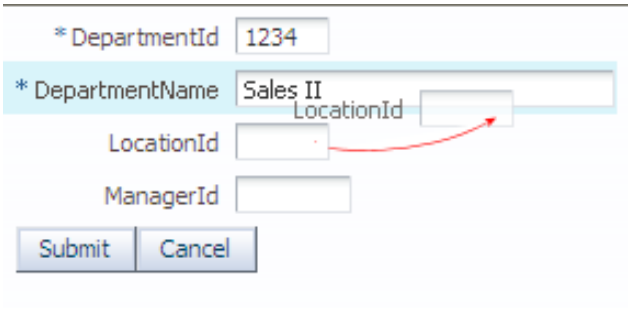

Figure 10: Changing and persisting the location change of an input text component using drag and drop and the `ReorderChildrenComponentChange` class

To enable drag and drop on ADF Faces components, you use operation tags provided in the ADF Faces component library. The example below defines each inputText component as a drag source and drop target, which means that each component can be moved within the form. If a component is dropped onto another, then the drop component's drop listener is invoked, which calls a managed bean method, "handleInputTextMove" in the example.

Note: to learn more about drag and drop in Oracle ADF Faces applications, please refer to the "Adding Drag and Drop Functionality" section of the Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework

```
<af:panelFormLayout id="pfl1">
  <af:inputText value="#{bindings.DepartmentId.inputValue}"
                ...>
    <f:validator .../>
    <af:convertNumber .../>
    <af:componentDragSource/>
    <af:dropTarget
       dropListener="#{allEmployeesBean.handleInputTextMove}"
       actions="MOVE">
         <af:dataFlavor
             flavorClass="javax.faces.component.UIComponent"/>
    </af:dropTarget>
  </af:inputText>
   ... more input components ...
</af:panelFormLayout>
```

The managed bean method that is invoked by the dropTarget uses the drop event to determine the dragged component and the drop component to change the component order in the form layout. The changes are then persisted in MDS using the `ChangeManager` API.

```
public DnDAction handleInputTextMove(DropEvent dropEvent) {
  FacesContext fctx = FacesContext.getCurrentInstance();
  //access the payload of the drag and drop operation
  Transferable transferable = dropEvent.getTransferable();
```

```
 //get the UI component that is moved in the layout container
 UIComponent dragComponent =
       transferable.getData(DataFlavor.UICOMPONENT_FLAVOR);
 //get the component that received the drop. The dragged
 //component will be placed right below this component
 UIComponent dropComponent = dropEvent.getDropComponent();
 //the parent layout component af:panelFormLayout
 UIComponent dropParent = dropComponent.getParent();
 if (dragComponent != null && dragComponent != dropComponent) {
   //get the number of input components in the panelFormLayout
   int childCount = dropParent.getChildCount();
   //create a list to hold the component Ids in the new order
   List<String> newComponentList =
                new ArrayList<String>(childCount);
   List<UIComponent> inputfields = dropParent.getChildren();
   //iterate over all panelFormLayout children and compare the
   //child with the drag and the drop component to create the
   //new component order
   for (UIComponent currComponent : inputfields) {
     String currComponentId = currComponent.getId();
     if (!currComponentId.equals(dragComponent.getId())) {
       newComponentList.add(currComponentId);
         if (currComponentId.equals(dropComponent.getId())) {
           newComponentList.add(dragComponent.getId());
       }
    }
  }
  //Persist the changed component order
  ReorderChildrenComponentChange reorderedChildChange =
         new ReorderChildrenComponentChange(newComponentList);
  reorderedChildChange.changeComponent(dropParent);
  AdfFacesContext adfFacesCtx =
                     AdfFacesContext.getCurrentInstance();
  ChangeManager cm =  adfFacesCtx.getChangeManager();
  cm.addComponentChange(fctx, dropParent,
                        reorderedChildChange);
  //Tell the parent panelForm to re-draw
  adfFacesCtx.addPartialTarget(dropParent);
  return DnDAction.MOVE;
}
```

```
 return DnDAction.NONE;
}
```

Note: If you use MDS to persist component changes, ensure that the adf-config.xml file is configured to allow customization of the parent container component; af:panelFormLayout in the example. For example, to allow component changes in MDS, the following entry needs to be added to the taglib-config | taglib section of the adf-config.xml file for the example above

```
<tag name="panelFormLayout">
  <persist-operations>
    all
  </persist-operations>
  … optionally add attributes to persist …
</tag>
```

More Component Change APIs

In the Java API examples above, the following two APIs were used: AttributeComponentChange and ReorderChildrenComponentChange. There are more component change classes developers can use to persist user customization settings in MDS, all which are documented online in the "Allowing User Customizations at Runtime" section of the Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework product documentation. Additional options include:

- AddChildDocumentChange
- AttributeDocumentChange
- MoveChildComponentChange
- RemoveChildComponentChange
- SetFacetChildComponentChange
- SetFacetChildDocumentChange
- RemoveFacetComponentChange

How to control personalization through authorization

As explained, many ADF Faces components expose the *persist* and *dontPersist* property for developers to define a blank character delimited list of attributes to include in or exclude from personalization on the component instance level. The component level definition always overrides the configuration defined in the adf-config.xml file. The *persist* and *dontPersist* properties

do not, however, handle the case in where a defined group of users should be allowed to persist component changes whereas another group is not.

For authorizing customization on the component level, the "Customization allowed by" property can be set to a username or security role name, as shown in figure 11. The information about component instance level customization authorization is not stored within the page or view source but in the view layer project's mdsys | mdx folder. The folder is automatically created by Oracle JDeveloper when using one of the customization properties exposed for a component. The component change shown in the image below is configured in the mdx folder as shown next:

```
<rdf:Description rdf:about="t1">
  <customizationAllowed xmlns="http://xmlns.oracle.com/mds">
    true
  </customizationAllowed>
  <customizationAllowedBy xmlns="http://xmlns.oracle.com/mds">
   app_manager
  </customizationAllowedBy>
</rdf:Description>
```

Note that "t1" in the example code above is the Id property of the table configured in this example. The role name used can be the name of a Java EE security group, which we refer to as an Enterprise group, or an application role. Application roles are a specific semantic concept of the Oracle Platform Security Service (OPSS) that we talk a bit more about later in this paper.

Note: If the default role based component instance level authorization is not what you want, then you can write your own implementation of `ChangeManager` to exclude user groups from customization on a global level, for example using JAAS permissions and ADF Security. The `FilteredPersistenceChangeManager` class and the `MDSDocumentChangeManager` classes however are both marked as final and cannot be subclassed, so the starting point for any such custom development is the `SessionChangeManager` class in Apache Trinidad.

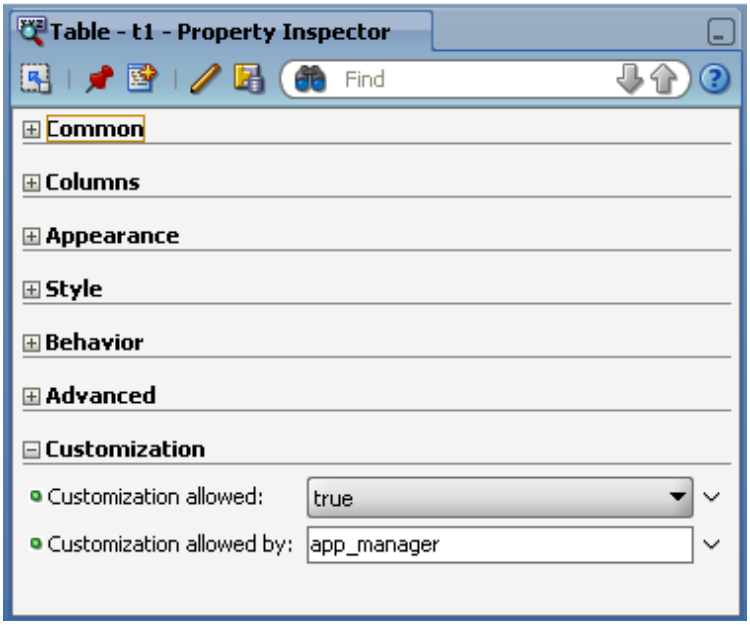

Figure 11: Customization authorization on the component level

What you should know about regions or templates

The ChangeManager saves and restores component changes in the context of the current view Id. A change that is applied for a component that resides in a region or page template, then this change is persisted only for the region or template instance on the current parent page or view. It does not affect any uses of the same region or page template in other pages or views.

Seeded Customization with MDS

Seeded customization allows developers to pre-define application settings for individuals or groups of users. For example, the salary attribute of an employee table might only be shown to managers, for all other roles it is completely removed from the page, something which is more secure than just trying to hide the data in the UI layer.

While user customization is created at runtime and applied at runtime, seeded customization is pre-defined during application development and dynamically applied to a document request at runtime. Customization includes changes on the UI layer, the controller layer, the binding and the business services layer, if ADF Business Components is being used.

Understanding MDS Configuration in Oracle JDeveloper

Seeded customization is enabled in the ADF view setting of the view layer project properties by checking the "Enable Seeded Customizations" checkbox in the project properties.

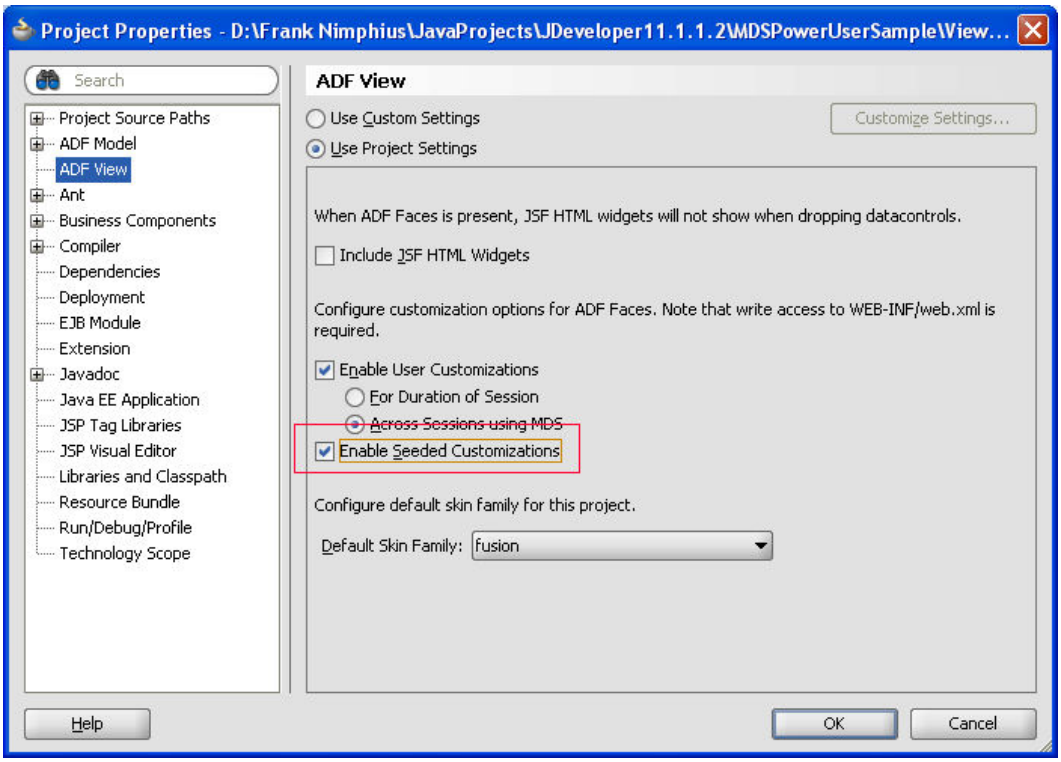

Figure 12: Enabling seeded customization in the view layer project properties

In addition, the Oracle JDeveloper design time needs to know about the customization layers and their possible values to display when running Oracle JDeveloper with the Customization Developer role.

The CustomizationLayerValues.xml configuration file

At design time, MDS customization layers are represented by a physical object, the customization class, and an entry in an Oracle JDeveloper wide configuration file that defines the customizable layers and their values. The configuration file is CustomizationLayerValues.xml and is located in the <JDeveloper WLS installation home>\jdeveloper\jdev directory. A valid layer configuration entry in the CustomizationLayerValues.xml file looks as shown below

```
<cust-layers  xmlns="http://xmlns.oracle.com/mds/dt">
  <cust-layer
        name="…"
        value-set-size="…"
```

```
        id-prefix="…">
    <cust-layer-value
        value="…"
        display-name="…"
        id-prefix="…">
     </cust-layer-value>
    …
  </cust-layer>
  …
</cust-layers>
```

The following table explains the individual configuration elements and attributes used in the CustomizationLayerValues.xml file

ELEMENTS AND ATTRIBUTES	MEANING OF VALUE
cust-layer	One to many entries that define the allowed customization layers for which a customization class exist.
name	The name of the layer as defined in the customization class. The UserCC customization class for example has a name defined as “user” which also must be the name of the configured customization layer when used with seeded customization
value-set-size	Allowed values are “no_values”,”small”, “large”. Setting the value to “no_values” hides the layer from the design time in which case it is used at runtime only. A value of “small” displays the list of allowed layer values in Oracle JDeveloper in a single select list, which is good to use for a smaller value sets. Using “large” as the size information renders the allowed layer values in a LOV
id-prefix	Optionally, but useful, MDS allows to specify an id for the layer itself that will be added as a prefix to the customization layer id. Setting an id-prefix for the layer helps avoiding conflicts that may occur for cust-layer-value definitions that use the same id prefix.
cust-layer-value	Defines one or more allowed values for a customization layer. The values are displayed in Oracle JDeveloper after selecting the customization layer and allow developers to define metadata changes for a specific group of users or site of installation
value	A value returned by the getValue method of the customization class. The UserCC class for example returns authenticated user names, whereas the AdfRoleCC class returns the roles the authenticated user is a member of.
display-name	A user friendly name that is shown in the Oracle JDeveloper IDE when working in the Customization Developer role
id-prefix	A unique string in the context of the customization layer that is used as the namespace for the customization metadata created for the cust-layer-value

Table 1: CustomizationLayerValues.xml configuration elements

Note: The CustomizationLayerValues.xml doesn't belong to the application but is an IDE wide artifact. This means that it will not be part of the source code tree for a particular workspace and therefore will not be under the same version control as the application. To version customization layer information with the application, we suggest to provide application specific customization layer definitions in a text file in a custom folder within the view layer project. For runtime and deployment this file is not required.

Default Customization Layers provided by ADF

As mentioned earlier, customization layers are identified by customization classes at runtime. Or to put this differently, every customization layer defined in CustomizationLayerValues.xml has a physical object representation at runtime that defines which customization to apply. JDeveloper contains a set of default customization classes that include the `UserCC`, `AdfRoleCC` and `SiteCC` objects.

UserCC

As we have seen, the `UserCC` customization class is used to identify user session for personalization. However, it can also be used for seeded user customization, although this is less common. The `UserCC` class reads the user name from the authenticated user principal class in the Java EE session. If the user information is stored in another location, like RDBMS, LDAP or a JSF managed bean, then developers can extend `UserCC`, overriding the *GetValue* method, or write a complete custom implementation of the "user" layer.

To enable the `UserCC` layer for seeded customization, edit the CustomizationLayerValues.xml configuration file and add an entry for the customization layer and its values as shown below. Note that "user" is the name returned by the `UserCC` customization class's *getName* method.

```
<cust-layer name="user" value-set-size="small" id-prefix="user_">
   <cust-layer-value value="sking"
               display-name="Steven King" id-prefix="sk"/>
    <cust-layer-value value="ahunold"
               display-name="A. Hunold" id-prefix="ah"/>
</cust-layer>
```

Opening Oracle JDeveloper in Customization Developer role shows an entry "user" for the customization layer, as well as "Steven King" and "A. Hunold" as the two options to provide seeded customization for. All changes that are configured for the user "Steven King" are stored in MDS using the "user_sk" namespace prefix, whereas document changes defined for "A.Hunold" have "user_ah" added as a prefix.

AdfRolesCC

The `AdfRolesCC` class reads user role memberships from the ADF security context. Security roles that are defined on the Java EE container are referred to as enterprise roles whereas roles defined in the jazn-data.xml configuration file are referred to as application roles. The difference between the two types of roles is their scope. Enterprise roles are Java EE container wide roles that can be accessed and used by all applications deployed on a server. Application roles are specific for a single application and are mapped to enterprise roles upon deployment. Application roles are used for fine grained authorization by the Oracle Platform Security Services (OPSS) and ADF Security.

The enterprise role membership of a user is determined by the identity management system that is accessed by the authentication provider handling the user login in Oracle Weblogic Server (WLS). Application roles have a narrow scope as they are defined for a single application only. They are mapped to users and/or enterprise roles that are available in the identity management system. In the example configuration below, the application roles are defined as employees and managers.

You configure the `AdfRolesCC` layer in the CustomizationLayerValues.xml configuration file as shown below.

```
<cust-layer name="role" value-set-size="large" id-prefix="role_">
  <cust-layer-value value="employees"
            display-name="Sales Employees" id-prefix="emp" />
  <cust-layer-value value="managers"
            display-name="Sales Managers" id-prefix="man" />
</cust-layer>
```

Note: The `AdfRoleCC` layer name is defined as "role" in the customization class.

SiteCC

The `SiteCC` layer returns a single value "site" and allows developers to customize an application for a point of installation. To make `SiteCC` configurable, you can override the getValue method in a subclass. For example, in the Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11g documentation, an example is provided in which the site information is read from a properties file that allows developers and administrators to easily change between the configured site. Other custom implementations may read site information from a database or the application context.

```
<cust-layer name="site" value-set-size="large">
  <cust-layer-value value="site"
     display-name="Default Site" id-prefix="s" />
</cust-layer>
```

Note: An interesting custom implementation of `SiteCC` that also shows how to integrate ADF Security in MDS customization layers is documented in the "How-to read URL request parameters" section later in this paper.

Building your own Customization Layer

Developers can build their own customization layers to shape the application runtime. To build a custom layer, the following steps are required:

- A customization class needs to be built that defines the customization layer name, as well as the values used to identify the change metadata to apply to a document.
- Non default customization classes need to be deployed as an Oracle JDeveloper extension into the <JDeveloper WLS installation home>\jdeveloper\jdev\lib\patches directory so they become accessible for the IDE when started in customization role
- The customization layer must be configured in the CustomizationLayerValues.xml file

How to develop and deploy custom customization classes

Customization classes are not usually application specific and so should be built in their own Oracle JDeveloper project. When creating a new customization class project, configure the project libraries to include the "MDS Runtime" and the "MDS Runtime Dependencies" libraries.

Customization classes extend the abstract `oracle.mds.cust.CustomizationClass` or an existing implementation of it, such as `UserCC`. They must provide an empty constructor, be thread safe and should be tuned as they are called for each user request. Customization classes that require access to external resources, such as LDAP or RDBMS serves, should release the resource handle at the end of a call and where possible cache values queried from the resource to reduce the need for repeated access to the external resource. Furthermore, the cache hint value returned by a customization class should make MDS to keep customization definitions in memory for as long as possible.

Note: The `CustomizationClass` abstract class has more public methods developers can override. For ADF application development and deployment purposes however we don't yet see a need for developers to change the default behavior of these.

When starting your own customization class development, you will need to provide implementations for the *getName* , *getCacheHint* and *getValue* methods at the very least:

getName

A customization is identified by the name and the possible values of the customization layer. The *getName* method defines the name for a customization layer, for example "user" for the `UserCC` layer that is used by developers in Oracle JDeveloper to select and customize the layer.

```
public String getName() {
  return "powerUserCC";
}
```

Note: The name value returned by the customization class must match one of the cust-layer names when defined in the CustomizationLayerValues.xml file.

getCacheHint

The cache hint information provides information about how long a change definition read from the MDS repository should be kept in memory and whether, or not, internally it should be cached for multiple users. The cache hint returned by a customization class can have one of the following values

- ALL_USERS – The customization is applied to all users of an application and therefore needs to stay in the cache much longer than other definitions
- MULTI_USER – The customization is applied to a group of users across application sessions
- USER – The customization is applied to an individual application user session, which is defined as the HTTP servlet session for web based application.
- REQUEST – The customization is for a single request only, it does not need to remain in memory

```
public CacheHint getCacheHint() {
  return CacheHint.ALL_USERS;
}
```

getValue

The *getValue* method determines the selected value for a customization layer. Because document changes in seeded customization are defined by a customization layer / value pair, the value(s) returned by the *getValue* method is the part that influences the visual appearance and behavior of an application.

The *getValue* method has two arguments, `RestrictedSession` and `MetadataObject`. The `RestrictedSession` represents a handle to the MDS session and provides developers with information about the MDS context and configuration through the `SessionOptions` object that it returns.

Note: Information like the user locale is directly exposed on the MDS `RestrictedSession` object. The direct access however is deprecated and will be removed in a future version. Use the `SessionOptions` object instead to access this type of information.

The `MetadataObject` argument provides developers with a handle to the top level metadata document and its element hierarchy exposed as JAXB compliant bean objects. The use of this object is an advanced concept that goes beyond the scope of this whitepaper.

The value returned by the *getValue* method is an array of `String` values. Each string in the array represents a customizable layer value for which developers can define seeded customization in Oracle JDeveloper. If the *getValue* method returns more than one value, then all of the defined customization changes are applied to the requested document in the order in which the values appear in the array. The document change definition of the first value in the array is applied to the base document, followed by the change definitions of the second value, and so on. The order of the values in the array is important as it determines which layer value wins in cases where two layer values lead to modification of the same metadata object.

Customization class template

Provided below is a skeleton template that you can build on for your own customization classes

```
import oracle.mds.core.MetadataObject;
import oracle.mds.core.RestrictedSession;
import oracle.mds.cust.CacheHint;
import oracle.mds.cust.CustomizationClass;

public class MyCustLayerCC extends CustomizationClass{
  public MyCustLayerCC () {
    super();
  }

  public CacheHint getCacheHint(){
    return CacheHint.<YOUR CHOICE>;
  }
  public String getName() {
    return "mycustlayer";
  }
  public String[] getValue(
     RestrictedSession restrictedSession,
     MetadataObject metadataObject) {
        //determine an array of return string, each representing
        //a customization layer value.
        return new String[0];

  }
}
```

How to configure custom Customization Layers in Oracle JDeveloper

To use a custom customization class in Oracle JDeveloper, you need to deploy it in a JAR file to the <JDeveloper WLS installation home>\jdeveloper\jdev\lib\patches directory. The custom layer class needs to be configured in the adf-config.xml file, which is accessible in Oracle JDeveloper from the Application Navigator. To edit this file, select Application Resources | Descriptors | ADF META-INF and double click onto the adf-config.xml file entry. Use the MDS Configuration panel and press the green plus icon to search and select the customization class, as shown in figure 13 below:

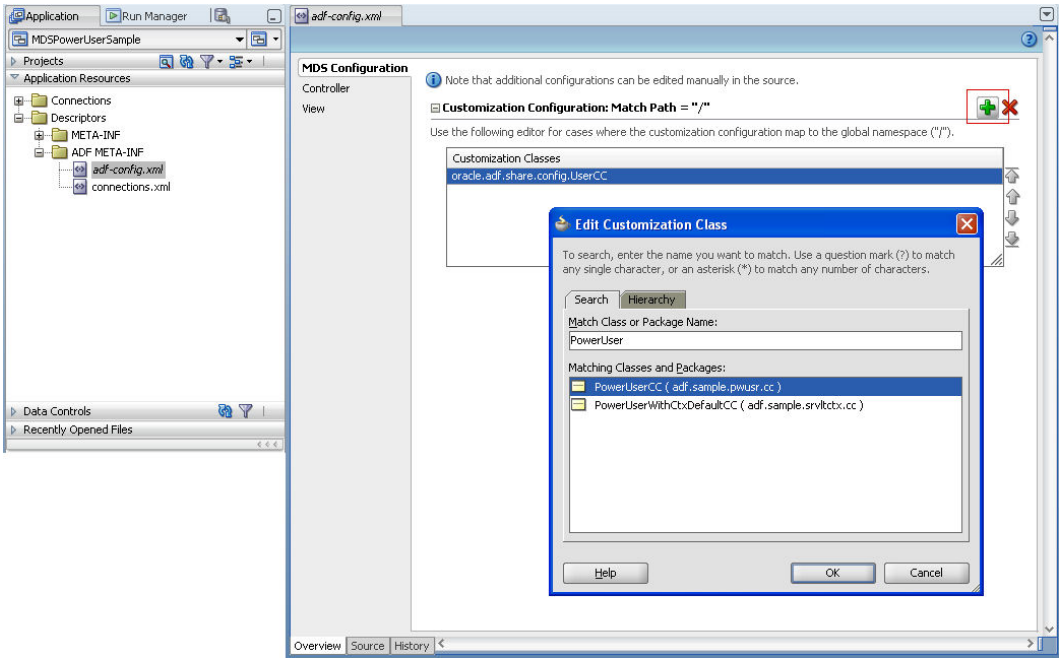

Figure 13: Searching or customization classes in the class path

After this, configure the customization layer name returned by the customization class in the CustomizationLayerValues.xml. In the case of the `PowerUserCC` customization class example shown in the image above, the layer name is "powerUserCC". If a user is a power user, then the customization class *getValue* method returns "poweruser" as a value, which can be used to define seeded MDS customizations configuration in the CustomizationLayerValues.xml, which is located in the <JDeveloper WLS installation home>\jdeveloper\jdev directory.

```
<cust-layer name="powerUserCC"
            value-set-size="small" id-prefix="pwusr">
<cust-layer-value
            value="poweruser" display-name="Power User"
```

```
            id-prefix="pu" />
</cust-layer>
```

Defining Seeded Customization for ADF Applications

In this section, we continue with the power user customization layer example that will be used to configure the application to display an advanced – dialog, LOV and wizard free, user interface to users that run an application in power user mode.

To define seeded customization for an application, start Oracle JDeveloper in the Customization Developer role to shape the IDE to only allow changes to customizable metadata. All other files, such as Java classes or project properties, are read only and cannot be changed in this mode.

For all document changes, Oracle JDeveloper creates a change document for the selected customization layer that contains MDS change directives for the modifications made by the application developer. At runtime, the change document based on the application MDS configuration and the value evaluation of the customization class, is merged with the base document.

Starting Oracle JDeveloper in Customization Developer Role

To start Oracle JDeveloper in Customization Developer role select this option in the "Select Role" dialog that shows by default when starting the IDE.

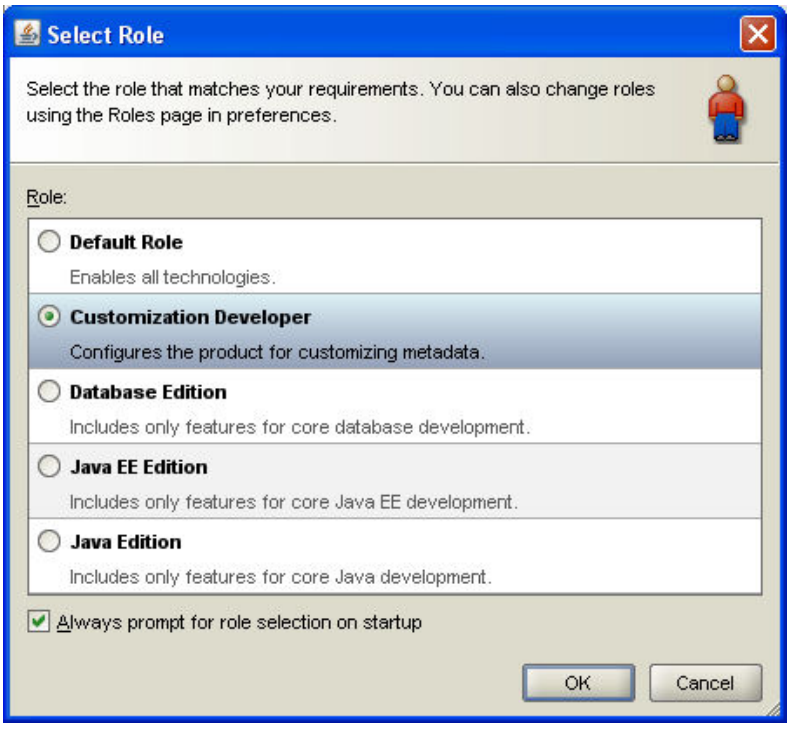

Figure 14: Select Role dialog displayed when starting Oracle JDeveloper

Another option to start Oracle JDeveloper in Customization Developer role is to use the **Roles** option in the Oracle JDeveloper **Preferences** dialog that you open from the Tools | Preferences menu option. This dialog also allows you to re-enable the default JDeveloper behavior to always show the role selection dialog at IDE startup.

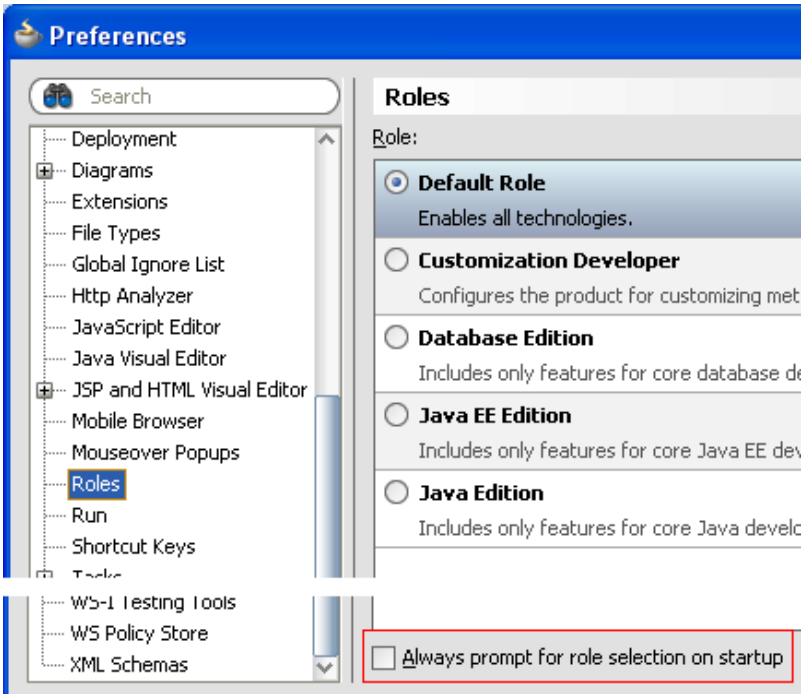

Figure 15: Selecting the Customization Developer role in the Oracle JDeveloper Preferences dialog

Starting Oracle JDeveloper in Customization Developer role shows an additional panel to select a customization layer and value from options defined in the CustomizationLayerValues.xml. Seeded customization metadata is then created based for that particular combination.

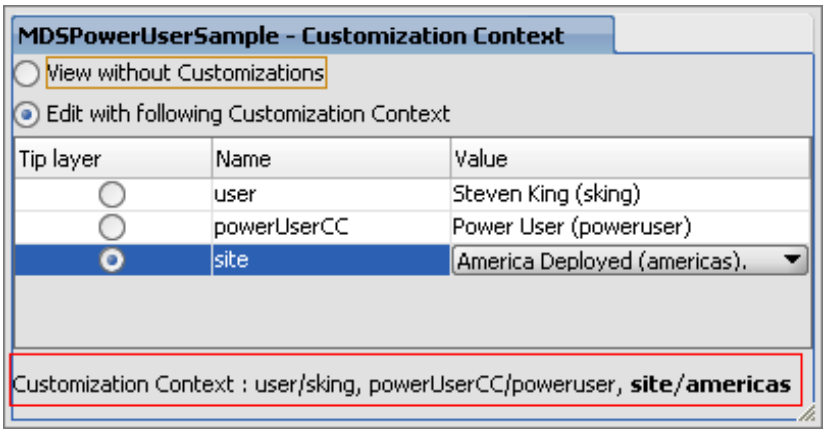

Figure 16: Customization layer selection window in Customization Developer role

In figure 16 above, customizations are created for the "America Deployed" value of the site customization layer, a custom implementation of the default `SiteCC` class, shown here:

```
public class DynamicSiteCC extends SiteCC{
  public DynamicSiteCC() {
    super();
  }
  @Override
  public String[] getValue(RestrictedSession restrictedSession,
                           MetadataObject metadataObject) {
      String[] _values = null;
      //determine site values …
      //americas, emea, apac
      return _values;
  }
}
```

Note: Later in this paper we come back to the `DynamicSiteCC` class and how to determine its site value

The Tip Layer selection shown in figure 16 defines the layer which has "design focus" in the IDE and for which new MDS metadata is created. However, customizations configured on other layers, for example the "powserUserCC" layer are active as well and display in the Oracle JDeveloper visual design editor when customizing the site layer. This behavior at design time mirrors the runtime behavior of MDS in which the resulting document is the merger of the base document with the tip layer and all other layers between. It also allows developers to override customizations performed in one of the other layers if they are not desired in the context of a specific tip layer value.

The "CustomizationContext" message outlined in figure 16 provides developers with the information about which customization layers and values contribute to the current document view. In this case, the resulting document is the outcome of the base document merged with the user layer set to the "sking", the powerUserLayer set "poweruser" and the site layer set to "America Deployed".

Note: When changing from the "Default Role" to the "Customization Developer" role in Oracle JDeveloper, pay extra attention to the selected workspace after the Oracle JDeveloper restart to avoid accidentally editing the wrong base application. It is probably a good idea to start the IDE with the –noreopen flag to ensure that the editor area is initially empty and does not contain open files from the wrong workspace.

Customizing Pages

When running in Customization Developer role, the Oracle JDeveloper IDE looks the same as in Default Role, which is used to build the base application. The only visual differences are a lock

overlay icon shown next to base application documents that are not customizable (such as Java source files) and the appearance of the Customization Layer Selection panel.

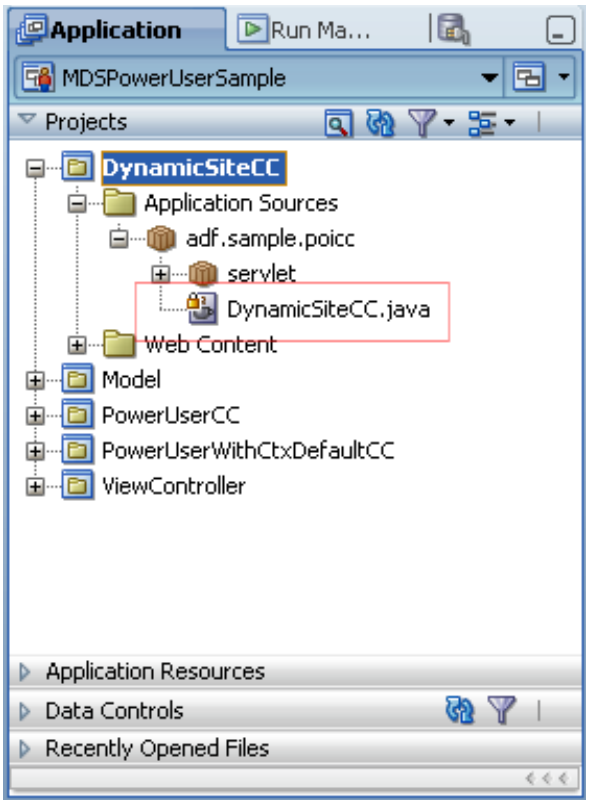

Figure 17: Overlay icons in Customization Developer role view

Note: MDS customizes metadata information only. To customize Java Objects like managed beans, create different versions of the Java Objects in the base application and change the bean reference in the task flow metadata definition in Customization Developer role.

MDS change documents are stored under a specific "mdssys" folder that is created for each customization layer and value combination. As developers create seeded customizations for the base document the metadata changes are stored in this directory tree in the form of XML "diff" files that contain the name of the changed base document.

To edit a JSF page document for a customization layer, double click the document so it opens in the visual page editor. The document can then be edited in the normal way, for example, you can use the component palette to add ADF Faces components to a document.

In the following example we show the change document created for the allEmployees.jspx page when the power user customization (powerUserCC) layer value is configured for "poweruser"

(See figure 18). In this case, all wizards and dialogs are replaced with input fields for faster data entry.

```
<mds:customization version="11.1.1.55.36"
                   xmlns:mds="http://xmlns.oracle.com/mds">
  <!-- Replace af:selectOneChoice in table filter with
       af:inputText -->
  <mds:replace node="soc4"/>
  <mds:insert parent="c3(xmlns(f=http://java.sun.com/jsf/core))/
              f:facet[@name='filter']"
              position="last"
              xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
    <af:inputText
              xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
              id="soc4"
              value="#{vs.filterCriteria.ManagerId}"
              label="Label 1"/>
    </mds:insert>
  <mds:replace node="soc3"/>
    <mds:insert
      parent="c2(xmlns(f=http://java.sun.com/jsf/core))
              /f:facet[@name='filter']"
      position="last"
      xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
    <af:inputText
       xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
       id="soc3"
       value="#{vs.filterCriteria.DepartmentId}"
       label="Label 2"/>
  </mds:insert>
  <!-- Delete af:commandButton that launches dialog to edit user
       data in bounded task flow -->
  <mds:replace node="cb2"/>
  <!-- Show additional af:inputText fields for the power user to
       edit employee data -->
  <mds:modify element="it3">
    <mds:attribute name="rendered" value="true"/>
  </mds:modify>
    …
</mds:customization>
```

Note: Explicitly closing and re-opening the base document forces a refresh of the document display in Oracle JDeveloper. Though it is not recommended to manually edit the MDS change documents located in the mdssys folders, if you do, then closing and then reopening the base document will apply and display the changes in the visual editor.

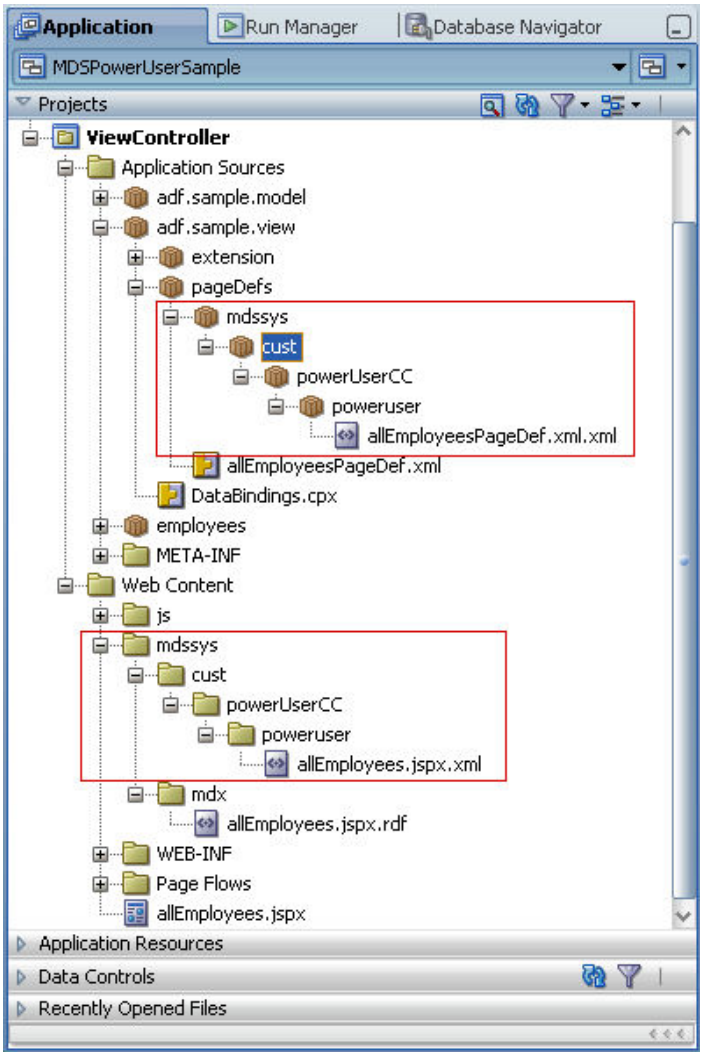

Figure 18: Application Navigator in Customization Developer role showing MDS change documents for an ADF binding definition and JSF document

Note: MDS change documents that are created when customizing an application cannot be deleted within the Customization Developer role. To delete a customization document that is no longer required, switch to the Default Role and delete it from the Application Navigator.

Testing customized ADF applications

For the purposes of this paper, testing and deployment is with and to the integrated Weblogic server (WLS) in Oracle JDeveloper 11g.

To test MDS customizations from Oracle JDeveloper 11g, select a page document and choose the run icon from the menu or context menu. Implicitly this creates all the required deployment artifacts to deploy and run the application. Like any other web application you run from Oracle JDeveloper, you can do so both in debug and non-debug mode.

For the integrated WLS to find custom customization classes, they must be deployed with the application. To ensure this, either reference the customization classes in a JAR file from the web layer project or add the customization class project to the list of dependencies. If you add JAR files configured as an Oracle JDeveloper library, make sure to select the **Deployed by Default** check box when creating the new library.

Common MDS Customization Development Requirements

The more flexible a customization should be in for a particular ADF application, the more likely it is that customization classes need to access the application or runtime context.

In the following example we illustrate how developers can access the current user session, web.xml file context parameters and URL request parameters all from within customization classes.

How-to read attributes from the http session

The HTTP Session is accessible in a customization class through the `ADFContext` class. The `ADFContext` class is located in a shared JAR file that is added to the classpath when the *MDS Runtime Dependencies Library* is configured for an Oracle JDeveloper project. To access the http session object in the customization code, you need to add the *Servlet Runtime* library to the project as well.

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import oracle.adf.share.ADFContext;
…
ADFContext adfCtx = ADFContext.getCurrent();
Object _request = adfCtx.getEnvironment().getRequest();
if (_request != null && _request instanceof HttpServletRequest) {
  HttpSession _session = null;
  _session = ((HttpServletRequest)_request).getSession(true);
```

```
   ...
}
```

This allows you to read information from the session to configure a customization layer, for example, in response to user preference settings.

How-to read context parameters defined in the web.xml file

Instead of reading initial customization layer settings from a properties file, as it is explained in the Oracle product documentation, you can also read them from context parameters in the web.xml file, as shown below

```
//accessing the MDS session
SessionOptions so = restrictedSession.getSessionOptions();
ServletContext servletContext = null;
servletContext = (ServletContext) so.getServletContextAsObject();
//read context parameter
Object powerUserServletContextFlag =
               servletContext.getInitParameter("powerUserCC");
```

The restrictedSession variable is of type `RestrictedSession` and is provided by default as an argument to the customization class's *getValue* method.

How-to read URL request parameters into a customization class

Using the `ADFContext` object, it is possible to access request parameters directly from the customization layer. In practice, however, this approach proves difficult to use because of the ADF Faces lifecycle.

The default behavior in ADF Faces is to lazily load data for better performance when rendering views. For this, multiple partial requests are submitted to the server to fetch data behind the scenes during page rendering. Only the first GET request in this sequence will have the request parameter added to the URL. Therefore developers need to make sure the value of the parameter is persisted so it survives subsequent partial requests used to populate the page.

Even if developers can ensure that subsequent requests do not override the parameter value defined on the initial request, for example by detecting the partial submit nature of a request, the value must be stored somewhere so it is available for read in the customization class. One option is to write the parameter value as a property into the MDS session. However, the MDS session only lives for the duration of the request, which means that subsequent requests will grab a new MDS session and not include this information. Therefore, a better option is store the value onto the HTTP Session and use the technique that we have already demonstrated for utilizing that value from within the getValue() method.

Another, alternative to using the HTTP Session directly is to use a servlet filter to write the request parameter to the session and to enforce security on the parameter usage.

The use case shown to illustrate this technique is a customization that reads the site customization value from a context parameter configured in the web.xml file. Using a URL request parameter – siteCC – MDS administrators can temporarily override the site value defined in the web.xml file, for example to test seeded customization. To protect the ADF application from malicious use of this request parameter, the request parameter is restricted by a JAAS permission enforced by ADF Security. If the authenticated user is not allowed to override the site customization then the request parameter is ignored. The code for the servlet filter that implements this shown next:

```
final String SITE_URL_PARAM = "siteCC";
final String SITE_CC_SESSION_ATTIBUTE =
                              "adf.sample.cc.SiteCC.value";
```

…

```
public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain)
                     throws IOException, ServletException
  //check if siteCC parameter is on the request URL
  HttpServletRequest _req = (HttpServletRequest) request;
  HttpSession _session = _req.getSession(false);
  Object siteCCUrlParam = _req.getParameter(SITE_URL_PARAM);
  //set parameter if user is authorized to override the
  //site customization value
  if(siteCCUrlParam!= null && isAllowedOverride()){
    //if the request value is "reset", then remove the session
    //attribute so the default site customization configured in
    //web.xml shows
    if(((String)siteCCUrlParam).equalsIgnoreCase("RESET")){
        _session.removeAttribute(SITE_CC_SESSION_ATTIBUTE);
     }else{
        _session.setAttribute(SITE_CC_SESSION_ATTIBUTE,
                              siteCCUrlParam);
     }
  }
  chain.doFilter(request, response);
}
```

```
  private boolean isAllowedOverride(){  …   }
```

Note: The code to check authorization is explained later in this paper

Because the ADF security context is used to check authorization, the servlet filter needs to be configured to execute after the ADF binding filter in the web.xml file.

…

```
//adf bindings filter
```

…

```
<filter>
  <filter-name>SiteCCFilter</filter-name>
  <filter-
class>adf.sample.poicc.servlet.DynamicSiteCCFilter</filter-class>
</filter>
```

…

```
//adf binding filter mapping
```

…

```
<filter-mapping>
  <filter-name>SiteCCFilter</filter-name>
  <url-pattern>/faces/*</url-pattern>
</filter-mapping>
```

Next the `DynamicSiteCC` customization class is created to read the site value from the web.xml context. If the site information is overridden in the http session and the user has the permission to use it, then this information is used instead.

This customization class also performs an authorization check before applying the customization value changes. It is good practice to carry out these security checks on multiple layers even if they appear to duplicate functionality.

```
final String ctxParameter = "adf.sample.cc.Site";
final String SITE_CC_SESSION_ATTIBUTE =
                            "adf.sample.cc.SiteCC.value";
```

…

```
public String[] getValue(RestrictedSession restrictedSession,
                         MetadataObject metadataObject) {
  //accessing the MDS session
  SessionOptions so = restrictedSession.getSessionOptions();
  ServletContext servletCtx = null;
  servletCtx = (ServletContext) so.getServletContextAsObject();
  //read site information from context parameter
```

```
  Object siteCCValue = null;
  siteCCValue = servletCtx.getInitParameter(ctxParameter);
  //try to read session attribute to check if site information is
  //overridden using the siteCC URL request parameter
  ADFContext adfCtx = ADFContext.getCurrent();
  Object _request = adfCtx.getEnvironment().getRequest();
  Object siteCCSessionAttribute  = null;
  if (_request != null && _request instanceof HttpServletRequest{
    HttpSession _session = null;
    _session = ((HttpServletRequest)_request).getSession(true);
    siteCCSessionAttribute  =
               _session.getAttribute(SITE_CC_SESSION_ATTIBUTE);
  }

  if(siteCCSessionAttribute !=null && isAllowedOverride()){
    //site information is overridden
    siteCCValue = siteCCSessionAttribute;
  }

  //if siteValue is set, return it
  if (siteCCValue != null){
    return new String[]{(String)siteCCValue};
  }
  return new String[0];
}

private boolean isAllowedOverride(){ … }
```

Note: The content of the isAllowedOverride method is explained later in this paper

To configure the context parameter read by the DynamicSiteCC, add the following content to the web.xml file

```
<!-- web.xml context parameter read by the customization
     class to determine the site value. Sample values:
     america, emea, apac , reset -->
 <context-param>
  <param-name>adf.sample.cc.Site</param-name>
  <param-value>america</param-value>
</context-param>
```

Using ADF Security in MDS customization and personalization

Many developers are familiar with ADF Security as a declarative way to protect ADF pages and task flows. However, ADF security provides more than that and leverages the Oracle Platform Security Services (OPSS) security framework for JAAS authorization. Customization classes, as well as managed bean code, within ADF applications execute in the context of the ADF binding context, which also handles the MDS lifecycle. Therefore, OPSS programming interfaces can be used in the Java code to check permissions defined in the jazn-data.xml file.

The custom `SiteCC` implementation – `DynamicSiteCC` – discussed earlier in this paper allows administrators to use a URL request parameter to override the current value of the customization layer and thus the applied customization. The URL request parameter in the example is "siteCC" and is considered for customization only if the authenticated user has a specific permission granted to do so. As you can see from this example, layered security using JAAS can be implemented to harden MDS customization.

About the OPSS Resource Permission

The Oracle Platform Security Services (OPSS) ResourcePermission is a generic Java security permission that can be used to protect various targets based on a defined resource types and multiple supported actions. At design time, the ResourcePermission is configured in the jazn-data.xml policy store, using the syntax below

```
<permission>
  <class>oracle.security.jps.ResourcePermission</class>
    <name>resourceType=type,resourceName=name</name>
  <actions>comma separated list of actions</actions>
</permission>
```

If you are familiar with JAAS, you will recognize the special use of the "name" element, which in JAAS defines the target name of a protected object. Using the OPSS `ResourcePermission`, this name is defined in two parts: a resource type and the resource name.

The resource *type* qualifies the type of resource to protect and helps you to organize and structure permission grants in the jazn-data.xml file. A resource type is an arbitrary string and, for example, could be defined as Page, Task Flow, Attribute, Entity, Method, Operation or whatever resource categories developers come up with. The resource *name* then defines the target. For example, an identifier for the site layer that we are checking. So to define a permission to be used with MDS we will have a permission like this:

```
<permission>
  <class>oracle.security.jps.ResourcePermission</class>
  <name>resourceType=Customization,resourceName=adf.sample.cc.Site</name>
  <actions>override</actions>
</permission>
```

The *resourceType* attribute value "Customization" is and arbitrary token chosen by the author of this whitepaper to identify MDS types of resources.

How-to check OPSS Permissions in Java

OPSS permissions can be checked in Java and from Expression Language (EL). For the use with customization classes in MDS, the Java option is of course used. So, to check if a user has the adf.sample.cc.Site permission defined in the section above, the following Java method can be build in a servlet filter or a customization class

```
public static final String ACTION = "override";
public static final String RESOURCE_NAME = "adf.sample.cc.Site";
public static final String RESOURCE_TYPE = "Customization";
…
private boolean isAllowedOverride (){
  boolean hasPermissionGranted = false;
  ADFContext adfCtx = ADFContext.getCurrent();
  SecurityContext securityCtx = adfCtx.getSecurityContext();
  ResourcePermission sitePermission = null;
  sitePermission =
    new ResourcePermission(RESOURCE_TYPE,RESOURCE_NAME,ACTION);
  hasPermissionGranted =
    securityCtx.hasPermission(sitePermission);
  return hasPermissionGranted;
}
```

Note: The above code was referred to earlier when discussing the servlet filter security and `DynamicSiteCC` customization layer security.

How-to protect MDS customization with OPSS

ADF Security checks are performed from the ADF security context, which, in turn, is accessible from the ADF context. The ADF context is accessible in a servlet filter, the customization class, managed beans and ADF phase listeners. This provides a multi layered environment to

implement security when using MDS customization. As we have shown it is always good to think about defense in depth and not rely on a single security check that could be possibly bypassed.

To configure ADF Security for an application, choose Application | Secure | Configure ADF Security from the Oracle JDeveloper 11g menu. In the opened security wizard, enable ADF Security for authentication at least.

Note: For the MDS customization layer security example in this whitepaper you don't need to enable authorization to be enforced on the ADF framework. However, Oracle recommends to also using ADF Security authorization to protect ADF applications in production environments.

After finishing the configuration wizard, choose Application | Secure | ADF Policies to create users, enterprise roles and application roles.

The visual policy editor does not yet provide an option to declaratively configure custom JAAS permissions and the OPSS `ResourcePermission` grants to application roles, and user principals.

To grant custom permissions to application roles in the jazn-data.xml policy store, developer can use the configuration editor that is accessible from the Structure window in Oracle JDeveloper. For this,

- Select the jazn-data.xml file in the Application Resources | Descriptors | META-INF location in Oracle JDeveloper
- Open the Structure window
- Expand the jazn-data | policy-store | applications | jazn-policy node. This node contains all the grant entries, which contain the information about the grantee. Choose the grant entry that belongs to the Principal that should have a specific permission granted
- If the grant entry is found, select the "permission" child entry and choose Insert inside permissions | permission from the right mouse menu. This launches the permission grant editor
- Use the permission grant editor to define the Permission class, name and the actions to grant to the grantees. If multiple actions are allowed then add them as a comma separated list.

For example, to grant the adf.sample.cc.Site customization to the "manager" application role, the permission grant looks as follows

```
<jazn-policy>
  <grant>
    <grantee>
      <principals>
        <principal>
```

```
          <class>
          oracle.security.jps.service.policystore.ApplicationRole
          </class>
          <name>app_manager</name>
        </principal>
      </principals>
    </grantee>
    <permissions>
      <permission>
       <class>
          oracle.security.jps.ResourcePermission
       </class>
       <name>
          resourceType=Customization,
          resourceName=adf.sample.cc.Site
       </name>
       <actions>override</actions>
     </permission>
   </permissions>
  </grant>
</jazn-policy>
```

Note: To learn more about ADF Security, please refer to the "Enabling ADF Security in a Fusion Web Application" chapter of the Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework product documentation

A Developer Check List for implementing MDS Customizations

This paper has taken you on a long trip that started with the positioning of MDS, followed by a technical introduction of MDS for site level and end user customization and specific development topics. The following checklist will help you to get started with MDS in your own ADF application.

To enable customization for an ADF web application, you perform the following steps

- Enable the view layer project for session based or MDS based persistence
- Optionally, enable seeded customization
- If MDS persistence is chosen, configure the adf-config.xml file with the components you want to save state information about
- If seeded customization is enabled, configure the CustomizationLayerValues.xml containing the IDE wide customization layer settings

- Optionally, when using seeded customization, create and configure your own customization objects
- Open JDeveloper in Customization Developer role to define customization metadata for seeded customization

Summary

The Web 2.0 paradigm introduced smarter web applications but also imposed the requirement for personalization and customization to rich Internet applications, a goal that is either expensive or impossible to reach for many development technologies and application runtime environments. Oracle Metadata Services in Oracle Fusion Middleware and the Fusion Development platform allows adding personalization and customization to Fusion web applications without the need to change the base application. It uses a layered approach that not only proves to be flexible at runtime but is robust and secure for large scale deployments. This whitepaper introduced MDS and its architecture and explored how developers build and configure MDS customizable applications.

Appendix: Redirecting a JSF view to itself

The "power user" customization example shown in Figure 1 of this whitepaper allows users to dynamically switch between the two modes, "Normal User" and "Power User", using a menu item. The logic behind the menu sets a session attribute for the customization layer to read so the requested customization can be applied to the application. To apply the customization, a redirect of the JSF document is required, which is performed by the code shown below

```
private void redirectToSelf(){
  FacesContext fctx = FacesContext.getCurrentInstance();
  ExternalContext ectx = fctx.getExternalContext();
  String viewId = fctx.getViewRoot().getViewId();
  ControllerContext controllerCtx = null;
  controllerCtx = ControllerContext.getInstance();
  String activityURL =
         controllerCtx.getGlobalViewActivityURL(viewId);
  try {
    ectx.redirect(activityURL);
    fctx.responseComplete();
   } catch (IOException e) {
    //Can't redirect
    e.printStackTrace();
    fctx.renderResponse();
  }
}
```

The ControllerContext.getInstance() method appends the ADF controller _adf.ctrl-state token to the redirect URL it creates to ensure that when the document re-renders, the page state, such as the current row selection of a table is preserved.

Building Customizable Oracle ADF Business Applications with Oracle Metadata Services (MDS)
June 2010
Author: Frank Nimphius
Contributing Authors: Duncan Mills

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment

0109