**Best Practices For Oracle ADF Development**

## General coding guidelines

A few guidelines as listed below apply to any piece of code or program written in any technology, language or framework. All the specific guidelines, recommendations or best practices in a framework will help the core technology developers to achieve one or more of these in some way or the other.

* Always look for solution that’s most efficient in terms of performance and memory usage.
* Consider multi-language requirements
* Make the code readable, scalable, consistent, maintainable
* Always look for a declarative solution first before starting complex coding. This makes the code less prone to error and easier to handle if migrating to future releases
* Always use re-usable components wherever possible
* Keep a segregation between different logics and different objects so that it is easier to locate a certain type of file or certain piece of code.

## Recommended naming conventions

One general convention that should be followed across all the object names’ is Camel casing convention. Other specific recommendations are as below.

#### Packages/Directories

Prefer to start the package name with the organization name or the web domain name. This avoids clash with any reusable code from other organizations. Besides this, try to have the package names as distinct as possible along with maintaining the clarity on the purpose of the package i.e. a package name should indicate that what is contained in it. Also, have different packages for different types of objects to have clear segregation.

Recommendations (taking the example of a simple CRUD application)–

|  |  |
| --- | --- |
| **Type of objects contained** | **Package/Directory Name** |
| Entity objects | com.mfgorc.crud.model.entities |
| Associations | com.mfgorc.crud.model.entities.assoc |
| View objects | com.mfgorc.crud.model.dataaccess |
| View links | com.mfgorc.crud.model.dataaccess.link |
| Resource bundle files for Model project | com.mfgorc.crud.model.resources |
| Application Modules | com.mfgorc.crud.model.facade |
| Managed beans | com.mfgorc.crud.view.backing |
| Java utility classes | com.mfgorc.crud.view.util |
| Resource bundle files for VC project | com.mfgorc.crud.view.resources |
| Fragments | com.mfgorc.crud.view.fragments |
| CSS files | com.mfgorc.crud.view.css |
| Image files | com.mfgorc.crud.view.images |
| Jspx pages | Public\_html |
| Task flows | com.mfgorc.crud.view.flow |

#### Projects

Try not to use default project names like Model, ViewController. Instead, use names that define the specific purpose of your project.

Examples – for a CRUD application, you can have the names as CrudModel, CrudVC or CrudView

#### Entity object

Entity object represents a row in a certain table. Hence, it is recommended to use a singular noun for EO name.

Examples – Employee, EmployeeEO

#### Association

Use ‘Assoc’ suffix to the name of association, which should also indicate the source n destination EOs.

Example – EmpDeptAssoc

#### View Objects

VO name should indicate the functionality or the set of data it represents. It does not necessarily indicate the base EO or table name. Use ‘VO’ suffix for simple view objects and ‘LOV’ suffix for VOs that represent list of values to be used in the application.

Examples – EmployeeVO, DeptLOV

#### View Links

Use suffix ‘Link’ to the view link name, which should also indicate the source n destination view objects.

Example – EmpDeptLink

#### Application Modules

These should be named according to the functionality, service or the set of view objects it contains. You can define a specific suffix to use consistently across all the model projects/AMs in the application.

Examples – CrudAM, CrudService

#### Task flows

Have the task flow id as short as possible. Also, it should represent the use case it represents. Use suffix ‘btf’ or ‘flow’. For example, to have a task flow that helps search for an employee, we can use id as emp-search-flow or emp-search-btf

#### Fragments/Pages

You can normally differentiate between fragments and pages from their extension jsff/jspx but its better to define a suffix for each to make it consistent along with having the names indicative of the functionality that each provides.

Examples – EmpSearchFr.jsff, EmpSearchPg.jspx

#### Resource Bundle keys

Prefer to have a defined prefix for each key type that will help to find a specific key when needed. Have all the keys defined in UPPER case. Try to have the key names indicative of what they are used for.

Recommendations/Examples

|  |  |
| --- | --- |
| **Purpose/Type** | **Example key names** |
| Command components (command button, command link etc) | CMD\_SEARCH, CMD\_CANCEL |
| Output components | MSG\_WELCOME, OUT\_LOADING\_SCREEN |
| Field labels | LABEL\_EMP\_ID |
| Table component | TBL\_SUMMARY\_EMP, TBL\_HEADER\_EMP\_ID |
| Headings | TITLE\_EMP\_SEARCH |
| Warning messages | WARN\_NO\_DATA |
| Error messages | ERR\_EMP\_SEARCH |

#### CSS files

Use suffix ‘css’ or ‘style’. For naming, you can use string to indicate the target for which this CSS would be used. This is mainly applicable if you are targeting your applications for browsers having different skin families. For example, you can use AndroidStyle.css, IPhoneStyle.css, BlackBerryStyle.css, ExplorerStyle.css etc. If you have multiple skins in your application, you can define names on the basis of theme of your skin like ChristmasStyle.css, EnterpriseStyle.css etc

#### Managed Beans

Use a consistent suffix/prefix like ‘Bean’, ‘Backing’, ‘MB’ to name your managed beans. Start your managed bean name with an upper case letter.

Example – EmployeeBean.java

## Best practices for ADF Development

#### ADF Application – Best practices in general

* + 1. Structure

1. Use different projects for grouping different modules and use library files for dependencies between them, for example in case you are handling different department processes in the application like human resources, purchasing, inventory, you can have different Model and VC projects created for each and then use their libraries in a common application as dependencies.

**Advantage** – Helps in partitioning, reuse and easy maintenance

1. Always detach the business logic from the UI layer.

**Advantage** – Easier reuse

**Example** – Developers tend to retrieve the values from UI using the EL evaluators to get the values from the pageFlowScope, sessionScope etc directly in the Business component Java classes like AppModuleImpl. This should be strictly avoided. Instead, methods should use the input parameters to retrieve such values.

* + 1. Resource Bundle

Ensure to use resource bundles in your application for any text that would appear on screen. This applies to attribute labels in EOs, VOs, ADF components (like outputText, inputText) labels, headings and other attributes. You can define a separate properties file for Model and VC project.

**Advantage**

– User viewable text becomes consistent across application screens like there could be chances that you define the label for Employee id field on one screen as ‘Emp Id’, another screen as ‘Employee Id’ and yet another screen as ‘Emp ID’(notice the upper case in ID here). To avoid this inconsistency, define a key, value for employee id label in properties file like *EMP\_ID\_LABEL = Employee Id* and use it in the control hints of your EO attribute’s properties.

* Helps in implementing internationalization i.e. allowing user to view the same application in different languages
  + 1. XML files

1. Ensure all XML files are valid and complete without any structure errors.

**Advantage** – Avoids any compilation/ run time errors

1. Use declarative editors for modifying XML over source code

**Advantage** – Avoids structure errors in XML file and makes the updates fast

* + 1. Java files

1. Use proper indentation and have detailed comments on your different logical pieces of code. Add Java docs to the Java methods

**Advantage** – Increases readability and understandability of the code

1. Ensure all catch blocks re-throw exceptions or log them.

**Advantage** – Avoids bugs in the application

1. Don’t use System.exit()

**Advantage** – This forces JVM container to quit. So, better to avoid\

1. Remember to close the callableStatement, preparedStatements, iterators, connections etc that you initialize or create.
   * 1. Logging
2. Ensure to use ADF Loggers in your application’s Java classes, especially when catching any kind of exception.

**Advantage** – Helps in debugging, log tracking.

1. Log critical errors at SEVERE level

**Advantage** – Most production application servers will restrict logging output to SEVERE only during routine operations. So, this will help finding out the reasons of the critical show stoppers that occurred in the application.

1. Log only the important information and that too at the right log levels. Do not log everything at SEVERE level and likewise, do not log everything at INFO level.

**Advantage** – Will prevent performance blocks and allow to categorize the logs and find the actual information that one is looking for.

1. Do not use system.out.\* functions for outputs. If at all you use for testing, remove them before releasing code.

**Advantage** – These do not put any information on the server, in fact are an overhead on the server. So, avoiding these will make code more optimized.

#### Model project specific best practices

* + 1. Structure

1. Have different types of business components in different packages like you can create a package for entity objects, another for view objects, and another one for associations and so on.

**Advantage -** In real time enterprise projects where there could be thousands of business component objects, it is practically difficult to find the relevant object in the model project. So, it is better to keep a segregation so as to allow ease of viewing and managing the objects.

1. Create your own base class while creating business component objects, instead of using the default base class.

**Advantage –** Provides a layer of abstraction on top of the default base class i.e. if the default behavior of the base class does not suit your needs then the behavior can be implemented in your own base class

* + 1. Application Module

1. All database operations should use the connection from the application module i.e. there should not be any connection details hardcoding in the Java classes.

Also, ensure that the application module should make use of connection from the data source defined on the weblogic server i.e. app module configuration should make use of JDBC data source and not JDBC URL.

**Advantage –** Database server details if change, will not impact application code

1. Data model in application module should expose only VOs that are required by the UI layer. Do not expose each and every VO that you create.
2. Ensure to tune your application module
   * 1. Entity objects
   1. Every entity object should have a primary key defined
   2. Attempt to avoid using ROWID as the primary key
   3. Make use of history columns in entity objects wherever appropriate

**Advantage –** You will not need to programmatically populate these column values.

* 1. Set the default labels for the attributes that will be used on the UI

**Advantage –** Makes attributes user friendly

* 1. Define tooltip for the attributes that you think would need explanation to the user.

**Advantage –** Makes attributes user friendly

* 1. Ensure that the labels, tooltips etc you define are taken from the resource bundle.

**Advantage –** Supports internationalization

* 1. Do not generate unnecessary Java classes for every EO. In fact, whenever you want to populate an attribute value, consider using Groovy expression and not Java code.
     1. View objects

1. Use declarative SQL view objects where possible
2. Every VO should have a primary key defined on some attribute based on the underlying EO preferably
3. Never use select \* in your VO query.
4. Ensure that the labels, tooltips etc you define are taken from the resource bundle.

**Advantage –** Supports internationalization

1. Do not generate unnecessary Java classes for every VO. In fact, whenever you want to populate an attribute value, consider using Groovy expression and not Java code.
2. Ensure to tune each VO in your application

#### View Controller project specific best practices

* + 1. General

1. Stored information’s scope should be as short as possible. Mainly, session and application scopes should be avoided as far as possible or at least keep track of when and why these are needed and used.

**Advantage** - Avoids memory leaks in the applications

1. Ensure to have handled the session expiration of your application, in case you have sensitive user specific data. By default, session expiration leads to a screen with a message that allows to reload the same screen on which session expiry happened. You can handle this to redirect to a custom error screen or say login screen after invalidating the session.

**Advantage –** Disallows wrong user access to sensitive information.

* + 1. Trinidad-config.xml

1. Have animation-enabled=false in Trinidad-config.xml file if applicable

**Advantage** – Better browser performance

* + 1. web.xml

1. Specify welcome page definition in the application’s xml file.

**Advantage** – Avoids 403/404 error when user presses Ctrl+N or Ctrl+T that opens up a new browser window/tab and seeks a welcome page to open.

* + 1. Task flows

1. When adding parameters to the task flows, always specify a value in the Class field explicitly, even if the value is java.lang.String

**Advantage** – Ensures that the object of correct type is passed.

1. If the conditions for navigation logic can be expressed using an EL, prefer to use a router in the task flow

**Advantage** – Makes the taskflow conditional information more readable. Also, makes it easier for future modifications.

1. In case you need to invoke a method before the page is rendered, prefer to use a method call activity on the task flow instead of using invokeAction on the pageDefinition. This best practice can be ignored if

* Method needs to be executed in more than 1 phase of the page’s lifecycle.
* Method is required to be tied to the page, and page needs to be reused.

**Advantage** – Easier to invoke the same logic between pages, also makes the information more readable and useful to another developer.

1. Merge duplicate navigation rules in task-flow using wildcard control activity

**Advantage** – Makes the navigation logic more readable

* + 1. Pages/Fragments

1. Create XML-based JSP document i.e. \*.jspx document instead of \*.jsp file

**Advantage**

* + Tree for UI component tags is well formed
  + Avoids mixing of Java code and component tags

1. Make use of page templates.

**Advantage** – Help define a common page structure that can used across multiple pages in the application, avoids re-writing the same layout structure code at multiple places.

1. Have the component ids in ADF pages as short as possible.

**Advantage** – Improves performance

1. Do not bind every component to the managed bean variables. In fact, keep the binding to a minimum.
2. To have javascript on ADF screens, use af:resource component instead of adding script tag in f:verbatim tag.

**Advantage** – Provides better performance

1. Avoid use of inlineStyle attribute

**Advantage** – Scope of style will not be limited to specific components, styles will be reusable

1. Don’t use multiple af:form tags on one page as multiple 'form' tags are not supported in ADF faces, though it does not give any error. Usage of 'subform' is recommended instead
2. Use styleClass="AFStretchWidth" for stretching ADF components across full width as usage of width in % is not recommended as it fights the internal geometry management of ADF

**Advantage** – Better performance

1. Avoid full page refresh on action components like commandButton, commandLink etc. Try to use partialSubmit = “true” on these and then refresh only the components needed by using partialTriggers.

**Advantage** – Reduce the load on cycle to the server, hence leading to better performance

1. Use rendered = ‘false’ instead of visible = ‘false’ whenever possible.
2. Avoid using af:inlineFrame, af:verbatim, jsp:include, jsp:scriptlet tags
3. All your pages/fragments should follow the LIQUID design i.e. increasing/decreasing the window size or font size of browser should not affect the screen layout. Make use of panel components appropriately to ensure this. While unit testing, ensure to test you page layouts by increasing/decreasing the browser font and window size.
   * 1. Managed Beans
4. Do not define beans in faces-config.xml, but do that in adfc-config.xml
5. Make use of common utility classes (ADFUtils, JSFUtils) for common functions.

**Advantage** – Increases code reuse and reduces code quantity

#### Deployment to development/ testing server

1. You can consider to have the following property values in application’s web.xml file to enable easy debugging/testing

oracle.adf.view.rich.ASSERT\_ENABLED=true

org.apache.myfaces.trinidad.DEBUG\_JAVASCRIPT=true

org.apache.myfaces.trinidad.DISABLE\_CONTENT\_COMPRESSION=true

oracle.adf.view.rich.LOGGER\_LEVEL=true

However, any testing server where performance is a key parameter to be noticed, do not use these.

#### Deployment to production servers

1. Remember to disable the debug features before deploying application to standalone server. For eg, web.xml properties should be like

oracle.adf.view.rich.automation.ENABLE=false5 [default value]

oracle.adf.view.rich.ASSERT\_ENABLED=false6 [default value]

org.apache.myfaces.trinidad.CHECK\_FILE\_MODIFICATION=false [default value]

org.apache.myfaces.trinidad.COMPRESS\_VIEW\_STATES=true

org.apache.myfaces.trinidad.DEBUG\_JAVASCRIPT=false

org.apache.myfaces.trinidad.DISABLE\_CONTENT\_COMPRESSION=false [default value]

oracle.adf.view.rich.libraryPartitioning.DISABLED=false [default value]

oracle.adf.view.rich.LOGGER\_LEVEL=false [default value]

javax.faces.STATE\_SAVING\_METHOD=client

**Advantage** – Improves performance

1. Do a cleaning up/optimization exercise on your code before deploying/releasing code to standalone/production server which should include the following-
   1. Reformat all your code. This can be done by using Jdeveloper ‘Reformat’ option.
   2. Ensure there are no unused strings in the resource bundle properties file. If there are, comment/delete them
   3. Ensure that no code line exceeds 80 characters
   4. Remove any unused classes/packages/images/CSS/JSPs
   5. Remove any unused business component(EO, VO etc)
   6. Remove any unused bind variables, view criterias, LOVs in any view object
   7. Verify that there are no unused tag libraries in ADF pages
   8. Ensure that there are no hard coded labels or outputs in ADF pages but are retrieved from business components or resource bundle files
   9. Remove any unused iterator or component bindings from page definitions.
   10. Remove any unused navigation rules in the task flows
   11. Remove any unused import/method/attribute in any Java class

**References:**

* <http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adf-code-guidelines-v1-00-1845659.pdf>
* Oracle ADF Developer Guide