

# **Oracle® Banking Platform**

UI Extensibility Guide

Release 2.6.2.0.0

**E95189-01**

May 2018

Oracle Banking Platform UI Extensibility Guide, Release 2.6.2.0.0

E95189-01

Copyright © 2011, 2018, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate failsafe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This software or hardware and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

# Contents

---

<b>Preface</b> .....	<b>16</b>
Audience .....	16
Documentation Accessibility .....	16
Related Documents .....	16
Conventions .....	16
<b>1 About This Guide</b> .....	<b>19</b>
1.1 Sections Not Applicable for Oracle Banking Enterprise Product Manufacturing .....	19
<b>2 Objective and Scope</b> .....	<b>21</b>
2.1 Overview .....	21
2.2 Objective and Scope .....	21
2.2.1 Extensibility Objective .....	21
2.2.2 Document Scope .....	21
2.3 Complementary Artefacts .....	22
2.4 Out of Scope .....	23
<b>3 Overview of Use Cases</b> .....	<b>25</b>
3.1 Extensibility Use Cases .....	25
3.1.1 ADF Screen Customization Using UI Extensions .....	25
3.1.2 ADF Screen Customization Using MDS .....	26
3.1.3 Print Receipt Functionality .....	26
<b>4 OBP Extensibility Support Using Eclipse Plugin</b> .....	<b>29</b>
4.1 Configure Eclipse Preferences for OBP Service Plugin .....	29
4.2 Support for UI Extension .....	33
4.2.1 Generate UI Extension .....	33

---

4.2.2 Configure OBP Extensibility Server Explorer - View .....	42
4.2.3 Exposed Webservice for UI Extensions .....	48
4.2.4 Deploy UI Extension to Server .....	48
4.2.5 Database Inserts: UI Extension Deployment .....	52
4.2.6 Fetching Deployed UI Extension .....	55
4.2.7 Undeploying UI Extension .....	57
4.2.8 Case of Multiple UI Extensions .....	59
4.2.9 Inclusion of Groovy Extension in Actual Code Flow .....	60
<b>5 ADF Screen Customizations Using UI Extensions .....</b>	<b>63</b>
5.1 UI Extension Interface .....	65
5.2 Default UI Extension .....	66
5.3 UI Extension Executor .....	67
5.4 Extension Configuration .....	70
5.5 Customization Examples .....	71
5.5.1 Replacing skin .....	71
5.5.2 Changing the logo in the branding bar .....	73
5.5.3 Modifying fonts .....	73
5.5.4 Modifying images .....	74
5.5.5 Graphics .....	74
5.5.6 Adding a simple field to a product screen .....	74
5.5.7 Adding a complex field popup to a product screen (popup, table, tree, region, tf) .....	75
5.5.8 Removing an existing field from a product screen .....	75
5.5.9 Making certain product optional product fields mandatory or optional .....	75
5.5.10 Adding a new column to an existing product grid .....	75
5.5.11 Hiding columns from an existing product grid .....	76



---

5.5.12 Graying out certain columns from an existing product grid .....	77
5.5.13 Modifying properties of product table (rows or tablesummary) .....	77
5.5.14 Adding a new section to an existing product screen .....	77
5.5.15 Hiding a section from a product screen .....	78
5.5.16 Adding a new tab to an existing product screen made of tabs .....	78
5.5.17 Hiding a tab from a product screen made of multiple tabs .....	79
5.5.18 Adding new buttons or links .....	79
5.5.19 Overriding / Customizing the product behaviour on certain actions like button clicks or tab-outs .....	80
5.5.20 Overriding the product validation pattern .....	80
5.5.21 Overriding the product lengths (min/max) .....	80
5.5.22 Disable / Enable certain product fields .....	80
5.5.23 Change certain product fields to read-only either on load or based on cer- tain conditions .....	80
5.5.24 Change label of existing product fields .....	80
5.5.25 DC validation .....	81
5.6 Using the JSFF Utils .....	81
5.6.1 How to Use JSFF Utils .....	81
5.6.2 Sample JSFF Utils Code Snippet .....	81
<b>6 ADF Screen Customizations Using MDS .....</b>	<b>83</b>
6.1 Seeded Customization Concepts .....	83
6.2 Customization Layer .....	84
6.3 Customization Class .....	84
6.4 Enabling Application for Seeded Customization .....	86
6.5 Customization Project .....	89
6.6 Customization Role and Context .....	89

---

6.7 Customization Layer Use Cases .....	92
6.7.1 Adding a UI Table Component to the Screen .....	92
6.7.2 Approvals Framework .....	105
6.7.3 Override the product managedBean .....	147
<b>7 ADF Taskflow Generation for Customization or Localization .....</b>	<b>149</b>
7.1 Introduction .....	149
7.2 Installing the Generator .....	149
7.3 Using the Generator .....	150
7.4 ADF TaskFlow Generation Usecases .....	157
7.4.1 Extensive UI components to be included on screen .....	157
<b>8 Human Task Screen Extension .....</b>	<b>159</b>
8.1 Introduction .....	159
8.2 Custom CSS Skin .....	159
8.2.1 Create New ADF Skin .....	159
8.2.2 Apply New Skin .....	160
<b>9 Receipt Printing .....</b>	<b>163</b>
9.1 Prerequisite .....	163
9.1.1 Identify Node Element for Attributes in Print Receipt Template .....	163
9.1.2 Receipt Format Template (.rtf) .....	165
9.2 Configuration .....	166
9.2.1 Parameter Configuration in the BROPCONFIG.properties .....	166
9.2.2 Configuration in the ReceiptPrintReports.properties .....	167
9.3 Implementation .....	167
9.3.1 Default Nodes .....	168
9.4 Special Scenarios .....	168

---

<b>10 Extensibility Usage – OBP Localization Pack .....</b>	<b>171</b>
10.1 Localization Implementation Architectural Change .....	172
10.2 Customizing UI Layer .....	173
10.2.1 JDeveloper and Project Customization .....	173
10.2.2 Generic Project Creation .....	179
10.2.3 MAR Creation .....	179
10.3 Source Maintenance and Build .....	187
10.3.1 Source Check-ins to SVN .....	187
10.3.2 .mar files Generated during Build .....	188
10.3.3 adf-config.xml .....	188
10.4 Packaging and Deployment of Localization Pack .....	188

# List of Figures

---

Figure 3–1 ADF Screen Extensions .....	25
Figure 3–2 ADF Screen Customization .....	26
Figure 3–3 Print Receipt Functionality .....	27
Figure 4–1 Java Eclipse - Select Preferences .....	29
Figure 4–2 Preferences Dialog Box - OBP Service Plugin .....	30
Figure 4–3 Folder Selection .....	31
Figure 4–4 Browse For Folder .....	32
Figure 4–5 Configuring MWLib Path Parameter .....	33
Figure 4–6 Generate UI Extension .....	34
Figure 4–7 UI Extension Configuration .....	35
Figure 4–8 UI Extension Configuration - Search for Base UI File .....	36
Figure 4–9 UI Extension Configuration - Base UI File Selected .....	37
Figure 4–10 UI Extension Configuration - Class Name and Package .....	38
Figure 4–11 UI Extension Configuration - Generate Extension Code .....	39
Figure 4–12 UI Extension Configuration - Code Generated with Extension Hooks ..	40
Figure 4–13 Project - UI Extension Created .....	41
Figure 4–14 Configuring OBP Extensibility Server Explorer .....	42
Figure 4–15 Create Server Explorer .....	43
Figure 4–16 Server Explorer View tab .....	44
Figure 4–17 Server Explorer - Create Server Connection .....	45
Figure 4–18 Server Connection Configuration .....	46
Figure 4–19 Server Configured .....	47
Figure 4–20 ExtensionApplicationServiceSpi Web Service for UI Extensions .....	48
Figure 4–21 Deploy UI Extension to Server .....	49

---

Figure 4–22 Select Server Explorer to Deploy UI Extension .....	50
Figure 4–23 UI Extensions Deployed to Server Explorer .....	51
Figure 4–24 Single Record View - UI Extension Deployment .....	52
Figure 4–25 Single Record View - UI Extension Deployment .....	53
Figure 4–26 View Value - UI Extension Deployment .....	54
Figure 4–27 Single Record View - UI Extension Deployment .....	55
Figure 4–28 Fetch Deployed UI Extension .....	56
Figure 4–29 Extension Saved in Project .....	57
Figure 4–30 Undeploy UI Extension from Server .....	58
Figure 4–31 UI Extension Undeployed .....	59
Figure 4–32 Multiple UI Extensions .....	60
Figure 4–33 UIExtensionFactory.java .....	60
Figure 4–34 UIGroovyExtensions .....	61
Figure 5–1 UI Extension Pre Hook and Post Hook Taskflow .....	64
Figure 5–2 Save Method in IntegrableTaskflowHelper .....	65
Figure 5–3 Example of UI Extension .....	66
Figure 5–4 Example of Default UI Extension .....	67
Figure 5–5 UI Extension Executor Class Taskflow .....	68
Figure 5–6 Example of UI Extension Executor Class .....	69
Figure 5–7 Example of UI Extension Executor Class .....	70
Figure 5–8 Replacing skin .....	72
Figure 5–9 Replacing skin .....	72
Figure 5–10 Example: Replacing skin .....	73
Figure 5–11 Replacing the logo .....	73
Figure 5–12 Example: To modify images .....	74

---

Figure 5–13 Example: To add a simple field to a product screen .....	75
Figure 5–14 Example: To remove an existing field from a region .....	75
Figure 5–15 Example: To add a new column to an existing prouct grid .....	76
Figure 5–16 Example: To hide columns from an existing product grid .....	77
Figure 5–17 Example: To modify the properties of product table .....	77
Figure 5–18 Example: To add a new section to an existing product screen .....	78
Figure 5–19 Example: To add a new tab to existing product screen made of tabs ...	79
Figure 5–20 Example: To hide a tab from a product screen made of multiple tabs ...	79
Figure 5–21 Example: To add new buttons or links .....	80
Figure 5–22 Example: To override the product validation pattern .....	80
Figure 5–23 Example of JSFF Utils .....	81
Figure 6–1 Customization Application View .....	83
Figure 6–2 CustomizationLayerValues.xml .....	84
Figure 6–3 Customization Class .....	85
Figure 6–4 Implementation for the abstract methods of CustomizationClass .....	86
Figure 6–5 Enable Seeded Customizations .....	87
Figure 6–6 Adding com.ofss.fc.demo.ui.OptionCC.jar .....	87
Figure 6–7 Adding com.ofss.fc.demo.ui.OptionCC.OptionCC .....	88
Figure 6–8 Adf-config.xml .....	88
Figure 6–9 Customization Developer .....	90
Figure 6–10 Selecting Always Prompt for Role Selection on Start Up .....	91
Figure 6–11 View Customization Context .....	92
Figure 6–12 Adding a UI Table Component - Party Search screen .....	93
Figure 6–13 Adding a UI Table Component - Related Party screen .....	93
Figure 6–14 Creating Binding Bean Class .....	94

---

Figure 6–15 Create Event Consumer Class .....	95
Figure 6–16 Creating Managed Bean .....	95
Figure 6–17 Create Data Control .....	96
Figure 6–18 Adding View Object Binding to Page Definition - Add Tree Binding ....	97
Figure 6–19 Adding View Object Binding to Page Definition - Update Root Data Source .....	98
Figure 6–20 Page Data Binding Definition - Insert Item .....	99
Figure 6–21 Page Data Binding Definition - Create Action Binding .....	100
Figure 6–22 Edit Event Map .....	101
Figure 6–23 Event Map Editor .....	102
Figure 6–24 Add UI Components to Screen .....	103
Figure 6–25 Application Navigator .....	104
Figure 6–26 Party Search .....	105
Figure 6–27 Contact Point Screen .....	106
Figure 6–28 Create Table .....	107
Figure 6–29 Create Java Project .....	107
Figure 6–30 Create Domain Objects .....	108
Figure 6–31 Create Interface .....	108
Figure 6–32 Create Class .....	109
Figure 6–33 Set OBP Plugin Preferences .....	109
Figure 6–34 Set OBP Plugin Preferences .....	110
Figure 6–35 Set OBP Pugin Prefernces .....	110
Figure 6–36 Create Application Service .....	111
Figure 6–37 Application Service Classes Generated .....	112
Figure 6–38 Modify Data Transfer Object (DTO) .....	113
Figure 6–39 Generate Service and Facade Layer Sources .....	114

---

Figure 6–40 Modify ContactExpiryApplicationServiceSpi.java .....	115
Figure 6–41 Modify ContactExpiryApplicationServiceSpi.java .....	116
Figure 6–42 Modify ContactExpiryApplicationServiceSpi.java .....	117
Figure 6–43 Java Packages .....	117
Figure 6–44 Export Java Project as JAR .....	118
Figure 6–45 Create ContactExpiry.hbm.xml .....	119
Figure 6–46 Configure hostapplicationlayer.properties .....	119
Figure 6–47 Configure ProxyFacadeConfig.properties .....	120
Figure 6–48 Configure JSONServiceMap.properties .....	120
Figure 6–49 Create Model Project .....	121
Figure 6–50 Create Model Project - Configure Java Settings .....	122
Figure 6–51 Create Application Module .....	123
Figure 6–52 Set Package and Name of Application Module .....	124
Figure 6–53 Summary of Application Module Created .....	125
Figure 6–54 Create View Object .....	126
Figure 6–55 View Attribute .....	127
Figure 6–56 Application Module .....	128
Figure 6–57 Create View Object - Summary .....	129
Figure 6–58 Create View Controller Project .....	130
Figure 6–59 Name your Project .....	131
Figure 6–60 Libraries and Classpath .....	132
Figure 6–61 Dependencies .....	132
Figure 6–62 Create Maintenance State Action Interface .....	133
Figure 6–63 Create Update State Action Class .....	134
Figure 6–64 Create Update State Action Class .....	135



---

Figure 6–65 DemoContactPoint.java displays the View Objects .....	136
Figure 6–66 DemoCreateContactPoint / DemoUpdateContactPoint .....	137
Figure 6–67 Create Contact Expiry DTO .....	137
Figure 6–68 Value Change Event Handler for the Expiry Date UI Component .....	138
Figure 6–69 Value Change Event Handlers for Existing UI Components .....	139
Figure 6–70 Method to fetch Screen Data using Contact Expiry Proxy Service .....	140
Figure 6–71 Create Managed Bean .....	141
Figure 6–72 Create Event Customer Class .....	141
Figure 6–73 Create Data Control .....	142
Figure 6–74 Generated contactPoint.jsff.xml .....	143
Figure 6–75 Add an attributeValues binding .....	143
Figure 6–76 Create Attribute Binding .....	144
Figure 6–77 Add a methodAction binding .....	144
Figure 6–78 Create Action Binding .....	145
Figure 6–79 Select the Event Consumer Method .....	146
Figure 6–80 Generated contactPoint.jsff.xml .....	146
Figure 6–81 PI041 - Contact Point Screen .....	147
Figure 7–1 Preferences screen .....	150
Figure 8–1 Sample trinidad-skins.xml .....	159
Figure 8–2 Package Structure .....	160
Figure 8–3 Sample Data .....	160
Figure 8–4 Sample Implementation .....	161
Figure 9–1 Input Property Files .....	163
Figure 9–2 Build Path of Utility .....	164
Figure 9–3 Utility Execution .....	165

---

Figure 9–4 Excel Generation .....	165
Figure 9–5 Receipt Format Template .....	166
Figure 9–6 Receipt Print Reports .....	167
Figure 9–7 Sample of Print Receipt .....	168
Figure 10–1 Perfection Capture Screen .....	171
Figure 10–2 Localization Implementation Architectural Change .....	172
Figure 10–3 Package Structure .....	173
Figure 10–4 Customization of the JDeveloper .....	174
Figure 10–5 Customization of the JDeveloper .....	174
Figure 10–6 Configure Design Time Customization layer .....	175
Figure 10–7 Enabling Seeded Customization .....	176
Figure 10–8 Library and Class Path .....	177
Figure 10–9 MDS Configuration .....	178
Figure 10–10 MDS Configuration .....	179
Figure 10–11 MAR Creation .....	180
Figure 10–12 MAR Creation - Application Properties .....	181
Figure 10–13 MAR Creation - Create Deployment Profile .....	182
Figure 10–14 MAR Creation - MAR File Selection .....	183
Figure 10–15 MAR Creation - Enter Details .....	184
Figure 10–16 MAR Creation - ADF Library Customization .....	185
Figure 10–17 MAR Creation - Edit File .....	186
Figure 10–18 MAR Creation - Application Assembly .....	187
Figure 10–19 Package Deployment .....	189

# List of Tables

---

Table 10–1 Path Structure ..... 187

# Preface

This guide explains customization and extension of Oracle Banking Platform.

This preface contains the following topics:

- [Audience](#)
- [Documentation Accessibility](#)
- [Related Documents](#)
- [Conventions](#)

## Audience

This guide is intended for the users of Oracle Banking Platform.

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/us/corporate/accessibility/index.html>.

### Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/us/corporate/accessibility/support/index.html#info> or visit <http://www.oracle.com/us/corporate/accessibility/support/index.html#trs> if you are hearing impaired.

## Related Documents

For more information, see the following documentation:

- For installation and configuration information, see the Oracle Banking Platform Installation Guide - Silent Installation.
- For a comprehensive overview of security, see the Oracle Banking Security Guide.
- For the complete list of licensed products and the third-party licenses included with the license, see the Oracle Banking Licensing Guide.
- For information related to setting up a bank or a branch, and other operational and administrative functions, see the Oracle Banking Administrator's Guide.
- For information on the functionality and features, see the respective Oracle Banking Functional Overview documents.

## Conventions

The following text conventions are used in this document:

Convention	Meaning
<b>boldface</b>	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.



# 1 About This Guide

This guide is applicable for the following products:

- Oracle Banking Platform (OBP)
- Oracle Banking Enterprise Product Manufacturing (OBEPM)
- Oracle Banking Enterprise Originations (OBEO)
- Oracle Banking Enterprise Collections (OBEC)

References to Oracle Banking Platform or OBP in this guide apply to all the above mentioned products. The chapters and sections that are not applicable for any of the products are listed in this chapter.

## 1.1 Sections Not Applicable for Oracle Banking Enterprise Product Manufacturing

The following chapters and sections in this guide are not applicable for Oracle Banking Enterprise Product Manufacturing.

- [Section 3.1.3 Print Receipt Functionality](#)





# 2 Objective and Scope

This chapter defines the objective and scope of this document.

## 2.1 Overview

Oracle Banking Platform (OBP) is designed to help banks respond strategically to today's business challenges, while also transforming their business models and processes to reduce operating costs and improve productivity across both front and back offices. It is a one-stop solution for a bank that seeks to leverage Oracle Fusion experience for its core banking operations, across its retail and corporate offerings.

OBP provides a unified yet scalable IT solution for a bank to manage its data and end-to-end business operations with an enriched user experience. It comprises pre-integrated enterprise applications leveraging and relying on the underlying Oracle Technology Stack to help reduce in-house integration and testing efforts.

## 2.2 Objective and Scope

Most product development can be accomplished through highly flexible system parameters and business rules. Further competitive differentiation can be achieved through IT configuration and extension support. In OBP, additional business logic required for certain services is not always a part of the core product functionality but could be a client requirement. For these purposes, extension points and customization support have been provided in the application code which can be implemented by the bank and / or by partners, wherein the existing business logic can be added with or overridden by customized business logic. This way the time consuming activity of custom coding to enable region specific, site specific or bank specific customizations can be minimized.

### 2.2.1 Extensibility Objective

The broad guiding principles with respect to providing extensibility in OBP are summarized below:

- Strategic intent for enabling customers and partners to extend the application.
- Internal development uses the same principles for client specific customizations.
- Localization packs
- Extensions by Oracle Consultants, Oracle Partners, Banks or Bank Partners.
- Extensions through the addition of new functionality or modification of existing functionality.
- Planned focus on this area of the application. Hence, separate budgets specifically for this.
- Standards based - OBP leverages standard tools and technology
- Leverage large development pool for standards based technology.
- Developer tool sets provided as part of JDeveloper and Eclipse for productivity.

### 2.2.2 Document Scope

The scope of this document is to explain the customization and extension of OBP for the following use cases:

- Customizing OBP UI
- Adding an ADF screen side validation to an existing field
- Adding a new field or a table on the screen
- Removing fields from the UI
- Customizing OBP application services and implementing composite application services
- Adding pre-processing or post processing validations in the application services extension
- Altering the product behavior at customizations hooks provided as adapter calls in functional areas that are prone to change (for example, loan schedule generation) and in between modules that can be replaced (for example, alerts, content management)
- Adding new fields to the OBP domain model and including it on the corresponding screen.
- Adding a new report
- Adding a new batch program
- Customizing SOA based BPEL process with adding a partner link or a human task to an existing process.
- Adding new steps as a sub-process
- Adding or customizing facts and business rules in the application and configuring them for different modules
- Adding or customizing ID generation logic with options of automated, manual or custom generation
- Processing of the uploaded files data
- Printing of receipt once the transaction is over
- Defining the security related access and authorization policies
- Defining different security related rules, validator and processing logics
- Customizing different functionalities like user search, role evaluation and limit exclusion in the application related to security

This document is a useful tool for Oracle Consulting, bank IT and partners for customizing and extending the product.

## 2.3 Complementary Artefacts

The document is a developer's extensibility guide and does not intend to work as a replacement of the functional or technical specification, which would be the primary resource covering the following:

- OBP Zen training course
- OBP installation and configuration
- OBP parameterization as part of implementation
- Functional solution and product user guide

References to plugin indicate the eclipse based OBP development plugin for relevant version of OBP being extended. The plugin is not a product GA artefact and is a means to assist development. Hence, the same is not covered under product support.

## 2.4 Out of Scope

The scope of extensibility does not intend to suggest that OBP is forward compatible.



# 3 Overview of Use Cases

The use cases that are covered in this document shall enable the developer in applying the discipline of extensibility to OBP. While the overall support for customizations is complete in most respects, the same is not a replacement for implementing a disciplined, thoughtful and well-designed approach towards implementing extensions and customizations to the product.

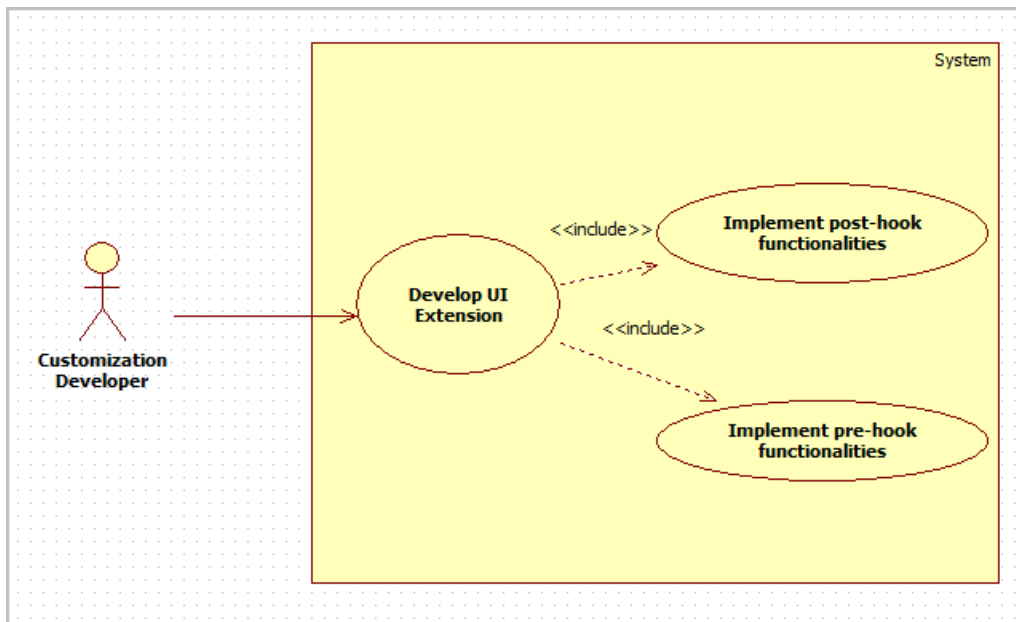
## 3.1 Extensibility Use Cases

This section gives an overview of the extensibility topics and customization use cases to be covered in this document. Each of these topics is detailed in the further sections.

### 3.1.1 ADF Screen Customization Using UI Extensions

In OBP, additional business logic or UI component changes might be required for certain ADF screen. This additional logic is not part of the core product functionality, but could be a client requirement. For this purpose, hooks have been provided in the application code wherein additional business logic can be added with custom business logic.

**Figure 3–1 ADF Screen Extensions**



---

**Note**

Screen changes can be implemented using the UI extensions or ADF Screen Customization. It is recommended to use the UI extensions where possible as migration path to higher release of OBP is easier.

---

**UI Extension:**

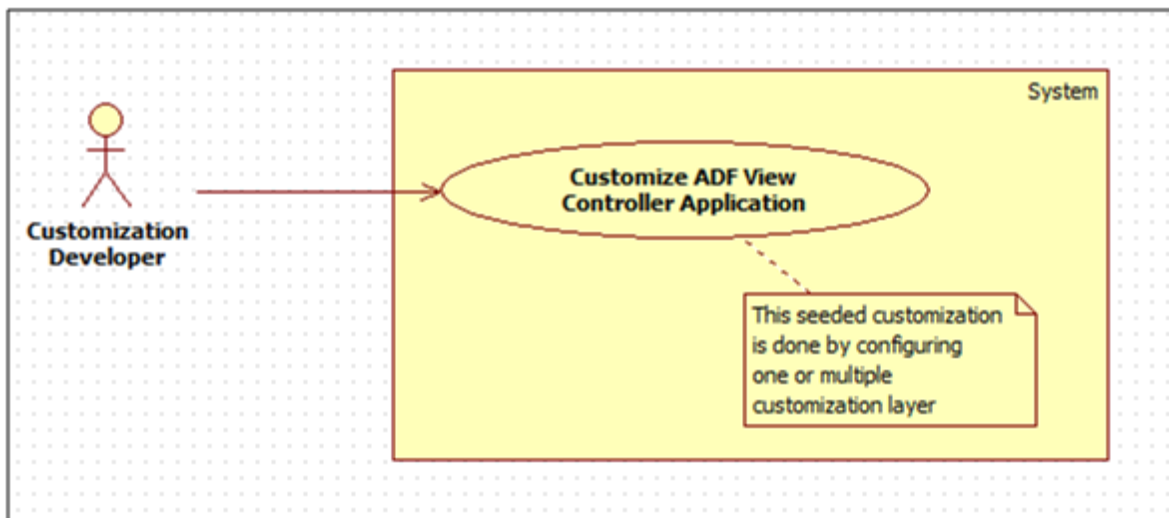
This hook resides in the ADF taskflow. This hook is present before as well as after the actual UI event execution. The additional business logic has to implement the interface `I<taskflow_name>UIExt` and extend and override the default implementation `Void<taskflow_name>UIExt` provided for the taskflow. Multiple implementations can be defined for a particular taskflow. The UI extensions executor invokes all the implementations defined for the particular taskflow both before and after the actual UI event execution.

#### 3.1.2 ADF Screen Customization Using MDS

OBP application may need to be customized for certain additional requirements. However, since these additional requirements differ from client to client, and the base application functionality remains the same, the code to handle the additional requirements is kept separate from the code of the base application. For this purpose, Seeded Customizations (built using Oracle Meta-data Services framework) can be used to customize an application.

When designing seeded customizations for an application, one or more customization layers need to be specified. A customization layer is used to hold a set of customizations. A customization layer supports one or more customization layer value which specifies the set of customizations to apply at runtime.

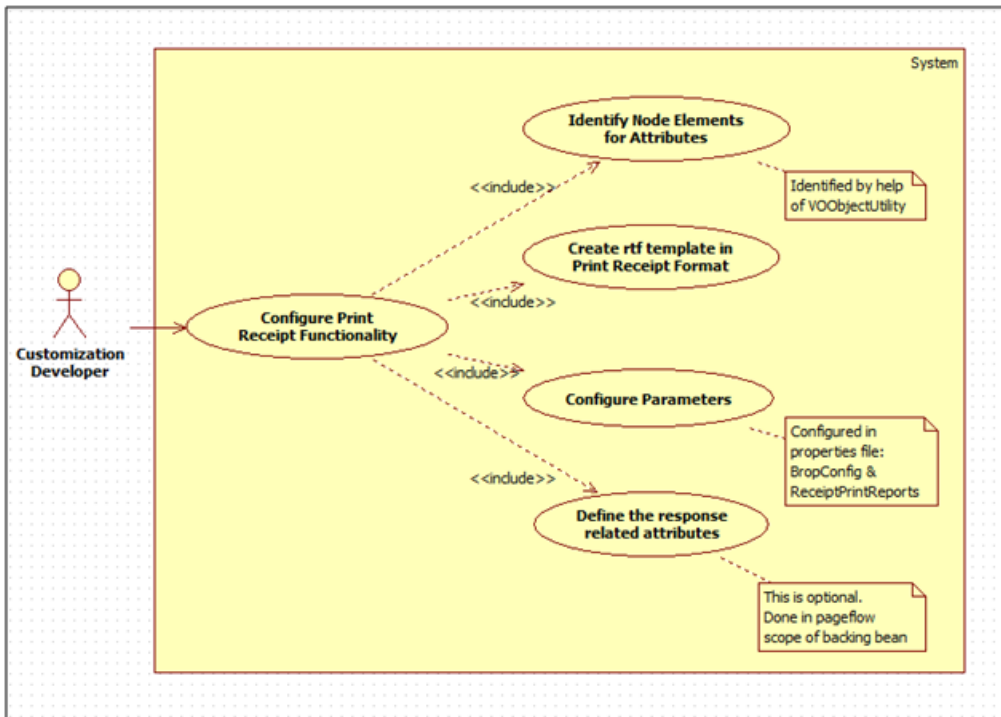
*Figure 3–2 ADF Screen Customization*



#### 3.1.3 Print Receipt Functionality

OBP has many transaction screens in different modules where it is desired to print the receipt with different details about the transaction. This functionality provides the print receipt button on the top right corner of the screen which gets enabled on the completion of the transaction and can be used for printing of receipt of the transaction details.

Figure 3–3 Print Receipt Functionality







# 4 OBP Extensibility Support Using Eclipse Plugin

OBP Eclipse Plugin has been updated to support OBP Extensibility features like run-time inclusion of Application Service SPI Extensions and Business Policy Extensions in the form of uploadable Groovy files.

## 4.1 Configure Eclipse Preferences for OBP Service Plugin

1. Click on Windows > Preferences.

*Figure 4–1 Java Eclipse - Select Preferences*

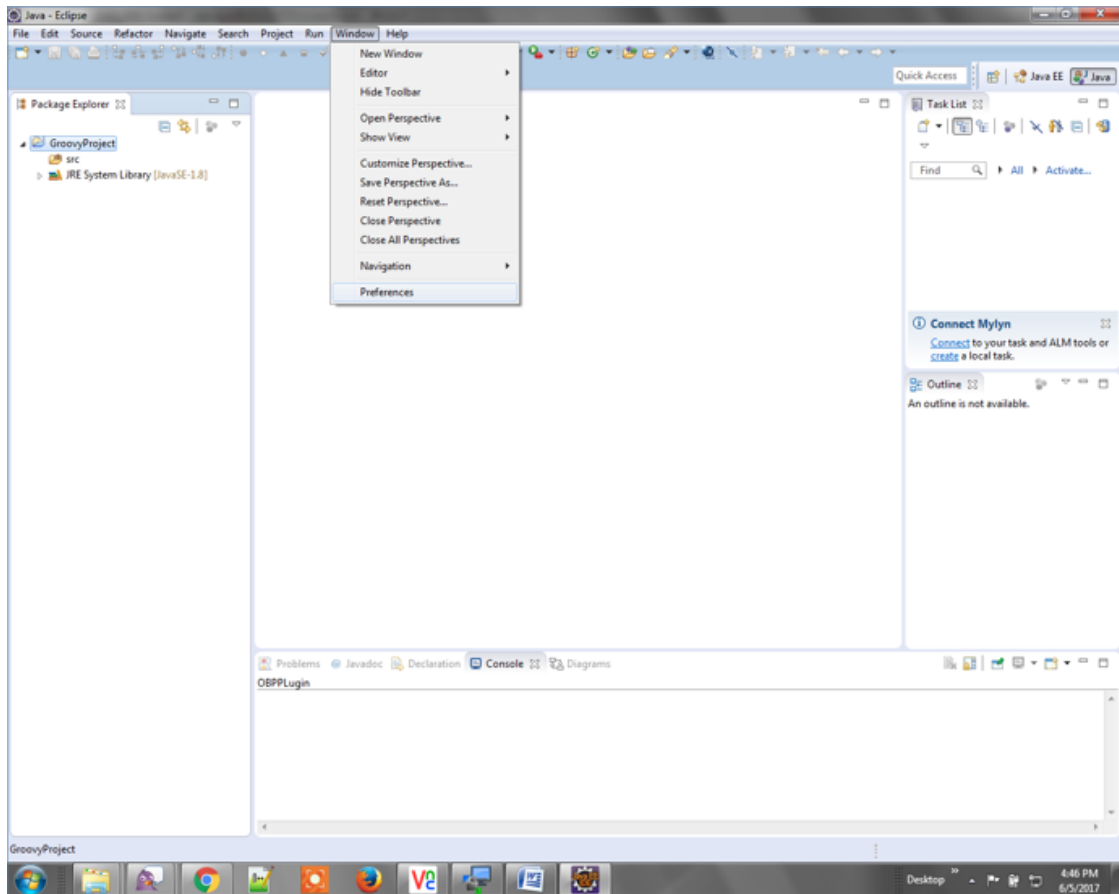


Figure 4–2 Preferences Dialog Box - OBP Service Plugin

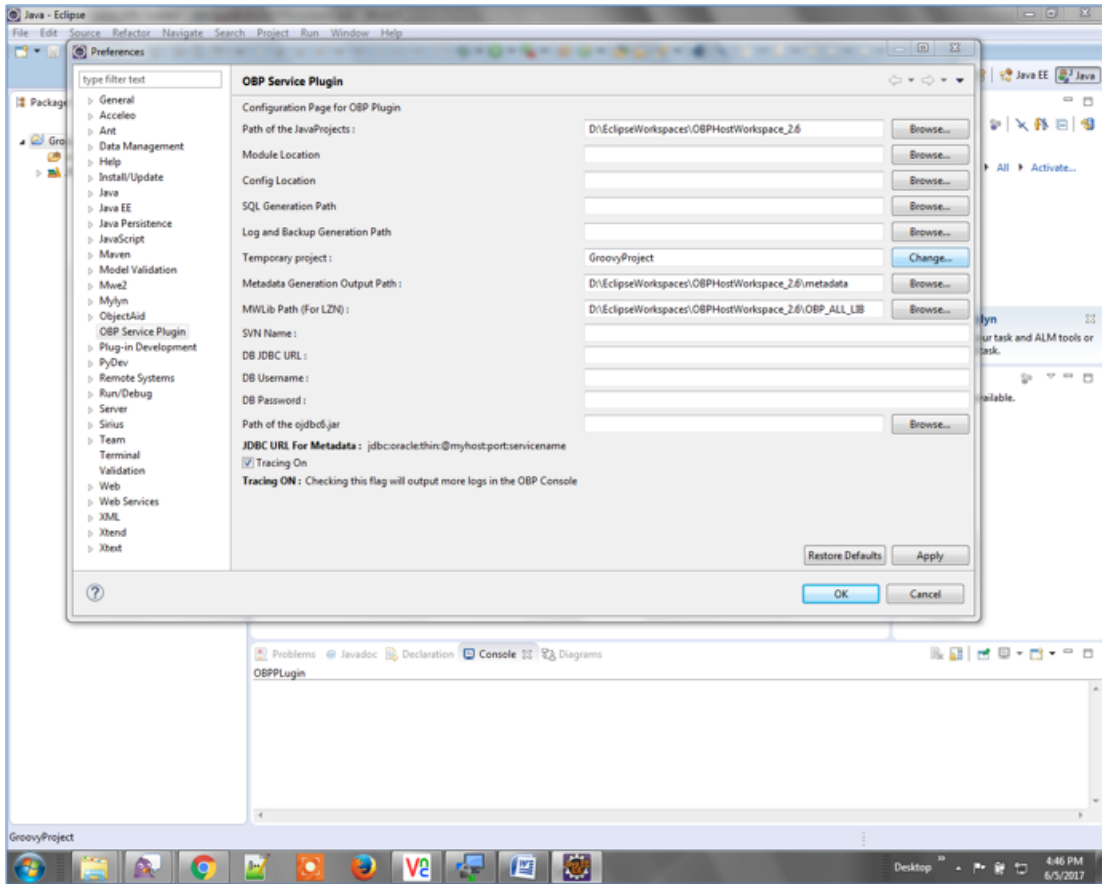
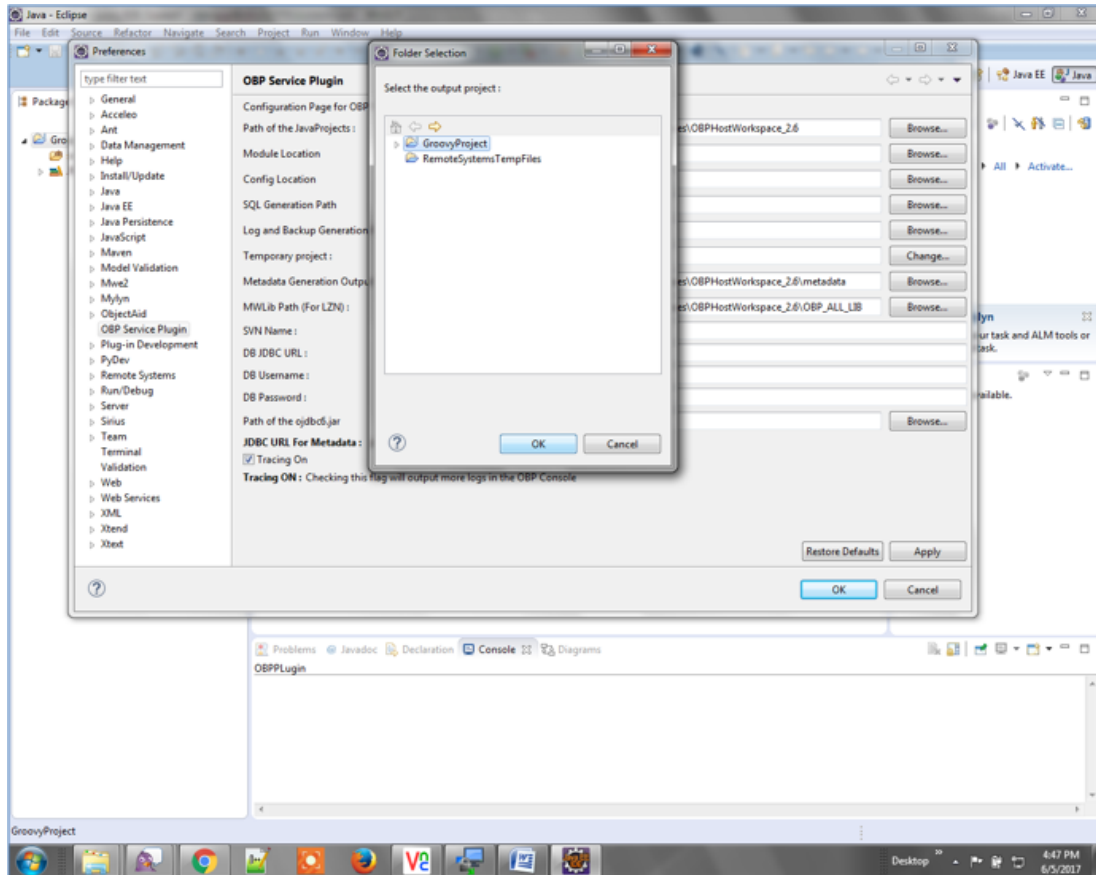
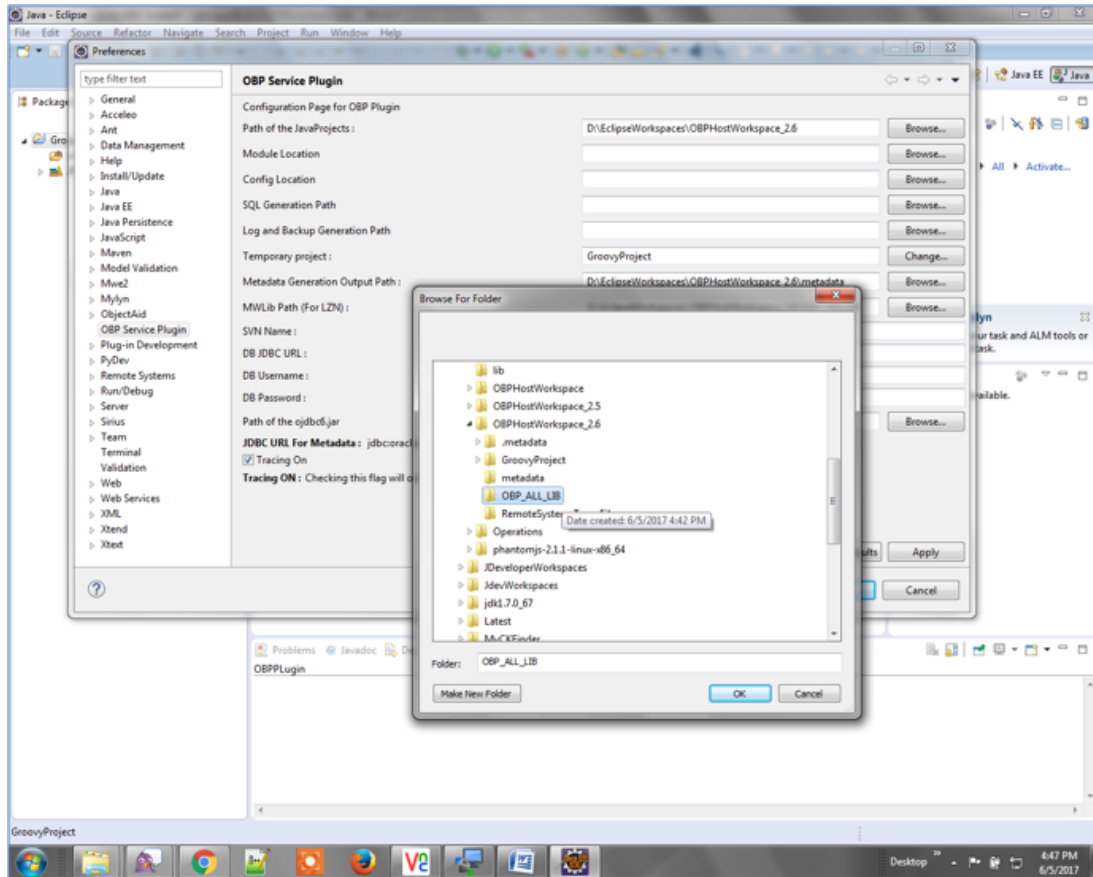


Figure 4–3 Folder Selection



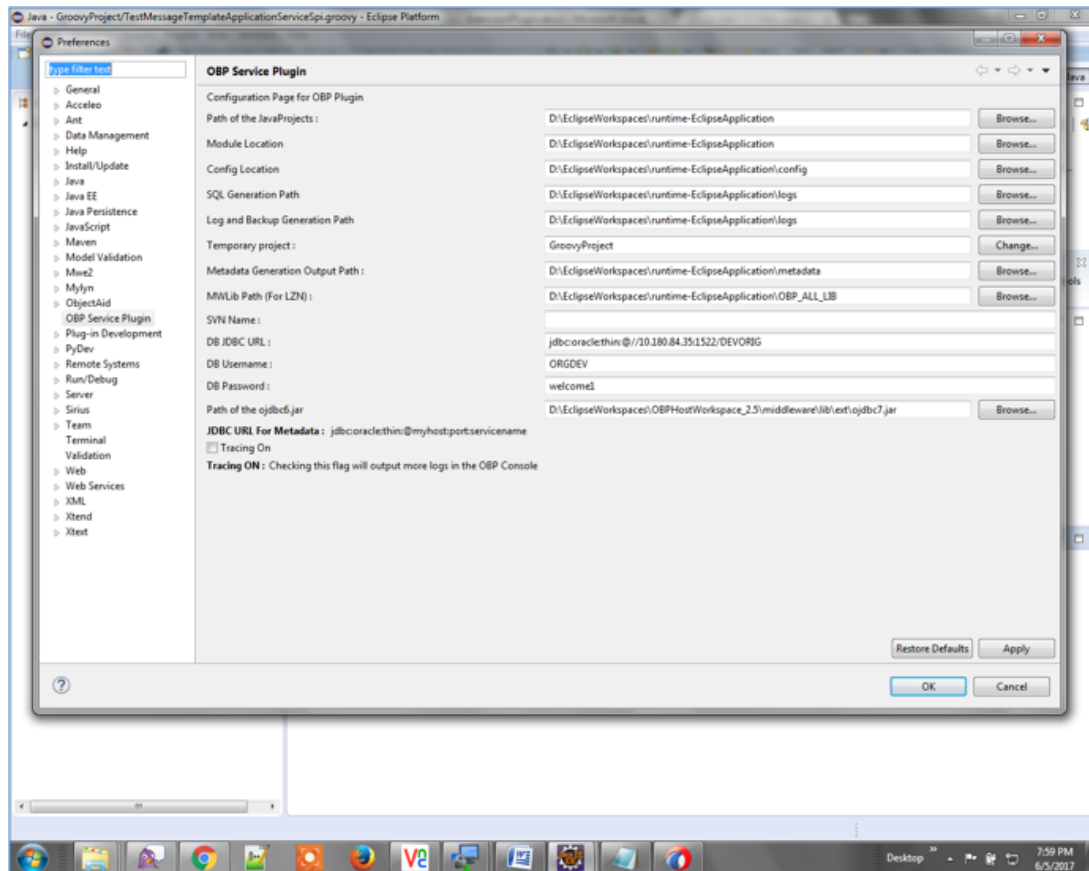
2. The parameter “Temporary Project” has to be configured to point to the base project where the Groovy Extension Files have to be saved.

Figure 4–4 Browse For Folder



3. The parameter “MWLib Path” has to be configured to point to the directory where all the OBP Host jar files have been kept.

Figure 4–5 Configuring MWLib Path Parameter



4. The rest of the OBP Service Plugin parameters can be configured as usual.

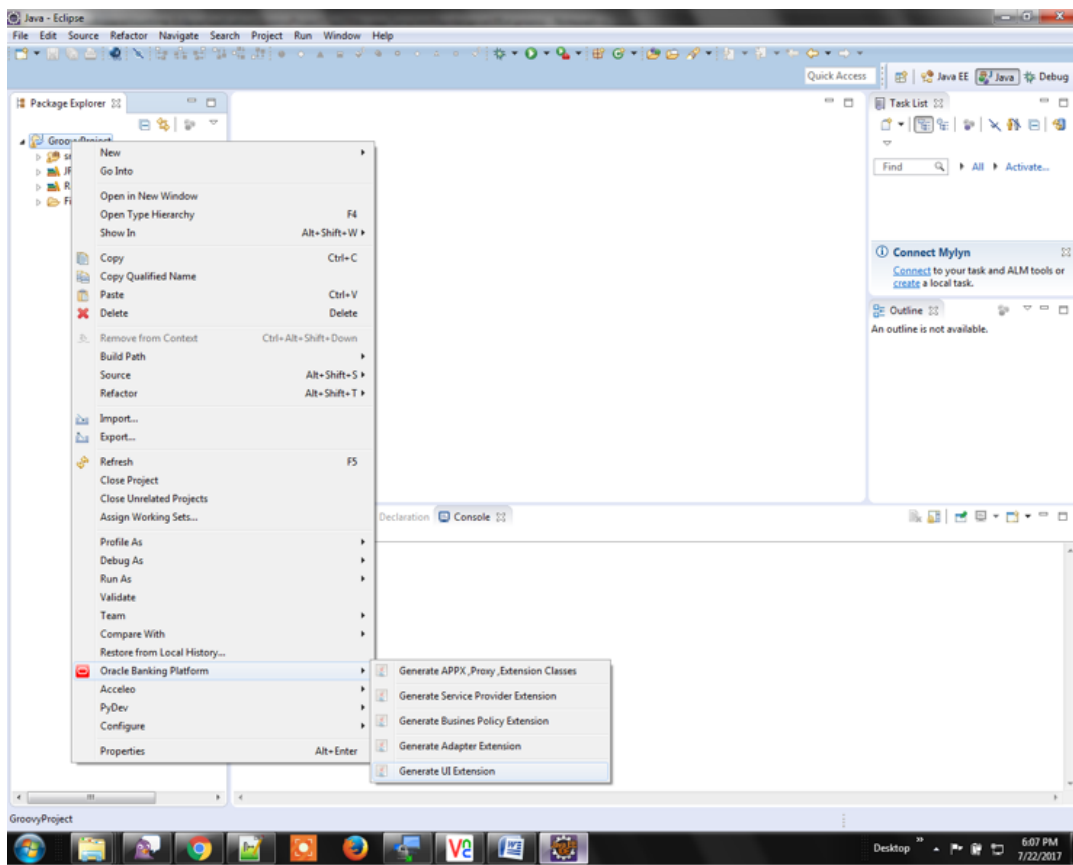
## 4.2 Support for UI Extension

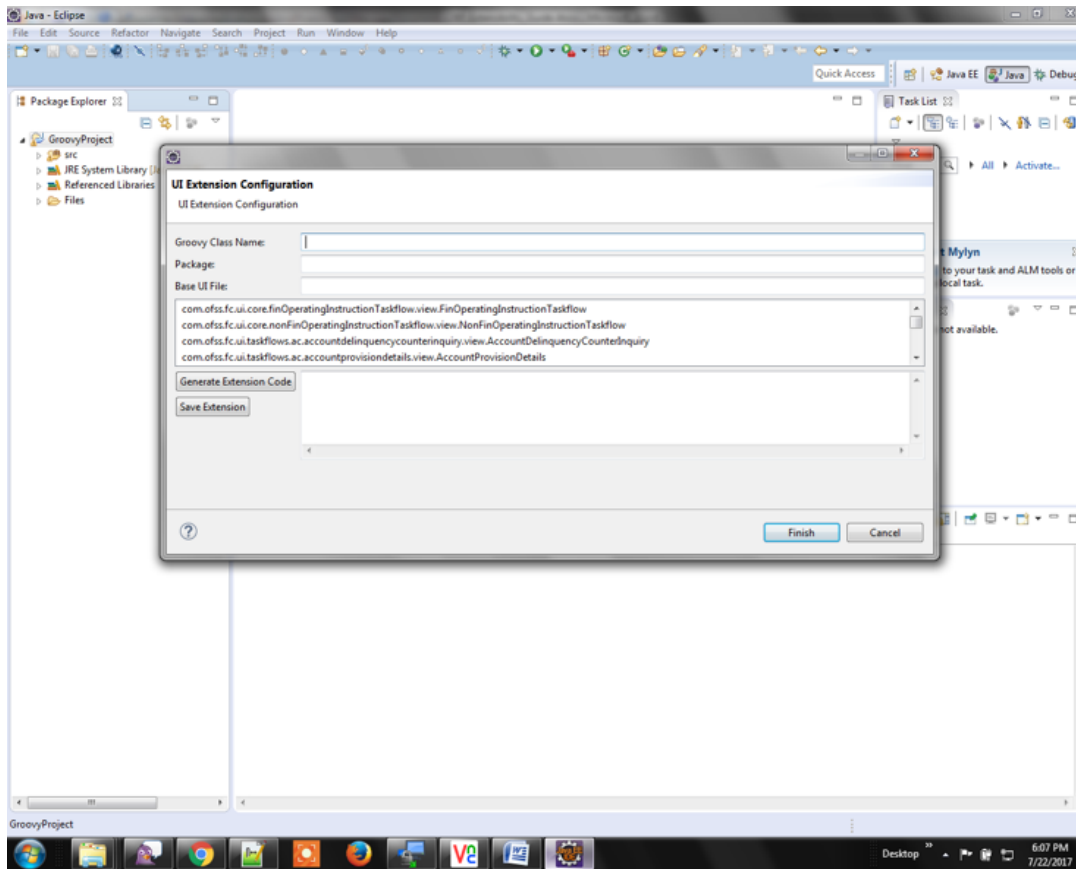
### 4.2.1 Generate UI Extension

The parameter “Temporary Project” has been configured to point to the base project where the Groovy Extension Files have to be saved. The parameter “MWLib Path” has to be configured to point to the directory where all the OBP Host jar files have been kept. Here we have to place all jar files in “obp.ui.domain” deployed in UI Server. Also it is mandatory to place `adf.jar`, `adf-richclient-api-11.jar` and `javax.faces.jsf-api.jar` available in JDeveloper 12C Installation Location.

1. Right click on this project and select Oracle Banking Platform > Generate UI Extension.

**Figure 4–6 Generate UI Extension**

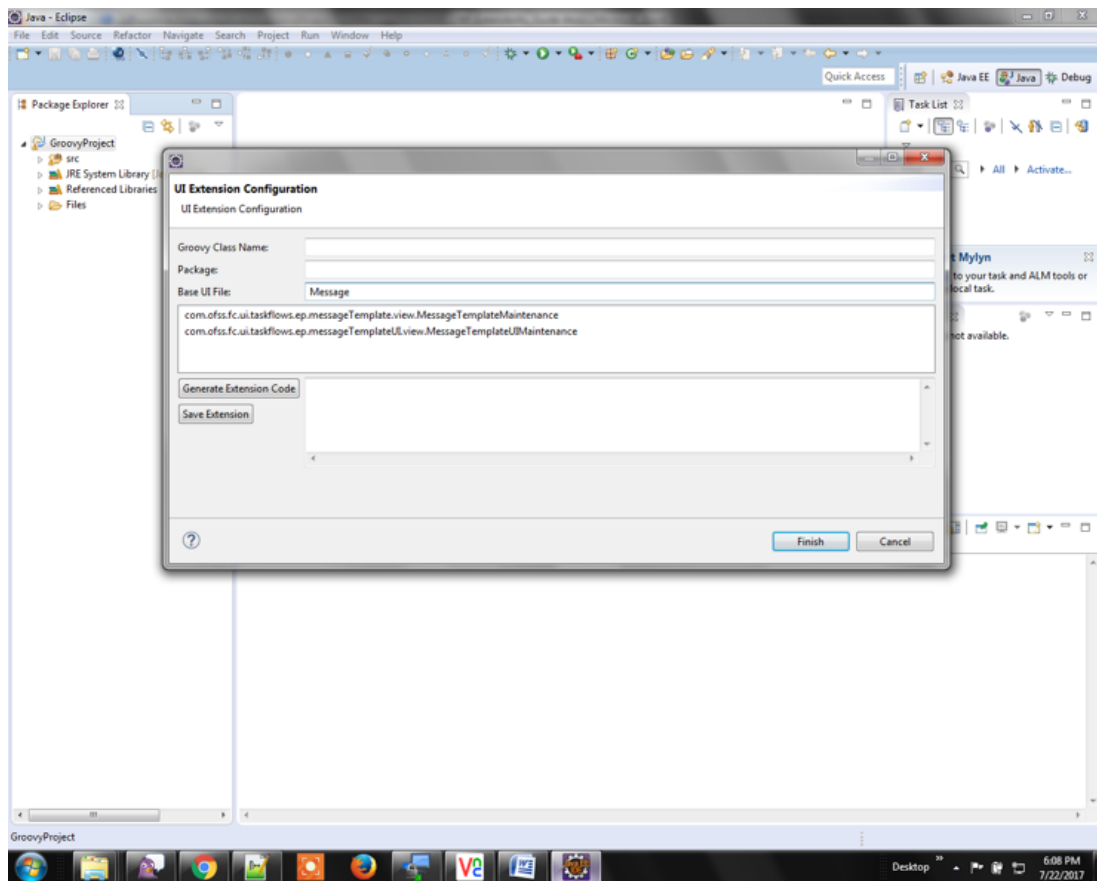


**Figure 4–7 UI Extension Configuration**

The above wizard appears with a list of Base UI Files.

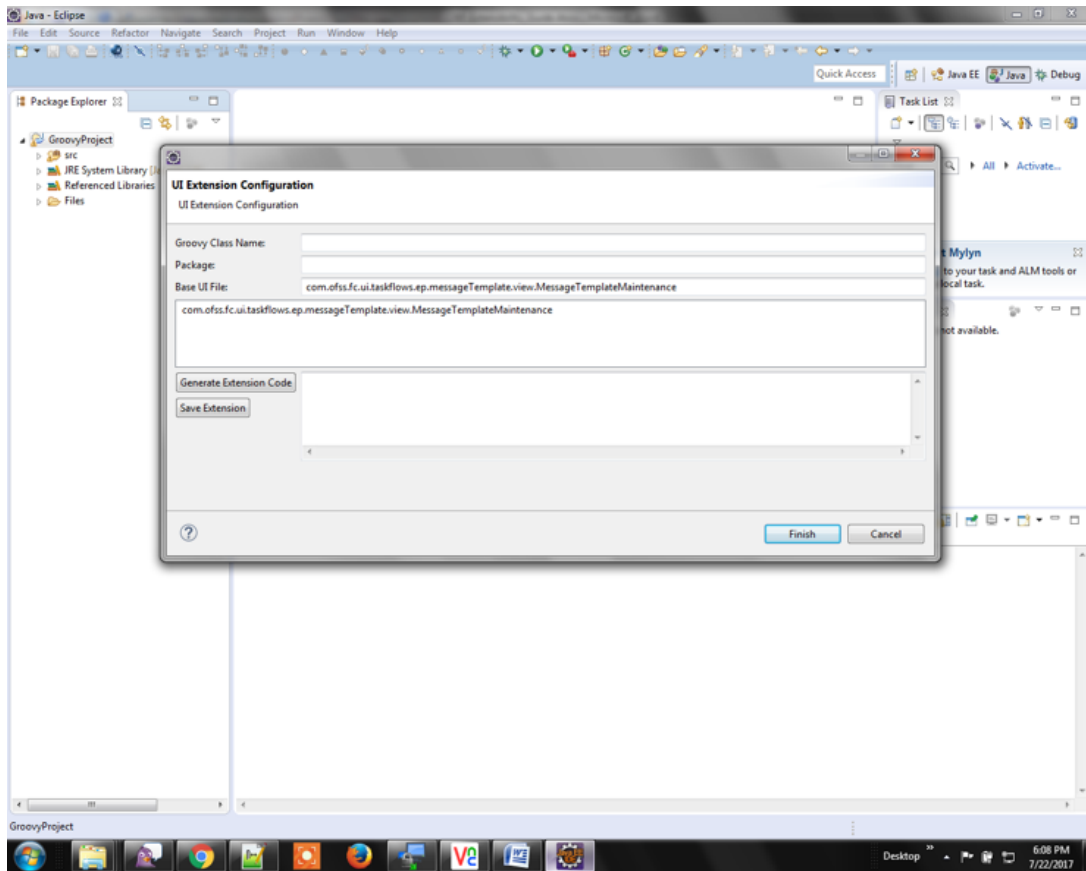
2. Enter a search keyword to filter the list for the required Base UI file.

**Figure 4–8 UI Extension Configuration - Search for Base UI File**



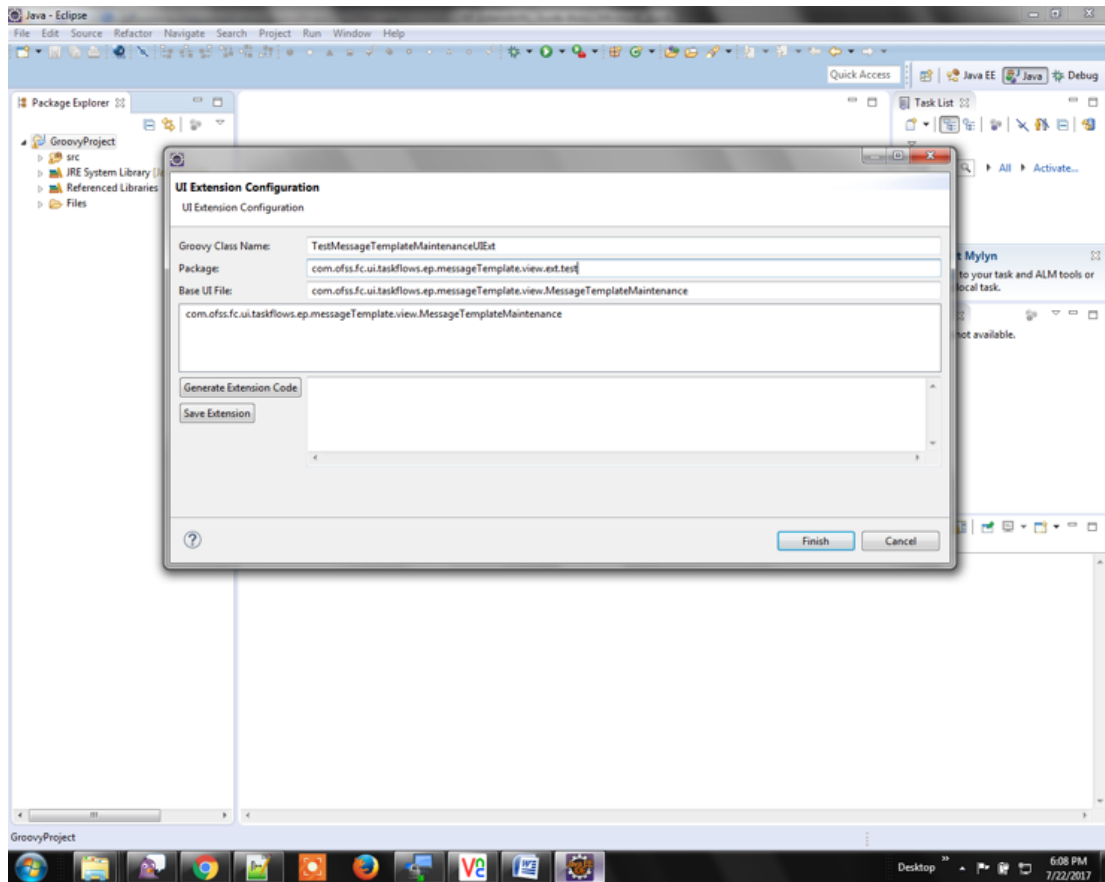
3. Select the filtered Base UI file from the list such that it appears as the input for Base UI file.



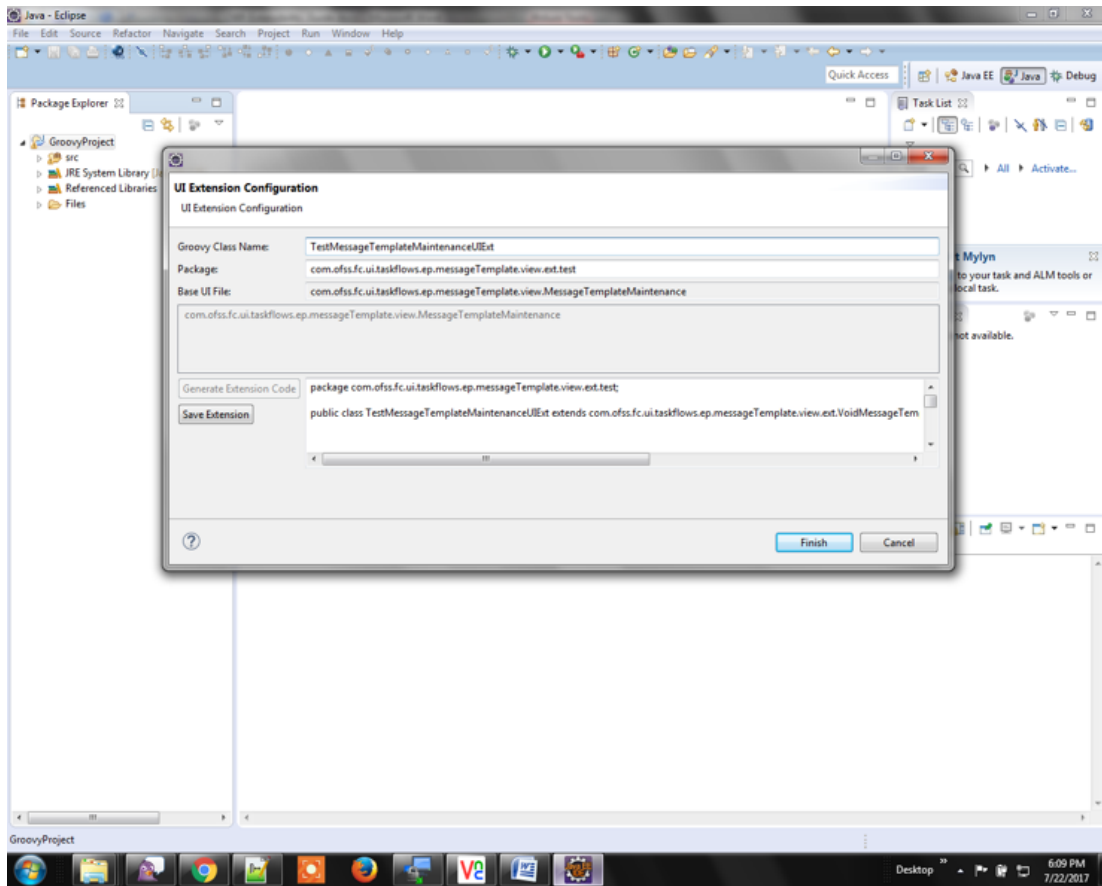
**Figure 4–9 UI Extension Configuration - Base UI File Selected**

4. Appropriately set values for the Extension Class Name and package.

**Figure 4–10 UI Extension Configuration - Class Name and Package**

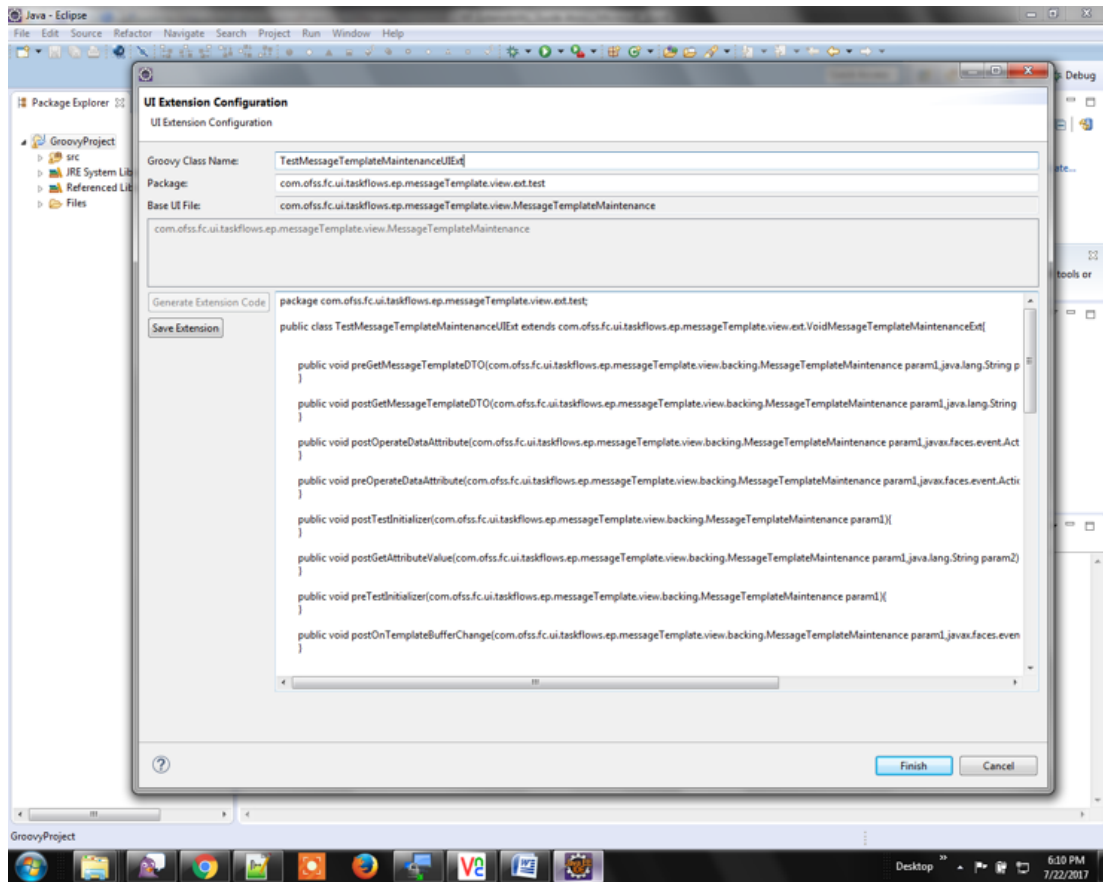


5. Click on “Generate Extension Code”.

**Figure 4–11 UI Extension Configuration - Generate Extension Code**

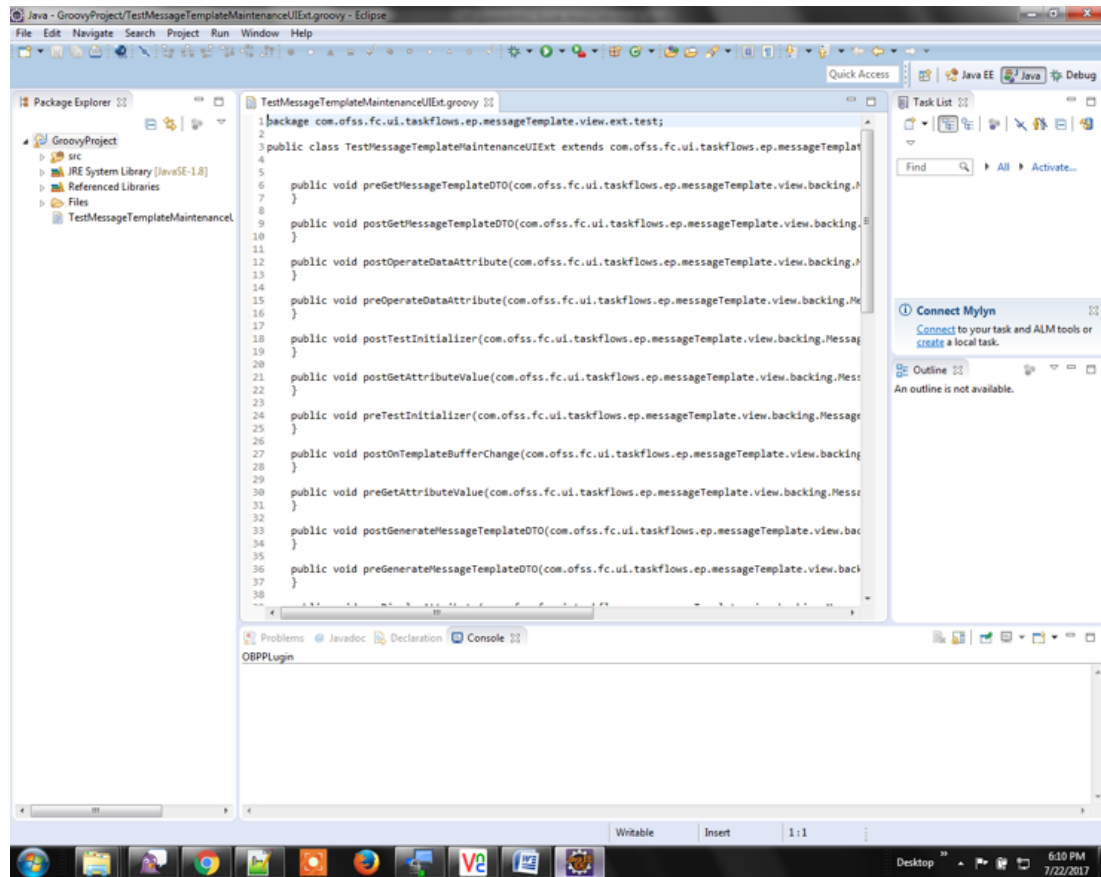
The code gets generated with the extension hooks.

Figure 4–12 UI Extension Configuration - Code Generated with Extension Hooks



6. Click on “Save Extension” and “Finish”.

Figure 4–13 Project - UI Extension Created

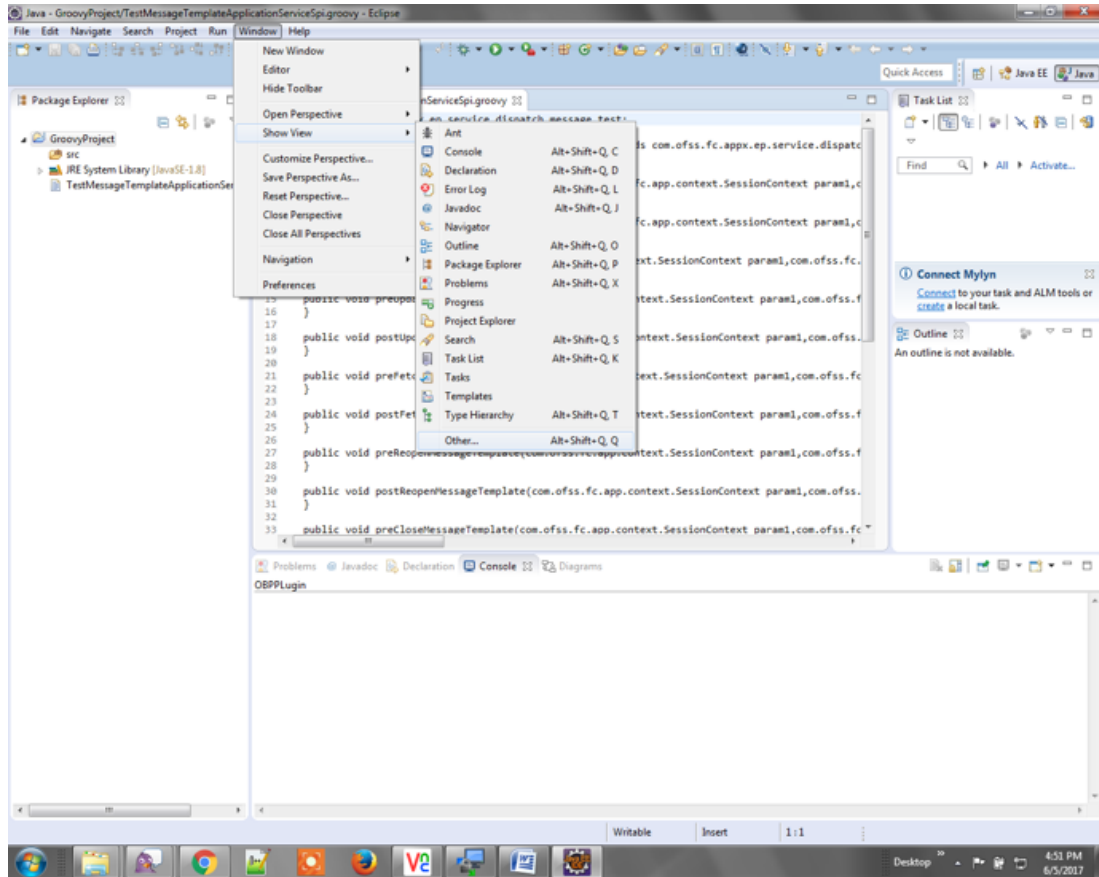


The extension gets saved in the project created to contain the Extension classes generated through the plugin. We can add all the code required in the pre and post hooks for extensibility of the base UI File.

## 4.2.2 Configure OBP Extensibility Server Explorer - View

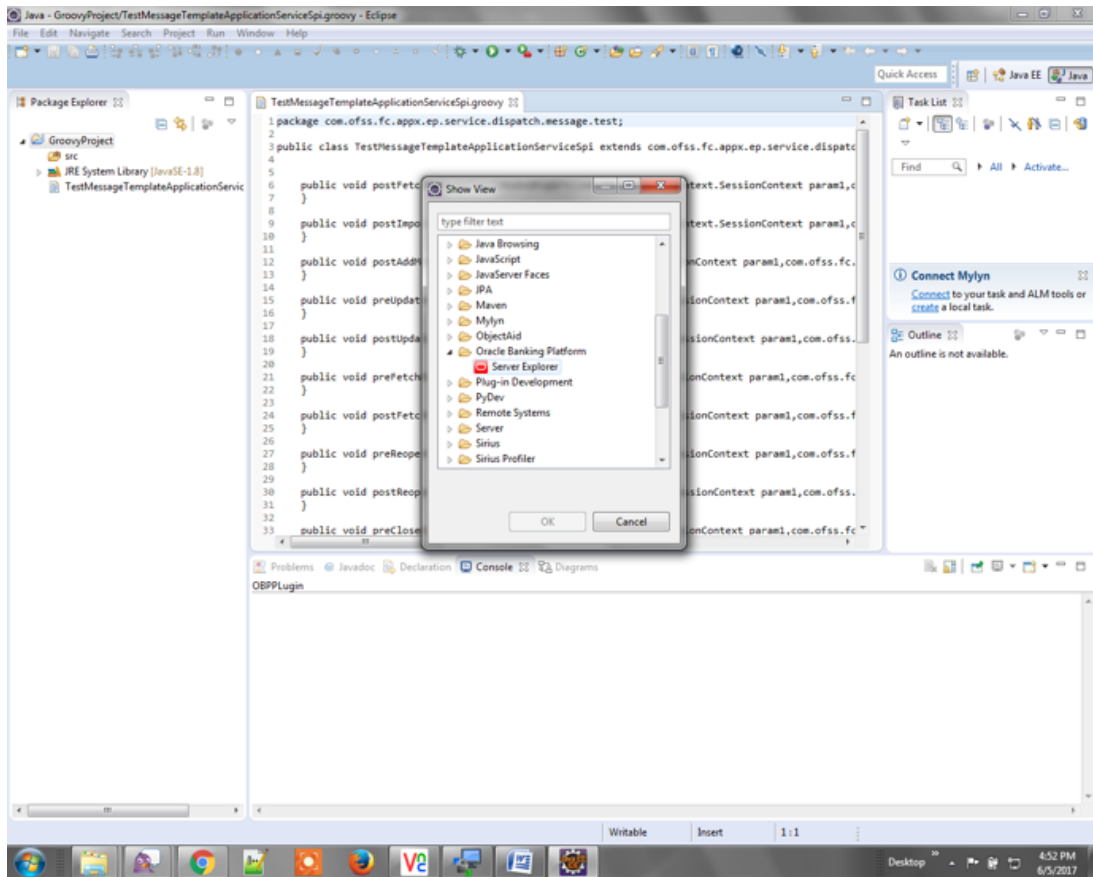
1. In eclipse click on Window > Show View > Other.

Figure 4–14 Configuring OBP Extensibility Server Explorer



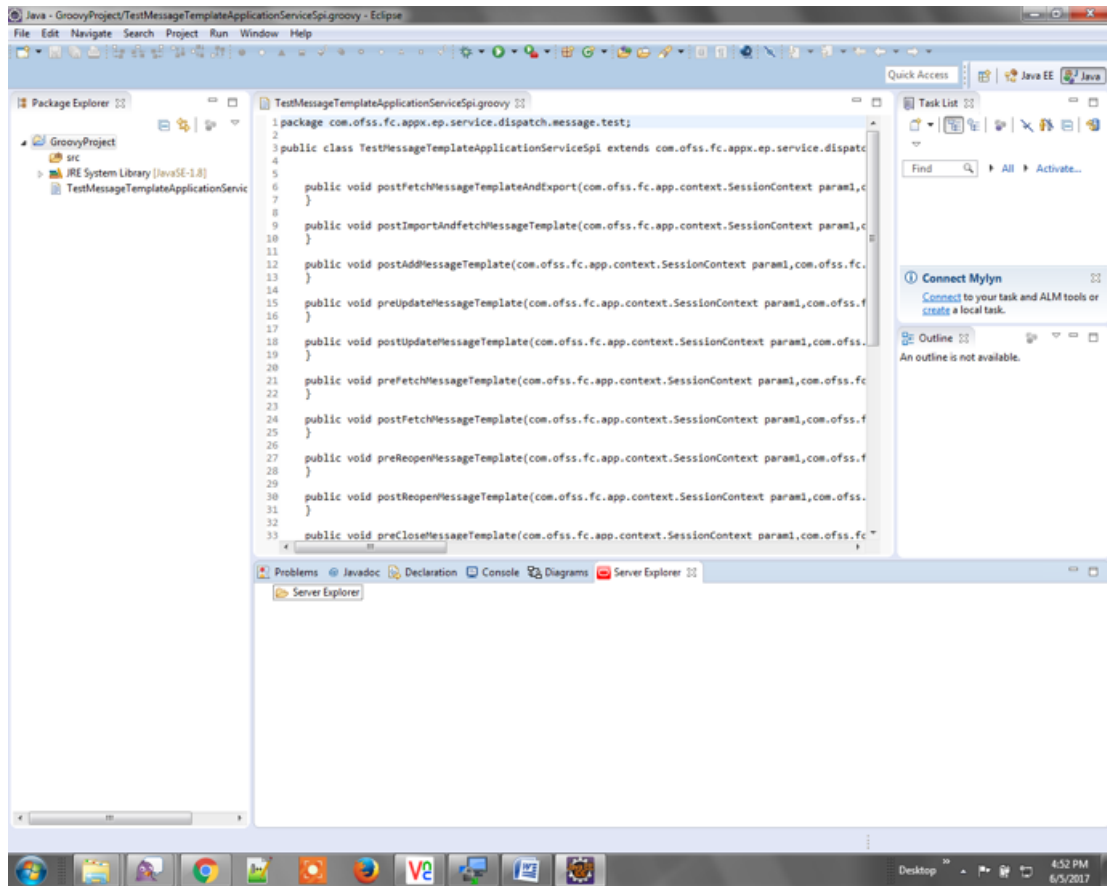
2. Click on Oracle Banking Platform > Server Explorer.

Figure 4–15 Create Server Explorer



The “ServerExplorer” view tab opens up.

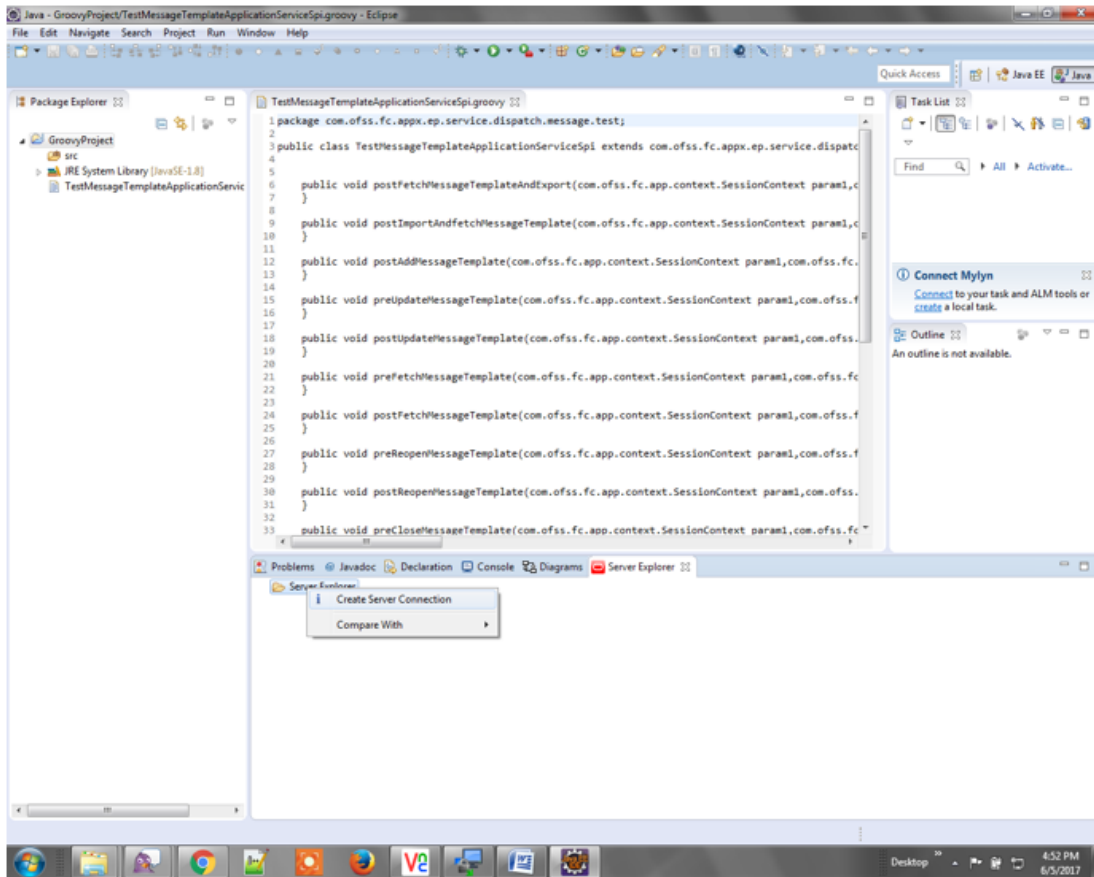
Figure 4–16 Server Explorer View tab



3. Right click on Server Explorer and click “Create Server Connection”.

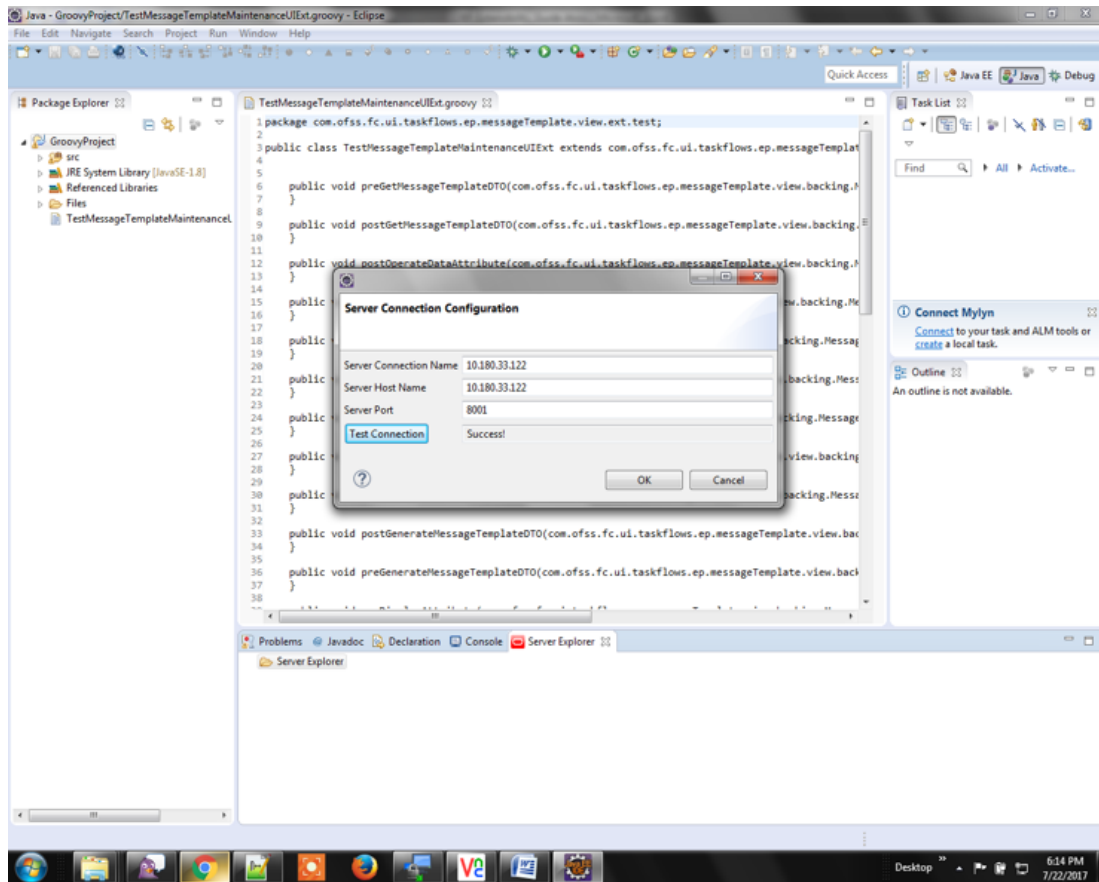


Figure 4–17 Server Explorer - Create Server Connection



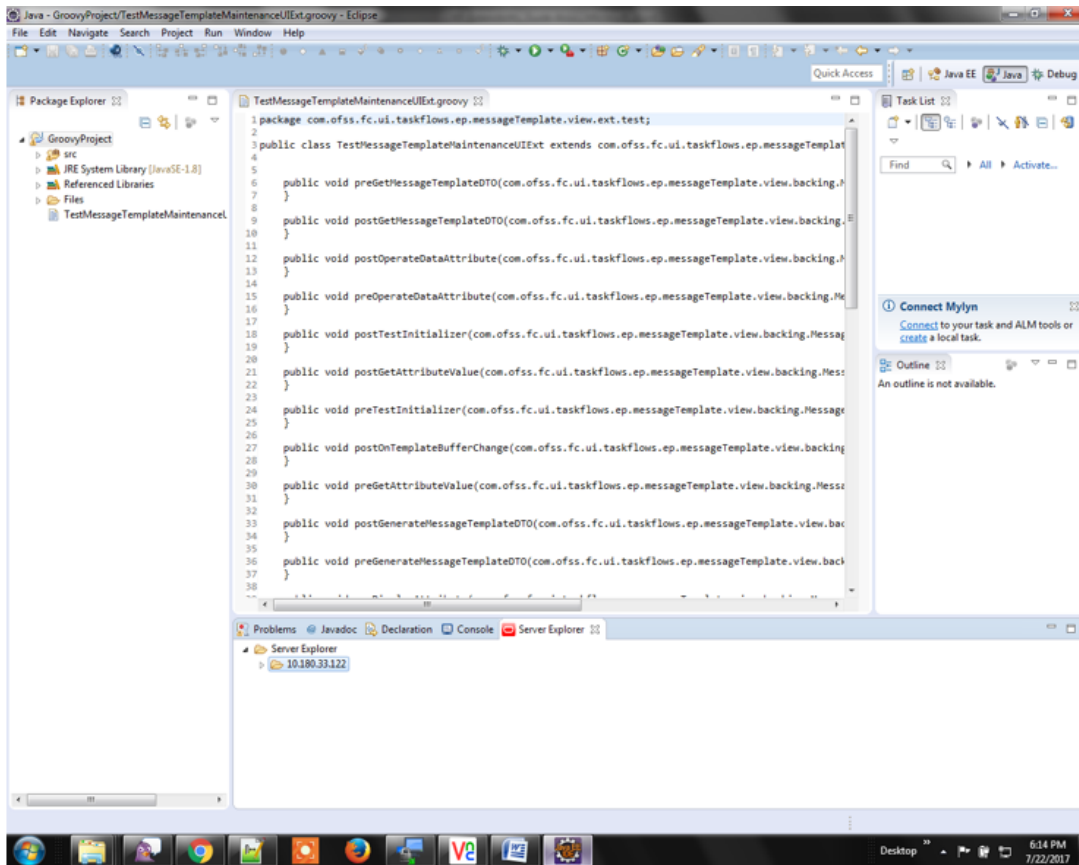
4. Provide values for Connection, name, ip, and port, and test connection and click OK.

Figure 4–18 Server Connection Configuration



The configured server appears as a child of “Server Explorer”. The configured server also lists all Application Service SPI Extensions and all Business Policy Extensions and all Adapter Extensions and all UI Extensions already deployed in server.

Figure 4–19 Server Configured



## 4.2.3 Exposed Webservice for UI Extensions

Figure 4–20 ExtensionApplicationServiceSpi Web Service for UI Extensions

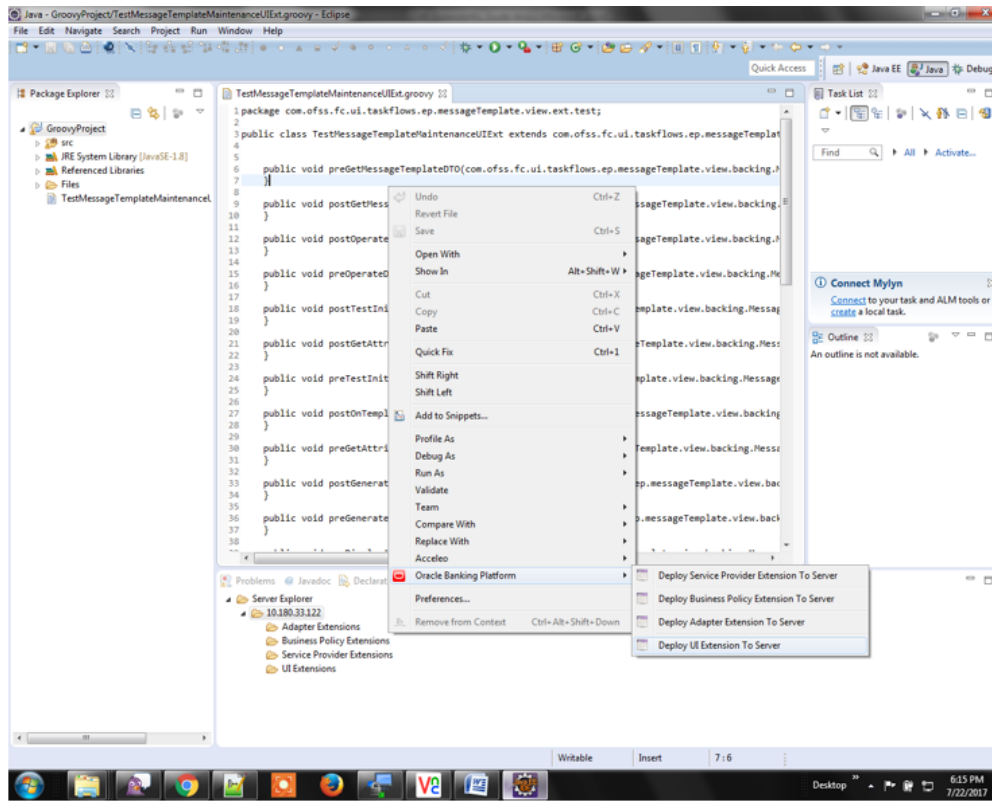


“ExtensionApplicationServiceSpi” is the web service that exposes the web services to add UI extensions, fetch UI extensions and delete UI extensions. These services are used by the eclipse plugin to deploy extensions, fetch extensions and undeploy extensions at runtime.

## 4.2.4 Deploy UI Extension to Server

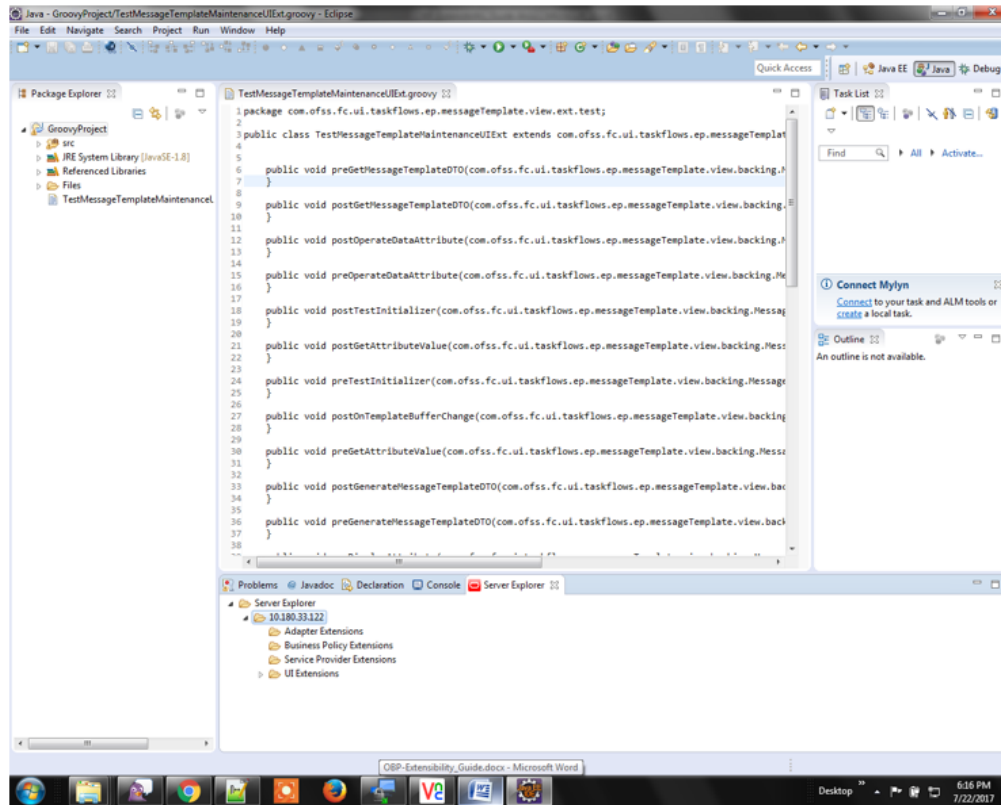
1. Right Click on the Extension Class in the Package Explorer or the Code Editor and click on Oracle Banking Platform > Deploy UI Extension To Server.

Figure 4–21 Deploy UI Extension to Server



2. It is compulsory to have selected the Server under “Server Explorer” in which the deployment has to occur.

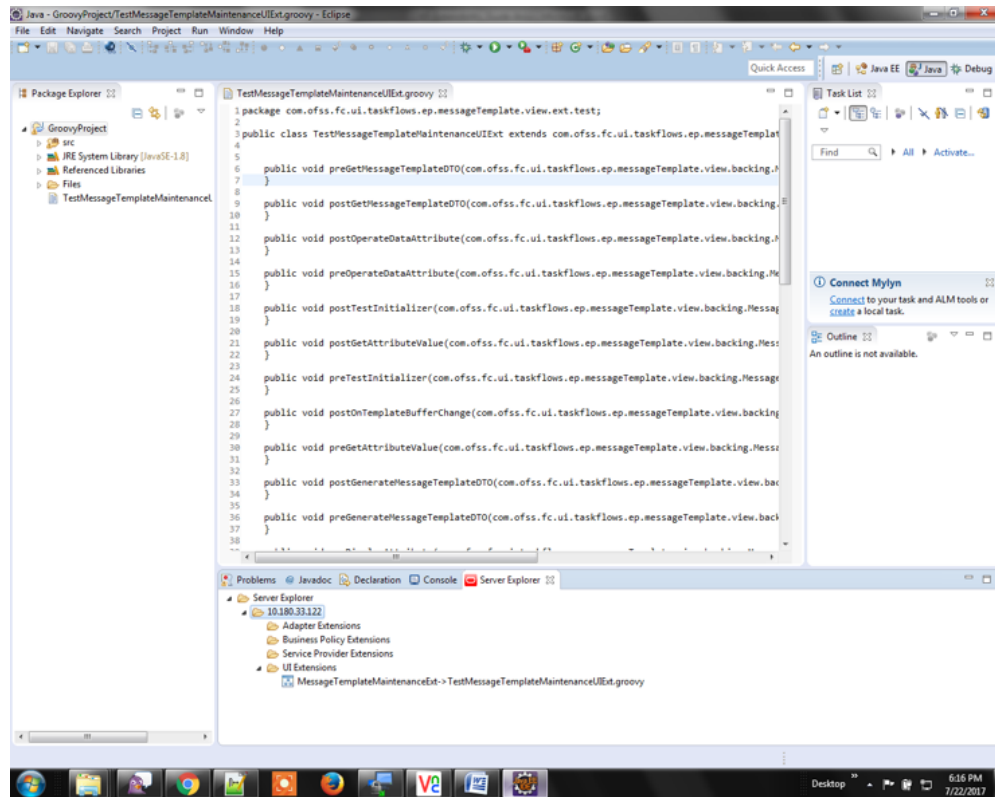
Figure 4–22 Select Server Explorer to Deploy UI Extension



The Extension gets deployed in the server.

3. The Extension gets deployed in the server and appears under the appropriate child of Server configured under “Server Explorer”.

Figure 4–23 UI Extensions Deployed to Server Explorer



## 4.2.5 Database Inserts: UI Extension Deployment

Figure 4–24 Single Record View - UI Extension Deployment

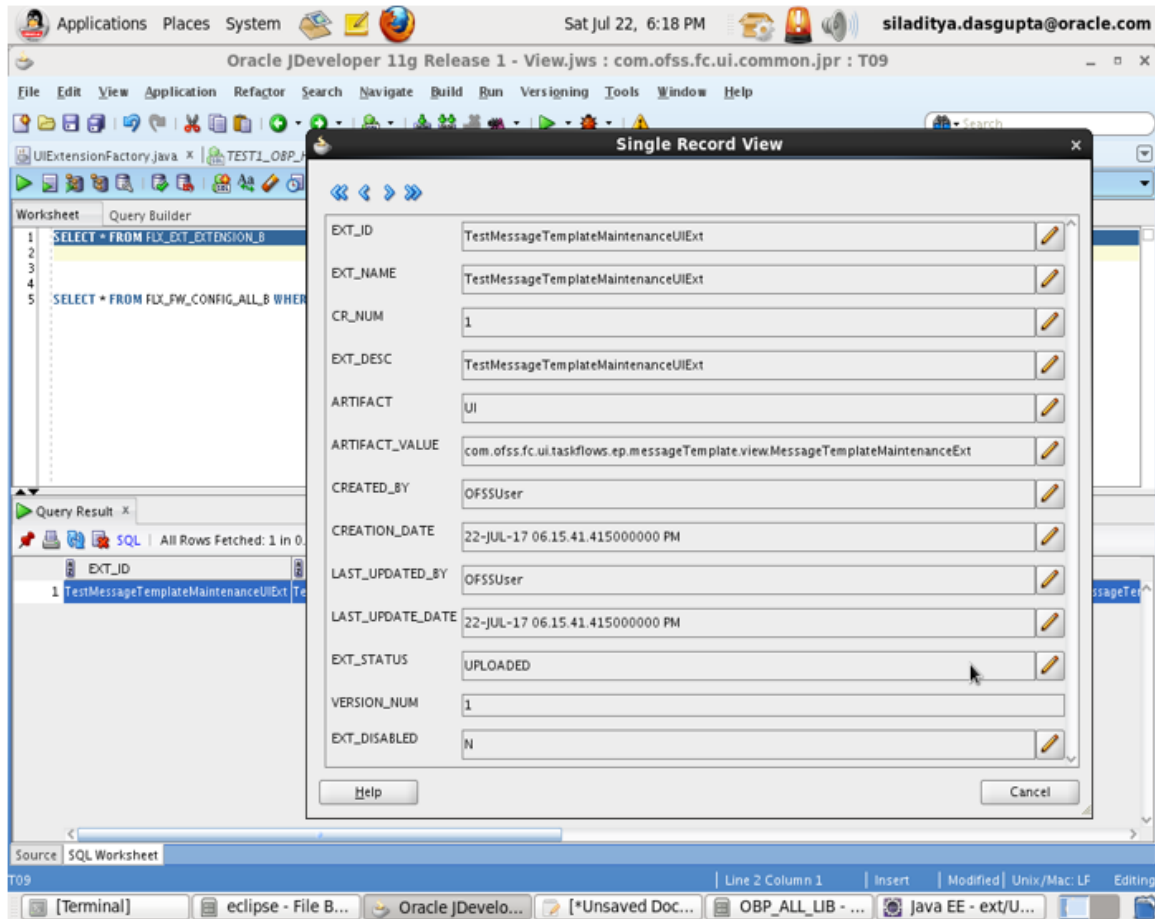




Figure 4–25 Single Record View - UI Extension Deployment

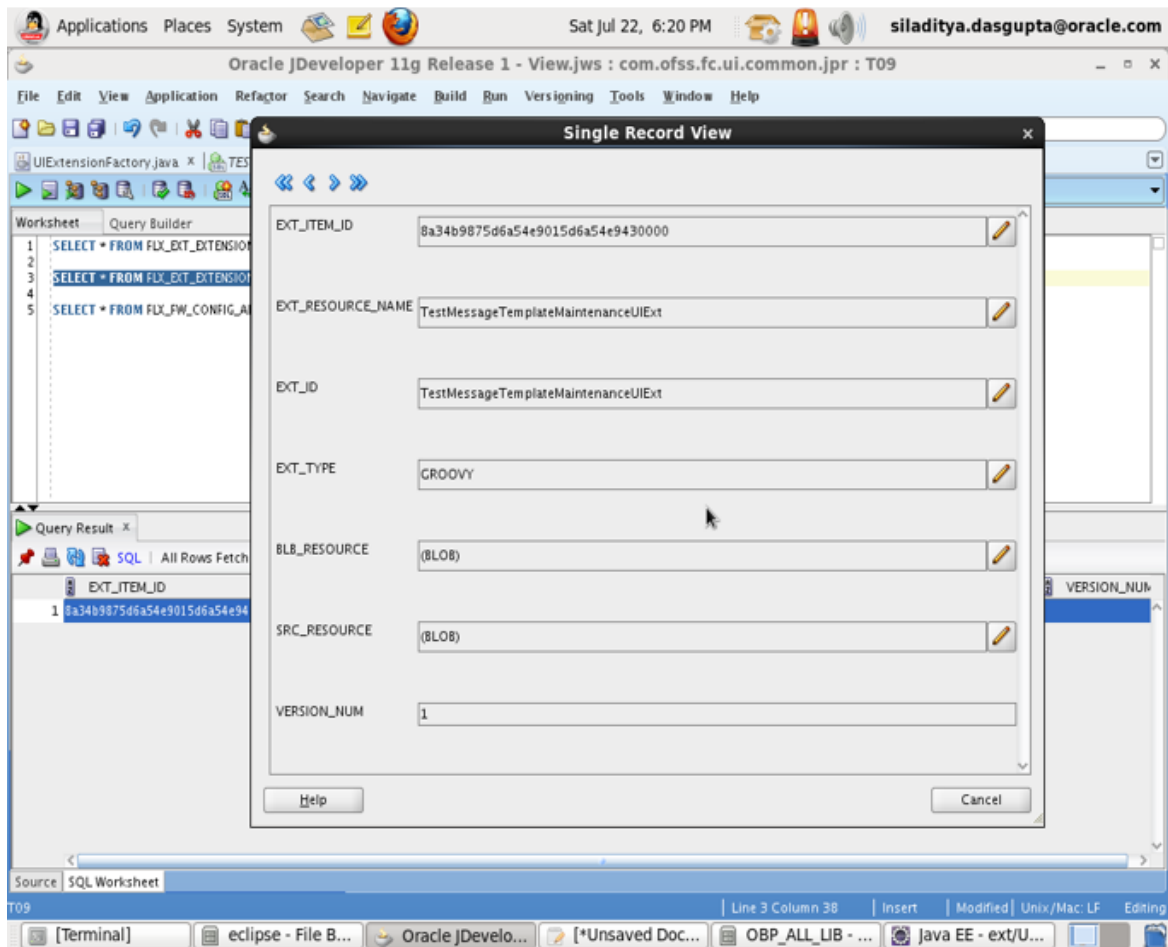


Figure 4–26 View Value - UI Extension Deployment

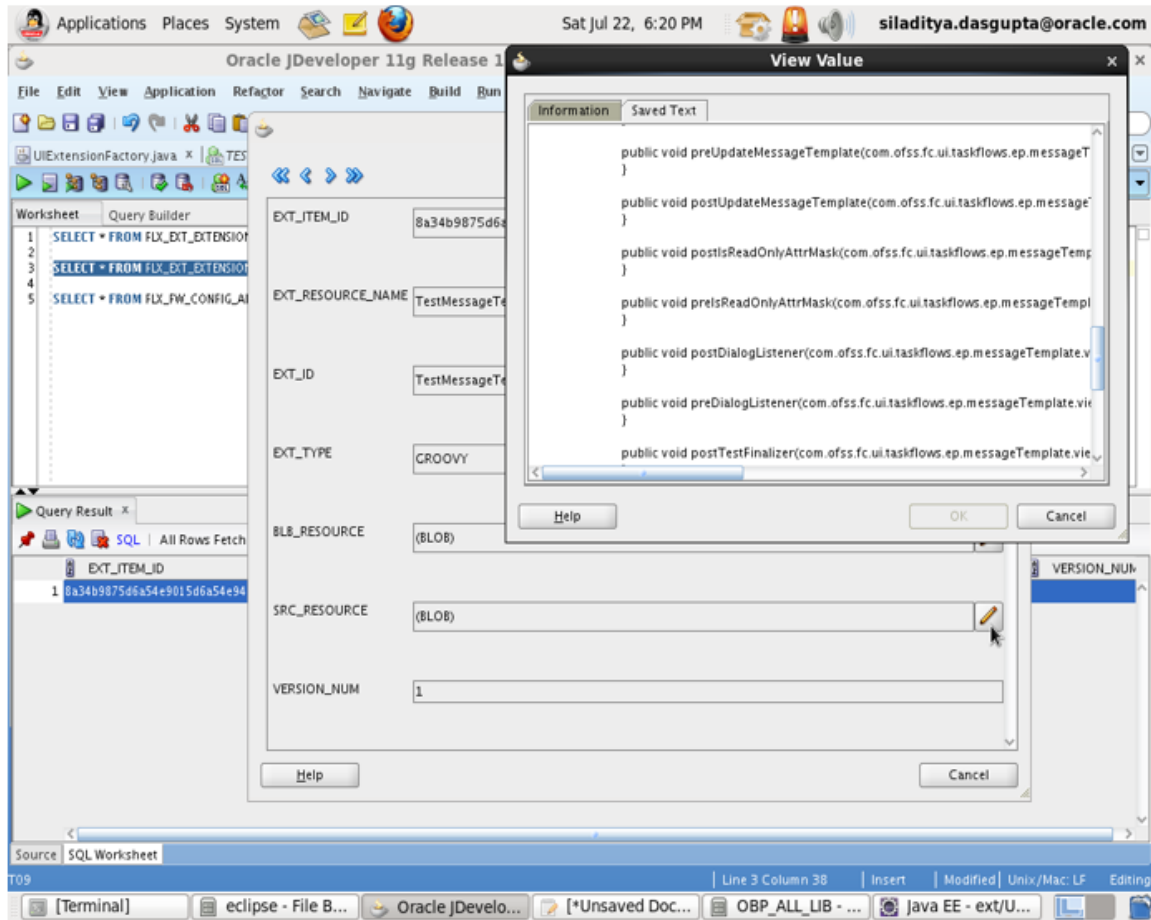
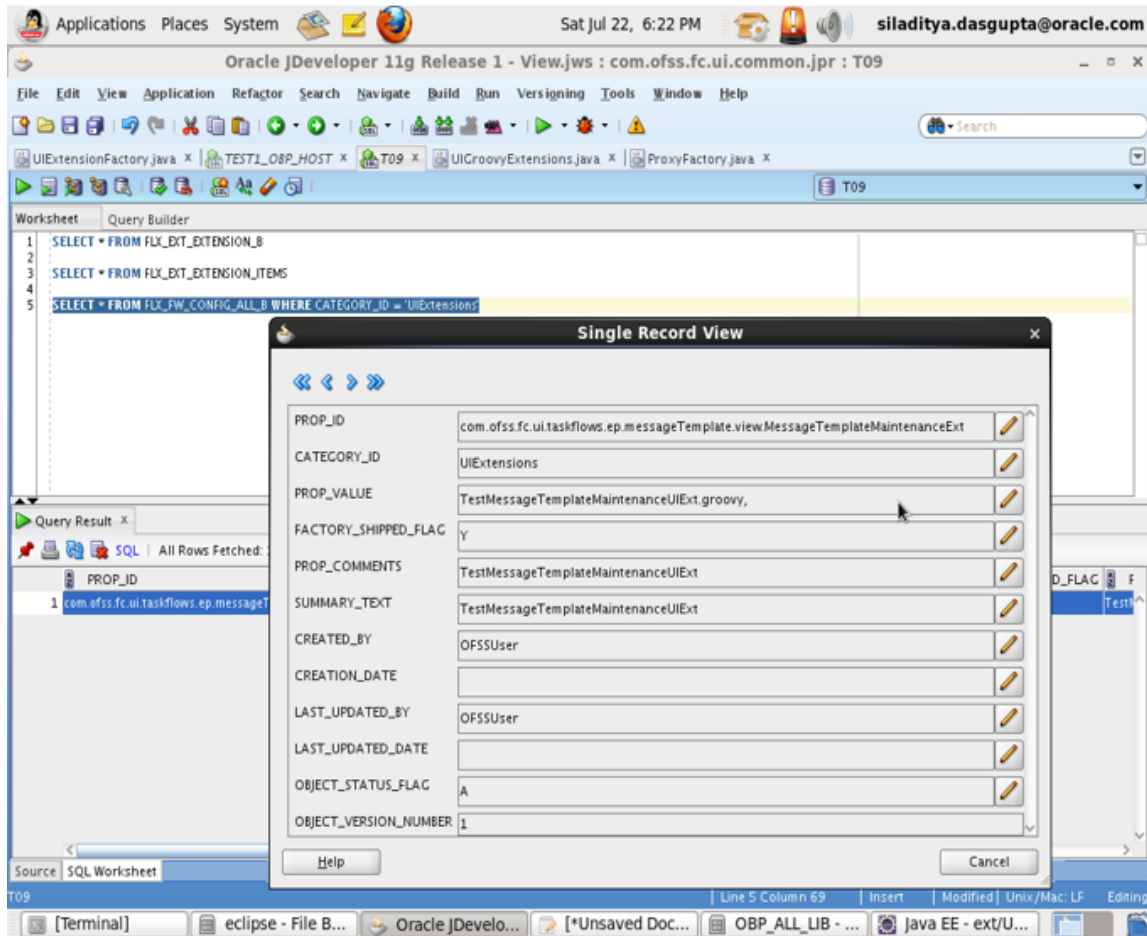


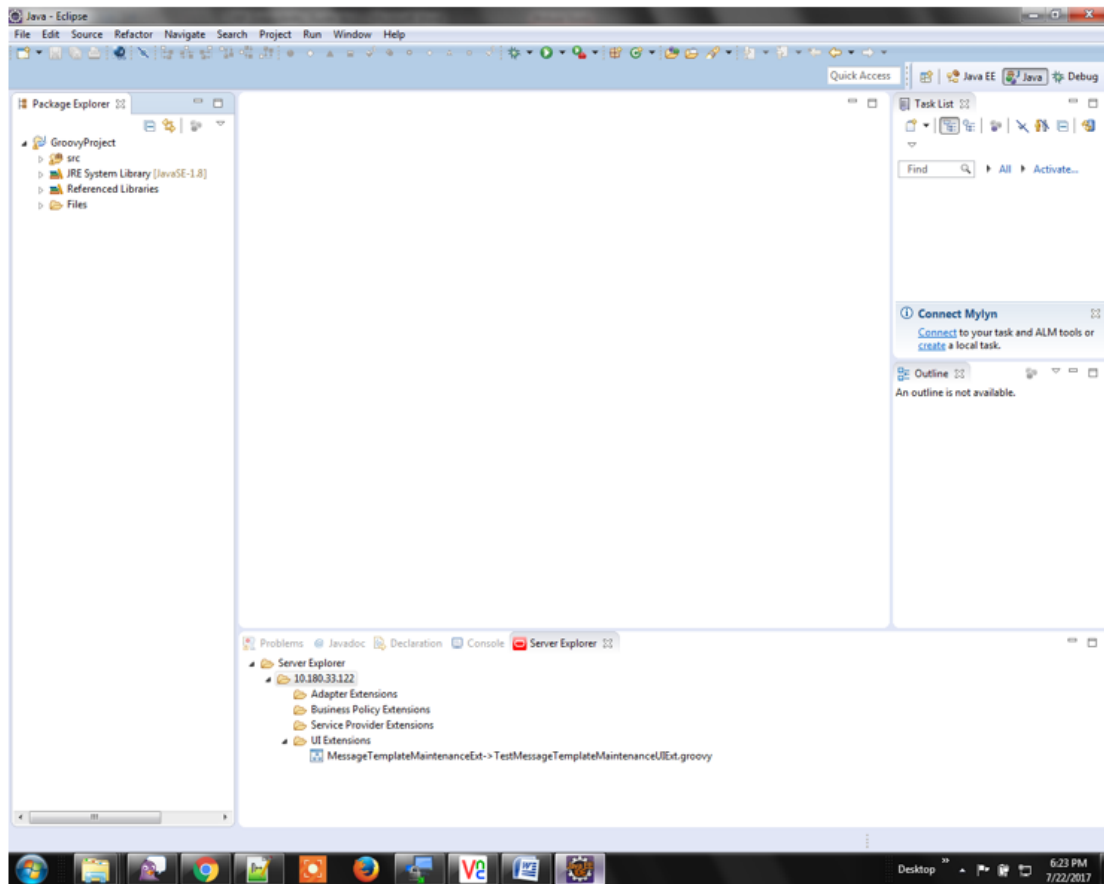
Figure 4–27 Single Record View - UI Extension Deployment



### 4.2.6 Fetching Deployed UI Extension

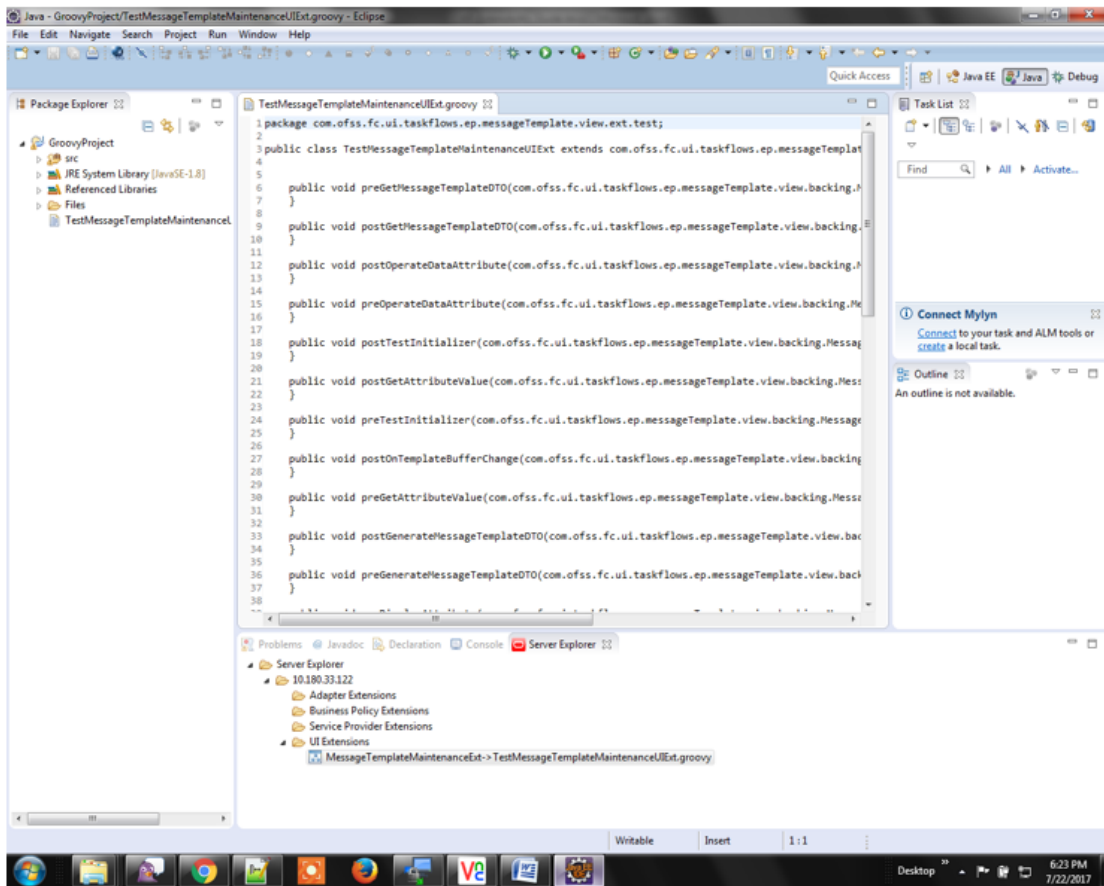
1. For the purpose of fetching a deployed Extension from the Server, we can click on the appropriate Extension under the appropriate child under “Server Explorer”.

**Figure 4–28 Fetch Deployed UI Extension**



2. The extension gets saved in the project created to contain the Extension classes generated through the plugin. Also the Extension code opens up in the Code Editor.

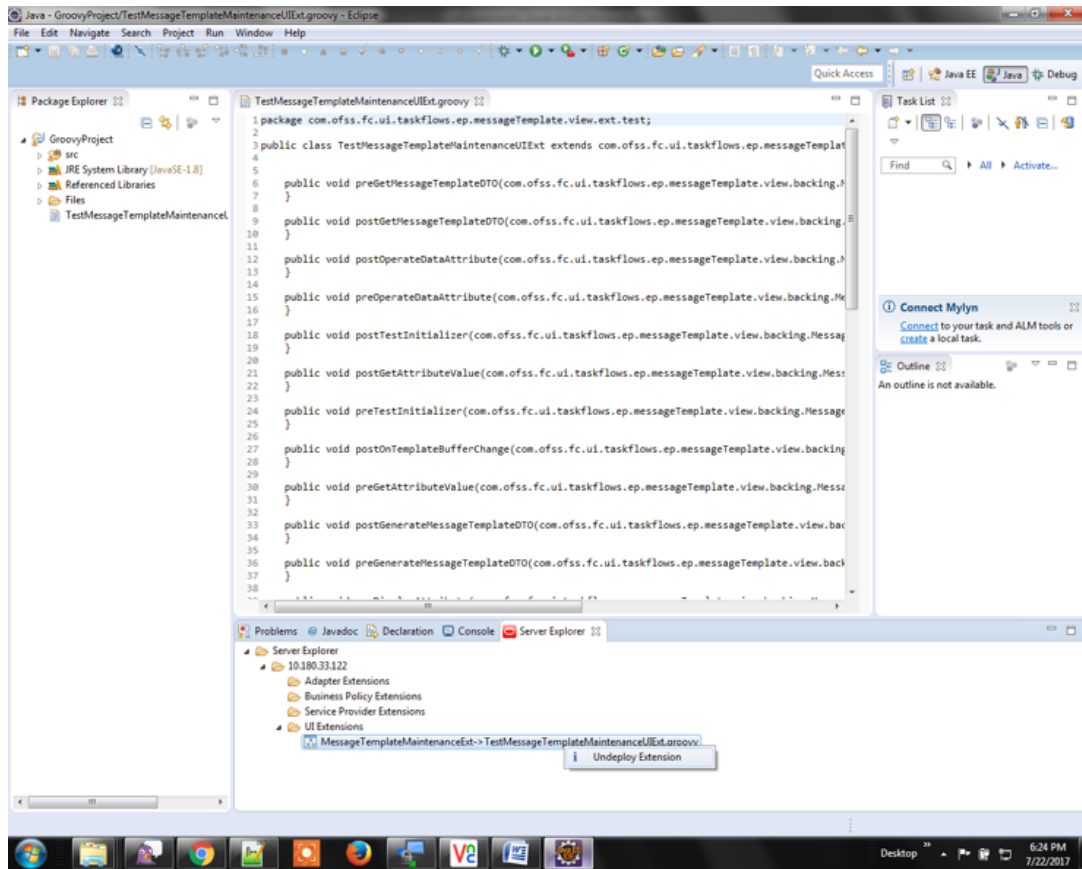
Figure 4–29 Extension Saved in Project



### 4.2.7 Undeploying UI Extension

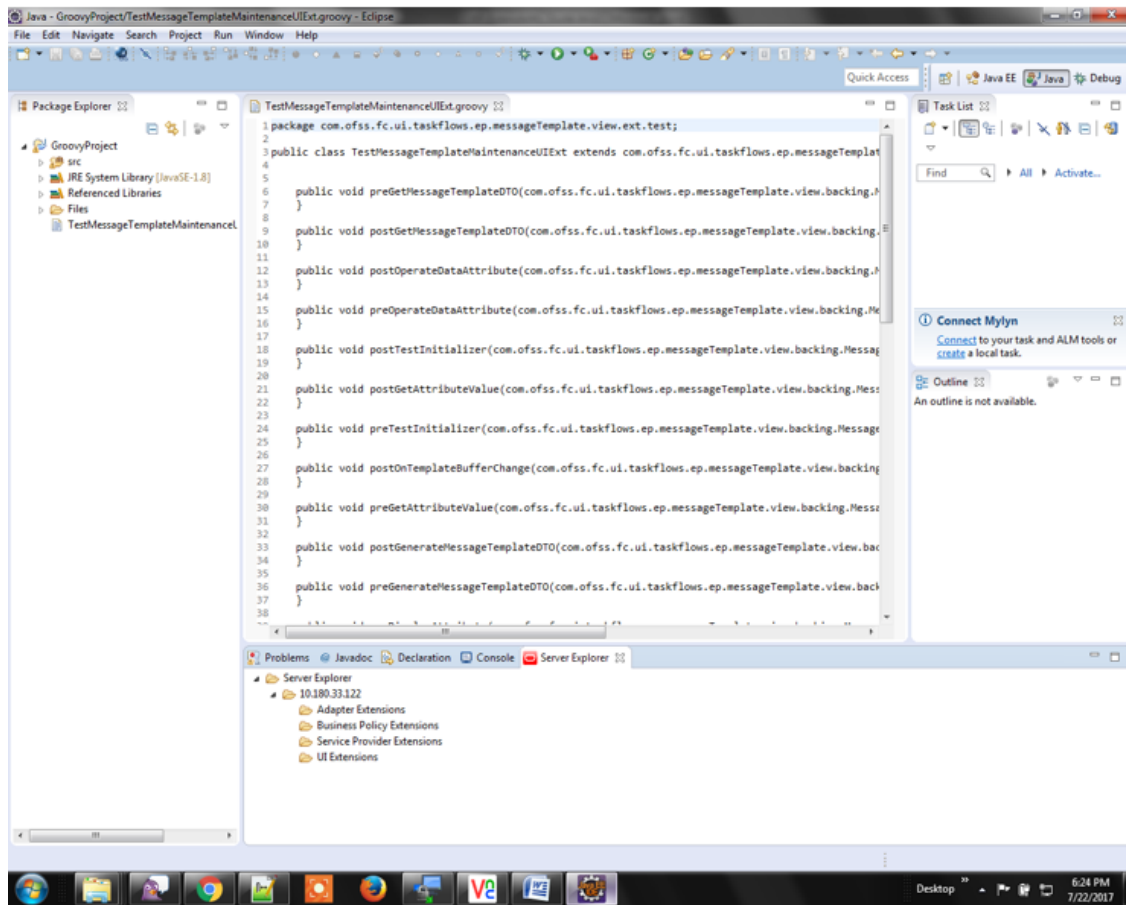
1. For the purpose of undeploying the Extension from the Server, we right click on the specific Extension under the appropriate child of the Server configured under “Server Explorer”.

Figure 4–30 Undeploy UI Extension from Server



2. The specific Extension under the appropriate child of the Server configured under “Server Explorer” disappears and gets undeployed from the Server.

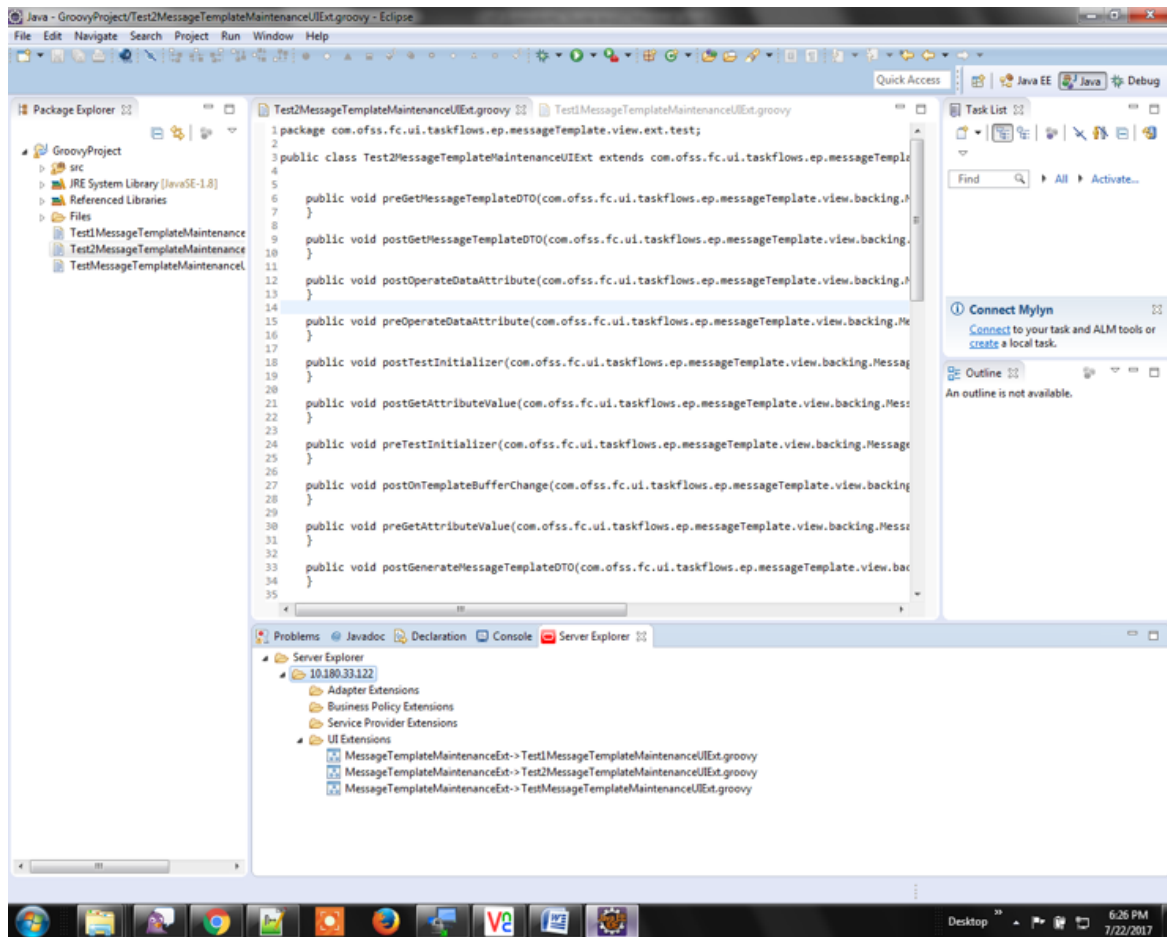
Figure 4–31 UI Extension Undeployed



### 4.2.8 Case of Multiple UI Extensions

We can choose to add multiple Groovy extensions for the same UI Base File. These will appear in the order in which they were added under Server Explorer > Server > Service Provider Extensions.

Figure 4–32 Multiple UI Extensions



### 4.2.9 Inclusion of Groovy Extension in Actual Code Flow

Changes have been included in UIExtensionFactory.java to fetch the Groovy Class from a Map that gets populated after a defined interval in a Singleton.

Figure 4–33 UIExtensionFactory.java





UIGroovyExtensions is a singleton class that runs as a TimerTask to fetch all UIExtensions deployed in the Host Server and populates a map with the Groovy Classes. This gets accessed by the UIExtensionFactory.java above.

Figure 4–34 UIGroovyExtensions

```

104  */
105  private void initializeUIGroovyExtensions() {
106      if (logger.isLoggable(Level.FINE)) {
107          logger.log(Level.FINE, MultiEntityLogger.getUniqueInstance().formatMessage(
108              "initializeUIGroovyExtensions entry."));
109      }
110      try {
111          groovyExtensionClassMap.clear();
112          IExtensionApplicationServiceProxy proxy =(IExtensionApplicationServiceProxy)ProxyFactory.getInstance().getProxy("ExtensionApplicati
113          ExtensionInquiryFilteredResponse response =proxy.fetchUIExtensions(getCloudSessionContext(), null, null,null);
114          for (ExtensionDTO extensionDTO :response.getExtensionDTOs()) {
115              try {
116                  String groovySource =new String(extensionDTO.getResources()[0].getResourceSource(), StandardCharsets.UTF_8);
117                  GroovyClassLoader groovyClassLoader =new GroovyClassLoader(uniqueInstance.getClass().getClassLoader());
118                  Class clazz =groovyClassLoader.parseClass(groovySource);
119                  if(clazz!=null){
120                      groovyExtensionClassMap.put(extensionDTO.getKey().getExtensionId(),clazz.newInstance());
121                  }
122              } catch (CompilationFailedException e) {
123                  logger.log(Level.SEVERE, INVALID_EXTENSION, e);
124              } catch (Throwable e) {
125                  logger.log(Level.SEVERE, INVALID_EXTENSION, e);
126              }
127          }
128      } catch (ServiceException se) {
129          logger.log(Level.SEVERE, INVALID_EXTENSION, se);
130      } catch (FatalException fe) {
131          logger.log(Level.SEVERE, INVALID_EXTENSION, fe);
132      }
133  }
134  }
135  }

```



# 5 ADF Screen Customizations Using UI Extensions

This chapter describes how additional business logic can be added prior to (pre hook) and / or post the execution (post hook) of a particular UI event business logic on the UI side. Extension prior to a UI event execution may be required for the purposes of additional input validation, input manipulation, custom logging, and so on. A few examples in which the UI extensions in the form of pre and post hook are required are mentioned below.

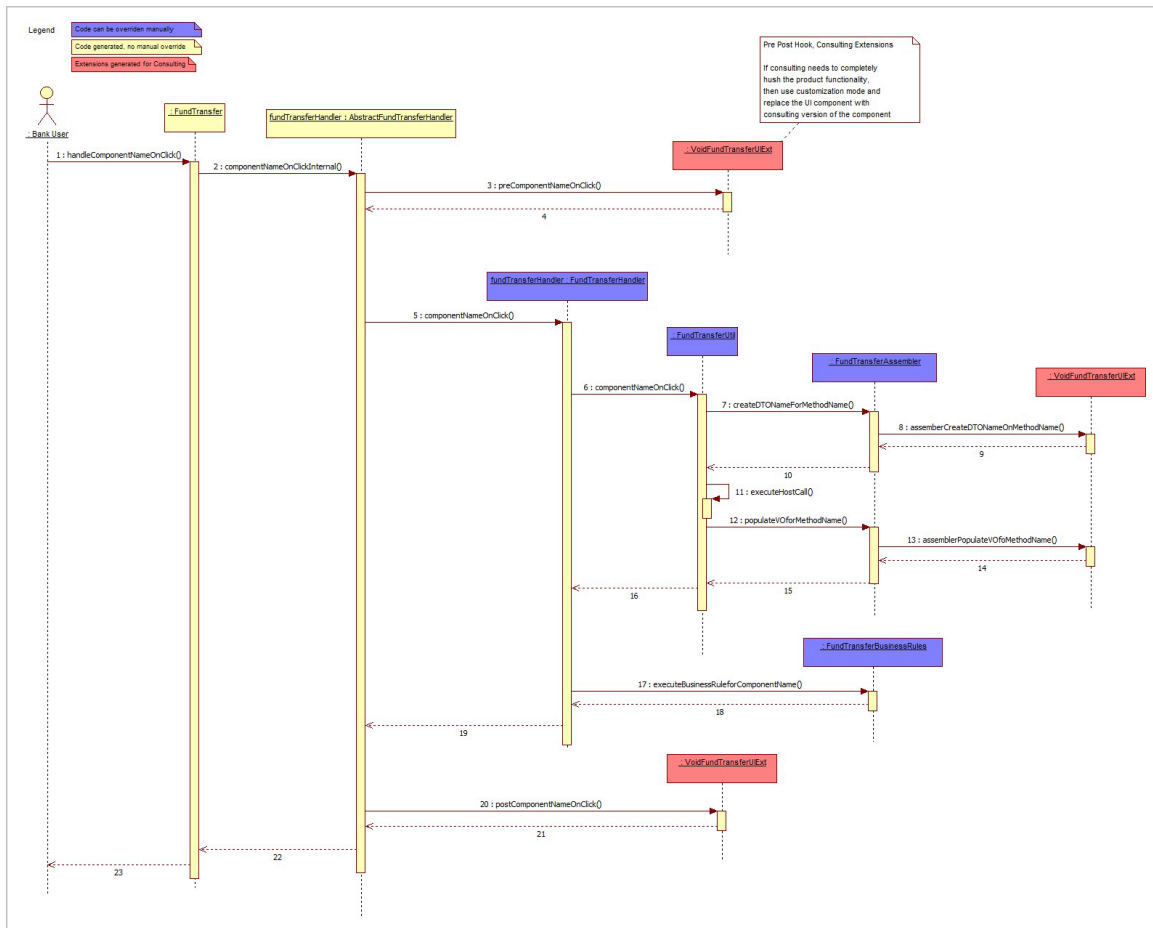
A UI extension in the form of a pre hook can be important in the following scenarios:

- Additional input validations
- Execution of business logic, which necessarily has to happen before going ahead with normal event execution
- Request manipulation prior to making host call

A UI extension in the form of a post hook can be important in the following scenarios:

- Output response manipulation
- Custom UI components rendering, changing to read only

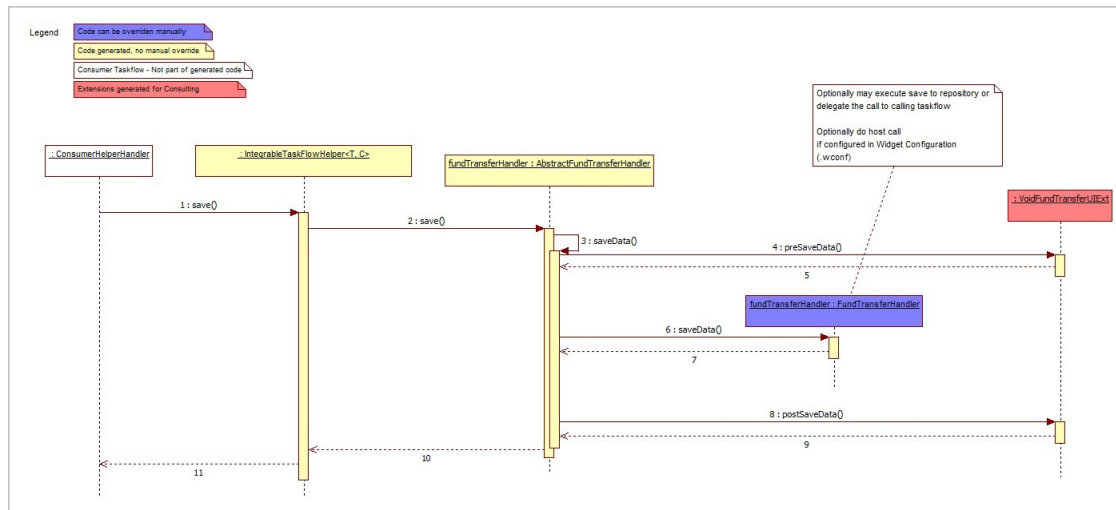
**Figure 5–1 UI Extension Pre Hook and Post Hook Taskflow**



The pre hook is provided after the invocation of *UIevent* call inside the Abstract Taskflow Handler. The extension method is provided with the ADF event and the Taskflow Handler Instance as parameters. The handler instance may be required in such cases where the VO attributes or the UI components need to be accessed as a part of the customization.

The post hook is provided after the event business logic. Similar parameters are provided in the post extension. Hooks are provided in handler and assembler methods, for taskflows using the Integrable Taskflow framework. Hooks are provided in backing bean methods for all other taskflows.

Figure 5–2 Save Method in IntegrableTaskflowHelper



For taskflows implementing the ADF Integrable Taskflow Framework, the pre and post hooks are provided for the common Integrable taskflow helper methods. Refer to the above sequence diagram for the Save method in IntegrableTaskflowHelper.

The following sections detail the important concepts which should be understood for extending in this UI layer.

## 5.1 UI Extension Interface

The OBP ADF Taskflow Generator generates an interface for the extensions of a particular taskflow. The interface name is of the form `I<Taskflow_Name>UIExt`. This interface has a pair of pre and post method definitions for each public method present in the Abstract Taskflow Handler and the Integrable Taskflow Helper. The signatures of these methods are:

```
public void pre<Method_Name>(<Method_Parameters>) throws
    FatalException;
public void post<Method_Name>(<Method_Parameters>) throws
    FatalException;
```

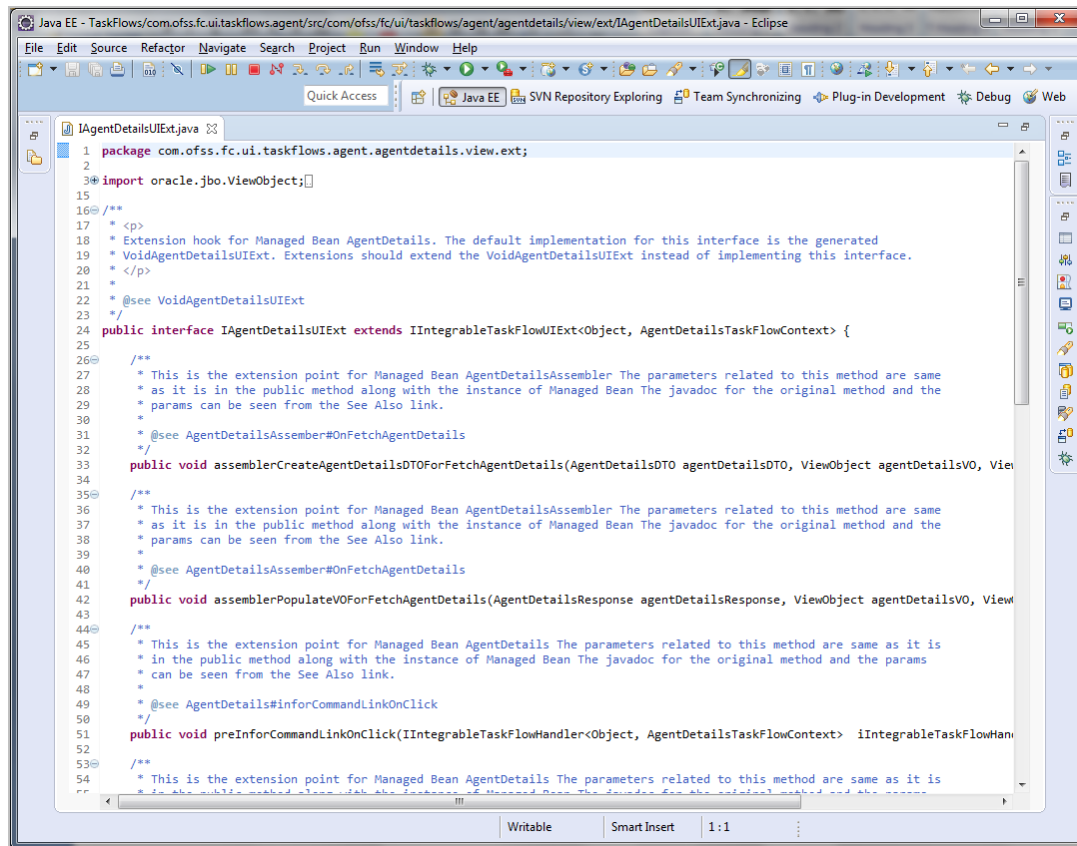
A single method is provided for Integrable Assembler. The signature as below:

```
public void assembler<Method_Name>(<Method_Parameters>) throws
    FatalException;
```

A UI extension class has to implement this interface. The pre method of the extension is executed before the actual method and the post method of the extension is executed after the method.

The return type for certain methods are boolean (for example, `public boolean preValidateData`).

Figure 5–3 Example of UI Extension

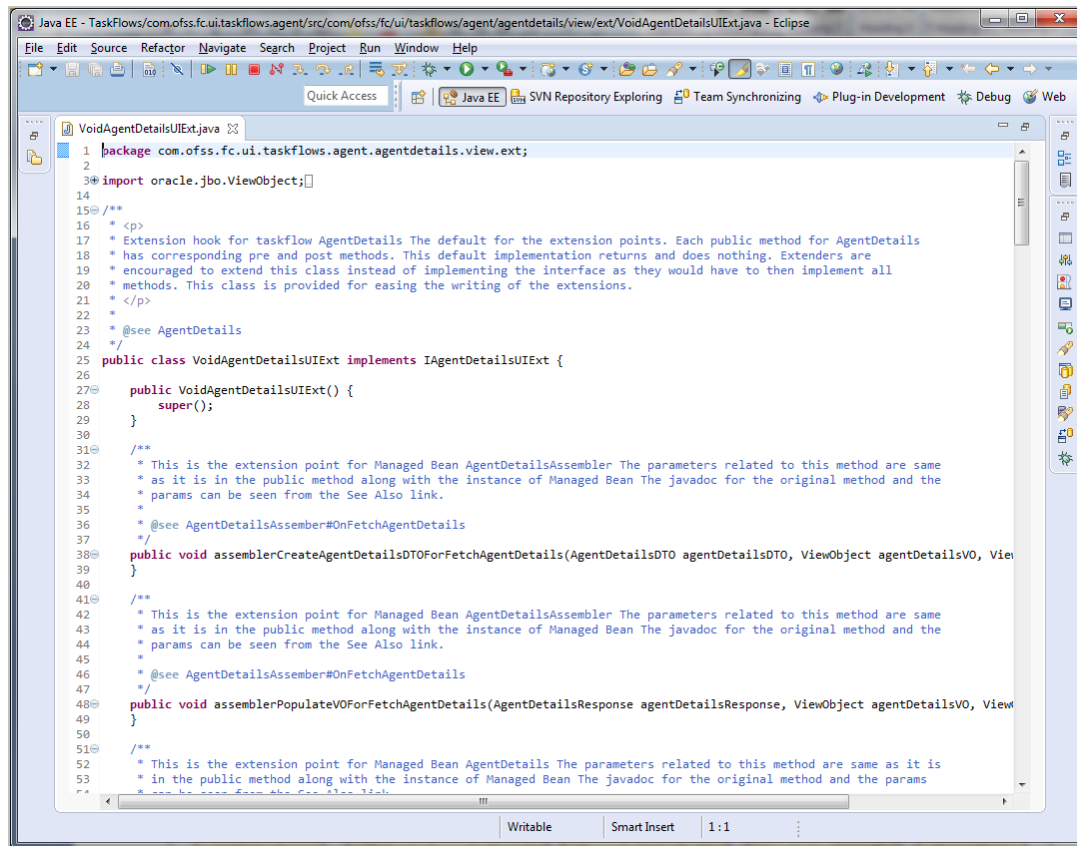


## 5.2 Default UI Extension

The OBP plug-in generates a default extension for a particular taskflow in the form of the class **Void<Taskflow\_Name>UIExt**. This class implements the aforementioned UI extension interface without any business logic, that is, the implemented methods are empty.

The default extension is a useful and convenient mechanism to implement the pre and / or post extension hooks for specific methods of a taskflow. Instead of implementing the entire interface, one should extend the default extension class and override only the required methods with the additional business logic. Product developers DO NOT implement any logic, including product extension logic, inside the default extension classes. This is because the classes are auto-generated, reserved for product use, and get overwritten as a part of a bulk generation process.

Figure 5–4 Example of Default UI Extension



## 5.3 UI Extension Executor

The OBP plug-in for Eclipse generates a UI extension executor interface and an implementation for the executor interface. The naming convention for the generated executor classes which enable "extension chaining" is shown below:

```

Interface : I<Taskflow_Name>UIExtExecutor
Implementation : < Taskflow_Name >UIExtExecutor

```

The UI extension executor class, on load, creates an instance each of all the extensions defined in the UI extensions configuration. If no extensions are defined for a particular service, the executor creates an instance of the default extension for the taskflow. The executor also has a pair of pre and post methods for each method. These methods in turn call the corresponding methods of all the extension classes defined for the taskflow.

Figure 5–5 UI Extension Executor Class Taskflow

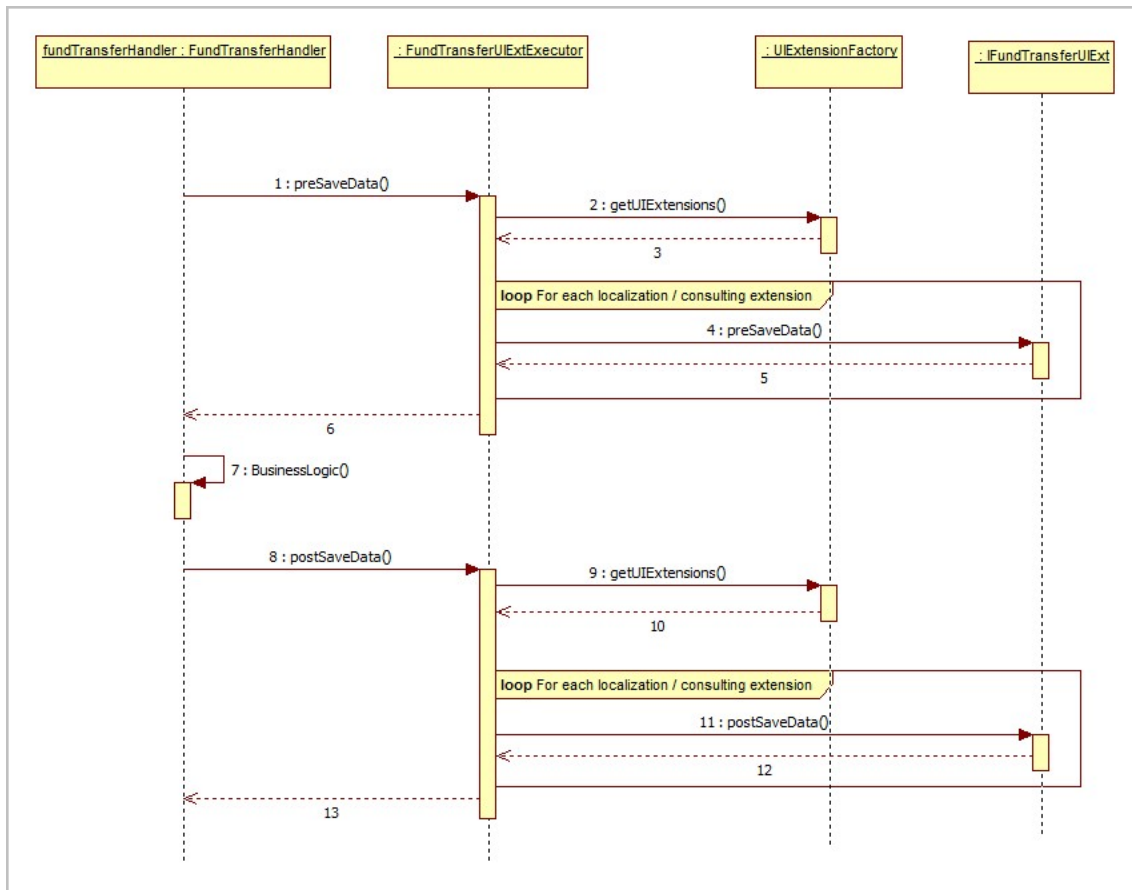




Figure 5–6 Example of UI Extension Executor Class

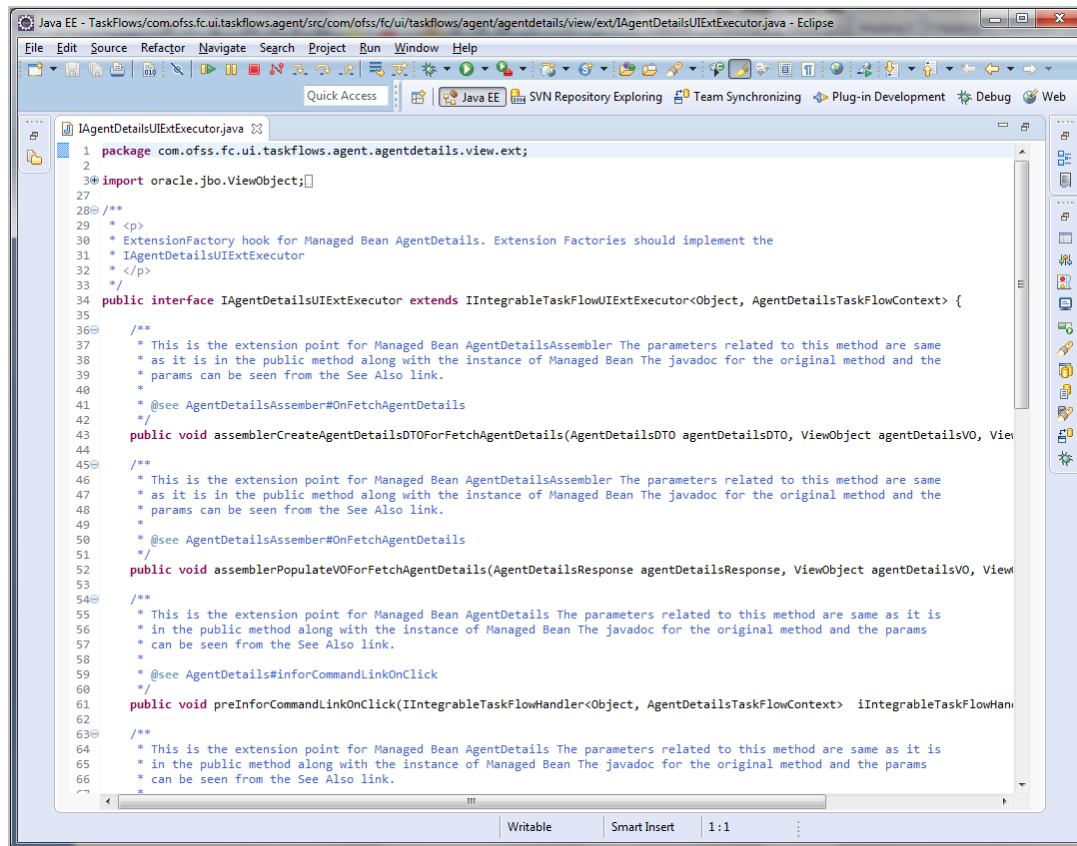
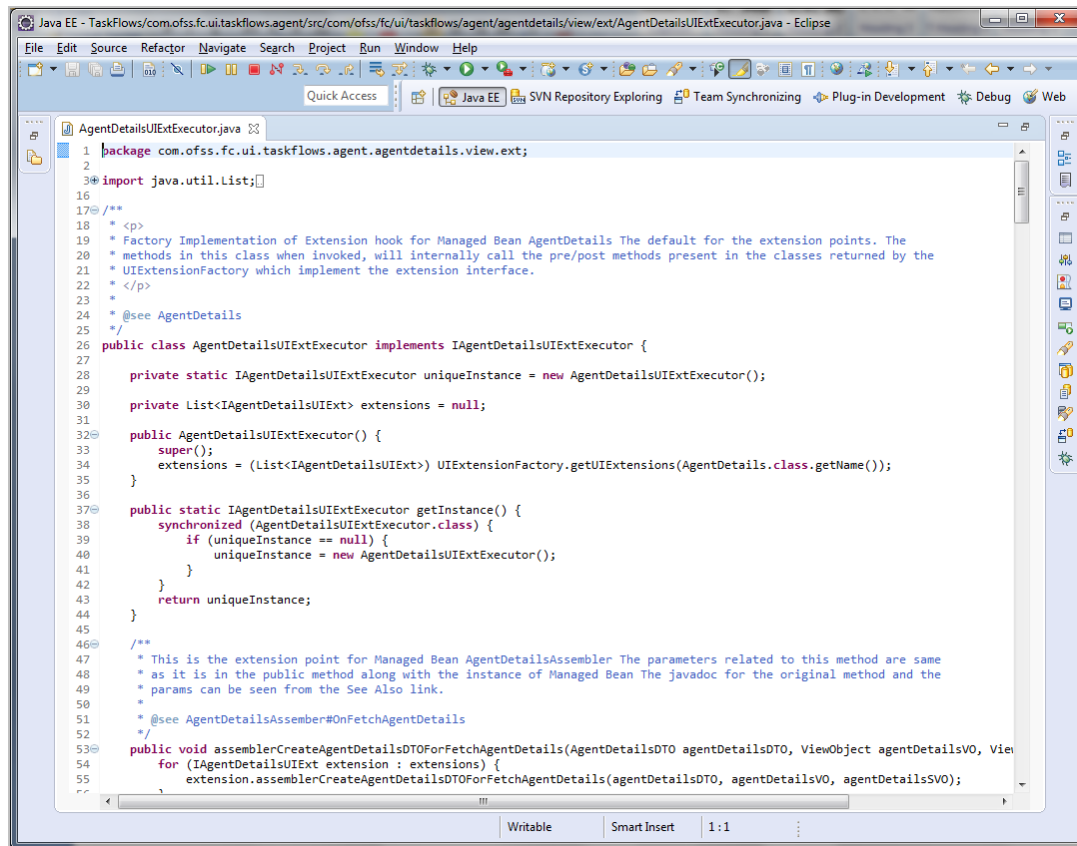


Figure 5–7 Example of UI Extension Executor Class



## 5.4 Extension Configuration

The extension classes that implement the extension interface are mapped to the taskflow with the help of seed data in **FLX\_FW\_CONFIG\_ALL\_B**.

Following is a sample implementation.

### Single Extension Class

```

insert into
FLX_FW_CONFIG_ALL_B (CATEGORY_ID, PROP_ID, PROP_VALUE, PROP_
COMMENTS, OBJECT_VERSION_NUMBER, CREATED_BY, CREATION_DATE, LAST_
UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS_FLAG, FACTORY_SHIPPED_
FLAG)
values
('UIExtensions', 'com.ofss.fc.ui.taskflows.account.accountholderpre
ferencesetup.view.backing.AccountHolderPreferenceSetup', 'com.ofss.
fc.lz.au.ui.taskflows.account.accountholderpreferencesetup.view.ex
t.RegionalAccountHolderPreferenceSetupUIExt', '', 1, 'ofssuser', SYSDA
TE, 'ofssuser', SYSDATE, 'A', 'y');

```

### Multiple Extension Classes

```

insert into

```

```

FLX_FW_CONFIG_ALL_B(CATEGORY_ID,PROP_ID,PROP_VALUE,PROP_
COMMENTS,OBJECT_VERSION_NUMBER,CREATED_BY,CREATION_DATE,LAST_
UPDATED_BY,LAST_UPDATED_DATE,OBJECT_STATUS_FLAG,FACTORY_SHIPPED_
FLAG)
values
('UIExtensions','com.ofss.fc.ui.taskflows.account.accountholderpre
ferencesetup.view.backing.AccountHolderPreferenceSetup','com.ofss.
fc.lz.au.ui.taskflows.account.accountholderpreferencesetup.view.ex
t.RegionalAccountHolderPreferenceSetupUIExt','','1','ofssuser',SYSDA
TE,'ofssuser',SYSDATE,'A','y');
insert into
FLX_FW_CONFIG_ALL_B(CATEGORY_ID,PROP_ID,PROP_VALUE,PROP_
COMMENTS,OBJECT_VERSION_NUMBER,CREATED_BY,CREATION_DATE,LAST_
UPDATED_BY,LAST_UPDATED_DATE,OBJECT_STATUS_FLAG,FACTORY_SHIPPED_
FLAG)
values
('UIExtensions','com.ofss.fc.ui.taskflows.account.accountholderpre
ferencesetup.view.backing.AccountHolderPreferenceSetup','com.ofss.
fc.lz.au.ui.taskflows.account.accountholderpreferencesetup.view.ex
t.RegionalAccountHolderPreferenceSetupUIExtForUseCase1','','1','ofss
user',SYSDATE,'ofssuser',SYSDATE,'A','y');
insert into
FLX_FW_CONFIG_ALL_B(CATEGORY_ID,PROP_ID,PROP_VALUE,PROP_
COMMENTS,OBJECT_VERSION_NUMBER,CREATED_BY,CREATION_DATE,LAST_
UPDATED_BY,LAST_UPDATED_DATE,OBJECT_STATUS_FLAG,FACTORY_SHIPPED_
FLAG)
values
('UIExtensions','com.ofss.fc.ui.taskflows.account.accountholderpre
ferencesetup.view.backing.AccountHolderPreferenceSetup','com.ofss.
fc.lz.au.ui.taskflows.account.accountholderpreferencesetup.view.ex
t.RegionalAccountHolderPreferenceSetupUIExtForUseCase2','','1','ofss
user',SYSDATE,'ofssuser',SYSDATE,'A','y');

```

It is possible to configure multiple implementations of pre or post extensions for a taskflow in this layer. This is achieved with the help of the extension executor. It has the capability to loop through a set of extension implementations, which conform to the extension interface supported by the taskflow.

## 5.5 Customization Examples

Following are some examples of customization.

### 5.5.1 Replacing skin

Colours are maintained as a variable in the css lib files of the respective modules. Skin can be replaced to change the colours.

Replace skin: inside `preCustomBranding()`

`@Override`

```
public void preCustomBranding(Main main) {
```

```

/*setting skin */
FacesContext fc = FacesContext.getCurrentInstance();
ELContext elc = fc.getELContext();
String skinId = "skyros";
ExpressionFactory exprFact = fc.getApplication().getExpressionFactory();
ValueExpression ve = exprFact.createValueExpression(elc, "#{sessionScope.skinFamily}", Object.class);
ve.setValue(elc, skinId);
/* setting fonts */
main.setFontPath("/css/lato.css");
/* set this flag to false so as to execute pre hook only once when main is loaded */
ELHandler.set("#{pageFlowScope.isCustomBranding}", "false");
super.preCustomBranding(main);
}

```

Figure 5–8 Replacing skin

```

@Override
public void preCustomBranding(Main main) {
    /*setting skin */
    FacesContext fc = FacesContext.getCurrentInstance();
    ELContext elc = fc.getELContext();
    String skinId = "skyros";
    ExpressionFactory exprFact = fc.getApplication().getExpressionFactory();
    ValueExpression ve = exprFact.createValueExpression(elc, "#{sessionScope.skinFamily}", Object.class);
    ve.setValue(elc, skinId);
    /* setting fonts */
    main.setFontPath("/css/lato.css");
    /* set this flag to false so as to execute pre hook only once when main is loaded */
    ELHandler.set("#{pageFlowScope.isCustomBranding}", "false");
    super.preCustomBranding(main);
}

```

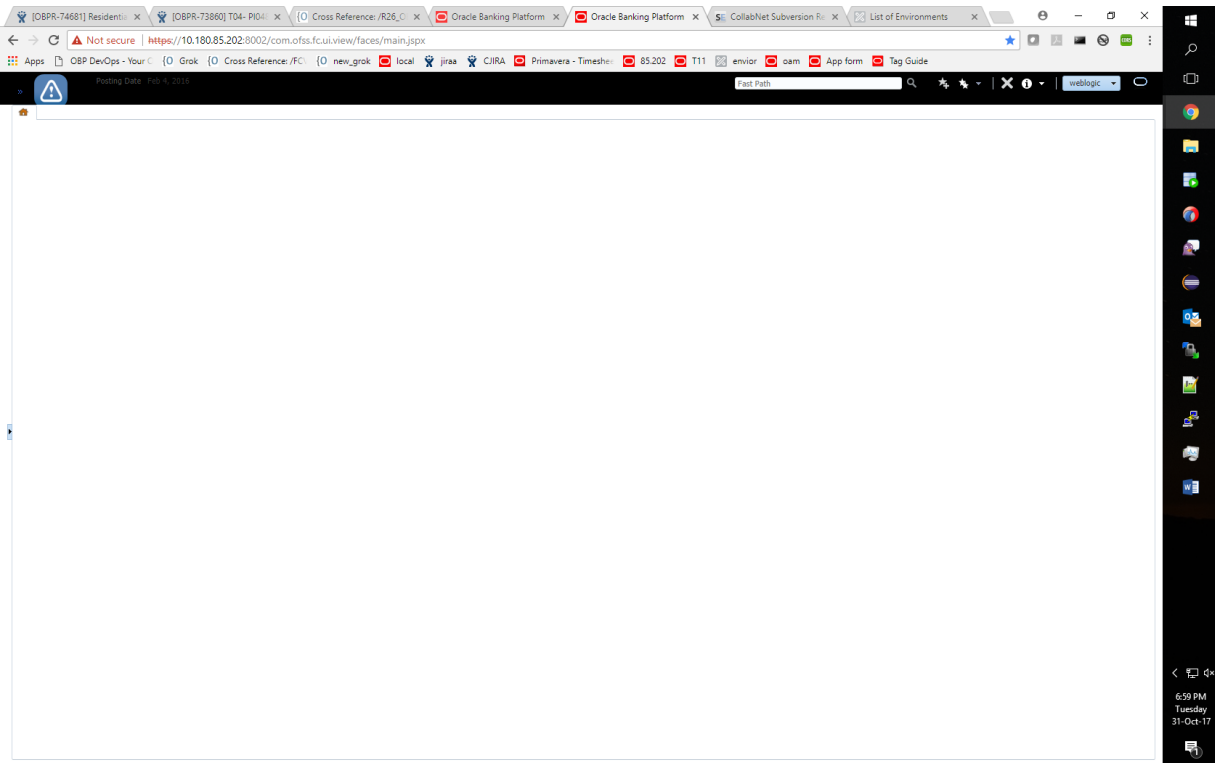
Figure 5–9 Replacing skin

```

public Main() {
    if (voidMainUIExt == null || mainUIExtExecutor == null) {
        voidMainUIExt = new VoidMainUIExt();
        mainUIExtExecutor = new MainUIExtExecutor();
        extension = (IMainUIExtExecutor) UIExtensionFactory.getUIExtensionExecutor(Main.class.getName());
    }
    populateTargetUnitSOC();
    /* set the value of flag isCustomBranding as false when overridden in customization */
    if (ELHandler.get("#{pageFlowScope.isCustomBranding}") == null || Boolean.valueOf(ELHandler.get("#{pageFlowScope.isCustomBranding}").toString())) {
        customBranding();
    }
}

```

Figure 5–10 Example: Replacing skin



### 5.5.2 Changing the logo in the branding bar

Given the multi-brand nature, the ability is provided to display appropriate brand in OBP. For example, Westpac, St George, Bank SA & Bank of Melbourne. Logos are given in the jsp/jspf files in the current code 'Oracle' logo is maintained in "main.jspx" file. To replace a logo, refer to the following screen shot.

Figure 5–11 Replacing the logo

```

617
618     @Override
619     public void postViewBeforePhaseListener(Main main, PhaseEvent phaseEvent) {
620
621         main.getLogoImage().setSource("/images/common/images/risks.png");
622         super.postViewBeforePhaseListener(main, phaseEvent);
623     }
624
625     @Override
626     public void postViewPoll(Main main, PollEvent pollEvent) {

```

### 5.5.3 Modifying fonts

Font-family is maintained as a variable and inherited the variable in the mixins which are used to style the various ADF components. Hence if changed the variable's value font will change.

Variable for href to be maintained in backing bean and this variable will be overridden in customization.

Example:

Main.jspx has a placeholder:

```
href="{pageContext.request.contextPath}/{Main.fontPath}"
```

Main.java holds the path of variable

```
private String fontPath = "/css/roboto.css";
```

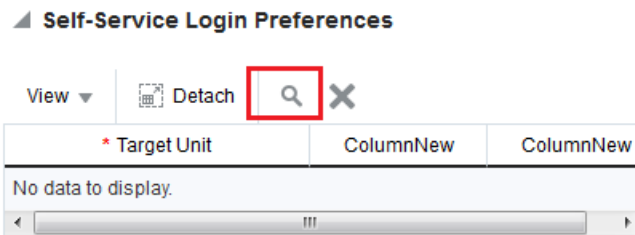
### 5.5.4 Modifying images

Images are maintained in the jsff as well as in css files.

Many ADF components provide provisions to give icons for different states of the components.

Example: set the icon and hover icon attribute

**Figure 5–12 Example: To modify images**



```
this.getButtonAdd().setIcon("/images/common/search/search_16_ena.png");
```

```
this.getButtonAdd().setHoverIcon("/images/common/search/search_16_ena.png");
```

Also wherever component's image can be replaced using css by applying it to the particular selector if component exposes the selector. Same as graphics.

### 5.5.5 Graphics

Graphics include buttons, warnings, and so on.

Button styles are maintained in the respective css files. (Handled through [Section 5.5.1 Replacing skin](#))

Warning text is maintained in the resource bundle (". properties") files: Replace the properties file in config/resources/taskflows/module

SamplePath : config/resources/taskflows/BankPolicyDefinition\_en.properties

### 5.5.6 Adding a simple field to a product screen

Example: Adding input text to a panel form layout

```
/* create component and set relevant properties */
```

```
RichInputText ui = new RichInputText();
```

```
ui.setId("rit1");
```

```
ui.setLabel("Input text");
```

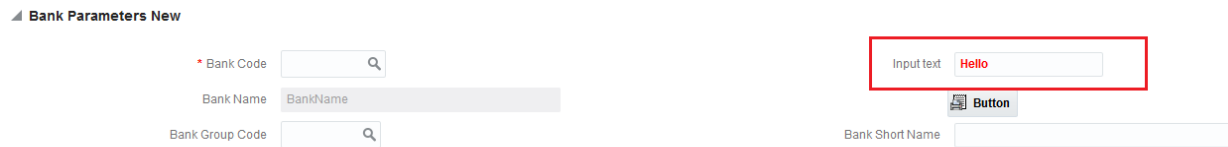
```
ui.setValue("Hello");
```

```
ui.setContentStyle("font-weight:bold;color:red");
```

```
/*add the newly created component to existing form */
```

```
this.getPf1().getChildren().add(ui);
```

**Figure 5–13 Example: To add a simple field to a product screen**



### 5.5.7 Adding a complex field popup to a product screen (popup, table, tree, region, tf)

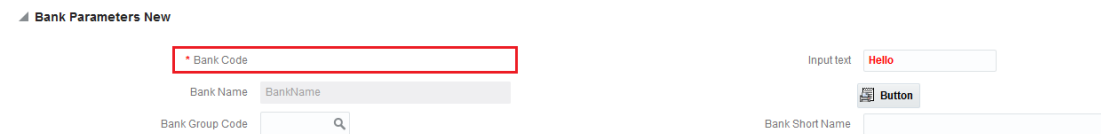
```
/*Handle via making mds changes*/
```

### 5.5.8 Removing an existing field from a product screen

Example: Hiding an LOV from panel form layout

```
this.getBankCodeLOV().setVisible(false);
```

**Figure 5–14 Example: To remove an existing field from a region**



### 5.5.9 Making certain product optional product fields mandatory or optional

Example: Setting bank name to required

```
this.getBankName().setRequired(true);
```

### 5.5.10 Adding a new column to an existing product grid

Example: Adding a new column to a table in CS26

```
/* create new column component */
```

```
RichColumn ui1 = new RichColumn();
ui1.setHeaderText("ColumnNew");
ui1.setId("col3");
ui1.setAlign("center");
ui1.setRowHeader("unstyled");
```

```
/* get the table from bindings where column needs to be added */
```

```

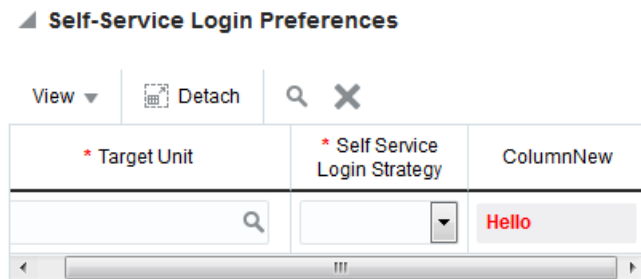
getT1().getChildren().add(ui1);
AdfFacesContext.getCurrentInstance().addPartialTarget(getT1());

/* set the value in column 3 as required on any event */

RichInputText ui11 = new RichInputText();
ui11.setId("rit1");
ui11.setLabel("Input text");
ui11.setValue("Hello");
ui11.setContentStyle("font-weight:bold;color:red");
ui11.setReadOnly(false);
getT1().getChildren().get(3).getId();
getT1().getRowIndex();
getT1().getChildren().get(3).getChildren().add(ui11);
AdfFacesContext.getCurrentInstance().addPartialTarget(getT1());

```

**Figure 5–15 Example: To add a new column to an existing product grid**



### 5.5.11 Hiding columns from an existing product grid

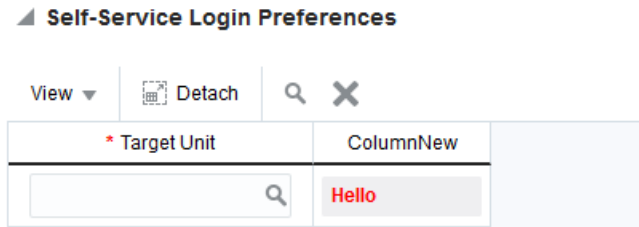
Example: Hiding an existing column from a table in CS26

```
/* get the corresponding column and set its rendered property to false */
```

```
this.getT1().getChildren().get(1).setRendered(false);
```



**Figure 5–16 Example: To hide columns from an existing product grid**



## 5.5.12 Graying out certain columns from an existing product grid

*/\* disabling the component that was set inside the column \*/*

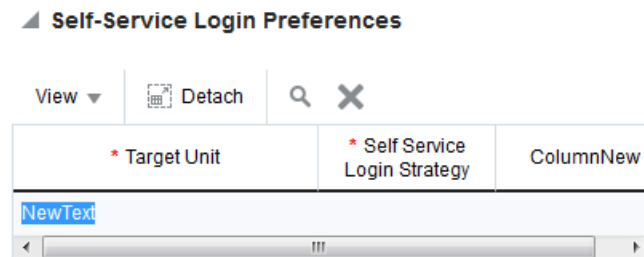
```
this. ui11.setDisabled(true);
```

## 5.5.13 Modifying properties of product table (rows or tablesummary)

```
this.getT1().setEmptyText("NewText");
```

in case where properties are picked up via RB the file itself can be replaced in customization

**Figure 5–17 Example: To modify the properties of product table**



## 5.5.14 Adding a new section to an existing product screen

*/\* create a new panel form layout \*/*

```
RichPanelFormLayout pfl111 = new RichPanelFormLayout();
pfl111.setId("pfl111");
pfl111.setMaxColumns(2);
pfl111.setRows(1);
pfl111.setFieldWidth("60%");
pfl111.setLabelWidth("40%");
getPb1().getChildren().add(pfl111);
AdfFacesContext.getCurrentInstance().addPartialTarget(getPb1());
```

```

/* create components to be added to that section */

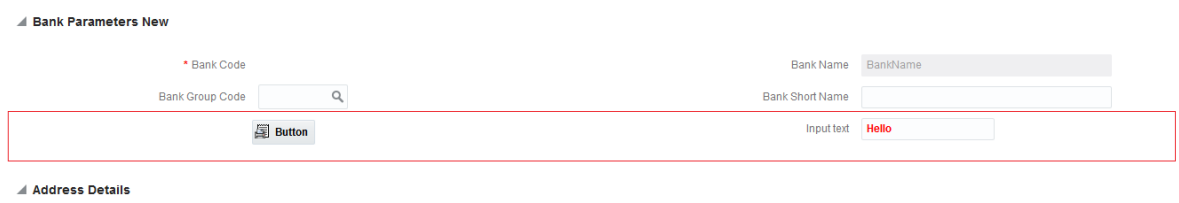
RichInputText ui = new RichInputText();
ui.setId("rit1");
ui.setLabel("Input text");
ui.setValue("Hello");
ui.setContentStyle("font-weight:bold;color:red");
getPfl1().getChildren().add(ui);
AdfFacesContext.getCurrentInstance().addPartialTarget(getPfl1());
RichCommandButton ui2 = new RichCommandButton();
ui2.setId("ch1");
ui2.setText("Button");
ui2.setInlineStyle("font-weight:bold;");
ui2.setIcon("/images/common/search/search_16_ena.png");
ui2.setIcon("/images/common/print/printreceipt_16_ena.png");

/*add new components to the new section */

getPb1().getChildren().get(2).getChildren().add(ui2);
getPb1().getChildren().get(2).getChildren().add(ui);

```

**Figure 5–18 Example: To add a new section to an existing product screen**



### 5.5.15 Hiding a section from a product screen

```
/* Hiding all components inside the panel form layout */
```

```
this.getPfl1().setVisible(false);
```

### 5.5.16 Adding a new tab to an existing product screen made of tabs

```
/* create a new tab and add its relevant properties */
```

```
RichCommandNavigationItem ui2 = new RichCommandNavigationItem();
ui2.setId("newTab");
ui2.setSelected(false);
ui2.setText("newTab");

/* add it to the navigation pane */

this.getNp1().getChildren().add(ui2);
```

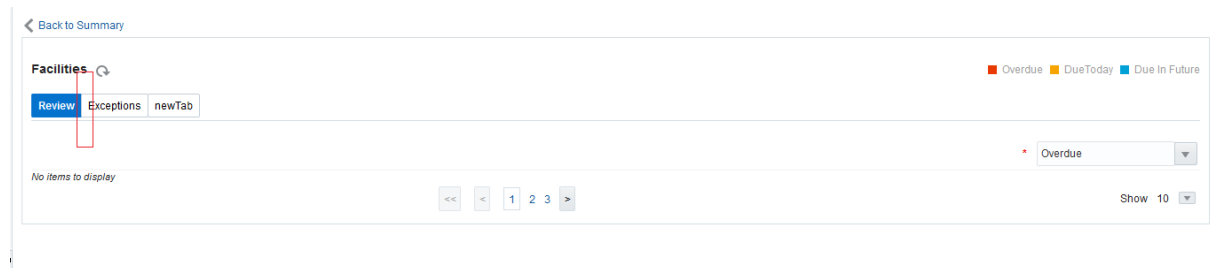
**Figure 5–19 Example: To add a new tab to existing product screen made of tabs**



### 5.5.17 Hiding a tab from a product screen made of multiple tabs

```
this.getNp1().getChildren().get(1).setRendered(false);
```

**Figure 5–20 Example: To hide a tab from a product screen made of multiple tabs**



### 5.5.18 Adding new buttons or links

This approach will not work for "Approvals" and "UI level security"

*/\* Create a new command button and set its relevant properties\*/*

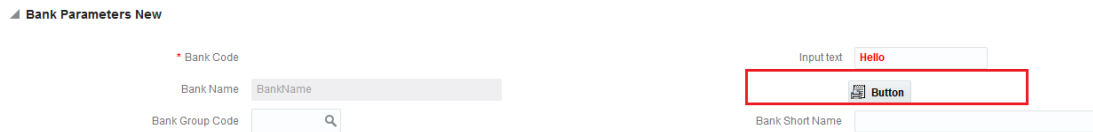
```
RichCommandButton ui2 = new RichCommandButton();
ui2.setId("ch1");
ui2.setText("Button");
ui2.setInlineStyle("font-weight:bold;");
ui2.setIcon("/images/common/search/search_16_ena.png");
```

```
/* add it to the relevant panel component */
```

```
this.getPf1().getChildren().add(ui2);
```

```
AdfFacesContext.getCurrentInstance().addPartialTarget(getPf1());
```

**Figure 5–21 Example: To add new buttons or links**



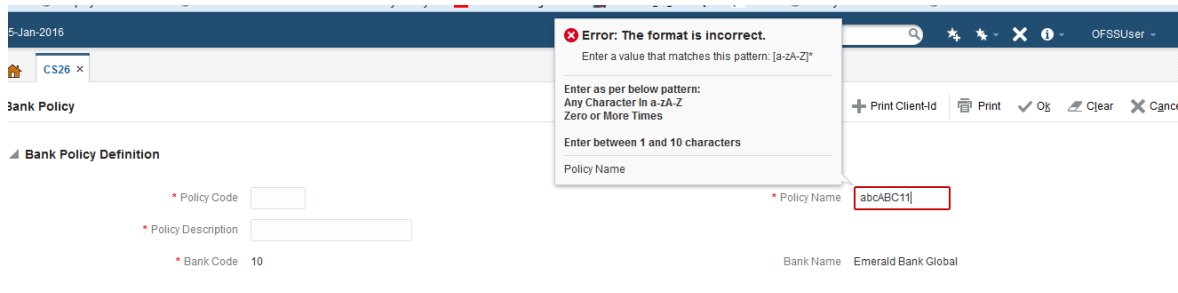
### 5.5.19 Overriding / Customizing the product behaviour on certain actions like button clicks or tab-outs

```
/* need to create a new link programmatically and link the action listener method to it */
```

### 5.5.20 Overriding the product validation pattern

```
this.getPolicyName().setPattern("[a-zA-Z]*");
```

**Figure 5–22 Example: To override the product validation pattern**



### 5.5.21 Overriding the product lengths (min/max)

```
this.getPolicyName().setMaxLength("10");
```

### 5.5.22 Disable / Enable certain product fields

```
this.getBankName().setDisabled(true);
```

### 5.5.23 Change certain product fields to read-only either on load or based on certain conditions

```
this.getBankName().setReadOnly(true);
```

### 5.5.24 Change label of existing product fields

```
this.getBankName().setValue("BankName");
```

### 5.5.25 DC validation

The text for error message comes from "CommonValidationMessages\_en.properties" and this file can be replaced in customization. However the values for Min and Max length inside the message can be overridden.

## 5.6 Using the JSFF Utils

### 5.6.1 How to Use JSFF Utils

Following is the example to use JSFFUtils:

```
JSFUtils.insertPanelHeader
("rphRomanianDetails", "Details", parentId, uiComponent);
JSFUtils.insertRichPanelFormLayoutEnd
("rpfRomanian1", "60%", "40%", 2, 1, "rphRomanianDetails", uiComponent);
```

**Figure 5–23 Example of JSFF Utils**

### 5.6.2 Sample JSFF Utils Code Snippet

```
/**
 * This method adds af:PanelFormLayout ADF component at the end of
 * the given parent.
 * @param id sets id attribute on the af:PanelFormLayout.Type
 * String.
 * @param fieldWidth sets fieldWidth attribute on the
 * af:PanelFormLayout.Type String.
 * @param labelWidth sets labelWidth attribute on the
 * af:PanelFormLayout.Type String.
 * @param maxColumns sets maxColumns attribute on the
 * af:PanelFormLayout.Type integer.
 * @param row sets row attribute on the af:PanelFormLayout.Type
 * integer.
 * @param parentId is the id of the immediate parent component where
 * af:PanelFormLayout need to be appended.Type String
 * @param superParent is the component where parentId is placed.Type
 * UIComponent
 */
public static void insertRichPanelFormLayoutEnd(String id,String
fieldWidth, String labelWidth, int maxColumns,int row, String
parentId, UIComponent superParent) {
    UIComponent uiComponentPGL = superParent.findComponent(parentId);
```

```
RichPanelFormLayout richPanelFormLayout = new RichPanelFormLayout
();
richPanelFormLayout.setId(id);
richPanelFormLayout.setFieldWidth(fieldWidth);
richPanelFormLayout.setLabelWidth(labelWidth);
richPanelFormLayout.setMaxColumns(maxColumns);
richPanelFormLayout.setRows(row);
uiComponentPGL.getChildren().add(richPanelFormLayout);
}
*This method adds af:panelHeader ADF component at the end of the
given parent.
* @param id sets id attribute on the af:panelHeader.Type String.
* @param text sets text attribute on the af:panelHeader.Type
String.
* @param parentId is the id of the immediate parent component where
af:panelHeader need to be appended.Type String
* @param superParent is the component where parentId is placed.Type
UIComponent
*/
public static void insertPanelHeader(String id,String text,String
parentId,UIComponent superParent){
UIComponent uiComponentParent = superParent.findComponent
(parentId);
RichPanelHeader richPanelHeader = new RichPanelHeader();
richPanelHeader.setId(id);
richPanelHeader.setText(text);
uiComponentParent.getChildren().add(richPanelHeader);
}
```

# 6 ADF Screen Customizations Using MDS

OBP provides the extensibility to an application for customizing certain additional requirements of a client. However, since these additional requirements differ from client to client, and the base application functionality remains the same, the code to handle the additional requirements should be kept separate from the code of the base application. For this purpose, **Seeded Customizations** (built on the Oracle Metadata Services framework) can be used to customize an application.

---

## Note

It is recommended to use ADF screen extensions for UI changes instead of mds where ever possible as it is easier to upgrade to new version of product.

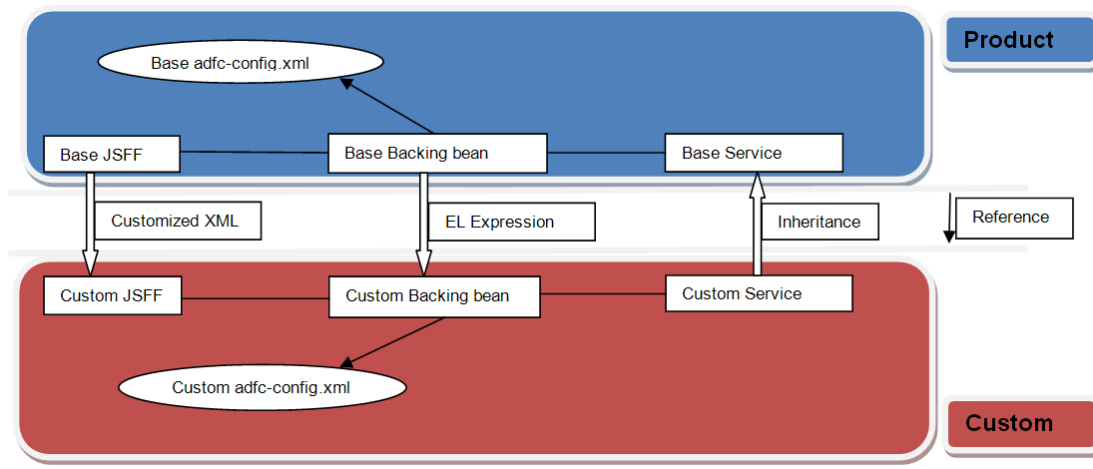
---

## 6.1 Seeded Customization Concepts

When designing seeded customizations for an application, one or more customization layers need to be specified. A customization layer is used to hold a set of customizations. A customization layer supports one or more customization layer value which specifies which set of customizations to apply at runtime.

Custom Application View can be represented as follows:

*Figure 6–1 Customization Application View*



Oracle JDeveloper 11g includes a special role for designing customizations for each customization layer and layer value called the Customization Developer Role.

The following section explains the details about the Oracle JDeveloper customization mode as well as customizing and extending of the ADF application artifact. The detailed documentation for customizing and extending ADF Application Artifacts is also available at the Oracle website:

[http://docs.oracle.com/cd/E25178\\_01/fusionapps.1111/e16691/ext\\_busobjedit.htm](http://docs.oracle.com/cd/E25178_01/fusionapps.1111/e16691/ext_busobjedit.htm)

## 6.2 Customization Layer

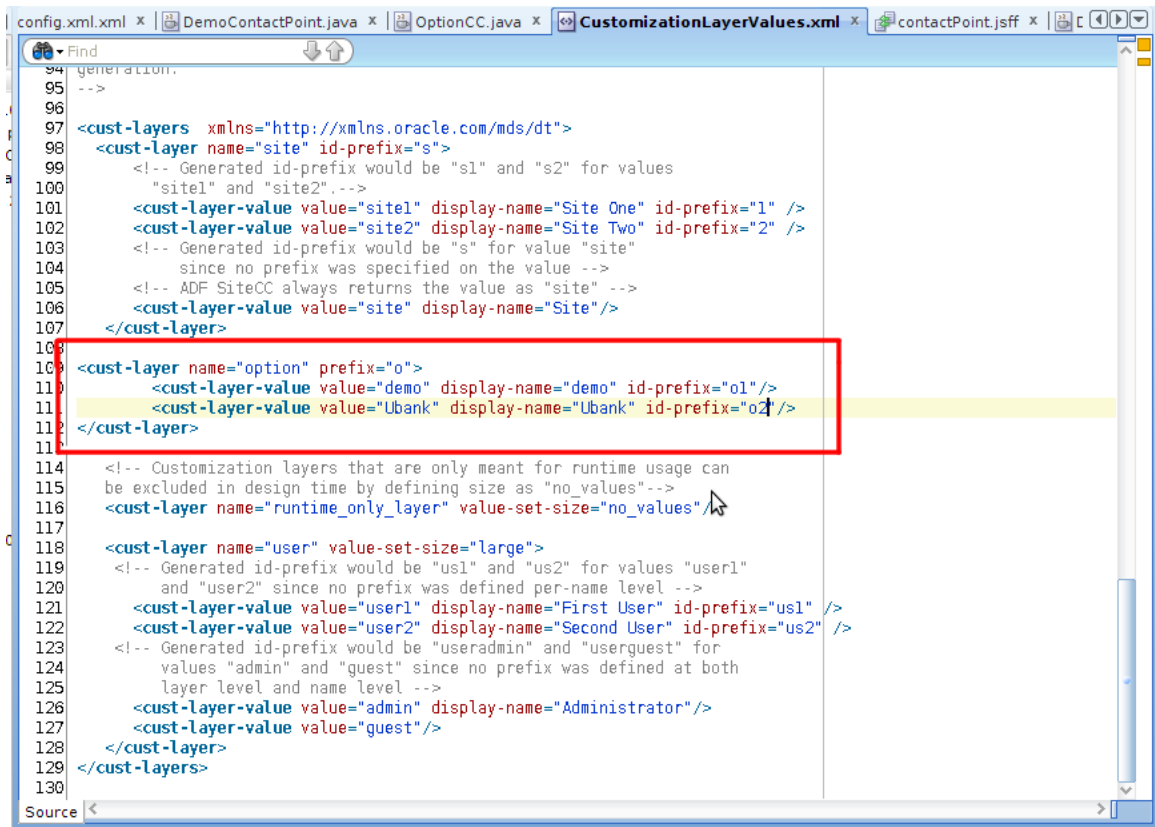
To customize an application, you must specify the customization layers and their values in the CustomizationLayerValues.xml file, so that they are recognized by JDeveloper.

For example, you can create a customization layer with the name **option** and values **demo** and another bank name.

To create the customization layer, follow these steps:

1. From the main menu, choose the **File -> Open** option. Locate and open the file CustomizationLayerValues.xml which is found in the <JDEVELOPER\_HOME>/jdeveloper/jdev directory. In the XML editor, add the entry for a new customization layer and values as shown in the following image.

Figure 6–2 CustomizationLayerValues.xml



```

94 generation.
95 -->
96
97 <cust-layers xmlns="http://xmlns.oracle.com/mds/dt">
98 <cust-layer name="site" id-prefix="s">
99 <!-- Generated id-prefix would be "s1" and "s2" for values
100 "site1" and "site2".-->
101 <cust-layer-value value="site1" display-name="Site One" id-prefix="1" />
102 <cust-layer-value value="site2" display-name="Site Two" id-prefix="2" />
103 <!-- Generated id-prefix would be "s" for value "site"
104 since no prefix was specified on the value -->
105 <!-- ADF SiteCC always returns the value as "site" -->
106 <cust-layer-value value="site" display-name="Site"/>
107 </cust-layer>
108
109 <cust-layer name="option" prefix="o">
110 <cust-layer-value value="demo" display-name="demo" id-prefix="o1"/>
111 <cust-layer-value value="Ubank" display-name="Ubank" id-prefix="o2"/>
112 </cust-layer>
113
114 <!-- Customization layers that are only meant for runtime usage can
115 be excluded in design time by defining size as "no_values"-->
116 <cust-layer name="runtime_only_layer" value-set-size="no_values"/>
117
118 <cust-layer name="user" value-set-size="large">
119 <!-- Generated id-prefix would be "us1" and "us2" for values "user1"
120 and "user2" since no prefix was defined per-name level -->
121 <cust-layer-value value="user1" display-name="First User" id-prefix="us1" />
122 <cust-layer-value value="user2" display-name="Second User" id-prefix="us2" />
123 <!-- Generated id-prefix would be "useradmin" and "userquest" for
124 values "admin" and "quest" since no prefix was defined at both
125 layer level and name level -->
126 <cust-layer-value value="admin" display-name="Administrator"/>
127 <cust-layer-value value="quest"/>
128 </cust-layer>
129 </cust-layers>
130
Source

```

2. Save and close the file.

## 6.3 Customization Class

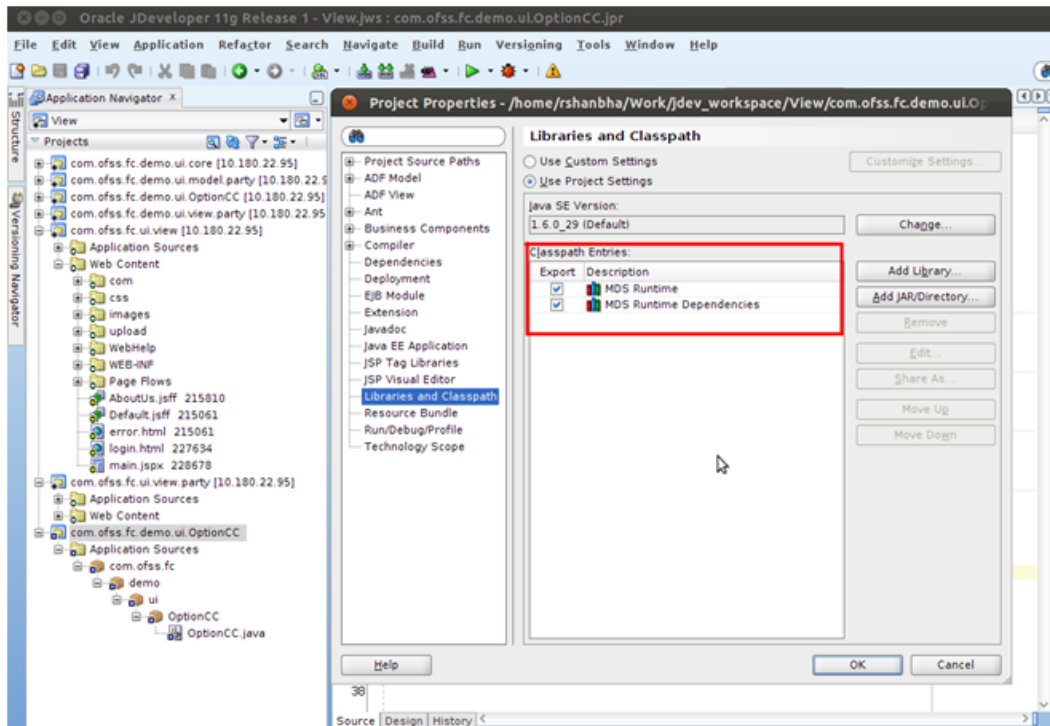
Before customizing an application, a customization class needs to be created. This class represents the interface that the *Oracle Metadata Services* framework uses to identify the customization layer that should be applied to the application's base metadata.

To create a customization class, follow these steps:



1. From the main menu, choose **File -> New**.
2. Create a generic project and give a name (*com.ofss.fc.demo.ui.OptionCC*) to the project.
3. Go to **Project Properties** for this project and add the required **MDS** libraries in the classpath of the project.

Figure 6–3 Customization Class



4. Create the customization class in this project. The customization class **must** extend the *oracle.mds.cust.CustomizationClass* abstract class.

Following are the abstract methods of the CustomizationClass:

- `getCacheHint()` - This method will return the information about whether the customization layer is applicable to all users, a set of users, a specific HTTP request or a single user.
- `getName()` - This method will return the name of the customization layer.
- `getValue()` - This method will return the customization layer value at runtime.

The screenshot below depicts an implementation for the methods:

Figure 6–4 Implementation for the abstract methods of CustomizationClass

```

1 package com.ofss.fc.demo.ui.OptionCC;
2
3 import oracle.mds.core.MetadataObject;
4 import oracle.mds.core.RestrictedSession;
5 import oracle.mds.cust.CacheHint;
6 import oracle.mds.cust.CustomizationClass;
7
8 public class OptionCC extends CustomizationClass {
9
10     private static final String LAYER_NAME = "option";
11     private static final String DEFAULT_LAYER = "demo";
12
13     public OptionCC() {
14         super();
15     }
16
17     public CacheHint getCacheHint() {
18         return CacheHint.REQUEST;
19     }
20
21     public String getName() {
22         return LAYER_NAME;
23     }
24
25     public String[] getValue(RestrictedSession restrictedSession,
26                             MetadataObject metadataObject) {
27         String[] layerValues = null;
28
29         try {
30             //Add Code to fetch layer values from property resources
31         } catch(Exception e) {
32             layerValues = new String[]{DEFAULT_LAYER};
33         }
34
35         return layerValues;
36     }
37 }
38

```

5. Build this class and deploy the project as a JAR file (com.ofss.fc.demo.ui.OptionCC.jar). This JAR file should only contain the customization class.
6. Place this JAR file in the location <JDEVELOPER\_HOME>/jdeveloper/jdev/lib/patches so that the customization class is available in the classpath of JDeveloper.

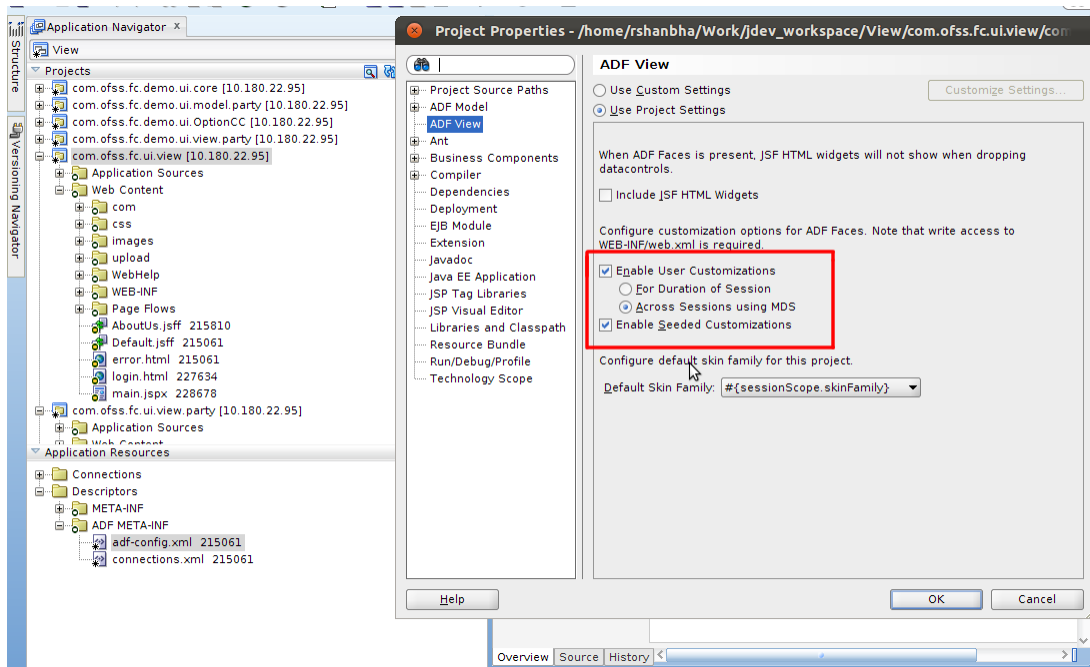
## 6.4 Enabling Application for Seeded Customization

Seeded customization of an application is the process of taking a generalized application and making modifications to suit the needs of a particular group. The generalized application first needs to be enabled for seeded customization before any customizations can be done on the application.

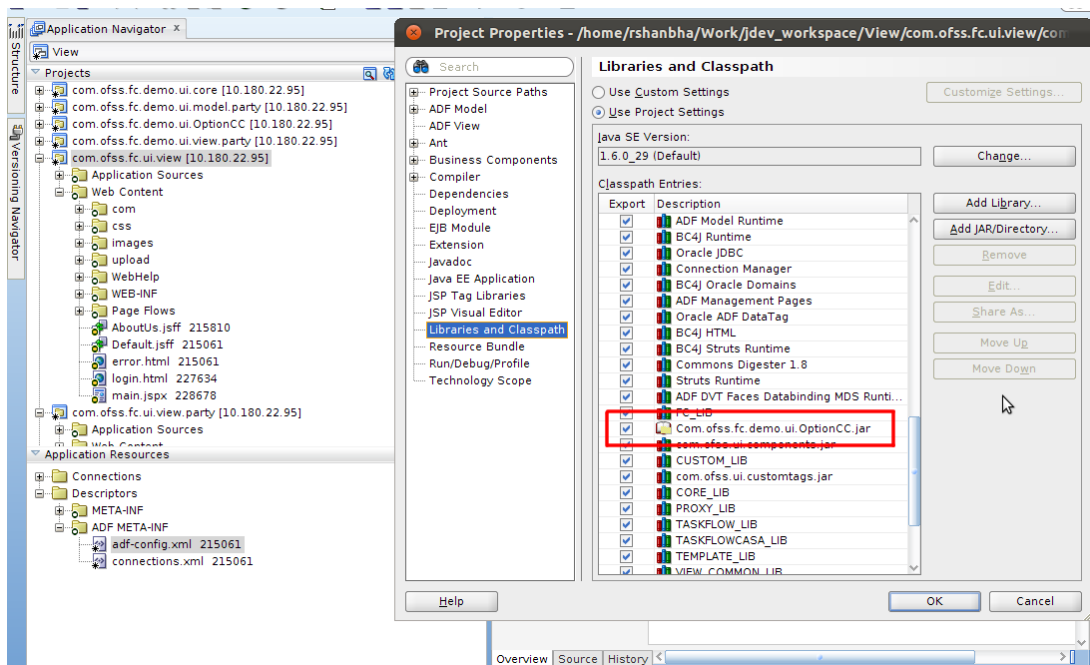
To enable seeded customization for the application, follow these steps:

1. Go to the **Project Properties** of the application's project.
2. In the **ADF Views** section, check the **Enable Seeded Customizations** option.

Figure 6–5 Enable Seeded Customizations



3. In the Libraries and Classpath section, add the previously deployed `com.ofss.fc.demo.ui.OptionCC.jar` which contains the customization class.

Figure 6–6 Adding `com.ofss.fc.demo.ui.OptionCC.jar`

- In the Application Resources tab, open the `adf-config.xml` present in the Descriptors/ADF META-INF folder. In the list of Customization Classes, remove all the entries and add the `com.ofss.fc.demo.ui.OptionCC.OptionCC` class to this list.

Figure 6–7 Adding `com.ofss.fc.demo.ui.OptionCC.OptionCC`

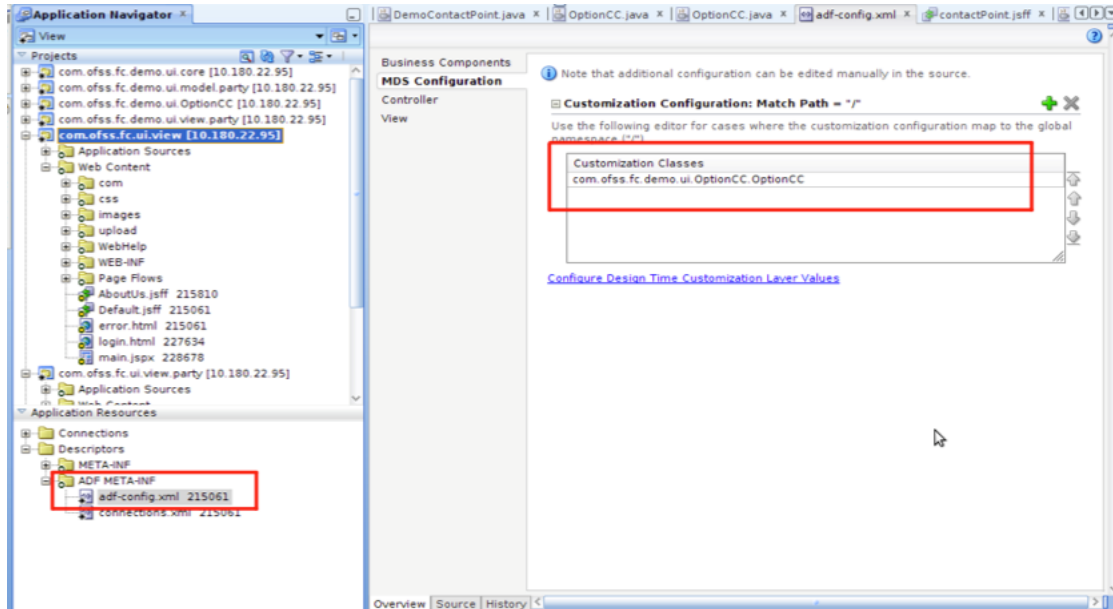
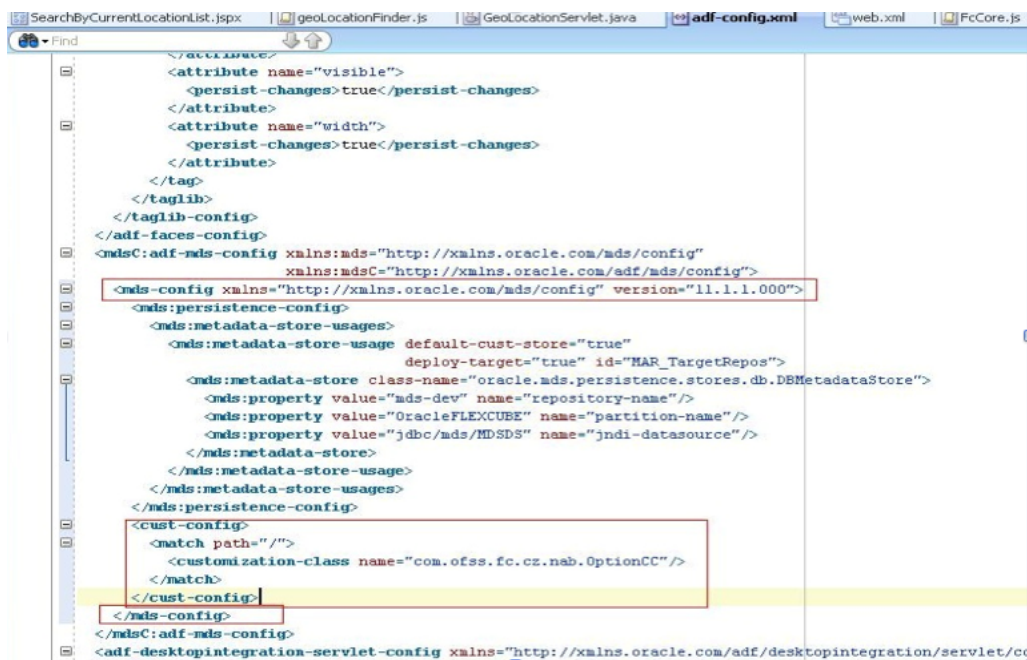


Figure 6–8 `Adf-config.xml`



## 6.5 Customization Project

After creating the Customization Layer and the Customization Class and enabling the application for Seeded Customizations, the next step is to create a project which will hold the customizations for the application.

To create the customization project, follow these steps:

1. From the main menu, choose **File -> New**. Create a new Web Project with the following technologies:
  - ADF Business Components
  - Java
  - JSF
  - JSP and Servlets
2. Go to the **Project Properties** of the project and in the classpath of the project, add the following jars:
  - Customization class JAR (com.ofss.fc.demo.ui.OptionCC.jar)
  - The project JAR which contains the screen / component to be customized. For example, if you want to customize the *Party -> Contact Information -> Contact Point* screen, the related project JAR is com.ofss.fc.ui.view.party.jar.
  - All the dependent JARS / libraries for the project JAR.
  - Enable this project for **Seeded Customizations**.

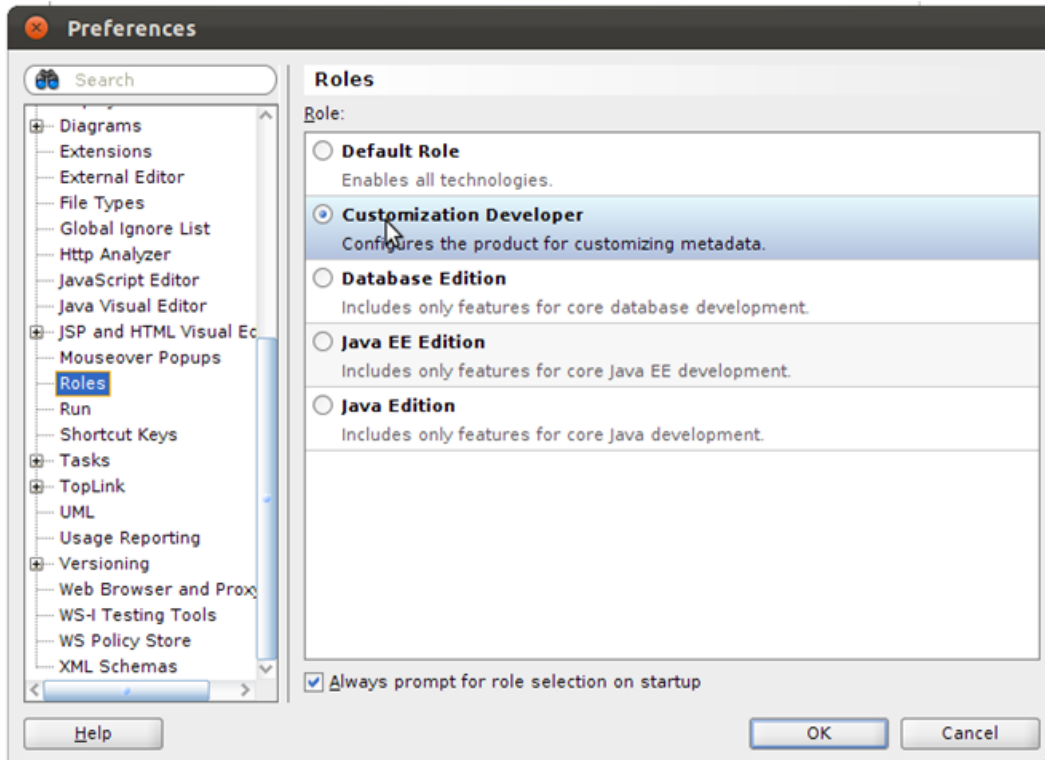
## 6.6 Customization Role and Context

Oracle JDeveloper 11g includes a specific role called Customization Developer Role that is used for editing seeded customizations.

To edit customizations to an application, you will need to switch JDeveloper to that role, follow these steps:

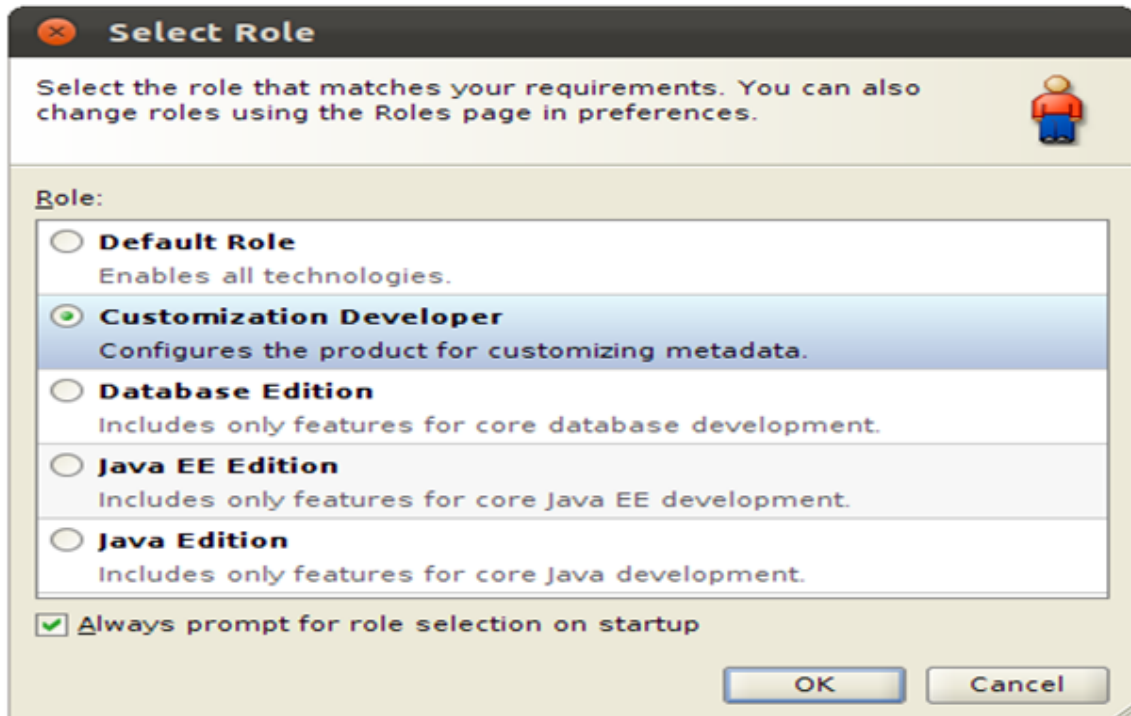
1. In **Tools > Preferences > Roles**, select the Customization Developer Role.

*Figure 6–9 Customization Developer*



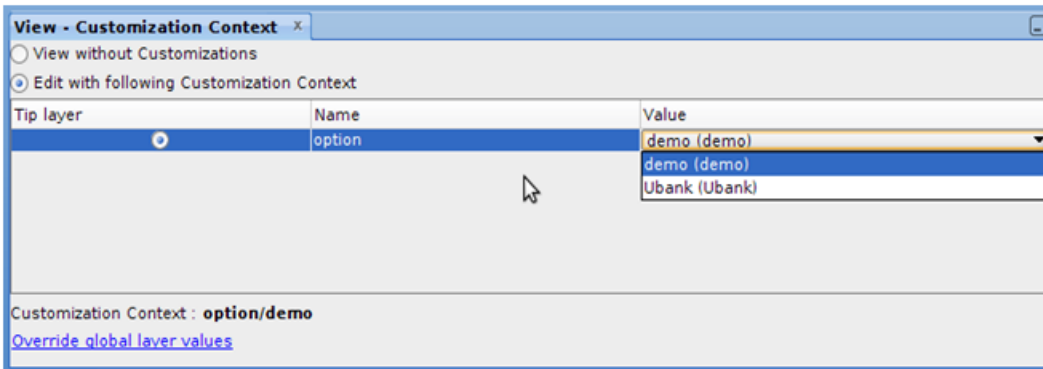
2. Select the "Always prompt for role selection on start up" option.

Figure 6–10 Selecting Always Prompt for Role Selection on Start Up



3. On restarting JDeveloper, you will be prompted for role selection. Select *Customization Developer Role*.
4. Once Oracle JDeveloper 11g has restarted, ensure that the application to be customized is selected in the Application Navigator and have a look around the integrated development environment. You will notice a few changes from the Default Role. The first change you might notice is that files (such as Java classes), that are not customizable, are now read only. The Customization Developer Role can only be used for editing seeded customizations. Anything that is not related to seeded customizations will be disabled. The second major difference you might notice is the *MDS - Customization Context* window that is displayed.
5. Check the *Edit with following Customization Context* option. You will see a list of customization layer name and customization layer values which were defined in the *CustomizationLayerValues.xml* file.
6. Select the Customization Context for which, the customizations you edit should be applicable.

Figure 6–11 View Customization Context



All the customizations which are done to the application are now stored for the selected Customization Context.

## 6.7 Customization Layer Use Cases

### 6.7.1 Adding a UI Table Component to the Screen

This second example of customization, explains adding a table *UI Component*, which displays data to a screen.

**Use Case Description:** The Advanced Search screen is used to display the related accounts and their details for a party. The *Party -> On-Boarding -> Related Party* screen displays the related parties for a party. This section explains adding the table UI component used for displaying the related parties on the *Related Party* screen to the *Advanced Search* screen and populate data in this table on search and selection of a party.



Figure 6–12 Adding a UI Table Component - Party Search screen

The screenshot shows a web application interface for party search. At the top, there are input fields for Party ID (000005296), Full Name, Last Name, Short Name, and Email ID. Below these are 'Search', 'Reset', and 'Clear' buttons. The main section is titled 'Party Search Results' and contains a table with the following data:

Party ID	Name	Type	Number of Roles	Date of Birth or Incorporation	Party Class	Email ID
000005296	Daniel Johnson	Individj2		05-Dec-1980	Others	dipika.patnaik@oracle.com

Below the table are two sections: 'Account Details' and 'Account Specific Details'. 'Account Details' shows a table with one row:

Serial Number	Account Number	Account Type
1	000000000000751LON	

'Account Specific Details' provides further information:

- Account Number: 0000000000007510
- Account Title: Daniel Corp
- Account Opening Date: 15-Jan-2016
- Account Currency: AUD
- Party ID: 000005297
- Party Name: Daniel Corp
- Offer: LOF003 NAB TAILORED HOME LOAN - LOF003
- Branch: 082991 U Bank Operations BR
- Facility Code: FC20160150018764
- Facility Name: Home Loan
- Total Disbursed Amount: \$200,000.00
- Last Disbursement Date: 31-Jan-2016
- Date Of Maturity: 15-Jan-2017
- Accrual Status: Normal
- Approved Amount: \$200,000.00
- Next Installment Amount: \$0.00

Figure 6–13 Adding a UI Table Component - Related Party screen

The screenshot shows the 'Related Party' screen. It includes a toolbar with 'Read', 'Create', and 'Update' buttons. The main section is titled 'Primary Party Information' and contains 'Party Details' and 'Address Details'. 'Party Details' shows:

- Party ID: 000005296
- Full Name: Daniel Johnson
- Home Branch: 082991-U Bank Operations BR
- Party Class: Others
- Party Type: IND
- Date of Birth: 05-Dec-1980
- Gender: Undisclosed
- Roles: Customer, Director
- Onboarding Date: 15-Jan-2016

'Address Details' shows a table with one row:

Serial No.	Party Id	Related Party ID	Relationship Type	Direct Relation Name	Inverse Relation Name	Share Collateral	Share Exposure	Share Income
1	000005295	000005296	Business	Authorized Signat	Authorized Signat	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

The table is highlighted with a red border. At the bottom, it says 'Columns Hidden 1'.

To create the customization as mentioned in this use case, start JDeveloper in the *Default Role* and follow these steps:

### Step 1 Create Customization Project

1. As mentioned in the section **Customization Project**, create a project (*com.ofss.fc.demo.ui.view.party*) to hold the customization.
2. Add the required libraries and JARS along with JAR which contains the above screen (*com.ofss.fc.ui.view.party.jar*).
3. Enable the project for seeded customizations.

### Step 2 Create Binding Bean Class

You will need to create a class which will contain the binding for the *UI Components* which will be added to the screen during customization. Create the class with the following features:

- Private members for the UI Components and public accessors for the same.
- Private member for the backing bean of the screen (*PartySearchMaintenance*) which is initialized in the constructor of this class.
- Private member for the parent UI Component of the newly added UI components and public accessors which returns the corresponding component of the backing bean.

Figure 6–14 Creating Binding Bean Class

```

package com.ofss.fc.demo.ui.view.party.partySearch.backing;

import ...;

public class DemoPartySearchMaintenance {

    public static final String PARTY_RELATIONSHIP_TABLE_VO = "RelatedPartiesAndDetailsTableVOIterator";
    public static final String PARTY_SEARCH_MAINTENANCE_PAGE_DEFN = "com_ofss_fc_ui_view_party_PartySearchMaintenancePageDef1";

    private RichPanelGroupLayout pgll;
    private RichPanelBox olpbl;
    private RichPanelCollection olpcl;
    private RichTable otl;
    private RichOutputText olotl;
    private PartySearchMaintenance partySearchMaintenance;

    public DemoPartySearchMaintenance() {
        super();
        partySearchMaintenance = (PartySearchMaintenance) ELHandler.get(PartyProxyConstants.BACKING_BEAN_PARTY_SEARCH);
    }

    public void setPgll(RichPanelGroupLayout pgll) {
        this.pgll = pgll;
    }

    public RichPanelGroupLayout getPgll() {
        this.pgll = partySearchMaintenance.getPgll();
        return pgll;
    }

    public void setOlpbl(RichPanelBox olpbl) {
        this.olpbl = olpbl;
    }

    public RichPanelBox getOlpbl() {
        return olpbl;
    }

    public void setOlpcl(RichPanelCollection olpcl) {
        this.olpcl = olpcl;
    }
}

```

### Step 3 Create Event Consumer Class

You will need to create a class which contains the business logic for populating the table UI component with the related parties' data. The search and selection of a party in the *Advanced Search* screen raises an event. By binding this event consumer class to the party's selection event, the business logic for populating the related party's data will be executed automatically on selection of a party by the user.

The original event consumer class bound to this event contains the business logic for populating the accounts data. Since your event consumer class would be over-riding the original binding, you will need to incorporate the original business logic for populating the accounts data in your event consumer class.

Figure 6–15 Create Event Consumer Class

```

package com.ofss.fc.demo.ui.view.party.partySearch.event;

import ...;

public class DemoPartySearchConsumer {

    private static final String THIS_COMPONENT_NAME = DemoPartySearchConsumer.class.getName();
    private final Logger logger = MultiEntityLogger.getUniqueInstance().getLogger(this.getClass().getName());

    public DemoPartySearchConsumer() {
        super();
    }

    public void handleAccountTaskCodeAndPartyRelationshipEvent(Object object) {
        PartySearchTaskFlowHelper helper = (PartySearchTaskFlowHelper)object;
        String partyId = helper.getSelectedPartyId();
        fillAccountsTable(partyId);
        fillPartyRelationshipTable(partyId);
    }

    private void fillPartyRelationshipTable(String partyId) {
        DemoPartySearchMaintenance demoPartySearchMaintenance = (DemoPartySearchMaintenance)ELHandler.get("#{requestScope.DemoPartySearchMaintenance}").getO1p1().setVisible(true);

        PartyRelationshipResponse response = null;

        ViewObject partyRelationshipTableVO = IteratorHandler.getViewObject(DemoPartySearchMaintenance.PARTY_SEARCH_MAINTENANCE, partyRelationshipTableVO.clearCache());

        try {
            IPartyRelationshipApplicationServiceProxy client =
                (IPartyRelationshipApplicationServiceProxy)ProxyFactory.getInstance().getProxy(com.ofss.fc.ui.common.consta
                SessionContext sessionContext = SessionContextFactory.getSessionContextFactory().getSessionContextInstance();
                sessionContext.setServiceCode(RelatedPartyConstants.Task_Code);

            response = client.fetchAllRelatedPartiesAndRelationships(sessionContext, partyId);

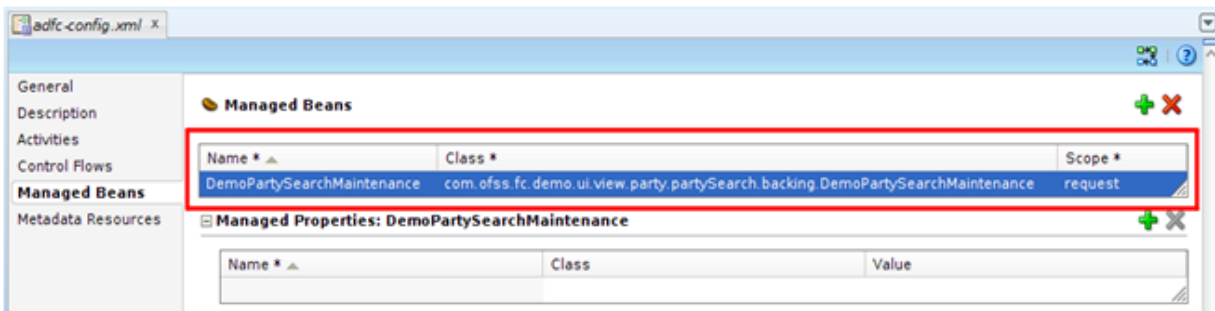
            if (response != null && response.getStatus() != null && response.getStatus().getErrorCode().equals("0")) {
                if (response.getPartyRelationshipsDTO().length > 0) {

```

#### Step 4 Create Managed Bean

You will need to register the binding bean class as a managed bean. Open the project's adfc-config.xml which is present in the WEB-INF folder. In the Managed Beans tab, add the binding bean class as a managed bean with request scope as follows:

Figure 6–16 Creating Managed Bean

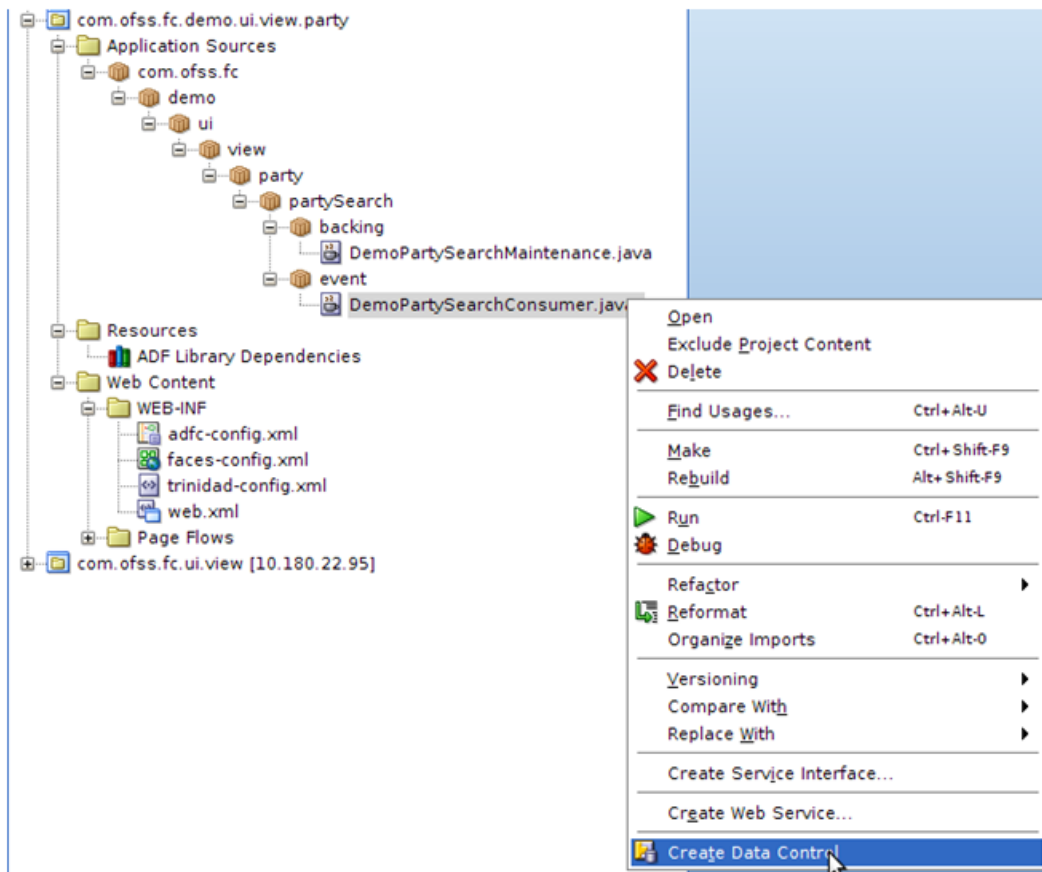


#### Step 5 Create Data Control

For the event consumer class's method to be exposed as an event handler, you will need to create a *data control* for this class.

1. In the *Application Navigator*, right-click the event consumer Java file and create data control.
2. On creation of data control, an XML file is generated for the class and a *DataControls.dcx* file is generated containing the information about the data controls present in the project. You will be able to see the event consumer data control in the *Data Controls* tab.

**Figure 6–17 Create Data Control**



3. Restart JDeveloper in the *Customization Developer Role* to edit the customizations.
4. Ensure that the appropriate *Customization Context* is selected.

### Step 6 Add View Object Binding to Page Definition

You will need to add the view object binding to the page definition of the screen. To open the page definition of the screen, follow these steps:

1. In the Application Navigator, open the Navigator Display Options for Projects tab and check the Show Libraries option.
2. In the navigator tree, locate the JAR that contains the screen (*com.ofss.fc.ui.view.party.jar*).
3. Inside this JAR, locate and open the page definition XML (*com.ofss.fc.ui.view.party.partySearch.pageDefn.PartySearchMaintenancePageDef.xml*)
4. After opening the page definition XML, add a tree binding for the view object

(RelatedPartiesAndDetailsTableVO1) as follows:

**Figure 6–18 Adding View Object Binding to Page Definition - Add Tree Binding**

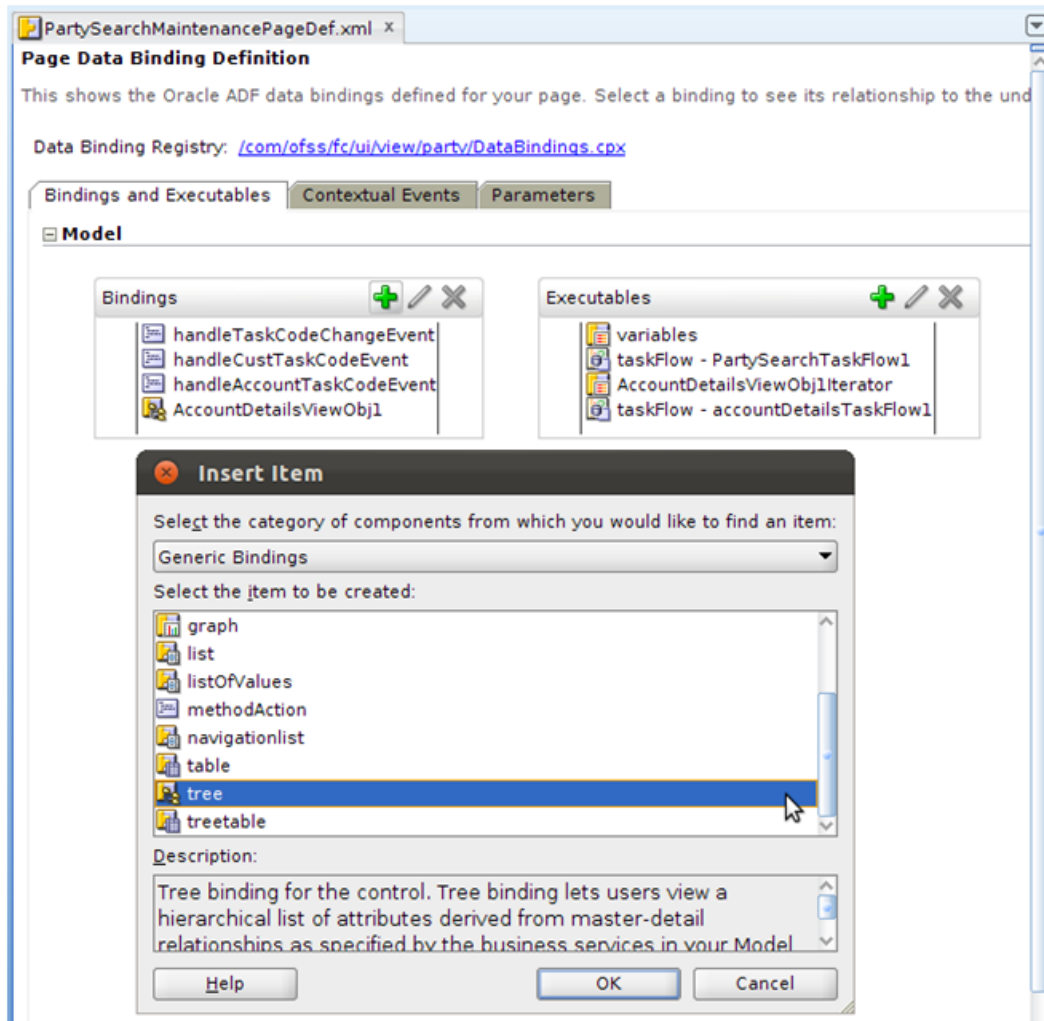
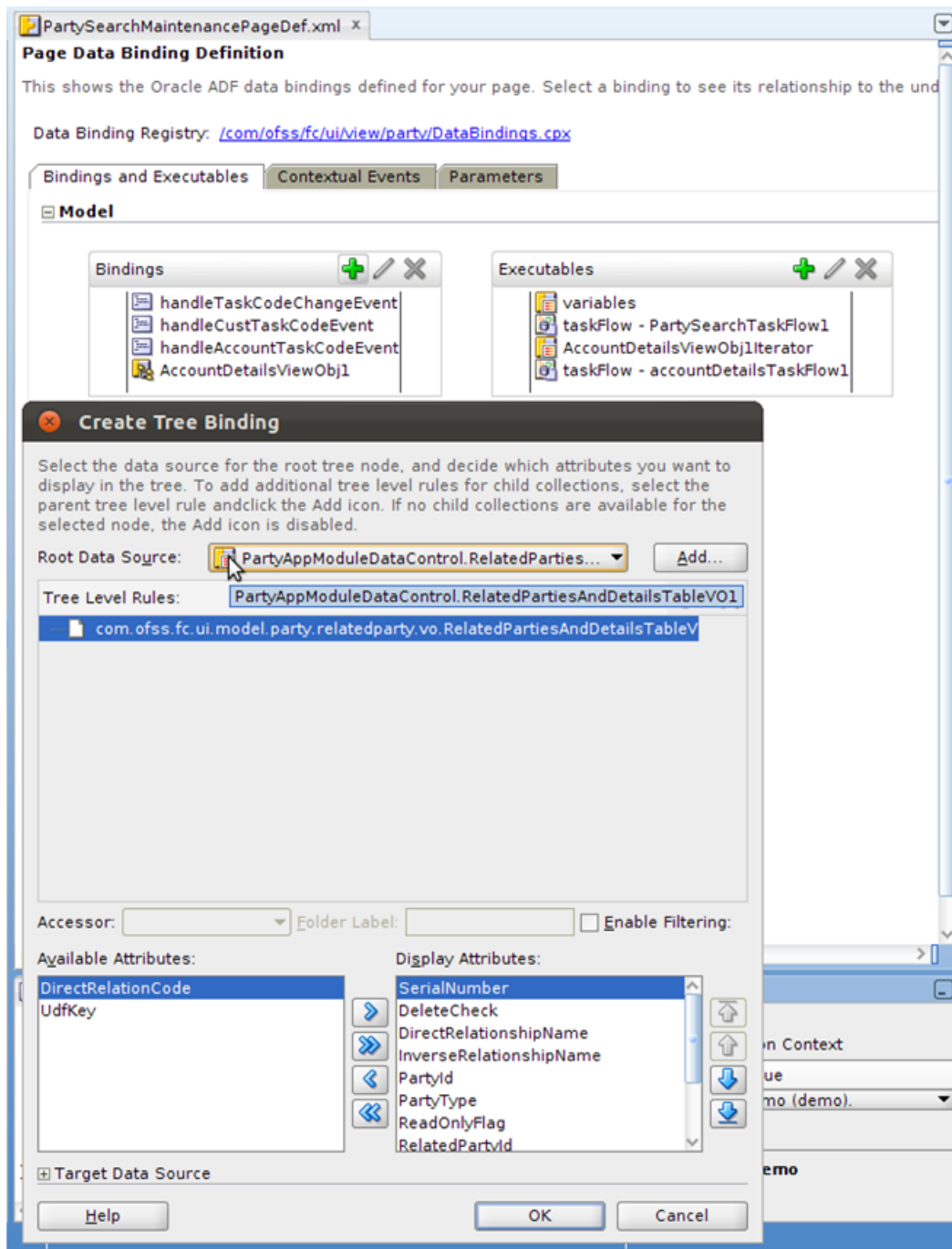


Figure 6–19 Adding View Object Binding to Page Definition - Update Root Data Source



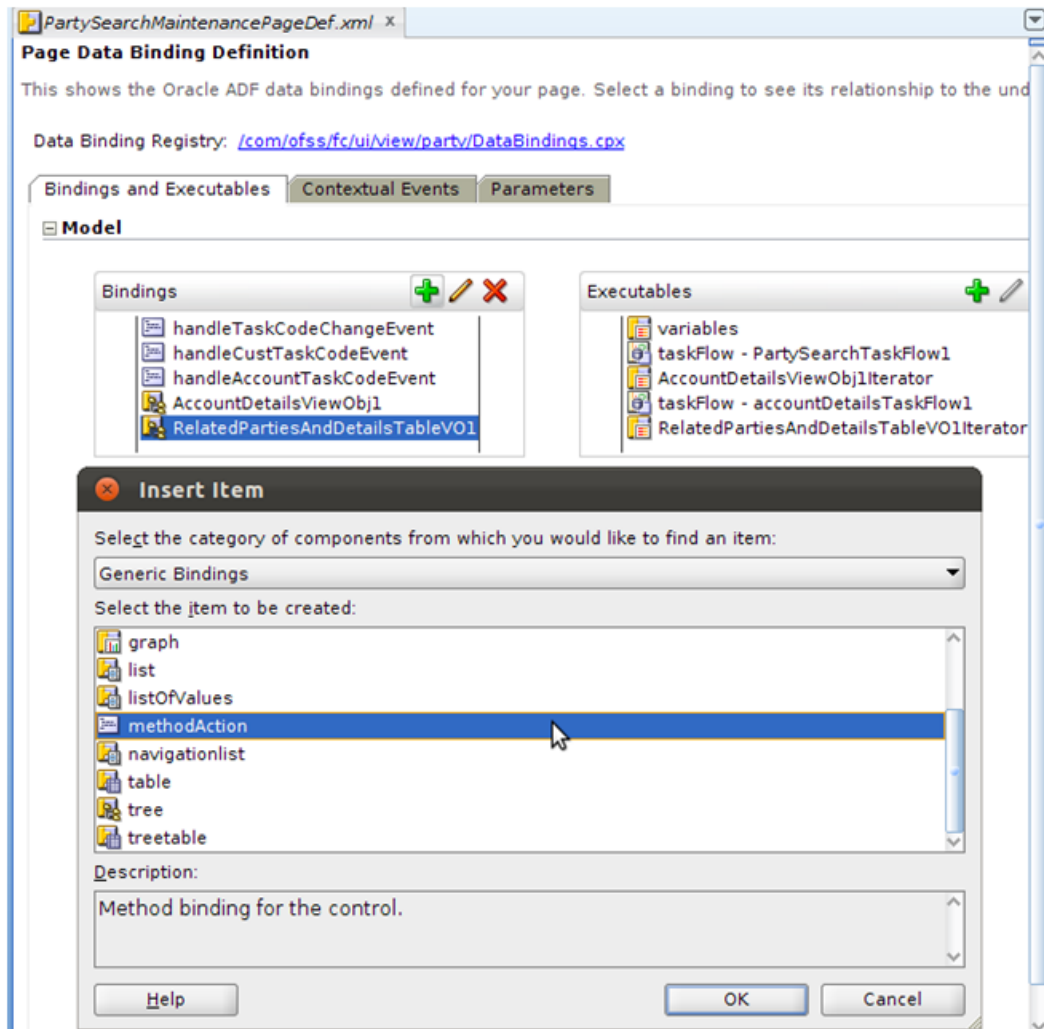
- In Root Data Source, locate the view object which is present in the *PartyAppModuleDataControl*. Select the required display attributes and click **OK**.

### Step 7 Add Method Action Binding to the Page Definition

You will need to add the method action binding for the event consumer data control to the page definition of the screen.

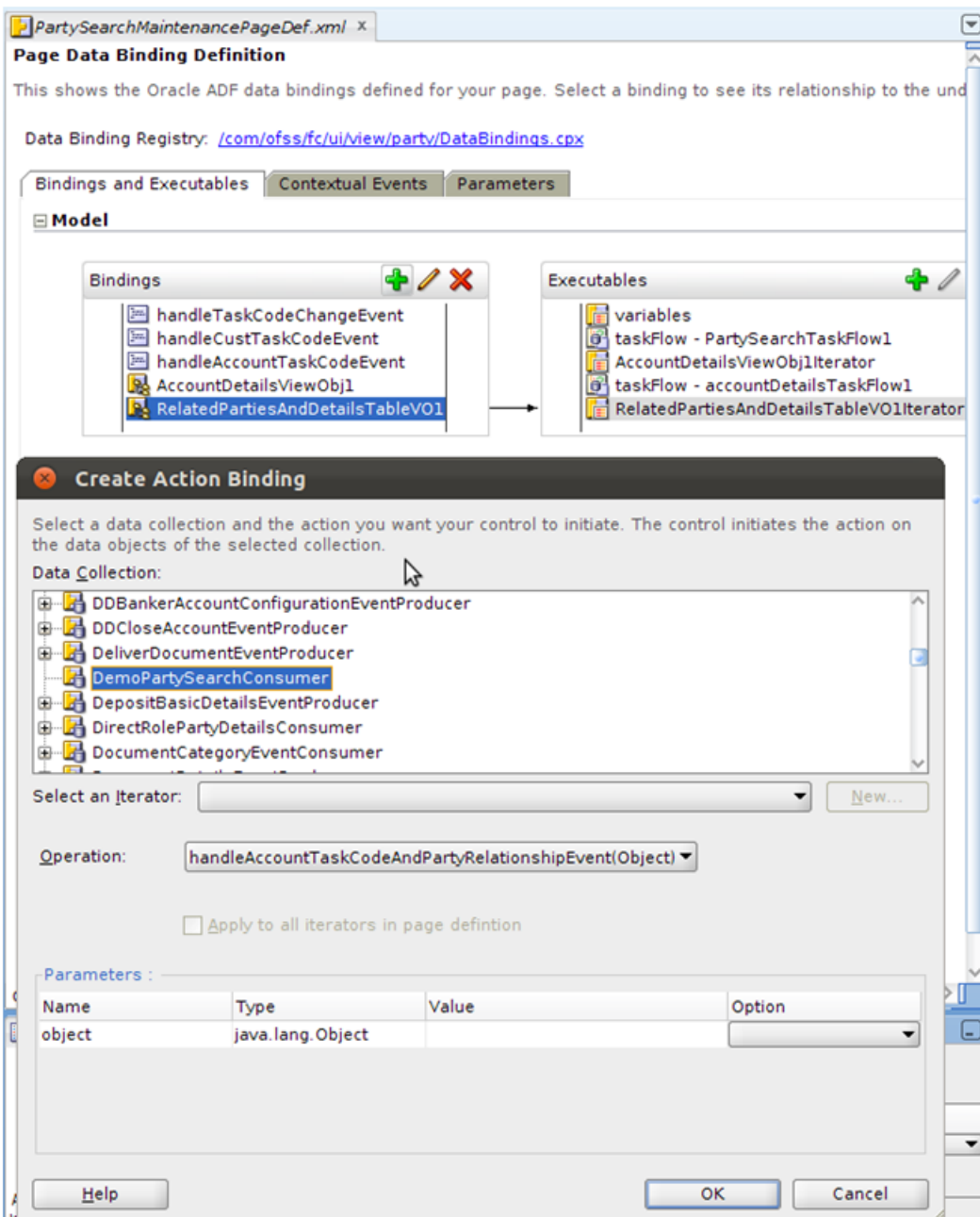
1. After opening the page definition XML, add the method action binding for the *DemoPartySearchConsumer* data control to the page definition as follows:

**Figure 6–20 Page Data Binding Definition - Insert Item**



2. Browse and locate the data control and click **OK**.

Figure 6–21 Page Data Binding Definition - Create Action Binding



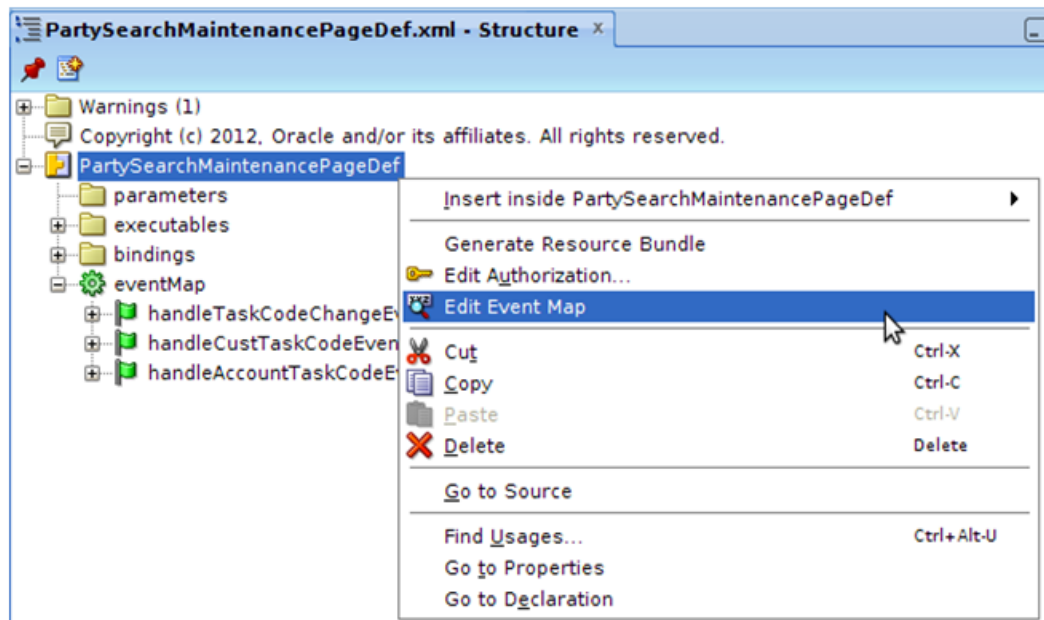
### Step 8 Edit Event Map

You will need to map the *Event Producer* for the party selection event to the **Event Consumer** defined by you in the page definition.

1. In the *Application Navigator*, select the page definition XML file.
2. In the *Structure panel* of JDeveloper, right-click the page definition XML and select *Edit Event Map*.

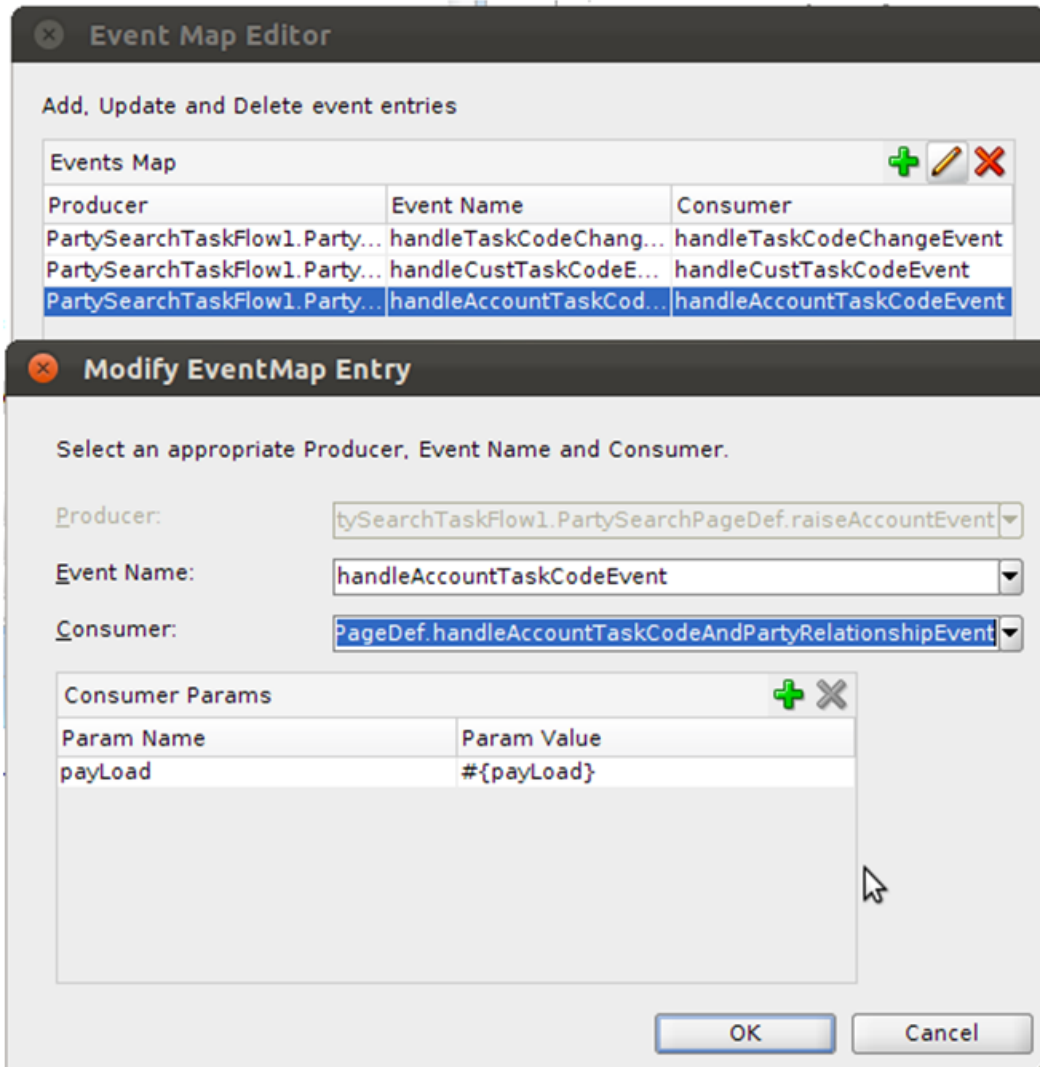


Figure 6–22 Edit Event Map



3. In the **Event Map Editor** panel, edit the mapping for the required event.
4. Select the newly added Event Consumer's method.

Figure 6–23 Event Map Editor

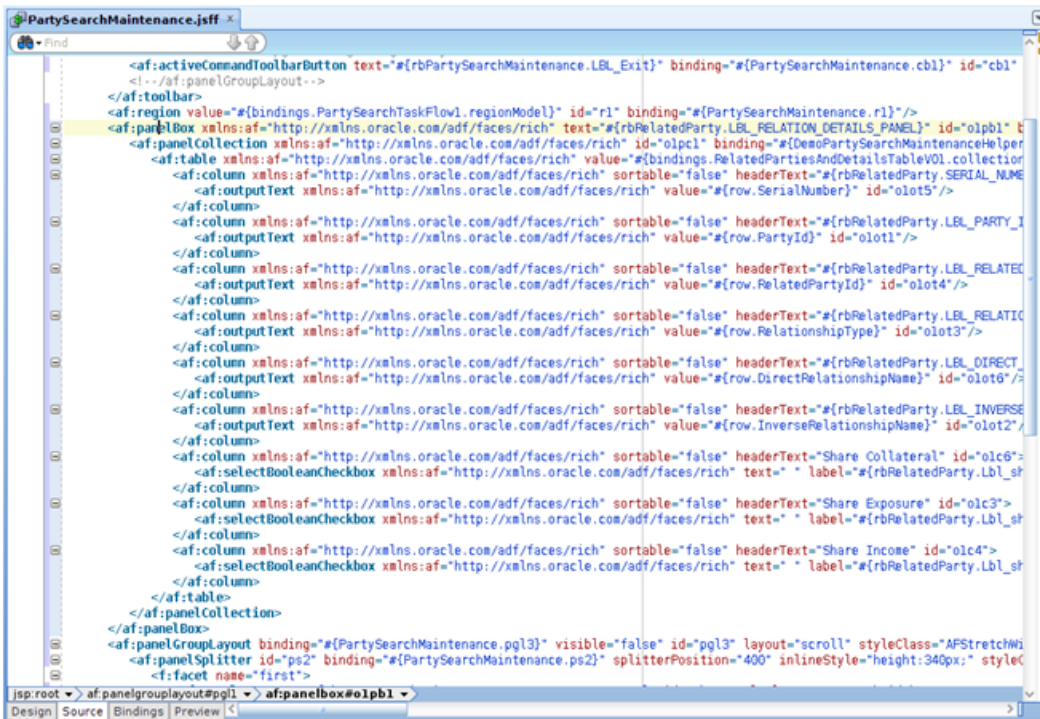


### Step 9 Add UI Components to Screen

After making the required changes to page definition of the screen, you will need to add the UI components to the screen JSFF. After opening the JSFF for the screen (*com.ofss.fc.ui.view.party.partySearch.PartySearchMaintenance.jsff*), follow these steps:

1. Drag and drop the *Panel Box*, *Panel Collection* and *Table* components onto the screen.
2. Set the required columns for the *Table* component.
3. Drag and drop the *Output Text* or *Check Box* components as required inside the columns.
4. For each component, set the required attributes using the *Property Inspector* panel of JDeveloper.
5. Add the binding for required components to the binding bean members.
6. Add the view object binding to the *Table* component.
7. Save changes made to the JSFF.

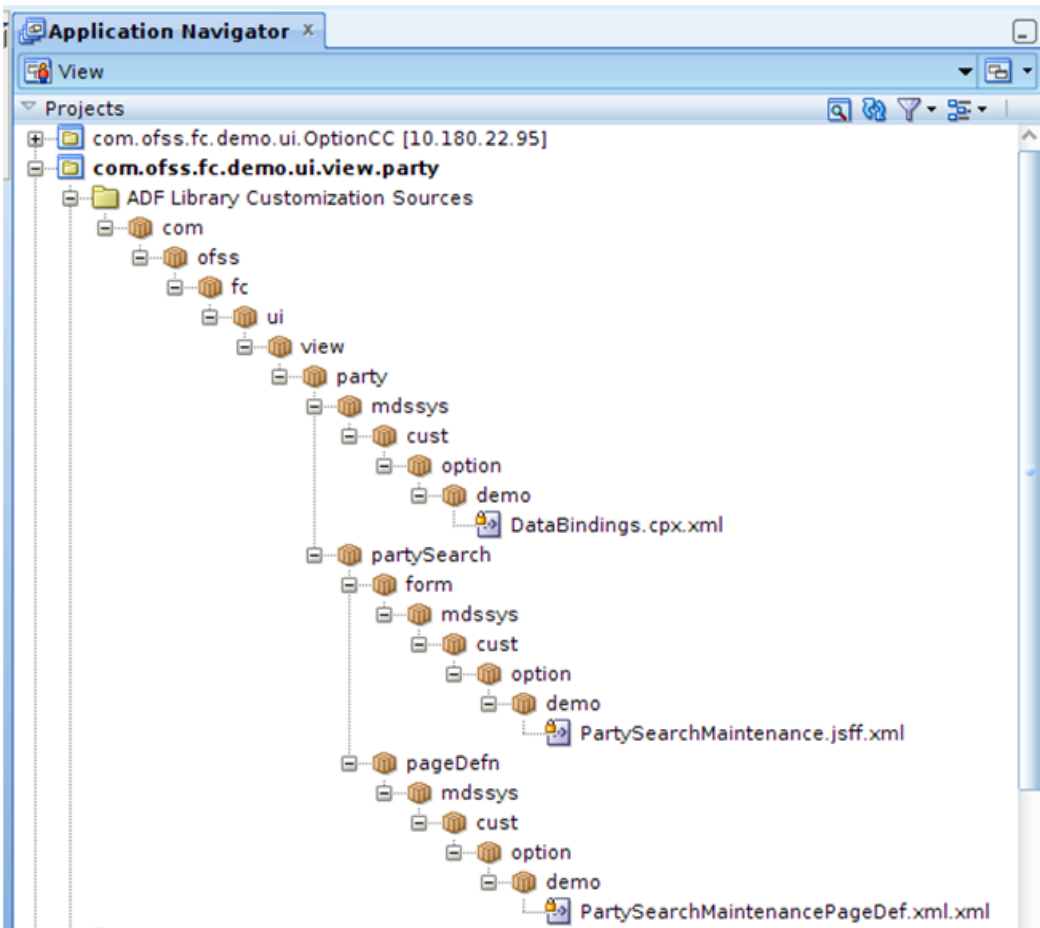
Figure 6–24 Add UI Components to Screen



After saving all these changes, you will notice that JDeveloper has created a customization XML for each of the customized entities in the *ADF Library Customizations Sources* folder packaged as per the corresponding base document's package and customization context (*Customization Layer Name & Customization Layer Value*). These XML's store the difference between the base and customized entity. In our customization, you can see the following generated XML's:

- PartySearchMaintenancePageDef.xml for the page definition customizations.
- DataBindings.cpx.xml for the data binding (view object binding) customizations.
- PartySearchMaintenance.jsff.xml for the UI customization to the screen JSFF.

Figure 6–25 Application Navigator



### Step 10 Deploy Customization Project

After finishing the customization changes, exit the *Customization Developer Role* and start JDeveloper in *Default Role*. Deploy the customization project as an ADF Library JAR (*com.ofss.fc.demo.ui.view.party.jar*).

1. Go to the **Project Properties** of the main application project and in the *Libraries* and *Classpath*, add the following JARS:
  - Customization Project JAR (*com.ofss.fc.demo.ui.view.party.jar*)
  - Customization Class JAR (*com.ofss.fc.demo.ui.OptionCC.jar*)
  - All dependency libraries and JARS for the project.
2. Start the application and navigate to the *Advanced Search* screen.
3. Search for a party ID and select a party from the *Party Search Results* table.
4. On selection of a party, the *Relation Details* panel containing the related party's data is displayed.

Figure 6–26 Party Search

The screenshot displays a web application interface for party search. It is divided into several sections:

- Search Individual:** Contains input fields for Party ID (000005296), Full Name, First name, Last Name, Short Name, and Email ID. Search and Reset buttons are present.
- Party Search Results:** A table with columns: Party ID, Name, Type, Number of Roles, Date of Birth or Incorporation, Party Class, and Email ID. One result is shown for Daniel Johnson.
- Relation Details:** A table with columns: Serial No., Party Id, Related Party ID, Relationship Type, Direct Relation Name, Inverse Relation Name, Share Collateral, Share Exposure, and Share Income. One relation is shown as Business.
- Account Details:** A table with columns: Serial Number, Account Number, and Account Type. One account is shown.
- Account Specific Details:** A detailed view of the account with fields for Account Number, Account Opening Date, Party ID, Offer, Facility Code, Total Disbursed Amount, Date of Maturity, Approved Amount, Outstanding Balance, Account Title, Account Currency, Party Name, Branch, Facility Name, Last Disbursement Date, Accrual Status, Next Instalment Amount, and Account Status.

## 6.7.2 Approvals Framework

It is recommended to use ADF screen extensions for UI changes instead of mds in this scenario as it is easier to upgrade to new version of product however the mds approach is described below.

This third example of customization explains adding a Date Component to an existing screen to capture date input from the input. This input is saved in the database.

**Use Case Description:** The Party → Contact Information → Contact Point screen is used to store the various contact point details for a party. In the Contact Point Details tab, the user can select a Contact Point Type and a Contact Preference Type and provide details for the same. User will be adding a field Expiry Date as a date component to this tab. User will be adding a table to the database to save the user input for this field and services for this screen will be added or modified.

Figure 6–27 Contact Point Screen

**PIQ41 Contact Point**

Read Create Update Ok Clear Exit Print

**Party Details**

Party ID 00005295 Date of Incorporation  
 Home Branch 082991-U Bank Operations BR  
 Company Name Daniel trustee Roles • Customer • Trustee  
 Party Class FOREIGN PUBLIC BODY  
 Party Type LEG Onboarding Date 15-Jan-2016

**Address Details**

**Contact Point Details**

Contact Point Type Mobile Contact Preference Type Home  
 Seasonal Start Date Seasonal End Date  
 Allowed Purposes  Communication  Alert  
 Preferred Contact  Preferred Contact  
 Marketing Consent  Marketing Consent  
 Marketing Consent Start Date Marketing Consent End Date

**Telephone Details**

Country Code Area Code  
 Number 32577789 Extension  
 Service Provider VOIP Code

**Timing Preferences**

DND  DND  
 DND Start DND End  
 Weekdays  Weekdays  
 From To  
 Weekends  Weekends  
 From To

**Hide Modification History**

Created By	ofssuser	On	24-Aug-2012 12:00:00 AM	Approved	<input checked="" type="checkbox"/>
Approved By	ofssuser	On	24-Aug-2012 12:00:00 AM	Active	<input checked="" type="checkbox"/>

2 OF

To create the customization as mentioned in this use case, follow these steps:

### Step 1 Host Application Changes

Since in this use case you need to save the input data in the database of the application, you need to do certain modifications on the host application before creating the customizations on the client application. Following are the changes that need to be done to the host application.

### Step 2 Create Table in Application Database

To save the input data for the Expiry Date field, create a table in the application database. The table will also need to have the Key columns for this field and the columns needed to store information about the record. Create appropriate primary and foreign keys for the table as well.

Figure 6–28 Create Table

```

CREATE TABLE "FLX_PI_CONTACT_EXPIRY"
(
  "PARTY_ID"          VARCHAR2(40 BYTE) NOT NULL ENABLE,
  "CONTACT_POINT_TYPE" VARCHAR2(3 BYTE) NOT NULL ENABLE,
  "CONTACT_PREF_TYPE" VARCHAR2(4 BYTE) NOT NULL ENABLE,
  "EXPIRY_DATE"      DATE,
  "CREATED_BY"       VARCHAR2(254 BYTE) NOT NULL ENABLE,
  "CREATION_DATE"    TIMESTAMP (6) NOT NULL ENABLE,
  "LAST_UPDATED_BY"  VARCHAR2(254 BYTE) NOT NULL ENABLE,
  "LAST_UPDATE_DATE" TIMESTAMP (6) NOT NULL ENABLE,
  "OBJECT_VERSION_NUMBER" NUMBER(9,0) NOT NULL ENABLE,
  "OBJECT_STATUS_FLAG" CHAR(1 BYTE) NOT NULL ENABLE,
  CONSTRAINT "FLX_PI_CONTACT_EXPIRY_PK" PRIMARY KEY ("PARTY_ID", "CONTACT_POINT_TYPE", "CONTACT_PREF_TYPE") ENABLE,
  CONSTRAINT "FLX_PI_CONTACT_EXPIRY_FK1" FOREIGN KEY ("PARTY_ID") REFERENCES "FLX_PI_PARTIES_B" ("PARTY_ID") ENABLE
);

```

Key Columns

Expiry Date Field

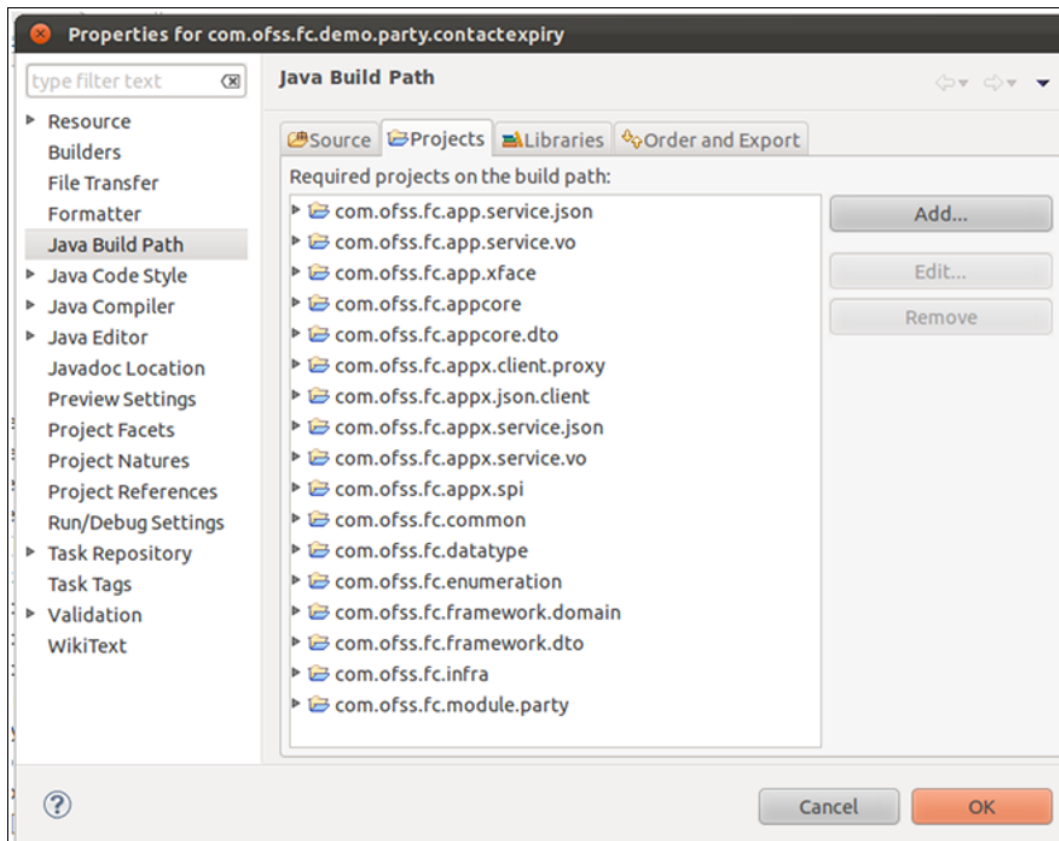
Record Information Columns

After creating the table, you need to create the domain object and service layers. To create these entities, follow these steps.

### Step 3 Create Java Project

To contain the domain object and service layer classes, create a Java Project in eclipse. Give a title to the project (com.ofss.fc.demo.party.contactexpiry) and add the required projects to the classpath of the project.

Figure 6–29 Create Java Project

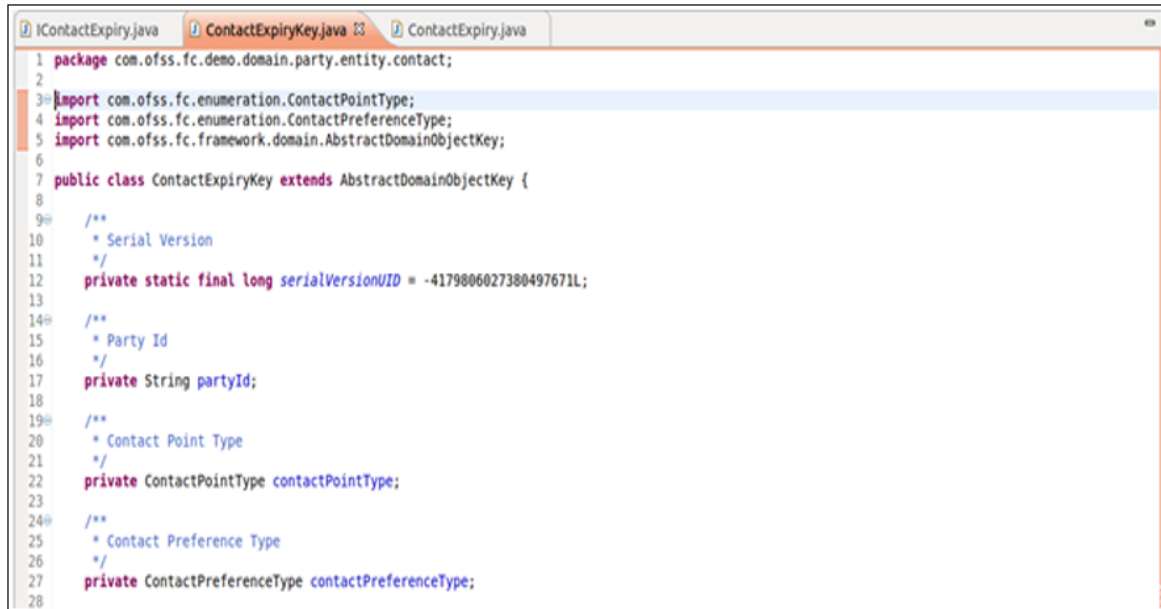


### Step 4 Create Domain Objects

You need to create the domain objects for the newly added table. As per the structure and package conventions of OBP, create the domain objects as follows:

1. Create class (`com.ofss.fc.demo.domain.party.entity.contact.ContactExpiryKey`) for the key columns of the table. This class must extend the `com.ofss.fc.framework.domain.AbstractDomainObject` abstract class. Add the properties, getters and setters for the key columns of the table in this class. Implement the abstract methods of the superclass.

Figure 6–30 Create Domain Objects



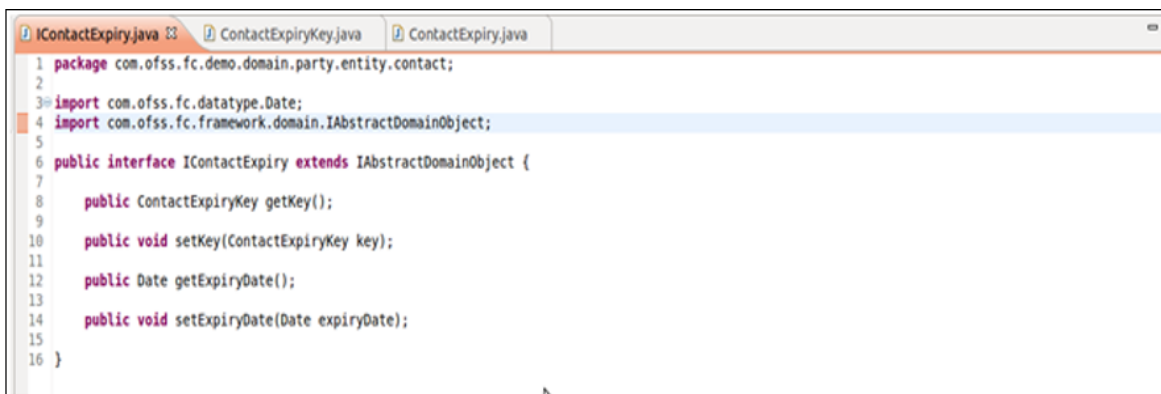
```

1 package com.ofss.fc.demo.domain.party.entity.contact;
2
3 import com.ofss.fc.enumeration.ContactPointType;
4 import com.ofss.fc.enumeration.ContactPreferenceType;
5 import com.ofss.fc.framework.domain.AbstractDomainObjectKey;
6
7 public class ContactExpiryKey extends AbstractDomainObjectKey {
8
9     /**
10      * Serial Version
11      */
12     private static final long serialVersionUID = -4179806027380497671L;
13
14     /**
15      * Party Id
16      */
17     private String partyId;
18
19     /**
20      * Contact Point Type
21      */
22     private ContactPointType contactPointType;
23
24     /**
25      * Contact Preference Type
26      */
27     private ContactPreferenceType contactPreferenceType;
28

```

2. Create interface (`com.ofss.fc.demo.domain.party.entity.contact.IContactExpiry`) for the domain object class with getters and setters abstract methods for the Key domain object and the field Expiry Date. This interface must extend the interface `com.ofss.fc.framework.domain.AbstractDomainObject`.

Figure 6–31 Create Interface



```

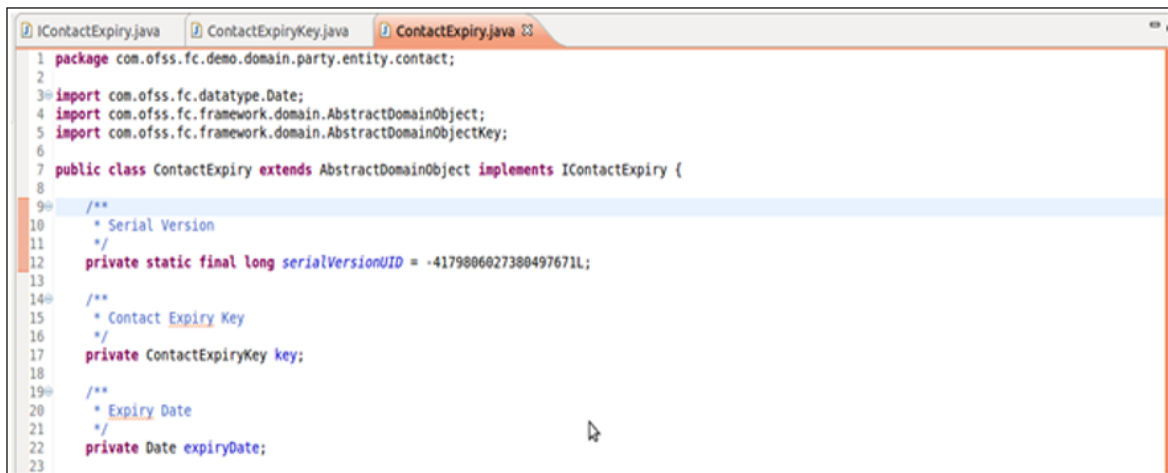
1 package com.ofss.fc.demo.domain.party.entity.contact;
2
3 import com.ofss.fc.datatype.Date;
4 import com.ofss.fc.framework.domain.IAbstractDomainObject;
5
6 public interface IContactExpiry extends IAbstractDomainObject {
7
8     public ContactExpiryKey getKey();
9
10    public void setKey(ContactExpiryKey key);
11
12    public Date getExpiryDate();
13
14    public void setExpiryDate(Date expiryDate);
15
16 }

```

3. Create class (`com.ofss.fc.demo.domain.party.entity.contact.ContactExpiry`) for the domain object. This class must implement the previously created interface and extend `com.ofss.fc.framework.domain.AbstractDomainObject` abstract class. Add the properties, getters and setters for Key object and Expiry Date field. Implement the abstract methods of the superclass.



Figure 6–32 Create Class



```

1 package com.ofss.fc.demo.domain.party.entity.contact;
2
3 import com.ofss.fc.datatype.Date;
4 import com.ofss.fc.framework.domain.AbstractDomainObject;
5 import com.ofss.fc.framework.domain.AbstractDomainObjectKey;
6
7 public class ContactExpiry extends AbstractDomainObject implements IContactExpiry {
8
9     /**
10      * Serial Version
11      */
12     private static final long serialVersionUID = -4179806027380497671L;
13
14     /**
15      * Contact Expiry Key
16      */
17     private ContactExpiryKey key;
18
19     /**
20      * Expiry Date
21      */
22     private Date expiryDate;
23

```

After creating the domain objects, build the project. You need to use the Flex cube development eclipse plug-in to generate the service layers.

### Step 5 Set OBP Plugin Preferences

Before using the plug-in for generating service layer classes, you will need to set the required preferences for the plug-in. In eclipse, go to Windows → Preferences → OBP Development and the set the preferences as follows.

Figure 6–33 Set OBP Plugin Preferences

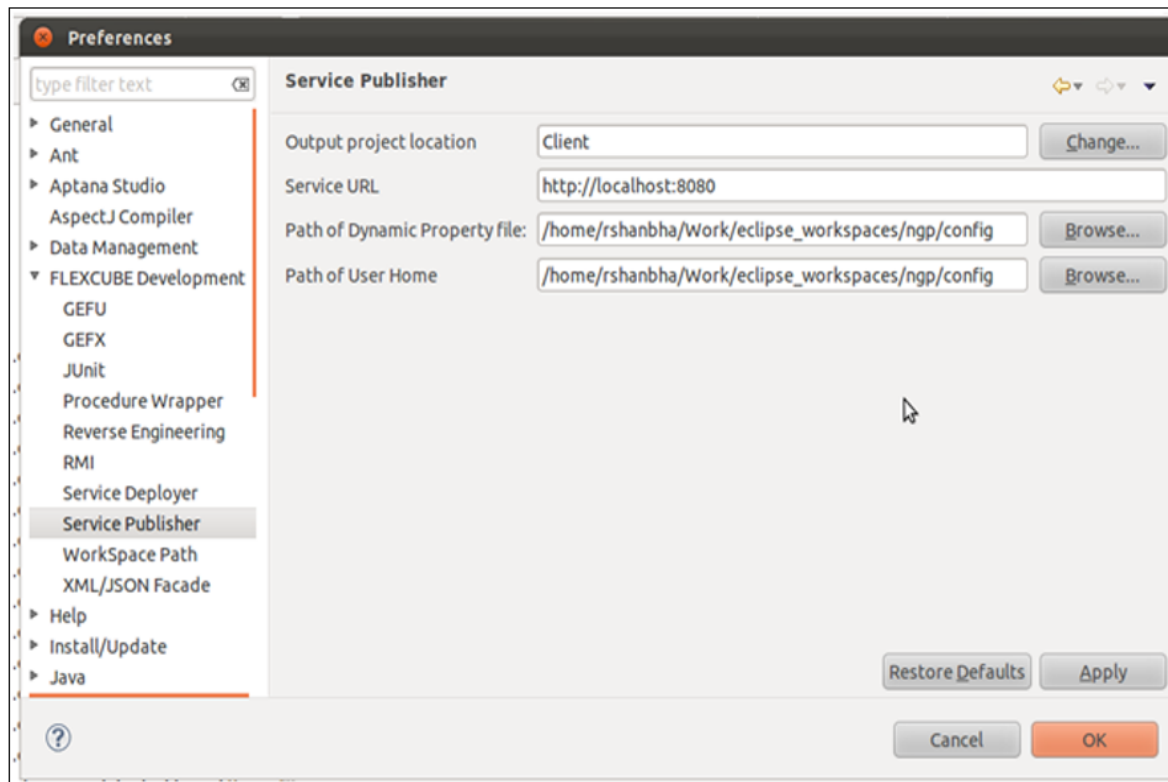


Figure 6–34 Set OBP Plugin Preferences

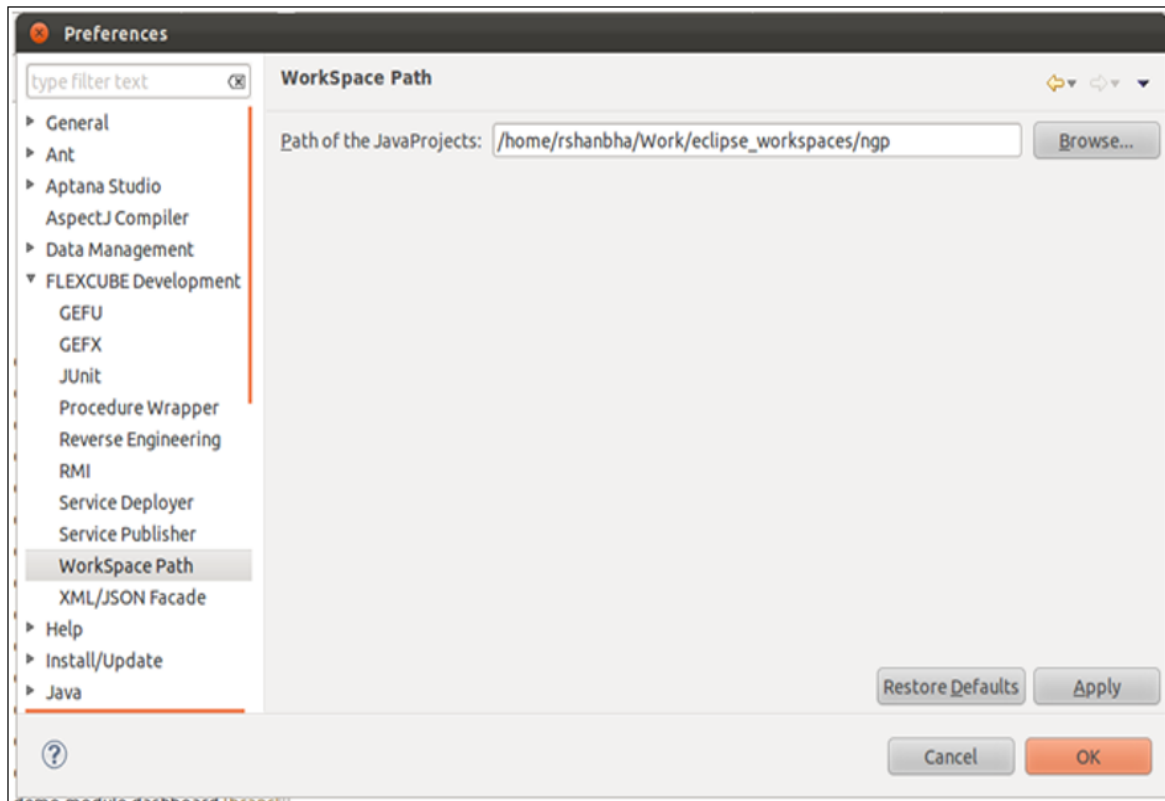
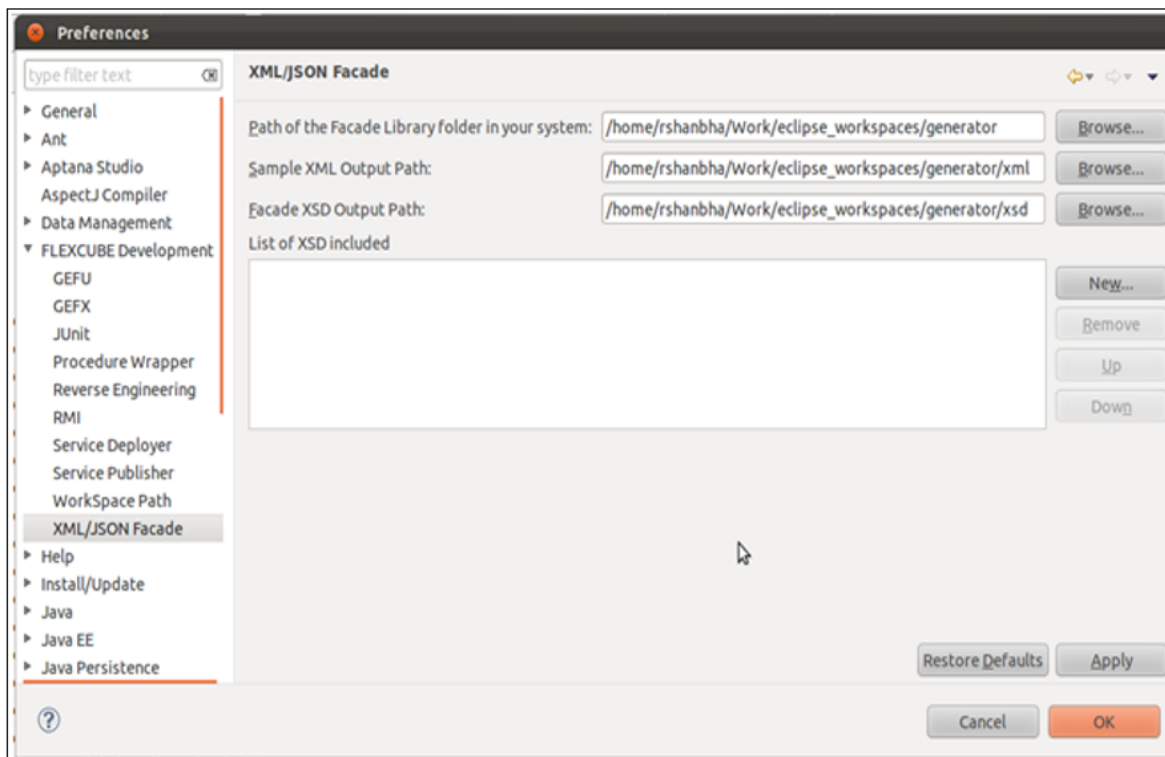


Figure 6–35 Set OBP Pugin Preferences

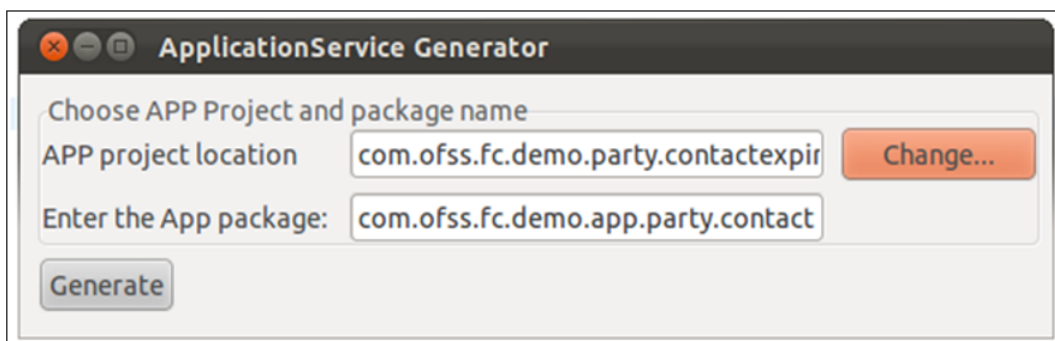


### Step 6 Create Application Service

You need to generate the application service layer classes using the OBP development plugin. Follow these steps:

1. Open the domain object class (ContactExpiry).
2. On the getter method of the Key object, add a javadoc comment @PK.
3. Right click on the editor window and from context menu that opens, choose OBP Development → Generate Application Service.
4. In the dialog that opens, select the Java project for generated classes. You can use the project previously created by you.

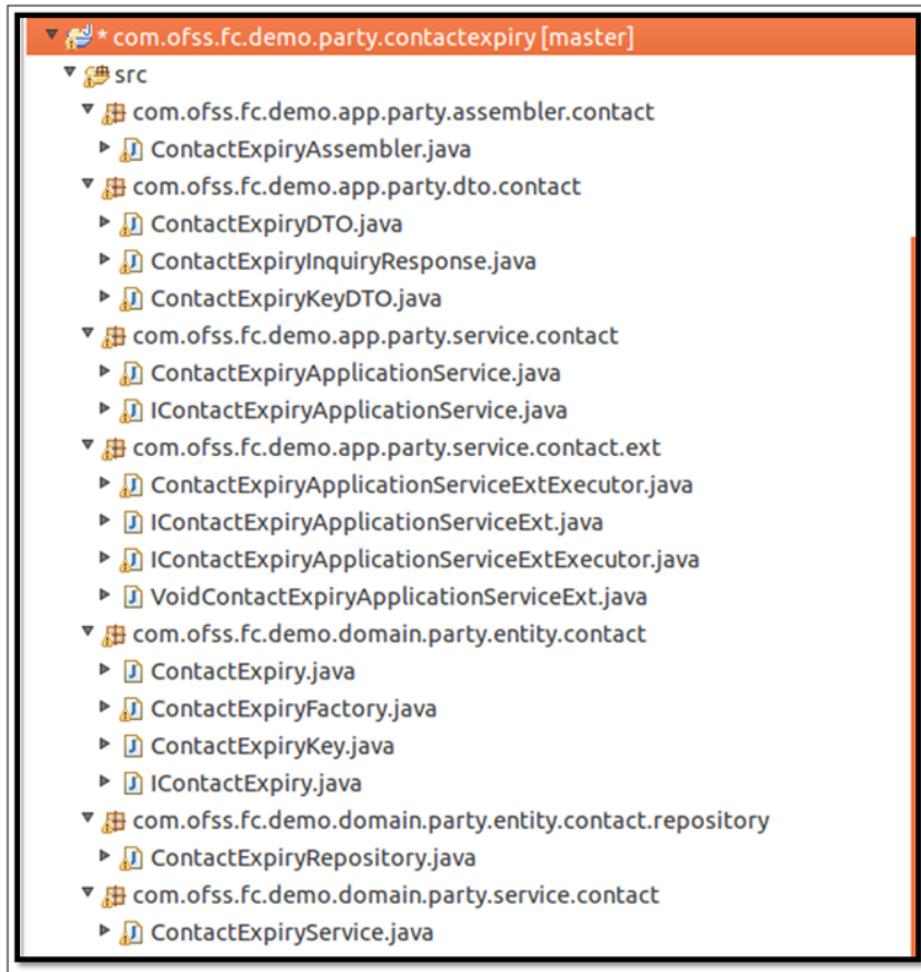
*Figure 6–36 Create Application Service*



5. Click on Generate. Application Service classes is generated in the project.

The Java source might contain some compilation errors due to syntax. Fix these errors and build the project. The following classes should have been generated in the project.

Figure 6–37 Application Service Classes Generated



### Step 7 Generate Service and Facade Layer Sources

Before generating the service and facade layer sources, you need to modify the Data Transfer Object (DTO). When a service call is made from the client application for a transaction related to Contact Point, the Contact Expiry transaction for the newly added Expiry Date field should be done in addition to the Contact Point transaction. Hence, the DTO for this transaction should also contain the DTO for the Contact Point transaction.

To modify the Data Transfer Object:

1. Open the ContactExpiryDTO class.
2. Delete the member ContactExpiryKey member and add ContactPoint member.
3. Re-factor references of the deleted member with the added member.

Figure 6–38 Modify Data Transfer Object (DTO)

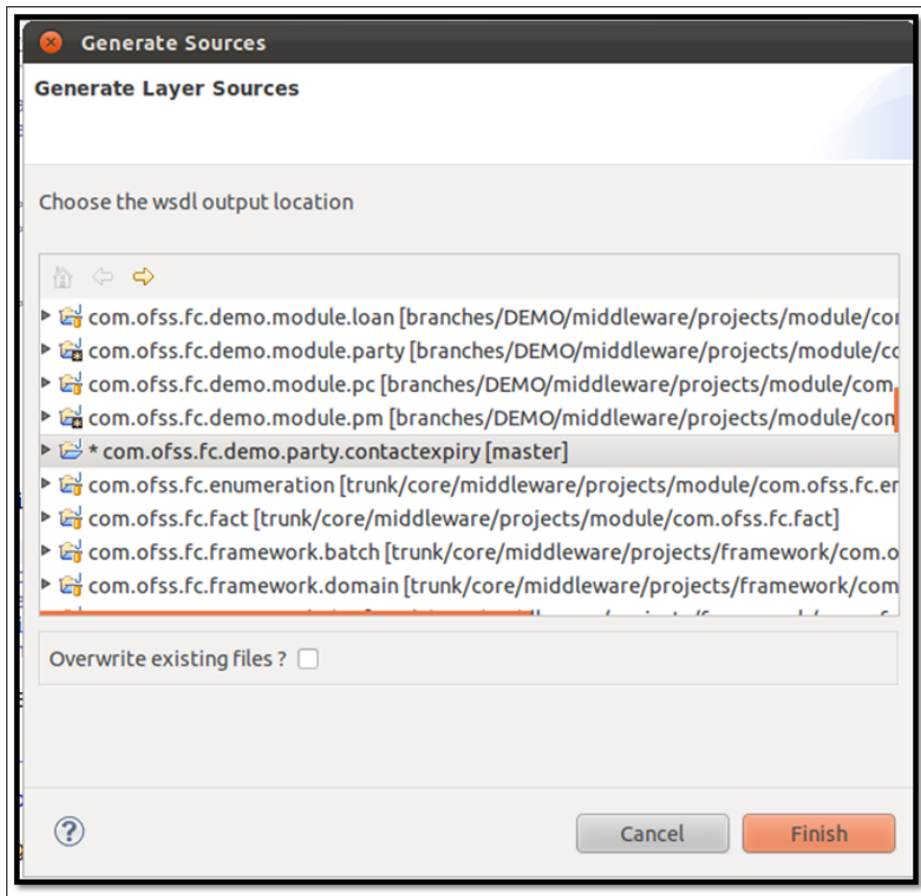
```

37 * This class represents the DTO for the ContactExpiry entity.
38 * TODO: Auto generated comment: Please write detailed description for this class here.
39 * @author rshanhba
40 * @version 1.0
41 */
42 public class ContactExpiryDTO extends DomainObjectDTO {
43
44     private static final long serialVersionUID = 1L;
45
46     /**
47      * Contact Point DTO which has the key for Contact Expiry DTO
48      */
49     private ContactPointDTO contactPointDTO;
50
51     /**
52      * This indicates the expiryDate attribute
53      */
54     private Date expiryDate;
55
56     /**
57      * This represents the constructor for the class.
58      * @fcb.param MI Date expiryDate This indicates the expiryDate attribute
59      * @fcb.param MI ContactPreferenceType contactPreferenceType This indicates the contactPreferenceType attribute
60      * @fcb.param MI ContactPointType contactPointType This indicates the contactPointType attribute
61      * @fcb.param MI String partyId This indicates the partyId attribute
62      */
63     public ContactExpiryDTO(Date expiryDate, ContactPointDTO contactPointDTO) {
64         setContactPointDTO(contactPointDTO);
65         setExpiryDate(expiryDate);
66     }
67
68     /**
69      * This represents the constructor for the class.
70      * @fcb.param MI
71      */
72     public ContactExpiryDTO() {
73     }
74
75     /**
76      * This method returns PartyId
77      * @return partyId
78      */
79     public ContactPointDTO getContactPointDTO(){
80         return contactPointDTO;
81     }
82
83     /**

```

To generate the service and facade layer sources:

1. Open the application service class (ContactExpiryApplicationService).
2. Right click on the editor window and from the context menu that opens, choose OBP Development → Generate Service and Facade Layer Sources.
3. In the dialog box that opens, select the Java project for the generated classes. You can use the project previously created by you. Un-check the Overwrite Existing Files option.

**Figure 6–39 Generate Service and Facade Layer Sources**

4. Click Finish.

Service and facade layer sources is generated in the project.

5. Certain classes might be generated twice. Delete the newly created copy of the classes and keep the original.
6. Certain compilation errors might be present in the generated classes due to erroneous syntax. Fix these compilation errors.
7. You will need to include a corresponding call to the Contact Point Application Service in the add, update and fetch transactions of the Contact Expiry Application Service.
8. Open ContactExpiryApplicationServiceSpi and modify the code as shown below.

Figure 6–40 Modify ContactExpiryApplicationServiceSpi.java

```
72     .getName();
73
74     private transient Logger logger = MultiEntityLogger.getUniqueInstance()
75     .getLogger(THIS_COMPONENT_NAME);
76
77     public TransactionStatus addContactExpiry(
78         com.ofss.fc.app.context.SessionContext sessionContext,
79         ContactExpiryDTO contactExpiryDTO, FeeDetailsDTO feeDetails,
80         LinkedUDFDTO linkedUDFDTO) throws FatalException {
81
82         com.ofss.fc.demo.app.party.service.contact.ContactExpiryApplicationService manager = new com.ofss.fc.demo.app.party.service.contact Contac
83
84         Interaction.begin(sessionContext);
85
86         com.ofss.fc.demo.appx.party.service.contact.ext.IContactExpiryApplicationServiceSpiExt helper = (com.ofss.fc.demo.appx.party.service.conta
87         .getInstance()
88         .getServiceProviderExtension(
89             "com.ofss.fc.demo.appx.party.service.contact.ext.IContactExpiryApplicationServiceSpiExt",
90             "com.ofss.fc.demo.appx.party.service.contact.ext.ContactExpiryApplicationServiceSpiExt");
91
92         TransactionStatus transactionStatus = fetchTransactionStatus();
93         String taskCode = null;
94         try {
95             helper.preAddContactExpiry(sessionContext, contactExpiryDTO,
96                 feeDetails, linkedUDFDTO);
97
98         /* Code added for Contact Point transaction */
99         ContactPointApplicationServiceSpi cpServiceSpi = new ContactPointApplicationServiceSpi();
100         transactionStatus = cpServiceSpi.createContactPoint(sessionContext,
101             contactExpiryDTO.getContactPointDTO(), feeDetails,
102             linkedUDFDTO);
103         /* Code added for Contact Point transaction */
104
105         transactionStatus = manager.addContactExpiry(sessionContext,
106             contactExpiryDTO);
107         taskCode = Interaction.fetchCurrentTask();
108         transactionStatus = applyServiceCharge(sessionContext, feeDetails,
109             taskCode);
110         fillTransactionStatus(transactionStatus);
111         Map<String, Object> parentDTOMap = new HashMap<String, Object>();
112         transactionStatus = addUDF(sessionContext, linkedUDFDTO,
113             parentDTOMap);
114         fillTransactionStatus(transactionStatus);
115
116         helper.postAddContactExpiry(sessionContext, contactExpiryDTO,
117             feeDetails, linkedUDFDTO);
118     }
```

Figure 6–41 Modify ContactExpiryApplicationServiceSpi.java

```
128     }
129     return transactionStatus;
130 }
131
132 public TransactionStatus updateContactExpiry(
133     com.ofss.fc.app.context.SessionContext sessionContext,
134     ContactExpiryDTO contactExpiryDTO, FeeDetailsDTO feeDetails,
135     LinkedUDFDTO linkedUDFDTO) throws FatalException {
136
137     com.ofss.fc.demo.app.party.service.contact.ContactExpiryApplicationService manager = new com.ofss.fc.demo.app.party.service.contact.Contact
138
139     Interaction.begin(sessionContext);
140
141     com.ofss.fc.demo.appx.party.service.contact.ext.IContactExpiryApplicationServiceSpiExt helper = (com.ofss.fc.demo.appx.party.service.conta
142     .getInstance()
143     .getServiceProviderExtension(
144         "com.ofss.fc.demo.appx.party.service.contact.ext.IContactExpiryApplicationServiceSpiExt",
145         "com.ofss.fc.demo.appx.party.service.contact.ext.ContactExpiryApplicationServiceSpiExt");
146
147     TransactionStatus transactionStatus = fetchTransactionStatus();
148     String taskCode = null;
149     try {
150         helper.preUpdateContactExpiry(sessionContext, contactExpiryDTO,
151             feeDetails, linkedUDFDTO);
152
153         /* Code added for Contact Point transaction */
154         ContactPointApplicationServiceSpi cpServiceSpi = new ContactPointApplicationServiceSpi();
155         transactionStatus = cpServiceSpi.updateContactPoint(sessionContext,
156             contactExpiryDTO.getContactPointDTO(), feeDetails,
157             linkedUDFDTO);
158         /* Code added for Contact Point transaction */
159
160         transactionStatus = manager.updateContactExpiry(sessionContext,
161             contactExpiryDTO);
162         taskCode = Interaction.fetchCurrentTask();
163         transactionStatus = applyServiceCharge(sessionContext, feeDetails,
164             taskCode);
165         fillTransactionStatus(transactionStatus);
166         Map<String, Object> parentDTOMap = new HashMap<String, Object>();
167         transactionStatus = updateUDF(sessionContext, linkedUDFDTO,
168             parentDTOMap);
169         fillTransactionStatus(transactionStatus);
170
171         helper.postUpdateContactExpiry(sessionContext, contactExpiryDTO,
172             feeDetails, linkedUDFDTO);
173
174         fillTransactionStatus(transactionStatus);
```



Figure 6–42 Modify ContactExpiryApplicationServiceSpi.java

```

182     Interaction.close();
183 }
184     return transactionStatus;
185 }
186
187 public ContactExpiryInquiryResponse fetchContactExpiry(
188     com.ofss.fc.app.context.SessionContext sessionContext,
189     ContactExpiryDTO contactExpiryDTO, FeeDetailsDTO feeDetails,
190     LinkedUDFDTO linkedUDFDTO) throws FatalException {
191
192     com.ofss.fc.demo.app.party.service.contact.ContactExpiryApplicationService manager = new com.ofss.fc.demo.app.party.service.contact.Contact
193
194     Interaction.begin(sessionContext);
195
196     com.ofss.fc.demo.appx.party.service.contact.ext.IContactExpiryApplicationServiceSpiExt helper = (com.ofss.fc.demo.appx.party.service.conta
197     .getInstance()
198     .getServiceProviderExtension(
199         "com.ofss.fc.demo.appx.party.service.contact.ext.IContactExpiryApplicationServiceSpiExt",
200         "com.ofss.fc.demo.appx.party.service.contact.ext.ContactExpiryApplicationServiceSpiExt");
201
202     ContactExpiryInquiryResponse response = null;
203     TransactionStatus transactionStatus = fetchTransactionStatus();
204     String taskCode = null;
205     try {
206         helper.preFetchContactExpiry(sessionContext, contactExpiryDTO,
207             feeDetails, linkedUDFDTO);
208
209         response = manager.fetchContactExpiry(sessionContext,
210             contactExpiryDTO);
211
212         /* Code added for Contact Point transaction */
213         ContactPointApplicationServiceSpi cpServiceSpi = new ContactPointApplicationServiceSpi();
214         ContactPointResponse cpResponse = cpServiceSpi.fetchContactPoint(
215             sessionContext, contactExpiryDTO.getContactPointDTO(),
216             feeDetails, linkedUDFDTO);
217         if (cpResponse.getContactPoints() != null
218             && cpResponse.getContactPoints().length != 0) {
219             response.getContactExpiryDTO().setContactPointDTO(
220                 cpResponse.getContactPoints()[0]);
221         }
222         /* Code added for Contact Point transaction */
223
224         taskCode = Interaction.fetchCurrentTask();
225         transactionStatus = applyServiceCharge(sessionContext, feeDetails,
226             taskCode);
227         fillTransactionStatus(transactionStatus);
228         Map<String, Object> parentDTOMap = new HashMap<String, Object>();

```

- The project should contain the Java packages as shown below:

Figure 6–43 Java Packages

```

com.ofss.fc.demo.party.contactexpiry [master]
├── src
│   ├── com.ofss.fc.demo.app.party.assembler.contact
│   ├── com.ofss.fc.demo.app.party.dto.contact
│   ├── com.ofss.fc.demo.app.party.service.contact
│   ├── com.ofss.fc.demo.app.party.service.contact.ext
│   ├── com.ofss.fc.demo.app.party.service.contact.service.client.proxy
│   ├── com.ofss.fc.demo.app.party.service.contact.service.json
│   ├── com.ofss.fc.demo.app.party.service.contact.service.json.client
│   ├── com.ofss.fc.demo.app.party.service.contact.vo
│   ├── com.ofss.fc.demo.appx.party.service.contact
│   ├── com.ofss.fc.demo.appx.party.service.contact.ext
│   ├── com.ofss.fc.demo.appx.party.service.contact.service.client.proxy
│   ├── com.ofss.fc.demo.appx.party.service.contact.vo
│   ├── com.ofss.fc.demo.appx.party.service.contact.vo.service.json
│   ├── com.ofss.fc.demo.appx.party.service.contact.vo.service.json.client
│   ├── com.ofss.fc.demo.domain.party.entity.contact
│   ├── com.ofss.fc.demo.domain.party.entity.contact.repository
│   └── com.ofss.fc.demo.domain.party.service.contact

```

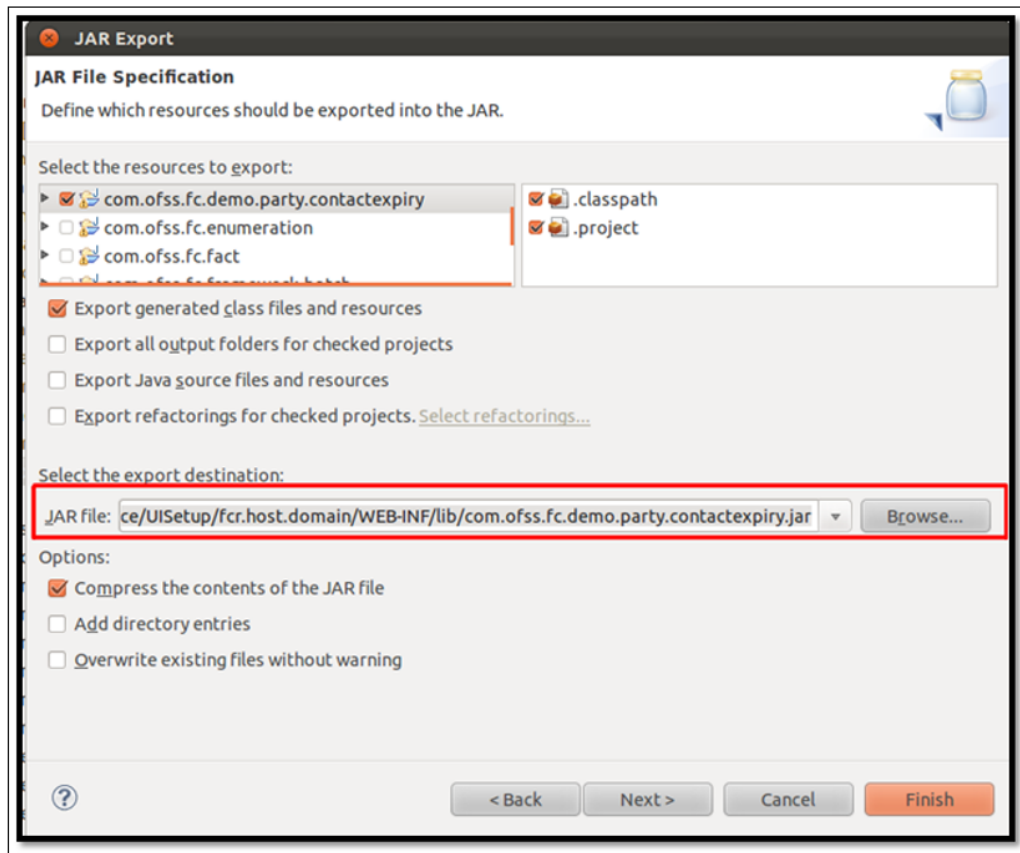
### Step 8 Export Project as a JAR

You need to export the Java project containing the domain object, application service and facade layer source as a JAR.

To export java project as JAR:

1. Right click on the project and choose Export.
2. Choose JAR File in the export options.
3. Provide an export path and name (com.ofss.fc.demo.party.contactexpiry.jar) for the JAR file.
4. Click Finish.

**Figure 6–44 Export Java Project as JAR**



### Step 9 Create Hibernate Mapping

You need to create a hibernate mapping to map the database table to the domain object.

Follow these steps:

1. Create ContactExpiry.hbm.xml file in the orm/hibernate/hbm folder of the config project of the host application.
2. Add the entry for this XML in the orm/hibernate/cfg/party-mapping.cfg.xml hibernate configuration XML.
3. Add the mapping in ContactExpiry.hbm.xml as shown below.

Figure 6–45 Create ContactExpiry.hbm.xml

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!-- Copyright (c) 2012, Oracle and/or its affiliates. All rights reserved. -->
3 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD/EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4 <hibernate-mapping auto-import="true" default-access="field" default-cascade="none" default-lazy="false">
5   <class dynamic-insert="true" dynamic-update="true" lazy="false" mutable="true"
6     name="com.ofss.fc.demo.domain.party.entity.contact.ContactExpiry" optimistic-lock="version" polymorphise="implicit"
7     select-before-update="false" table="FLX_PI_CONTACT_EXPIRY">
8     <composite-id class="com.ofss.fc.demo.domain.party.entity.contact.ContactExpiryKey" mapped="false" name="key" unsaved-value="undefined">
9       <key-property column="party_id" name="partyId" type="string"/>
10      <key-property column="contact_point_type" name="contactPointType">
11        <type name="com.ofss.fc.infra.enumeration.EnumUserType">
12          <param name="Enum">com.ofss.fc.enumeration.ContactPointType</param>
13        </type>
14      </key-property>
15      <key-property column="contact_pref_type" name="contactPreferenceType">
16        <type name="com.ofss.fc.infra.enumeration.EnumUserType">
17          <param name="Enum">com.ofss.fc.enumeration.ContactPreferenceType</param>
18        </type>
19      </key-property>
20    </composite-id>
21    <version column="OBJECT_VERSION_NUMBER" generated="never" name="version" type="java.lang.Integer" unsaved-value="undefined"/>
22    <property column="EXPIRY_DATE" generated="never" lazy="false" name="expiryDate" optimistic-lock="true"
23      type="com.ofss.fc.datatype.Date" unique="false"/>
24    <property column="created_by" generated="never" lazy="false" name="createdBy" optimistic-lock="true" type="string" unique="false"/>
25    <property column="creation_date" generated="never" lazy="false" name="creationDate" optimistic-lock="true"
26      type="com.ofss.fc.datatype.Date" unique="false"/>
27    <property column="last_updated_by" generated="never" lazy="false" name="lastUpdatedBy" optimistic-lock="true"
28      type="string" unique="false"/>
29    <property column="last_update_date" generated="never" lazy="false" name="lastUpdatedDate" optimistic-lock="true"
30      type="com.ofss.fc.datatype.Date" unique="false"/>
31    <property column="object_status_flag" generated="never" lazy="false" name="entityStatus" optimistic-lock="true"
32      type="EntityStatus" unique="false"/>
33  </class>
34 </hibernate-mapping>

```

### Step 10 Configure Host Application Project

You need to configure the Contact Expiry Application Service and Facade Layer in the host application.

To configure, follow these steps:

1. Configure APPX layer as the service layer for Contact Expiry service.
2. Open properties/hostapplicationlayer.properties present in the configuration project and add an entry as shown below.

Figure 6–46 Configure hostapplicationlayer.properties

```

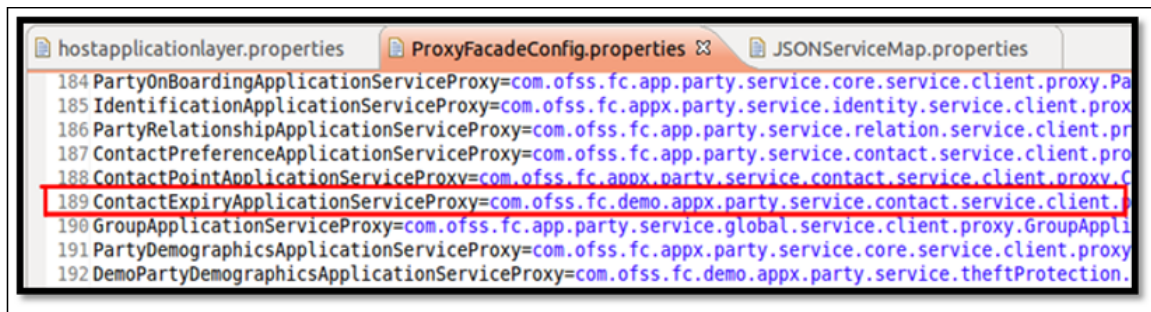
hostapplicationlayer.properties
ProxyFacadeConfig.properties
JSONServiceMap.properties
6 PartyAddressApplicationServiceProxy=APPX
7 CreditAssessmentApplicationServiceProxy=APPX
8 PartyNameApplicationServiceProxy=APPX
9 IdentificationApplicationServiceProxy=APPX
10 EmploymentHistoryApplicationServiceProxy=APPX
11 ContactPointApplicationServiceProxy=APPX
12 ContactExpiryApplicationServiceProxy=APPX
13 PartyDemographicsApplicationServiceProxy=APPX
14 PartyAccountRelationshipApplicationServiceProxy=APPX
15 CommentApplicationServiceProxy=APPX
16 BlacklistReportApplicationServiceProxy=APPX

```

3. Configure APPX layer proxy as the proxy for Contact Expiry service.
4. Open properties/ProxyFacadeConfig.properties present in the configuration project and add an entry as

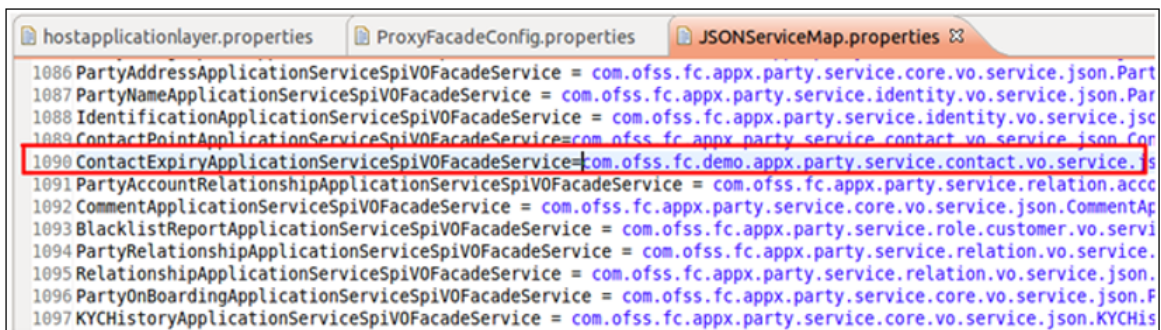
shown below.

**Figure 6–47 Configure ProxyFacadeConfig.properties**



5. Configure the JSON and Facade layer mapping for Contact Expiry service.
6. Open properties/JSONServiceMap.properties present in the configuration project and add the two entries as shown below.

**Figure 6–48 Configure JSONServiceMap.properties**



### Step 11 Deploy Project

After performing all the above mentioned changes, deploy the project as follows:

1. Add this project (com.ofss.fc.demo.party.contactexpiry) to the classpath of the branch application project.
2. Open the launch configuration of the Tomcat Server. Add this project to the classpath of the server as well.
3. Deploy the branch application project on the server and start it.
4. Client Application Changes.

After creating database table to hold the input data and after creating the related domain objects and service and facade layers, you can customize the user interface. The customizations to the application have to be done on the client application. To customize the UI, follow these steps.

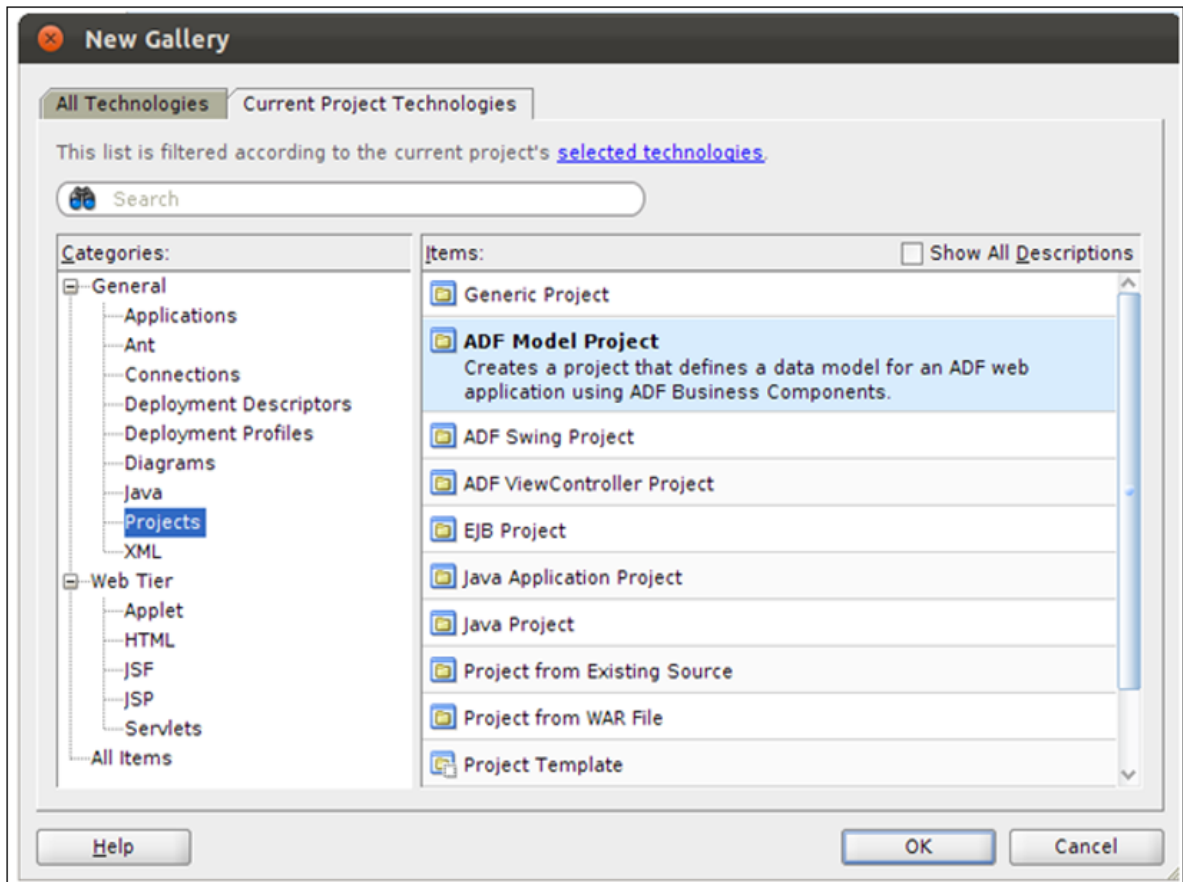
### Step 12 Create Model Project

You need to create a model project to hold the required view objects and application module.

To create the model project, follow these steps:

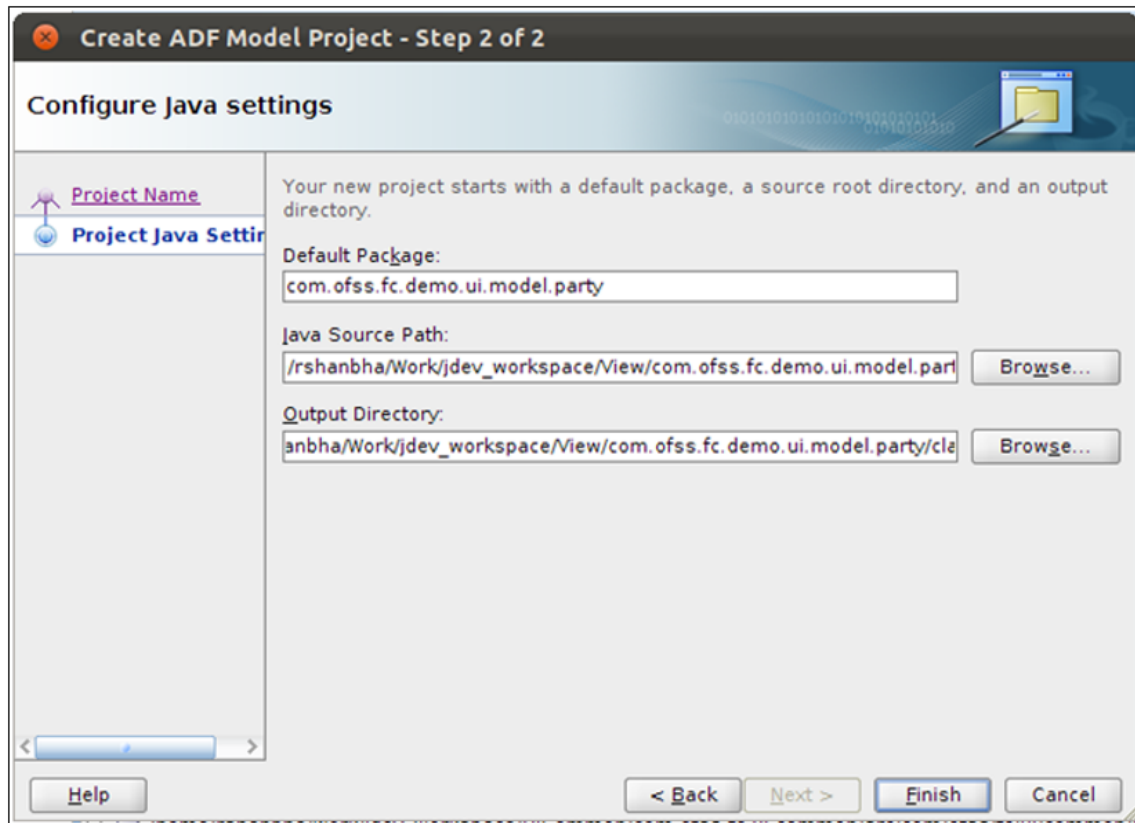
1. In the client application, create a new project of the type ADF Model Project.

**Figure 6–49 Create Model Project**



2. Give the project a title (com.ofss.fc.demo.ui.model.party) and set the default package as the same.

Figure 6–50 Create Model Project - Configure Java Settings



3. Click on Finish to create the project.

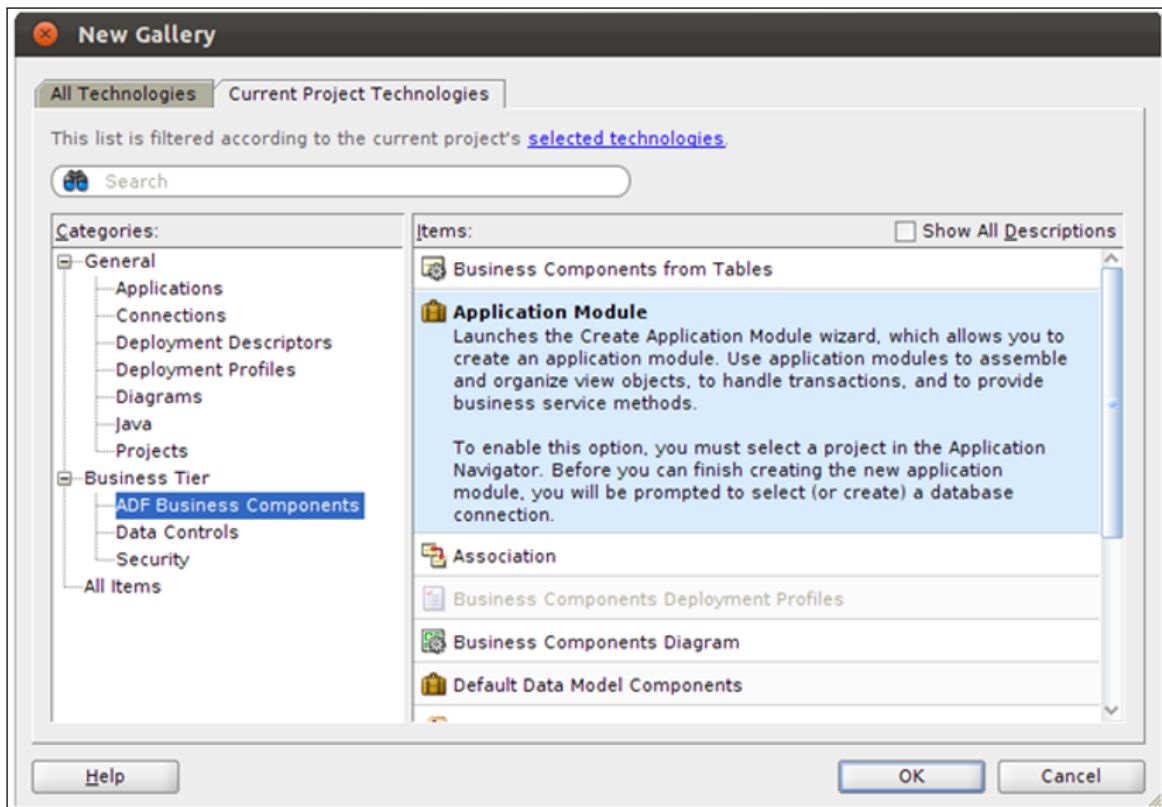
### Step 13 Create Application Module

You need to create an application module to contain the information of all the view objects that you need to create. To create an application module, follow these steps:

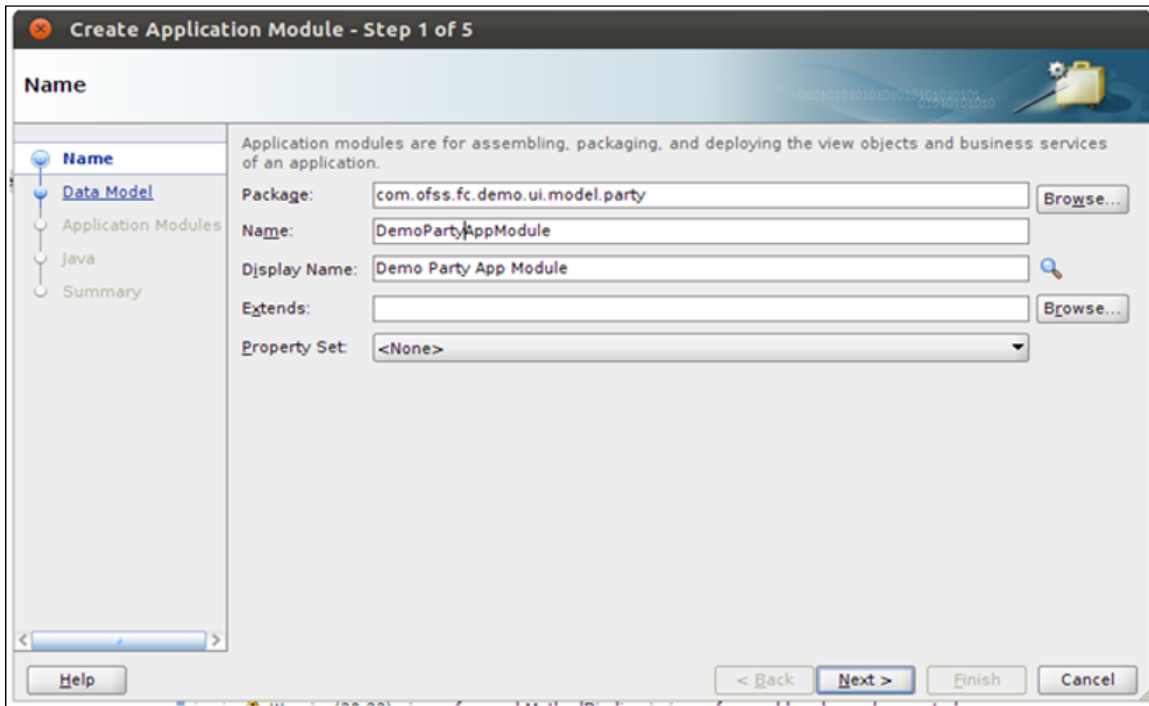
1. Right click on the model project and select New.
2. Choose Application Module from the dialog box that opens.



Figure 6–51 Create Application Module



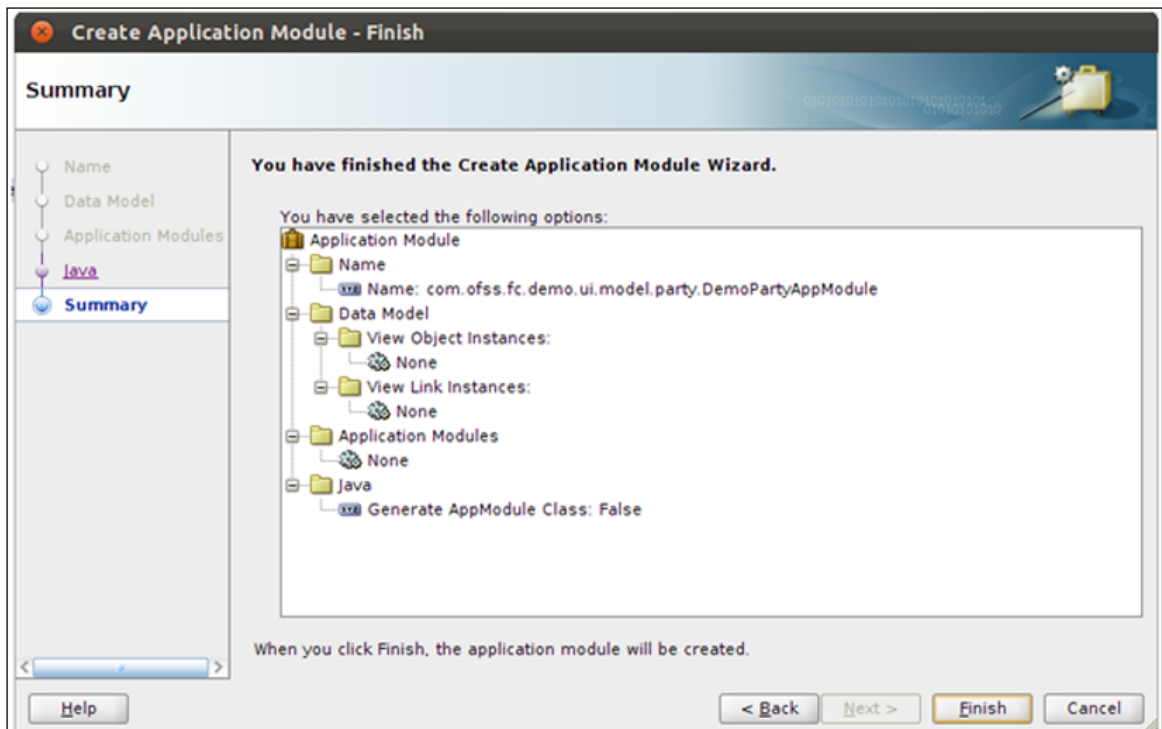
3. Set the package of the application module to the default package (com.ofss.fc.demo.ui.model.party).
4. Provide a name to the application module (DemoPartyAppModule).

**Figure 6–52 Set Package and Name of Application Module**

5. Click on Next and let the rest of the options be set to the default options.
6. You will see a summary screen for the application module. Click on Finish to create the application module.



Figure 6–53 Summary of Application Module Created



#### Step 14 Create View Object

You need to create a view object for the newly added Expiry Date field. This view object is used on the screen to display the value of the field as well as to take the input for the field.

To create the view object, follow these steps:

1. Right click on the Java package `com.ofss.fc.demo.ui.model.party` and select New View Object.
2. In the dialog box that opens, provide a name (`ContactExpiryVO`) for the view object.
3. Provide a package (`com.ofss.fc.demo.ui.model.party.contactexpiry`) for the view object.
4. For the Data Source Type option, select Rows populated programmatically, not based on a query.
5. Click on Next.
6. In the Attributes dialog, create a new attribute for Expiry Date field.
7. Provide a name (`ExpiryDate`) and type (`Date`) for the attribute.
8. For the Updatable option, select Always.

Figure 6–54 Create View Object

**Create View Object - Step 1 of 9**

**Name**

View objects are for joining, filtering, projecting, and sorting your business data for the specific needs of a given application task.

Package:

Name:

Display Name:

Extends:

Property Set:

Select the data source type you want to use as the basis for this view object.

Updatable access through entity objects

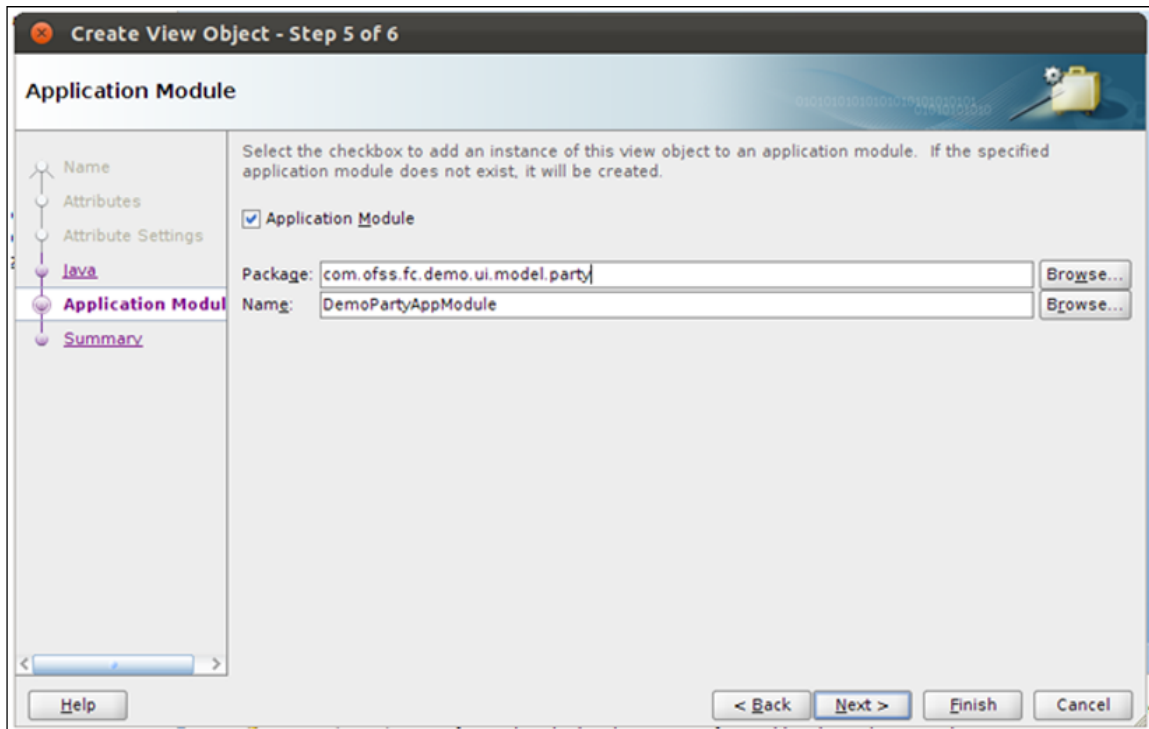
Read-only access through SQL query

Rows populated programmatically, not based on a query

Rows populated at design time (Static List)

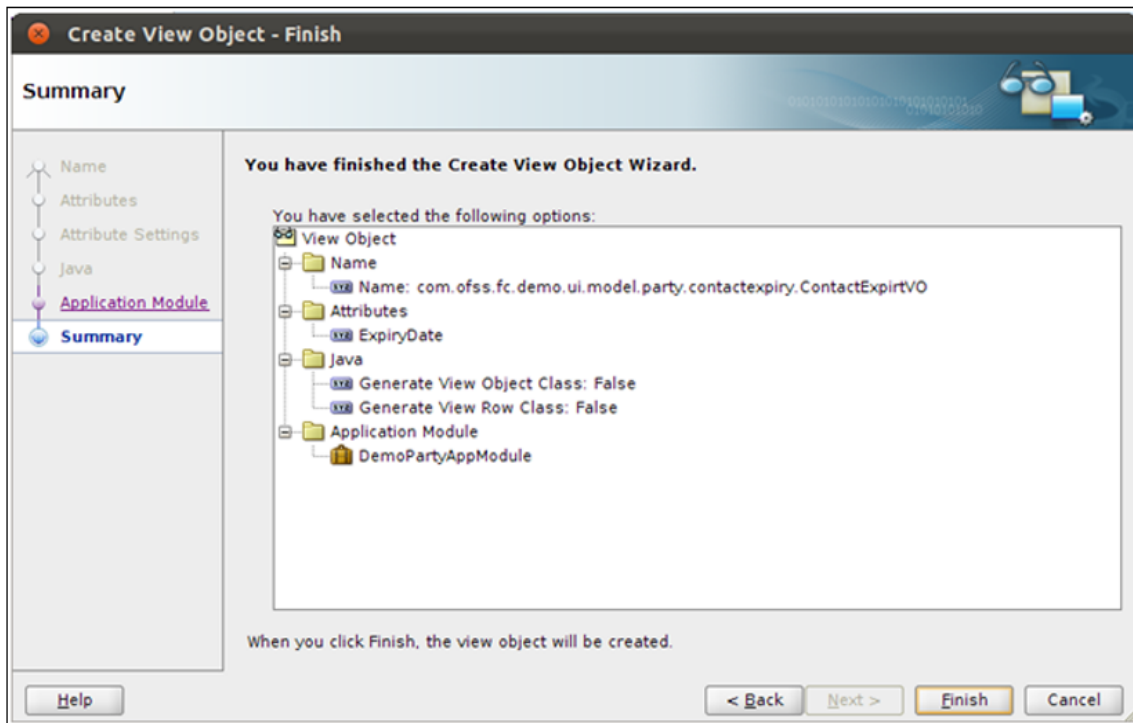
Figure 6–55 View Attribute

9. Click Next. On the Application Module dialog, browse for the previously created DemoPartyAppModule.

**Figure 6–56 Application Module**

10. For all other dialogs, keep the default options.
11. Click Next till you reach the summary screen as shown below.
12. Click on Finish to create the view object.

Figure 6–57 Create View Object - Summary



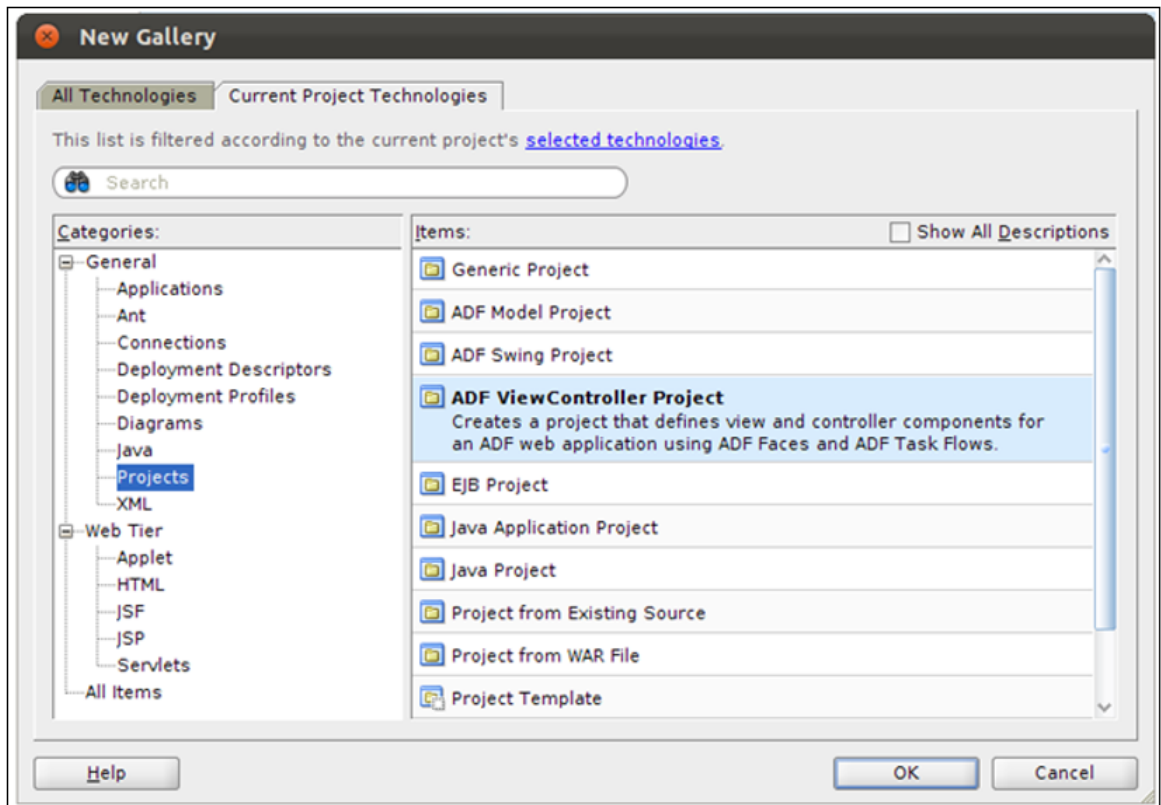
### Step 15 Create View Controller Project

You need to create a view controller project to contain the UI elements. This project will also hold the customizations to the application.

To create the view controller project, follow these steps:

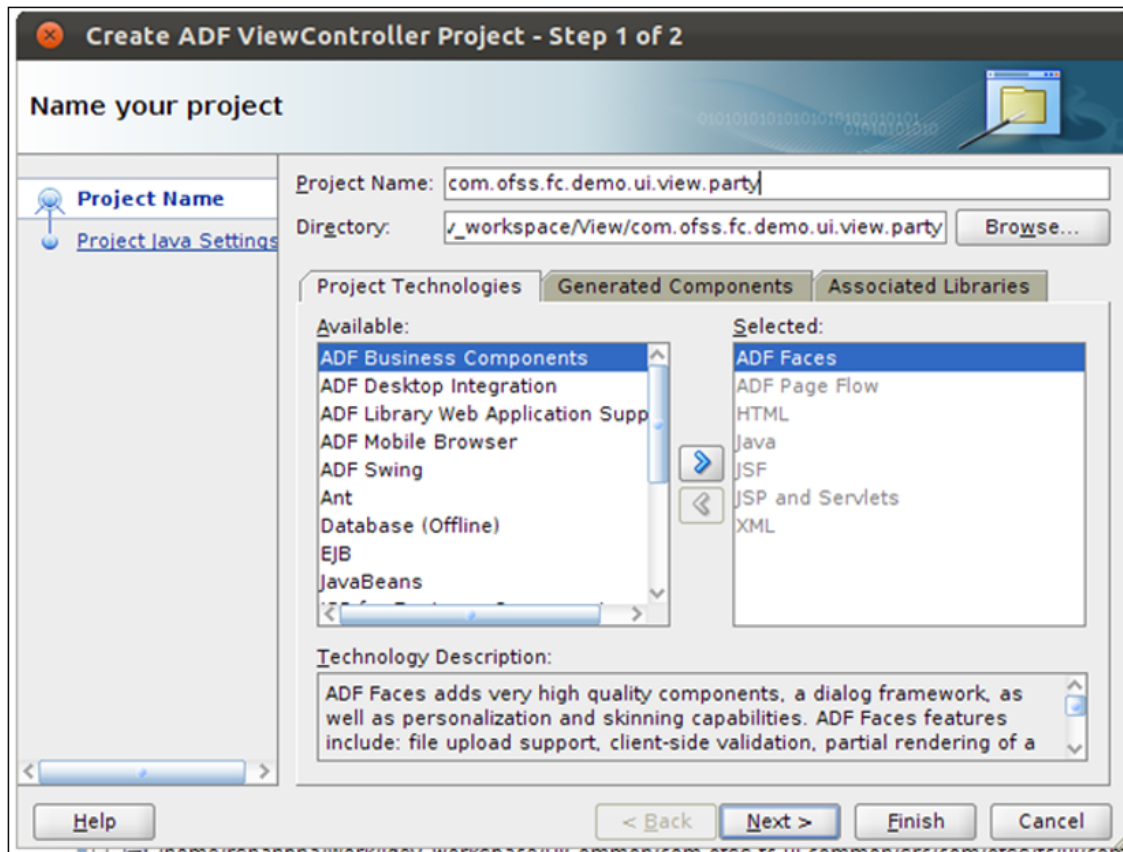
1. In the client application, create new project of the type ADF View Controller Project.
2. Give the project a title (com.ofss.fc.demo.ui.view.party) and set the defaults package to the same.

Figure 6–58 Create View Controller Project



3. Click on Finish to finish creating the project.

Figure 6–59 Name your Project



4. Right click on the project and go to Project Properties. In the Libraries and Classpath tab, add the following:
5. The Jar containing the screen to be customized (com.ofss.fc.ui.view.party.jar).
6. The Jar containing the domain objects and services for Contact Expiry (com.ofss.fc.demo.party.contactexpiry.jar) as created in host application project.
7. All the required dependent Jars for the above Jars.
8. The Jar containing the customization class (com.ofss.fc.demo.ui.OptionCC.jar).
9. In the Dependencies tab, browse for and add the previously created adf model project (com.ofss.fc.demo.ui.model.party).
10. In the ADF View tab, check the Enable Seeded Customizations option to enable this project for customizations.

Figure 6–60 Libraries and Classpath

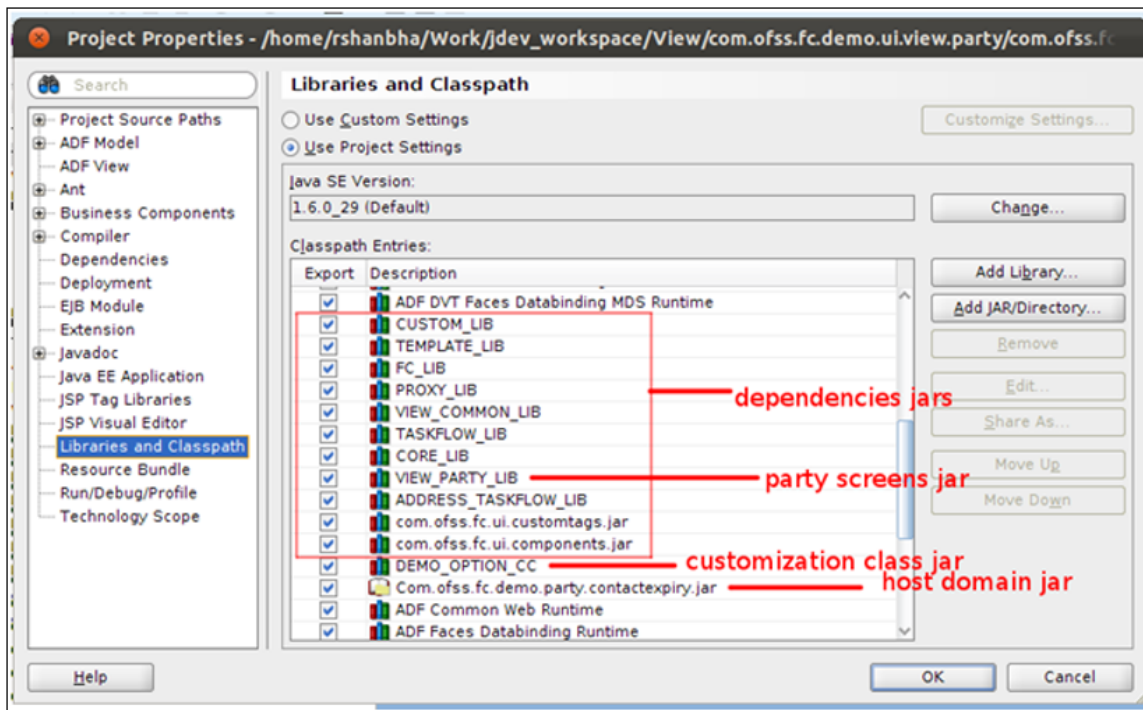
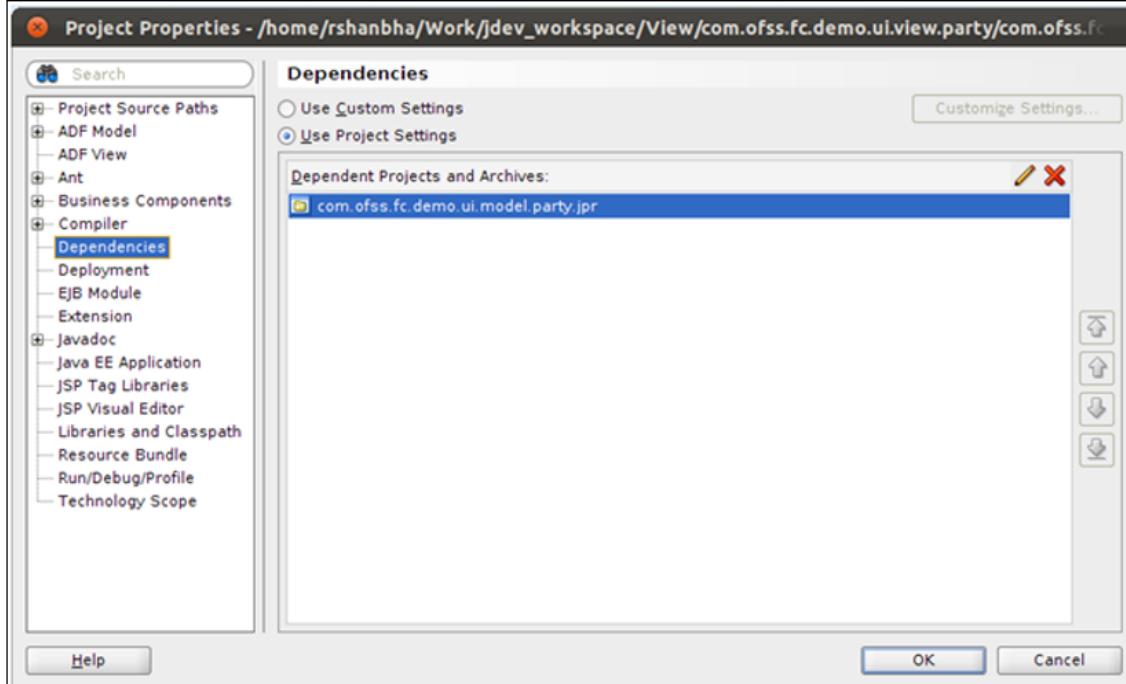


Figure 6–61 Dependencies



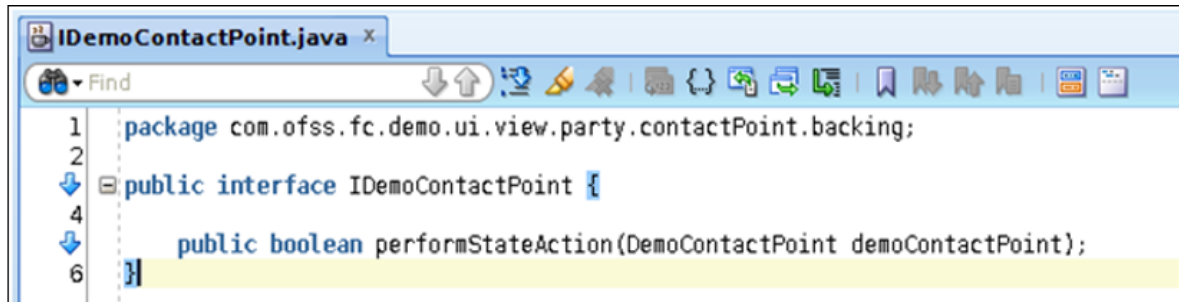
11. Save the changes by clicking OK and rebuild the project.

## Step 16 Create Maintenance State Action Interface



Create an interface containing the method definition for a maintenance action. This interface is implemented by the required maintenance state actions classes for the screen to be customized. The state action method will take the instance of the backing bean as a parameter.

**Figure 6–62 Create Maintenance State Action Interface**



```
1 package com.ofss.fc.demo.ui.view.party.contactPoint.backing;
2
3
4 public interface IDemoContactPoint {
5
6     public boolean performStateAction(DemoContactPoint demoContactPoint);
7 }
```

### Step 17 Create State Action Class

You need to create a class which will contain the business logic for the create transaction for this screen. This class should have following features:

- Implements the previously created state action interface.
- Creates the Contact Point DTO from the users input.
- Creates an instance of the Contact Point service proxy.
- Calls the add method of the service passing the DTO.

### Step 18 Create Update State Action Class

You will need to create a class which will contain the business logic for the update transaction for this screen. This class should have following features:

- Implements the previously created state action interface.
- Creates the Contact Point DTO from the users input.
- Creates an instance of the Contact Point service proxy.
- Calls the update method of the service passing the DTO.

Figure 6–63 Create Update State Action Class

```

22 public class DemoCreateContactPoint implements IDemoContactPoint {
23     private final Logger logger = MultiEntityLogger.getUniqueInstance().getLogger(DemoCreateContactPoint.class.getName());
24
25     public boolean performStateAction(DemoContactPoint demoContactPoint) {
26         // Create the DTO from the screen and call proxy.
27         boolean status = false;
28
29         SessionContext context = SessionContextFactory.getSessionContextFactory().getSessionContextInstance();
30         context.setServiceCode(Constants.SERVICE_CODE);
31         TransactionStatus transactionStatus = null;
32
33         ContactExpiryDTO contactExpDTO = demoContactPoint.createContactExpDTO();
34         IContactExpiryApplicationServiceProxy contactExpProxy = null;
35
36         try {
37             contactExpProxy = (IContactExpiryApplicationServiceProxy) ProxyFactory.getInstance().getProxy(
38                 DemoContactPoint.CONTACT_EXPIRY_PROXY);
39             if (logger.isLoggable(Level.FINE)) {
40                 logger.log(Level.FINE, "Calling addContactExpiry service");
41             }
42             transactionStatus = contactExpProxy.addContactExpiry(SessionContextFactory.getSessionContextFactory()
43                 .getSessionContextInstance(), contactExpDTO);
44             status = true;
45             if (transactionStatus != null && (transactionStatus.getErrorCode().equals("0"))) {
46                 MessageHandler.addMessage(transactionStatus);
47             }
48         } catch (FatalException e) {
49             MessageHandler.addMessage(e);
50             logger.log(Level.SEVERE, MultiEntityLogger.getUniqueInstance().formatMessage(
51                 "Exception while creating contact point", e));
52         } catch (ServiceException e) {
53             MessageHandler.addMessage(e);
54             logger.log(Level.SEVERE, MultiEntityLogger.getUniqueInstance().formatMessage(
55                 "Service exception while creating contact point", e));
56         } catch (Throwable e) {
57             MessageHandler.addErrorMessage("Internal error occurred. Please contact system administrator");
58         }
59         return status;

```

Figure 6–64 Create Update State Action Class

```

DemoUpdateContactPoint.java
Find
public class DemoUpdateContactPoint implements IDemoContactPoint {
20
21     private final Logger logger = MultiEntityLogger.getUniqueInstance().getLogger(DemoUpdateContactPoint.class.getName());
22
23     public boolean performStateAction(DemoContactPoint demoContactPoint) {
24         // Create the DTO from the screen and call proxy.
25         boolean status = false;
26
27         SessionContext context = SessionContextFactory.getSessionContextFactory().getSessionContextInstance();
28         context.setServiceCode(Constants.SERVICE_CODE);
29         TransactionStatus transactionStatus = null;
30
31         ContactExpiryDTO contactExpDTO = demoContactPoint.createContactExpDTO();
32         IContactExpiryApplicationServiceProxy contactExpProxy = null;
33
34         try {
35             contactExpProxy = (IContactExpiryApplicationServiceProxy) ProxyFactory.getInstance().getProxy(
36                 DemoContactPoint.CONTACT_EXPIRY_PROXY);
37             if (logger.isLoggable(Level.FINE)) {
38                 logger.log(Level.FINE, "Calling addContactExpiry service");
39             }
40             transactionStatus = contactExpProxy.updateContactExpiry(SessionContextFactory.getSessionContextFactory().
41                 getSessionContextInstance(), contactExpDTO);
42             status = true;
43             if (transactionStatus != null && (transactionStatus.getErrorCode().equals("0"))) {
44                 MessageHandler.addMessage(transactionStatus);
45             }
46         } catch (FatalException e) {
47             MessageHandler.addMessage(e);
48             logger.log(Level.SEVERE, MultiEntityLogger.getUniqueInstance().formatMessage(
49                 "Exception while updating contact point", e));
50         } catch (ServiceException e) {
51             MessageHandler.addMessage(e);
52             logger.log(Level.SEVERE, MultiEntityLogger.getUniqueInstance().formatMessage(
53                 "Service exception while updating contact point", e));
54         } catch (Throwable e) {
55             MessageHandler.addErrorMessage("Internal error occured. Please contact system administrator");
56         }
57         return status;
58     }
}

```

### Step 19 Create Backing Bean

You need to create a backing bean class for the screen to be customized. This class should have the following features:

- Should implement the interface ICoreMaintenance.
- Private members for the to be added UI Components in customization and public accessors for the same.
- Private member for the backing bean of the original backing bean of the screen (ContactPoint) which is initialized in the constructor of this class.
- Private member for the parent UI Component of the newly added UI components and public accessors which returns the corresponding component of the backing bean.
- Private member for the newly added view object (ContactExpiryVO) and the current view objects present on the screen.

Figure 6–65 DemoContactPoint.java displays the View Objects

```

45 public class DemoContactPoint implements ICoreMaintenance {
46
47     private static final String VO_CONTACT_EXP = "ContactExpiryVOIterator";
48     private static final String EXPIRY_DATE = "ExpiryDate";
49
50     protected static final String CONTACT_EXPIRY_PROXY = "ContactExpiryApplicationServiceProxy";
51
52     private UIGroup formData;
53     private ContactPoint contactPoint;
54     private ViewObject contactPointVO = IteratorHandler.getViewObject(Constants.PAGE_DEF, Constants.VO_CONTACT_POINT);
55
56     ContactPointBusinessRules contactPointBR = new ContactPointBusinessRules();
57     private RichPanellLabelAndMessage plan18;
58     private DateComponent expiryDate;
59     private ViewObject contactExpVO = IteratorHandler.getViewObject(Constants.PAGE_DEF, VO_CONTACT_EXP);
60
61     private transient Logger logger = MultiEntityLogger.getUniqueInstance().getLogger(DemoContactPoint.class.getName());
62
63
64     public DemoContactPoint() {
65         super();
66         contactPoint = (ContactPoint)ELHandler.get("#{ContactPoint}");
67     }
68
69     public void setFormData(UIGroup formData) {
70         this.formData = formData;
71     }
72
73     public UIGroup getFormData() {
74         this.formData = contactPoint.getFormData();
75         return formData;
76     }

```

- clear() method which handles the user action Clear.
- save() method which handles the maintenance state actions Create and Update.
- Depending on the current state action, the save() method should instantiate either DemoCreateContactPoint or DemoUpdateContactPoint and perform the corresponding state action methods.

Figure 6–66 DemoCreateContactPoint / DemoUpdateContactPoint

```

public boolean save() {
    boolean status = false;
    boolean flag = contactPoint.validateAllInputs();
    if (flag) {
        IDemoContactPoint demoContactPointAction = null;
        if (MaintenanceHelper.getCurrentState().equals(MaintenanceHelper.CREATE)) {
            demoContactPointAction = new DemoCreateContactPoint();
        } else if (MaintenanceHelper.getCurrentState().equals(MaintenanceHelper.UPDATE)) {
            demoContactPointAction = new DemoUpdateContactPoint();
        }
        status = demoContactPointAction.performStateAction(this);
    }
    return status;
}

public boolean clear() {
    if (contactPoint.clear()) {
        contactExpV0.clearCache();

        this.getExpiryDate().reset();
        this.getExpiryDate().setReadOnly(true);

        initializeContactExpV0();

        return true;
    }
    return false;
}

```

- A public method to create the Contact Expiry DTO from the user's input on the screen.

Figure 6–67 Create Contact Expiry DTO

```

public ContactExpiryDTO createContactExpDTO() {
    Date expiryDate = null;
    if (contactExpV0.getCurrentRow().getAttribute(EXPIRY_DATE) != null) {
        expiryDate = new Date(((oracle.jbo.domain.Date)contactExpV0.getCurrentRow().getAttribute(EXPIRY_DATE)).date);
    }

    ContactPointDTO contactPointDTO = contactPoint.createContactPointDTO();

    ContactExpiryDTO contactExpDTO = new ContactExpiryDTO();
    contactExpDTO.setContactPointDTO(contactPointDTO);
    contactExpDTO.setExpiryDate(expiryDate);

    return contactExpDTO;
}

```

- A value change event handler for the Expiry Date UI Component.

Figure 6–68 Value Change Event Handler for the Expiry Date UI Component

```

public void onExpiryDateChange(ValueChangeEvent valueChangeEvent) {
    if (logger.isLoggable(Level.FINE)) {
        logger.log(Level.FINE,
            MultiEntityLogger.getUniqueInstance().formatMessage("Entering onExpiryDateChange method."));
    }
    Date processdate =
        new com.ofss.fc.datatype.Date(((oracle.jbo.domain.Date)ELHandler.get("#{pageFlowScope.defaultValues.posti
if (valueChangeEvent.getNewValue() != null) {
    Date expDate =
        new Date(((oracle.jbo.domain.Date)valueChangeEvent.getNewValue()).dateValue());
    //TOOO: fix the process date error
    if (!expDate.isBefore(processdate)) {
        MessageHandler.addErrorMessage(getExpiryDate().getClientId(),
            "Expiry date should not be less than the current date",
            null);
        contactExpV0.getCurrentRow().setAttribute(EXPIRY_DATE,
            null);
        this.getExpiryDate().reset();
        AdfFacesContext.getCurrentInstance().addPartialTarget(expiryDate);
    }
} else if (valueChangeEvent.getNewValue() == null) {
    MessageHandler.addErrorMessage(getExpiryDate().getClientId(),
        "Select Expiry Date", null);
}
}

```

- Value change event handlers for the existing UI Components on change of which the screen data is to be fetched.

Figure 6–69 Value Change Event Handlers for Existing UI Components

```

public void onContactPreferenceChange(ValueChangeEvent valueChangeEvent) {
    if (MaintenanceHelper.getCurrentState().equals(MaintenanceHelper.READ)) {
        clearContactExpiryDetails();
        initializeContactExpVO();
        if (contactPointVO.getCurrentRow().getAttribute(Constants.PARTYID) != null
            && contactPointVO.getCurrentRow().getAttribute(Constants.CONTACT_POINT_TYPE) != null) {
            contactPointVO.getCurrentRow().setAttribute(Constants.CONTACT_PREF_TYPE,
                valueChangeEvent.getNewValue().toString());

            ContactExpiryDTO contactExpDTO = fetchContactExp();
            if(contactExpDTO != null) {
                setContactExpDetails(contactExpDTO);
            }
        }
    }

    contactPoint.onContactPreferenceChange(valueChangeEvent);
}

public void onContactPointTypeChange(ValueChangeEvent valueChangeEvent) {
    if (MaintenanceHelper.getCurrentState().equals(MaintenanceHelper.READ)
        || MaintenanceHelper.getCurrentState().equals(MaintenanceHelper.CREATE)) {
        clearContactExpiryDetails();
        if (MaintenanceHelper.getCurrentState().equals(MaintenanceHelper.READ)) {
            initializeContactExpVO();
        }
    }

    contactPoint.onContactPointTypeChange(valueChangeEvent);
}

```

- Method containing the business logic to fetch screen data using Contact Expiry proxy service.



Figure 6–70 Method to fetch Screen Data using Contact Expiry Proxy Service

```

private ContactExpiryDTO fetchContactExp() {
    ContactPointType cpType = null;
    if (contactPointV0.getCurrentRow().getAttribute(Constants.CONTACT_POINT_TYPE) != null) {
        cpType = (ContactPointType)EnumerationHelper.getInstance().fromValue(ContactPointType.class,
            (String)contactPointV0.getCurrentRow().getAttribute(Constants.CONTACT_POINT_TYPE));
    }
    ContactPreferenceType contactPrefType = null;
    if (contactPointV0.getCurrentRow().getAttribute(Constants.CONTACT_PREF_TYPE) != null) {
        contactPrefType = (ContactPreferenceType)EnumerationHelper.getInstance().fromValue(ContactPreferenceType.class,
            (String)contactPointV0.getCurrentRow().getAttribute(Constants.CONTACT_PREF_TYPE));
    }
    String partyId = (String)contactPointV0.getCurrentRow().getAttribute(Constants.PARTYID);

    ContactExpiryDTO contactExpDTO = new ContactExpiryDTO();

    ContactPointDTO contactPointDTO = new ContactPointDTO();
    contactPointDTO.setContactPoint(cpType);
    contactPointDTO.setPreferenceType(contactPrefType);
    contactPointDTO.setPartyId(partyId);

    contactExpDTO.setContactPointDTO(contactPointDTO);

    SessionContext context = SessionContextFactory.getSessionContextFactory().getSessionContextInstance();
    context.setServiceCode(Constants.SERVICE_CODE);

    IContactExpiryApplicationServiceProxy contactExpProxy = null;
    ContactExpiryInquiryResponse response = null;

    try {
        contactExpProxy = (IContactExpiryApplicationServiceProxy) ProxyFactory.getInstance().getProxy(CONTACT_EXPIRY_PROXY);
        if (logger.isLoggable(Level.FINE)) {
            logger.log(Level.FINE, "Calling fetchContactExp service");
        }

        response = contactExpProxy.fetchContactExpiry(SessionContextFactory.getSessionContextFactory().getSessionContextInstance(), contactExpDTO);

        if (response != null &&
            (response.getStatus().getErrorCode().equals("0"))) {
            contactExpDTO = response.getContactExpiryDTO();
        }
    } catch (FatalException e) {

```

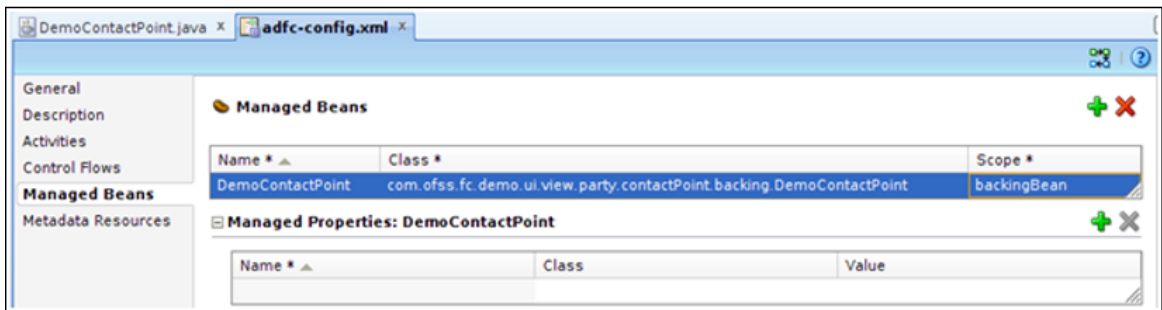
## Step 20 Create Managed Bean

You need to register the DemoContactPoint backing bean as a managed bean with a backing bean scope.

1. Open the project's adfc-config.xml which is present in the WEB-INF folder.
2. In the Managed Beans tab, add the binding bean class as a managed bean with backing bean scope as follows:



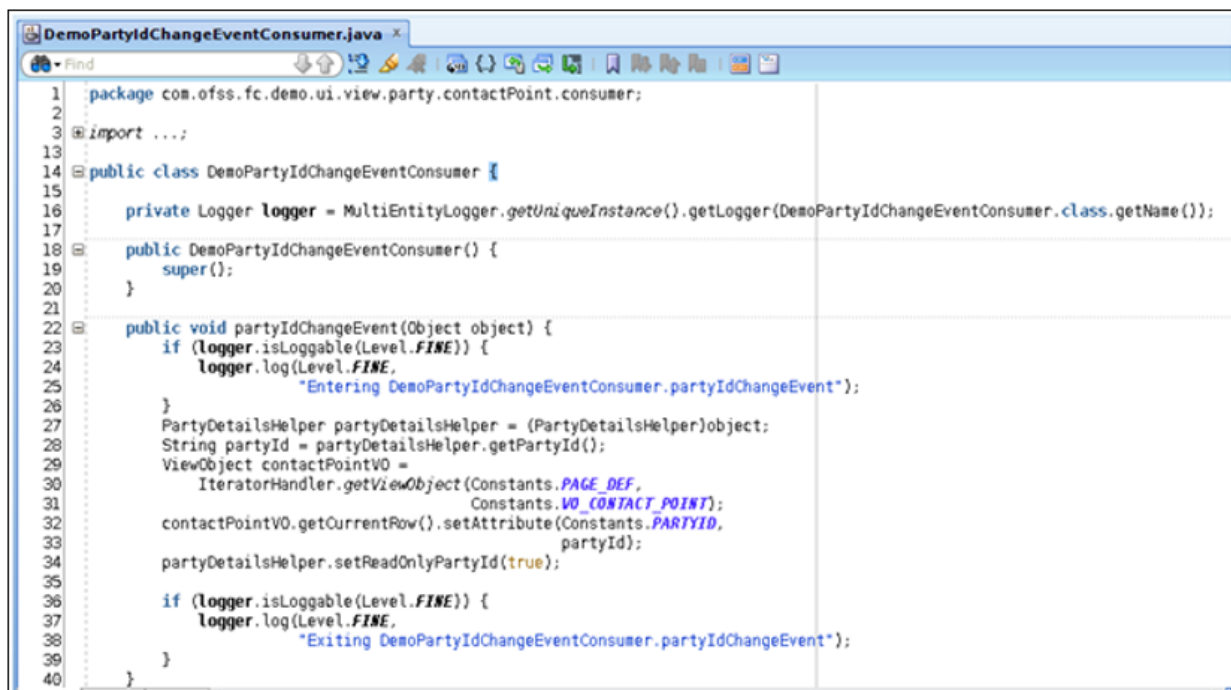
Figure 6–71 Create Managed Bean



### Step 21 Create Event Consumer Class

You need to create an event consumer class to consume the Party Id Change event. When the user inputs a party id on the screen, the business logic in this event consumer class will be executed automatically.

Figure 6–72 Create Event Customer Class



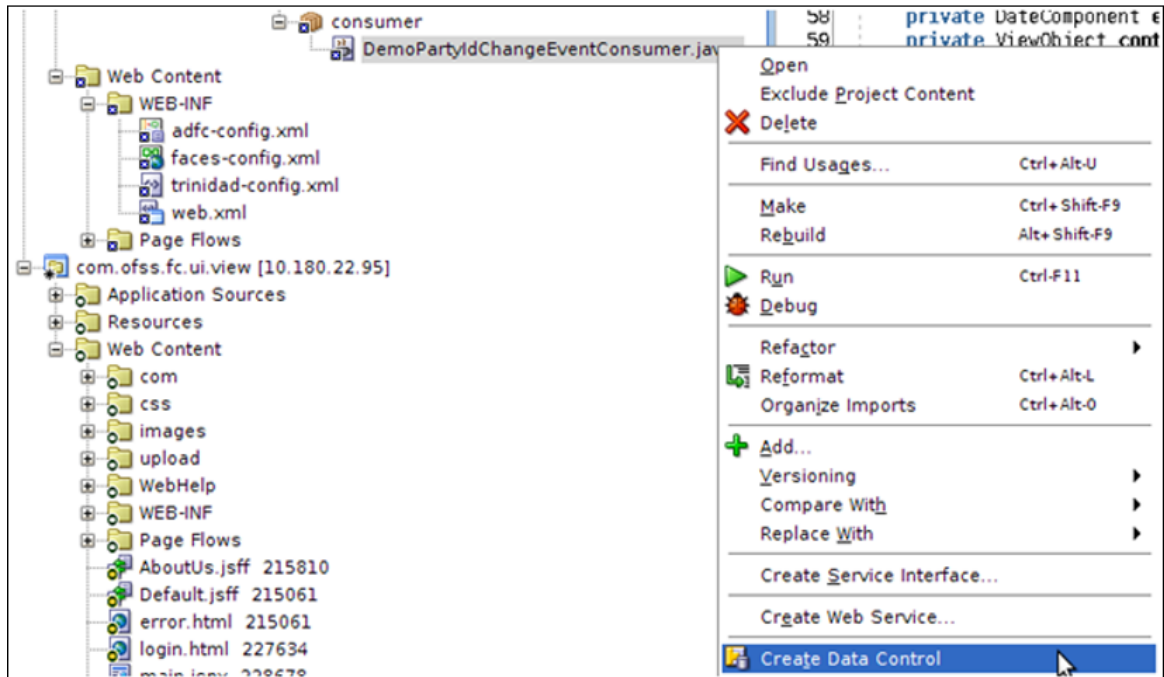
### Step 22 Create Data Control

For the event consumer class's method to be exposed as an event handler, you need to create a data control for this class.

1. In the Application Navigator, right click on the event consumer Java file and create data control.

On creation of data control, an XML file is generated for the class and a DataControls.dcx file is generated containing the information about the data controls present in the project. You will be able to see the event consumer data control in the Data Controls tab.

Figure 6–73 Create Data Control



2. You should now restart JDeveloper in the Customization Developer Role to edit the customizations. Ensure that the appropriate Customization Context is selected.

### Step 23 Add UI Components to Screen

Browse and locate the JSFF for the screen to be customized (com.ofss.fc.ui.view.party.contactPoint.contactPoint.jsff) inside the JAR (com.ofss.fc.ui.view.party.jar). Open the JSFF and do the required changes as follows:

1. Drag and drop the Panel Label & Message and Date UI components at the required position on the screen.
2. For each component, set the required attributes using the Property Inspector panel of JDeveloper.
3. Modify the containing Panel's width and number of columns attributes as required.
4. For each component, add the binding to the DemoContactPoint backing bean's corresponding members.
5. Add the value change event binding for the Expiry Date UI component to the backing bean's corresponding method.
6. Change the value change event binding for the existing UI component on change of which the screen data is fetched.
7. Change the backing bean attribute of the screen to the previously created DemoContactPoint backing bean.
8. Save the changes. You will notice that JDeveloper has created a customization XML in the ADF Library Customizations folder to save the differences between the base JSFF and the customized JSFF. The generated contactPoint.jsff.xml should look similar to as shown below.

Figure 6–74 Generated contactPoint.jsff.xml

```

1 <nds:customization version="11.1.1.61.92" xmlns:nds="http://xmlns.oracle.com/nds">
2 <nds:insert parent="formData" after="srcAllowedpurpose" xmlns:af="http://xmlns.oracle.com/adf/faces/rich" xmlns:fc="/com/ofss/fc/ui/components">
3 <af:panelLabelAndMessage xmlns:af="http://xmlns.oracle.com/adf/faces/rich" label="Expiry Date" id="plae18" binding="#{DemoContactPoint.plae18}">
4 <fc:date xmlns:fc="/com/ofss/fc/ui/components" label="Expiry Date" id="expiryDate" binding="#{DemoContactPoint.expiryDate}"
5 value="#(bindings.ExpiryDate.inputValue)" autoSubmit="true" readOnly="true" postValueChange="#{DemoContactPoint.onExpiryDateChange}"/>
6 </af:panelLabelAndMessage>
7 </nds:insert>
8 <nds:modify element="pt1(xmlns(f=http://java.sun.com/jsf/core))f:attribute[name='BackingBeanClass']">
9 <nds:attribute name="value" value="com.ofss.fc.demo.ui.view.party.contactPoint.backing.DemoContactPoint"/>
10 </nds:modify>
11 <nds:modify element="pf13">
12 <nds:attribute name="maxColumns" value="2"/>
13 <nds:attribute name="fieldwidth" value="60"/>
14 <nds:attribute name="labelwidth" value="40"/>
15 </nds:modify>
16 <nds:modify element="socContactPointType">
17 <nds:attribute name="valueChangeListener" value="#{DemoContactPoint.onContactPointTypeChange}"/>
18 </nds:modify>
19 <nds:modify element="socContactPref">
20 <nds:attribute name="valueChangeListener" value="#{DemoContactPoint.onContactPreferenceChange}"/>
21 </nds:modify>
22 </nds:customization>
23

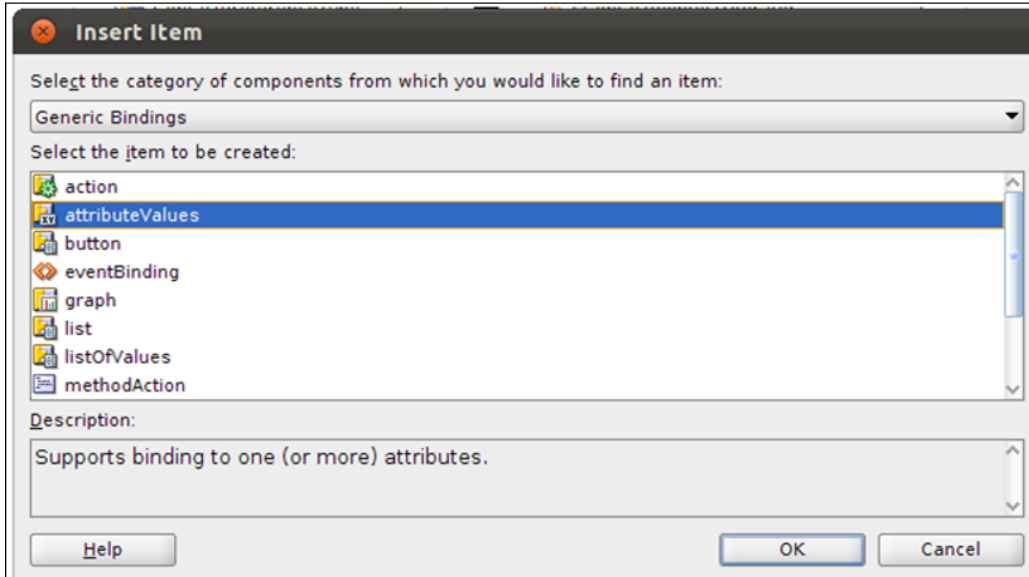
```

### Step 24 Add View Object Binding to Page Definition

You need to add the view object binding for the previously created ContactExpiryVO view object to the page definition of the screen to be customized.

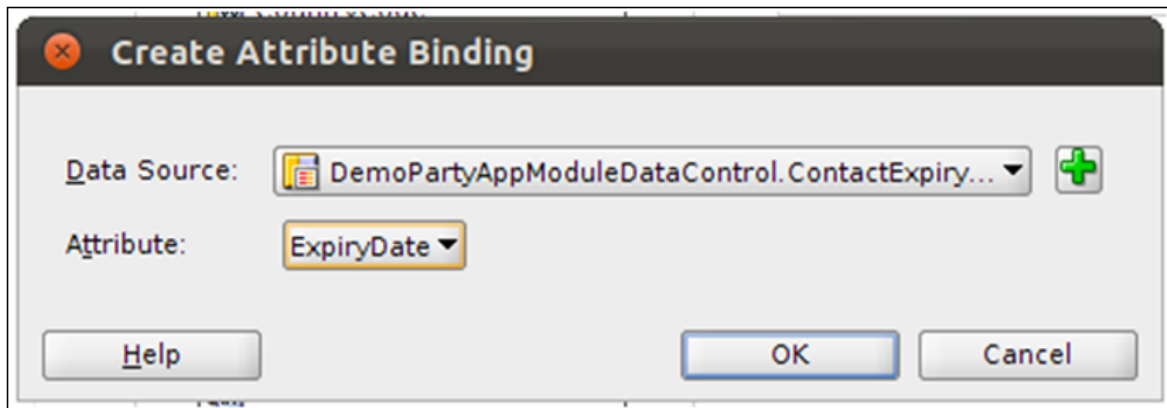
1. Browse and locate the page definition for the screen to be customized (com.ofss.fc.ui.view.party.contactPoint.pageDef.ContactPointPageDef.xml) and open it.
2. Add an attributeValues binding as shown below.

Figure 6–75 Add an attributeValues binding



3. For Data Source option, locate the previously created ContactExpiryVO view object present in the DemoPartyAppModule.
4. For Attribute option, choose the ExpiryDate attribute present in the view object.

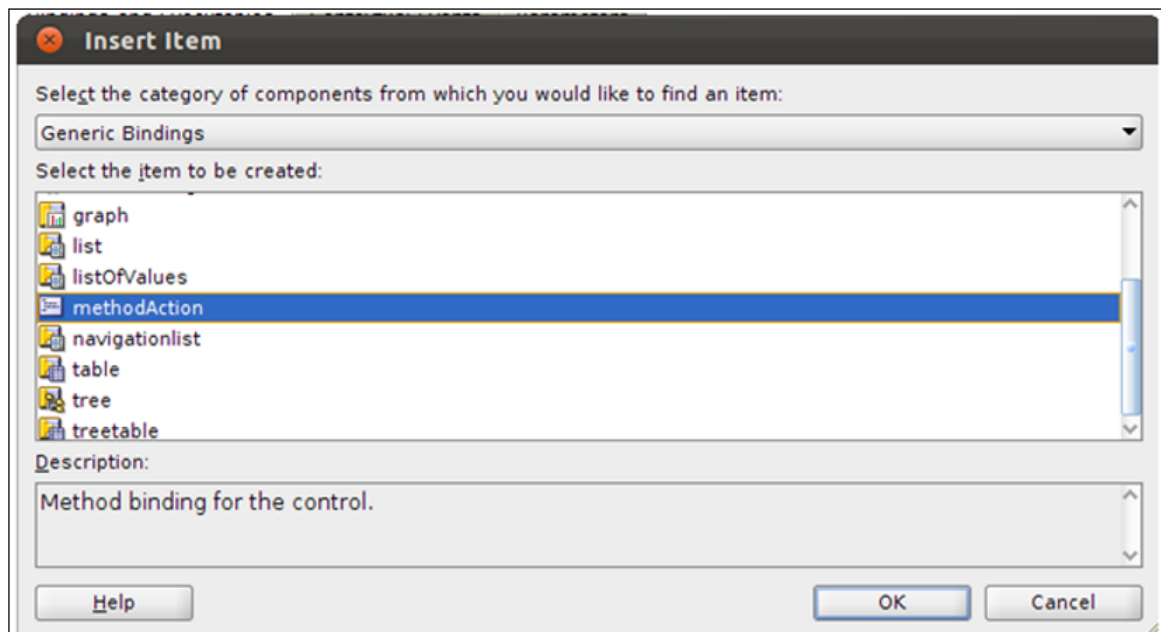
Figure 6–76 Create Attribute Binding



### Step 25 Add Method Action Binding to Page Definition

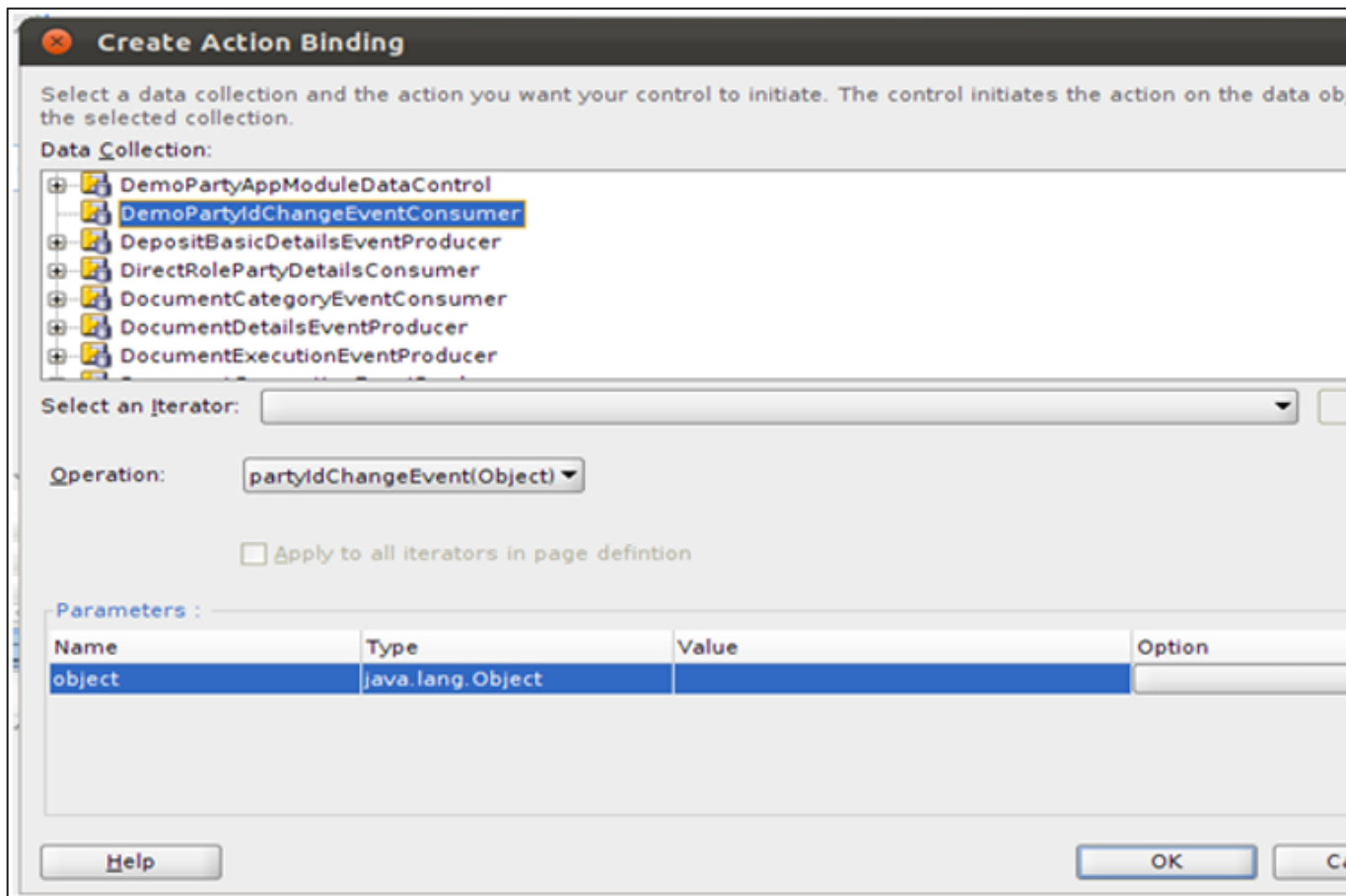
You need to add the method action binding for the previously created `DemoPartyIdChangeEventConsumer` event consumer class to the page definition of the screen to be customized.

1. Add a `methodAction` binding as shown below.

Figure 6–77 Add a `methodAction` binding

2. For the Data Collection option, locate the previously created `DemoPartyIdChangeEventConsumer` data control.

Figure 6–78 Create Action Binding

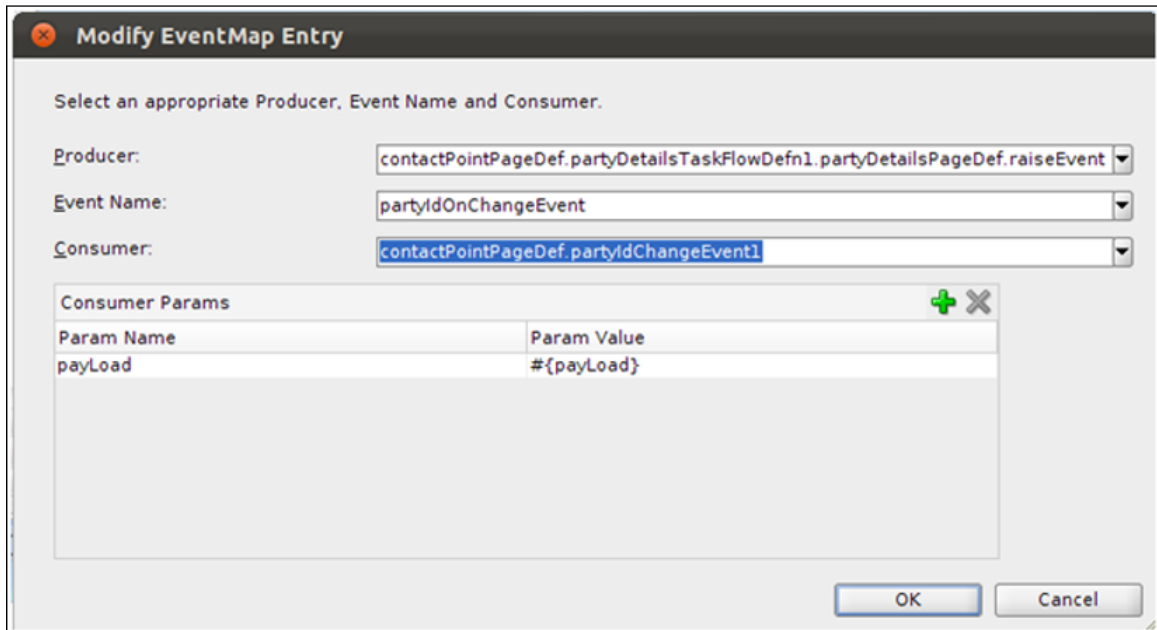


### Step 26 Edit Event Map of Page Definition

You need to map the Event Producer for the party id change event to the previously created Event Consumer.

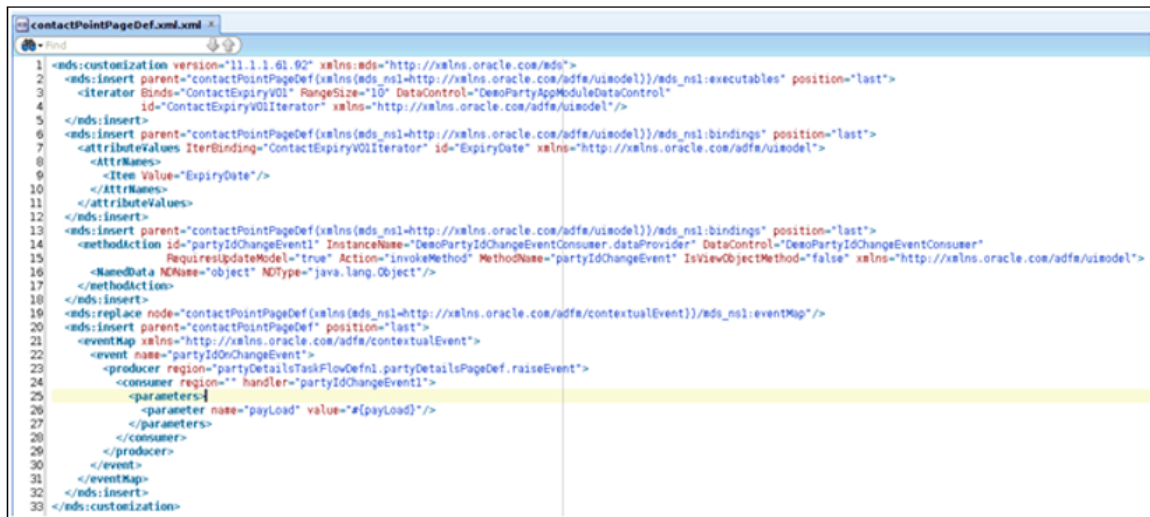
1. In the Structure panel of JDeveloper, right click on the page definition and select Edit Event Map.
2. In the Event Map Editor dialog that opens, edit the mapping for the party id change event. Select the previously created Event Consumer's method.

Figure 6–79 Select the Event Consumer Method



3. Save the changes. You will notice that JDeveloper has created a customization XML in the ADF Library Customizations folder to save the differences between the base JSFF and the customized JSFF. The generated contactPoint.jsff.xml should look similar to as shown below.

Figure 6–80 Generated contactPoint.jsff.xml



### Step 27 Deploy Customization Project

After finishing the customization changes, exit the Customization Developer Role and start JDeveloper in Default Role. Deploy the view controller project as an ADF Library Jar (com.ofss.fc.demo.ui.view.party.jar).

Go to Project Properties of the main application project and in the Libraries and Classpath, add the following:

1. View controller project Jar (com.ofss.fc.demo.ui.view.party.jar)
2. Host domain Jar (com.ofss.fc.demo.party.contactexpiry.jar)
3. Customization Class Jar (com.ofss.fc.demo.ui.OptionCC.jar)
4. All dependency libraries and Jars for the project.
5. Start the application and navigate to Party → Contact Information → Contact Point screen. Input a party id on the screen and perform the read, create and update actions on Contact Point. You need to input data and fetch value for the newly added Expiry Date field.

**Figure 6–81 PI041 - Contact Point Screen**

The screenshot displays the 'Contact Point' screen for PI041. The interface includes a top navigation bar with 'Read', 'Create', and 'Update' buttons. The 'Update' button is highlighted with a red box. The main content area is divided into several sections:

- Party Details:** Displays Party ID (00005295), Home Branch (082991-U Bank Operations BR), Company Name (Daniel trustee), Party Class (FOREIGN PUBLIC BODY), Party Type (LEG), Date of Incorporation, Roles (Customer, Trustee), and Onboarding Date (15-Jan-2016). Two 'NO IMAGE AVAILABLE' placeholders are present.
- Address Details:** Contains Contact Point Type (Mobile), Seasonal Start Date, Allowed Purposes (Communication, Alert), Preferred Contact (Preferred Contact, Marketing Consent), and Marketing Consent Start/End Dates.
- Contact Point Details:** Shows Contact Preference Type (Home) and a highlighted 'Personal Exp Date' field with an 'Expiry Date' of '31-Aug-2012'.
- Telephone Details:** Includes Country Code, Number (32577789), Service Provider, Area Code, Extension, and VOIP Code.
- Timing Preferences:** Features DND (Do Not Disturb) options for Weekdays and Weekends, with fields for 'From' and 'To' times.

### 6.7.3 Override the product managedBean

Screen customizations could be used to handle a product code which does not serve the necessary functionality and needs to be re-written.





# 7 ADF Taskflow Generation for Customization or Localization

## 7.1 Introduction

OBP product uses Oracle FUSION technology. The Presentation layer is built using Oracle ADF (Application Development Framework). It is recommended that the Presentation layer should not have any business logic; business policy checks should be in the host application service. Business rules in presentation should be limited to making UI components visible or read-only etc to make the UI convenient to use.

In an effort to improve productivity, maintainability, and extendibility, the ADF Taskflow Generator has been developed.

The key features of the ADF Taskflow Generator are:

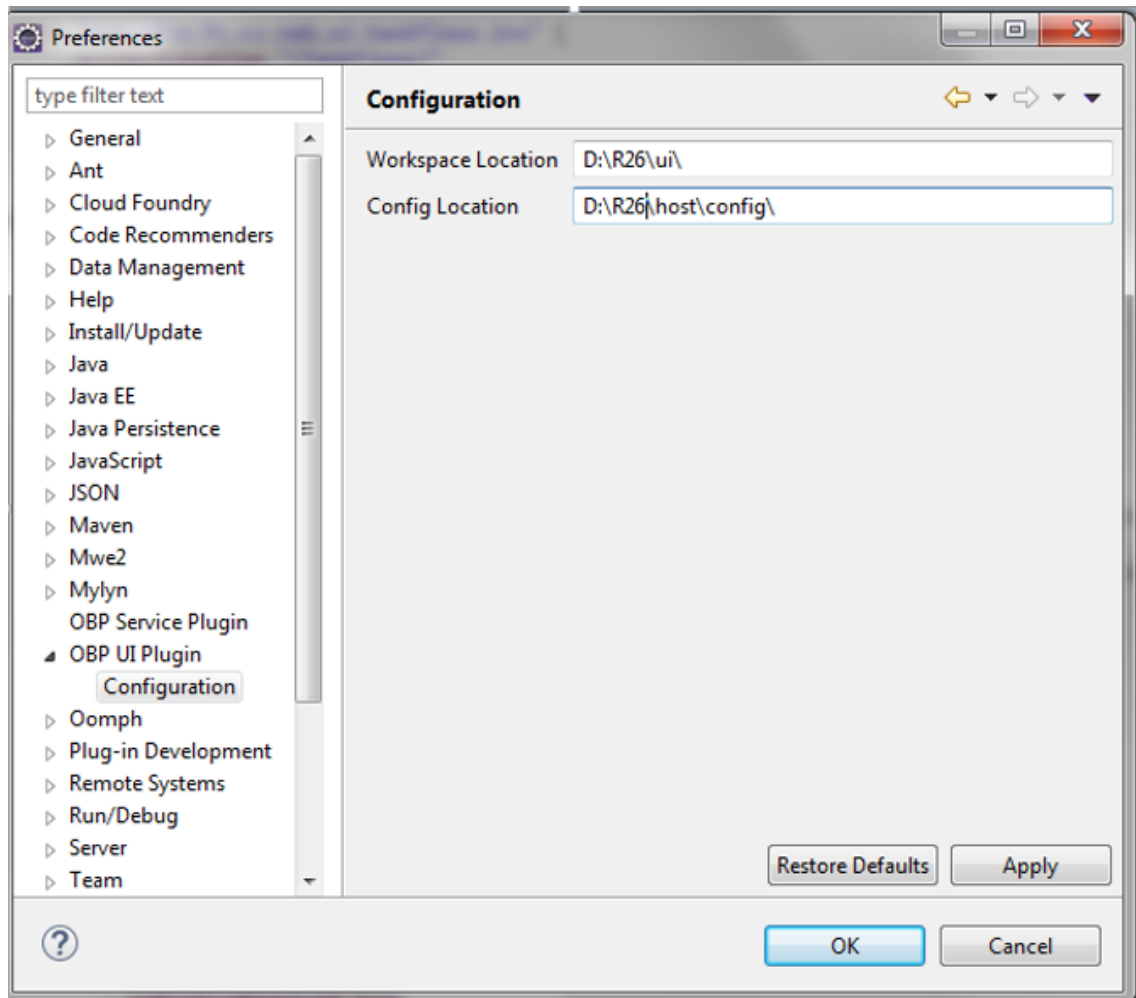
- Complies to new Integrable Taskflow framework.
  - Common Helper for taskflows “IntegrableTaskFlowHelper.java”. It acts as a mediator to facilitate encapsulation.
  - Widget specific Handler class to handle helper and backing bean events.
  - Widget specific “TaskFlowContext” for specifying parameters or behaviors expected by embedded taskflow.
  - Widget specific “DataObject” for passing DTO’s to embedded taskflow.
  - Widget specific “Event Producer” facilitating standard events.
  - Extensions for localization or customization.
- The Widget code can be re-generated without any impact to the manually written code.
- Input of Widget Configurations via Xtext editor.

## 7.2 Installing the Generator

### Installation Steps

1. Install eclipse on your machine and add the following jars in the plugins folder:
  - com.ofss.fc.ui.generator.widgetmodel.jar
  - com.ofss.fc.ui.generator.widgetmodel.ui.jar
2. Configure “OBP UI Plugin -> Configuration” in Preferences. Refer below screenshot. The combination of the “Workspace Location” in Configuration and “projectLocation” in the .wconf is where the plug-in expects to generate the code. The “Config Location” is where the plug-in expects to generate the Java Property files.

Figure 7–1 Preferences screen



## 7.3 Using the Generator

### Instructions on how to Create Widget Configuration

1. Start eclipse and create new file with extension `.wconf` using menu File --> New --> File.
2. Input widget configuration. Refer to the sample `wconf` for consulting.
3. Use shortcut key CTRL+SPACE to be prompted for valid grammar. The `.wconf` grammar syntax check is rudimentary (e.g. `valueChangeHandlerRequired` though not applicable for a component type "OutputText" will not show as error until you build the project, in such cases fix the `.wconf` to remove the `valueChangeHandlerRequired` attribute or manual remove the generated attribute).

### Instructions on how to Generate Widget Code

1. Open the `.wconf` file in eclipse.
2. Right click on the editor (`.wconf` file) and select Generate Widgets. The widgets for the particular widget configuration will be generated.
3. Error logs can be checked at `C:\Users\\UI_Plugin_Logs`.

4. Certain manual step are required to complete the taskflow creation.
5. Refresh the model until the xml files for the two VO's are visible. Close and reopen the project if necessary.
6. RowImpl for the VO's needs to be generated using following steps:
  - a. Open the corresponding VO file in ADF.
  - b. Go to Overview.
  - c. Click on Edit java Option.
  - d. Select checkbox Generate view object class.
  - e. Select checkbox Include bind variable accessors.
  - f. Select checkbox Include custom java data source methods.
  - g. Select checkbox Generate view Row Class.
  - h. Select the subset Select InClass Accessors.
  - i. Click on Ok.
7. For the generation of corresponding vo entry in \*.jspx.
  - a. The project once opened in ADF, the model needs to be refreshed two to three times. ADF generates the entries automatically.
8. If the taskflow has a Event Consumer execute following to generate the corresponding Data control:
  - a. Right click on \*EventConsumer.java.
  - b. Select Create Data Control.
  - c. If the taskflow has a Event Producer, execute the following to generate the corresponding Data control.
  - d. Right click on \*EventProducer.java.
  - e. Select Create Data Control.
9. Rebuild the project after the above changes are made to make sure that all the changes are effectively reflected.

### Steps to Integrate the Generated Code

1. Code is generated directly in Target Package. Re-build & re-deploy the taskflows jar.
2. If the taskflow is a simple embeddable taskflow, include new Taskflow as region on existing screen.
3. If the taskflow uses the Maintenance/ Transaction template and has to be added as a screen, add the new TaskCode entries in following files:
  - menu\_elements\_en.csv"
  - taskflowDefn\_element.csv
  - \*-taskflow-definition located in com.ofss.fc.view.\* (only for taskflow as a screen)

### Manual Coding

1. Adding UI components manually.
2. Add component to JSFF, corresponding backing event method and delegate call to taskflow handler.

3. A JAPA utility will generate the required consulting hooks (Entries in Abstract Handler and EXT files).
4. Component's entries for VO, State VO and Resource bundle will need to be done manually.
5. Manual code should be added to Handler, Utils and Assembler methods.
6. Host calls will be generated in Utils class, wiring to specific event to be done manually.
7. UI specific business rules like enabling/disabling UI components can be added to a Business Rules class (Invoked via Handler).
8. If VO's or any collection objects are manually created, ensure they are added to the viewobjectMap or the collectionVOMap. If not they need to be cleared manually in the destroyView method in handler.
9. Add code to renderView method in handler to utilize the Taskflow Context / Data Object set.
10. Add additional assembler code as per requirement.
11. Add attributes to taskflow context object as per requirement.
12. DO NOT make manual changes in abstract classes / ext classes.

### Extensions for Localization/Customization

- Extensions for localization/customization will be generated in package <ext>.
- Extensions are in similar lines to extensions in the OBP Application Service.

### Mandatory Steps for Localization/Customization specific wconf files

Refer cz.wconf and lz.wconf

1. Create new/empty files like AppModule, DataBindings and DataControls if not already present before generating using the generator
2. Specify the client name (Example : nab) for cz specific taskflows.
3. Specify the locale name (au or us) for lz specific taskflows.

### Cz.wconf sample

```
project "com.ofss.fc.cz.nab.ui.taskflows.ins" {
  projectLocation "/TaskFlows/"
  rbConfigLocation "/resources/cz/nab/ins/"
  packageName "com.ofss.fc.cz.nab.ui.taskflows.ins"
  clientName "nab"
}
widget "CCIPanelTest" {
  widgetType ADF
  appModule "NABINSTaskflowAppModule"
  description "This is the Access CCI Panel TaskFlow since R2.4"
  isConsumer false
  isProducer false
  fullyQualifiedTaskFlowDTO java.lang.Object
  RemoteProcedureCall {
    methodName "fetchEligibilityPremiumDetails"
    rpcDetails {
      fullyQualifiedHostDTO
```

```

com.ofss.fc.cz.nab.app.ins.cci.dto.ConsumerCreditInsuranceDetailsD
TO
endPointProxy CCIApplicationServiceProxy
fullyQualifiedAPI
com.ofss.fc.cz.nab.app.ins.service.cci.service.client.proxy.ICCIAp
plicationServiceProxy.fetchEligibilityPremiumDetails
(SessionContext,LendingProductGroupLinkageDTO,ApplicantDataDTO
[],ProductDetailsDTO,ConsumerCreditInsuranceDetailsDTO)
responseClass com.ofss.fc.cz.nab.app.ins.cci.dto.CCIResponse
}
}
component accessPanelGroup
{
componentType PanelGroupLayout
isBindingRequired false
attributesForPanelGroupLayout {
layout Horizontal
}
children
{
component applicantDetails {
componentType OutputText
isBindingRequired true
label "Applicant Details"
accessibilityText "Applicant Data Grid Start"
}
component applicantDataGrid{
componentType InputTableWithLabel
isBindingRequired true
attributesForTable
{
tableSummary "Applicant Details"
width "100"
immediate true
filterVisible false
columnDetails
{name "colSelAppl" fullyQualifiedName "String"
columnType SELECTONECHOICE headerText "SelectApplicant"}
{name "colApplName" fullyQualifiedName "String"
columnType OUTPUTTEXT headerText "ApplicantName"}
{name "colApplAge" fullyQualifiedName "String"
columnType OUTPUTTEXT headerText "ApplicantAge"}
{name "colApplPrem" fullyQualifiedName "String"
columnType OUTPUTTEXT headerText "ApplicantAnnualPremium"}
}
isVisible true
isRendered true
isEditable false

```

```
accessibilityText "Applicant Details"
}
component applicantDetailSpacer {
componentType Spacer
isBindingRequired false
attributesForSpacer {
width "5"
height "5"
}
}
component okButton {
componentType CustomButton
isBindingRequired true
label "Proceed With Protection"
accessibilityText "Proceed With Protection"
clickHandlerRequired true
}
component cancelButton {
componentType CustomButton
isBindingRequired true
label "Proceed With Protection"
accessibilityText "Proceed With Protection"
clickHandlerRequired true
}
}
}
end
}
```

### Lz.wconf sample

```
project "com.ofss.fc.lz.us.ui.taskflows.party" {
projectLocation "/TaskFlows/"
rbConfigLocation "/resources/taskflows/party/"
packageName "com.ofss.fc.lz.us.ui.taskflows.party"
locale "us"
}
widget "PartySampleWidget" {
widgetType ADF
appModule "LzUsTaskFlowsPartyAppModule"
description "Party Service LZ specific Taskflow"
isConsumer false
isProducer false
fullyQualifiedTaskFlowDTO java.lang.Object
component accessPanelGroup
{
componentType PanelGroupLayout
isBindingRequired false
attributesForPanelGroupLayout {
```

```
layout Horizontal
}
children
{
component applicantDetails {
componentType OutputText
isBindingRequired true
label "Applicant Details"
accessibilityText "Applicant Data Grid Start"
}
component applicantDataGrid{
componentType InputTableWithLabel
isBindingRequired true
attributesForTable
{
tableSummary "Applicant Details"
width "100"
immediate true
filterVisible false
columnDetails
{name "colSelAppl" fullyQualifiedName "String"
columnType SELECTONECHOICE headerText "SelectApplicant"}
{name "colApplName" fullyQualifiedName "String"
columnType OUTPUTTEXT headerText "ApplicantName"}
{name "colApplAge" fullyQualifiedName "String"
columnType OUTPUTTEXT headerText "ApplicantAge"}
{name "colApplPrem" fullyQualifiedName "String"
columnType OUTPUTTEXT headerText "ApplicantAnnualPremium"}
}
isVisible true
isRendered true
isEditable false
accessibilityText "Applicant Details"
}
component applicantDetailSpacer {
componentType Spacer
isBindingRequired false
attributesForSpacer {
width "5"
height "5"
}
}
component okButton {
componentType CustomButton
isBindingRequired true
label "Proceed With Protection"
accessibilityText "Proceed With Protection"
clickHandlerRequired true
```

```
    }  
    component cancelButton {  
        componentType CustomButton  
        isBindingRequired true  
        label "Proceed With Protection"  
        accessibilityText "Proceed With Protection"  
        clickHandlerRequired true  
    }  
}  
}  
end  
}
```

### Do's and Don'ts

#### Do's

- When generating widget
- Widget configuration (.wconf file) should be checked into SVN.
- List all components required on screen in a single .wconf.
- Widget generation is a onetime activity. Include host calls and all required components in .wconf to minimize manual coding.
- Manual changes
- Refer [Manual coding](#).
- Interfacing with Container/Consumer Taskflow.
- Modify the <widget>TaskFlowContext to include parameters / behaviors to be exposed to the consumer taskflow by your integrable taskflow.
- Modify the DataObject to include DTO's to be passed to your integrable taskflow by the consumer taskflow.

#### Don'ts

- DO NOT make changes in any abstract class in generated code. "Warning : Changes done to this file will be lost on re-generation" is mentioned in these files.
- DO NOT introduce your own helper, use the common IntegrableTaskFlowHelper.
- DO NOT use page flow scope attributes in Handler for UI Component Rendered / Visible / Disabled attributes. Add attributes to State VO instead.
- Please make sure that if you are changing taskflow DTO then you are changing it throughout the taskflow and not in few files only.
- Please make sure that Handler and assemblers extend their respective abstract classes.



## 7.4 ADF TaskFlow Generation Usecases

### 7.4.1 Extensive UI components to be included on screen

If there is more than an adequate number of fields to be included on an already existing product screen its advisable to create a new taskflow for the same using the taskflow generater following the steps mentioned above.

Sample Wconfs is available here [com.ofss.fc.taskflows.wconf](http://com.ofss.fc.taskflows.wconf). Once the the taskflow is generated include the taskflow using the Extension hooks.



# 8 Human Task Screen Extension

## 8.1 Introduction

Human task screen uses internally ADF taskflows except the primary worklist actions. Human screen extensions work similar to ADF screen extensions, where the backing bean and controller classes of human task have the provision for extension, similar to the UI extension as explained in [Section 5.1 UI Extension Interface](#), [Section 5.2 Default UI Extension](#), [Section 5.3 UI Extension Executor](#), and [Section 5.4 Extension Configuration](#). From the perspective of customization, the recommendation is to do customization on the taskflows used internally. For CSS customization, see the below the details.

## 8.2 Custom CSS Skin

If ADF skin is already available and you just want to change the skin family reference in “trinidad-config.xml”, directly apply the [Section 8.2.2 Apply New Skin](#) . If you want to create a new ADF skin and merge with the existing skin families to “trinidad-skins.xml”, start from [Section 8.2.1 Create New ADF Skin](#) .

### 8.2.1 Create New ADF Skin

To create a new ADF skin and package it as library to refer to new skin families files, perform the following steps:

1. Create ADF skin project and skin families as given in [https://docs.oracle.com/cd/E16162\\_01/user.1112/e17456/adfsg\\_project.htm#ADFSG490](https://docs.oracle.com/cd/E16162_01/user.1112/e17456/adfsg_project.htm#ADFSG490):

*Figure 8–1 Sample trinidad-skins.xml*

```
<?xml version="1.0" encoding="windows-1252"?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
  <skin>
    <id>NEWOBPALTASKIN.desktop</id>
    <family>NEWOBPALTASKIN</family>
    <extends>alta-v1.desktop</extends>
    <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
    <style-sheet-name>adf/skins/obp_alta-extend/obp_alta_extend.css</style-sheet-name>
  </skin>
</skins>
```

2. Package the above project as given in [https://docs.oracle.com/cd/E16162\\_01/user.1112/e17456/adfsg\\_apply.htm#ADFSG404](https://docs.oracle.com/cd/E16162_01/user.1112/e17456/adfsg_apply.htm#ADFSG404).

After deployment, the package structure should be as shown in the following screenshot:

Figure 8–2 Package Structure

```

ADFLibraryJARRootDirectory
+---META-INF
|   |   MANIFEST.MF
|   |   oracle.adf.common.services.ResourceService.sva
|   |   trinidad-skins.xml
|   |
|   +---adf
|       |   \---skins
|           |       \---skin1
|               |       \---images
|                   |       \---af_column
|                       |       sort_des_selected.png
|                       |
|                       \---skins
|                           \---skin1
|                               skin1.css
|
+---resources
|   skinBundle.properties
|
\---WEB-INF
    faces-config.xml

```

3. Deploy or add the deployed jar to obp ui library references.
4. Reference implementation can be provided on request.

## 8.2.2 Apply New Skin

- **Option #1:** The skin id can be configured with the help of property 'default.humantask.skinfamily' in seed data available as part of FLX\_FW\_CONFIG\_ALL\_B.

Figure 8–3 Sample Data

```

INSERT
INTO flx_fw_config_all_b
(
  PROP_ID, CATEGORY_ID, PROP_VALUE, FACTORY_SHIPPED_FLAG, PROP_COMMENTS,
  SUMMARY_TEXT, CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE,
  OBJECT_STATUS_FLAG, OBJECT_VERSION_NUMBER
)
VALUES
(
  'default.humantask.skinfamily', 'UiConfig', 'NEWOBPALTASKIN', 'Y',
  NULL, 'NEWOBPALTASKIN UiConfig', 'ofssuser', to_timestamp('25-JUL-17 02.13.06.000000000 PM','DD-MON-RR HH.MI.SSXXF AM'),
  'ofssuser', to_timestamp('25-JUL-17 02.13.06.000000000 PM','DD-MON-RR HH.MI.SSXXF AM'), 'A', 1
);

```

- **Option #2:** Over and above the above the option 1, the extension can be implemented for

---

“postCustomBranding” operation of CommonTaskFormPagePhaseListenerImpl [extension class: com.ofss.fc.workflow.ui.common.ext.ICommonTaskFormPagePhaseListenerImplUIExt]

**Figure 8–4 Sample Implementation**

```
FacesContext fc = FacesContext.getCurrentInstance();
ELContext elc = fc.getELContext();
String skinId = "NEWOBPALTASKIN";
ExpressionFactory exprFact = fc.getApplication().getExpressionFactory();
ValueExpression ve = exprFact.createValueExpression(elc, SS_SKIN_FAMILY, Object.class);
ve.setValue(elc, skinId);
ELHandler.set(IS_CUSTOM_BRANDING, "false");
```



# 9 Receipt Printing

OBP has many transaction screens in different modules where it is desired to print the receipt with different details about the transaction. This functionality provides the print receipt button on the top right corner of the screen which gets enabled on the completion of the transaction and can be used for printing of receipt of the transaction details.

For example, if the customer is funding his term deposit account, the print receipt option will print the receipt with the details like Payin Amount, Deposit Term etc at the end of the transaction. The steps to configure this option in the OBP application are given in the following section.

## 9.1 Prerequisite

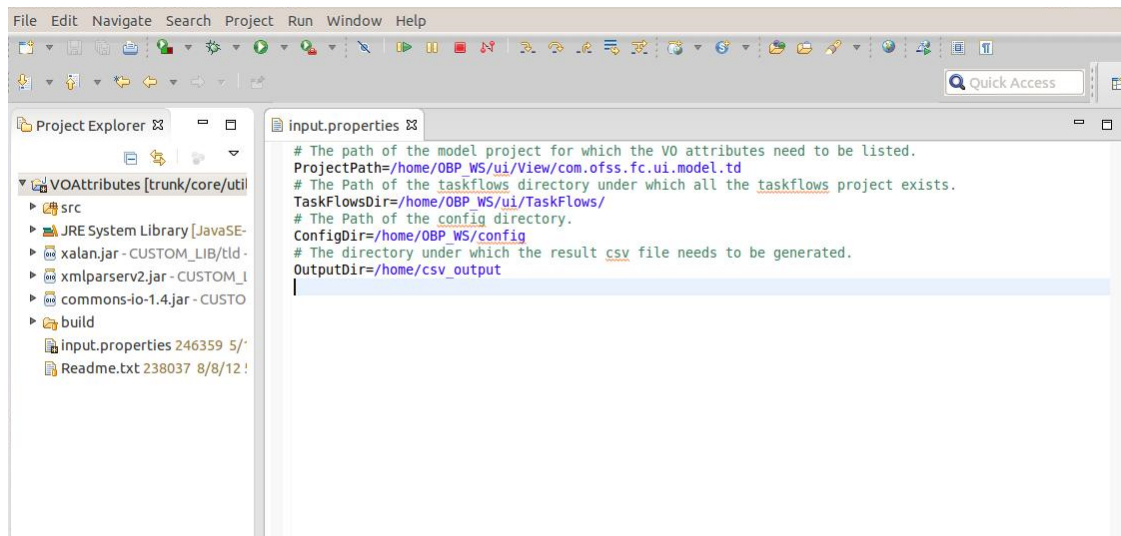
Following are the prerequisites for receipt printing.

### 9.1.1 Identify Node Element for Attributes in Print Receipt Template

The list of all the elements that are present in the particular task code screen and need to be displayed in the printed receipt can be identified with the help of the VO object utility. This utility helps in identifying all the node elements which are available on the screen and can be used in the print receipt template. This utility VObjectUtility can be used to generate the data required for the functionality to work.

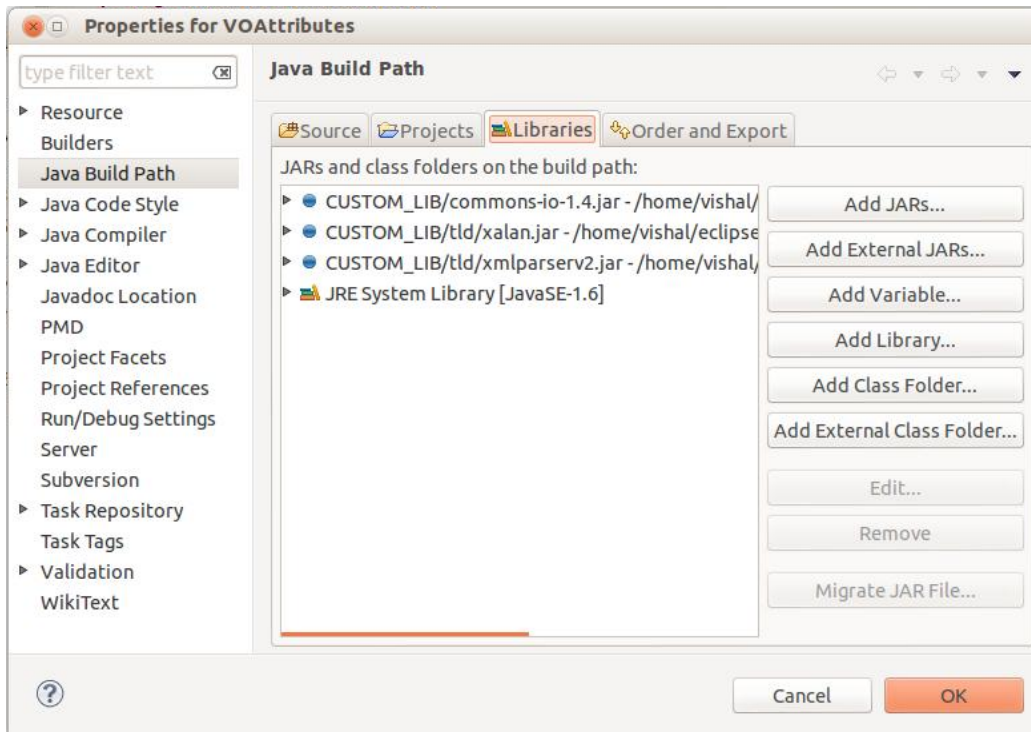
Once the utility is imported in the workspace, the input.properties file needs to be updated with the location of module's UI, location of task flow directory, location of config directory and the output directory where you want the output of the utility.

**Figure 9–1 Input Property Files**



In the build path of the utility, three libraries (commons-io, xalan and xmlparserv2) need to be added as they are required for execution of the utility.

Figure 9–2 Build Path of Utility



Then the main method of the VOAttributesFinder.java class in the utility is executed.



Figure 9–3 Utility Execution

```

package com.ofss.fc.voattribute;

import java.io.File;

public class VoAttributesFinder {

    * Represents the properties object to read the properties file.
    private Properties prop;
    private String outputFile;
    private StringBuilder voAttributes = new StringBuilder();
    private Map<String, String> taskCodeMap = new HashMap<String, String>();
    private Map<String, String> taskFlowVisitedMap =
    private Map<String, String> taskFlowVoAttributes =
    private static final String LINE_SEPARATOR =
    private static final String FILE_SEPARATOR =
    private static final String STARTING_STR = "<?";
    private static final String ENDING_STR = ">";

    public static void main(String[] args) throws IOException {

        new VoAttributesFinder().getVoAttributes();
    }

    public VoAttributesFinder() {

        init();
    }

    public void getVoAttributes() throws IOException {

        getAllVoAttributes();
    }

    private void getAllVoAttributes() throws IOException {

        String projectPath = prop.getProperty("ProjectPath");
        voAttributes.append("Task Code").append(",").append("View Object").append(",")
            .append("Attribute Name").append(",").append("Attribute Type")
            .append(",").append("RTF Node").append(LINE_SEPARATOR);

        System.out.println("Generating . . . ");
        populateTaskFlowVoAttributes();
    }
}

```

On the execution of the utility, the Excel file is generated. The task codes can be filtered in the Excel file for viewing different RTF node value of different attributes available on the particular screen.

Figure 9–4 Excel Generation

A	B	C	D	E
Task Code	View Object	Attribute Name	Attribute Type	RTF Node
15 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	accountNo	java.lang.String	<?FundTermDepositVO_accountNo?>
16 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	principalBalance	java.math.BigDecimal	<?FundTermDepositVO_principalBalance?>
17 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	payinAmount	java.math.BigDecimal	<?FundTermDepositVO_payinAmount?>
18 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	transactionRefNo	java.lang.String	<?FundTermDepositVO_transactionRefNo?>
19 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	userRefNo	java.lang.String	<?FundTermDepositVO_userRefNo?>
20 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	acctCCY	java.lang.String	<?FundTermDepositVO_acctCCY?>
21 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	productCode	java.lang.String	<?FundTermDepositVO_productCode?>
22 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	partyId	java.lang.String	<?FundTermDepositVO_partyId?>
23 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	branchId	java.lang.String	<?FundTermDepositVO_branchId?>
24 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	primaryReason	java.lang.String	<?FundTermDepositVO_primaryReason?>
25 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	secondaryReason	java.lang.String	<?FundTermDepositVO_secondaryReason?>
26 TD002	com.ofss.fc.ui.model.td.mixedpayin.vo.FundTermDepositVO.xml	narrative	java.lang.String	<?FundTermDepositVO_narrative?>

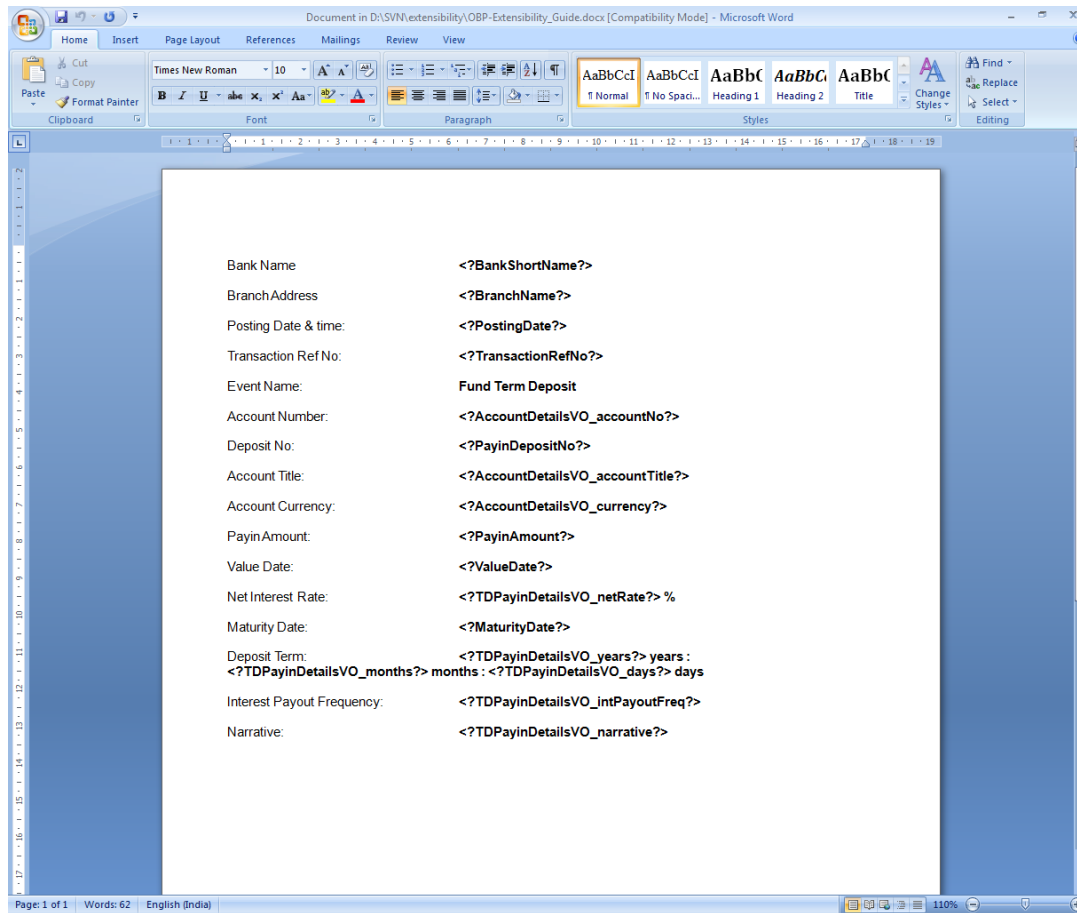
↑ Task Code      ↑ View Object Path (Screen/taskflow)      ↑ VO Attribute Name      ↑ Attribute Type      ↑ Reference in RTF template

## 9.1.2 Receipt Format Template (.rtf)

This template is used for defining the format of the output receipt. Different data elements which need to be shown in the output receipt are mentioned in this RTF report format template. The node will be taken from the above generated Excel file from 'RTF Node' column for specifying the output value in the final output RTF.

The sample rtf template is shown below:

**Figure 9–5 Receipt Format Template**



## 9.2 Configuration

This section describes the configuration details.

### 9.2.1 Parameter Configuration in the BROPConfig.properties

Following configuration parameters are required to be set in the BROPConfig.properties file.

- receipt.print.copy: Set to 'S' (default) if Single receipt is required. Else, set to 'M' for multiple receipts. The receipt will be stored in current posting date 'month/date' folder structure.
- receipt.base.in.location: Location for the RTF templates, which is configured as 'config\receipt\basein\template\' structure on the UI server. (For RTF development purpose this location will also have the XML generated while processing receipt.)
- receipt.base.out.location: Location for generated receipt, which is configured as 'config\receipt\baseout\' structure on the UI server.
- ui.service.url : UI URL http://IP:port format.

## 9.2.2 Configuration in the ReceiptPrintReports.properties

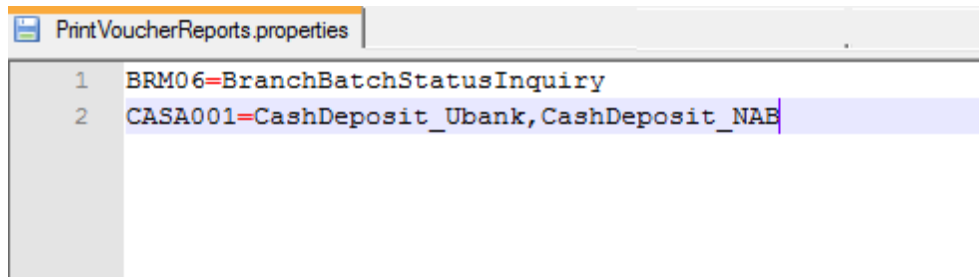
This file contains the key value pair of the Task Code of the screen and the corresponding template names, comma separated if more than 1 template is referred by screen.

### TaskCode=RTF Filename

Where TaskCode: task code of screen for which receipt print will be enabled and RTF Filename: name of the RTF template which will be used for the creation of the output with the same filename.

For example, TD002=FundTermDeposit

**Figure 9–6 Receipt Print Reports**



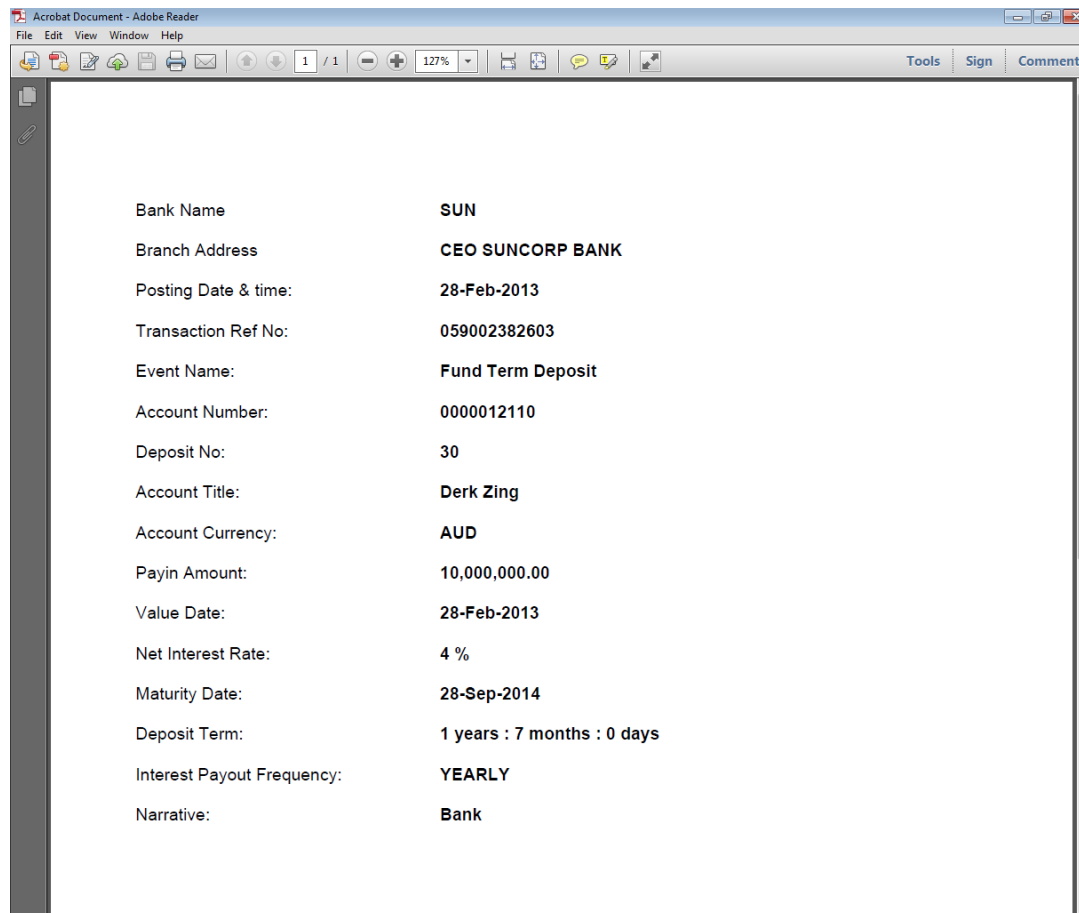
```
PrintVoucherReports.properties
1 BRM06=BranchBatchStatusInquiry
2 CASA001=CashDeposit_Ubank,CashDeposit_NAB
```

## 9.3 Implementation

The implementation for the print receipt functionality is explained in the following steps:

1. Once the screen is opened, Template checks '*ReceiptPrintReports.properties*' file if the Task code of the opened screen is present in the property file. The 'Receipt Print' button will be rendered in a disabled state.
2. On successful completion of transaction (successful Ok click), Receipt Print button gets enabled.
3. On click of Receipt Print button, all the VO's on current screen are fetched and created as a XML with data (for RTF development reference, this XML is not deleted at the moment but on environments these will be deleted). The RTF and XML merge up to create and open the receipt in the pdf format.
4. Receipt will be stored with the file name as <Logged in userId\_TemplateName>

The sample output receipt in the PDF form is shown below:

**Figure 9–7 Sample of Print Receipt**

### 9.3.1 Default Nodes

As per the functional specification requirement, some default nodes are already added in the generated XML. The list of those nodes are as follows:

- BankCode
- BankShortName
- BranchName
- PostingDate
- UserName
- BankAddress
- BranchAddress
- LocalDateTimeText

## 9.4 Special Scenarios

There are some cases, where some of the attributes are not available in the VOs of the screen and the value needs to be picked from the response of the transaction. There are also some data values which need to be

formatted first and then published in the PDF.

These values can be added to the pageFlowScope Map variable 'receiptPrintOtherDetailsMap'.

The population of those values needs to be done in the Backing Bean, after getting the response of the transaction in the following manner:

```

MessageHandler.sendMessage(payinResponse.getStatus());
receiptDetails.put("TransactionRefNo",payinResponse.getStatus
().getInternalReferenceNumber());
SimpleDateFormat receiptTimeFormat = new SimpleDateFormat("hh:mm:ss
a");
SimpleDateFormat receiptDateFormat = new SimpleDateFormat("dd-MMM-
YYYY");
HashMap<String,String> receiptDetails = new HashMap<String, String>
();
Date date=new Date(getSessionContext().getLocalDateTimeText());
receiptDetails.put("PostingTime", receiptTimeFormat.format
(date.getSQLTimestamp()));
if(payinResponse!=null && payinResponse.getValueDate()!=null) {
receiptDetails.put("ValueDate",receiptDateFormat.format
(payinResponse.getValueDate().getSqlDate()));
}
ELHandler.set("#{pageFlowScope.receiptPrintOtherDetailsMap}",
receiptDetails);

```

Internally, the functionality adds all the details in map variable, other than VO's data. While receipt printing, template checks the Map variable and if not null, it gets all the key-value from the map and show them in XML which is used later on for generation of receipt.



# 10 Extensibility Usage – OBP Localization Pack

OBP shall be releasing localization pack which ensures an optimized implementation period by adapting the product to different regions by implementing common region specific features pre-built and shipped. Every bank in different regions have different tax laws, different financial policies and so on. The policies in US will be different from those in Australia.

The localization packs leverage OBP extensibility to incorporate regional features and requirements by implementing different extension hooks for host and / or different JDeveloper customization functionalities for UI layer. This section presents a use case from OBP localization pack as implemented using the extensibility guidelines as a sample which can be referred to and followed as a guideline. Customization developers can implement bank's specific requirements on similar lines.

For example, in LCM022 'Perfection Capture' screen, the details section is shown with the additional fields which are defined for a particular location.

Figure 10–1 Perfection Capture Screen

The screenshot displays the 'Perfection Capture' screen in the LCM022 application. The interface includes a top navigation bar with menu items like 'Account', 'Back Office', 'CASA', 'Channel', 'Collection', 'LCM', 'Loan', 'Origination', 'Party', 'Payment And Collection', 'Securi', and 'Fast Path'. Below the navigation bar, the screen title 'Perfection Capture' is visible, along with action buttons: 'Read', '+ Create', 'Print', 'OK', 'Clear', and 'Cancel'. A 'Perfect Charge' button is located below the title. The main content area is divided into sections: 'Collateral Perfection Details' and 'View Document Details'. The 'Collateral Perfection Details' section contains several input fields and labels, including 'Charge Registration Number', 'Execution Date', 'Charge Registration Date', 'Date of Stamping', 'Registration Amount', 'Stamping Amount', 'End Date Changed', 'Amended Date', 'Removed Date', 'Token', 'Change Number', 'Giving Of Notice', 'Earlier Registration Number', 'Charge Status' (Proposed), 'Title Documents Status', 'Charge Registration Required' (Yes), 'Stamping Required' (Yes), 'Processed Date', 'Amended By', 'Removed By', 'Secured Party Group', 'Origination', and 'Transitional'. A red rectangular box highlights the fields from 'End Date Changed' to 'Earlier Registration Number'. The 'View Document Details' section at the bottom includes a 'View' dropdown and a 'Detach' button, followed by a table with columns 'Index Type' and 'Index value'.

## 10.1 Localization Implementation Architectural Change

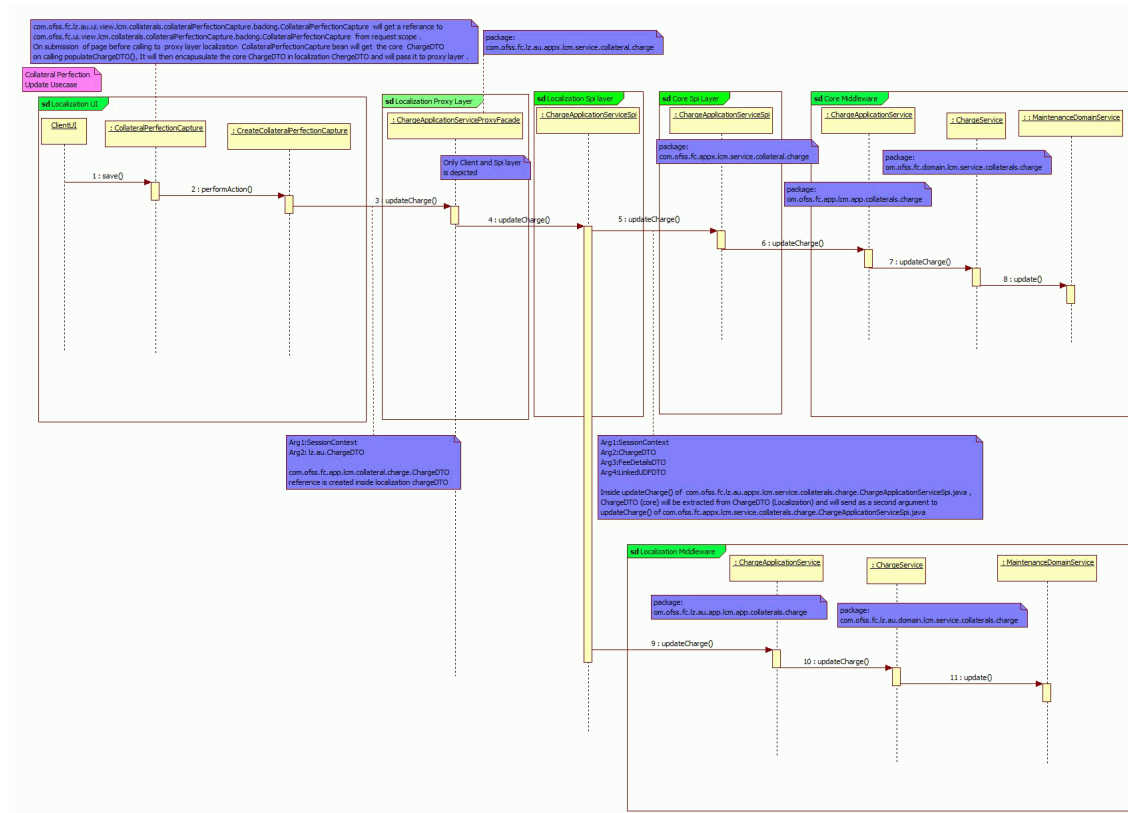
Architecturally, the following points are considered:

- Localization package will be over and above the product.
- Customization packages will be over the Localization and the Product.
- Any changes done for Localization should ensure that future product changes as well as customization changes will work seamlessly without any impact.

The additional fields which get identified and developed as part of localization requirements are in its own project, package, configuration, constant files and tables.

For example, the typical flow of the above mentioned perfection attributes added as part of localization requirement is shown below:

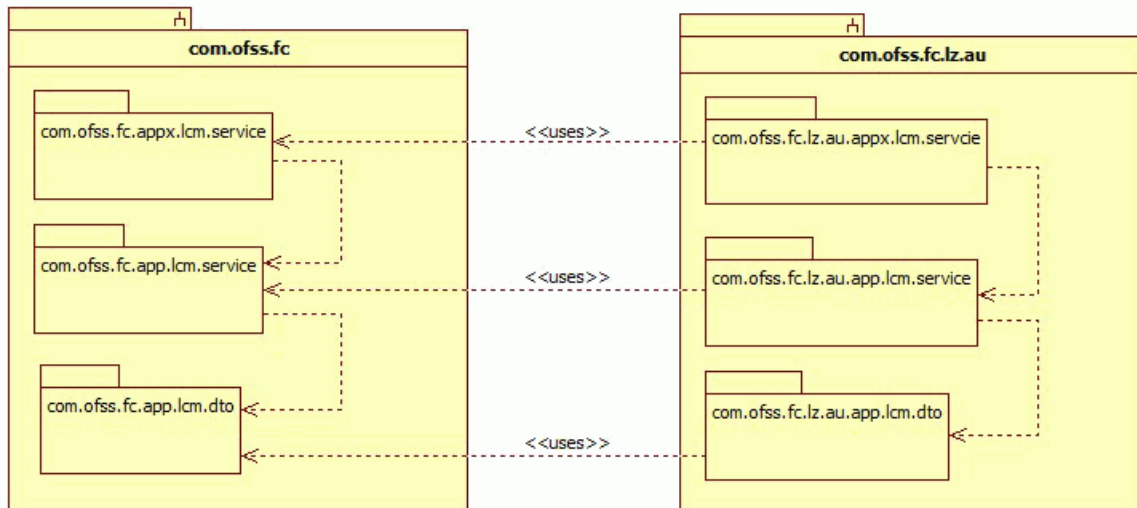
**Figure 10–2 Localization Implementation Architectural Change**



The Package structure for the implementation layer is shown below:



Figure 10–3 Package Structure



## 10.2 Customizing UI Layer

This section explains the customization of UI layer.

### 10.2.1 JDeveloper and Project Customization

For the customization of the UI layer, JDeveloper needs to be configured in the customizable mode as explained in the ADF Screen Customization Sections.

The example for the customization of the JDeveloper is described below:

#### CustomizationLayerValues.xml

Figure 10–4 Customization of the JDeveloper

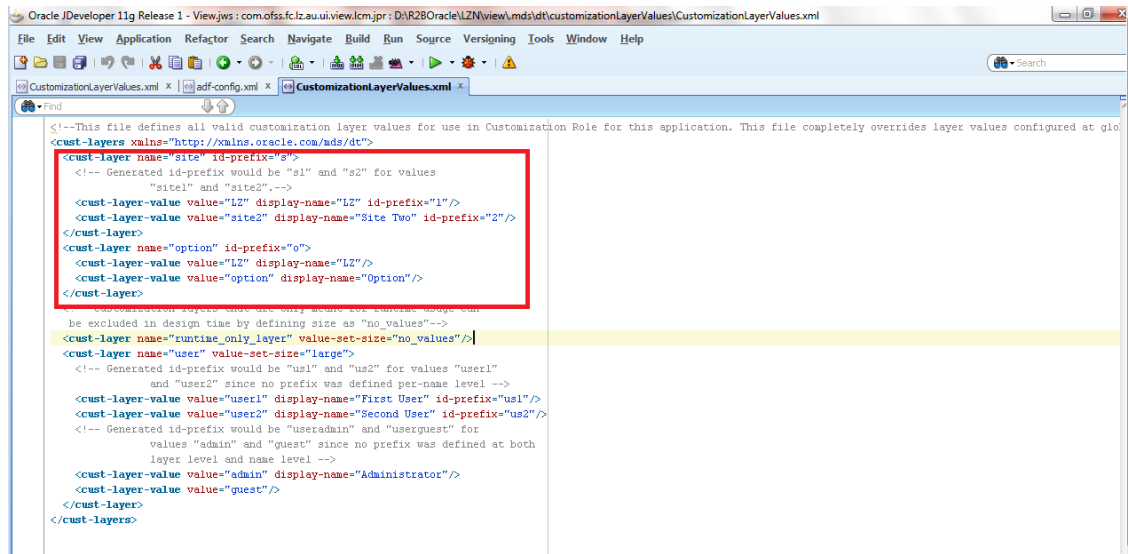
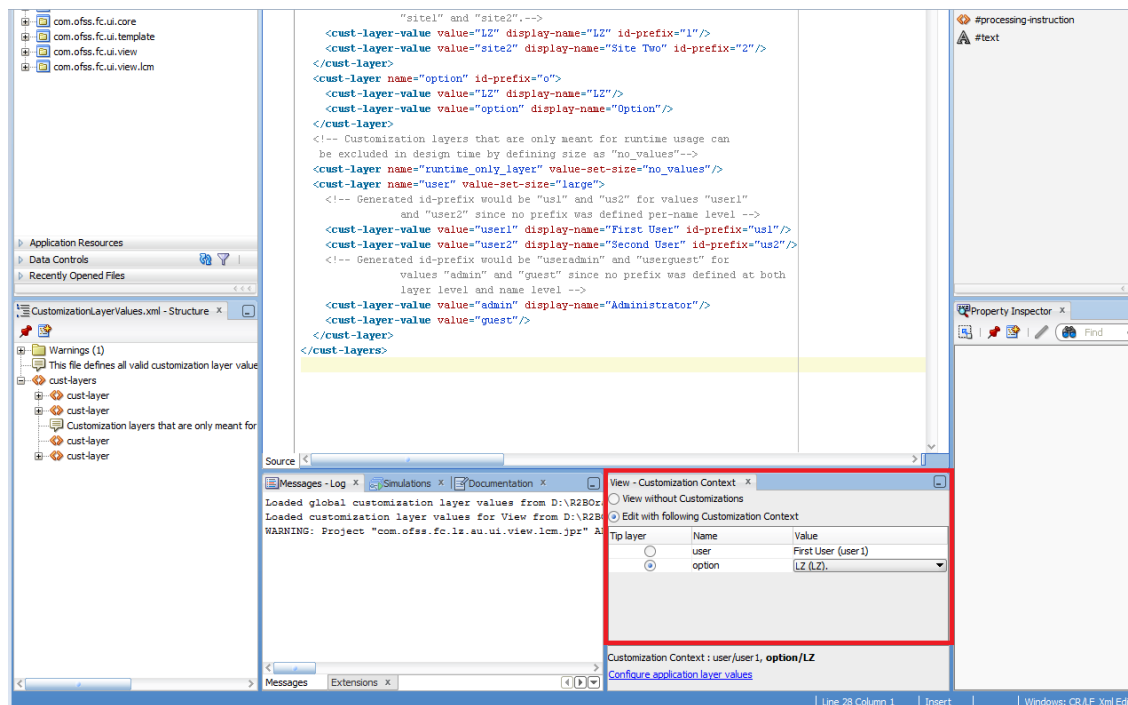


Figure 10–5 Customization of the JDeveloper



### adf-config.xml

If the changes are not reflecting, adf-config.xml needs to be opened from the application resources and *Configure Design Time Customization layer values* highlighted in the below image needs to be clicked. It will create a CustomizationLayerValues.xml inside MDS DT folder in application resources. All the content from

<JDEVELOPER\_HOME>/jdeveloper/jdev/CustomizationLayerValues.xml needs to be copied to this CustomizationLayerValues.xml. This is to ensure that the changes are reflected at the application level.

**Figure 10–6 Configure Design Time Customization layer**

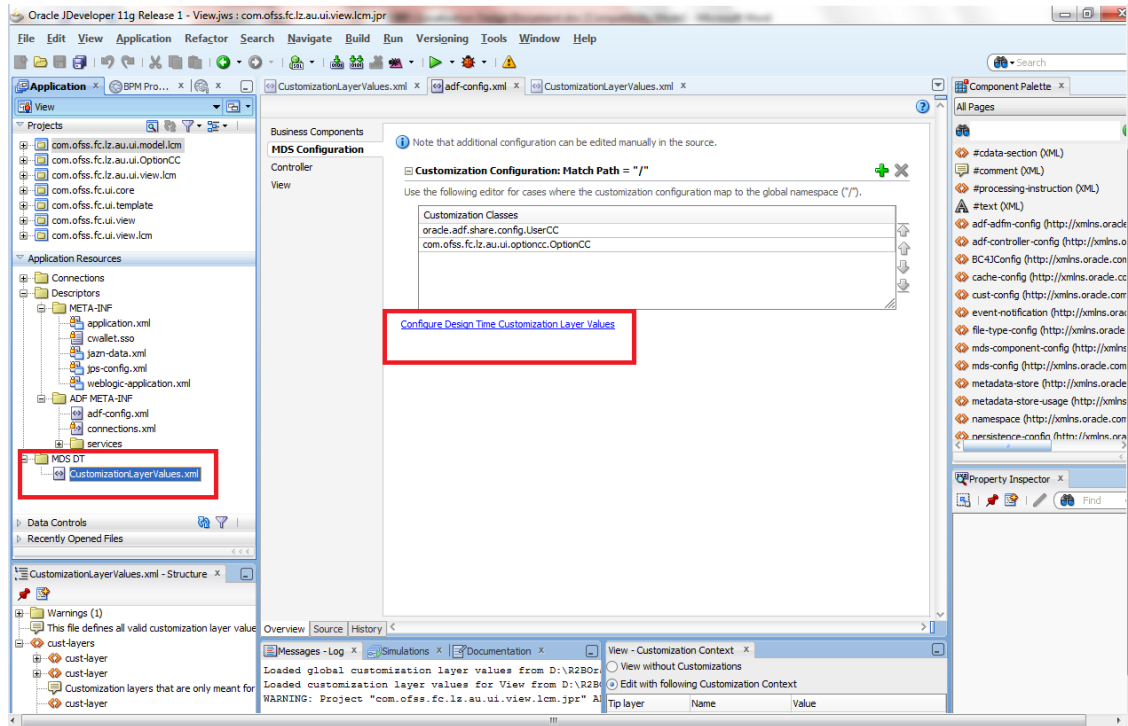
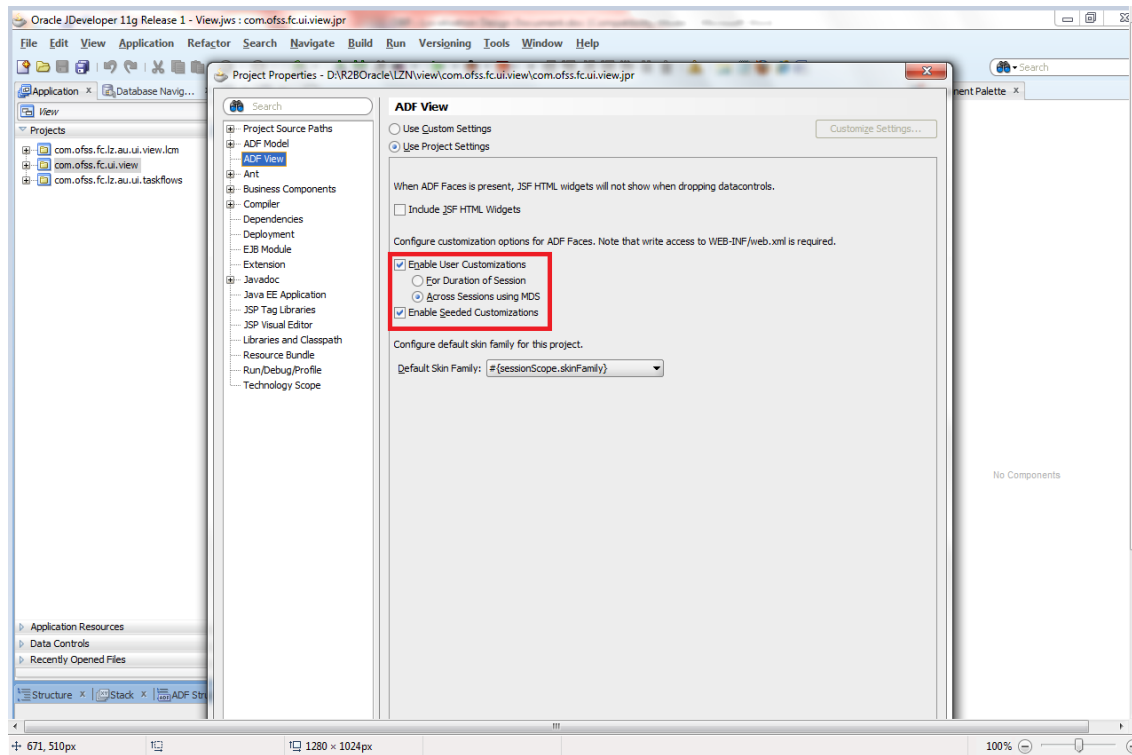


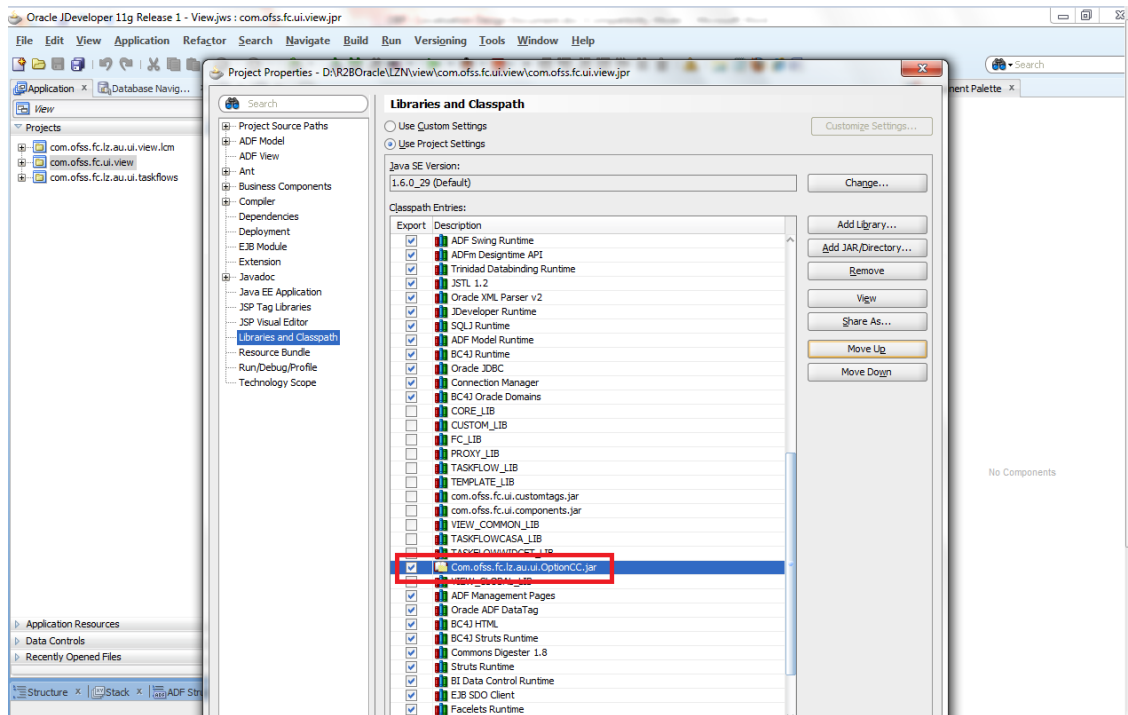
Figure 10–7 Enabling Seeded Customization



### Libraries and Classpath

In the "*Libraries and Classpath*" section, the previously deployed *com.ofss.fc.lz.au.ui.OptionCC.jar* containing the customization class then needs to be added.

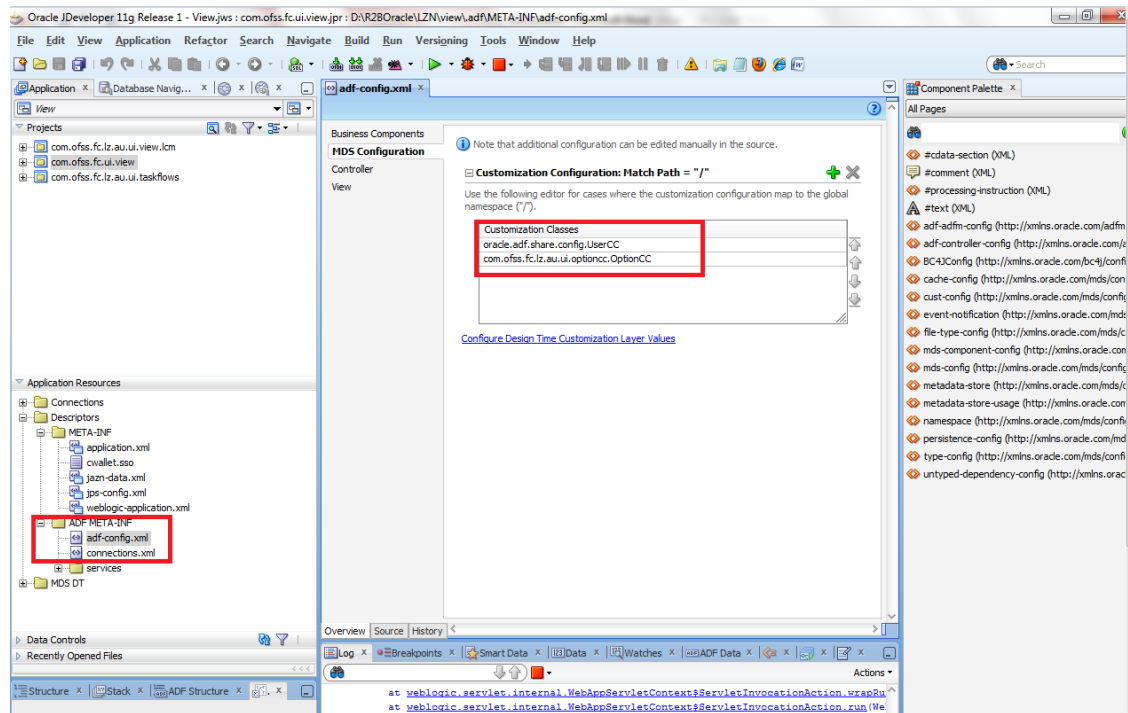
Figure 10–8 Library and Class Path



### adf-config.xml

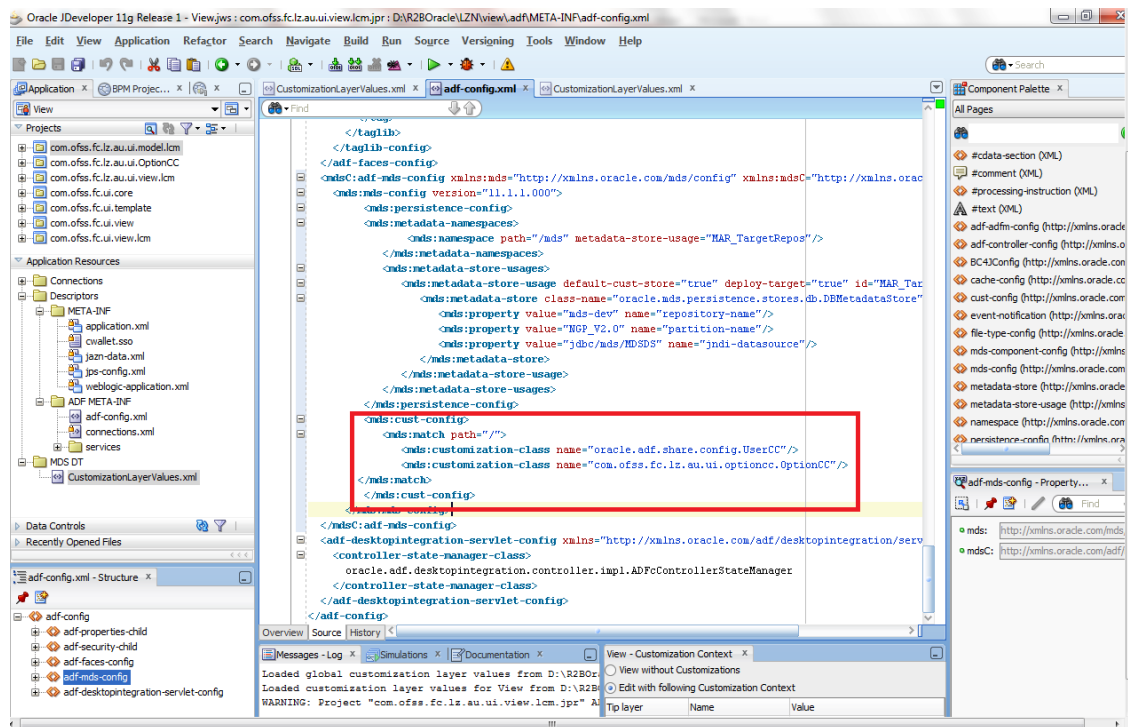
In the *Application Resources* tab, the *adf-config.xml* present in the *Descriptors/ADF META-INF* folder needs to be opened. In the list of *Customization Classes*, all the entries should not be removed and the *com.ofss.fc.lz.au.ui.OptionCC.OptionCC* class to this list needs to be added.

Figure 10–9 MDS Configuration



Jdeveloper is then restarted and the entry needs to be checked for *com.ofss.fc.lz.au.ui.OptionCC*. If the jar entry is not reflecting, then source needs to be clicked and the entry as highlighted and shown in the below image needs to be manually added.

Figure 10–10 MDS Configuration



## 10.2.2 Generic Project Creation

After creating the Customization Layer, Customization Class and enabling the application for Seeded Customizations, the next step is to create a project which will hold the customizations for the application. Generic project is then created with the following technologies:

- ADF Business Components
- Java
- JSF
- JSP and Servlets

Following jars must then be added to the Project Properties and in the classpath:

- Customization class JAR (*com.ofss.fc.lz.au.ui.OptionCC.jar*)
- The project JAR which contains the screen / component to be customized. For example, if you want to customize the Collateral Perfection Capture screen, the related project JAR is *com.ofss.fc.ui.view.lcm.jar*.
- All the dependent JARS / libraries for the project needs to added.
- Finally newly created project (example: '*com.ofss.fc.lz.au.view.lcm*') needs to be enabled for Seeded Customizations.

## 10.2.3 MAR Creation

After implementing customizations on objects from an ADF library, the customization metadata is stored by default in a subdirectory of the project called *libraryCustomizations*. Although ADF library customizations at

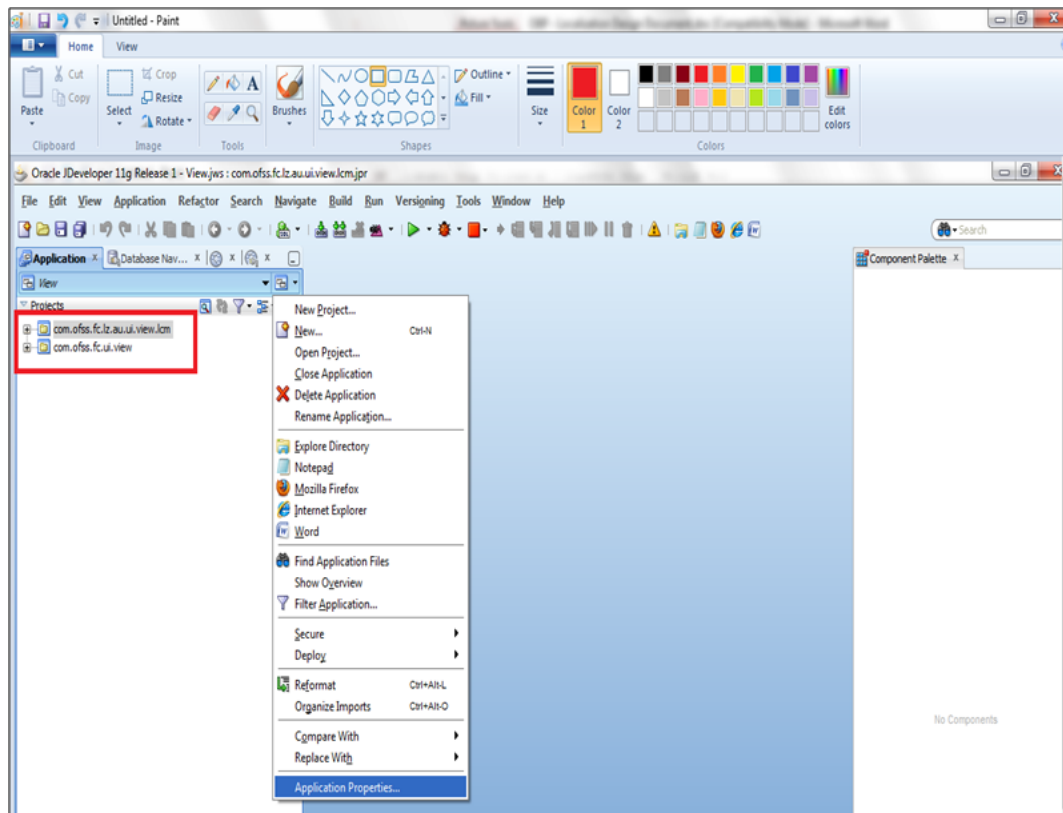
the project level is created and merged together during packaging to be available at the application level at runtime. Essentially, ADF libraries are JARs that are added at the project level, which map to library customizations being created at the project level. However, although projects map to web applications at runtime, the MAR (which contains the library customizations) is at the EAR level, so the library customizations are seen from all web applications.

Therefore, an ADF library artifact are customized in only one place in an application for a given customization context (customization layer and layer value). Customizing the same library content in different projects for the same customization context would result in duplication in MAR packaging. To avoid duplicates that would cause packaging to fail, customizations are implemented for a given library in only one project in your application.

### Step 1

Select the Application Properties.

**Figure 10–11 MAR Creation**

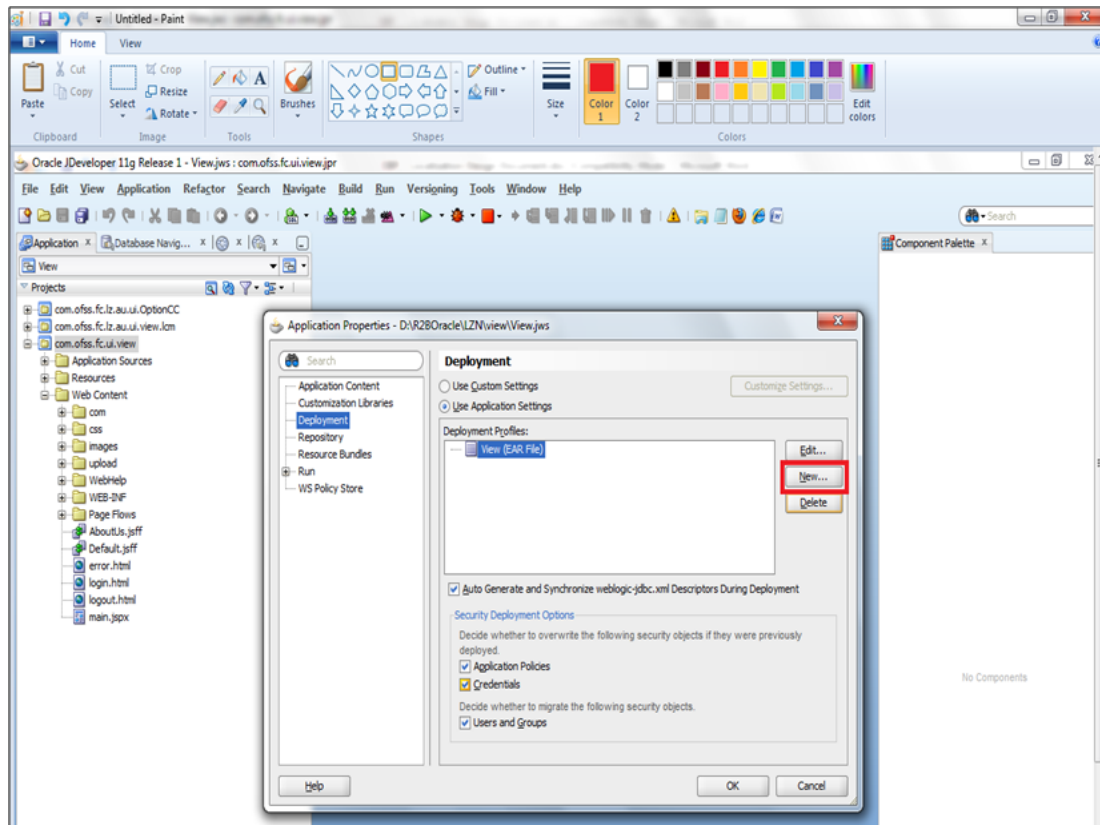


### Step 2

Import com.ofss.fc.lz.au.ui.view.lcm project into application. Click Application Menu and select Application Properties.



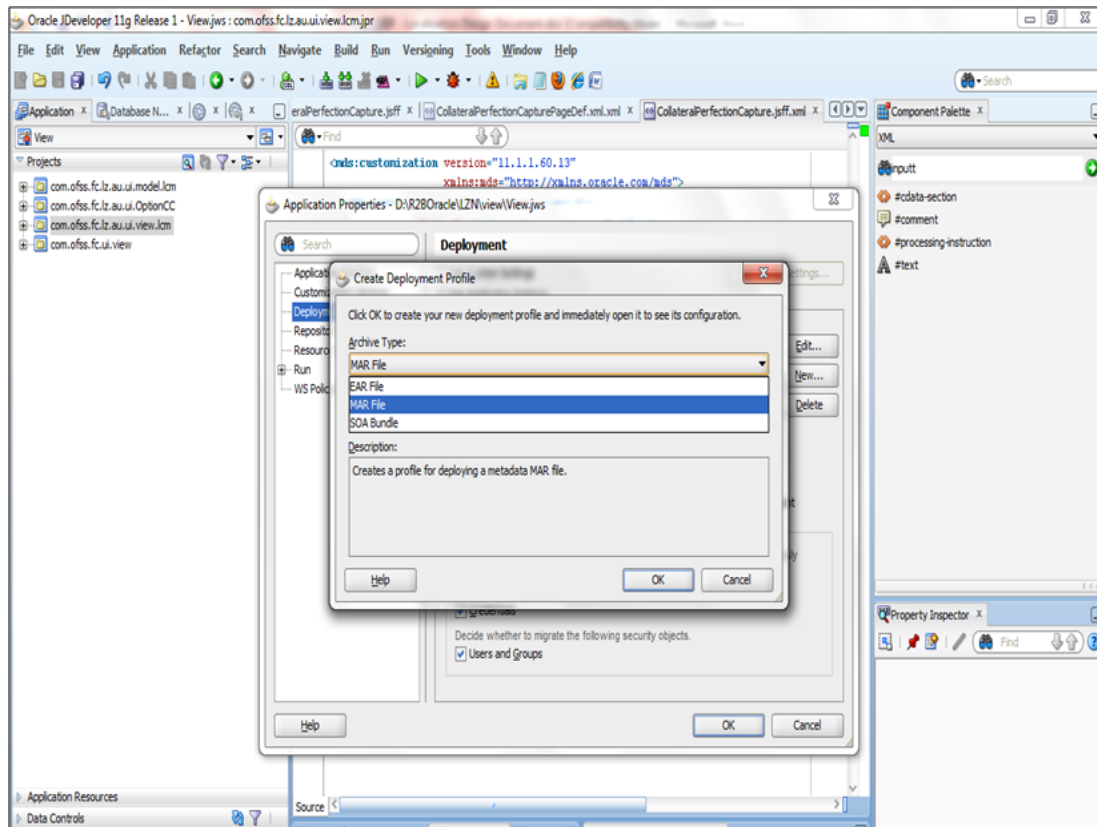
Figure 10–12 MAR Creation - Application Properties



### Step 3

Select Deployment and click **New**.

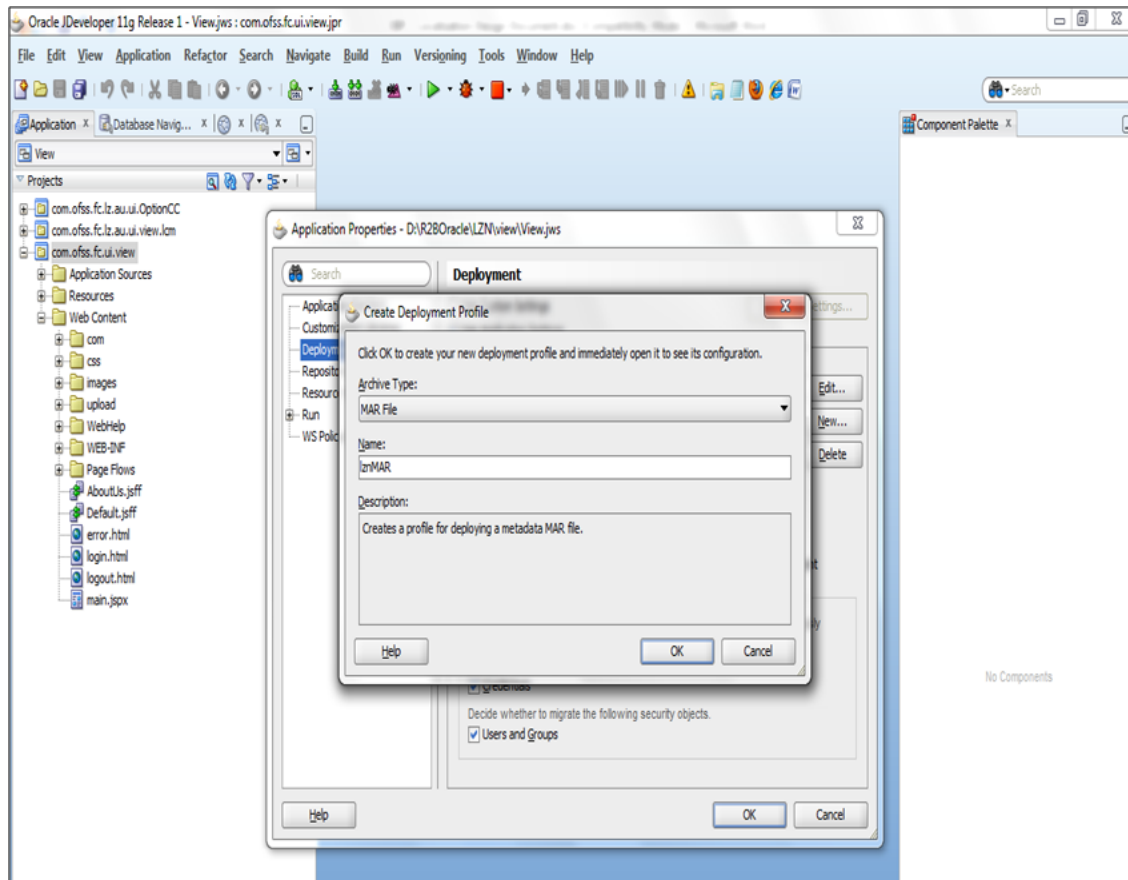
Figure 10–13 MAR Creation - Create Deployment Profile



**Step 4**

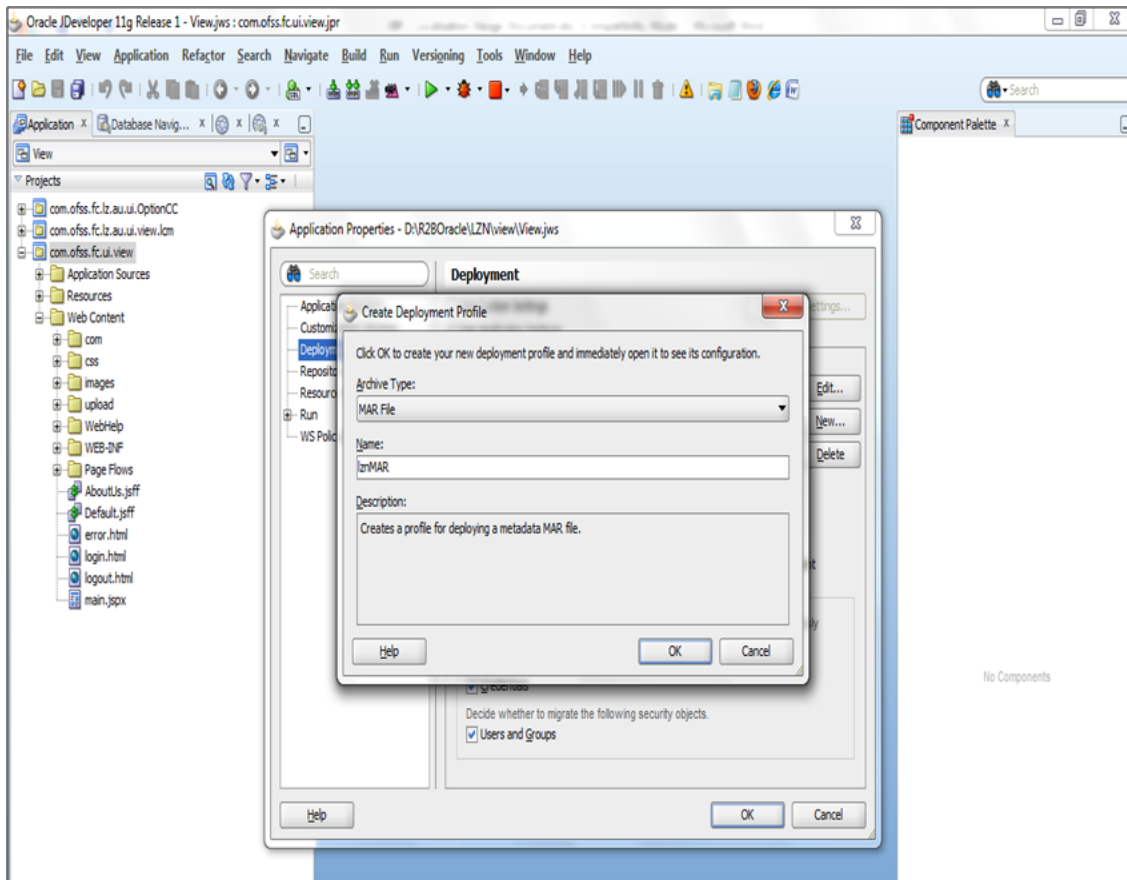
Select the MAR File option.

Figure 10–14 MAR Creation - MAR File Selection

**Step 5**

Select MAR from Archive Type and give a name ending with MAR and click **Ok**.

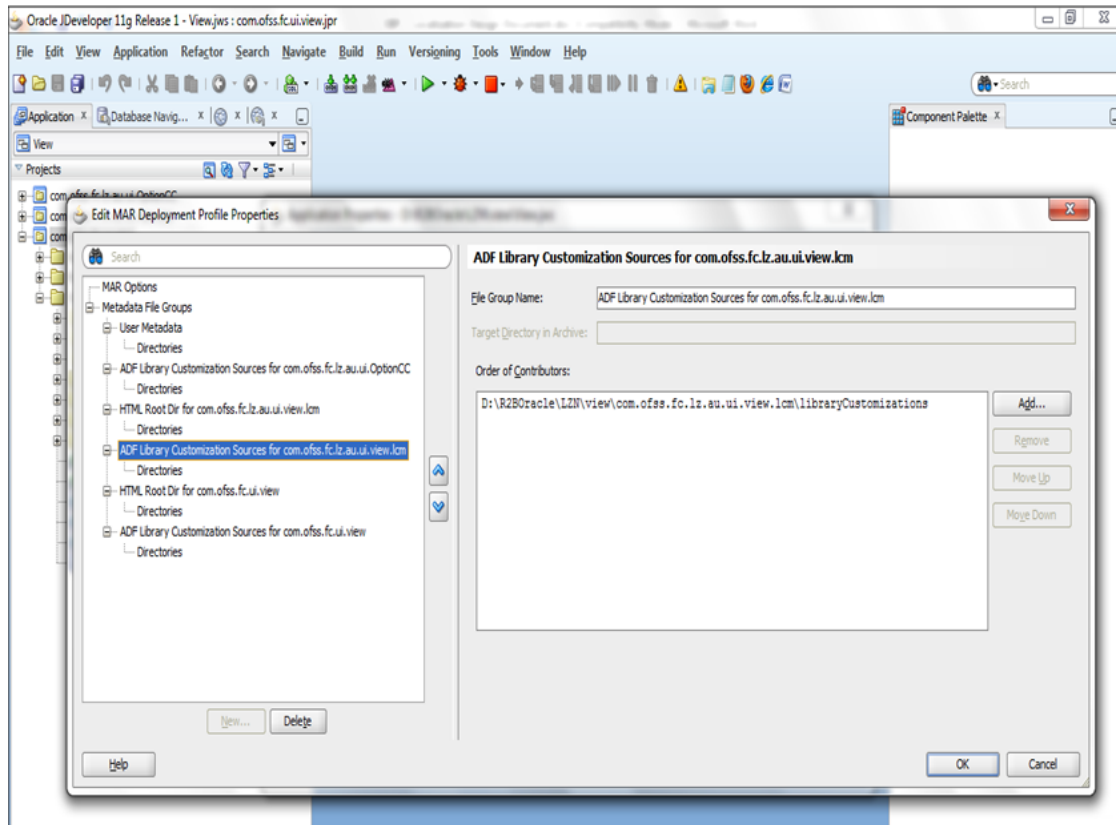
Figure 10–15 MAR Creation - Enter Details



**Step 6**

Select the ADF Library Customization for com.ofss.fc.lz.au.ui.view.lcm.

Figure 10–16 MAR Creation - ADF Library Customization

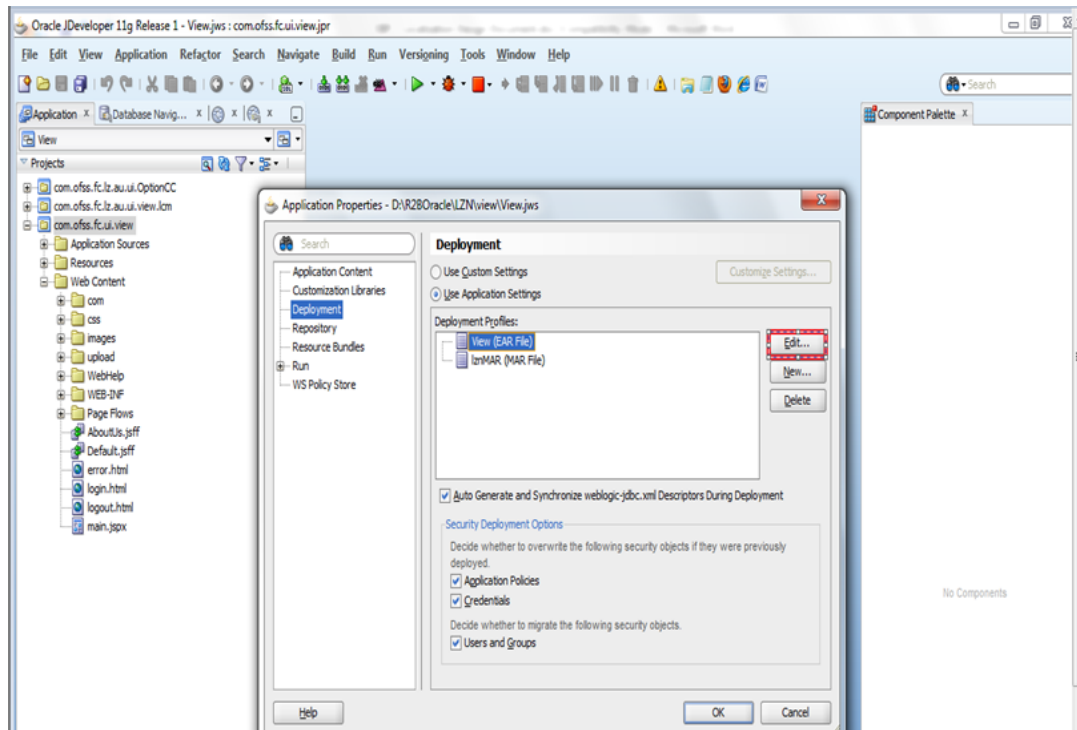
**Step 7**

Select the project for which Library Customization will be included in MAR (com.ofss.fc.lz.au.ui.view.lcm) and click **OK**.

**Step 8**

Select View (EAR File) and click Edit.

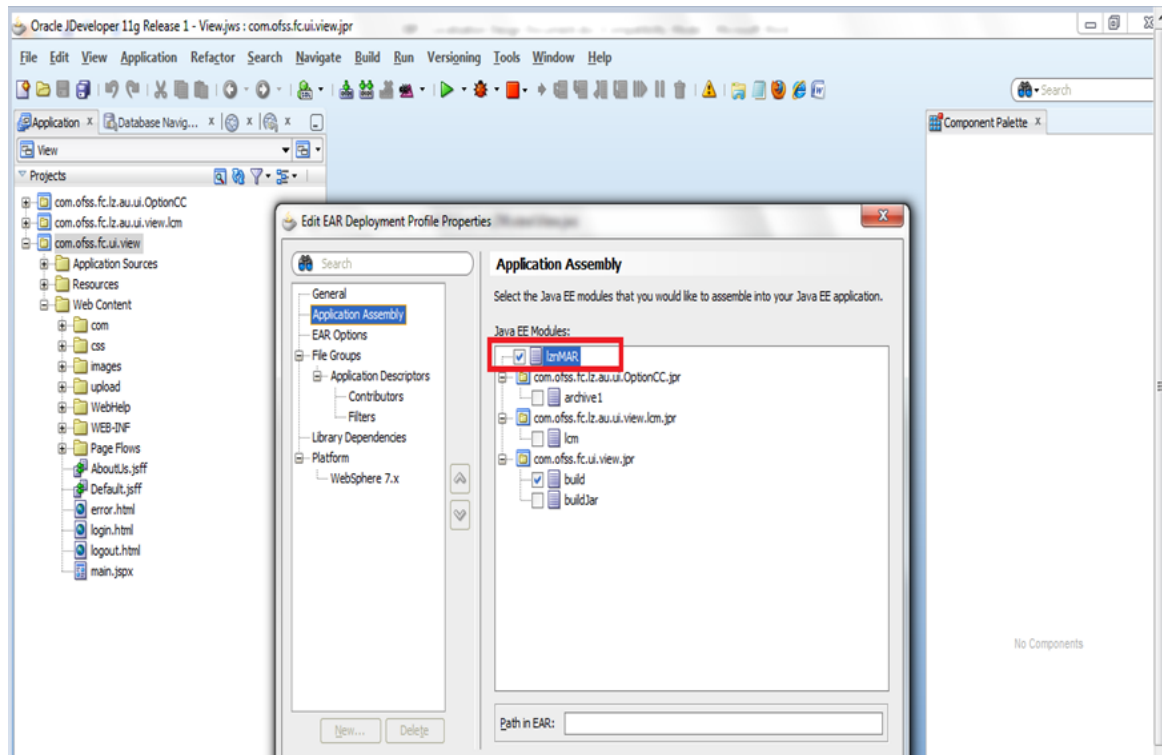
Figure 10–17 MAR Creation - Edit File



**Step 9**

Select Application Assembly and check the created MAR (lznMAR) and click ok on defaults.

Figure 10–18 MAR Creation - Application Assembly



## 10.3 Source Maintenance and Build

This section describes the source maintenance and build details.

### 10.3.1 Source Check-ins to SVN

Along with UI and middleware source maintenance, there is a set of metadata files required to be packaged in the deployable packages in order for customization. When performing any changes to a product screen in "customization mode" the corresponding <screen filename>.xml gets generated. In case of taskflows, the metadata file is <page definition filename>.xml. The path structure is provided in the below table.

**Table 10–1 Path Structure**

For page definition	
File name (with path)	adfmsrc/com/ofss/fc/ui/view/lcm/collaterals/collateralPerfectionCapture/pageDefn/CollateralPerfectionCapturePageDef.xml
Meta-data file name	com\ofss\fc\ui\view\lcm\collaterals\collateralPerfectionCapture\pageDefn\mdssys\cust\option\LZ\CollateralPerfectionCapturePageDef.xml.xml

(with path)	
<b>For Screens</b>	
File name (with path)	com/ofss/fc/ui/view/lcm/collaterals/collateralPerfectionCapture/form/CollateralPerfectionCapture.jsff
Meta-data file name (with path)	com\ofss\fc\ui\view\lcm\collaterals\collateralPerfectionCapture\form\mdssys\cust\option\LZ\CollateralPerfectionCapture.jsff.xml

These meta-data sources are checked into the METADATA folder in the product SVN under the localization path. During deployment, the EAR implementing these customizations must include these above mentioned sources in a .mar file.

### 10.3.2 .mar files Generated during Build

The localization specific build will include a last step, which is creation of .mar (metadata archive) file from the files checked-in the METADATA folder. This step will create separate .mar files, based on the modules which these represent. These MAR files are then packaged inside the deployable application EAR (com.ofss.fc.ui.view.ear).

Typical mar files generated during build will follow the naming convention com.ofss.fc.lz.au.ui.view.<module>.mar. Example, com.ofss.fc.lz.au.ui.view.pc.mar

### 10.3.3 adf-config.xml

adf-config.xml stores design time configuration information. The cust-config section (under mds-config) in the file contains a reference to the customization class. As part of the build activity, this file needs to be placed in the path com.ofss.fc.ui.view.ear@/adf/META-INF/. Also the customization class should be available in the classpath during deployment.

## 10.4 Packaging and Deployment of Localization Pack

In the OBP application, different projects will be shipped in the form of library jars which can be customized and the new localization-specific application libraries can be created. In the application, the assembly has been specifically modularized to take care of multiple localizations by prevention of mix-up of jars. The naming convention for the jars can be defined for different clients differently.

The new customized jars for hosts and UI needs to be packed with the original jars in the EAR files which will be deployed on the server. Let's say, we are creating the extension hooks of 'obp.host.app.domain' jar, then the separate jars can be defined as 'lz.au.obp.host.app.domain' and 'lz.us.obp.host.app.domain' for Australia and US respectively.

The similar structure can also be maintained for the other applications across UI and SOA channels. 'lz.au.obp.ui.domain' can be defined for the customized jar of the project 'obp.ui.domain'.



The new customized jars for hosts and UI are packed below with the original jars in the EAR files which will be deployed on the servers.

Figure 10–19 Package Deployment

