OraFormsFaces™

Developer's Guide v3.1.5

COMMIT CONSULTING

October 2010

Authored by: Wilfred van der Deijl

OraFormsFaces Developer's Guide

Primary Author: Wilfred van der Deijl

1. Contents

2. Preface

Welcome to the OraFormsFaces Developer's Guide! OraFormsFaces is an innovative component library that allows you to fully integrate Oracle Forms and JSF web applications, something that simply cannot be done with Oracle Forms alone.

2.1. Audience

This manual is intended for enterprise application developers with a basic understanding of Oracle Forms and Java Server Faces (JSF), and who need to create and deploy a JSF application that embeds one or more Oracle Forms modules using the OraFormsFaces JSF component library. This guide explains how to build these applications using Oracle JDeveloper, Oracle Forms and the OraFormsFaces component library.

2.2. Related Documents

For more information, see the following documents:

- ***Oracle Application Server Forms Services 10gR2 Deployment Guide*** at http://download.oracle.com/docs/cd/B25016_04/doc/dl/web/B14032_03/toc.htm
- ***Oracle Fusion Middleware Forms Services 11gR1 Deployment Guide*** at http://download.oracle.com/docs/cd/E14571_01/web.1111/e10240/toc.htm
- ***JavaServer Faces Documentation*** at http://java.sun.com/javaee/javaserverfaces/reference/docs/
- ***JavaServer Faces API Specifications*** at http://java.sun.com/javaee/javaserverfaces/reference/api/
- ***Oracle Application Development Framework 10gR3 Developer's Guide for Forms/4GL Developers*** at http://download.oracle.com/docs/html/B25947_01/toc.htm
- ***Oracle ADF 10.1.3 Developer's Guide*** at http://www.oracle.com/webapps/online-help/jdeveloper/10.1.3/
- ***Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11gR1*** at http://download.oracle.com/docs/cd/E14571_01/web.1111/b31974/toc.htm
- ***Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework 11gR1*** at http://download.oracle.com/docs/cd/E14571_01/web.1111/b31973/toc.htm

2.3. Conventions

The following text conventions are used in this document:

Convention	Meaning
Boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
Italic	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 appear on the screen, or text that you enter.

3. Getting Started with OraFormsFaces

3.1. Introduction to OraFormsFaces

OraFormsFaces is a 100% Java Server Faces (JSF) compliant component library that offers UI components for JSF application development to embed Oracle Forms in a JSF page. Based on the JSF JSR 127 specification, OraFormsFaces components can be used in any IDE that supports JSF.

OraFormsFaces allows you to integrate one or more Oracle Forms modules in a JSF web application. This means you can build JSF web applications that leverage your existing investments in Oracle Forms.

The component library has some key features that cannot easily be implemented without using the library, including:

- Drag-and-drop support to include an Oracle Form onto a JSF web page
- JavaScript based API to allow JSF components (like buttons) to raise Oracle Forms PL/SQL triggers
- Drag-and-drop support to pass data control items as parameters or globals to Oracle Forms
- Ability to initiate JSF actions and navigations from Forms PL/SQL code
- Possibility to pass information back and forth between Forms PL/SQL and the JSF web application
- Reuse of the same Oracle Forms applet instance on each JSF web page, so the typical startup time of the applet is only endured once for each session
- Possibility to show only a limited part of the form, effectively clipping the viewport
- Requires minimal or no changes to existing Oracle Forms modules to be included in a JSF web application

3.2. Upgrading OraFormsFaces

Upgrading OraFormsFaces is very similar to doing a fresh installation of OraFormsFaces. Nonetheless, be sure to check the Release Notes of your OraFormsFaces version for any remarks or changes you will have to make to your existing OraFormsFaces enabled applications.

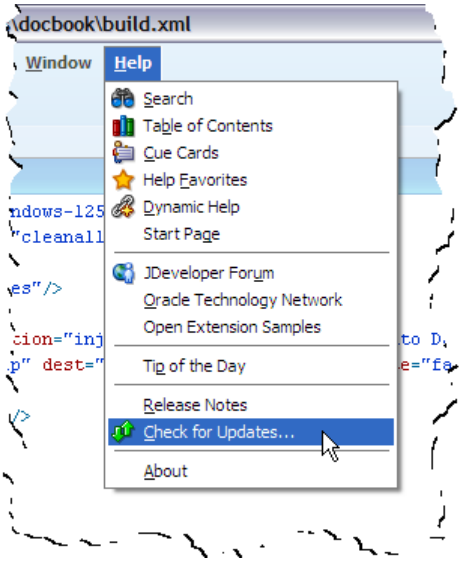
3.3. Installing OraFormsFaces

OraFormsFaces comes as a JDeveloper extension in a ZIP file and requires configuration of your Forms Server or Oracle Developer Suite. The instructions below are for the express setup assuming you are using JDeveloper. For manual installation instructions see 3.4 Manually Installing OraFormsFaces.

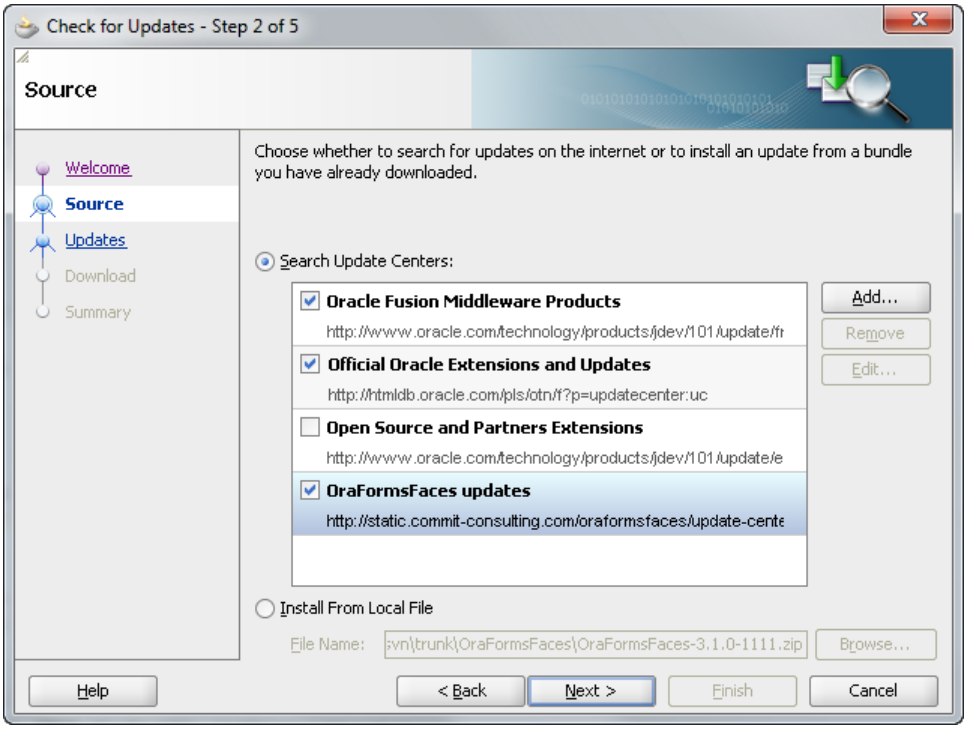
3.3.1. Installing JDeveloper Extension

This section describes how to install OraFormsFaces as a JDeveloper extension to Oracle JDeveloper 10.1.3.x or 11.1.x. These steps need to be performed only once for a JDeveloper installation and do not need to be repeated for each project

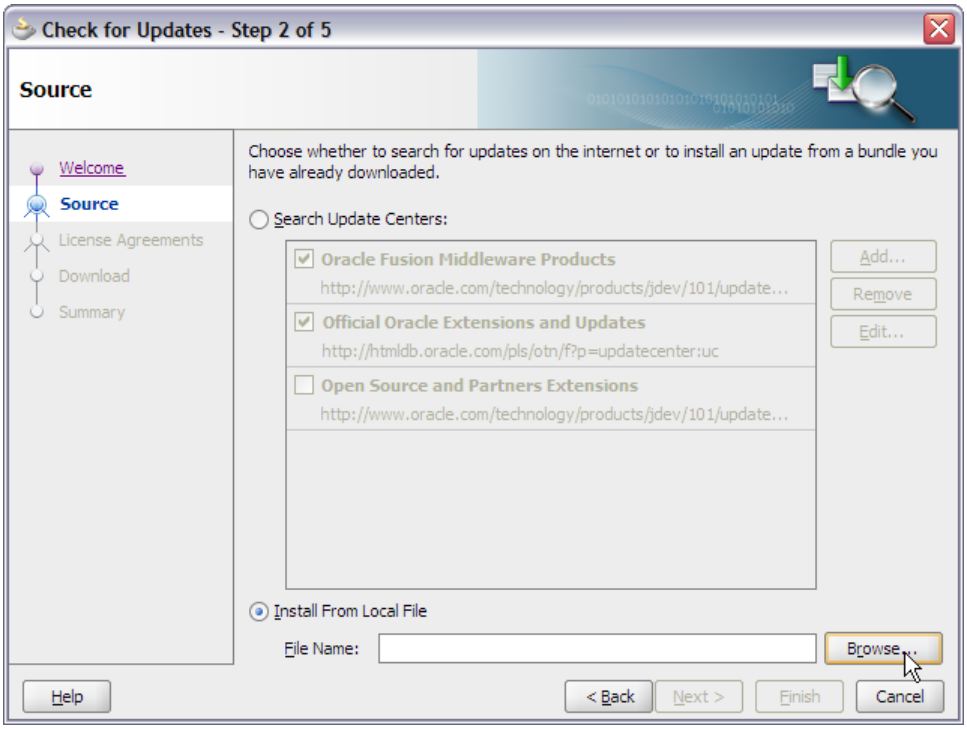
- Start JDeveloper
- Go to **Help > Check for Updates**



- If you already have a 3.x version of OraFormsFaces installed, the OraFormsFaces Update Center will be available for selection. If so, ensure the checkbox for this update center is checked and continue the wizard.:

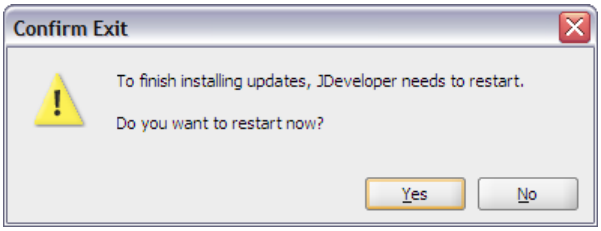


- If the OraFormsFaces Update Center is not available, check the radio button Install from Local File. Click the Browse… button and locate the OraFormsFaces ZIP file you downloaded from http://www.commit-consulting.com/oraformsfaces/



> **Note:** Be sure to use ZIP file corresponding to your JDeveloper version. There are different ZIP files for JDeveloper 10.1.3 and JDeveloper 11.1.1

- Finish the wizard to install the JDeveloper extension and restart JDeveloper



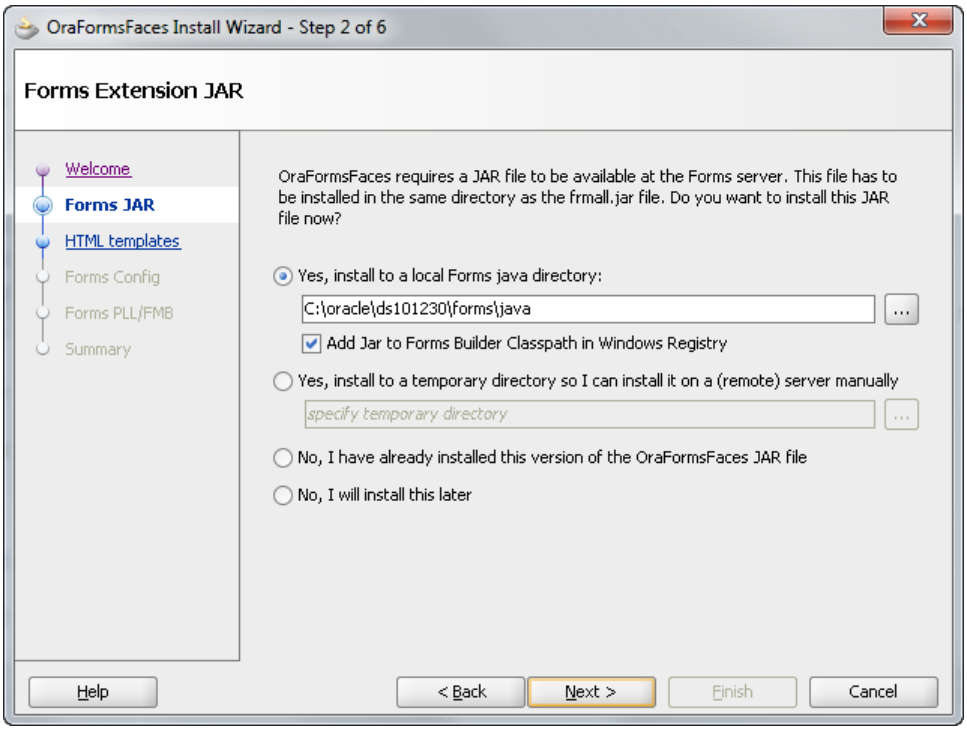
By installing the JDeveloper extension, the OraFormsFaces JAR file has been registered as a Java library and the JSP tag library it contains has also been registered. This allows you to add the OraFormsFaces library and JSP tags to any of your projects. It also hooked into JDeveloper so that attributes and methods from ADF Data Controls can be dragged and dropped onto a JSF page as OraFormsFaces `FormParameter` or `FormComponent` components while automatically registering the appropriate bindings in your ADF page definition files. More on this later

3.3.2. Running Installation Wizard

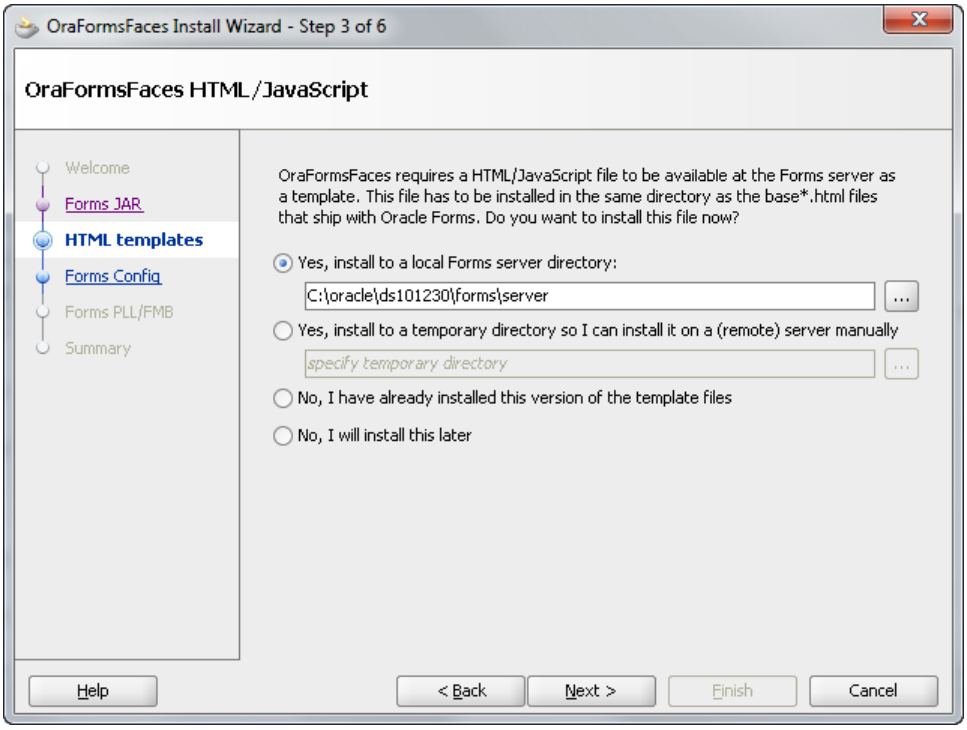
Since OraFormsFaces integrates JSF with Oracle Forms, there is also a substantial part of OraFormsFaces that is not implemented at the JSF server but rather at the Oracle Forms server. This requires some additional installation and configuration steps. These steps apply to both local Oracle Developer Suite installations as well as Oracle Application Server or Fusion Middleware installations. The OraFormsFaces Extension for JDeveloper 11 comes with a convenient installation wizard to make the required changes to your Oracle Forms environment. The installation wizard will automatically start after installing the JDeveloper 11 extension or it can be activated at any time from the **Tools** menu.

See *3.5 Manually Configuring Oracle Forms* for instructions on manually configuring Oracle Forms if you do not want to use the supplied wizard or run into problems using it.

- Activate the **OraFormsFaces Install Wizard** from the **Tools** menu in JDeveloper 11.1.x.
- The first wizard page is to specify the location of your Forms JAR files. OraFormsFaces needs to put additional JAR files in this directory with extensions to the Forms applet. The directory also contains the core `frmall.jar` and is typically located at `ORACLE_HOME\forms\java`

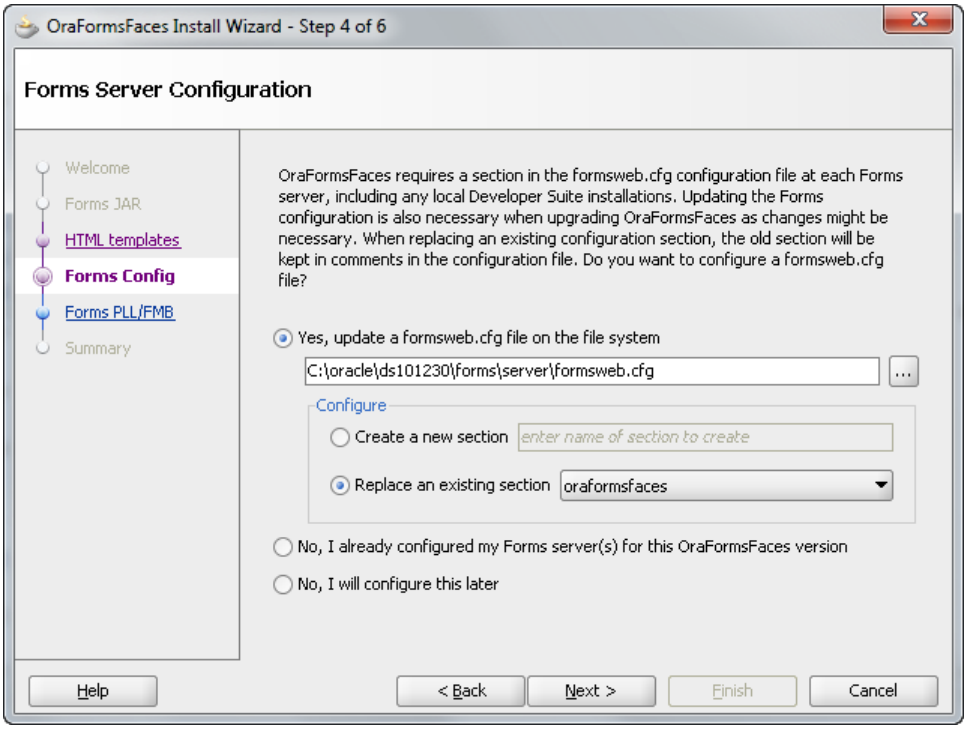


- The next page is to specify the directory where the Forms servlet template files are stored. OraFormsFaces needs to install an additional HTML/JavaScript template file. Please specify the directory that contains the Oracle Forms `base*.htm` files. For Forms 10.1.2 this typically is `ORACLE_HOME/forms/server` while Forms 11 stores these files in `INSTANCE_HOME/config/FormsComponent/forms/server`.

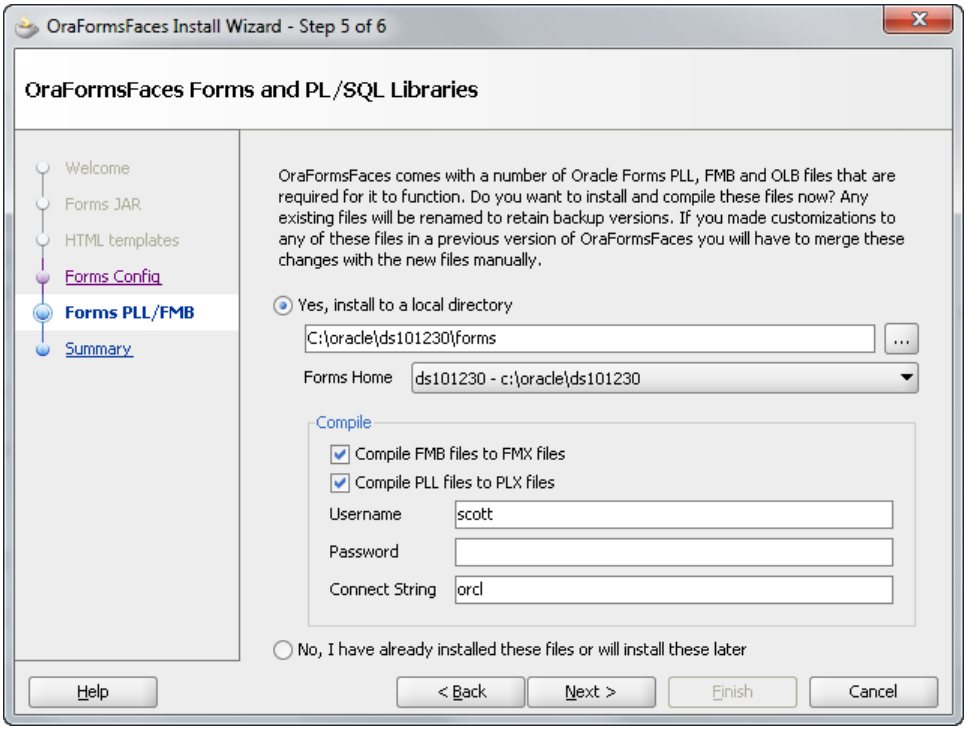


- Next specify the location of the Oracle Forms `formsweb.cfg` configuration file. OraFormsFaces needs a substantial number of custom parameters in this file. You can either choose to create a new configuration section where you can specify the name or select to replace an existing configuration section which is typically used when upgrading from a previous OraFormsFaces version. Be advised that replacing an existing configuration section will not merge the settings from the existing section with the new section. All settings will be overwritten and a copy of the old section will be retained in comments within the formsweb.cfg

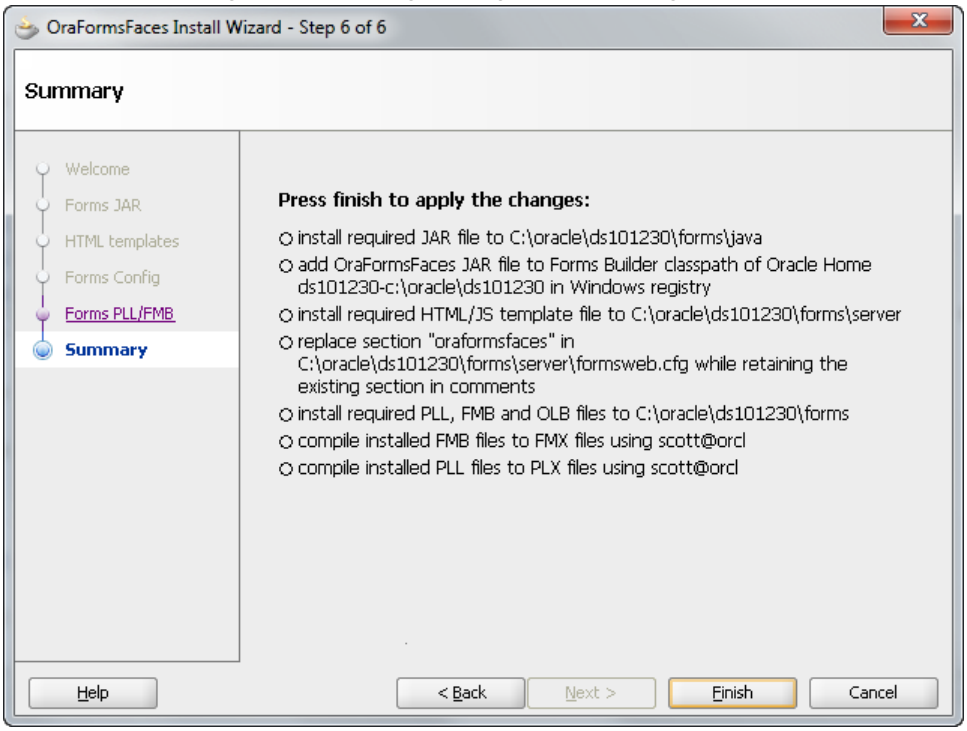
Note:	Be sure to review the `formsweb.cfg` file after the installation completes. Some parameters might need changes to fit your specific situation.



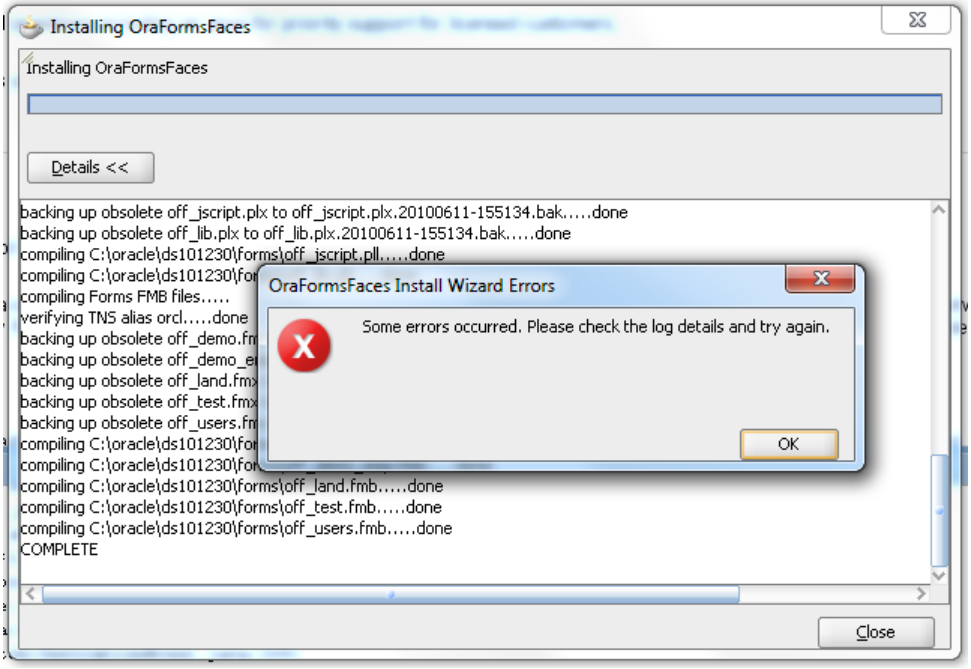
- Next specify the directory where OraFormsFaces can install its supporting FMB, PLL and OLB files. A default Forms installation is configured to read these files from `ORACLE_HOME/forms` for Forms 10.1.2.x or from `INSTANCE_HOME/FormsComponent/forms` for Forms 11.1.x. However, it is likely you normally use a different directory for these files. Feel free to select a different directory as long as you know your FORMS_PATH is setup to include this directory.
  You also need to specify which Forms Home to use for the Forms compiler. The installation wizard can compile the included files and needs to know which Forms compiler to use for this. If you select to compile the FMB or PLL files you will also need to enter database credentials to be used during compilation.



- The final step shows a summary of the installation tasks that will be performed when you press **Finish**. This is your last chance to go back and change settings before running the actual installation process.



- After pressing **Finish** in the **summary screen**, the actual installation will begin. A detailed log of the installation steps will show. If any errors occurred during the installation an alert will display.



3.4. Manually Installing OraFormsFaces

The easiest way to install OraFormsFaces is to install the JDeveloper extension as described in ***3.3.1 Installing JDeveloper Extension*** and then running the installation wizard as described in ***3.3.2 Running Installation Wizard***. The installation wizard is only available with JDeveloper 11 so even if you plan on only using OraFormsFaces with JDeveloper 10.1.3 you will have to install JDeveloper 11 and the OraFormsFaces extension at least once.

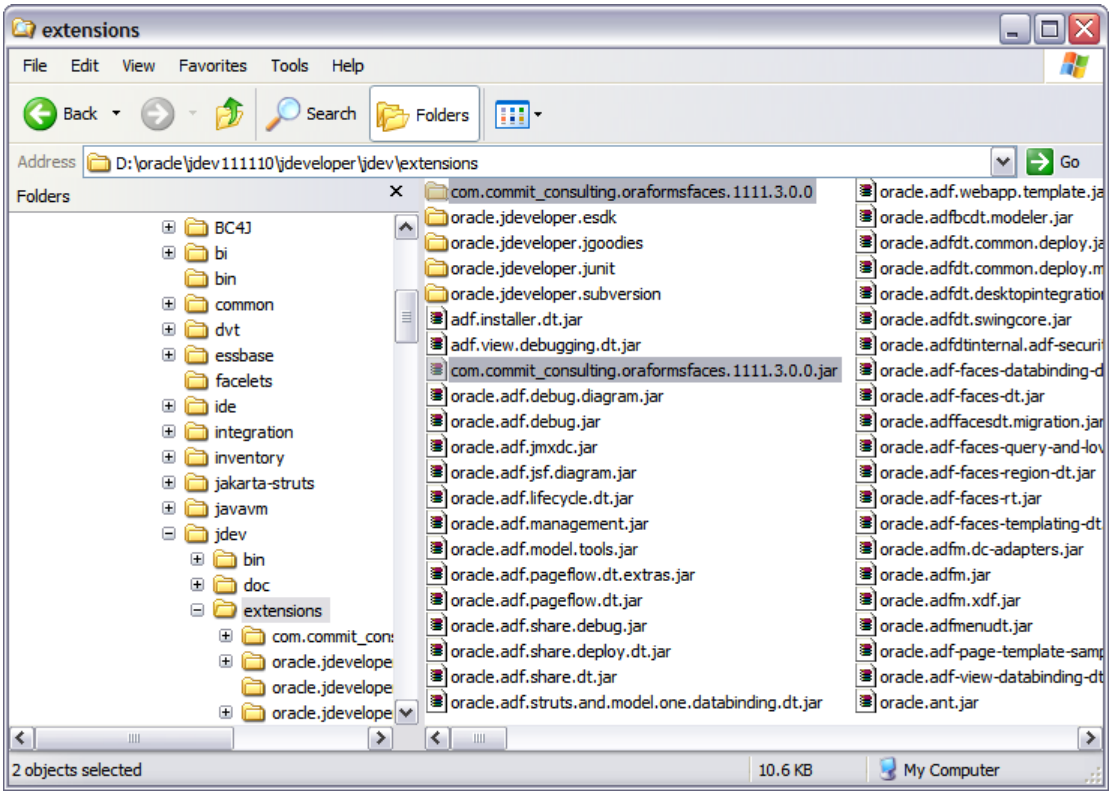
As an alternative to the installation wizard you can also perform the same installation steps manually. These manual steps are described below. These steps are still aimed at developers using JDeveloper but similar steps would have to be taken to configure other development environments.

3.4.1. Uninstalling Old JDeveloper Extensions

If you are upgrading OraFormsFaces and previously used an older version of OraFormsFaces it is likely you will need to perform some manual steps for uninstalling OraFormsFaces from JDeveloper.

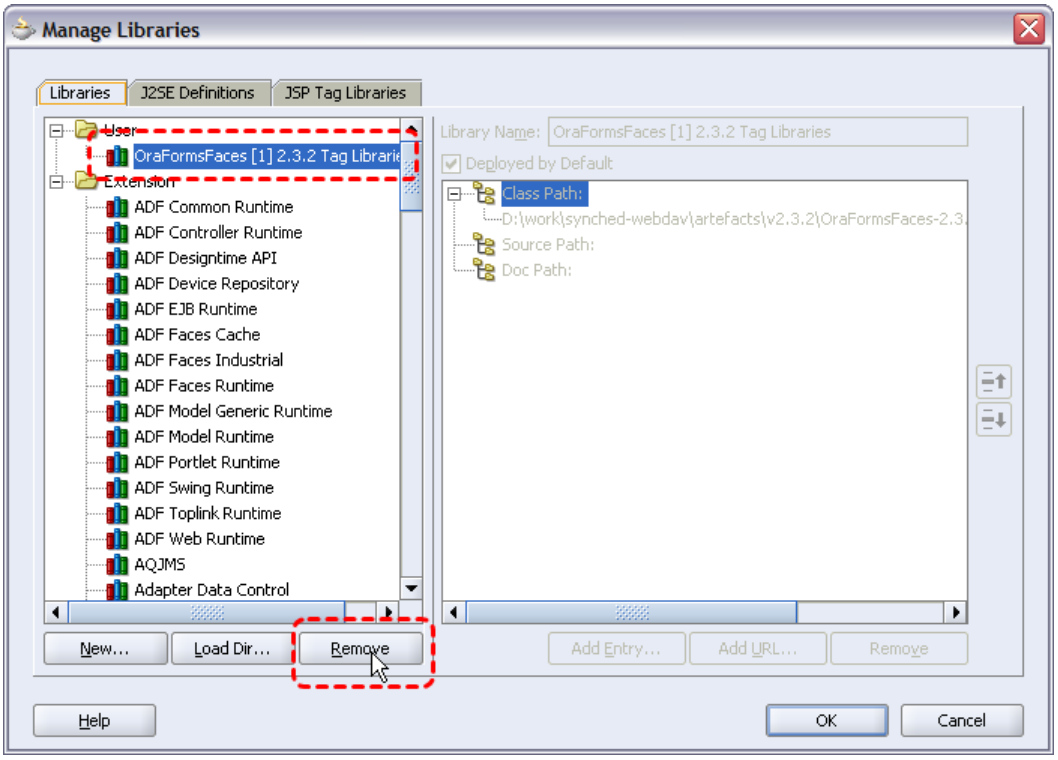
- If you used a ***technology preview of OraFormsFaces version 3.0.x***, you will need to uninstall this manually. If you are upgrading from OraFormsFaces 2.x you can safely skip this step.
  - Shutdown JDeveloper so we can safely remove any old version of OraFormsFaces.
  - Go to the directory `JDEV_HOME/jdev/extensions`
  - Delete any JAR file that starts with `com.commit_consulting.oraformsfaces`.

- Delete any subdirectory that starts with `com.commit_consulting.oraformsfaces`.

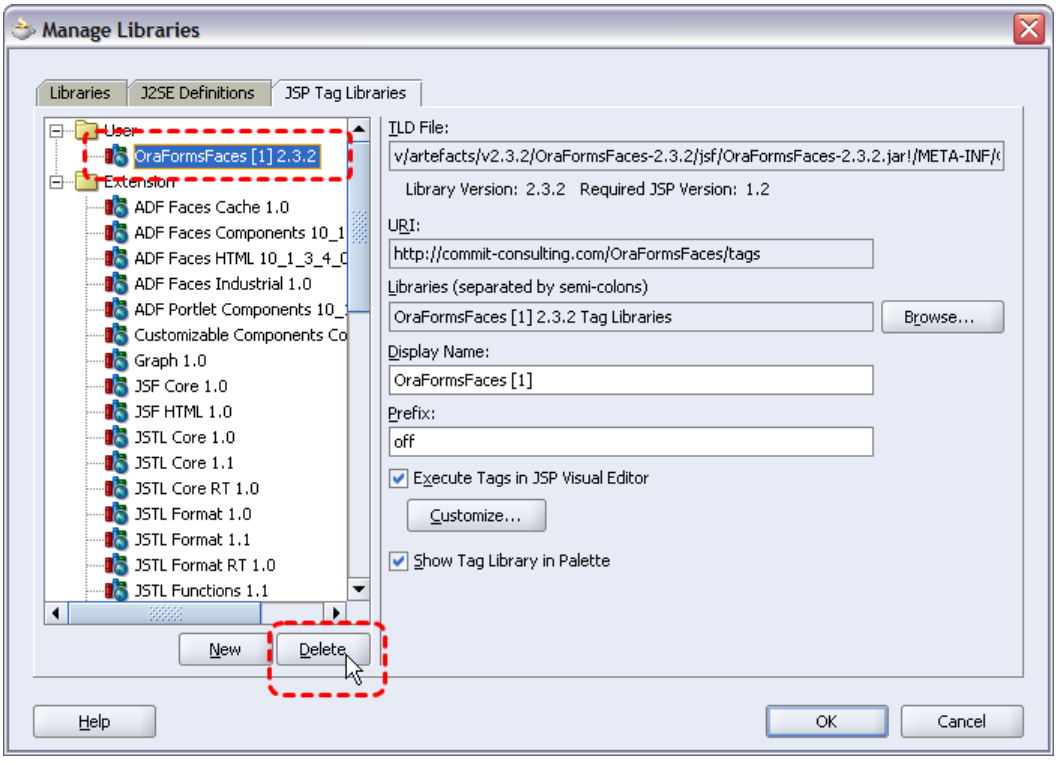


- If you previously installed ***OraFormsFaces 2.x with JDeveloper 10.1.3***, you need to revert some changes you made during the previous OraFormsFaces installation. If you are upgrading from a different version of OraFormsFaces or are using JDeveloper 11.x you can safely skip this step.
  - Shutdown JDeveloper 10.1.3
  - If you were using OraFormsFaces 2.x you made changes to the `faces_creator_configuration.xml` configuration file during installation. These changes are no longer necessary and can be reverted. To do so, use a text editor to open the `faces_creator_configuration.xml` file located in the `JDEV_HOME/jdev/system/oracle.adfm.dt.faces.xx.xx.xx.xx.xx/`directory.
  - Remove the following line from the prefixMappings at the top of the file:
    ```
    <prefixMapping prefix="off" namespace="http://commit-
    consulting.com/OraFormsFaces/tags" />
    ```
  - Further down the file, remove all the `<creator>` elements for OraFormsFaces. If you simply copied the example shipped with previous versions of OraFormsFaces, this entire section should start with the following comments:
    ```
    <!--
    ## OraFormsFaces CREATORS ################################
    ## Mind the prefixMapping at the top of this document ##
    -->
    ```
    Be sure to leave the closing `</creatorInfos>` and `</creatorConfiguration>` tags at the end of the file.

- If you previously used ***OraFormsFaces version 2.x in either JDeveloper 10.1.3 or 11.x*** you manually registered a Java library and JSP tag library. These can now be removed:
  - Start any JDeveloper version you used with OraFormsFaces 2.x and go to **Tools > Manage Libraries**
  - Go to the **JSP Tag Libraries** tab and remove any OraFormsFaces libraries under the **User** folder by selecting them and clicking the **Remove** button.



  - Go to the **Libraries** tab. Remove any OraFormsFaces libraries under the **User** folder by selecting it and clicking the **Remove** button.

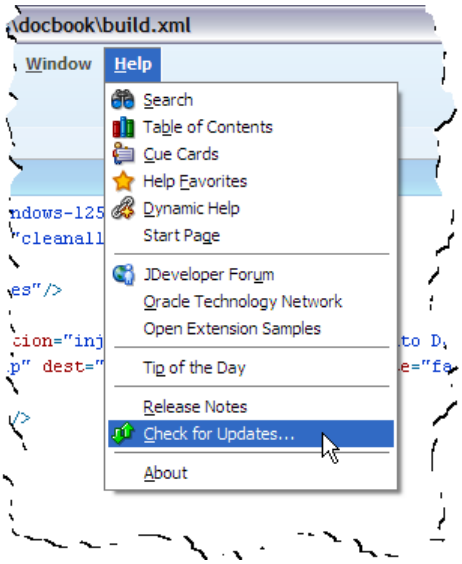


  - Close the dialog by pressing **OK**

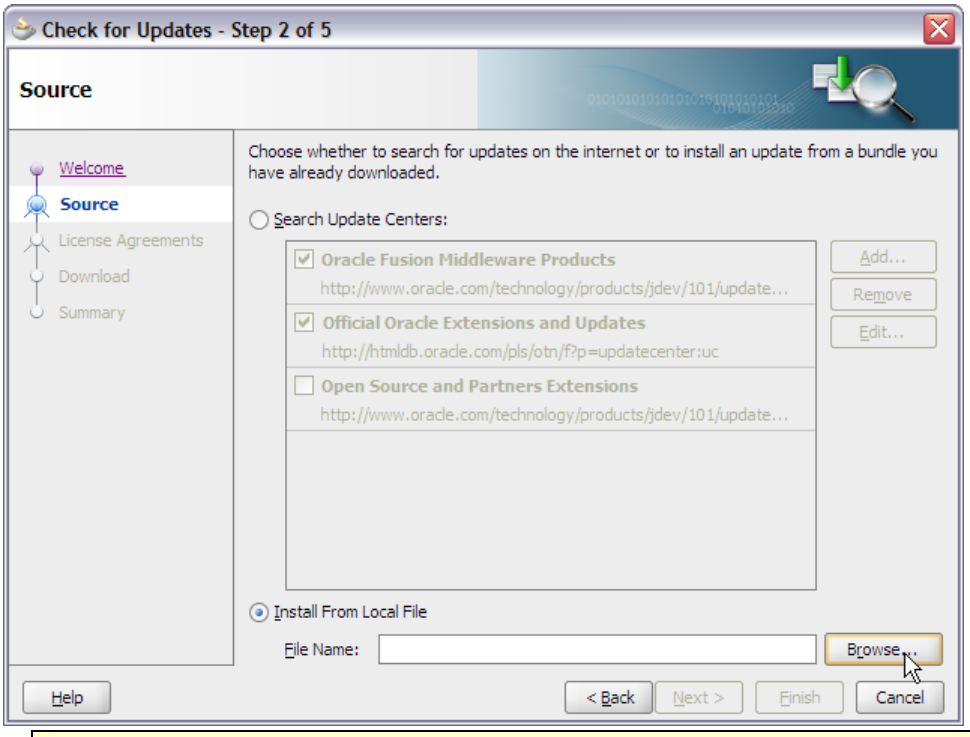
3.4.2. Installing JDeveloper Extension

This section describes how to install OraFormsFaces as a JDeveloper extension. These steps need to be performed only once for a JDeveloper installation and do not need to be repeated for each project

- If you are upgrading from a previous version of OraFormsFaces, be sure to follow the steps in the previous section to remove any old version before continuing with installation.
- Start JDeveloper
- Go to **Help > Check for Updates**

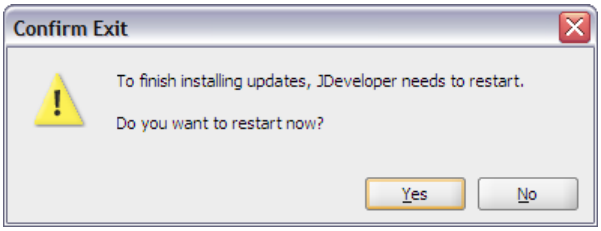


- Check the radio button Install from Local File. Click the Browse… button and locate the OraFormsFaces ZIP file you downloaded from http://www.commit-consulting.com/oraformsfaces/



Note: Be sure to use ZIP file corresponding to your JDeveloper version. There are different ZIP files for JDeveloper 10.1.3 and JDeveloper 11.1.1

- Finish the wizard to install the JDeveloper extension and restart JDeveloper



By installing the JDeveloper extension, the OraFormsFaces JAR file has been registered as a Java library and the JSP tag library it contains has also been registered. This allows you to add the OraFormsFaces library and JSP tags to any of your projects. It also hooked into JDeveloper so that attributes and methods from ADF Data Controls can be dragged and dropped onto a JSF page as OraFormsFaces `FormParameter` or `FormComponent` components while automatically registering the appropriate bindings in your ADF page definition files. More on this later

If you are using another IDE than Oracle JDeveloper, you will have to register the Java library and JSP tag library manually in your IDE of choice. You will lose the ability to drag-and-drop ADF data controls since other IDEs do not support Oracle ADF development.

3.5. Manually Configuring Oracle Forms

Since OraFormsFaces integrates JSF with Oracle Forms, there is also a substantial part of OraFormsFaces that is not implemented at the JSF server but rather at the Oracle Forms server. This requires some installation and configuration steps. These steps apply to both local Oracle Developer Suite installations as well as Oracle Application Server or Fusion Middleware installations. These instructions apply to both Oracle Forms 10gR2 (10.1.2) and 11gR1 (11.1.1).

The easiest way to configure Oracle Forms is to install the JDeveloper extension as described in ***3.3.1 Installing JDeveloper Extension*** and then running the installation wizard as described in ***3.3.2 Running Installation Wizard***. The installation wizard is only available with JDeveloper 11 so even if you plan on only using OraFormsFaces with JDeveloper 10.1.3 you will have to install JDeveloper 11 and the OraFormsFaces extension at least once.

As an alternative to the installation wizard you can also perform the same installation steps manually. These manual steps are described below.

3.5.1. Configuring Forms Server

These are the steps to install OraFormsFaces files to your Oracle Forms (Server) installation and make appropriate changes to the configuration files.

- In the previous section you installed the OraFormsFaces JDeveloper extension which also unpacked the distribution ZIP file into `JDEV_HOME/jdev/extensions/com.commit_consulting.oraformsfaces.v????.?.?.?`. Navigate to that directory and then navigate to the subdirectory appropriate for your Oracle Forms version: `forms1012` for Forms 10gR2(10.1.2) or `forms1111` for Forms 11gR1 (11.1.1).
- The OraFormsFaces extension JAR contains extensions to the Forms applet and needs to be available for download by the clients. Copy the OraFormsFaces extension JAR from the distribution `java` subdirectory to the `java` directory of your Forms server at `ORACLE_HOME/forms/java`. This should be the same directory as where `frmall.jar` lives.

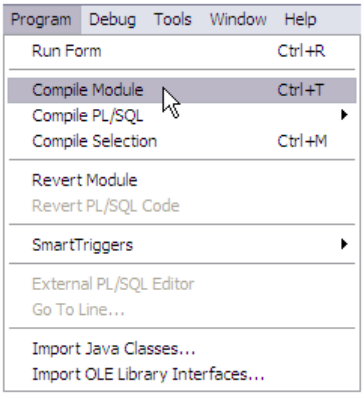
- Copy the two files (formsweb.cfg template and baseOraFormsFaces html file) from the distribution `server` subdirectory to the same directory where Oracle Forms keeps it configuration and base files. For Forms 10gR2 this is `ORACLE_HOME/forms/server`. Forms 11gR1 has a more complex directory structure. It keeps the formsweb.cfg file in `DOMAIN_HOME/config/fmwconfig/servers/WLS_FORMS/applications/formsapp_11.1.1/config` and the base html files in `ORACLE_INSTANCE/config/FormsComponent/forms/server/`.
- If you made customizations to the `base*.htm` files that ship with Oracle Forms or if you made customizations to the `base*` files from previous OraFormsFaces releases, you might want to review these and see if similar changes are required in the baseOraFormsFaces file for this release.
- The settings from the OraFormsFaces formsweb.cfg template file have to be merged with the `formsweb.cfg` file that Oracle Forms uses. To do so, open both files in a text editor and copy all content from the OraFormsFaces template file to the end of the Oracle Forms formsweb.cfg file.
- If you are ***upgrading*** from a previous OraFormsFaces version, your formsweb.cfg file will already contain an `[OraFormsFaces]` section. Since almost every version of OraFormsFaces introduces or changes certain parameters, you will need to take the version from the template and overwrite the existing `[OraFormsFaces]` section. Review and perhaps re-apply any changes you might have made to the parameters in the `[OraFormsFaces]` section in the past.
- You could also change the name of the `[OraFormsFaces]` section or even have multiple differently configured sections. Later on, when configuring your JSF/ADF web application you will need to refer to the appropriate name of the configuration section.
- The `userid` parameter is setup to use a fixed username, password and connect-string to connect to a database. This likely needs to be changed for your individual situation. Having fixed credentials in the `formsweb.cfg` file is the easiest to setup, but might not be a desired solution. Simply removing the `userid` parameter will prompt the user for a logon when first using Oracle Forms in a session. Since the user probably already logged on to the JSF/ADF web application, this can be confusing. As an alternative you could use Oracle Single Sign On or an OraFormsFaces solution where you determine the database credentials from a JSF managed bean. See the rest of this guide for more details.

3.5.2. Installing Forms and Libraries

OraFormsFaces comes with a number of PLL libraries, OLB Object Libraries and supporting FMB Forms. These need to be installed at the Forms Server but also in the Forms Builder development environment:

- In the previous section you installed the OraFormsFaces JDeveloper extension which also unpacked the distribution ZIP file into `JDEV_HOME/jdev/extensions/com.commit_consulting.oraformsfaces.v????.?.?.?`. Navigate to that directory and then navigate to the subdirectory appropriate for your Oracle Forms version: `forms1012` for Forms 10gR2(10.1.2) or `forms1111` for Forms 11gR1 (11.1.1).
- Copy the `oraformsfaces.olb` file to the directory where you normally place your FMB and/or OLB files. This directory has to be in the `FORMS_PATH` that is used when compiling Forms. For demonstration purposes you can use `ORACLE_HOME/forms` where Oracle Forms' own `test.fmb` and `test.fmx` also reside.
- Also copy the PL/SQL libraries (.pll files) and Forms (.fmb files) that come with OraFormsFaces to that same directory.

- Compile all FMB files to FMX files. You can do this with the OraFormsFaces supplied batch file (see the `jdapi` subdirectory) or do it directly from Forms Builder. To compile from Forms builder, open the FMB file in Forms Builder and for each module select **Compile Module** from the **Program** menu:



- To compile all the FMB files at once with the OraFormsFaces supplied batch file, open a Command/DOS box and issue the following commands:

```
set ORACLE_HOME=c:\oracle\product\10.1.2\ds (use your own Oracle home dir)
cd directory_with_fmb_files
c:\..JDEV_HOME..\jdev\extensions\com.commit_consulting.oraformsfaces....\jdapi\
compile.bat --forms "*.fmb"
```

- If you normally compile your PLL files to PLX files, you can do that as well for the OraFormsFaces PLL files. Remember that Forms will always use the PLX version of a library on the `FORMS_PATH` over the PLL version even if the PLX is in directory further down the lists of directories in the `FORMS_PATH`.
- If you normally not compile your PLL files to PLX files, open the OraFormsFaces PLL files in Forms Builder and select Program > Compile PL/SQL > All from the menu to recompile all PL/SQL code in the file. This might prevent issues when using a different Oracle Forms version that was used to originally construct the PLL files. Be sure to save the PLL file with your version of Forms Builder after completing this compilation.

> **Note:** Be sure to recompile **all** FMB and PLL files that ship with OraFormsFaces as we've found moving them between different platforms, Windows versions or processor architectures (32-bit vs. 64-bit) might require a recompile.

3.5.3. Setting up Forms Builder

These following steps are optional and only apply to Oracle Forms version ***prior to version 11gR1***. These prior Forms versions did not have a native JavaScript API. OraFormsFaces therefore uses a Pluggable Java Component (PJC) in all your forms to implement its own JavaScript API. If you do not include the OraFormsFaces JAR file in your Forms Builder, this could lead to warning messages in Forms Builder when you are using the Visual Editor. Forms Builder will try to access the OraFormsFaces PJC bean and requires the JAR file to do so. You can ignore this error or setup Forms Builder so it can find the appropriate JAR file. To setup Forms Builder follow these steps:

- Open the Windows Registry Editor, but selecting **Start > Run**, and typing `regedit`. Navigate to `HKEY_LOCAL_MACHINE`, then `SOFTWARE`, then `ORACLE`, then `KEY_xxxx` that represents your Developer Suite installation. In the right-hand pane, select the `FORMS_BUILDER_CLASSPATH` entry. Double click the entry to edit it.
- Add the full path to the OraFormsFaces extension JAR you installed in the previous section Configuring Forms Server. Be sure to separate this filename with a semi-colon (;) from the existing paths. An example path could be `c:\oracle\product\10.1.2\ds_1\forms\java\OraFormsFaces-extensions-3.0.3.jar`.

3.6. Accessing OraFormsFaces Documentation

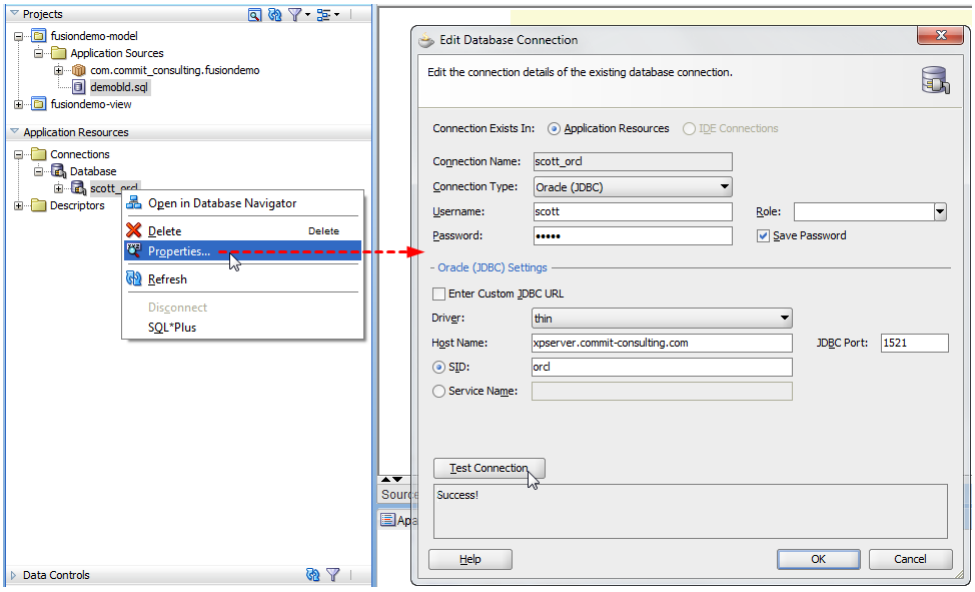
OraFormsFaces adds an entry to the **Help** menu in JDeveloper to open the OraFormsFaces Documentation Index. This gives access to documentation like the Release Notes and the Developer's Guide as well as the bundled JavaDoc and JSDoc documenting the supplied Java classes and JavaScript classes. It also contains links to open a demo application that shows some of the capabilities of OraFormsFaces in a simple ADF 11 application although most techniques also apply to ADF 10.1.3 or other JSF environments.

3.7. Accessing OraFormsFaces Fusion Demo Application

The OraFormsFaces extension for JDeveloper 11 comes with a simple ADF 11 Fusion application showing many of the OraFormsFaces features. Although this application requires JDeveloper 11 and ADF 11 most of the concepts also apply to ADF 10.1.3 or other JSF applications.

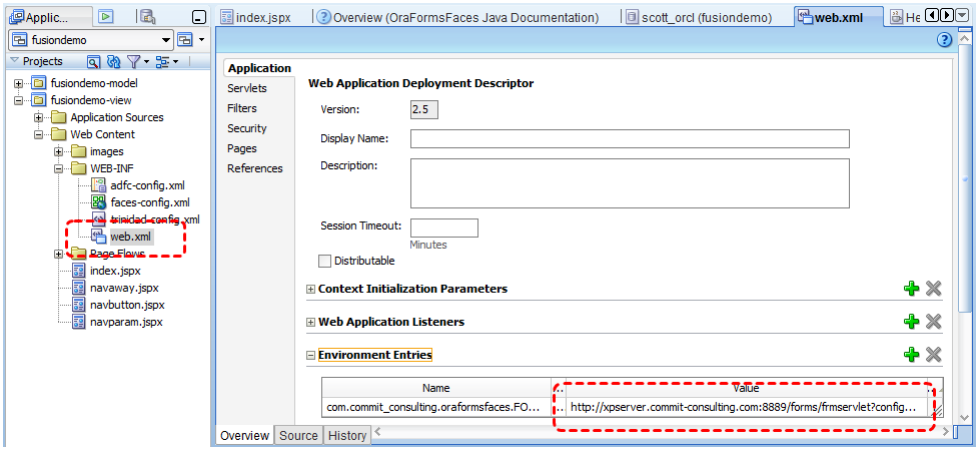
The Fusion Demo Application can be opened from the **Open OraFormsFaces Demo Application** in the **Help** menu of JDeveloper 11g. This will install the demo application in your default JDeveloper working directory and will automatically open its workspace. Be sure to configure the application for your specific environment by reviewing the following settings:

- Make sure you followed the installation instructions to configure your Forms server as described in ***3.3.2 Running Installation Wizard***.
- Run the `demobld.sql` script to create the necessary database objects. This file is included in the **Application Sources** folder of the **fusiondemo-model** project in JDeveloper.
- Configure the `scott_orcl` database connection in the **fusiondemo** Application Resources. It is easiest to keep the same name for the connection as this is used in other parts of the application. Just right-click the connection and select **Properties**. Change the username and database information to match the schema used for installing the demo database objects.



- Review the `formsweb.cfg` configuration file from your (possibly local) Forms server. After installing OraFormsFaces it should contain an `oraformsfaces` section. Ensure the `userid` parameter in this section contains the same credentials.

- The JSF application needs to know the URL of your Forms server. This is configured in the `web.xml` file of the **fusiondemo-view** project and needs to match your situation.



- This should be enough to run the demo application by right clicking the **fusiondemo-view** project and selecting **Run**. Navigate through the different tabs where each tab will show a different feature of OraFormsFaces with some explanation of what is happening. Inspect the source code of `index.jspx` in JDeveloper for more detailed or technical documentation.

3.8. Configuring a JDeveloper 10.1.3 Application

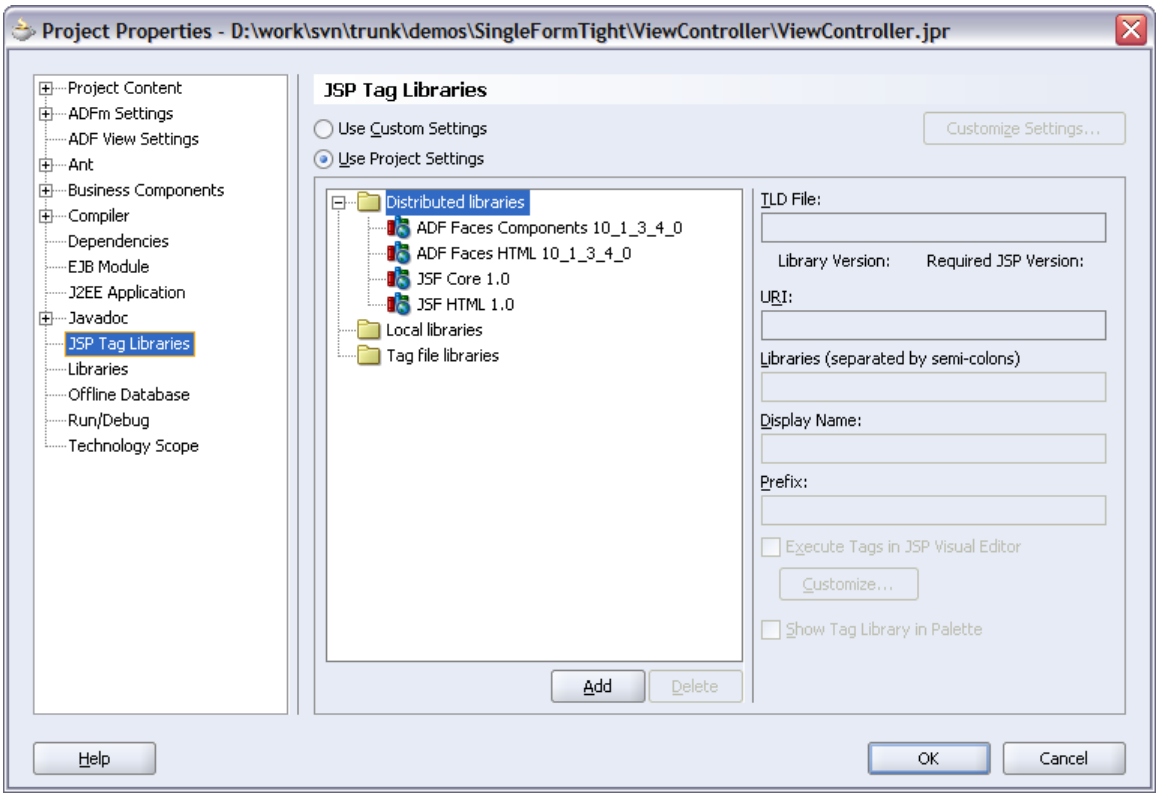
This section describes how to setup a JDeveloper version 10.1.3 workspace to use OraFormsFaces. You need to follow these steps to develop a JSF application that uses the OraFormsFaces components to embed Oracle Forms in your JSF application. You need to follow these steps only once for each of your JSF applications.

> **Note**: These instructions are specific for JDeveloper 10.1.3. If you are using JDeveloper 11.1.1 please skip to Configuring up a JDeveloper 11.1.1 Application below.

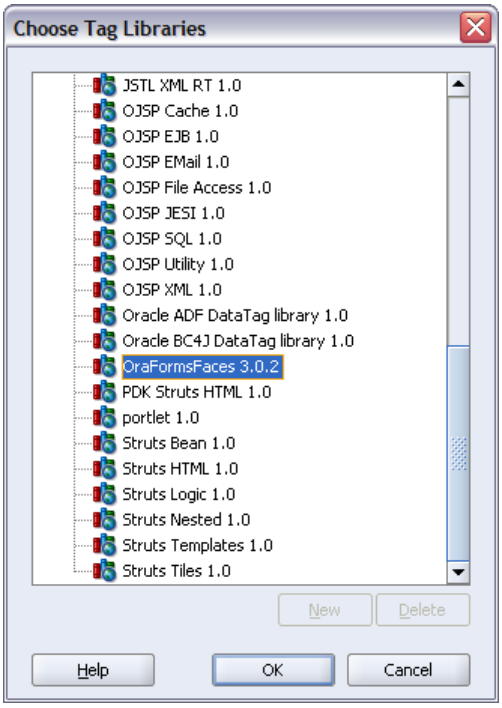
To setup your application for OraFormsFaces use the following steps:

- If you are enabling an existing JSF application to use OraFormsFaces skip these first few steps to create a new, empty application:
  - Right click the **Applications** node in the Applications Navigator and choose **New Application**.
  - In the **Create Application** dialog, select an appropriate Application Template. A Web Application template using JSF and ADF Business Components is most common for former Forms developers. Click **OK** to create a default workspace.
- Continue with the next steps whether you are setting up a new application or enabling OraFormsFaces for an existing application.

- Double click the **ViewController** project to open the project properties dialog and select the **JSP Tag Libraries** node:

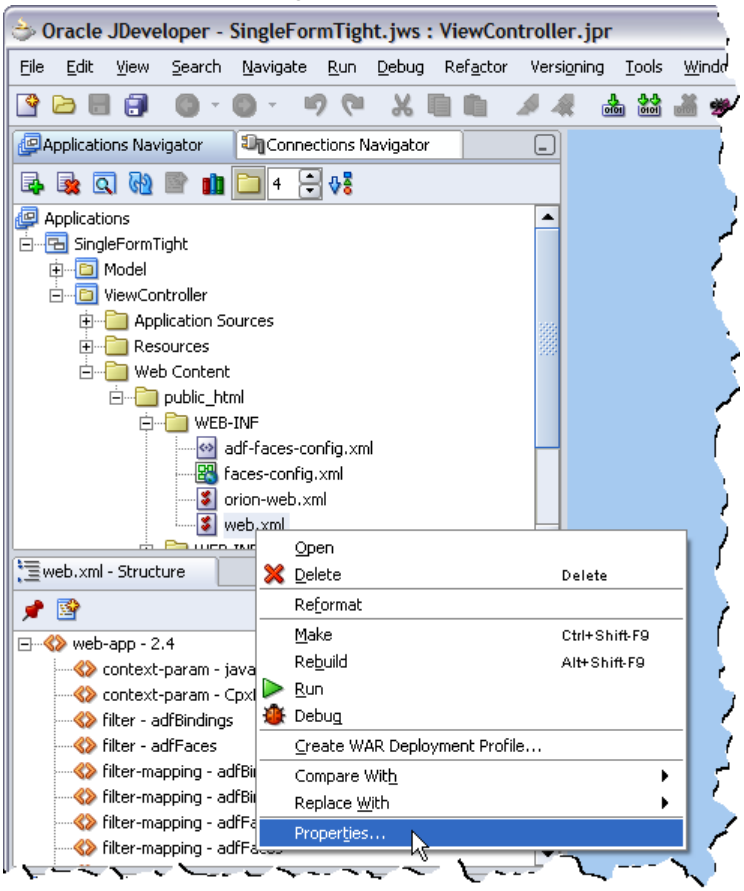


- Click the **Add** button to add a new JSP Tag Library to your application. Select **OraFormsFaces** from the available libraries and press **OK**. If the OraFormsFaces library is not listed ensure you installed the OraFormsFaces JDeveloper Extension.

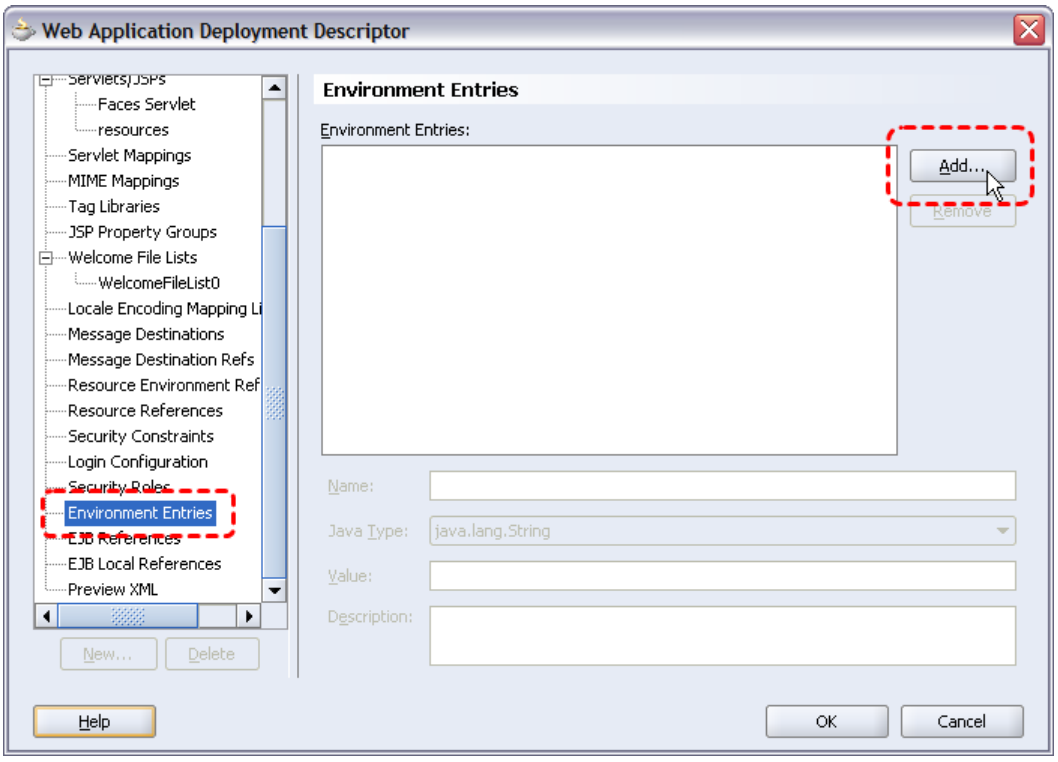


- Close all dialog boxes by pressing **OK**.

- Expand the **ViewController** project node then the **Web Content** node, then **WEB-INF**, and finally select the **web.xml** file. Right click and select **Properties**.



- Select the **Environment Entries** node near the bottom of the navigator and press **Add** at the right-hand side to add a new environment entry.



- Enter `com.commit_consulting.oraformsfaces.FORMS_SERVLET_URL` as the name of the parameter and select **java.lang.String** as the Java Type. Finally enter `http://`*`server.example.com:8888`*`/forms/frmservlet?config=OraFormsFaces` as the value of the parameter where the server name and port have been changed to match your particular situation. This URL should point to the URL to start Oracle Forms with the configuration section you have setup for OraFormsFaces. It is this URL that will be embedded in your JSF application to start the Oracle Forms applet. You can verify the correctness of the URL by requesting it from your web browser. It should return an OraFormsFaces JavaScript.

> **Note:** Remember that Environment Entries can be changed on or after deploying your application to the Java EE application server. This allows you to use the same application deployment file to deploy to development, staging, and production environments and have them use different Forms Servlet URLs for the appropriate environment.

- Close all dialogs by pressing **OK**

3.9. Configuring a JDeveloper 11.1.1 Application

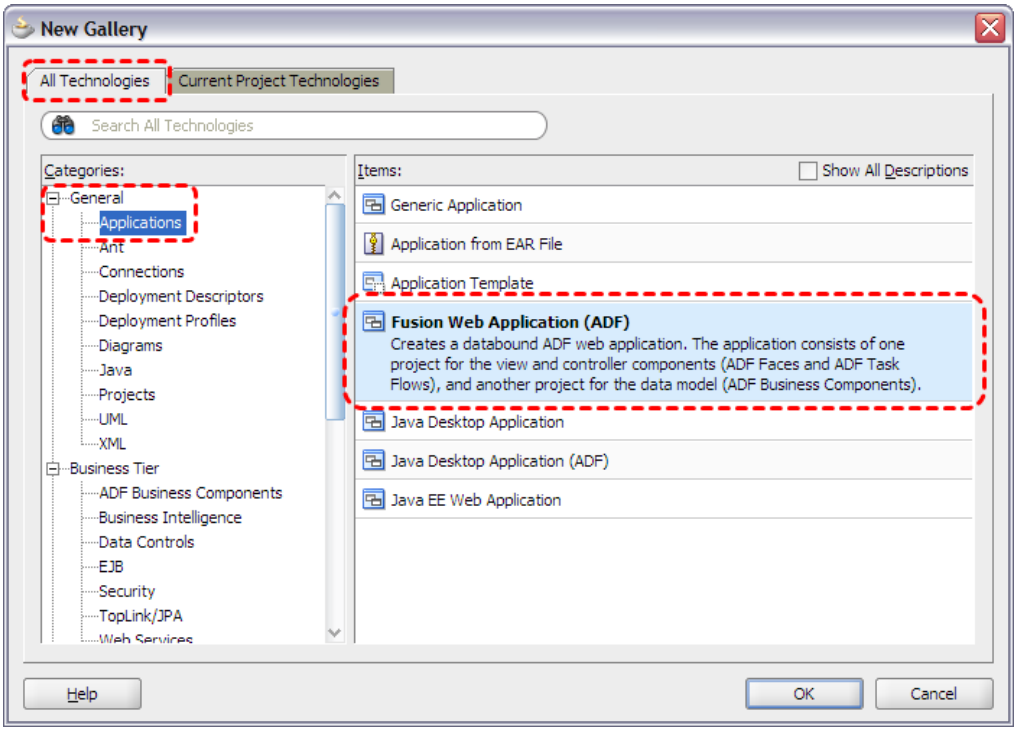
This section describes how to setup a JDeveloper version 11.1.1 workspace to use OraFormsFaces. You need to follow these steps to develop a JSF application that uses the OraFormsFaces components to embed Oracle Forms in your JSF application. You need to follow these steps only once for each of your JSF applications.

> **Note:** These instructions are specific for JDeveloper 11.1.1. If you are using JDeveloper 10.1.3 please skip to Configuring up a JDeveloper 10.1.3 Application above.

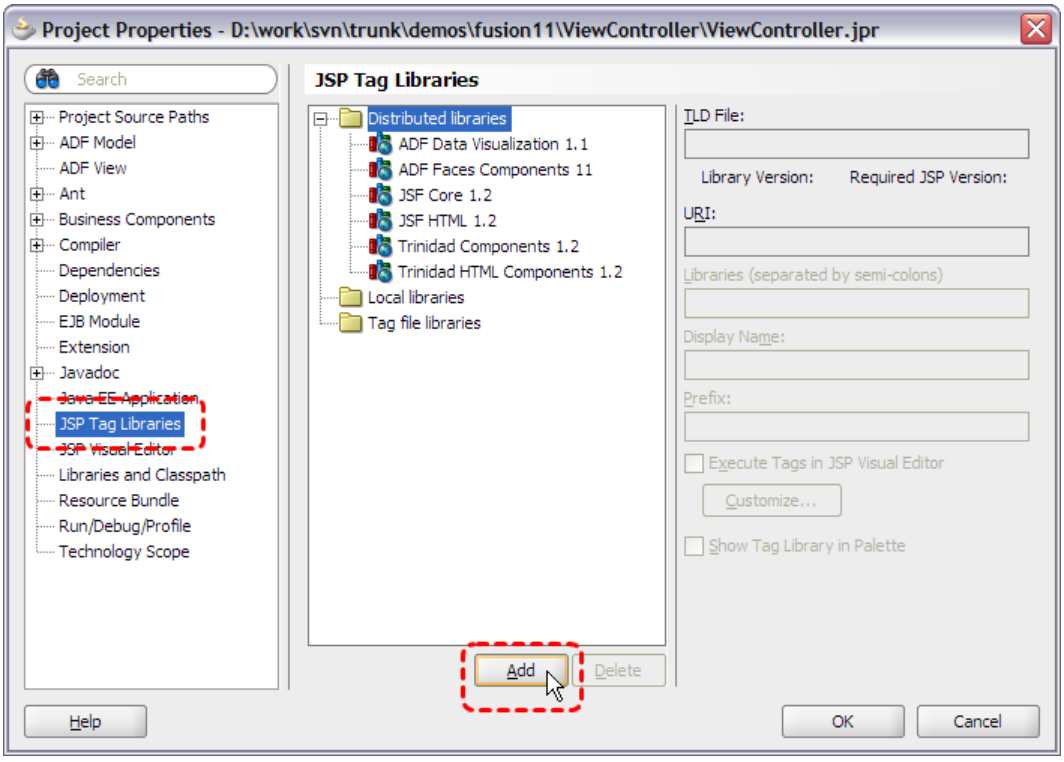
To setup your application for OraFormsFaces use the following steps:

- If you are enabling an existing JSF application to use OraFormsFaces skip these first few steps to create a new, empty application:
  - Select **New** from the **File** menu.

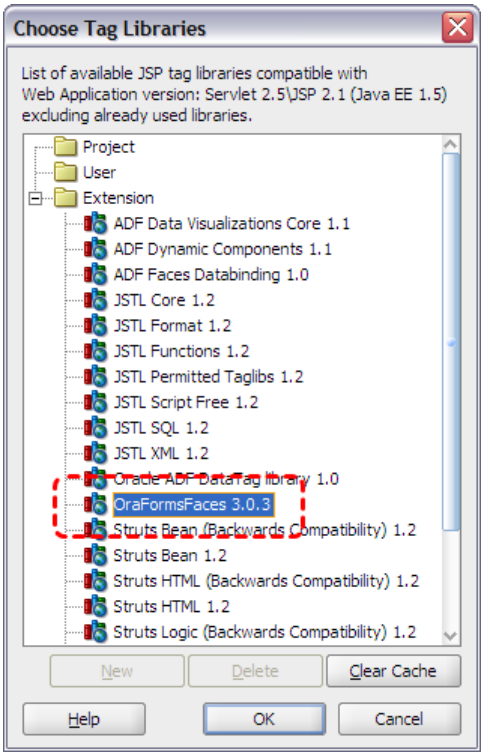
- In the **New Gallery**, select an appropriate application template from the **General > Applications** category. For most former Forms developers, the **Fusion Web Application (ADF)** is the most appropriate choice. Click **OK** to create a new application.



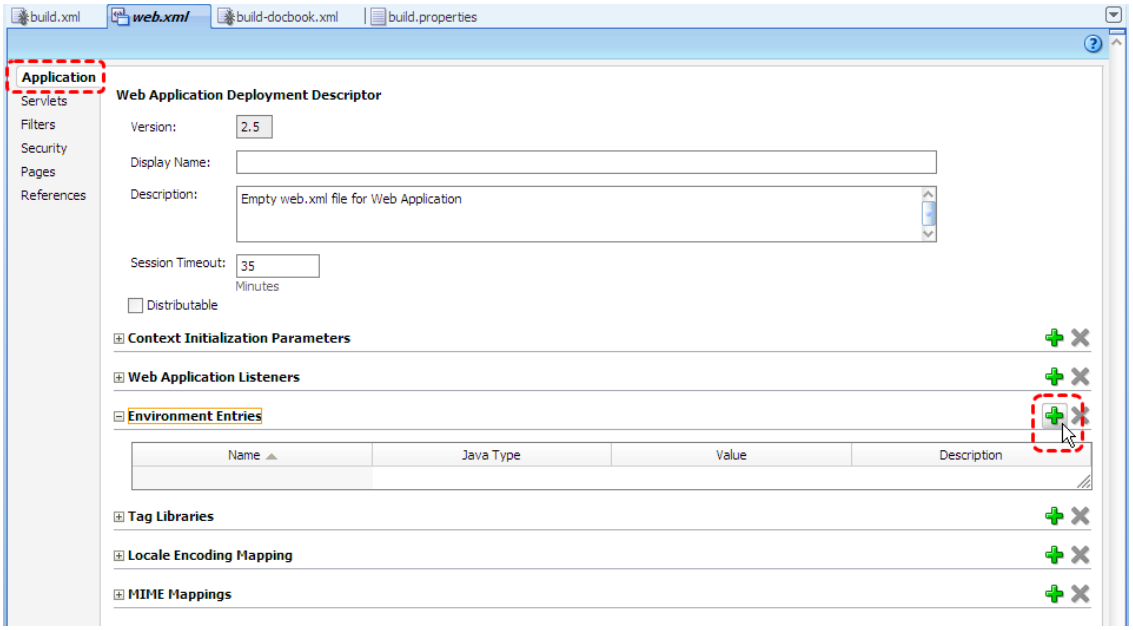
- Continue with the next steps whether your are setting up a new application or enabling OraFormsFaces for an existing application.
- Double click the **ViewController** project to open the project properties dialog and select the **JSP Tag Libraries** node.



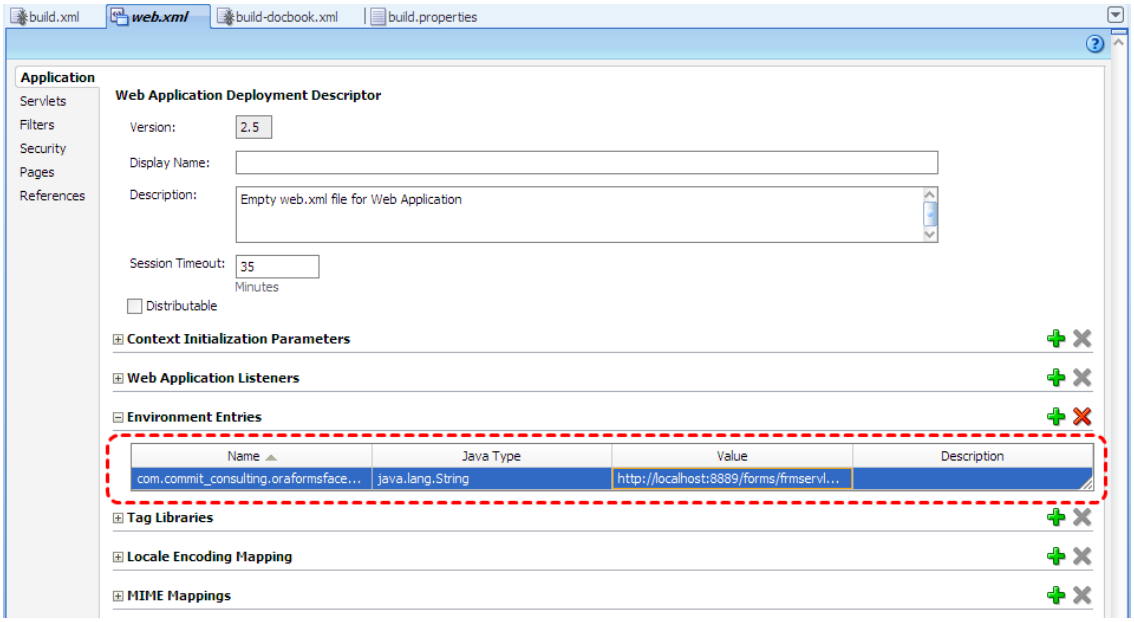
- Click the **Add** button to add a new JSP Tag Library to your application. Select **OraFormsFaces** from the available libraries and press **OK**. If the OraFormsFaces library is not listed ensure you installed the OraFormsFaces JDeveloper Extension.



- Close all dialog boxes by pressing **OK**
- Expand the **ViewController** project node then the **Web Content** node, then **WEB-INF**, and finally double click the **web.xml** file to open it.
- Click the **green plus** to add a new Environment Entry:



- Enter `com.commit_consulting.oraformsfaces.FORMS_SERVLET_URL` as the name of the parameter and select **java.lang.String** as the Java Type. Finally enter `http://`*`server.example.com:8888`*`/forms/frmservlet?config=OraFormsFaces` as the value of the parameter where the server name and port have been changed to match your particular situation. This URL should point to the URL to start Oracle Forms with the configuration section you have setup for OraFormsFaces. It is this URL that will be embedded in your JSF application to start the Oracle Forms applet. You can verify the correctness of the URL by requesting it from your web browser. It should return an OraFormsFaces JavaScript.



> **Note:** Remember that Environment Entries can be changed on or after deploying your application to the Java EE application server. This allows you to use the same application deployment file to deploy to development, staging, and production environments and have them use different Forms Servlet URLs for the appropriate environment

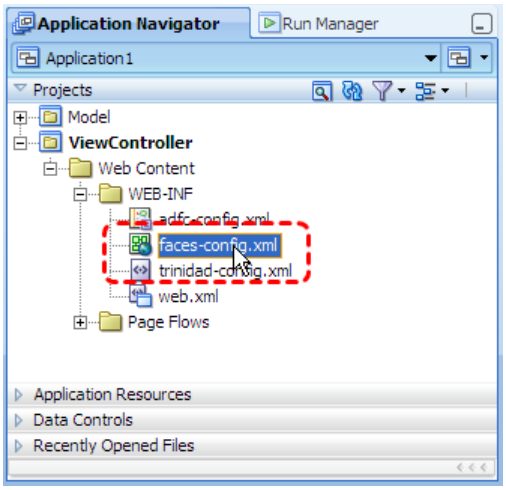
- Save all changes.

3.10. Verify Installation

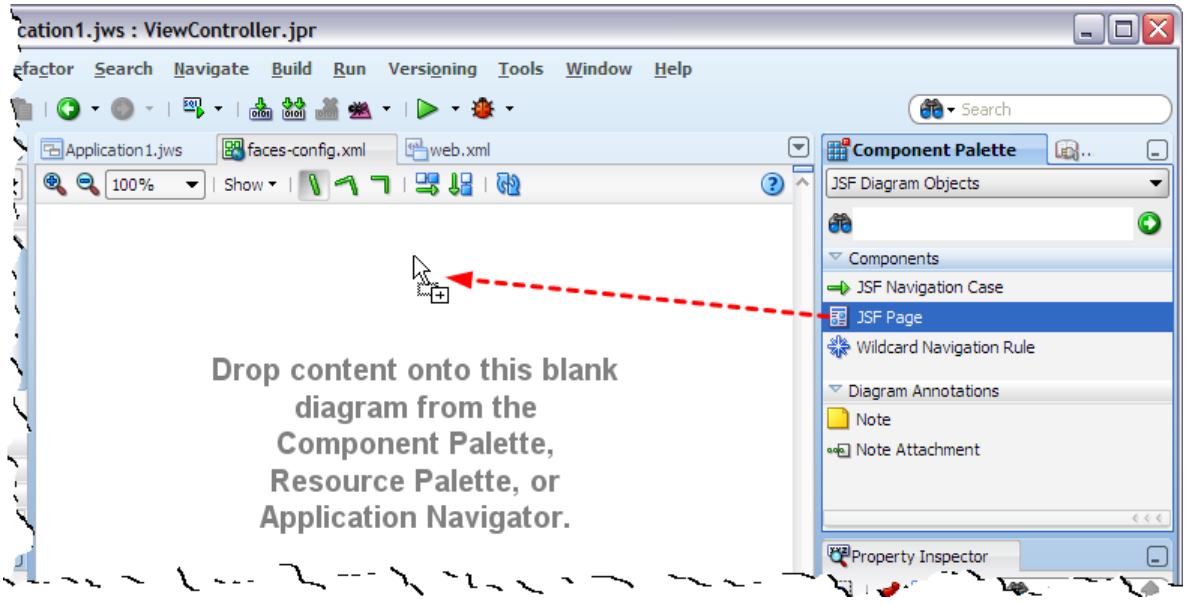
After installing OraFormsFaces, you can create a simple page that uses the OraFormsFaces components to validate your installation. These are the few simple steps to create a web page that embeds an Oracle Forms form. The steps and screenshots are for JDeveloper 11gR1, but the steps in JDeveloper 10.1.3 are basically the same.

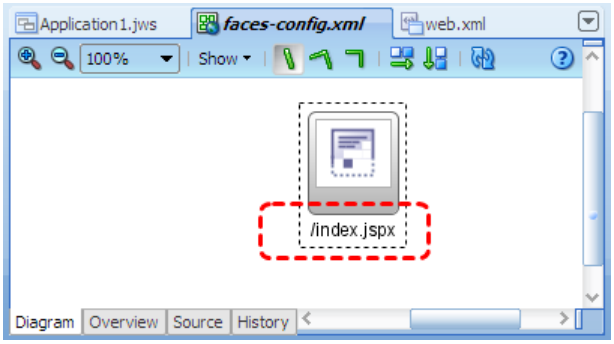
- Follow the instructions in previous sections to configure both Oracle Forms and Oracle JDeveloper.
- Follow the instructions in previous sections to create a new application that is setup for OraFormsFaces.

- In the **Application Navigator** navigate to **ViewController > Web Content > WEB-INF > faces-config.xml**. Double click the **faces-config.xml** file to open it. In this simple demonstration where using the JSF standard `faces-config.xml` and not the ADF controller `adfc-config.xml`. In your actual application you might consider using the ADF controller. See the JDeveloper documentation for more details.



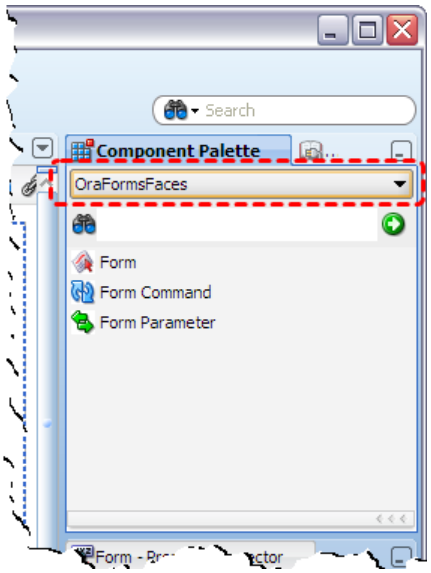
- Drag-and-drop a **JSF Page** from the right-hand top **Component Palette** to the faces-config.xml visual editor. Enter a name for the page, for example ***/index.jspx***



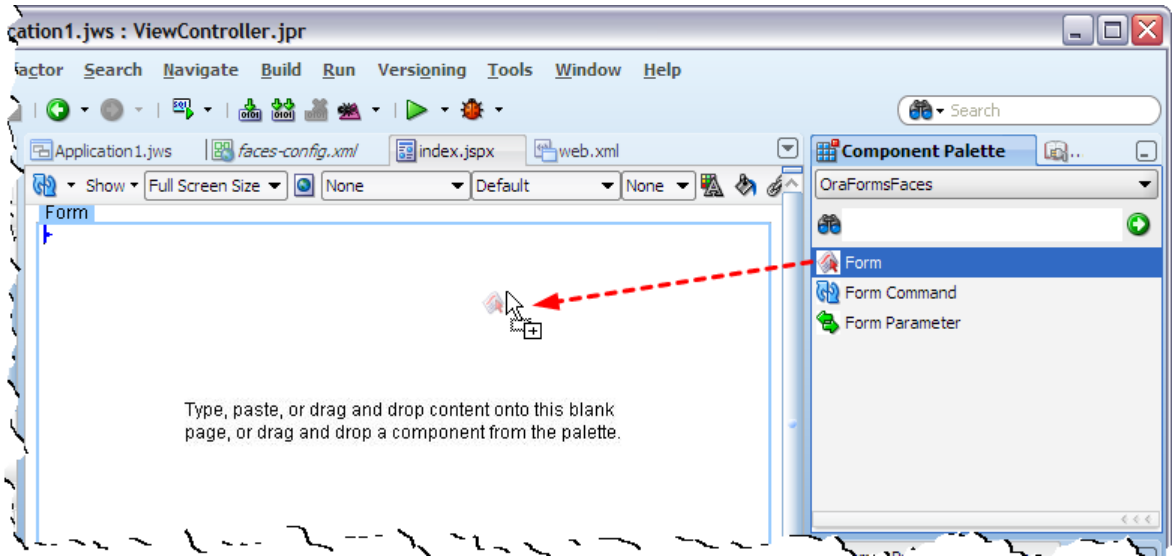


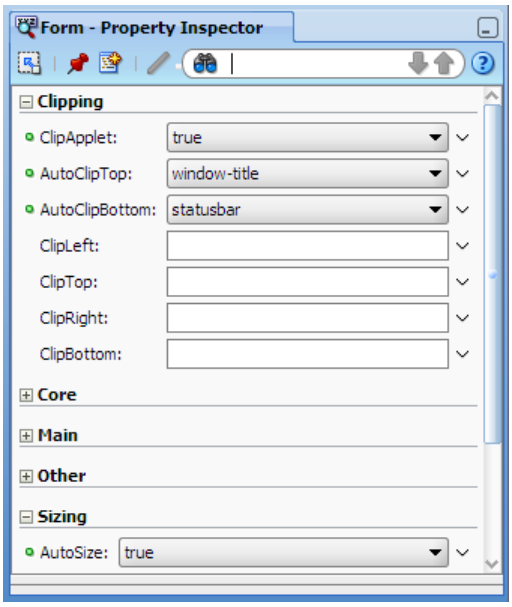
- **Double click** the page icon to open the **Create JSF Page** dialog. Accept all default and just press **OK** to open the new page in the visual editor.

- In the **Components palette** at the right-hand top select the **OraFormsFaces** family. If OraFormsFaces is not listed here, be sure you completed the steps at Configuring a JDeveloper 11.1.1 Application.



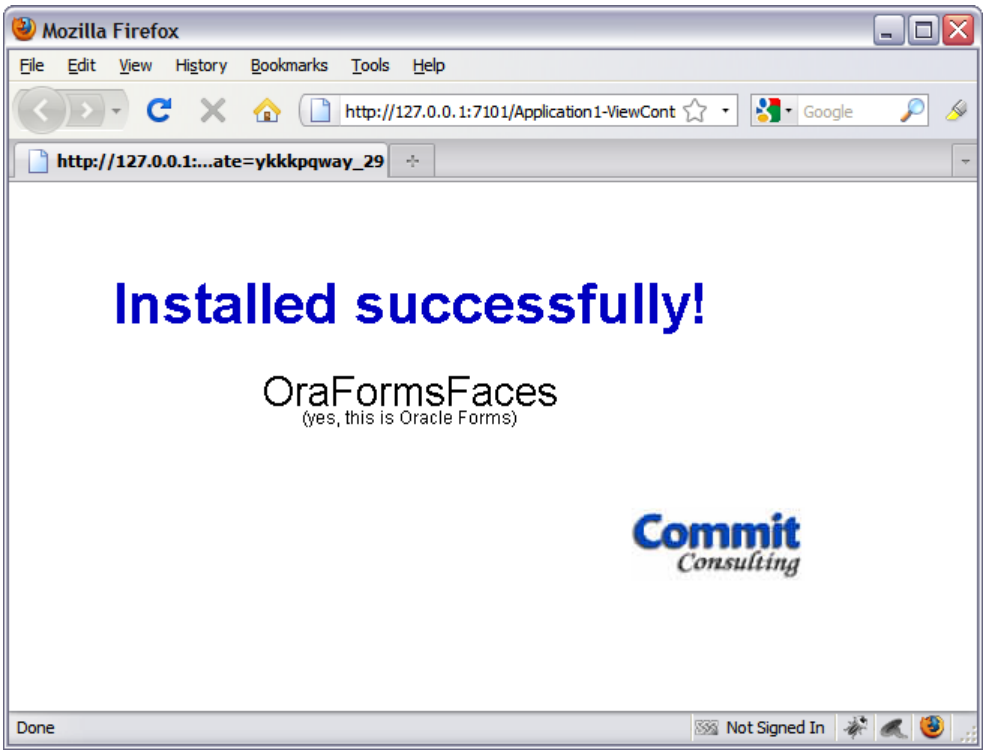
- Next you can drag-and-drop a **Form** component to your page. A dialog or property palette will appear to set the properties of the Form component:





- Be sure to set the following properties:
  - **ClipApplet** = `true` to enable clipping (hiding bits and pieces from the Forms applet)
  - **AutoClipTop** = `window-title` to let OraFormsFaces automatically calculate how much to clip from the top of the applet to hide the menu, toolbar and window title.
  - **AutoClipBottom** = `statusbar` to let OraFormsFaces automatically calculate how much to clip from the bottom of the applet to hide the status bar.
  - **AutoSize** = `true` to let OraFormsFaces automatically determine the appropriate size of the applet so it can display the running form.
- Ensure your Oracle Forms OC4J engine or Oracle Forms Weblogic Server is running

- Run the JSF page to see the embedded Forms applet in a JSF page:



- More advanced features like configuring clipping, parameters and JSF commands will be covered elsewhere in this manual

4. Embedding Oracle Forms in a JSF Page

The key feature of OraFormsFaces is the capability to embed an Oracle Forms application in a JSF page. This chapter will describe how to accomplish this. More advanced features are covered in subsequent chapters.

> **Caution:** Be sure to read and complete Getting Started with OraFormsFaces before taking the steps in this chapter.

4.1. Preparing the JDeveloper Workspace

The first thing you need is a JDeveloper Workspace to develop a JSF or ADF Faces application that is going to use OraFormsFaces. You can either start with a brand new workspace or take an existing JSF/ADF Faces workspace and enable OraFormsFaces. Instructions for these scenarios can be found in the previous sections on how to setup a workspace for JDeveloper 10.1.3 or JDeveloper 11.1.1.

4.2. Preparing the Oracle Forms Modules

Each Form Module (FMB) that is going to be used in an OraFormsFaces enabled web application needs the OraFormsFaces PLL libraries attached. Each Form also needs a certain trigger or Pluggable Java Component (PJC) depending on your Forms version.

OraFormsFaces uses a JavaScript API to enable communication between the web application and the Oracle Forms applet. In Oracle Forms versions prior to Forms version 11, this requires the use of a Pluggable Java Component (PJC) in each and every Form module. This PJC is capable of receiving JavaScript events from the web page and pass them to a PL/SQL trigger in your Oracle Forms module. It also enables the PL/SQL code in your Form to execute JavaScript to enable communication the other way around.

The required PJC is a part of the OraFormsFaces component library and is included on the OLB (Object Library) file. You will need to subclass an object group from this object library to each and every form you are going to use in an OraFormsFaces enabled application.

> **Note:** Oracle Forms version 11gR1 introduced a native JavaScript API. OraFormsFaces leverages this native API but still requires the OraFormsFaces Object Group to be subclassed into each of your Forms. The difference with Forms 11gR1 is that the OraFormsFaces object group no longer contains the former PJC but the new `WHEN-CUSTOM-JAVASCRIPT-EVENT` trigger.
>
> **Note:** OraFormsFaces uses the same object group name and object library file in Forms 11gR1 as in earlier Forms versions. The content of the object group differs per Oracle Forms version. This should make upgrading to Oracle Forms 11gR1 a lot easier. You can replace the `oraformsfaces.olb` with the Forms 11gR1 version. You would only need to recompile all your Forms to pick up on the changes in the object group.

You could manually open all your Forms in Forms Builder and subclass the OraFormsFaces object group from the `oraformsfaces.olb` file. However, a much more productive approach is to use the convenient batch file shipped with OraFormsFaces. To use it follow the next simple steps:

- Open a DOS box or Command Prompt. On Windows this can be done by selecting **Start** (or Vista button) **> All Programs > Accessories > Command Prompt**

- Change the current directory to the `jdapi` directory that was in the OraFormsFaces ZIP file that is extracted as a JDeveloper extension. For example:

```
cd \jdev1111\jdev\extensions
cd com.commit_consulting.oraformsfaces.v1111.3.0.3
cd jdapi
```

- Set an environment variable with the name `ORACLE_HOME` to the Oracle home directory of your Developer Suite or Application Server installation. For example:

```
set ORACLE_HOME=c:\oracle\product\10.1.2\ds_1
```

- Now run the OraFormsFaces batch file to subclass the OraFormsFaces object group to your FMB file(s). This batch file requires three arguments. The first one (`forms`) specifies which FMB file(s) to process. This can contain wildcards so you can process all FMB files in a single directory in one go. If you do use wildcards be sure to enclose this argument with double quotes. The second argument (`olb`) specifies from which OLB file to subclass. The final argument (`objectgroups`) specifies which object group(s) to subclass. If you want to subclass multiple object groups in one go, you can just specify multiple names separated by spaces. For example:

```
objectgroup --forms "w:\*.fmb" --olb w:\oraformsfaces.olb --objectgroups oraformsfaces
Processing w:\test.fmb
..subclassing object group ORAFORMSFACES
..saving form
Processing w:\testwebutil.fmb
..subclassing object group ORAFORMSFACES
..saving form
Processed 2 files
```

> **Note:** The tool will fail with an exception if a required PLL library could not be found. It will also fail if the source OLB or FMB file for a subclassed object could not be found. In this situation be sure to set the FORMS_PATH to an appropriate value in the Windows registry or as an environment key before starting the tool.

- Now run the OraFormsFaces batch file to attach the OraFormsFaces PL/SQL libraries to your FMB file(s). This batch file requires two arguments. The first one (`forms`) specifies which FMB file(s) to process. Again, this can contain wildcards so you can process all FMB files in a single directory in one go. If you do use wildcards be sure to enclose this argument with double quotes. The final argument (`libs`) specifies which PL/SQL library files (PLL's) to attach. If you want to subclass multiple libraries in one go, you can just specify multiple files separated by spaces. OraFormsFaces comes with a number of PLL files but the main library is `off_lib.pll`. Since the other libraries are attached to `off_lib.pll` itself you only need to attach `off_lib.pll`. For example:

```
attachlibs --forms "w:\*.fmb" --libs w:\off_lib.pll
Processing w:\test.fmb
..attaching library OFF_LIB
..saving form
Processing w:\testwebutil.fmb
..attaching library OFF_LIB
..saving form
Processed 2 files
```

> **Note:** The tool will fail with an exception if a required PLL library could not be found. It will also fail if the source OLB or FMB file for a subclassed object could not be found. In this situation be sure to set the FORMS_PATH to an appropriate value in the Windows registry or as an environment key before starting the tool.

- Finally, compile all the modified forms. Again, OraFormsFaces comes with a time saving batch file to all of this in one go. This batch file requires two arguments. The first one (`userid`) specifies the username, password, and connect string to connect to the database. The final argument (`forms`) specified which FMB file(s) to process. Again, this can contain wildcards so you can process all FMB files in a single directory in one go. If you do use wildcards be sure to enclose this argument with double quotes. For example:

```
compile --userid scott/tiger@orcl --forms "w:\*.fmb"
Connecting to the database...
Connected
Compiling w:\test.fmb
Compiling w:\testwebutil.fmb
Processed 2 files
```

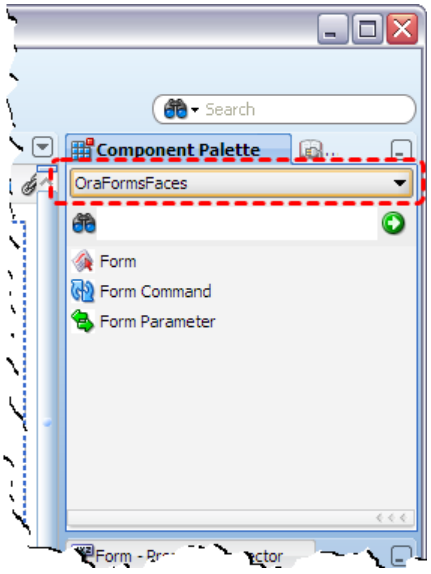
Please remember to make these changes to each and every form you are going to use with OraFormsFaces. You can continue to use these modified Forms in an environment not using OraFormsFaces. The Forms will just have an (invisible) Pluggable Java Component or redundant trigger which does not do anything when used outside of an OraFormsFaces environment. OraFormsFaces also comes with a PL/SQL function to detect if the form is running in an OraFormsFaces enabled application. You can use this in cases where you want your custom PL/SQL code to behave differently when running in OraFormsFaces or in legacy Oracle Forms.

If you are not on a Windows platform, you can create shell scripts for your own platform with the same functionality as the OraFormsFaces batch files. The actual work is done by cross platform Java code. The batch files/shell scripts are just wrappers to make the execution of these Java classes simpler.

4.3. Embedding Oracle Forms in a JSF page

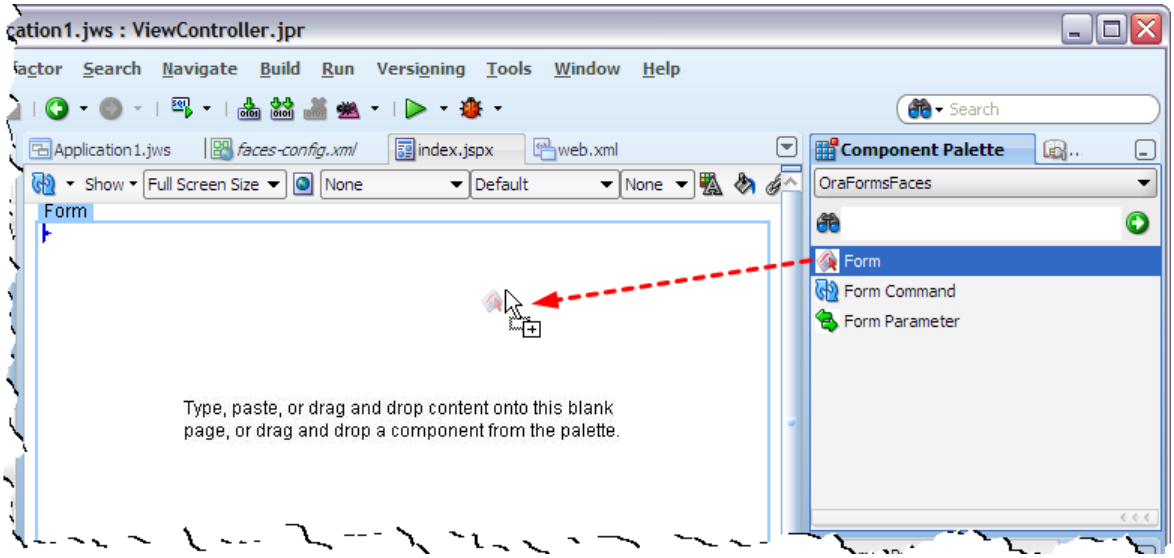
Before you start, ensure you have a JDeveloper project that is prepared for the use of the OraFormsFaces component library and the Oracle Forms modules you want to embed are prepared for OraFormsFaces. Previous sections describe how to setup JDeveloper and how to prepare your Forms modules. Once these preparations are complete, use the following steps to embed a Form in your JSF page. These instructions are for JDeveloper 11gR1 but the steps for JDeveloper 10.1.3 are virtually the same.

- Open an existing JSF page, page fragment or template if you want to embed Oracle Forms in an existing page or create a new JSF page, page fragment or template. Make sure the visual editor is showing.
- In the **Components** palette at the right-hand top select the **OraFormsFaces** family. If OraFormsFaces is not listed here, be sure you completed the installation of the JDeveloper extension:



Caution: If you have just added the OraFormsFaces tag libraries to your application, you might have to restart JDeveloper before the component family shows up or when the property dialogs are not working.

- Next you can drag-and-drop a **Form** component to you page. Depending on your JDeveloper version a dialog or property palette will appear where you can specify the properties of the OraFormsFaces Form component. If you are using JDeveloper 10g a dialog appears that does not show some of the advanced properties. You can access all properties after closing the dialog and selecting the Form component in the visual editor.



- The working of the properties is explained below:
  - Clipping properties
    - **ClipApplet** – This property indicates if you want to visually clip the Forms applet to show only a portion of the default Oracle Forms application. The default value is false. See Visual Integration for a full description of this feature.
    - **AutoClipTop** – When clipping is enabled (**ClipApplet** is set to `true`), this property specifies which chrome to remove from the top of the Forms applet. At runtime, the OraFormsFaces enhanced Oracle Forms applet will figure out how many pixels to clip from the top of the applet to just remove these elements. The default value is **none** which doesn't clip anything from the top. Other possible values are **menu**, **toolbar**, and **window-title**. Setting the property to **menu** will remove the menu bar from the top of the applet no longer allowing users to select options from the menu. This makes the applet look and feel more like a true web application. Navigation to other pages and forms should now be handled by the JSF application. Setting the property to **toolbar** removes both the menu and the toolbar/button bar to have the Forms applet blend in even more with the surrounding web page. The **window-title** is the most extreme value. It will not only remove the menu and toolbar but will also remove the title bar at the top of the Forms window.

- **AutoClipBottom** – When clipping is enabled (**ClipApplet** is set to `true`), this property specifies if the status bar should be removed from the bottom of the Forms applet. The default value is **none** which will not remove the status bar. Setting it to **statusbar** will remove the status bar. If the status bar is clipped and the **AutoClipTop** property is set to `window-title` and **AutoSize** is set to `true` it will even cause all window edges to be clipped to have the Forms applet fully blend with the web page. Remember that removing the status bar might also hide messages from the user that would normally appear in this status bar. You should take care that these messages remain noticeable for end users, for example by pushing them to JavaScript alerts as described in ***Error and Message Handling with Applet Clipping*** in the Visual Integration chapter.
- **ClipLeft**, **ClipTop**, **ClipRight** and **ClipBottom** – These properties specify the number of pixels to clip from the applet at each side. When using the auto-clipping feature the values from these properties are added to the automatically determined values by the auto-clipping feature. This means you can specify positive or negative values to increase/decrease the automatically determined clipping. When not using the auto-clipping feature these properties simply specify the number of pixels to clip from each side. See Visual Integration later in the guide for a full description of the clipping feature

- Core properties
  - **Id** – This is the unique ID of the JSF component within this JSF page. If you leave this property empty at design time, JSF will assign a system generated ID to the component at runtime. If you later add or remove components from the page, the generated ID can change. This is why it is best to set the ID to a fixed value at design time. You will be referring to this value from PL/SQL code and JavaScript with the more advanced integration. If you leave the ID generation to the JSF framework, the ID might change when you add or remove other components from the page, causing problems in your PL/SQL code.
  - **Rendered** – This is a default JSF property which indicates if the component should be rendered (included) in the HTML response to the client. You can set this false, to disable the component or use an Expression Language (EL) expression to render the component conditionally. For most usages, you can leave it to its default value of `true`.
- Main properties
  - **FormModuleName** – This is the name of the Forms module you want to run. You can specify the name of the Oracle Forms module with or without the `.fmx` extension. Normally you specify the name without a full directory path, since the Forms server will be setup to search in the directories where you put your FMX files. If you do not specify a value for **FormModuleName**, the default `off_test.fmx` will be used which shows the test form included with OraFormsFaces.
  - **ShowDevControls** – This property indicates if you want Development Controls at runtime. These controls allow you to change the sizing and clipping properties of the Form component at runtime in your web browser. It is not possible to show the Oracle Forms form in the JDeveloper visual editor. This is why these Development Controls have been added to change the sizing properties at runtime. Once you found appropriate values, you can set these at design time in the JSF page and disable the Development Controls. The default value is `false`. See the chapter on Visual Integration for a more detailed description and examples of the Development Controls.

- **LoadingImage** – While the applet is starting it is covered with an animated image. This property specifies which image to show. You can specify a URL to an image to use for this feature or use one of the preset images that are included with OraFormsFaces. Set this property to ***presetX*** where ***X*** is a number from 1 through 5. The five preset image are:

> **Note:** If you do not want to use any of the preset images and do not have a custom image to use, there are plenty of website that can generate a custom animated GIF on the fly. Some examples are http://www.ajaxload.info/, http://www.loadinfo.net/, and http://preloaders.net/.

- **UniqueAppletKey** – Normally OraFormsFaces will try to use a single applet instance for all pages of your application during a single session. However, by specifying a unique applet key to can enforce that a page does not reuse the same applet as another page. Only OraFormsFaces Form components with the same UniqueAppletKey will be allowed to reuse the same applet instance. See ***13 Forms Java Applet Instance Reuse*** for more information.

- Other properties
  - **Binding** – This is a default JSF property which binds the OraFormsFaces Form component to a backing bean property. It will instruct the JSF framework to call the setter of the backing bean property to pass a reference to the OraFormsFaces Form component. This can be used for programmatic access to the Form component at runtime. For most usages, this property can be left to its default (empty) value which disables binding to a backing bean.
- Sizing properties
  - **AutoSize** – This enables or disabled the auto-sizing feature of the OraFormsFaces applet. By default this is set to `false` (disabled) so applications upgraded from previous OraFormsFaces without this feature work the same. If you set **AutoSize** to `true` en thus enable this feature, the OraFormsFaces enhancements to the Oracle Forms applet will figure out the sizing of the window in your Forms module at runtime and will size the applet in the web page accordingly. Each time another form is opened in the same applet, the applet will resize itself according to the size of the current form.
  - **Width** – This is the width of the component in pixels. When the auto-sizing feature is enabled this specifies the initial size of the applet which is resized once the applet finished initialization and it knows the size of the Form it is running. When no specific value is set, the default of 640 will be used.
  - **Height** – This is the height of the component in pixels. When the auto-sizing feature is enabled this specifies the initial size of the applet which is resized once the applet finished initialization and it knows the size of the Form it is running. When no specific value is set, the default of 480 will be used.

It was as simple as that. It's just dragging-and-dropping a single OraFormsFaces Form component to your JSF page to embed an existing Oracle Form in that page. The following sections explain how to pass parameters and events between JSF and Forms as well as some more advanced usages of OraFormsFaces.

5. Passing Parameters

The previous section described how to embed an Oracle Forms form in a JSF page using OraFormsFaces. This all works great, but it will probably start the requested form with no data selected and ready to insert a new record. You could add a `WHEN-NEW-FORM-INSTANCE` trigger to automatically query the appropriate record. However, you probably need to pass some parameters from the JSF application to Oracle Forms and the other way around.

The OraFormsFaces component library includes a component called a **FormParameter**. This component can be used to pass parameters from JSF to Oracle Forms, from Oracle Forms back to JSF or both combined in a single parameter.

5.1. Defining a FormParameter by Dragging an ADF Data Control

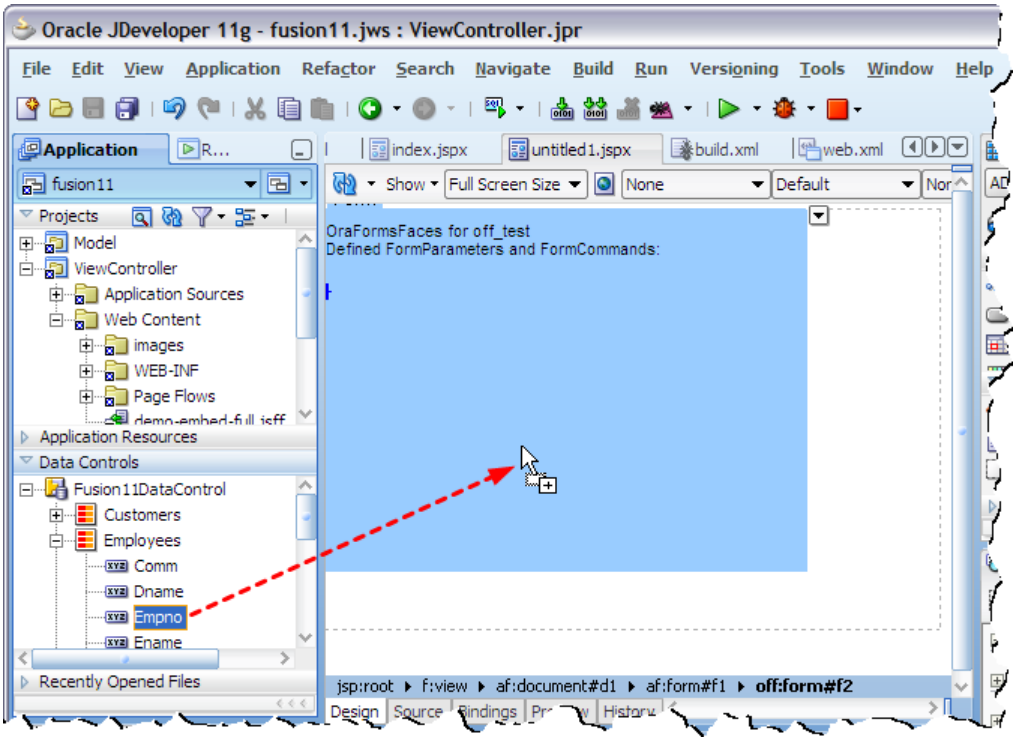
There are two ways to add an OraFormsFaces FormParameter component to a JSF page. Which method is more convenient depends on where the value of the FormParameter should come from. If the value comes from an ADF Data Control like a ViewObject, the drag-and-drop method from this section is best. If the value FormParameter comes from a normal JSF or ADF Managed Bean the next section on manually adding a FormParameter is more appropriate.

If you want to pass the value of an ADF Data Control to Oracle Forms, you can use the OraFormsFaces FormParameter component. There are two important things. As with any data bound control, you have to add the FormParameter component itself to the page and you also need to make sure the appropriate ADF bindings are setup to get the correct value from the ADF Data Control. If you have experience building an ADF application you know you can drag items from the Data Control Palette onto the JSF visual editor and create data bound JSF components that way. JDeveloper will then automatically create or re-use the appropriate bindings in the page definition.

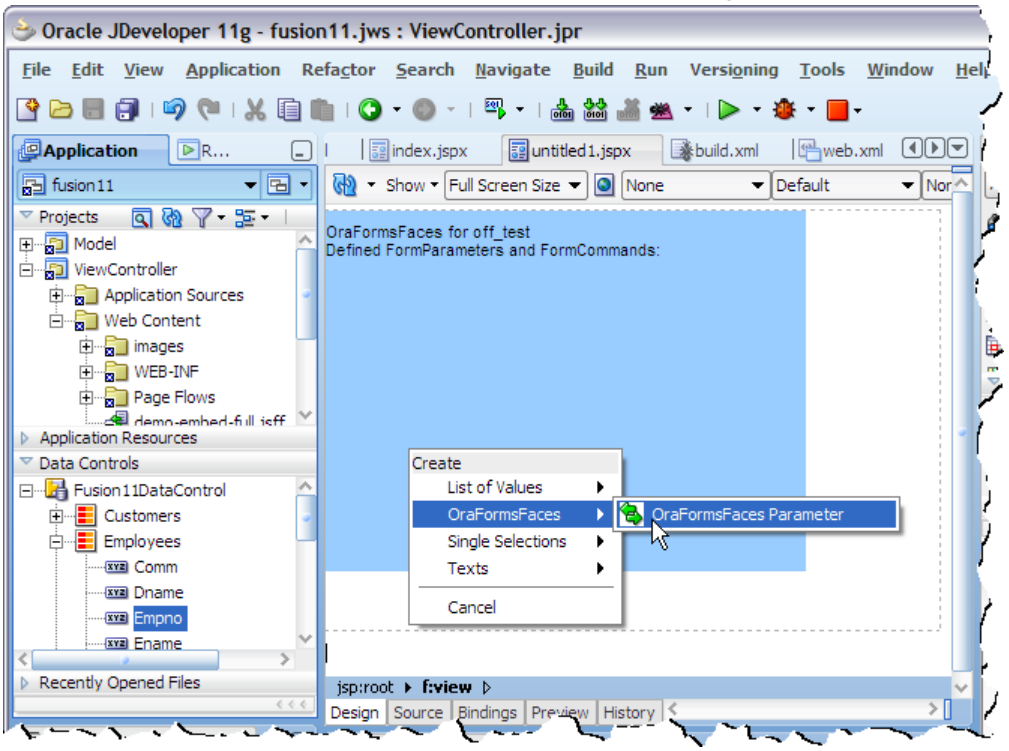
By installing the OraFormsFaces JDeveloper Extensions you have also added drag-and-drop support for OraFormsFaces components. This is why creating an ADF data-bound FormParameter component only requires a small number of steps:

- Ensure you have opened the **JSF page** with the OraFormsFaces Form component in JDeveloper's **JSP Visual Editor**. You also need to already have an OraFormsFaces **Form component** on the page as described in a previous section.
- Locate the **Data Control Palette**. In JDeveloper 11gR1 this is at the left hand side, while JDeveloper 10g had it at the right hand side. Expand the data control until you have located the attribute you want to pass to Oracle Forms as an attribute.

- Drag the attribute and drop it onto the OraFormsFaces Form component in the visual editor:

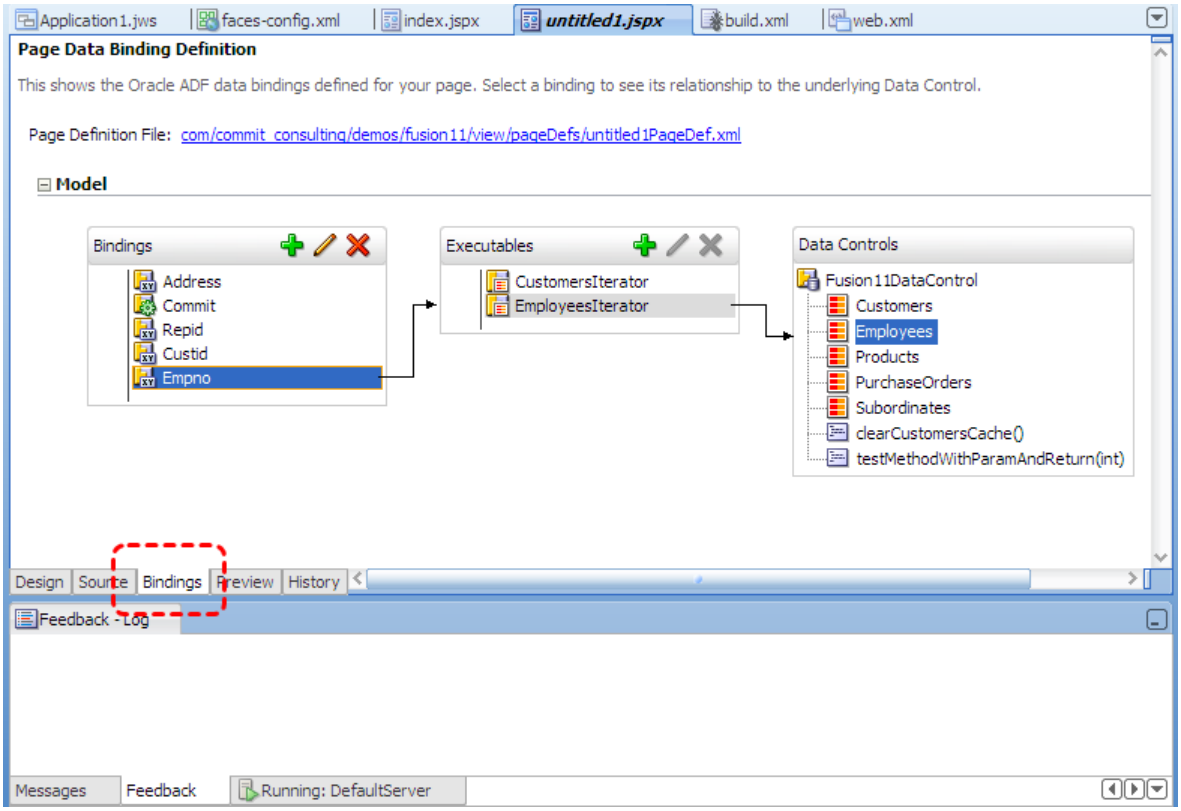


- A popup appears for you to select the type of component you want to create. Select the OraFormsFaces **FormParameter** component from the **OraFormsFaces** or **Texts** group:



Note:	In JDeveloper 10.1.3 the OraFormsFaces Parameter component is in the Texts menu along with some standard ADF and/or JSF components.

- This will create a **FormParameter** component with default settings. The advantage of this method is that the necessary **bindings** are automatically created in the page definition. If you are new to ADF development this is a convenient way to automatically create the bindings for you. More experienced ADF developers might want to review or change the binding properties in the page definition:



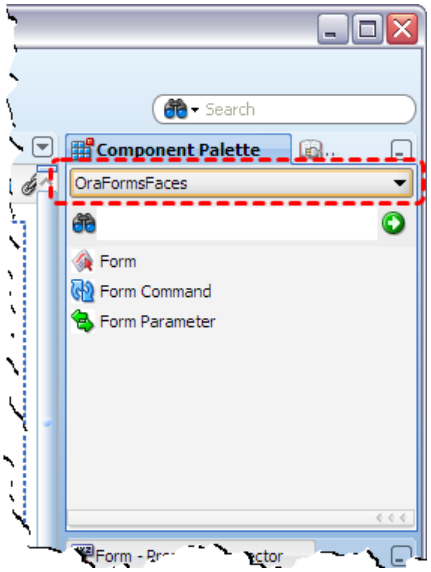
> **Note:** In JDeveloper 10.1.3 the ADF Bindings can be reviewed by right-clicking the visual editor and selecting Go to Page Definition from the popup menu.

- You can review the **FormParameter** component settings in the **Property Inspector**. If it is not visible, select **Property Inspector** from the **View** menu. Ensure you have the new FormParameter component selected in the visual editor, or select it from the document structure in the structure pane to be sure you select the correct component. See Setting the FormParameter properties later on for a description of all the properties.

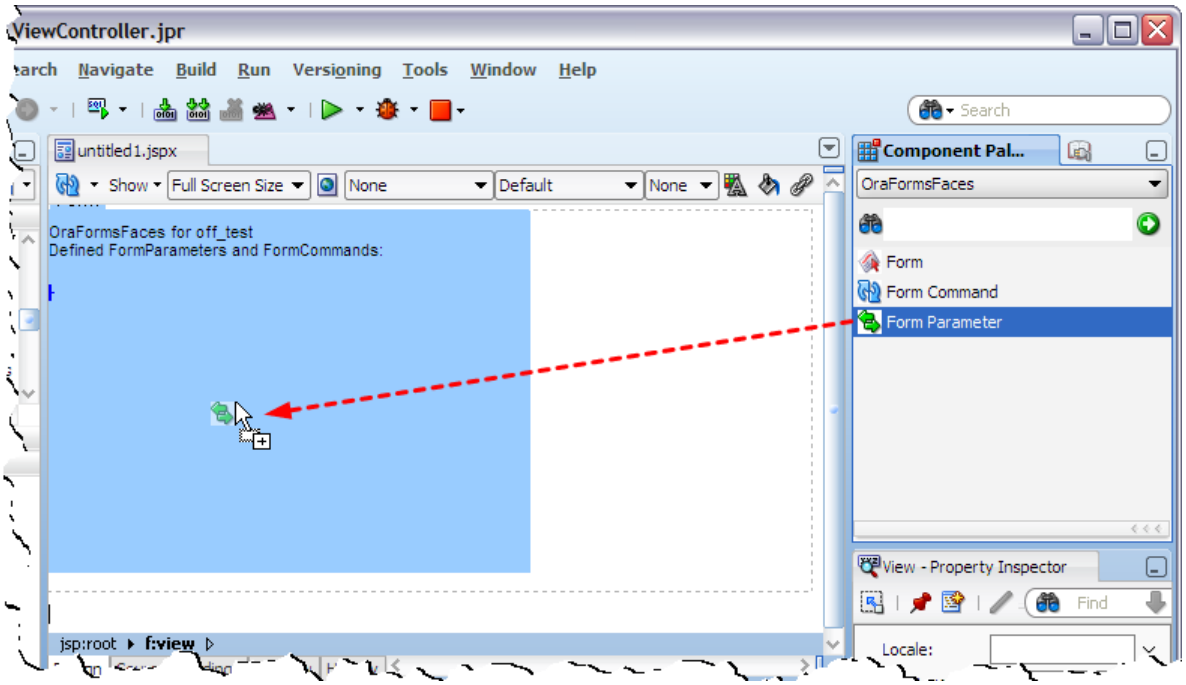
5.2. Defining a FormParameter Component Manually

When you are not using the drag-and-drop option from the previous section, you can also add a FormParameter component to your page manually. This is typically used if you want to pass a static value as a FormParameter or get the value from a JSF or ADF managed bean. Follow these steps to create a FormParameter component from scratch:

- Ensure the **OraFormsFaces** library is selected in the **Component palette** at the right-hand top of the screen. If the Component palette is not displayed, you can enable it using **Component Palette** from the **View** menu:



- Drag a **FormParameter** component from the Component palette and drop it on the **Form** component in the visual editor or the **<off:form>** component in the Structure panel:



- This will add a default **FormParameter** component without any attributes set. See ***5.3 Setting the FormParameter Design*** Time Properties for a description of the properties of the **FormParameter** component

Note:	You can also create a FormParameter component by right clicking on the Form component in your visual editor or in the Structure Panel. From the popup menu you can select **Insert inside > OraFormsFaces > FormParameter**

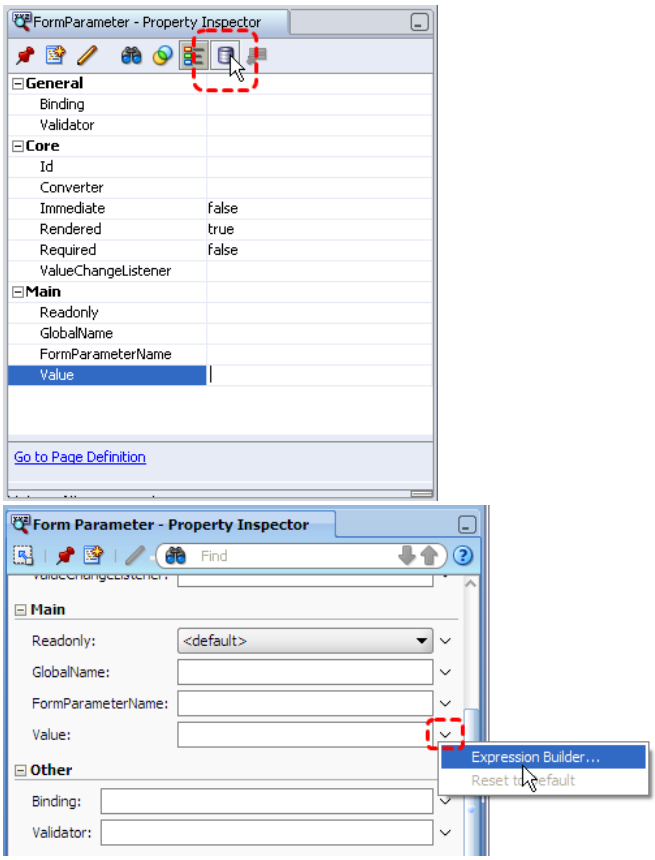
5.3. Setting the FormParameter Design Time Properties

Whether you created a **FormParameter** component by dragging an ADF Data Control or directly adding the FormParameter component, you end up with a FormParameter component in your page with several properties you can set. These properties can be set in the property inspector when the FormParameter component is selected in the visual editor or the structure pane. Alternatively you can type the properties and their values directly in the JSPX XML file.

Below is a description of the FormParameter properties and their usage:

- Core properties
  - **Id** – This can be used to set the ID of the FormParameter component. This is the unique ID of the JSF component within this JSF page. If you leave this property empty, JSF will assign a system generated ID to the component at runtime. It is best to set this property to a specific value. This ID can be used to retrieve the value of the FormParameter component from Oracle Forms PL/SQL. If you leave the ID generation to the JSF framework, the ID might change when you add or remove other components from the page, causing problems in your PL/SQL code.
  - **Converter** – This is a default JSF property to specify a Converter for a component. Due to the nature of the HTTP protocol when the value of a property is transmitted to or from the client web browser it has to be represented as a string. The Converter is responsible for converting the native data type of the managed bean from the value property (e.g. an integer, date, double, or a complex custom type) to a string and to convert the submitted string back to a native data type.
  - **Immediate** – This is also a default JSF property which indicates that the submitted value of the parameter should be set to the managed bean early in the JSF page lifecycle. If you are not yet familiar with the JSF page life cycle, it is best to keep this property at its default value.
  - **Rendered** – This is a default JSF property which indicates if the component should be rendered (included) in the HTML response to the client. You can set this `false`, to disable the component or use an Expression Language (EL) expression to render the component conditionally. For most usages, you can leave it to its default value of `true`.
  - **Required** – This property indicates if a value is required for this parameter. Again, this is a default JSF component property for input components.
  - **ValueChangeListener** – This is also a default JSF property for input components. You can specify a managed bean method to be fired whenever the value of this FormParameter component changes. This is typically when the page that embeds the Form and the FormParameter is submitted back to the JSF web server. For simple usages, this can be left empty.
- Main properties
  - **Readonly** – This boolean property indicates if the parameter is read-only. If this property is set to `true`, the parameter is seen as read-only for the JSF framework. This means that when the page is submitted back to the JSF server the value of the parameter is ignored. If the Readonly property is set to `false`, the parameter value can be changed from within Oracle Forms PL/SQL. On a subsequent page request the value is set to the bean property specified in the Expression Language (EL) expression of the FormParameter's **value** property. With the readonly property set to `true` a number of other FormParameter properties become irrelevant: **validator**, **immediate**, **required**, and **valueChangeListener**.

- **GlobalName** – The value of any FormParameter component can be retrieved from Forms PL/SQL based on the FormParameter component ID. However it is very common for existing Forms PL/SQL code to use globals to receive external parameters. By setting this **globalName** property, OraFormsFaces will automatically create an Oracle Forms global with the specified name and the appropriate value before executing the Oracle Forms form. For instance, setting this property to `custid`, will create an Oracle Forms global named `:global.custid` with the value of this FormParameter. This can be used to integrate existing Oracle Forms modules that rely on the existence of globals. By default, this property does not have a value and OraFormsFaces will not create a Forms global in that case.
- **FormParameterName** – The value of any FormParameter component can be retrieved from Forms PL/SQL based on the FormParameter component ID. However it is very common for existing Forms PL/SQL code to use parameters defined at design time in Forms Builder to receive external parameters. By setting this **formParameterName** property, OraFormsFaces will pass the value of this FormParameter to the Oracle Forms form as a true Oracle Forms parameter. Internally, OraFormsFaces will use a `CALL_FORM` built-in to start the specified Oracle Forms form. Before calling the form, OraFormsFaces will create an Oracle Forms parameter list and will pass this parameter list to the `CALL_FORM` built-in. For instance, setting this property to `custid` will cause OraFormsFaces to create `:parameter.custid` before calling the requested form. When using this functionality, ensure that the Oracle Forms form being called does have a parameter defined with this same name.
- **Value** – This is the actual value of the FormParameter component. You can either set this to a fixed value or use an Expression Language (EL) expression to refer to a managed bean or ADF binding property. When the value from an ADF Data Control is needed, the easiest way is to drag-and-drop the data control as described in ***5.1 Defining a FormParameter by Dragging an ADF Data Control***. If you want to get the value from an JSF Managed Bean, the easiest way is to use the EL expression builder that comes with JDeveloper:



- Other properties
  - **Binding** – This is a default JSF property which binds the OraFormsFaces FormParameter component to a backing bean property. It will instruct the JSF framework to call the setter method of the backing bean property to pass a reference to the OraFormsFaces FormParameter component. This can be used for programmatic access to the FormParameter component at runtime. For most usages, this property can be left to its default (empty) value which disables binding to a backing bean.
  - **Validator** – This can specify a JSF Validator to validate the value of the parameter. This is relevant for non-read only parameters. By specifying a validator, you can validate the value of the parameter. This is default JSF functionality for input components.

5.4. Passing JSF Parameters to Oracle Forms

The value of each FormParameter component can be retrieved from within Oracle Forms PL/SQL. For this, the `off_lib.pll` PL/SQL library contains a package called `offParams` with a number of utility procedures and functions. The function to get the value of a FormParameter component is `offParams.getParamValue`, for example:

```
v_custid := offParams.getParamValue(offParams.getParameters, 'custid');
```

This example retrieves the value of a FormParameter component into a PL/SQL variable named `v_custid`. The ID of the FormParameter component is ***custid***. This is why it is important to set the Id property of the FormParameter component to a logical name. You are referring to the FormParameter component based on its Id. You can leave the Id property empty in the JSF page. If you do that, JSF will assign a system-generated Id to the component. Technically, you can refer to the FormParameter component from within PL/SQL based on this system-generated Id. But if you add or remove components from the JSF page, the Id of the FormParameter component might change and you will have to change the PL/SQL code accordingly.

The `offParams.getParamValue` requires two arguments. The first one is an OraFormsFaces data structure with all the parameter information and the second one is the actual ID of the parameter you are requesting. The first argument can simply be retrieved by calling `offParams.getParameters`. This extra step is required for performance reasons. Getting the parameter information requires a roundtrip from the Forms server to the client. With the separate call to `getParameters` only one roundtrip is required even if multiple parameters are required. For instance:

```
v_params := offParams.getParameters;
v_custid := offParams.getParamValue(v_params,'custid');
v_ordid  := offParams.getParamValue(v_params,'ordid');
```

The above examples showed how you can retrieve the value of a FormParameter from within PL/SQL. If you are integrating existing Oracle Forms forms in a JSF application, it is very likely that these forms are already built to use **parameters** or **globals**. They might, for example, query the appropriate record when the form is started based on a parameter. OraFormsFaces comes with two convenient ways to handle this as described in the next sections.

5.4.1. Setting Oracle Forms Globals

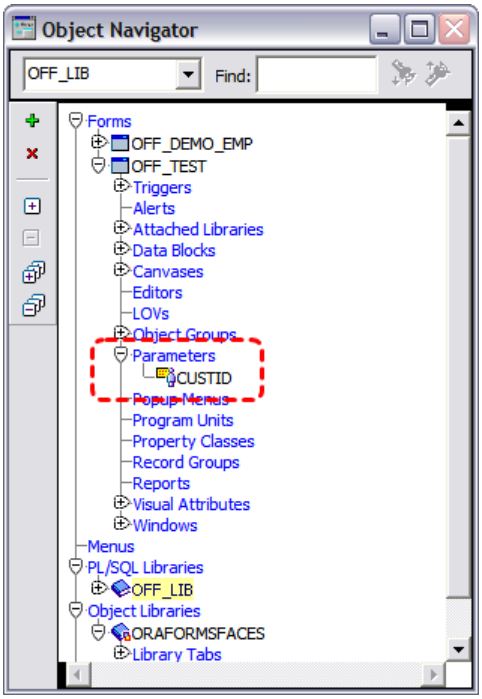
Oracle Forms **globals** are a common way to set and pass parameters between Oracle Forms modules. Your existing Oracle Forms application might, for example, use a global named `:global.custid`. The customer form might have a `when-new-form-instance trigger` that automatically queries the appropriate customer record based on this global.

In such a scenario you can set the **GlobalName** property of the **FormParameter** component in the JSF page. For instance, if you set the **GlobalName** property to ***custid***, the supporting OraFormsFaces code will automatically create an Oracle Forms global named `:global.custid` and set its value to the FormParameter component **value** before starting the form specified in the OraFormsFaces Form component.

This feature saves you the trouble of adding PL/SQL code to the form to retrieve the value of the FormParameter value using the `offParams` package. This means an existing Oracle Forms form can continue to use globals without any changes to the PL/SQL code in the form.

5.4.2. Setting Oracle Forms Parameters

Oracle Forms form modules can also define **Parameter** objects. These objects can be seen in the **Parameters node** in the Object Navigator when the form is opened in Oracle Forms Builder. When an Oracle Forms form is called from another form or from the Oracle Forms menu, a value can be passed for this parameter. The customer form might have a parameter `CUSTID` defined and a `when-new-form-instance trigger` that automatically queries the appropriate customer based on this customer ID parameter.



In such a scenario you can use the **FormParameterName** property of the **FormParameter** component in the JSF page. For instance, if you set the **FormParameterName** property to ***custid***, the supporting OraFormsFaces code will automatically create a parameter list that includes a parameter named `custid` when calling the form specified in the OraFormsFaces Form component.

This feature saves you the trouble of adding PL/SQL code to the form to retrieve the value of the FormParameter value using the `offParams` package. This means an existing Oracle Forms form can continue to use Parameters without any changes to the PL/SQL code in the form.

Remember that your Oracle Forms form must have a parameter with the specified name setup at design time. If the form does not have a parameter but is nonetheless called with a parameter, it will throw an error.

5.5. Passing Values from Oracle Forms PL/SQL to JSF

If the FormParameter's **ReadOnly** property is set to `false`, the component is also capable of "receiving" values from Oracle Forms. The value of such a **FormParameter** component can be set from within Oracle Forms PL/SQL. For this, the `off_lib.pll` PL/SQL library contains a package called `offParams` with a number of utility procedures and functions. The function used to set the value of a FormParameter component is `offParams.setParamValue`, for example:

```
offParams.setParamValue('custid', :cust.id);
```

This sets the value of the FormParameter component to the value of the **ID** item in the **CUST** Forms block. Obviously, you can also use this to set a FormParameter to a fixed value of the value of a PL/SQL variable. The ID of the FormParameter component is ***custid***. This is why it is important to set the Id property of the FormParameter component to a logical name. You are referring to the FormParameter component based on its Id. You can leave the Id property empty in the JSF page. If you do that, JSF will assign a system-generated Id to the component. Technically, you can refer to the FormParameter component from within PL/SQL based on this system-generated Id. But if you add or remove components from the JSF page, the Id of the FormParameter component might change and you will have to change the PL/SQL code accordingly.

The above call to `offParams.setParamValue` only sets the value of a hidden HTML element in the browser HTML page. The value is not submitted to the JSF server immediately but will be in the next page submit. The OraFormsFaces FormParameter then passes value on to the managed bean used in the EL expression of the **Value** property of the FormParameter component. It will also apply any specified validators, change listeners, converters, etc. You can use an OraFormsFaces FormCommand component to submit the JSF page back to the server from Forms PL/SQL code. See ***6 Invoking JSF*** Commands for more details.

> **Caution:** Calling `offParams.setParamValue` only sets the value of a hidden element in the HTML page. The value is not applied to the JSF managed bean or ADF data binding until the next page submit. You can also initiate this page submit from PL/SQL with the OraFormsFaces FormCommand component.

6. Invoking JSF Commands from Forms PL/SQL

OraFormsFaces enables triggering of events beyond technology borders. This includes (server side) JSF beans and Java code, client side JavaScript and PL/SQL code executing at the Forms server. To achieve this, OraFormsFaces comes with a JSF component and a number of supporting PL/SQL and JavaScript libraries. This chapter will focus on the OraFormsFaces **FormCommand** JSF component that allows PL/SQL code to postback to the JSF server thereby submitting any pending values and invoking JSF server side actions and commands.

To trigger JSF events (or Commands as they are called in JSF) you need a supporting **FormCommand** component as a child of the OraFormsFaces **Form** component. JSF has a group of Command components. Typical examples are Command Button and Command Hyperlink. These types of components can be triggered by the user and will invoke methods of managed beans and/or initiate navigation to other JSF pages.

OraFormsFaces also comes with a Command component, called **FormCommand**. This JSF component can be triggered from within Oracle Forms PL/SQL. This allows the developer to invoke a server side JSF managed bean method or initiate navigation to another JSF page from within Oracle Forms PL/SQL.

Let's begin by adding a FormCommand component to your JSF page that uses an OraFormsFaces Form component.

6.1. Defining a FormCommand by Dragging an ADF Data Control Method

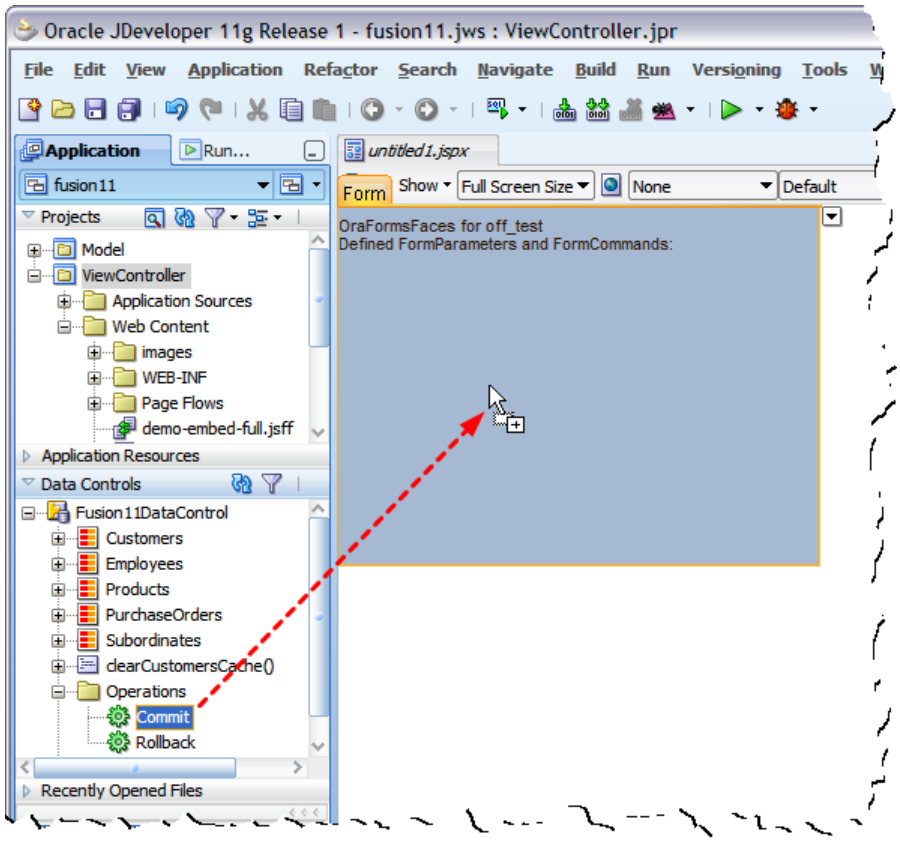
There are two ways to add an OraFormsFaces FormsCommand component to a JSF page. Which method is more convenient depends on which Java method should be executed when the FormsCommand is triggered. If the method is exposed through an ADF Data Control like an Application Module or a ViewObject, the drag-and-drop method from this section is best. If the method to be triggered by the FormCommand comes from a normal JSF or ADF Managed Bean it is better to skip to the next section on manually adding a FormCommand.

If you want to trigger an ADF Data Control method from Oracle Forms, you can use the FormCommand component. There are two important things. As with any data bound control, you have to add the FormCommand component itself to the page and you also need to make sure the appropriate ADF bindings are setup to invoke the correct method from the ADF Data Control. If you have experience building an ADF application you know you can drag methods from the Data Control Palette onto the JSF visual editor and create data bound JSF command components that way. JDeveloper will then automatically create or re-use the appropriate bindings in the page definition.

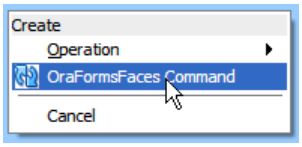
By installing the OraFormsFaces JDeveloper Extensions you have also added drag-and-drop support for OraFormsFaces components. This is why creating an ADF data-bound FormCommand component only requires a small number of steps:

- Ensure you have opened the **JSF page** with the OraFormsFaces Form component in JDeveloper's **JSP Visual Editor**.
- Go to the **Data Control Palette**. In JDeveloper 11gR1 this is at the left hand side, while JDeveloper 10g had it at the right hand side. Expand the data control until you have located the method you want to invoke from within Oracle Forms.

- Drag the method and drop it onto the OraFormsFaces Form component in the visual editor:

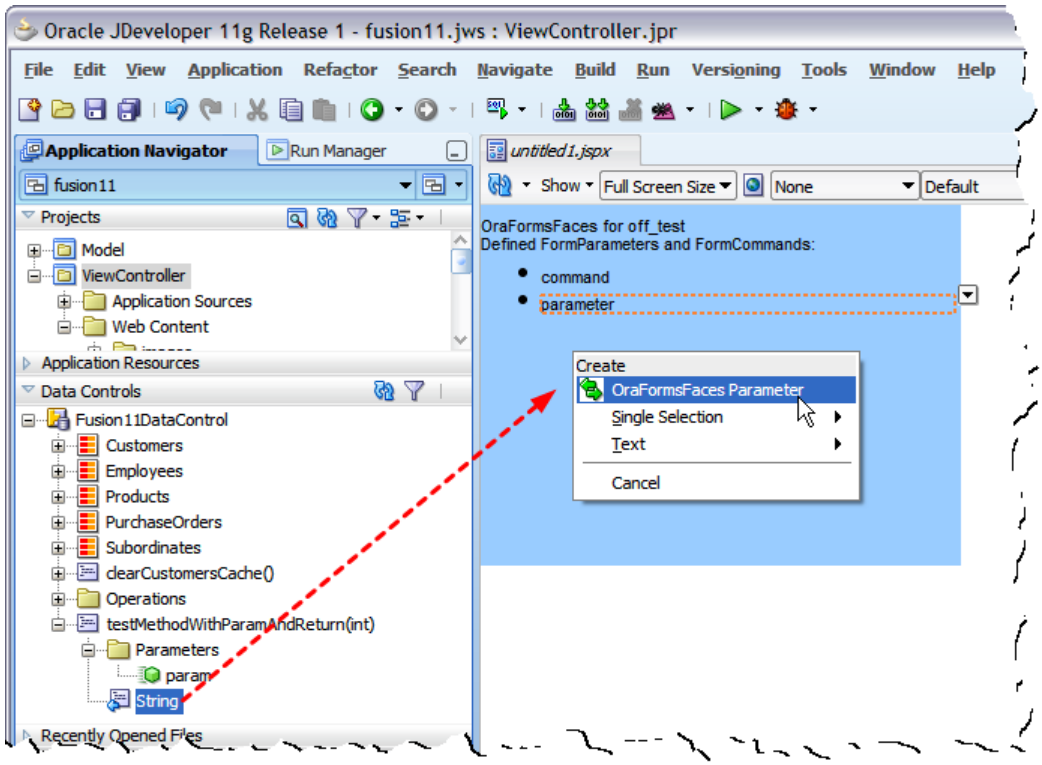


- A popup appears for you to select the type of component you want to create. Select the OraFormsFaces **FormCommand** component:



- This will create a **FormCommand** component with default settings. The advantage of this method is that the necessary **bindings** are automatically created in the page definition. If you're new to JSF development this is a convenient way to automatically create the bindings for you. More experienced JSF developers might want to review or change the binding properties in the page definition.

- If you have created a **FormCommand** for a method that requires parameters or had a return value, you might also want tocreate **FormParameter** components for these parameters and/or return values. See ***5.1 Defining a FormParameter by Dragging an ADF Data Control*** for an overview on how to create FormParameter components. You can use the same method to drag-and-drop the method parameters or return value from the data control palette onto the Form component and expose them as FormParameter components:



Exposing method parameters or return values as FormParameter components allows you to set the values of parameters before invoking the FormCommand and to retrieve the return value after invoking the FormCommand. This can all be controlled from Forms PL/SQL.

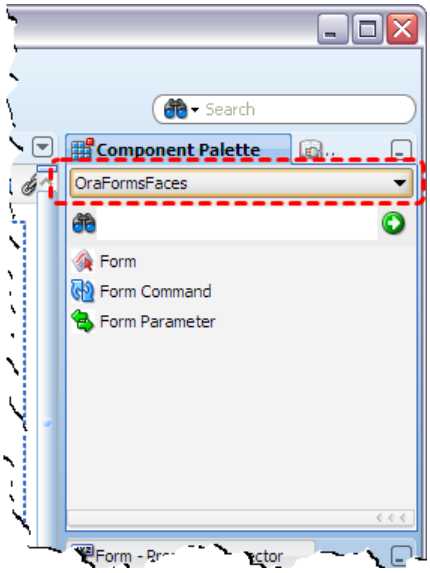
- You can review the **FormCommand** component settings in the **Property Inspector**. If it is not visible, select **Property Inspector** from the **View** menu. Ensure you have the new FormCommand component selected in the visual editor, or select it from the document structure in the structure pane to be sure you select the correct component. See ***6.3 Setting the FormCommand Design*** Time Properties for a description of all the properties.

6.2. Defining a FormCommand Component Manually

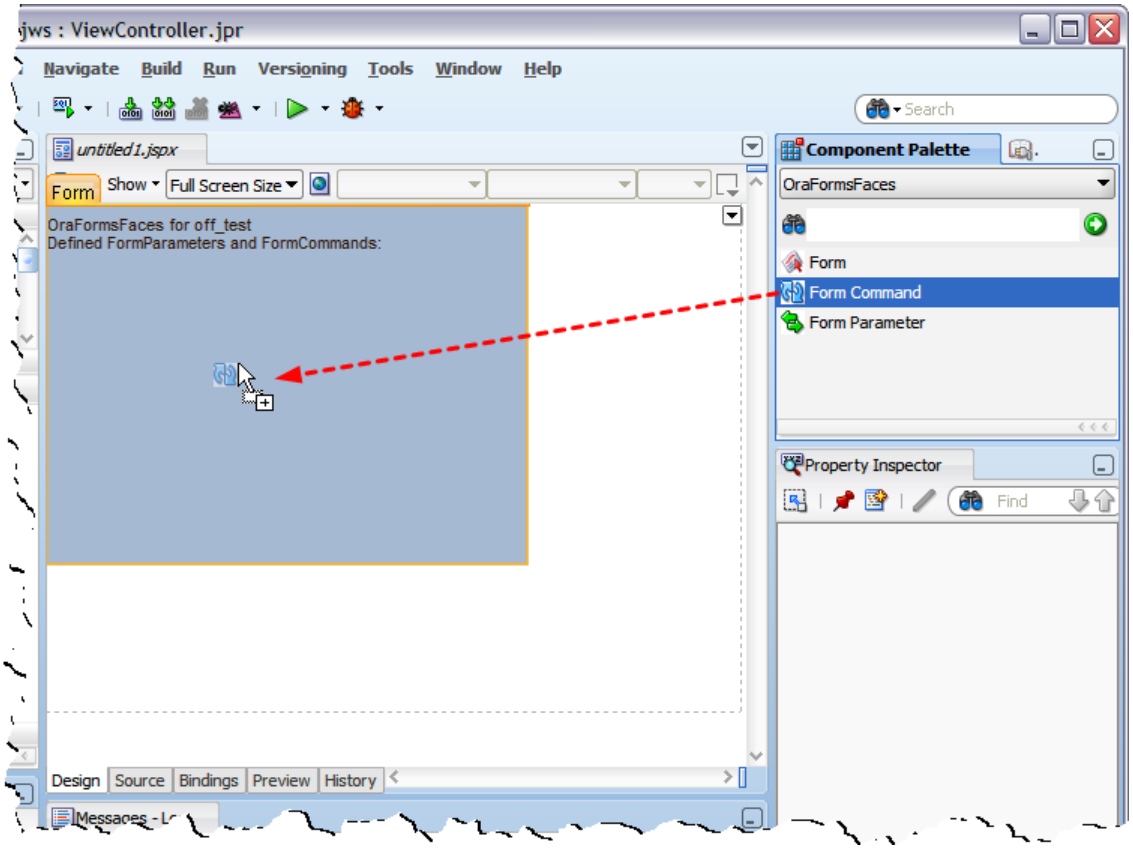
When you are not using the drag-and-drop option from the previous section, you can also add a FormCommand component to your page manually. This is typically used if you want to invoke a method from a JSF or ADF managed bean or just want to invoke a JSF navigation case to a different page.

Use the following steps to add a FormCommand component from the Component Palette:

- Ensure the **OraFormsFaces** library is selected in the **Component palette** at the right-hand top of the screen. If the Component palette is not displayed, you can enable it using **Component Palette** from the **View** menu:



- Drag a **FormCommand** component from the Component palette and drop it on the **Form** component in the visual editor or the **<off:form>** component in the Structure panel:



- This will add a default **FormCommand** component without any attributes set. See ***6.3 Setting the FormCommand Design Time Properties*** for a description of the properties of the **FormCommand** component.

> **Note:** You can also create a FormCommand component by right clicking on the Form component in your visual editor or in the Structure Panel. From the popup menu you can select **Insert inside > OraFormsFaces > FormCommand**

6.3. Setting the FormCommand Design Time Properties

A FormCommand component has a number of properties. These can be set in the Property Inspector when the FormCommand component is selected in the visual editor or the structure pane. Alternatively, you can type the properties and their values directly in the JSPX XML file.

Below is a description of the FormCommand properties and their usage:

- Core properties
  - **Action** – This property can have an Expression Language (EL) expression referring to a managed bean method. This method will be executed when the FormCommand is triggered. The method should return a String. The JSF framework will look in the `faces-config.xml` file for a navigation case originating from the current page and with a **from-outcome** property set to the value of the string returned from the action method. If such a navigation case is found, the JSF framework will follow this navigation case and render the page the case is pointing to.
  - **Id** – This can be used to set the ID of the FormCommand component. This is the unique ID of the JSF component within this JSF page. If you leave this property empty, JSF will assign a system generated ID to the component at runtime. It is best to set this property to a specific value. This ID will be used to trigger the FormCommand component from within Oracle Forms PL/SQL. If you leave the ID generation to the JSF framework, the ID might change when you add or remove other components from the page, causing problems in your PL/SQL code.
  - **ActionListener** – This is a default JSF property for command components. You can specify a managed bean property for this property as well. This method will also be executed when the FormCommand component is triggered. The difference with the Action property is that an ActionListener does not invoke any JSF navigation case. The method specified in the ActionListener is just executed when the FormCommand component is triggered, but it cannot influence the JSF navigation. The method can have any name, must be public, return void, and accept an ActionEvent as its only parameter.
  - **Immediate** – This is also a default JSF property which indicates that the Action and ActionListener should be executed early in the JSF page lifecycle. If you're not yet familiar with the JSF page life cycle, it is best to keep this property at its default value.
  - **Rendered** – This is a default JSF property which indicates if the component should be rendered (included) in the HTML response to the client. You can set this `false`, to disable the component or use an Expression Language (EL) expression to render the component conditionally. For most usages, you can leave it to its default value of `true`.
- Other properties
  - **Binding** – This is a default JSF property which binds the OraFormsFaces FormCommand component to a backing bean property. It will instruct the JSF framework to call the setter method of the backing bean property to pass a reference to the OraFormsFaces FormCommand component. This can be used for programmatic access to the FormCommand component at runtime. For most usages, this property can be left to its default (empty) value which disables binding to a backing bean.

6.4. Invoking FormCommand Components from Oracle Forms PL/SQL

A FormCommand component can be triggered from within Oracle Forms PL/SQL. For this, the `off_lib.pll` PL/SQL library contains a package called `offInterface` with a number of utility procedures and functions. The single procedure used to execute a FormCommand component is offInterface.execJSFCommand, for example:

```
offInterface.execJSFCommand('gocustdetails');
```

This will post back the current HTML page to the JSF web server. JSF will process the post back and will execute the OraFormsFaces FormCommand component which has its Id property set to ***gocustdetails***. This is why it is important to set the Id property of the FormCommand component to a logical name. You are referring to the FormCommand component based on its Id. You can leave the Id property empty in the JSF page. If you do that, JSF will assign a system-generated Id to the component. Technically, you can refer to the FormCommand component from within PL/SQL based on this system-generated Id. But if you add or remove components from the JSF page, the Id of the FormCommand component might change and you will have to change the PL/SQL code accordingly.

A FormCommand component can be used in combination with one or more FormParameter components. From within PL/SQL you can first set the value of one or more FormParameter components and then post the page to the server using a FormCommand component. This sends the values of the FormParameter components back to the JSF web server. The values will be set to the managed beans behind the FormParameter components and the managed bean behind the FormCommand component can process the changes.

6.5. Example: Master-Detail Synchronization

Below is a more advanced example of using a combination of a FormParameter and FormCommand components to have master-detail synchronization between an Oracle Forms master and JSF detail. Assume you have a page using a Form component to show a form with departments. The user can browse through multiple departments and press a button within the Form to show the employees for this department. However, the employees are displayed by a JSF page that does not use Oracle Forms. The challenge is to pass the department ID from Forms to JSF and initiate navigation to that page when the user clicks the Forms button.

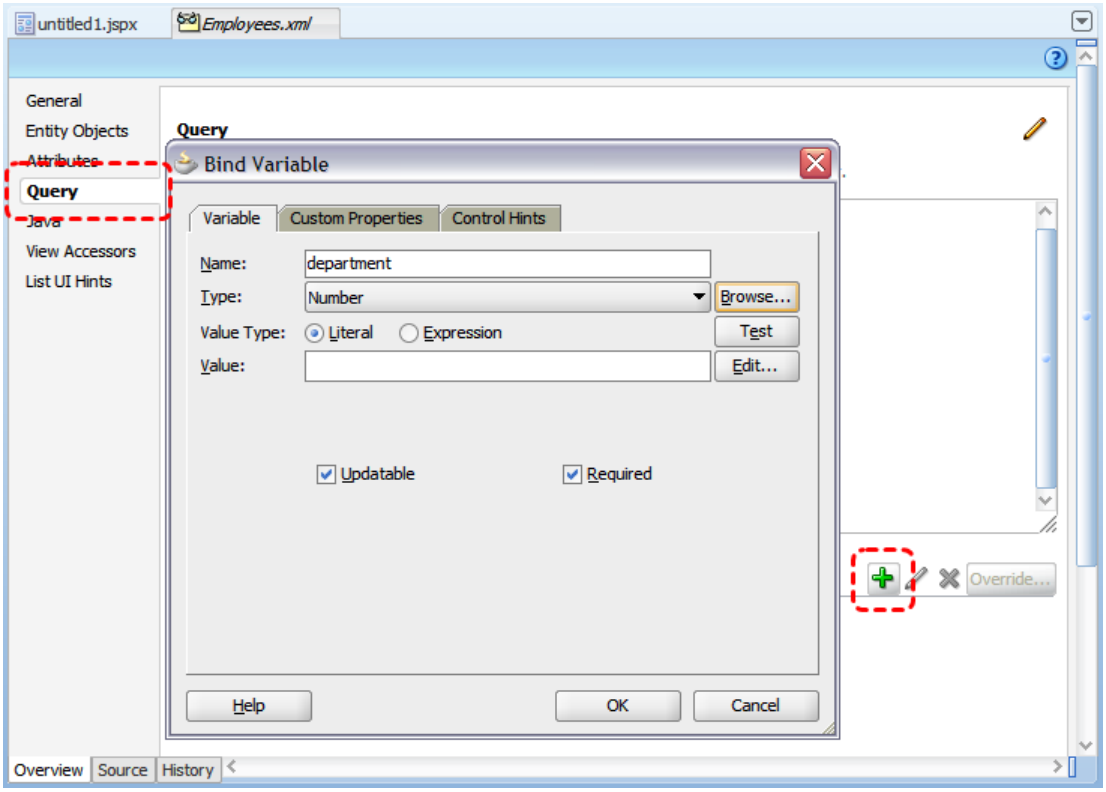
This master-detail synchronization combines most of the features explained in previous sections in one comprehensive example. This is a somewhat complex example that demonstrates a number of features. Make sure you first understand the basics of the OraFormsFaces components as explained earlier in this guide. Take the following steps to create this master-detail synchronization assuming you are using Oracle ADF Business Components in JDeveloper (although other technologies would use a similar approach):

- Create a page and include an OraFormsFaces Form component to show the departments. Detailed instructions on how to accomplish this can be found at ***4 Embedding Oracle Forms in a JSF Page***.
- You probably want to add a `WHEN-NEW-FORM-INSTANCE` trigger to the Form being used to automatically query records by executing the query:

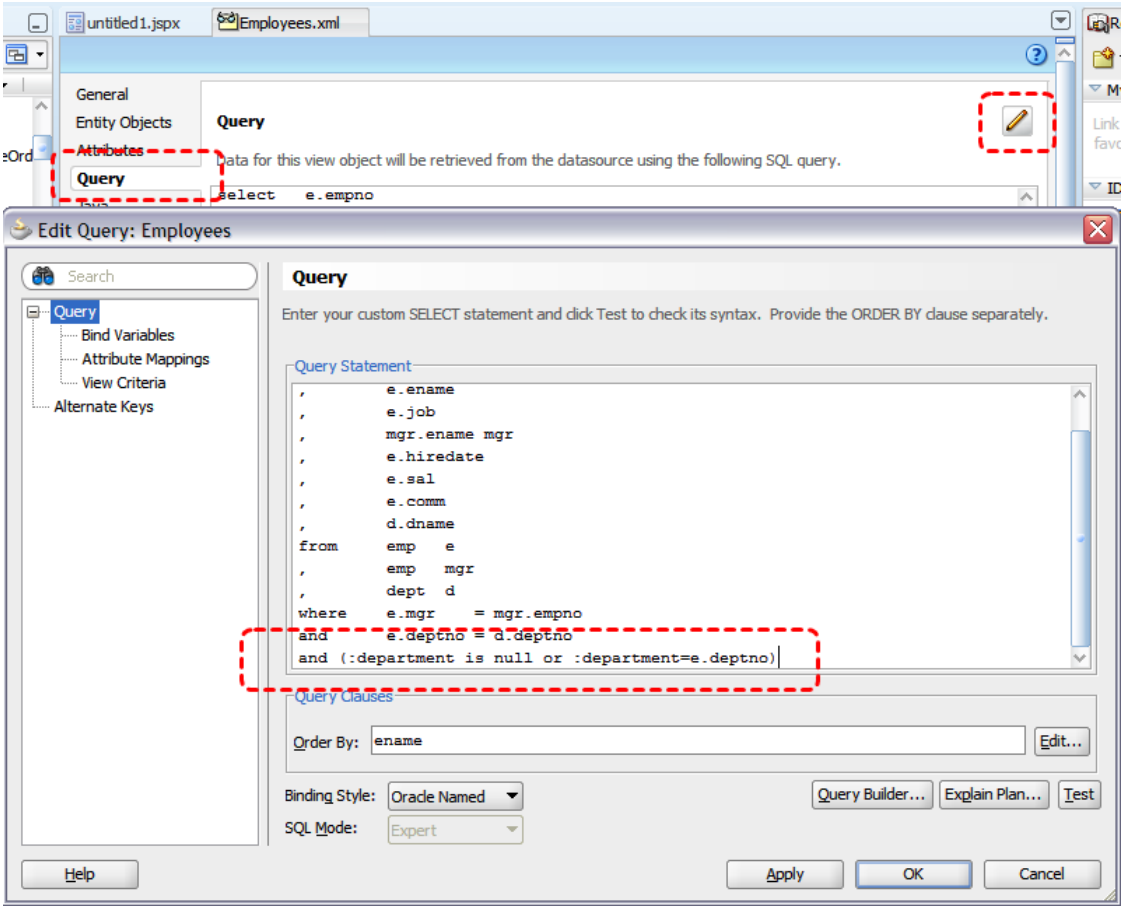
  ```
  go_block('departments');
  do_key('execute_query');
  ```

- Ensure you have an ADF Business Components ViewObject for the Employees and that it is exposed to the ViewController project through an ADF Business Components ApplicationModule.

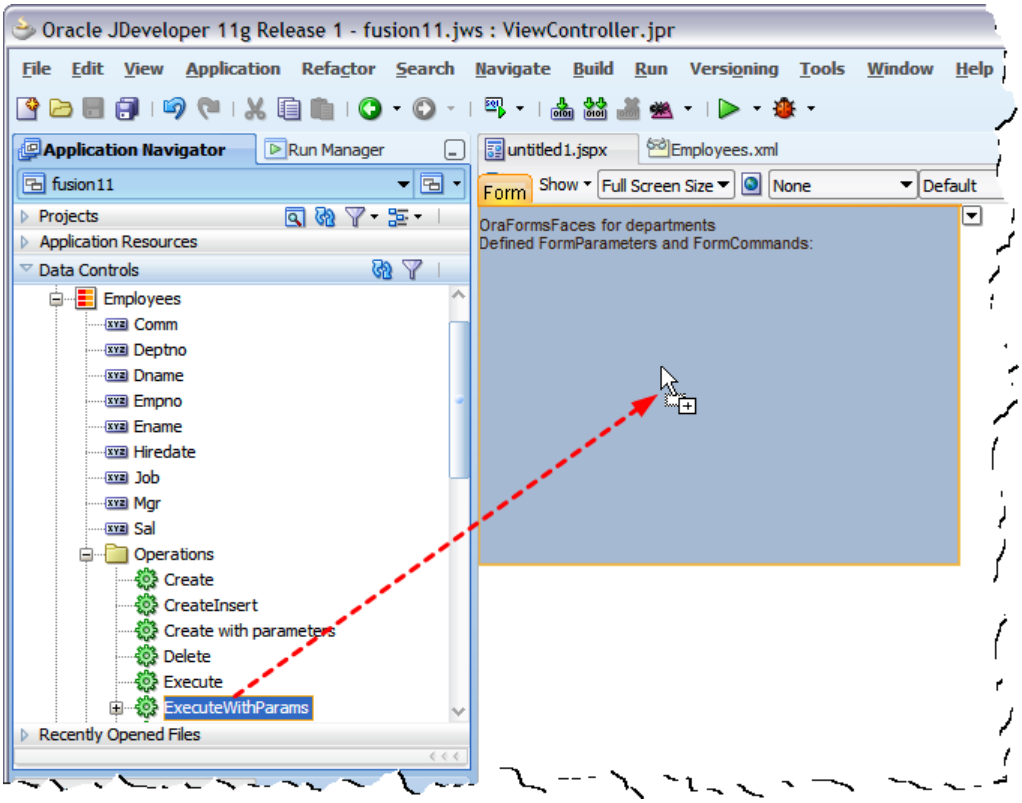
- Double click the ADF Business Components ViewObject in the Model project and go to the Bindings. Create a new binding called ***department***:



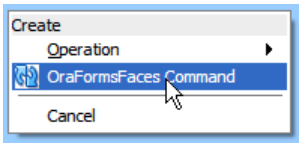
Next edit the SQL statement and make sure the where clause uses the bind variable appropriately. For example:



- Now that the ViewObject has a bind parameter it exposes the **ExecuteWithParams** method to the view layer to execute a query passing values for the bind parameter(s). It is this method that we want to execute from Forms PL/SQL while passing the appropriate department ID.
- Go back to the JSF page that embeds the departments Form. From the Data Control Palette drag the ExecuteWithParams method of the Employees data control onto the Form component:

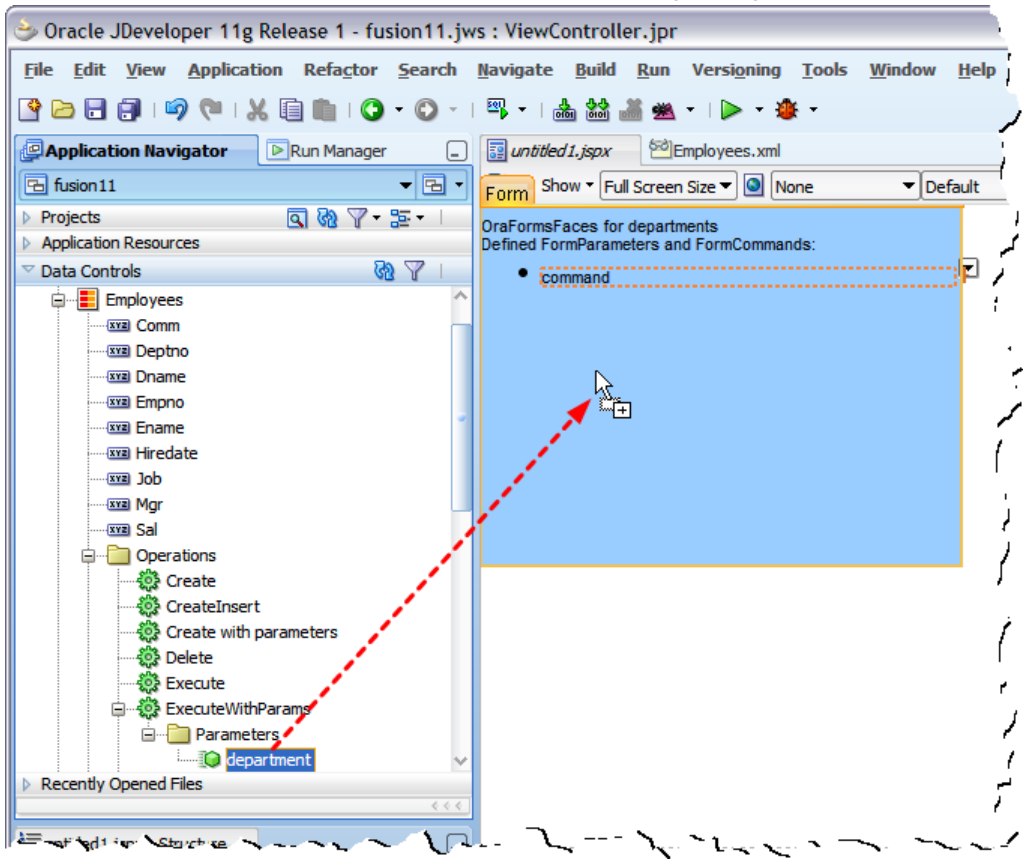


- JDeveloper asks the component type to create. Select the OraFormsFaces FormCommand component:

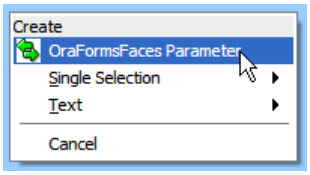


- By dragging the data control operation on the page, JDeveloper automatically created the bindings to the method and also created the bindings to the method argument to supply the department ID.

- To be able to set the value of this method argument, we need to drag it onto the Form as well. Expand the ExecuteWithParams method in the data control palette and drag the argument to the Form:



- Choose to drop the argument as a FormParameter component:



- Set the ID of the new **FormCommand** component to ***querydetails*** and the ID of the new **FormParameter** component to ***queryarg***. This ensures we can reference these components from PL/SQL.
- Set the action property of the FormCommand component to the name of the navigation case leading to the employees JSF page. This is defined in the ViewController JSF Navigation diagram.
- Now create a Forms button with the following `WHEN-BUTTON-PRESSED` trigger:

```
offParams.setParamValue('queryarg', :dept.deptno);
offInterface.execJSFCommand('querydetails');
```

- This PL/SQL first sets the value of the ***queryarg*** FormParameter and the immediately invokes the ***querydetails*** FormCommand.
- This assumes the Forms button is the only way to invoke the ExecuteWithParams method and navigate to this next page. If you also have other ways to directly invoke the ExecuteWithParams method, such as JSF buttons, tabs, or links then this won't work. The value of the FormParameter component would not be set. If you're in this situation you probably want to create a `WHEN-NEW-RECORD-INSTANCE` trigger on the departments block to immediately set the FormParameter whenever the user navigates to another record. This ensures the (hidden) value on the web page is always in sync with the current value in the Forms applet. So whatever method the user uses to invoke ExecuteWithParams, the correct value will be applied.

- This example assumes you want to navigate to a different page to show the employees. If you want to stay on the same page but still submit the page to invoke a FormCommand, you'll need to take advantage of ADF partial page rendering as described in the next section.

6.6. Example: Synchronization with ADF Partial Page Rendering

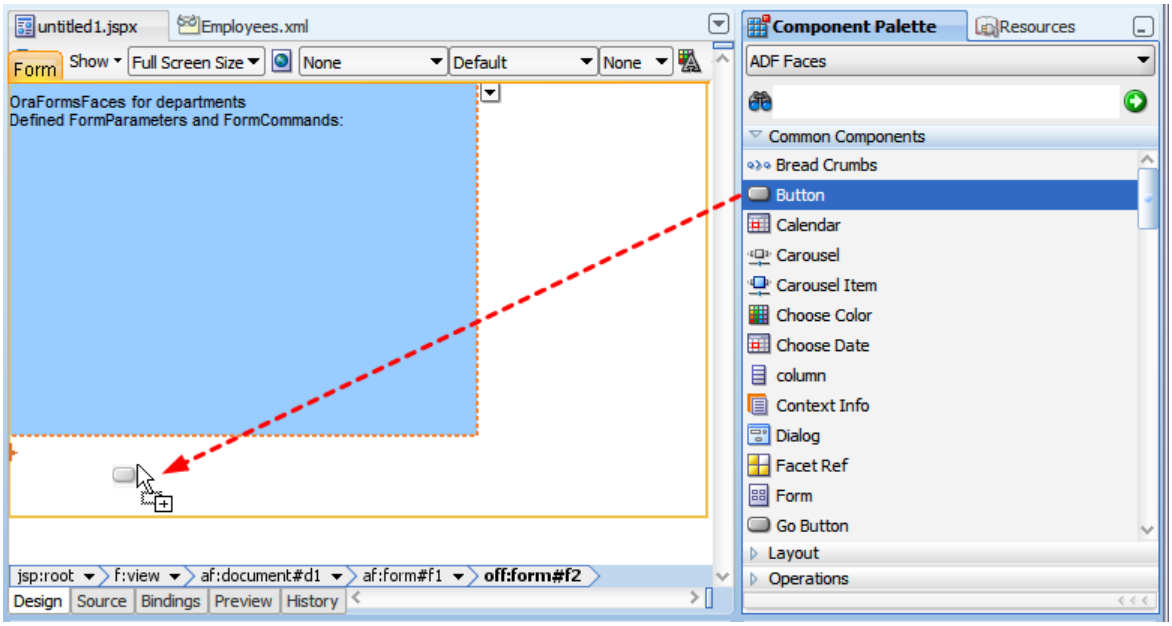
Calling `offInterface.execJSFCommand` will postback the current web page to the JSF server. If you haven't setup the FormCommand with an action that navigates to a different page, the current page will redisplay resetting the Oracle Forms applet to its initial state. If you don't want this to happen and only want to refresh parts of the page, you need a technique like Oracle ADF's partial page rendering.

OraFormsFaces is 100% JSF compliant and not dependent on Oracle ADF. This ensures OraFormsFaces can also be used in environments without Oracle ADF. However in this example we are going to benefit from Oracle ADF's partial page rendering feature. This feature allows you to submit the page back to the server yet only refresh parts of the page and not reload the entire page.

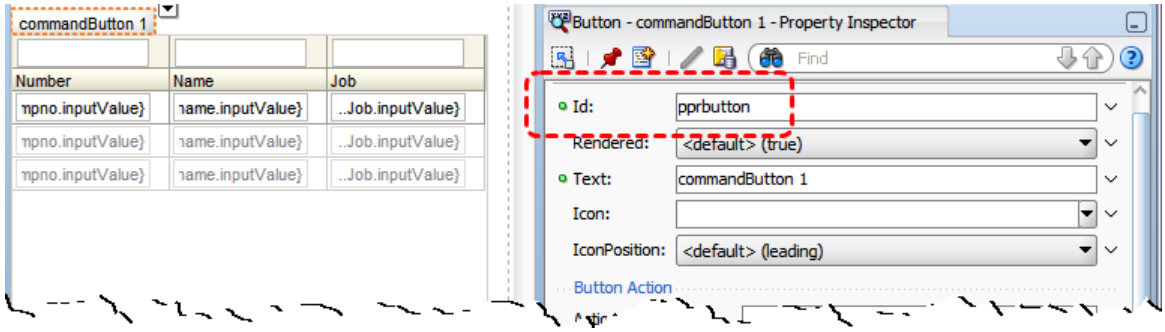
Partial page rendering requests can only be initiated from ADF components. Since OraFormsFaces is not an Oracle ADF component it cannot invoke a partial page rendering request directly. We can, however, use a workaround with an invisible ADF component to do the work for us.

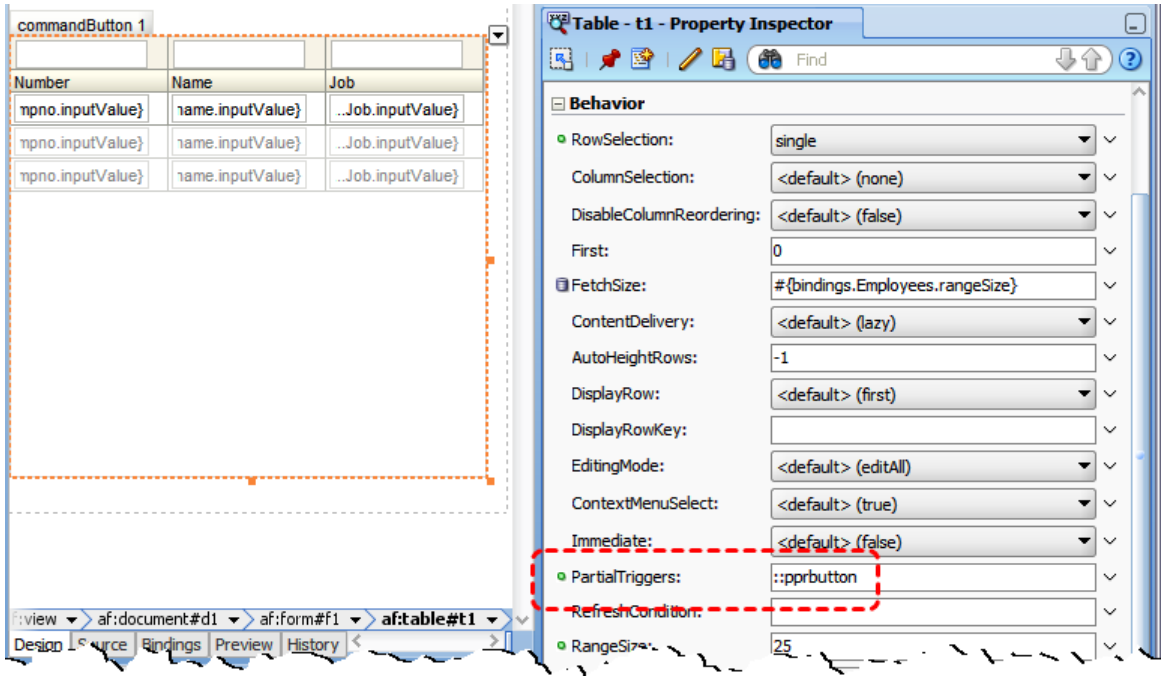
Take the following steps to use this workaround:

- Drag-and-drop an ADF Faces Button (In ADF 10 this was called a CommandButton) from the Component Palette to your JSF page:

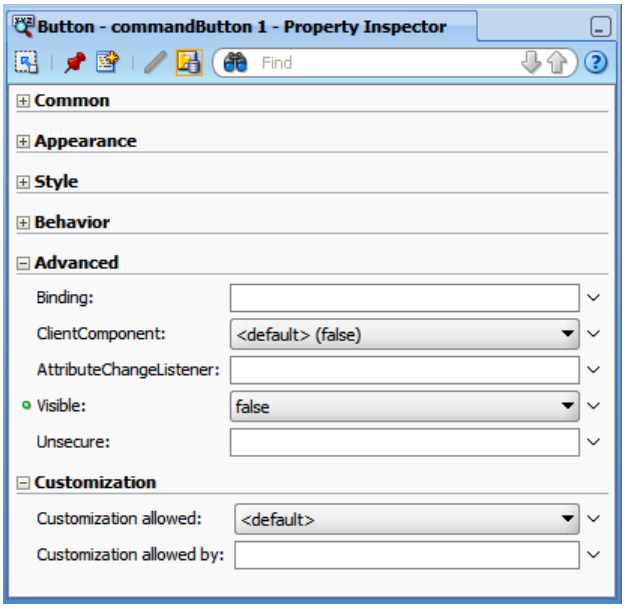


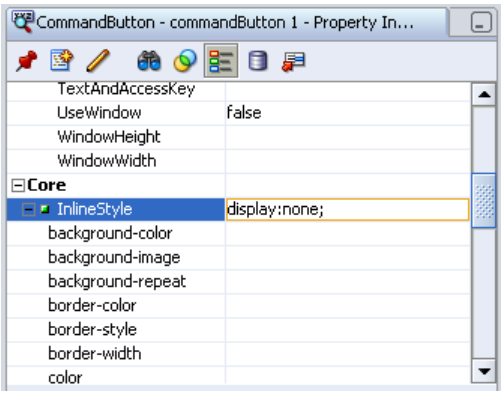
- Setup partial page rendering for this CommandButton they way you are used to by setting the ID of the CommandButton and the PartialTriggers property of the components you want to refresh on a partial-page-rendering request:





- Next we need to accomplish that the button is included in the client side HTML, but is not visible to the user. Having the button available in the page (but invisible) allows us to "push" it from Forms PL/SQL. Do not use the **Rendered** property to hide the button as that would remove the button entirely from the client side HTML. Instead use the ADF 11 property **Visible** and set it to `false`. In ADF 10 there is no Visible property but you can achieve the same result by setting the **InlineStyle** property to `display:none;`





- OraFormsFaces comes with a supporting PL/SQL function that uses JavaScript to simulate as if the user clicks an element in the page. This can be used to activate the button from Forms PL/SQL. For example:

```
offInterface.triggerClick('#pprbutton');
```

The argument to the offInterface.triggerClick procedure is a jQuery selector used to select the appropriate HTML element in the page. See the section on interacting with OraFormsFaces from JavaScript for more information on jQuery and its selectors.

- You can setup the CommandButton to call data control methods to requery certain view objects and using the PartialTriggers property you can refresh other elements on the page. This would allow you to create things like master-detail pages where the master is an Oracle Form and the detail is JSF table. It would also allow you to embed JSF graphs, trains, or other graphical components that refresh when certain actions occur in Forms PL/SQL.

Note:	ADF 11 Rich Client components introduced a comprehensive JavaScript object model at the client side to interact with ADF Components. This introduced many more methods to dynamically change or update parts of the page. You can always use the `offJavaScript` PL/SQL package to execute raw JavaScript. See the ADF Developer's Guides for more information on the possibilities.

7. Invoking PL/SQL Events from JavaScript

The previous sections described how to invoke a JSF command from Forms PL/SQL. But it is also possible to invoke events in the other direction. This means it is possible for other elements on a page to send an event to the OraFormsFaces applet on the page effectively triggering PL/SQL code in the Form. It is most likely this will be a button or link to be clicked by the user, but it could also be an ADF 11 Client-Side Event Handler.

The event is triggered by JavaScript and OraFormsFaces ensures the event is delivered to PL/SQL at the Forms server. There you can add your own custom handling or rely on the default handling of some predefined events. It is even possible to trigger PL/SQL events that return a value to the JavaScript that triggered the event in the first place.

7.1. Triggering Events from JavaScript

Invoking a Forms PL/SQL event can be achieved with a simple JavaScript call, for example:

```
OraFormsFaces["frm1"].sendMessage("event", "payload");
```

This first gets an OraFormsFacesForm component from the core OraFormsFaces object based on its ID of `frm1`. This OraFormsFacesForm JavaScript object is a client side representation of the OraFormsFaces Form component. It has a number of methods to interact with the component. The `sendMessage` method which we use here sends a message to the Forms applet effectively raising a server side PL/SQL event. The method requires two parameters. The first parameter is the name of the event to send. The second and final parameter is the payload for the event.

Note:	The ID used to access the OraFormsFacesForm object needs to be the fully qualified ID as used when rendering the page to the browser. If the Form component is contained in one or more Naming Containers, the IDs of these naming containers are prefixed to the ID you specified for the Form component at design time. Inspect the client side HTML to see the actual fully qualified ID used for the Form component. This is included in the HTML comment marking the beginning of the OraFormsFaces Form component.

The payload can be a simple string or event an empty string if not payload is required for an event. It is also possible to pass complex data structures like (nested) arrays or (nested) objects using JSON. OraFormsFaces comes with JSON libraries in JavaScript, PL/SQL and Java to convert complex data structures to text and to convert text back to complex data structures. See **Error! Reference source not found. Error! Reference source not found.** for more details.

One of the typical scenarios is to call `sendMessage` from a button onclick handler. Normally an onclick handler for a button is executed before invoking the default handling of the button. In a web browser this likely causes the page to be submitted and refreshed. This would de-activate the Forms applet preventing it to finish handling its event in PL/SQL. In fact, it is likely that the PL/SQL handling wants to interact even further with the page by setting FormParameter values or using some other JavaScript feature. This would not be possible if the browser is already navigating to the next page. If you do want the link or button to initiate page navigation after the event is handled in PL/SQL you can achieve the same result by invoking the page submit or navigation from the PL/SQL handling code as the last step of this handler. To achieve this see ***6 Invoking JSF Commands from Forms PL/SQL***.

You can prevent the default handling from executing leaving you with a button that only executes the client-side handling code specified in the onclick handler. To achieve this, the JavaScript expression specified in the onclick

property has to return false. For your convenience the `sendMessage` method returns false itself. All you have to do to stop the default form submit is return this value as the result of the onclick handler itself:

```
<h:commandButton value="Save" id="cb1"
        onclick="return OraFormsFaces['frm1'].sendMessage('event', 'payload')"/>
```

Notice the `return` keyword at the start of the onclick JavaScript expression to ensure the result of the `sendMessage` method is returned as a result of the onclick JavaScript expression itself.

> **Note:** The use of onclick or any other JavaScript event handlers can interfere with the complex JavaScript event model in Oracle ADF 11 Rich Client Components. Use of these event handlers is only advised with core JSF, ADF 10, or other simple components. See the next section for using ADF ClientListeners to achieve similar functionality in ADF 11 Rich Client Components.

7.2. Triggering Events from an ADF ClientListener

The `sendMessage` method from the previous section can be used to send an event with payload to Forms PL/SQL. You normally call `sendMessage` in response to some client-side event typically invoked by the end user. When using ADF 11 Rich Client Components you cannot use the normal JavaScript event handlers like onclick, onfocus, etc. directly on the ADF Rich Client Component.

Instead you have to add an ADF ClientListener to the ADF Rich Client Component. The ADF ClientListener specifies for which event a listener should be executed and it contains the name of the JavaScript function to execute. The referred JavaScript function has to accept a single argument of type ActionEvent.

OraFormsFaces comes with a convenience JavaScript function `OraFormsFaces.sendMessageADFClientListener` that implements the signature required by an ADF ClientListener. The function than inspects the component the ClientListener is attached to. This component has to have three ClientAttributes attached. These three attributes have to be named oraformsfaces_id, oraformsfaces_event and oraformsfaces_payload. These three attributes specify the ID of the Form components, the event to raise and the payload to pass. The values of these attributes can either be static values or use EL expressions to refer to dynamically generated content.

An example of this scenario is below:

```
<af:commandButton text="Save Form" id="cb1" clientComponent="true">
  <af:clientListener type="action"
                     method="OraFormsFaces.sendMessageADFClientListener"/>
  <af:clientAttribute name="oraformsfaces_id"      value="frm1"/>
  <af:clientAttribute name="oraformsfaces_event"   value="do_key"/>
  <af:clientAttribute name="oraformsfaces_payload" value="commit_form"/>
</af:commandButton>
```

This example shows an ADF Rich Client **CommandButton**. The button has a **ClientListener** attached that will fire on the action (=click) event. The **ClientListener** refers to the `sendMessageADFClientListener` method that is made available by OraFormsFaces. The button also has three attached **ClientAttributes** that contain the information required by `sendMessageADFClientListener` to send an event to the specified Form.

> **Note:** `sendMessageADFClientListener` internally also calls `ActionEvent.cancel()` to stop the normal processing of the event which would normally submit and refresh the page.

7.3. Handling Events in PL/SQL

OraFormsFaces ensures the event from the `sendMessage` method is relayed to the OraFormsFaces PL/SQL code. The `offTriggers.whenCustomJavascriptEvent` procedure handles the incoming event. This procedure first calls the `offCustom.handleEvent` function. The `offCustom` PL/SQL package contains a number of skeleton functions and procedures that offer extension points to OraFormsFaces. You can add your custom code to these procedures and functions.

The `offCustom.handleEvent` function has two parameters which are the event name and payload as they were submitted from JavaScript. You can add your own event handling code here:

```
function handleEvent(eventName in varchar2, payload in varchar2)
return   boolean is
begin
    if upper(name_in('system.current_form')) = 'OFF_DEMO_EMP' and
       eventName = 'refresh_subordinates' then
        set_block_property('emp', onetime_where, 'mgr=' || payload);
        go_block('emp');
        execute_query;
        return true; -- event consumed
    end if;
    return false; -- event not consumed
end handleEvent;
```

The `handleEvent` function has to return a boolean indicating if you handled/consumed the event or not. The default implementation of `offCustom.handleEvent` as shipped with OraFormsFaces does not handle any event and always returns `false` to indicate it did not consume the event.

If `offCustom.handleEvent` returns `false` it indicates it did not consume the event. This will cause `offTriggers.whenCustomJavascriptEvent` to continue and check if the event matches one of the predefined events that OraFormsFaces can handle. If so, the default handling of these events will be invoked. See ***7.5 Predefined Events*** for a list of predefined event and how they are handled. Having the `offCustom.handleEvent` in front of this allows you to event stop the default handling of predefined events.

> **Note:** An error will be thrown if the event is not handled by `offCustom.handleEvent` and it is also not a predefined OraFormsFaces event.

7.4. PL/SQL Events Returning Values to JavaScript

The previous example showed a simple example where JavaScript raised an event that was handled in PL/SQL. This is similar to doing a remote PL/SQL procedure call from JavaScript. With OraFormsFaces 3.0 an additional feature was introduced that allows the PL/SQL code to return a value back to JavaScript. This makes it analogous to doing a remote call to a PL/SQL function instead of a PL/SQL procedure.

Calling a PL/SQL event that returns a value from JavaScript is similar to raising an event without a return value:

```
OraFormsFaces["frm1"].sendReturningMessage("event", "payload",
    function(result) {
        alert("PL/SQL result is " + result);
    }
);
```

This calls the `sendReturningMessage` method on the OraFormsFacesForm JavaScript object. Besides the event and payload that were also required for `sendMessage`, this method has an additional parameter. This uses a JavaScript feature that functions are also objects that can be passed around. The third parameter has to be a

function with a single argument that handles the value returning from PL/SQL. This could either be a locally defined function as in this example or a reference to a previously defined function:

```
function handler(result) {
    alert("PL/SQL result is " + result);
}
...
...
OraFormsFaces["frm1"].sendReturningMessage("event", "payload", handler);
```

At the Forms server the event is initially handled by OraFormsFaces PL/SQL code in `offTriggers.whenCustomJavascriptEvent`. The handling is analogous to events without a returning value. The event is first passed to the `offCusom.handleEventCallback` function which you can implement to add your own custom handling:

```
function handleEventCallback(eventName in varchar2, payload in varchar2,
                             result in out varchar2)
return   boolean is
begin
    if upper(name_in('system.current_form')) = 'OFF_DEMO_EMP' and
       eventName = 'demoretevent' then
        result := get_block_property(payload, QUERY_HITS);
        return true; -- event consumed
    end if;
    return false; -- event not consumed
end handleEventCallback;
```

The `handleEventCallback` function has to return a boolean indicating if you handled/consumed the event or not. The default implementation of `offCustom.handleEventCallback` as shipped with OraFormsFaces does not handle any event and always returns `false` to indicate it did not consume the event.

Besides the boolean return, the function also has to set the value of the third (out) parameter. This is the string that will be returned to the client side JavaScript as the result of the event. The result value has to be a string. You can also pass complex data structures like (nested) arrays or (nested) objects using JSON. OraFormsFaces comes with JSON libraries in JavaScript, PL/SQL and Java to convert complex data structures to text and to convert text back to complex data structures. See **Error! Reference source not found. Error! Reference source not found.** for more details.

If `offCustom.handleEventCallback` returns `false` it indicates it did not consume the event. This will cause `offTriggers.whenCustomJavascriptEvent` to continue and check if the event matches one of the predefined events that OraFormsFaces can handle. If so, the default handling of these events will be invoked. See ***7.5 Predefined Events*** for a list of predefined event and how they are handled. Having the `offCustom.handleCustomEvent` in front of this allows you to event stop the default handling of predefined events.

Note:	An error will be thrown if the event is not handled by `offCustom.handleEventCallback` and it is also not a predefined OraFormsFaces event.

You need custom JavaScript to call `sendReturningMessage` from an ADF ClientListener. This cannot be handled by using the convenience function `sendMessageADFClientListener` as you also need custom JavaScript to handle the result coming back from the Forms PL/SQL code. See the ADF Developer's Guide how to add a custom JavaScript function to your page using `<af:resource>` or `<trh:script>`. An event handling function would look something like:

```
function handler(event) {
    OraFormsFaces["frm1"].sendReturningMessage("event", "payload",
        function(result) {
            alert("PL/SQL result is " + result);
        }
    );
}
```

The ADF components triggering this handler could look something like:

```
<af:commandButton text="Invoke" id="cb3">
    <af:clientListener type="action" method="handler"/>
</af:commandButton>
```

If you want to make the event handler more generic you could have it get the necessary values from ClientAttributes specified on the event triggering component:

```
function handler(event) {
    var source = event.getSource();
    var formid = source.getProperty("form_id");
    var event = source.getProperty("event_name");
    var payload = source.getProperty("event_payload");
    OraFormsFaces[formid].sendReturningMessage(event, payload,
        function(result) {
            alert("PL/SQL result is " + result);
        }
    );
}
```

This would need ClientAttributes for the component that also has the ClientListener attached:

```
<af:commandButton text="Invoke" id="cb3">
    <af:clientListener type="action" method="handler"/>
    <af:clientAttribute name="form_id" value="frm1"/>
    <af:clientAttribute name="event_name" value="event"/>
    <af:clientAttribute name="event_payload" value="#{bean.property}"/>
</af:commandButton>
```

7.5. Predefined Events

When the custom event handlers in offCustom.handleEvent or offCustom.handleEventCallback do not consume an event, OraFormsFaces checks if the name of the event matches any of the predefined events. If this is the case the default handling for these predefined events is invoked. This section describes the predefined events that are available in OraFormsFaces.

7.5.1. Predefined Events without a Returning Value

These are the pre-defined events that are handled by the OraFormsFaces code in `offTriggers.whenCustomJavascriptEvent` and which can be called from `OraFormsFacesForm.sendMessage` or `OraFormsFaces.sendMessageADFClientListener:`

- **do_key** – It will call the Oracle Forms `do_key` built-in and will use the payload as the parameter for the `do_key` call. This can be very convenient to trigger common Forms commands from other UI elements. An example using `sendMessage` is below:

```
OraFormsFaces["frm1"].sendMessage("do_key", "commit_form");
```

  Another example using an ADF ClientListener:

```
<af:commandButton text="Save Form" id="cb1" clientComponent="true">
  <af:clientListener type="action"
                     method="OraFormsFaces.sendMessageADFClientListener"/>
  <af:clientAttribute name="oraformsfaces_id"      value="frm1"/>
  <af:clientAttribute name="oraformsfaces_event"   value="do_key"/>
  <af:clientAttribute name="oraformsfaces_payload" value="commit_form"/>
</af:commandButton>
```

  This **do_key** event is particularly useful when you hide the toolbar with the clipping feature as explained at *12 Visual Integration*. When the toolbar is hidden from the user you'll need to give the user some other means to execute these common functions. Triggering these functions from true web buttons gives the application a more "web look-and-feel" and makes the application feel less like an Oracle Forms application. This can be a strategic choice when (gradually) moving away from Oracle Forms. For more information on this approach see *14 System Administration*.

- **copy** – This will call the Forms `copy` built-in to copy a value into a Forms item or global. The payload for this event has to be a JSON object containing two properties; `source` and `destination`. The value of the `source` property is copied into the Forms item or global specified in the `destination` property:

```
var payload     = {source:1234,destination:"emp.empno"};
var jsonPayload = OraFormsFaces.JSON.stringify(payload);
OraFormsFaces["frm1"].sendMessage("copy", jsonPayload);
```

- **go_form** – This will call the Forms `go_form` built-in using the payload as the argument for `go_form` specifying the form to navigate to.
- **go_block** – This will call the Forms `go_block` built-in using the payload as the argument for `go_block` specifying the form to navigate to.
- **go_record** – This will call the Forms `go_record` built-in using the payload as the argument for `go_record` specifying the form to navigate to.
- **go_item** – This will call the Forms `go_form` built-in using the payload as the argument for `go_form` specifying the form to navigate to.
- **when-applet-activated** – This event is used internally by OraFormsFaces and ***should never be raised explicitly***. It is used when the applet is restored from the legacy lifecycle cache to signal the Forms PL/SQL code the applet is being restored. For more details on this feature see *13 Forms Java Applet Instance Reuse*.
- **when-applet-suspended** – This event is used internally by OraFormsFaces and ***should never be raised explicitly***. It is used when the applet is suspended into the legacy lifecycle cache to signal the Forms PL/SQL code the applet is being suspended. For more details on this feature see *13 Forms Java Applet Instance Reuse*.

- **off$callback** – This event is used internally by OraFormsFaces to wrap an event that expects a returning value as described at ***7.4 PL/SQL Events Returning Values to JavaScript***. This event should ***never be raised from custom code*** directly.

7.5.2. Predefined Events with a Returning Value

These are the pre-defined events that return a value. These are handled in the OraFormsFaces code in `offTriggers.whenCustomJavascriptEvent` and can be called from `OraFormsFacesForm.sendReturningMessage:`

- **name_in** – It will call the Oracle Forms `do_key` built-in and will use the payload as the parameter for the `do_key` call. This can be very convenient to trigger common Forms
- **name_in** – This will call the Forms `name_in` built-in using the payload as the argument for `name_in`. The result of `name_in` is returned as result of the `OraFormsFacesForm.sendReturningMessage` JavaScript method.
- **show_alert** – This will call the Forms `show_alert` built-in using the payload as the argument for `show_alert` specifying the name of the Forms alert to show. Please note that the Forms alert already has to be defined at design time in the FMB file. The result of `show_alert` is returned as result of the `OraFormsFacesForm.sendReturningMessage` JavaScript method.

8. Supplied JavaScript Functions and Objects

8.1. Supporting JavaScript Technologies

8.1.1. jQuery

OraFormsFaces bundles jQuery which is described on their website as:

> *jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.*

Normally jQuery is available both under the `jQuery` variable and the shorthand `$` variable. This would make the two following statements equivalent:

```
$("div p");
jQuery("div p");
```

Registering the jQuery version shipped with OraFormsFaces under these global variable names can cause conflicts if you include jQuery in your application yourself or other libraries that embed jQuery. To prevent these conflicts, OraFormsFaces ***only*** registers jQuery under `OraFormsFaces.jq`. This does remove the potential for naming conflicts but it also means that many jQuery plug-ins won't work as they assume the availability of `jQuery` and/or `$`. OraFormsFaces itself does not use jQuery plug-ins but you need to be aware of this if you want to use the `OraFormsFaces.jq` object in combination with jQuery plug-ins.

Note:	The global `OraFormsFaces.jq` object is only created when an `<off:form>` component is included in the page. This means you can only use jQuery as shipped with OraFormsFaces on pages that also use a `<off:form>` component. If you also want to use jQuery on other pages, please ensure you include the jQuery library yourself.

Many of the JavaScript functions that ship with OraFormsFaces use jQuery internally. If you want to use the `OraFormsFaces.jq` object from your custom JavaScript code or from Forms PL/SQL, please refer to the official jQuery documentation at http://docs.jquery.com/Main_Page

8.1.2. JSON (JavaScript Object Notation)

OraFormsFaces enables you to pass information between Java code at the JSF server, JavaScript at the client and PL/SQL code at the Forms server. Most of these integrations involve different technologies or require the information to be transmitted across the network. This means only character strings can be exchanged and no complex or native data types like objects or collections.

OraFormsFaces uses JSON (JavaScript Object Notation) to overcome this issue. JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. For more information see http://www.json.org/

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence. These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In JSON, they take on these forms:

- An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma). An example is :

```
{ sex:"male", age:53, resident:false }
```

  This example shows an object with three properties; sex, age and resident.

- An array is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma). An example demonstrating an array with four simple integer elements is:

```
[ 2, 4, 8, 16 ]
```

- A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. Since object and arrays can contain values themselves, (complex) nested structures are possible. Some examples:

```
[ "text", 123, true, false, null ]
{ att1:"demo", att2:null, att3:[2,4,8,16], att4:{x:-1, y:0: z:null} }
```

- A string is a collection of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string. This means JSON also handles escaping of special characters, quotes etcetera. Some examples are:

```
"test"
"text with a \"quote\""
"some \u00e9\u18a2 unicode"
"some \b\f\n\r\tcontrol chars"
```

- A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used. Number are always in decimal notation. Negative numbers are prefixed with a – (minus), can contain a decimal point for fractions and suffixes with `e` or `E` for scientific notation.
- Whitespace can be inserted between any pair of tokens.

OraFormsFaces includes JSON libraries to create and parse JSON text in all three relevant languages; Java, JavaScript and PL/SQL. This allows you to create complex data structures like (nested) objects and arrays in one language, convert them to JSON text and then use a library in another language to parse this text back to an object or array.

8.2. Global JavaScript

OraFormsFaces supports two global JavaScript variables, ORAFORMSFACES_LOGGING and ORAFORMSFACES_LOG_LOCAL. These can be used to enable JavaScript debugging as explained in ***17.2.3 Enabling Client Side JavaScript Tracing***.

- **`ORAFORMSFACES_LOGGING`** – Force the use of a popup window for logging even if native window.console is detected.
- **`ORAFORMSFACES_LOG_LOCAL`** – Global boolean that, when set to true, triggers OraFormsFaces to enable logging to Firebug in Firefox or a popup window in other browsers.

8.3. OraFormsFaces JavaScript Namespace

OraFormsFaces use a global JavaScript namespace `OraFormsFaces` that contains a number of field and methods. These are methods that are not associated to Forms, Command or Parameters but are more generic or global. They have been combined into a single namespace to prevent naming conflicts with other JavaScript objects.

> **Note:** This Developer's Guide only contains the public field and methods that are available to your custom JavaScript. OraFormsFaces contains a lot more field and methods that have been marked as private and are not intended for use outside of OraFormsFaces' own code. Private fields and methods can change in newer versions without notice or backward compatibility. Both the public and private field and methods are documented in the JSDoc that ships with OraFormsFaces. This documentation is available at `JDEV_HOME/jdev/extensions/com.commit_consulting.oraformsfaces.v????/doc`

8.3.1. Fields

- **`OraFormsFaces.jq`** – A reference to the jQuery object created specifically for OraFormsFaces. We intentionally do not register jQuery under its default global names jQuery and $ to prevent any conflicts with other frameworks. See ***8.4 OraFormsFaces.jq Object*** for more details.

8.3.2. Methods

- **`OraFormsFaces.findForm(formId)`** – Returns an OraFormsFacesForm object from the internal array of objects based on the id of the form.
- **`OraFormsFaces.sendMessageADFClientListener(actionEvent)`** – Function that can be used as an Oracle ADF 11 Rich Client ClientListener method that calls OraFormsFacesForm.sendMessage based on the ClientAttributes specified with the event triggering component.
  actionEvent.cancel() is called to prevent further processing of the event by Oracle ADF. After that, three clientAttributes are retrieved from the event triggering component: oraformsfaces_id, oraformsfaces_event and oraformsfaces_payload. The values from these three attributes are used to call OraFormsFaces.findForm(oraformsfaces_id).sendMessage(oraformsfaces_event,oraformsfaces_payload);
  Also see ***7.2 Triggering Events from an ADF ClientListener***.
- **`OraFormsFaces.triggerClick(selector)`** – Triggers the click event on element(s) in the DOM tree which appears as if the user clicked the specified element(s). This can be used to trigger (hidden) buttons or other UI elements. Also see ***6.6 Example: Synchronization with ADF Partial Page Rendering***.

- Methods for JavaScript console logging (also see ***17.2.3 Enabling Client Side JavaScript Tracing***)
  - **`OraFormsFaces.error(?)`** – Write a log entry to the console log decorated with an error icon. Tries to use the native window.console.info function as is available in Firefox+Firebug and Chrome. If this native function is not available the icon is just added to the message and passed on to OraFormsFaces.log. If logging is disabled (ORAFORMSFACES_LOGGING) this method is replaced with an empty stub function that does not do anything.
  - **`OraFormsFaces.group(?)`** – Writes a message to the console and opens a nested block to indent all future messages sent to the console. Call OraFormsFaces.groupEnd to close the block.
  - **`OraFormsFaces.groupEnd()`** – Closes the most recently opened block created by a call to OraFormsFaces.group.
  - **`OraFormsFaces.info(?)`** – Write a log entry to the console log decorated with an info icon. Tries to use the native window.console.info function as is available in Firefox+Firebug and Chrome. If this native function is not available the icon is just added to the message and passed on to OraFormsFaces#log. If logging is disabled (ORAFORMSFACES_LOGGING) this method is replaced with an empty stub function that does not do anything.
  - **`OraFormsFaces.log(?)`** – Write a log entry to the console log. Tries to use the native window.console.log function as is available in Firefox+Firebug and Chrome. If this native function is not available a popup window is opened where the log lines will be added to. If logging is disabled (ORAFORMSFACES_LOGGING) this is replaced with an empty stub function that does not do anything.
  - **`OraFormsFaces.warn(?)`** – Write a log entry to the console log decorated with a warning icon. Tries to use the native window.console.info function as is available in Firefox+Firebug and Chrome. If this native function is not available the icon is just added to the message and passed on to OraFormsFaces#log. If logging is disabled (ORAFORMSFACES_LOGGING) this method is replaced with an empty stub function that does not do anything.

8.4. OraFormsFaces.jq Object

OraFormsFaces.jq contains the jQuery object for the jQuery library as included with OraFormsFaces. For more details see ***8.1.1 jQuery***.

jQuery is a very extensive JavaScript framework. Including all its documentation here would be overkill. For jQuery documentation refer to http://docs.jquery.com/

8.5. OraFormsFaces.JSON JavaScript Object

As explained in ***8.1.2 JSON (JavaScript Object Notation)***, OraFormsFaces includes a JSON JavaScript library as well as a Java package and a PL/SQL package implementing the JSON standard. If available, OraFormsFaces will use the native JSON support of the browser. If the browser has no native JSON support OraFormsFaces will include the JavaScript JSON reference implementation from http://www.json.org/js.html. Whichever method is used, `OraFormsFaces.JSON` will contain a reference to the JSON object. Below is the documentation from json.org which has been adopted for renaming `JSON` to `OraFormsFaces.JSON`:

- **`OraFormsFaces.JSON.stringify(value, replacer, space)`**

  value — any JavaScript value, usually an object or array.

  replacer — an optional parameter that determines how object values are stringified for objects. It can be a function or an array of strings.

  space — an optional parameter that specifies the indentation of nested structures. If it is omitted, the text will be packed without extra whitespace. If it is a number, it will specify the number of spaces to

indent at each level. If it is a string (such as '\t' or ' '), it contains the characters used to indent at each level.

This method produces a JSON text from a JavaScript value.

When an object value is found, if the object contains a `toJSON` method, its `toJSON` method will be called and the result will be stringified. A `toJSON` method does not serialize: it returns the value represented by the name/value pair that should be serialized, or `undefined` if nothing should be serialized. The `toJSON` method will be passed the key associated with the value, and this will be bound to the value

For example, this would serialize Dates as ISO strings.

```
Date.prototype.toJSON = function (key) {
    function f(n) {
        // Format integers to have at least two digits.
        return n < 10 ? '0' + n : n;
    }
    return this.getUTCFullYear()   + '-' +
         f(this.getUTCMonth() + 1) + '-' +
         f(this.getUTCDate())      + 'T' +
         f(this.getUTCHours())     + ':' +
         f(this.getUTCMinutes())   + ':' +
         f(this.getUTCSeconds())   + 'Z';
};
```

You can provide an optional replacer method. It will be passed the key and value of each member, with this bound to the containing object. The value that is returned from your method will be serialized. If your method returns `undefined`, then the member will be excluded from the serialization.

If the replacer parameter is an array of strings, then it will be used to select the members to be serialized. It filters the results such that only members with keys listed in the replacer array are stringified.

Values that do not have JSON representations, such as `undefined` or functions, will not be serialized. Such values in objects will be dropped; in arrays they will be replaced with null. You can use a replacer function to replace those with JSON values.
`OraFormsFaces.JSON.stringify(undefined)` returns undefined.

The optional space parameter produces a stringification of the value that is filled with line breaks and indentation to make it easier to read.

If the space parameter is a non-empty string, then that string will be used for indentation. If the space parameter is a number, then the indentation will be that many spaces.

Example:

```
text = OraFormsFaces.JSON.stringify(['e', {pluribus: 'unum'}]);
// text is '["e",{"pluribus":"unum"}]'

text = OraFormsFaces.JSON.stringify(['e', {pluribus: 'unum'}], null, '\t');
// text is '[\n\t"e",\n\t{\n\t\t"pluribus": "unum"\n\t}\n]'

text = OraFormsFaces.JSON.stringify([new Date()], function (key, value) {
    return this[key] instanceof Date ? 'Date(' + this[key] + ')' : value;
});
// text is '["Date(---current time---)"]'
```

- **`OraFormsFaces.JSON.parse(text, reviver)`**
  This method parses a JSON text to produce an object or array. It can throw a `SyntaxError` exception.

  The optional reviver parameter is a function that can filter and transform the results. It receives each of the keys and values, and its return value is used instead of the original value. If it returns what it received, then the structure is not modified. If it returns `undefined` then the member is deleted.

  Example:

```
// Parse the text. Values that look like ISO date strings will
// be converted to Date objects.
myData = OraFormsFaces.JSON.parse(text, function (key, value) {
  var a;
  if (typeof value === 'string') {
    a = /^(\d{4})-(\d{2})-(\d{2})T(\d{2}):(\d{2}):(\d{2}(?:\.\d*)?)Z$/.exec(value);
    if (a) {
      return new Date(Date.UTC(+a[1], +a[2] - 1, +a[3], +a[4],
                      +a[5], +a[6]));
    }
  }
  return value;
});

myData = JSON.parse('["Date(09/09/2001)"]', function (key, value) {
  var d;
  if (typeof value === 'string' &&
      value.slice(0, 5) === 'Date(' &&
      value.slice(-1) === ')') {
    d = new Date(value.slice(5, -1));
    if (d) {
      return d;
    }
  }
  return value;
});
```

Note:	The `OraFormsFaces.JSON.stringify` and `OraFormsFaces.JSON.parse` methods can be very convenient when passing data between JavaScript and Forms PL/SQL. These integrations can only transfer simple strings. By using `stringify` you can convert any complex object into a simple string. Since JSON is supported in all OraFormsFaces technologies, you can use the `parse` method in the other technology to parse the string back to a complex data structure.

8.6. OraFormsFacesForm JavaScript Class

`OraFormsFacesForm` is a JavaScript representation of an OraFormsFaces JSF component instance. It has a number of utility methods to interact with the JSF component and its associated applet. You typically get a reference to an `OraFormsFacesForm` by calling `OraFormsFaces.findForm`.

> **Note:** This Developer's Guide only contains the public field and methods that are available to your custom JavaScript. OraFormsFaces contains a lot more field and methods that have been marked as private and are not intended for use outside of OraFormsFaces' own code. Private fields and methods can change in newer versions without notice or backward compatibility. Both the public and private field and methods are documented in the JSDoc that ships with OraFormsFaces. This documentation is available at `JDEV_HOME/jdev/extensions/com.commit_consulting.oraformsfaces.v????/doc`

8.6.1. Fields

- **`OraFormsFacesForm.id`** – The (immutable) fully qualified JSF component ID of this component.
- **`OraFormsFacesForm.jq`** – Reference to the jQuery object created specifically for OraFormsFaces. See `OraFormsFaces.jq`.
- **`OraFormsFacesForm.$autoclipbottom`** – JSF component property `AutoClipBottom`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$autocliptop`** – JSF component property `AutoClipTop`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$autosize`** – JSF component property `AutoSize`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$clipapplet`** – JSF component property `ClipApplet`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$clipbottom`** – JSF component property `ClipBottom`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$clipleft`** – JSF component property `ClipLeft`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$clipright`** – JSF component property `ClipRight`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$cliptop`** – JSF component property `ClipTop`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$height`** – JSF component property `Height`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.
- **`OraFormsFacesForm.$width`** – JSF component property `Width`. Be sure to call `syncVisual` after changing this property to ensure any necessary updates of the HTML DOM.

8.6.2. Methods

- **`OraFormsFacesForm.getApplet()`** – Returns the HTML element for the Forms applet.
- **`OraFormsFacesForm.getCommand(commandId)`** – Returns a child OraFormsFacesCommand object based on the supplied id. Also see ***8.8 OraFormsFacesCommand JavaScript Class***.
- **`OraFormsFacesForm.getParameter(paramId)`** – Returns a child OraFormsFacesParameter object based on the supplied id. Also see ***8.8 OraFormsFacesCommand JavaScript Class***.

- **`OraFormsFacesForm.getParameterCount()`** – Gets the number of parameters for the Form. This function is available in Forms PL/SQL as `offParams.getParamsCount`.
- **`OraFormsFacesForm.init()`** – Initialize the `OraFormsFacesForm` object and injects the initial HTML into the document DOM. This includes the (animated) image to cover the applet area during initialization, all the necessary `<div>` elements and the (optional) development controls. The applet itself is not injected until the document and window are completely finished loading.
- **`OraFormsFacesForm.repaint()`** – Forces the web browser to repaint the applet by shifting it 1px and then shifting it back.
- **`OraFormsFacesForm.sendMessage(event, payload)`** – Sends an event with associated payload to the Forms applet by calling the public `raiseEvent` method. This triggers the Forms PL/SQL code in OraFormsFaces to handle the event. Also see ***7 Invoking PL/SQL Events from JavaScript***.
- **`OraFormsFacesForm.sendReturningMessage(event, payload, callback)`** – Sends an event with associated payload to the Forms applet by calling the public `raiseEvent` method, expecting a result to come back from the applet which is then passed to the callback function. This triggers the Forms PL/SQL code in OraFormsFaces to handle the event. Also see ***7.4 PL/SQL Events Returning Values to JavaScript***.
- **`OraFormsFacesForm.syncVisual()`** – Set all the visual properties (like size, position, overflow, borders, development controls, etc) of the HTML elements to be in sync with the properties of the `OraFormsFacesForm` JavaScript object. This method must be called after changing any of the visual related properties of the object.
- Methods for JavaScript console logging (also see ***17.2.3 Enabling Client Side JavaScript Tracing***)
  - **`OraFormsFacesForm.error()`** – see `OraFormsFaces.error`
  - **`OraFormsFacesForm.group()`** – see `OraFormsFaces.group`
  - **`OraFormsFacesForm.groupEnd()`** – see `OraFormsFaces.groupEnd`
  - **`OraFormsFacesForm.info()`** – see `OraFormsFaces.info`
  - **`OraFormsFacesForm.log()`** – see `OraFormsFaces.log`
  - **`OraFormsFacesForm.warn()`** – see `OraFormsFaces.warn`

8.7. OraFormsFacesParameter JavaScript Class

`OraFormsFacesParameter` is a JavaScript representation of an OraFormsFaces JSF component instance. It has a number of utility methods to interact with the JSF component and its associated client side representation. You typically get a reference to an `OraFormsFacesParameter` by calling `OraFormsFacesForm.getParameter`.

Note:	This Developer's Guide only contains the public field and methods that are available to your custom JavaScript. OraFormsFaces contains a lot more field and methods that have been marked as private and are not intended for use outside of OraFormsFaces' own code. Private fields and methods can change in newer versions without notice or backward compatibility. Both the public and private field and methods are documented in the JSDoc that ships with OraFormsFaces. This documentation is available at `JDEV_HOME/jdev/extensions/com.commit_consulting.oraformsfaces.v????/doc`

8.7.1. Fields

- **`OraFormsFacesParameter.formId`** – Immutable ID of the `OraFormsFacesForm` component this parameter belongs to.
- **`OraFormsFacesParameter.formparam`** – Boolean flag indicating if this parameter should be passed as a Forms parameter to the called form.
- **`OraFormsFacesParameter.formparamName`** – JSF component property `FormParameterName` indicating the name of a Forms parameter to be set when calling the parent form.
- **`OraFormsFacesParameter.global`** – Boolean flag indicating if this parameter should be passed to the called form as a PL/SQL global.
- **`OraFormsFacesParameter.globalName`** – JSF component property `GlobalName` indicating the name of a Forms PL/SQL global to be set before calling the parent form.
- **`OraFormsFacesParameter.id`** – The (immutable) fully qualified JSF component ID of this component.
- **`OraFormsFacesParameter.value`** – The value of this `OraFormsFacesParameter`. Its initial value is set by the JSF component Value property. Do not change this value property directly but use the `setValue` method which not only changes this property but also makes the necessary DOM changes to submit the changed value back to the JSF server.

8.7.2. Methods

- **`OraFormsFacesParameter.setValue(newValue)`** – Sets the value of the parameter. The value is added as a (hidden) input element to the same HTML form as the `OraFormsFacesForm` component. This means that the next form/page submit will send the value to the JSF server for server side processing. Also see ***5.5 Passing Values from Oracle Forms PL/SQL to JSF***.
- Methods for JavaScript console logging (also see ***17.2.3 Enabling Client Side JavaScript Tracing***)
  - **`OraFormsFacesParameter.error()`** – see `OraFormsFaces.error`
  - **`OraFormsFacesParameter.group()`** – see `OraFormsFaces.group`
  - **`OraFormsFacesParameter.groupEnd()`** – see `OraFormsFaces.groupEnd`
  - **`OraFormsFacesParameter.info()`** – see `OraFormsFaces.info`
  - **`OraFormsFacesParameter.log()`** – see `OraFormsFaces.log`
  - **`OraFormsFacesParameter.warn()`** – see `OraFormsFaces.warn`

8.8. OraFormsFacesCommand JavaScript Class

`OraFormsFacesCommand` is a JavaScript representation of an OraFormsFaces JSF component instance. It has a number of utility methods to interact with the JSF component and its associated client side representation. You typically get a reference to an `OraFormsFacesCommand` by calling `OraFormsFacesForm.getCommand`.

> **Note:** This Developer's Guide only contains the public field and methods that are available to your custom JavaScript. OraFormsFaces contains a lot more field and methods that have been marked as private and are not intended for use outside of OraFormsFaces' own code. Private fields and methods can change in newer versions without notice or backward compatibility. Both the public and private field and methods are documented in the JSDoc that ships with OraFormsFaces. This documentation is available at `JDEV_HOME/jdev/extensions/com.commit_consulting.oraformsfaces.v????/doc`

8.8.1. Fields

- **`OraFormsFacesCommand.formId`** – Immutable ID of the `OraFormsFacesForm` component this command belongs to.
- **`OraFormsFacesCommand.id`** – The (immutable) fully qualified JSF component ID of this component.

8.8.2. Methods

- **`OraFormsFacesCommand.invoke()`** – Invoke the `OraFormsFacesCommand`. This will submit the page back to the JSF server so the JSF component can detect it being invoked and can fire server side actions and actionlisteners. Also see ***6 Invoking JSF Commands from Forms PL/SQL***.
- Methods for JavaScript console logging (also see ***17.2.3 Enabling Client Side JavaScript Tracing***)
  - **`OraFormsFacesCommand.error()`** – see `OraFormsFaces.error`
  - **`OraFormsFacesCommand.group()`** – see `OraFormsFaces.group`
  - **`OraFormsFacesCommand.groupEnd()`** – see `OraFormsFaces.groupEnd`
  - **`OraFormsFacesCommand.info()`** – see `OraFormsFaces.info`
  - **`OraFormsFacesCommand.log()`** – see `OraFormsFaces.log`
  - **`OraFormsFacesCommand.warn()`** – see `OraFormsFaces.warn`

8.9. Deprecated JavaScript Functions

OraFormsFaces also has a number of deprecated JavaScript function for backward compatibility. These functions should no longer be called and may be removed in future versions. To allow for a transition period, these functions currently still exist and will try to call a supported JavaScript function with the same or similar functionality. Here is a list of the included but deprecated functions:

- **`offEscape(txt)`** – You should now use OraFormsFaces.JSON.stringify
- **`offExecCommand(componentId, commandId)`** – You should now use OraFormsFacesCommand.invoke
- **`offFormsDown(formsURL)`**
- **`offGetAutoClipBottom(componentId)`** – You should now use OraFormsFacesForm.$autoclipbottom
- **`offGetAutoClipTop(componentId)`** – You should now use OraFormsFacesForm.$autocliptop
- **`offGetFormModuleName(componentId)`** – You should now use `OraFormsFacesForm.getFormModule`
- **`offGetFormsApplet(componentId)`** – You should now use `OraFormsFacesForm.getApplet`
- **`offGetFormsParamName(componentId, paramNumber)`** – You should now use `OraFormsFacesParameter.formParamName`
- **`offGetGlobalName(componentId, paramNumber)`** – You should now use `OraFormsFacesParameter.globalName`
- **`offGetHtmlForm(componentId)`** – You should now use `OraFormsFacesForm.getHTMLForm`
- **`offGetParamByName(componentId, paramName)`** – You should now use `OraFormsFacesParameter.value`
- **`offGetParamByNumber(componentId, paramNumber)`** – You should now use `OraFormsFacesParameter.value`
- **`offGetParameterCount(componentId)`** – You should now use `OraFormsFacesForm.getParameterCount`
- **`offGetParamId(componentId, paramNumber)`** – You should now use `OraFormsFacesParameter.id`
- **`offGetParamInfo(componentId)`** – You should now use `OraFormsFacesForm.getParameterInfo`
- **`offHideSplash(componentId)`** – You should now use `OraFormsFacesForm.hideSplash`

- **offIsAutoSize(componentId)** – You should now use `OraFormsFacesForm.$autosize`
- **offIsClipping(componentId)** – You should now use `OraFormsFacesForm.$clipApplet`
- **offIsFormsParam(componentId, paramNumber)** – You should now use `OraFormsFacesParameter.formparam`
- **offIsParamGlobal(componentId, paramNumber)** – You should now use `OraFormsFacesParameter.global`
- **offSetParamByName(componentId, paramName, paramValue)** – You should now use `OraFormsFacesParameter.setValue`
- **offSetRuntimeSizes(componentId, windowWidth, windowHeight)** – You should now use `OraFormsFacesForm.setRuntimeSizes`
- **offShowSplash(componentId, url)** – You should now use `OraFormsFacesForm.showSplash`
- **offSimulateClick(elementId)** – You should now use `OraFormsFaces.triggerClick`
- **OraFormsFacesMessageToForms(componentId, event, payload)** – You should now use `OraFormsFacesForm.sendMessage`

9. Supplied Oracle Forms Files

9.1. off_lib.pll PL/SQL Library

`off_lib.pll` is the main PL/SQL library that comes with OraFormsFaces. It implements all of the internal OraFormsFaces PL/SQL code into a number of packages. These packages tend to be well documented in the original source files, so feel free to open the PLL file in Forms Builder and have a look around.

Try to ***not make any changes*** to the PL/SQL packages, ***except for the offCustom*** package. The offCustom package is the only point where users are expected to extend the OraFormsFaces framework. Changes to other packages will be lost when upgrading OraFormsFaces to a subsequent version. Procedures, functions and even packages can change significantly or even disappear in future versions although we try to keep OraFormsFaces backwards compatible.

9.1.1. Event Hooks and Extension Points in offCustom PL/SQL Package

The `offCustom` package is the only package in the `off_lib.pll` library that is intended to be changed by developers using OraFormsFaces. It contains a number of procedures and functions that are called by the OraFormsFaces framework. Changing the implementation of these procedures and functions allows a developer to control, change or replace behavior of OraFormsFaces.

- **`procedure handleGlobalEvent(eventName in varchar2)`**
  This procedure is called by the OraFormsFaces with two possible parameters. It is called with a parameter `initapplet` when the applet is first started. This event does not fire when the same applet is being reused on a subsequent page. This procedure is also called with a parameter `prestartform` each time the applet is being started or reused right before the CALL_FORM is used to start the requested Form module. Both events can be used for initialization of globals, record groups, database state or any other things that might be required by the application.
- **`function handleEvent(eventName in varchar2, payload in varchar2) return boolean`**
  This function is called by OraFormsFaces when JavaScript sends an event to the applet. A Developer can change this function to handle custom events or predefined OraFormsFaces events before the default handling of OraFormsFaces tries to handle an incoming event. See ***7.3 Handling Events in PL/SQL*** for more details.
- **`function handleEventCallback(eventName in varchar2, payload in varchar2, result in out varchar2) return boolean`**
  This function is called by OraFormsFaces when JavaScript sends an event to the applet which is supposed to return a value. A Developer can change this function to handle custom events or predefined OraFormsFaces events before the default handling of OraFormsFaces tries to handle the incoming event. See ***7.4 PL/SQL Events Returning Values to JavaScript*** for more details.
- **`function shouldFormClose(prevForm in offJSON.JSON, prevParams in offParams.typeParams, newForm in offJSON.JSON, newParams in offParams.typeParams) return boolean`**
  This function gets called by OraFormsFaces when an applet instance is being resumed from the legacy lifecycle cache. The default handling of OraFormsFaces is to close all open forms which returns focus to the initial OFF_LAND form. This landing form than calls the requested Forms module. By overriding this function, a developer can control whether the current form should be closed or not. The function gets all properties of the JSF Form component as well as all the properties of associated JSF FormParameter components. The function gets these values both from the previous page where this applet instance was used as well as the values from the new page where this applet instance is being reused. This allows the developer to decide whether or not the form should be closed.

9.1.2. offGlobal PL/SQL Package

The `offGlobal` package contains a number of procedures and functions to set global OraFormsFaces variables. This is information that should live longer than the duration of a single form; hence their information is stored as global variables. The entire offGlobal package is considered private to the OraFormsFaces framework and is not intended to be used directly by developers.

9.1.3. offInterface PL/SQL Package

The `offInterface` package contains procedures and functions to let the server side Oracle Forms PL/SQL code interact with the client side Forms applet and HTML page.

Note that the actual `offInterface` package contains more functions and procedures than documented here. The Developer's Guide lists only the ones that are intended for use by other developers and ignores the ones that are intended for internal OraFormsFaces use only.

- **`DIMENSION_WIDTH`** and **`DIMENSION_HEIGHT`**
  Constants which can be used as argument value when calling `setSize`.
- **`SIDE_LEFT`**, **`SIDE_TOP`**, **`SIDE_RIGHT`** and **`SIDE_BOTTOM`**
  Constants which can be used as argument value when calling `setClipping`.
- **`PLL_VERSION`**
  OraFormsFaces version number (used to compare with other components of OraFormsFaces)
- **`procedure execJSFCommand(commandId in varchar2)`**
  Invoke a `FormCommand` component which should be a child of the current form. See ***6 Invoking JSF Commands from Forms PL/SQL*** for more details.
- **`procedure popMessage(msg in varchar2)`**
  Show a message in a Forms alert by writing a message to the status bar and immediately sending another (blank) message to the status bar forcing the first message to be pushed into a modal Forms alert.
- **`procedure setSize(dimension in binary_integer, value in binary_integer)`**
  Change the size of the Form applet at runtime. This is similar to setting the `Width` and `Height` properties of the OraFormsFaces Form JSF component but it allows you to determine the dimensions at runtime from PL/SQL. See ***12 Visual Integration*** for more information on sizing and clipping.
- **`procedure setClipping(side in binary_integer, value in binary_integer)`**
  Change the amount of clipping of the Forms applet at runtime. This is similar to setting the clipping properties of the OraFormsFaces Form JSF component but it allows you to determine the clipping at runtime from PL/SQL. See ***12 Visual Integration*** for more information on sizing and clipping.
- **`function isOraFormsFaces return boolean`**
  Returns a boolean to indicate if the current Forms session is running in an OraFormsFaces applet. This is typically used to build conditional code in forms that have to run in an OraFormsFaces environment and in a normal Oracle Forms environment without OraFormsFaces. By testing for this boolean you can include code in your form that only executes when in or out of OraFormsFaces
- **`function getComponentId return varchar2`**
  Gets the JSF component ID of the OraFormsFaces Form component used to run the current applet instance.
- **`function getFormsVersion return number`**
  Get the version of Oracle Forms as a single number. In this number, each component of the version string uses two digits and the total length is always 10 digits, for instance `1001020200` for version `10.1.2.2.0` and `1101010200` for version `11.1.1.2.0`. A single number can easily be compared with constants (e.g. `offInterface.getFormsVersion>=1100000000`)

- **`procedure triggerClick(selector in varchar2)`**
  Triggers the click event on selected HTML elements in the page. This can be used to simulate a user clicking on an element. This enables you to trigger all bound events on a clickable element, such as an Oracle ADF button with a partial-submit. The actual handling for such a click is in ADF JavaScript. See the `OraFormsFaces.triggerClick` JavaScript function it invokes for more details.
- **`function buildSafeJsForm return varchar2`**
  Builds a JavaScript expression that returns the OraFormsFacesForm JavaScript object (defined in oraformsfaces.js) for the current applet instance.This can be used as the basis for building a JavaScript expression to call one of the JavaScript methods that comes with OraFormsFaces.

9.1.4. offJavaScript PL/SQL Package

The `offJavaScript` package contains functions and procedures to execute JavaScript in the browser. This package is available in all Oracle Forms version. When using Forms 11 or later, it will use the native JavaScript API that comes with Forms. With older versions of Oracle Forms, OraFormsFaces will use its own JavaScript API offering the same capabilities. If all JavaScript is executed through this package no changes to the code are necessary when upgrading Forms 10g to Forms 11g.

- **`JAVASCRIPT_ERROR`**
  Exception that gets thrown by jsEval and jsExec when a JavaScript error occurs.
- **`function jsEval(expression in varchar2) return varchar2`**
  Evaluate (execute) a JavaScript expression in the current browser window and get the result back as a string. Internally, this function forces an immediate roundtrip from the Forms server to the client applet, since the function is getting the result back.
- **`procedure jsExec(expression in varchar2)`**
  Evaluate (execute) a JavaScript expression in the current browser window without getting the result string.
- **`procedure jsAlert(msg in varchar2, asynchronous in boolean := true)`**
  Show a JavaScript at the client using the JavaScript alert function.
- **`procedure writeConsole(msg in varchar2)`**
  Write a string to the client java console. This can be used for debugging. Be warned that this procedure forces an immediate roundtrip from the Forms server to the client applet, which might impact performance if used many times.
- **`function getLastError return varchar2`**
  Gets the text of the last occurred error during JavaScript execution. Call this after `jsEval` or `jsExec` threw a `JAVASCRIPT_ERROR`

9.1.5. offJSON PL/SQL Package

The `offJSON` package contains functions and procedures to construct and inspect a JSON object. See **8.1.2 JSON (JavaScript Object Notation)** for more information on JSON. A JSON object can be constructed from scratch with the procedures in this package or it can be created by parsing a JSON string with the `offJSONParser` package. A JSON object can also be converted to a string with the `offJSONParser`.

- **`JSON`**
  A type used to hold all information for a JSON object. It is not intended for direct usage, but is merely used as return type and parameter type for functions and procedures in this and other packages.
- **Functions to query a JSON type**
  These functions can be used to inspect which data type is contained in the generic JSON type. Once the type is known any of the remaining functions in this package can be used to get the actual content.
  - `function isNull   (json in JSON) return boolean`
  - `function isString (json in JSON) return boolean`

- `function isNumber (json in JSON) return boolean`
- `function isBoolean(json in JSON) return boolean`
- `function isObject (json in JSON) return boolean`
- `function isArray  (json in JSON) return boolean`

- **Functions to get the value from a JSON type**

  These functions get the value contained in the JSON type. Using a function which is not appropriate for the JSON type will show a message and throw a `no_data_found`.

  - `function getString (json in JSON) return varchar2`
  - `function getNumber (json in JSON) return number`
  - `function getBoolean (json in JSON) return boolean`
  - `function forceString(json in JSON) return varchar2`
    This gets the value contained in the JSON type as a string independent of the contained data type. Strings will simply be returned, numbers will be returned after applying a to_char, nulls will return null and booleans,objects and arrays will return the result of offJSONParser.stringify.

- **Function and procedures to handle JSON objects**

  These functions only apply to JSON objects. Calling them for simple values or arrays will raise a `no_data_found`. Be sure to check with `isObject` first.

  - `function hasProperty (obj in JSON, prop in varchar2) return boolean`
    Returns a flag indicating if the object contains the requested property.
  - `function getProperty (obj in JSON, prop in varchar2) return JSON`
    Returns the value of the property as another JSON type. Use the other functions in this package to determine its type or to get the contained value.
  - `procedure setProperty (obj in out nocopy JSON, prop in varchar2, newVal in JSON)`
    Sets the value of a property. The new value has to be passed as a JSON type which can be constructed with the other procedures in this package.
  - `procedure removeProperty(obj in out nocopy JSON, prop in varchar2)`
    Remove a property from an object.

- **Functions and procedures to handle JSON arrays**

  These functions only apply to JSON arrays. Calling them for simple values or objects will raise a `no_data_found`. Be sure to check with `isArray` first.

  - `function getSize (arr in JSON) return pls_integer`
    Returns the number of elements in the array.
  - `function getElement (arr in JSON, elem in pls_integer) return JSON`
    Returns the requested element in the array as another JSON type. Use the other functions in this package to determine its type or to get the contained value. The first element of the array has index 1, not 0 as in Java or some other programming languages.
  - `procedure addElement (arr in out nocopy JSON, elem in JSON)`
    Append an element to the array. The new value has to be passed as a JSON type which can be constructed with the other procedures in this package.
  - `procedure removeElement (arr in out nocopy JSON, idx in pls_integer)`
    Removes an element from the array. The first element of the array has index 1, not 0 as in Java or some other programming languages.
  - `procedure setElement (arr in out nocopy JSON, idx in pls_integer, elem in JSON)`
    Replace an element in the array. The new value has to be passed as a JSON type which can be constructed with the other procedures in this package. The first element of the array has index 1, not 0 as in Java or some other programming languages.

- **Functions to construct JSON types**
  These functions create a new JSON type for the requested data type. Strings, numbers and booleans are already initialized with the requested value, whereas returned objects and arrays are empty and can be filled with the other procedures in this package.
  - `function newNull return JSON`
  - `function newString (val in varchar2) return JSON`
  - `function newNumber (val in number) return JSON`
  - `function newBoolean(val in boolean) return JSON`
  - `function newObject return JSON`
  - `function newArray return JSON`

9.1.6. offJSONParser PL/SQL Package

The `offJSONParser` package contains functions to parse a JSON string to JSON objects or to stringify JSON objects to a JSON string. JSON objects can be inspected, created or changed using the `offJSON` package.

- **`PARSE_ERROR`**
  Exception that gets thrown when a string cannot be parsed as it does not appear to confirm to JSON standards.
- **`function stringify(json in offJSON.JSON) return varchar2`**
  Converts a JSON type to a JSON string whatever data type is contained in the JSON type.
- **`function stringify(val in number) return varchar2`**
  Converts a PL/SQL number to a JSON string.
- **`function stringify(bool in boolean) return varchar2`**
  Converts a PL/SQL boolean to a JSON string (true or false)
- **`function stringify(txt in varchar2) return varchar2`**
  Converts a PL/SQL varchar to a JSON string escaping any special characters and enclosing the string in quotes.
- **`function parse(jsonString in varchar2) return offJSON.JSON`**
  Parse a JSON string to a JSON data structure which can be inspected and changed using the `offJSON` package.

9.1.7. offParams PL/SQL Package

The `offParams` package contains functions and procedures to interact with `FormParameter` JSF components. See *5 Passing Parameters* for more information on setting up `FormParameters`.

> **Note:** This Developer's Guide only contains the public procedure and functions that are available to your custom PL/SQL code. The package contains more procedures and functions that are intended for internal OraFormsFaces use only. These private functions and procedures can change in newer versions without notice or backward compatibility. Both the public and private procedures and functions are documented in the PL/SQL source code of the package specification in the PLL file.

- **`typeParams`**
  A type used to hold all information for all defined `FormParameters`. This nested data structured can be retrieved with a single call to `getParameters` and is used as an input parameter to most other functions and procedures.

- **`function getParameters return typeParams`**
  Returns all the information about all defined FormParameters in a single call. This requires a roundtrip from the Forms server to the client as client side JavaScript is required. Retrieving all information in a single call is most efficient.
- **`function getParamsCount(params in typeParams) return binary_integer`**
  Returns the number of defined parameters.
- **`function getParamIndex(params in typeParams, paramName in varchar2) return binary_integer`**
  Gets the index (starting at 1) of a parameter based on its parameter JSF component ID.
- **`function getParamName(params in typeParams, paramIndex in binary_integer) return varchar2`**
  Returns the name of a parameters based on its index (starting at 1).
- **`function getParamValue(params in typeParams, paramName in varchar2) return varchar2`**
  Gets the value of a parameter.
- **`procedure setParamValue(paramName in varchar2, paramValue in varchar2)`**
  Sets the value of a parameter which will be transferred to the JSF server on the next page submit. See ***5.5 Passing Values from Oracle Forms PL/SQL to JSF*** for more details.

9.1.8. offTriggers PL/SQL Package

The `offTriggers` package contains functions and procedures related to handling of certain Oracle Forms triggers.

- **`procedure whenCustomJavascriptEvent`**
  This procedure is called from the `when-custom-javascript-event` trigger in Forms 11 or the `when-custom-item-event` trigger on the PJC in Forms 10gR2. This handles any events (and payloads) being sent to the OraFormsFaces applet through the JavaScript API. This procedure is not intended to be called in any other way than by these OraFormsFaces triggers. See ***7 Invoking PL/SQL Events from JavaScript*** for more information on event handling.
- **`procedure whenCustomJavascriptEventLand`**
  This procedure is similar to the `whenCustomJavascriptEvent` procedure as it is responsible for handling incoming events from JavaScript. However, this procedure is only called by the initial `OFF_LAND` landing form as it requires other event handling than normal forms.
- **`procedure whenNewFormInstanceLand`**
  This procedure is called from the `when-new-form-instance` trigger of the initial `OFF_LAND` landing form and initiates an OraFormsFaces session. It is not intended to be called in any other way than by this OraFormsFaces trigger.
- **`procedure onLogon(logonScreenOnError in boolean default true)`**
  This procedure is called from the `on-logon` trigger of the initial `OFF_LAND` landing form. Depending on the `userid` parameter it will either prompt the user for credentials (like normal Oracle Forms), use the specified fixed or SingleSignOn credentials (like normal Forms) or use an OraFormsFaces specific method of logging on as described in ***14.2 Database Sessions and Credentials***.
- **`procedure onError`**
  This procedure can be called from an `on-error` trigger in custom forms. It will determine the error number and text and throw a modal JavaScript alert showing this information. This is especially useful in situations where the Forms status bar is removed from the visual area as described in ***12.5 Error and Message Handling with Applet Clipping***.

- **`procedure onMessage`**
  This procedure can be called from an `on-message` trigger in custom forms. It will determine the message number and text and throw a modal JavaScript alert showing this information. This is especially useful in situations where the Forms status bar is removed from the visual area as described in ***12.5 Error and Message Handling with Applet Clipping***.
- **`function getEventName return varchar2`**
  This returns the name of the event currently being handled.
- **`function getEventPayload return varchar2`**
  This returns the payload of the event currently being handled.
- **`procedure checkSuccess`**
  Convenience procedure that checks if the last Forms built-in completed successfully (by inspecting `form_success`). If the last built-in was not successful, this procedure will throw a `form_trigger_failure`. This procedure should be called after calling a Forms built-in and if you want to abort the processing on a failure of that built-in.

9.2. off_jscript.pll PL/SQL Library

The `off_jscript.pll` PL/SQL library contains the implementation of the JavaScript API. Oracle Forms version 11 introduced a native JavaScript API that is used internally by OraFormsFaces. In previous version of Oracle Forms, OraFormsFaces uses its own JavaScript API. This is why a different version of `off_jscript.pll` is included for both versions of Oracle Forms. Both these libraries implement the same interface, so the main `off_lib.pll` can remain independent of the Oracle Forms version being used.

9.2.1. offJavaScriptImpl PL/SQL Package

> **Note:** This package is the internal implementation of the JavaScript API and should never be called directly from custom code. Instead, use the `offJavaScript` package from `off_lib.pll` which will defer actual execution to the `offJavaScriptImpl` package appropriate for your Oracle Forms version.

- **`function jsEval(expression in varchar2) return varchar2`**
  Executes a JavaScript expression and returns the result of the evaluated expression. Do not call directly but use `offJavaScript.jsEval` or `offJavaScript.jsExec` instead.
- **`procedure asyncAlert(jsonMsg in varchar2)`**
  Will show a JavaScript alert with this message without waiting for the dialog to be dismissed by the user. This is especially useful in Forms 10gR2 and earlier since having a modal JavaScript dialog will suspend the normal Forms heartbeat message which can result in a session timeout at the Forms server. Do not call directly but use `offJavaScript.jsAlert` instead.
- **`procedure writeConsole(msg in varchar2)`**
  Write a message to the Java console. Do not call directly but use `offJavaScript.writeConsole` instead.

9.3. oraformsfaces.olb Object Library

The oraformsfaces.olb Object Library contains two object groups that can be subclassed into Oracle Forms modules. OraFormsFaces deliberately uses a mechanism based on subclassed object groups from an object library. This allows OraFormsFaces to change the content of these object groups in newer versions of OraFormsFaces. Since these object groups are subclassed as a whole all changes to the object groups will automatically apply to all Forms using these object groups as soon as these Forms are recompiled.

9.3.1. OraFormsFaces Object Group

OraFormsFaces is the main object group in this object library. It must be subclassed into each and every Form being used in an OraFormsFaces context. See ***4.2 Preparing the Oracle Forms Modules*** for more information on preparing each Forms module for use in OraFormsFaces.

For Oracle Forms 11 this object group only contains a record group indicating the OraFormsFaces version and a single system trigger to capture incoming events from JavaScript. For Oracle Forms version 10gR2 this object group contains the same record group as well as an invisible window with an invisible block containing an invisible Pluggable Java Component with a trigger capturing the same incoming events from JavaScript.

9.3.2. OraFormsFaces_Msg_Triggers Object Group

A second object group, **oraformsfaces_msg_triggers**, is available with a default implementation of the `on-error` and `on-message` triggers to push any errors and messages into a modal JavaScript alert. This is especially useful in situations where the Forms status bar is removed from the visual area as described in ***12.5 Error and Message Handling with Applet Clipping***. Most enterprise applications already have global handling of on-error and on-message triggers where similar functionality can be implemented without replacing the existing triggers with the ones from this object group.

9.4. off_land.fmb Forms Module

The `off_land` Forms Module is the OraFormsFaces landing form. OraFormsFaces tries to use a single Forms applet on each page of your application to reduce applet startup times (see ***13 Forms Java Applet Instance Reuse*** for more information). As a consequence the applet declaration and all of its parameters have to be 100% identical on all pages. This means the name of the Forms module cannot be included as a normal parameter as with vanilla Oracle Forms. To workaround this issue, OraFormsFaces will always use `off_land` as the initial Forms module. The `off_land` form will then use the JavaScript API to determine the actual form to start and use a `CALL_FORM` to activate that Form.

The `off_land` landing form also plays a crucial role in the reuse of the same applet instance on a subsequent page. When resuming the applet, the active form will be closed by invoking `exit_form`. Since the form was started using `call_form` from `off_land`, the control will return to the `off_land` landing form. It will then once again determine which form to run and start it using `CALL_FORM`. This process will continue until the browser is closed or the user manually closes the form other than during an applet suspend/resume cycle. This process is described in more detail at ***13 Forms Java Applet Instance Reuse***. This also includes way how to change this default behavior.

9.5. Other Forms Modules

OraFormsFaces also ships with a number of additional FMB files: `off_test`, `off_users`, `off_demo` and `off_demo_emp`. These files are not required for OraFormsFaces but are just included for the supplied demos or for the installation verification steps from this Developer's Guide.

10. Supplied Java Classes and JSF Components

OraFormsFaces comes with full JavaDoc of all Java classes. This JavaDoc can be found at `JDEV_HOME/jdev/extensions/com.commit_consulting.oraformsfaces.v????/doc/javadoc/`. It is also registered as part of the OraFormsFaces library in JDeveloper. This means that if the library is attached to your project (**Project Properties** > **Libraries**), you can simply access the JavaDoc through JDeveloper itself by pressing **Ctrl-minus** (JDeveloper 10g) or **Alt-Shift-minus** (JDeveloper 11g).

The most commonly used classes are also included in this Developer's Guide, but for a ***full reference please refer to the JavaDoc***.

10.1. JSF Components and their Properties

The three OraFormsFaces JSF components are implemented by classes in the `com.commit_consulting.oraformsfaces.component.component` package. Each component has getters and setters to manipulate the component properties, as well as constructors for programmatically creating these components. Constructing component or setting properties from code is only required for very advanced usages. A typical application will only declare components in JSPX pages and specify the necessary properties there.

10.1.1. Form

This class represents an OraFormsFaces `Form` JSF component. Below are descriptions for all of the component properties which can be set directly in the JSF page or programmatically using setter methods on the `Form` object. See ***4.3 Embedding Oracle Forms in a JSF page*** for more information on including a Form component in a JSF page.

- Clipping properties
  - **ClipApplet** – This property indicates if you want to visually clip the Forms applet to show only a portion of the default Oracle Forms application. The default value is false. See Visual Integration for a full description of this feature.
  - **AutoClipTop** – When clipping is enabled (**ClipApplet** is set to `true`), this property specifies which chrome to remove from the top of the Forms applet. At runtime, the OraFormsFaces enhanced Oracle Forms applet will figure out how many pixels to clip from the top of the applet to just remove these elements. The default value is **none** which doesn't clip anything from the top. Other possible values are **menu**, **toolbar**, and **window-title**. Setting the property to **menu** will remove the menu bar from the top of the applet no longer allowing users to select options from the menu. This makes the applet look and feel more like a true web application. Navigation to other pages and forms should now be handled by the JSF application. Setting the property to **toolbar** removes both the menu and the toolbar/button bar to have the Forms applet blend in even more with the surrounding web page. The **window-title** is the most extreme value. It will not only remove the menu and toolbar but will also remove the title bar at the top of the Forms window.
  - **AutoClipBottom** – When clipping is enabled (**ClipApplet** is set to `true`), this property specifies if the status bar should be removed from the bottom of the Forms applet. The default value is **none** which will not remove the status bar. Setting it to **statusbar** will remove the status bar. If the status bar is clipped and the **AutoClipTop** property is set to `window-title` and **AutoSize** is set to `true` it will even cause all window edges to be clipped to have the Forms applet fully blend with the web page. Remember that removing the status bar might also hide messages from the user that would normally appear in this status bar. You should take care that these messages remain noticeable for end users, for example by pushing them to JavaScript alerts as described in ***Error and Message Handling with Applet Clipping*** in the Visual Integration chapter.

- **ClipLeft**, **ClipTop**, **ClipRight** and **ClipBottom** – These properties specify the number of pixels to clip from the applet at each side. When using the auto-clipping feature the values from these properties are added to the automatically determined values by the auto-clipping feature. This means you can specify positive or negative values to increase/decrease the automatically determined clipping. When not using the auto-clipping feature these properties simply specify the number of pixels to clip from each side. See Visual Integration later in the guide for a full description of the clipping feature

- Core properties
  - **Id** – This is the unique ID of the JSF component within this JSF page. If you leave this property empty at design time, JSF will assign a system generated ID to the component at runtime. If you later add or remove components from the page, the generated ID can change. This is why it is best to set the ID to a fixed value at design time. You will be referring to this value from PL/SQL code and JavaScript with the more advanced integration. If you leave the ID generation to the JSF framework, the ID might change when you add or remove other components from the page, causing problems in your PL/SQL code.
  - **Rendered** – This is a default JSF property which indicates if the component should be rendered (included) in the HTML response to the client. You can set this false, to disable the component or use an Expression Language (EL) expression to render the component conditionally. For most usages, you can leave it to its default value of `true`.
- Main properties
  - **FormModuleName** – This is the name of the Forms module you want to run. You can specify the name of the Oracle Forms module with or without the `.fmx` extension. Normally you specify the name without a full directory path, since the Forms server will be setup to search in the directories where you put your FMX files. If you do not specify a value for **FormModuleName**, the default `off_test.fmx` will be used which shows the test form included with OraFormsFaces.
  - **ShowDevControls** – This property indicates if you want Development Controls at runtime. These controls allow you to change the sizing and clipping properties of the Form component at runtime in your web browser. It is not possible to show the Oracle Forms form in the JDeveloper visual editor. This is why these Development Controls have been added to change the sizing properties at runtime. Once you found appropriate values, you can set these at design time in the JSF page and disable the Development Controls. The default value is `false`. See the chapter on Visual Integration for a more detailed description and examples of the Development Controls.
  - **LoadingImage** – While the applet is starting it is covered with an animated image. This property specifies which image to show. You can specify a URL to an image to use for this feature or use one of the preset images that are included with OraFormsFaces. Set this property to ***presetX*** where ***X*** is a number from 1 through 5. The five preset image are:

Note:	If you do not want to use any of the preset images and do not have a custom image to use, there are plenty of website that can generate a custom animated GIF on the fly. Some examples are http://www.ajaxload.info/, http://www.loadinfo.net/, and http://preloaders.net/.

- **UniqueAppletKey** – Normally OraFormsFaces will try to use a single applet instance for all pages of your application during a single session. However, by specifying a unique applet key to can enforce that a page does not reuse the same applet as another page. Only OraFormsFaces Form components with the same UniqueAppletKey will be allowed to reuse the same applet instance. See ***13 Forms Java Applet Instance Reuse*** for more information.

- Other properties
  - **Binding** – This is a default JSF property which binds the OraFormsFaces Form component to a backing bean property. It will instruct the JSF framework to call the setter of the backing bean property to pass a reference to the OraFormsFaces Form component. This can be used for programmatic access to the Form component at runtime. For most usages, this property can be left to its default (empty) value which disables binding to a backing bean.
- Sizing properties
  - **AutoSize** – This enables or disabled the auto-sizing feature of the OraFormsFaces applet. By default this is set to `false` (disabled) so applications upgraded from previous OraFormsFaces without this feature work the same. If you set **AutoSize** to `true` en thus enable this feature, the OraFormsFaces enhancements to the Oracle Forms applet will figure out the sizing of the window in your Forms module at runtime and will size the applet in the web page accordingly. Each time another form is opened in the same applet, the applet will resize itself according to the size of the current form.
  - **Width** – This is the width of the component in pixels. When the auto-sizing feature is enabled this specifies the initial size of the applet which is resized once the applet finished initialization and it knows the size of the Form it is running. When no specific value is set, the default of 640 will be used.
  - **Height** – This is the height of the component in pixels. When the auto-sizing feature is enabled this specifies the initial size of the applet which is resized once the applet finished initialization and it knows the size of the Form it is running. When no specific value is set, the default of 480 will be used.

10.1.2. FormParameter

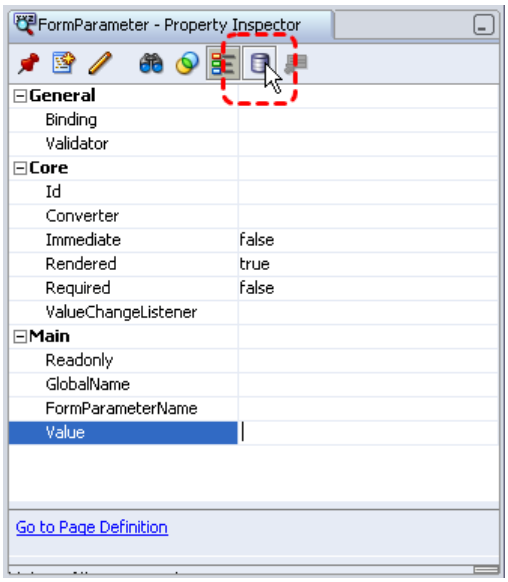
This class represents an OraFormsFaces `FormParameter` JSF component. Below are descriptions for all of the component properties which can be set directly in the JSF page or programmatically using setter methods on the `FormParameter` object. See ***5 Passing Parameters*** for more information on including a FormParameter component in a JSF page.

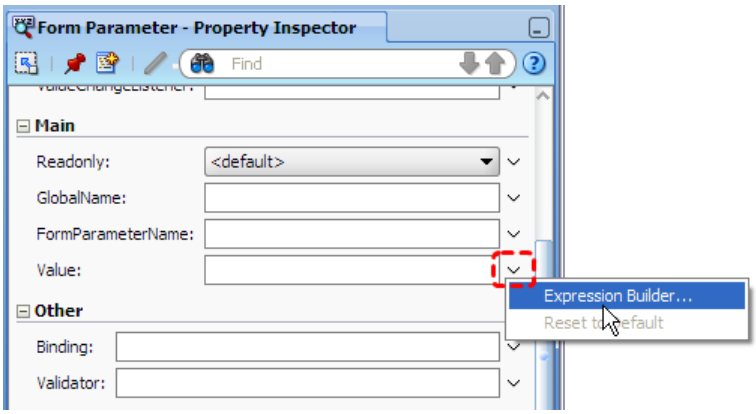
- Core properties
  - **Id** – This can be used to set the ID of the FormParameter component. This is the unique ID of the JSF component within this JSF page. If you leave this property empty, JSF will assign a system generated ID to the component at runtime. It is best to set this property to a specific value. This ID can be used to retrieve the value of the FormParameter component from Oracle Forms PL/SQL. If you leave the ID generation to the JSF framework, the ID might change when you add or remove other components from the page, causing problems in your PL/SQL code.
  - **Converter** – This is a default JSF property to specify a Converter for a component. Due to the nature of the HTTP protocol when the value of a property is transmitted to or from the client web browser it has to be represented as a string. The Converter is responsible for converting the native data type of the managed bean from the value property (e.g. an integer, date, double, or a complex custom type) to a string and to convert the submitted string back to a native data type.
  - **Immediate** – This is also a default JSF property which indicates that the submitted value of the parameter should be set to the managed bean early in the JSF page lifecycle. If you are not yet familiar with the JSF page life cycle, it is best to keep this property at its default value.

- **Rendered** – This is a default JSF property which indicates if the component should be rendered (included) in the HTML response to the client. You can set this `false`, to disable the component or use an Expression Language (EL) expression to render the component conditionally. For most usages, you can leave it to its default value of `true`.
- **Required** – This property indicates if a value is required for this parameter. Again, this is a default JSF component property for input components.
- **ValueChangeListener** – This is also a default JSF property for input components. You can specify a managed bean method to be fired whenever the value of this FormParameter component changes. This is typically when the page that embeds the Form and the FormParameter is submitted back to the JSF web server. For simple usages, this can be left empty.

- Main properties
  - **Readonly** – This boolean property indicates if the parameter is read-only. If this property is set to `true`, the parameter is seen as read-only for the JSF framework. This means that when the page is submitted back to the JSF server the value of the parameter is ignored. If the Readonly property is set to `false`, the parameter value can be changed from within Oracle Forms PL/SQL. On a subsequent page request the value is set to the bean property specified in the Expression Language (EL) expression of the FormParameter's **value** property. With the readonly property set to `true` a number of other FormParameter properties become irrelevant: **validator**, **immediate**, **required**, and **valueChangeListener**.
  - **GlobalName** – The value of any FormParameter component can be retrieved from Forms PL/SQL based on the FormParameter component ID. However it is very common for existing Forms PL/SQL code to use globals to receive external parameters. By setting this **globalName** property, OraFormsFaces will automatically create an Oracle Forms global with the specified name and the appropriate value before executing the Oracle Forms form. For instance, setting this property to `custid`, will create an Oracle Forms global named `:global.custid` with the value of this FormParameter. This can be used to integrate existing Oracle Forms modules that rely on the existence of globals. By default, this property does not have a value and OraFormsFaces will not create a Forms global in that case.
  - **FormParameterName** – The value of any FormParameter component can be retrieved from Forms PL/SQL based on the FormParameter component ID. However it is very common for existing Forms PL/SQL code to use parameters defined at design time in Forms Builder to receive external parameters. By setting this **formParameterName** property, OraFormsFaces will pass the value of this FormParameter to the Oracle Forms form as a true Oracle Forms parameter. Internally, OraFormsFaces will use a `CALL_FORM` built-in to start the specified Oracle Forms form. Before calling the form, OraFormsFaces will create an Oracle Forms parameter list and will pass this parameter list to the `CALL_FORM` built-in. For instance, setting this property to `custid` will cause OraFormsFaces to create `:parameter.custid` before calling the requested form. When using this functionality, ensure that the Oracle Forms form being called does have a parameter defined with this same name.

- **Value** – This is the actual value of the FormParameter component. You can either set this to a fixed value or use an Expression Language (EL) expression to refer to a managed bean or ADF binding property. When the value from an ADF Data Control is needed, the easiest way is to drag-and-drop the data control as described in ***5.1 Defining a FormParameter by Dragging an ADF Data Control***. If you want to get the value from an JSF Managed Bean, the easiest way is to use the EL expression builder that comes with JDeveloper:





- Other properties
  - **Binding** – This is a default JSF property which binds the OraFormsFaces FormParameter component to a backing bean property. It will instruct the JSF framework to call the setter method of the backing bean property to pass a reference to the OraFormsFaces FormParameter component. This can be used for programmatic access to the FormParameter component at runtime. For most usages, this property can be left to its default (empty) value which disables binding to a backing bean.
  - **Validator** – This can specify a JSF Validator to validate the value of the parameter. This is relevant for non-read only parameters. By specifying a validator, you can validate the value of the parameter. This is default JSF functionality for input components.

10.1.3. FormCommand

This class represents an OraFormsFaces `FormCommand` JSF component. Below are descriptions for all of the component properties which can be set directly in the JSF page or programmatically using setter methods on the `FormCommand` object. See ***6 Invoking JSF Commands from Forms PL/SQL*** for more information on including a FormCommand component in a JSF page.

- Core properties
  - **Action** – This property can have an Expression Language (EL) expression referring to a managed bean method. This method will be executed when the FormCommand is triggered. The method should return a String. The JSF framework will look in the `faces-config.xml` file for a navigation case originating from the current page and with a **from-outcome** property set to the value of the string returned from the action method. If such a navigation case is found, the JSF framework will follow this navigation case and render the page the case is pointing to.
  - **Id** – This can be used to set the ID of the FormCommand component. This is the unique ID of the JSF component within this JSF page. If you leave this property empty, JSF will assign a system generated ID to the component at runtime. It is best to set this property to a specific value. This ID will be used to trigger the FormCommand component from within Oracle Forms PL/SQL. If you leave the ID generation to the JSF framework, the ID might change when you add or remove other components from the page, causing problems in your PL/SQL code.
  - **ActionListener** – This is a default JSF property for command components. You can specify a managed bean property for this property as well. This method will also be executed when the FormCommand component is triggered. The difference with the Action property is that an ActionListener does not invoke any JSF navigation case. The method specified in the ActionListener is just executed when the FormCommand component is triggered, but it cannot influence the JSF navigation. The method can have any name, must be public, return void, and accept an ActionEvent as its only parameter.
  - **Immediate** – This is also a default JSF property which indicates that the Action and ActionListener should be executed early in the JSF page lifecycle. If you're not yet familiar with the JSF page life cycle, it is best to keep this property at its default value.
  - **Rendered** – This is a default JSF property which indicates if the component should be rendered (included) in the HTML response to the client. You can set this `false`, to disable the component or use an Expression Language (EL) expression to render the component conditionally. For most usages, you can leave it to its default value of `true`.
- Other properties
  - **Binding** – This is a default JSF property which binds the OraFormsFaces FormCommand component to a backing bean property. It will instruct the JSF framework to call the setter method of the backing bean property to pass a reference to the OraFormsFaces FormCommand component. This can be used for programmatic access to the FormCommand component at runtime. For most usages, this property can be left to its default (empty) value which disables binding to a backing bean.

10.2. Encrypted Forms Credentials Classes

The `com.commit_consulting.oraformsfaces.component.crypto` package contains a number of Java interfaces and classes for encrypting database credentials. These allow Java code at the JSF server to determine the credentials to be used for the Oracle Forms database session. A custom Java class can determine which credentials to use based on the current JSF application state like the user details used to logon to the JSF application. These database credentials are then encrypted before being transmitted to the OraFormsFaces applet. The ON-LOGON trigger of the OraFormsFaces landing form then decrypts this information at the Forms server

and uses the information to logon to the Oracle database. More details on this feature are described at ***14.2.2 Custom Java Class to Determine Database Credentials***.

10.2.1. FormsCredentialsProvider

Interface that needs to be implemented by a class providing the encrypted credentials to the `OraFormsFacesPhaseListener` when using JSF to determine the database credentials for an Oracle Forms session. The `FormsCredentialsImpl` class is a default implementation of this interface that is probably sufficient for most scenarios. Extending that class is more convenient that implementing this interface directly.

- **`String getEncryptedCredentials(FacesContext ctxt, String challenge) throws GeneralSecurityException`**
  Method that is called by the `OraFormsFacesPhaseListener` and that has to provide the encrypted credentials. The `FacesContext` can be used to determine current application state. The one-time `challenge` can be used to construct encrypted credentials that can be used only once and are only valid for this individual challenge.

10.2.2. FormsCredentialsImpl

Abstract class that implements the `FormsCredentialsProvider` interface with a default implementation that should be sufficient for most scenarios. Implementing sub classes only have to implement a few simple abstract methods.

- **`protected abstract Cipher createCipher()`**
  This method is called by the no-argument constructor and has to create a Cipher used to encrypt the credentials. To use the default AES encryption a simple implementation calling the static createCipher method suffices:

```
protected Cipher createCipher() throws GeneralSecurityException {
    return createCipher("8226B77F27119CA2A7CC2BB777CBB425");
}
```

> **Note**: Do not use the key from this example, but generate your own encryption key as described at ***11.1 KeyGenerator***.

- **`protected static Cipher createCipher(String hexKey)`**
  Convenience method that can be used to create an AES/CBC/PKCS5Padding Cipher based on the supplied key.
- **`protected abstract String getConnectString(FacesContext ctxt)`**
  Abstract method that should supply the unencrypted (plain) database connect string to be used for the Oracle Forms database connection. See the `getUsername` method for more details.
- **`String getEncryptedCredentials(FacesContext ctxt, String challenge)`**
  returns an encrypted string containing a JSON object with the database credentials to be used for an Oracle Forms database session. This method is called by the `OraFormsFacesPhaseListener` when processing an ajax request from the initializing Forms applet requesting this information.
- **`protected abstract String getPassword(FacesContext ctxt)`**
  abstract method that should supply the unencrypted (plain) database password to be used for the Oracle Forms database connection. See the `getUsername` method for more details.

- **`protected abstract String getUsername(FacesContext ctxt)`**
  abstract method that should supply the unencrypted (plain) database username to be used for the Oracle Forms database connection. This plain text information is requested by `getEncryptedCredentials` and will be encrypted before being returned to the client. This information is most likely based on information from the supplied `FacesContext` and the enclosed `ExternalContext` (`FacesContext.getExternalContext`) which included things like the user/principal and other session information.

10.3. JSON Classes

The `com.commit_consulting.org.json` package contains a number of Java classes to construct and parse JSON strings. OraFormsFaces includes libraries to generate and parse JSON from JavaScript, Java and PL/SQL. This makes JSON is an ideal data-interchange language. See ***8.1.2 JSON (JavaScript Object Notation)*** for more information on JSON.

OraFormsFaces includes the JSON reference implementation from http://www.json.org/java/. See the JavaDoc included with OraFormsFaces or the online version at http://www.json.org/java/.

10.3.1. JSONObject

A **`JSONObject`** is an unordered collection of name/value pairs. Its external form is a string wrapped in curly braces with colons between the names and values, and commas between the values and names. The internal form is an object having `get` and `opt` methods for accessing the values by name, and `put` methods for adding or replacing values by name. The values can be any of these types: `Boolean`, `JSONArray`, `JSONObject`, `Number`, `String`, or the `JSONObject.NULL` object. A `JSONObject` constructor can be used to convert an external form JSON text into an internal form whose values can be retrieved with the `get` and `opt` methods, or to convert values into a JSON text using the `put` and `toString` methods. A `get` method returns a value if one can be found, and throws an exception if one cannot be found. An `opt` method returns a default value instead of throwing an exception, and so is useful for obtaining optional values.

The generic `get()` and `opt()` methods return an object, which you can cast or query for type. There are also typed `get` and `opt` methods that do type checking and type coercion for you.

The `put` methods add values to an object. For example,

```
myString = new JSONObject().put("JSON", "Hello, World!").toString();
```

produces the string `{"JSON": "Hello, World"}`.

The texts produced by the `toString` methods strictly conform to the JSON syntax rules. The constructors are more forgiving in the texts they will accept:

- An extra `,` (comma) may appear just before the closing brace.
- Strings may be quoted with `'` (single quote).
- Strings do not need to be quoted at all if they do not begin with a quote or single quote, and if they do not contain leading or trailing spaces, and if they do not contain any of these characters: `{ } [ ] / \ : , = ; #` and if they do not look like numbers and if they are not the reserved words `true`, `false`, or `null`.
- Keys can be followed by `=` or `=>` as well as by `:` (colon).
- Values can be followed by `;` (semicolon) as well as by `,` (comma).
- Numbers may have the `0-` (octal) or `0x-` (hex) prefix.

10.3.2. JSONArray

A `JSONArray` is an ordered sequence of values. Its external text form is a string wrapped in square brackets with commas separating the values. The internal form is an object having `get` and `opt` methods for accessing the values by index, and put methods for adding or replacing values. The values can be any of these types: `Boolean`, `JSONArray`, `JSONObject`, `Number`, `String`, or the `JSONObject.NULL` object.

The constructor can convert a JSON text into a Java object. The `toString` method converts to JSON text.

A `get` method returns a value if one can be found, and throws an exception if one cannot be found. An `opt` method returns a default value instead of throwing an exception, and so is useful for obtaining optional values.

The generic `get()` and `opt()` methods return an object which you can cast or query for type. There are also typed `get` and `opt` methods that do type checking and type coercion for you.

The texts produced by the `toString` methods strictly conform to JSON syntax rules. The constructors are more forgiving in the texts they will accept:

- An extra `,` (comma) may appear just before the closing bracket.
- The `null` value will be inserted when there is `,` (comma) elision.
- Strings may be quoted with `'` (single quote).
- Strings do not need to be quoted at all if they do not begin with a quote or single quote, and if they do not contain leading or trailing spaces, and if they do not contain any of these characters: `{ } [ ] / \ : , = ; #` and if they do not look like numbers and if they are not the reserved words `true`, `false`, or `null`.
- Values can be separated by `;` (semicolon) as well as by `,` (comma).
- Numbers may have the `0-` (octal) or `0x-` (hex) prefix.

11. Supplied Command Line Utilities

OraFormsFaces comes with a number of command line utilities. It includes a tool to generate a random 128-bit AES encryption key to be used for encryption of the Oracle Forms database credentials and tools for batch manipulation of Oracle Forms FMB files.

11.1. KeyGenerator

OraFormsFaces includes an option to let a Java class at the JSF server determine the credentials to be used for the Oracle Forms database session. More details on this feature are described at ***14.2.2 Custom Java Class to Determine Database Credentials***. The default implementation of this feature uses 128-bits AES encryption to encrypt the credentials before transmitting them to the Forms server. For maximum security each OraFormsFaces installation should generate its own random AES key. The KeyGenerator that comes with OraFormsFaces does just that:

```
c> cd JDEV_HOME/jdev/extensions
c> cd com.commit_consulting.oraformsfaces.v????
c> java -jar OraFormsFaces-3.0.3.jar
Randomly generated 128-bits (16 bytes) AES key:
219A857A269EC2E4876B2ACBB9596998
```

The generated key should be used in a custom subclass of `FormsCredentialsImpl` at the JSF server as well as being included in the `userid` parameter at the Forms server. See ***14.2.2 Custom Java Class to Determine Database Credentials*** for more details on setting up this feature.

Note:	Do not use the key from this example but generate your own random key for maximum security.

11.2. JDAPI tools

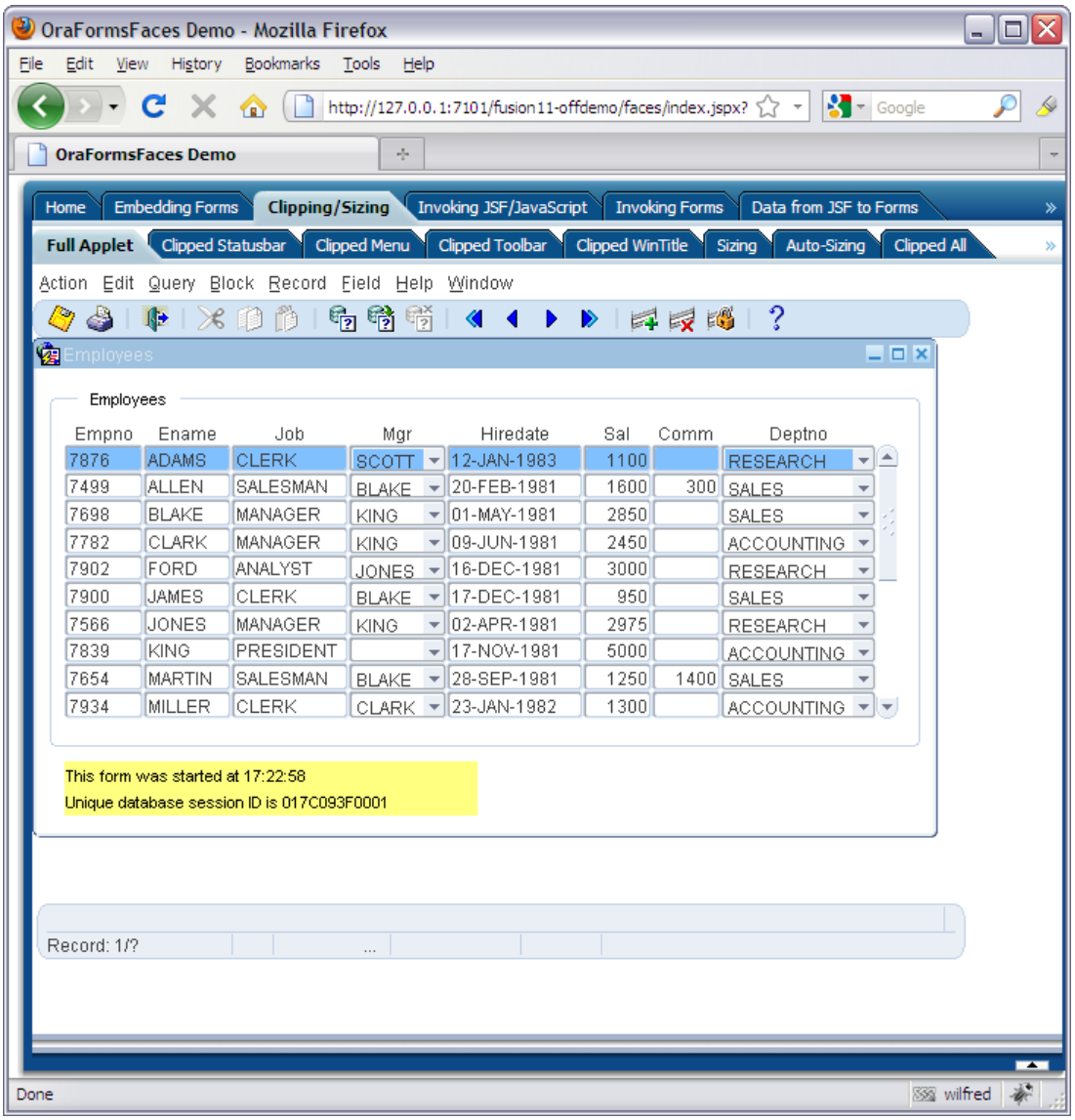
OraFormsFaces comes with three batch files that can be used from the command line to:

- **`objectgroup.bat`** to subclass one or more object groups from an object library to a set of Forms. This is typically used to subclass the main `oraformsfaces` object group to all FMB files of an application.
- **`attachlibs.bat`** to attach one or more PLL PL/SQL libraries to a set of Forms. This is typically used to attach the `off_lib.pll` library to all FMB files of an application.
- **`compile.bat`** to compile a set of FMB files in one go. This can be used to compile all FMB files of an application by calling a single batch file.

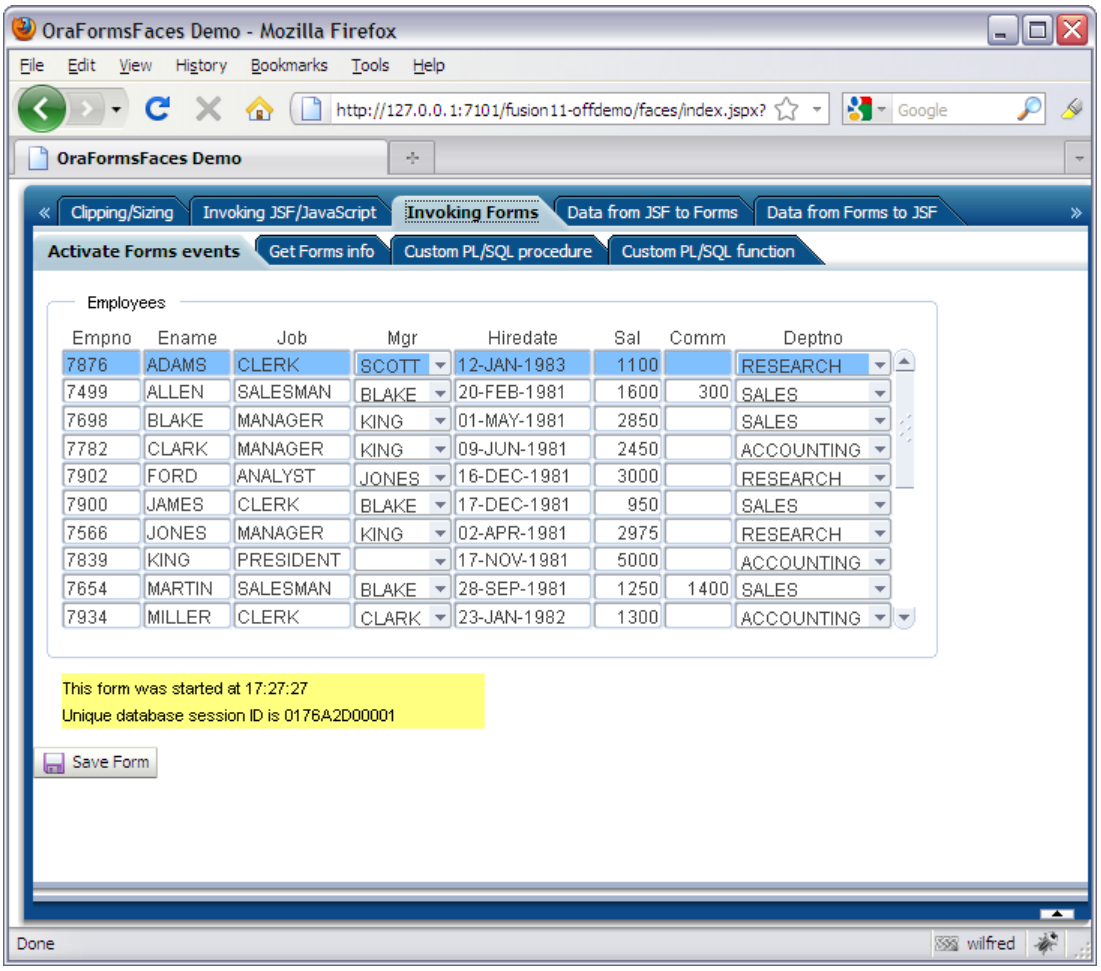
See ***4.2 Preparing the Oracle Forms Modules*** for details and example of using these tools. The supplied batch files are for Microsoft Windows only. However, all they do is call Java with some utility Java classes shipped with OraFormsFaces. Similar shell scripts for Linux or other operating systems can be created easily.

12. Visual Integration

By default, an Oracle Forms application looks and feels like a typical Oracle Forms application. You can recognize it being an Oracle Forms application instantly. It has some default user interface controls, like the status bar at the bottom, a menu bar at the top, a toolbar with buttons, etc. You can use OraFormsFaces to embed forms in a JSF application and keep this typical Oracle Forms look-and-feel. This would leave to something like the picture below. You can see the Forms applet is running in an ADF Faces application with top level tabs and navigation links at the bottom of the page. At the same time you see a fully functional Forms applet with menu, tool bar, and status bar.



Alternatively you can use an OraFormsFaces feature called **Applet Clipping**. This feature allows you to "cut out" a specific part of your Forms applet and only show that selected portion. You can do this without making any changes to the FMB file, although in the example below the background color of the canvas has been set to white to blend in with the web application. This would lead to something like the picture below:



In this example you see the same Oracle Forms form being used. But there are some differences. The Oracle Forms menu bar, button tool bar, and status bar are not visible. This completely blends the Oracle Forms form into the JSF page. The user cannot easily see if she is using a JSF page that embeds an Oracle Form or a true JSF page without Oracle Forms.

12.1. Enable and Configure Applet Clipping

The Applet Clipping feature is controlled by a number of properties of the OraFormsFaces `Form` component. Open the JSF page with the Form component, click on the Form component in the visual editor, to select it. Make sure it is the `off:form` component that is selected in the structure pane. If so, the properties of the Form component should be displayed in the Property Inspector of JDeveloper.

The important properties for Applet Clipping are:

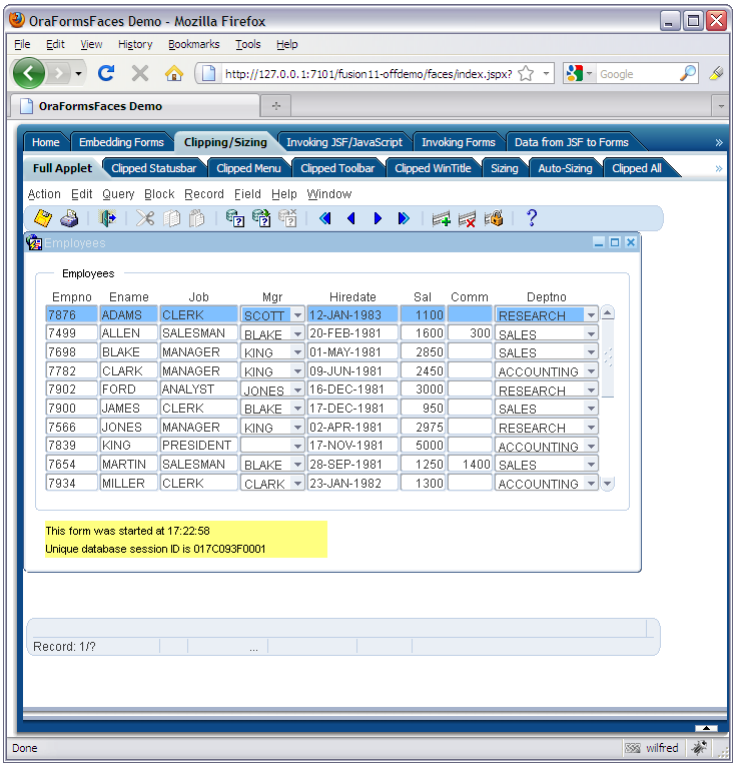
- **ClipApplet** – This boolean property determines if the Applet Clipping feature is enabled. Its default value is `false`, so you have to set this to `true` explicitly to enable the feature. Having a default of false ensures backwards compatibility with older versions of OraFormsFaces which did not support clipping.

- **AutoClipTop** – When clipping is enabled, this property specifies which chrome to remove from the top of the Forms applet. At runtime, the OraFormsFaces enhanced Oracle Forms applet will figure out how much to clip from the top of the applet to just remove these elements. The default value is `none` which doesn't clip anything from the top. Other possible values are `menu`, `toolbar`, and `window-title`. Setting the property to `menu` will remove the menu bar from the top of the applet no longer allowing users to select options from the menu. This makes the applet look and feel more like a true web application. Navigation to other pages and forms should now be handled by the JSF application. Setting the property to `toolbar` removes both the menu and the toolbar/button bar to have the Forms applet blend in even more with the surrounding web page. The `window-title` is the most extreme value. It will not only remove the menu and toolbar but will also remove the title bar at the top of the Forms window. See ***12.2 Auto Clipping Examples*** for example screenshots of the different options.
- **AutoClipBottom** – When clipping is enabled, this property specified if the status bar should be removed from the bottom of the Forms applet. The default value is `none` which will not remove the status bar. Setting it to `statusbar` will remove the status bar. If the status bar is clipped and the `AutoClipTop` property is set to `window-title` and `AutoSize` is enabled, it will even cause all window edges to be clipped to have the Forms applet fully blend with the web page. See ***12.2 Auto Clipping Examples*** for example screenshots of the different options.
  Remember that removing the status bar might also hide messages from the user that would normally appear in this status bar. You should take care that these messages remain noticeable for end users, for example by pushing them to JavaScript alerts as described in ***12.5 Error and Message Handling with Applet Clipping***.
- **AutoSize** – This enables or disabled the auto-sizing feature of the OraFormsFaces applet. By default this is disabled so applications upgraded from previous OraFormsFaces without this feature work the same. If you enable this feature, the OraFormsFaces enhancements to the Oracle Forms applet will figure out the sizing of the window in your Forms module at runtime and will size the applet in the web page accordingly.
- **Width** and **Height** – This is the number of pixels of the viewport shown in the web page. This is the amount of screen real estate that will be used by the OraFormsFaces Form component. When `AutoSize` is enabled these settings are only used for the initial rendering of the web page. As soon as the applet finished startup the OraFormsFaces area will be sized accordingly to the size of the active Oracle Forms window.
- **ClipLeft**, **ClipTop**, **ClipRight**, and **ClipBottom** – This is the number of pixels to remove from the original Forms applet. When the auto clipping (top and/or bottom) features are enabled these values are added/subtracted from the automatically calculated clipping. This means you can use positive or negative values to increase or decrease the calculated clipping.
- **ShowDevControls** – When set to true, the running web page in the browser will have controls to enable/disable clipping at runtime and controls to set the width, height, and number of clipping pixels. These controls can be used to easily find the correct settings for all these properties at runtime. See ***12.4 Using Development Controls*** for a full description of their usage.

12.2. Auto Clipping Examples

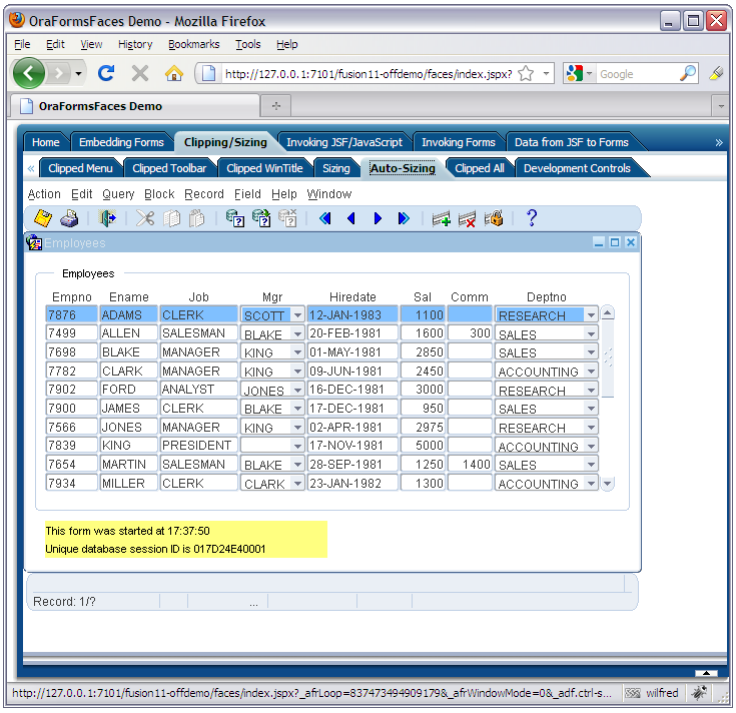
The visual properties for (auto) sizing and (auto) clipping can be combined in a number of ways. Below are some screenshots of typical settings.

12.2.1. Default Settings



The screenshot above shows an OraFormsFaces `Form` component with all default settings. This will use the `width` and `height` properties to size the viewing area. When these are not set to specific values the size will be 640x480 pixels.

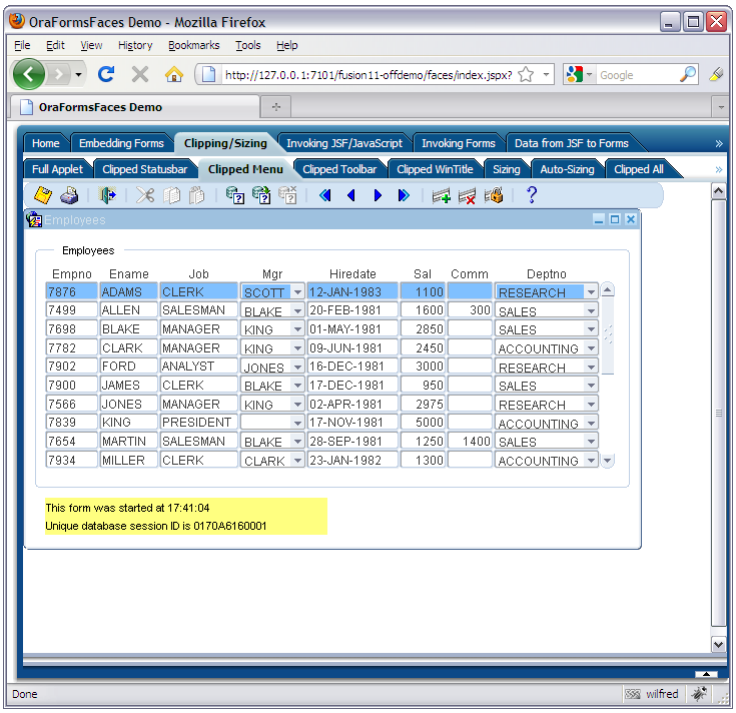
12.2.2. Auto Sizing Enabled



This screenshot above shows the same applet but with the auto-sizing feature enabled. This automatically determines the size of the OraFormsFaces applet to just fit the size of the form being displayed. In this case the size is determined to be 618x435 pixels. This feature makes for a tighter fit of the applet saving valuable screen real estate for other purposes.

The auto-sizing does causes the applet area to resize when the applet finishes its startup. This can cause other items on the page to shift slightly. If you want to prevent this, find out the actual sizing values after resizing and be sure to also set them at design time so the initial rendering is already in the correct size. To find the actual sizing values just enable the development controls and run your page once more.

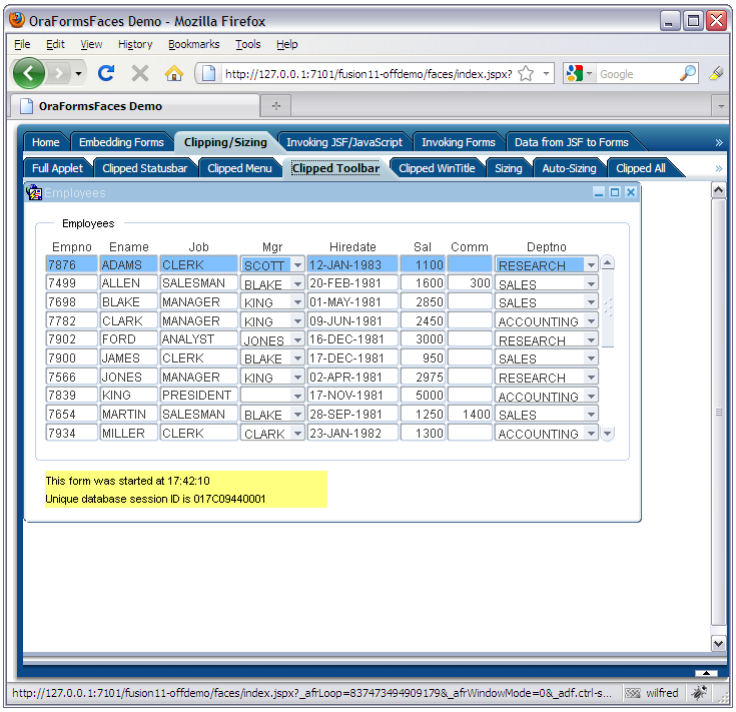
12.2.3. Auto Clip the Menu



The above screenshot is result of setting `ClipApplet` to `true` and `AutoClipTop` to `menu`. As you can see the menu is now clipped from the applet. This makes the applet blend in more with the surrounding web page. It also forces the end user to use JSF/ADF controls for navigating to other business functions. This can be a first step in slowly migrating away from Oracle Forms to a full JSF/ADF application. When the application flow control is fully handled by the JSF application it becomes easy to take out individual Forms and replace them with new JSF pages without changing the rest of the application.

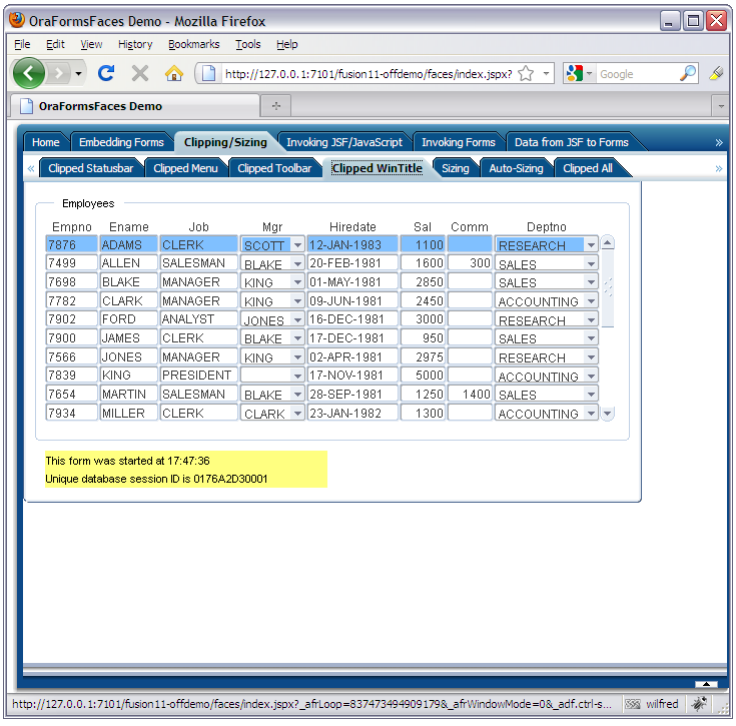
Note:	When removing the menu it is likely you want JSF buttons on the same page to trigger vital actions that used to be listed in the menu. Examples are buttons to Save changes, or perform record navigation. See ***7 Invoking PL/SQL Events from JavaScript*** for examples how to trigger Forms actions from JSF components, including pre-defined actions like `do_key`.

12.2.4. Auto Clip the Menu and Toolbar



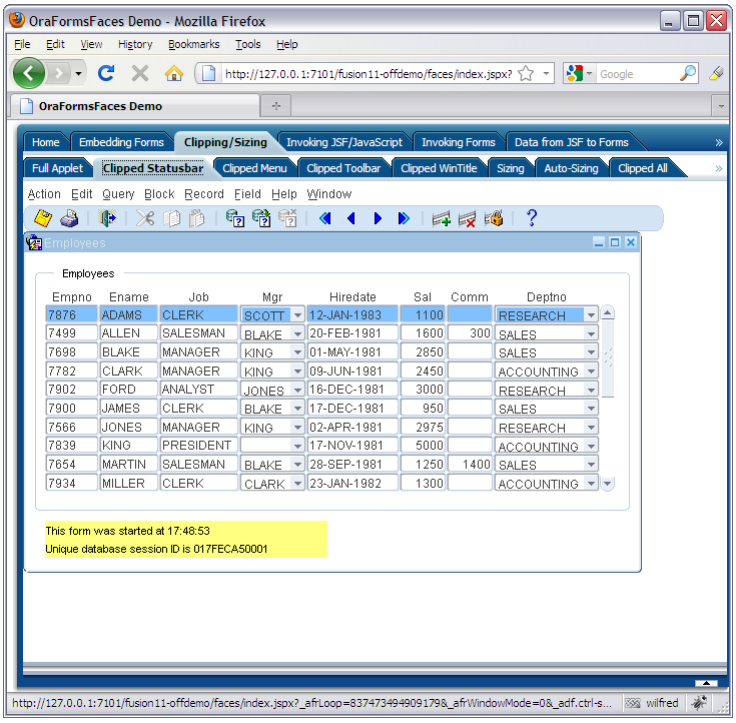
The above screenshot is result of setting `ClipApplet` to `true` and `AutoClipTop` to `toolbar`. As you can see both the menu and toolbar are now clipped from the applet. This makes the applet blend in even more with the surrounding web page.

12.2.5. Auto Clip the Menu, Toolbar and Window title



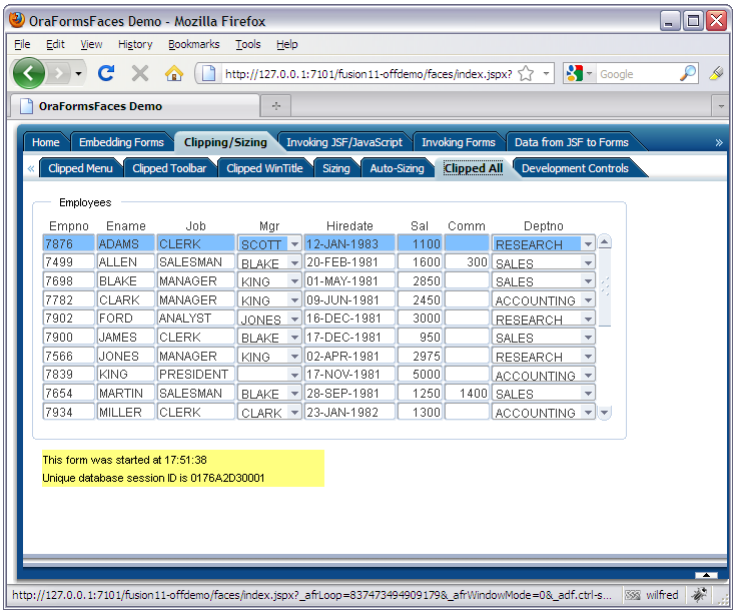
This screenshot shows even more clipping than the previous one. Here the `AutoClipTop` property is set to `window-title` which also removes the window-title bar. Since the left, right, and bottom edges of the window are still visible we keep a one pixel edge of the window-title bar to get a full border.

12.2.6. Auto Clip the Status bar



This screenshot shows the result of setting the `AutoClipBottom` property to `status bar`. In this example the `AutoClipTop` has been reset to `none`. The `AutoClipBottom` feature removes the status bar from the user's view. Note that this might also hide messages that appear in the status bar from the end user. See ***12.5 Error and Message Handling with Applet Clipping*** for an alternative.

12.2.7. Auto Clip the Menu, Toolbar, Window title and Status bar



This screenshot shows the most extreme form of clipping with the `AutoClipTop` property set to `window-title` and the `AutoClipBottom` property set to `status bar`. This removes all chrome from the Forms applet. In the previous example where `AutoClipTop` was set to window-title we kept a 1 pixel border for the window. In this extreme clipping example that is also removed since we are able to remove the bottom part of the window since the status bar is also hidden and because `AutoSize` is set to `true`.

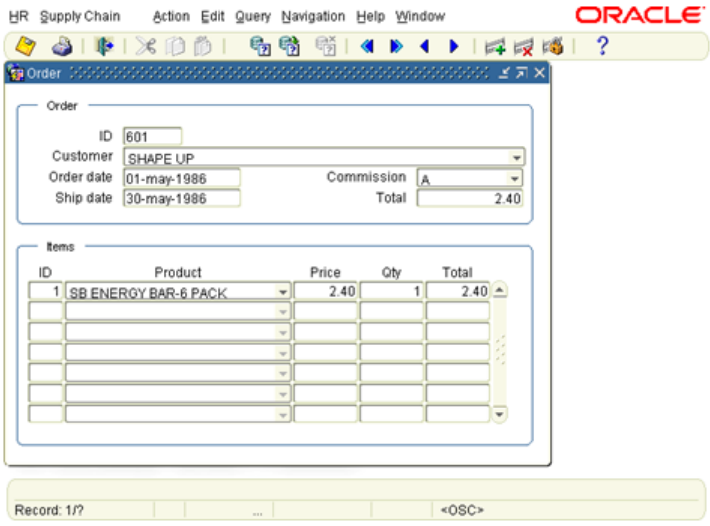
This most extreme clipping makes the applet fully blend in with the surrounding page. The remaining blue border you see is an item group border from the actual Form. You can even remove this in Forms Builder if you like.

> **Note:** It is likely you want JSF buttons on the same page to trigger vital actions that used to be listed in the menu. Examples are buttons to Save changes, or perform record navigation. See ***7 Invoking PL/SQL Events from JavaScript*** for examples how to trigger Forms actions from JSF components, including pre-defined actions like `do_key`.

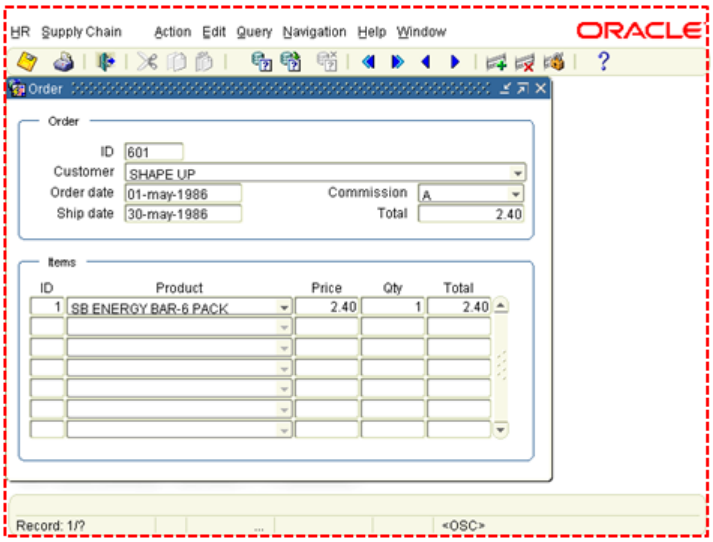
12.3. How Applet Clipping works

As stated, the Applet Clipping feature works without any required changes to the Oracle Forms modules involved. It is strictly a client side feature using standard HTML and CSS (Cascading Style Sheet). Technically, what happens is that the actual Forms Java applet running the web browser is set to be larger than the specified width and height of the Form component. The Forms Java applet is then placed in a smaller containing box with limits the viewport to the width and height of the Form component. Finally, the X- and Y-position of the Forms Java applet are set to negative values relative to the surrounding (smaller) box. This effectively shifts the Forms Java applet to the left and to the top removing the edges there.

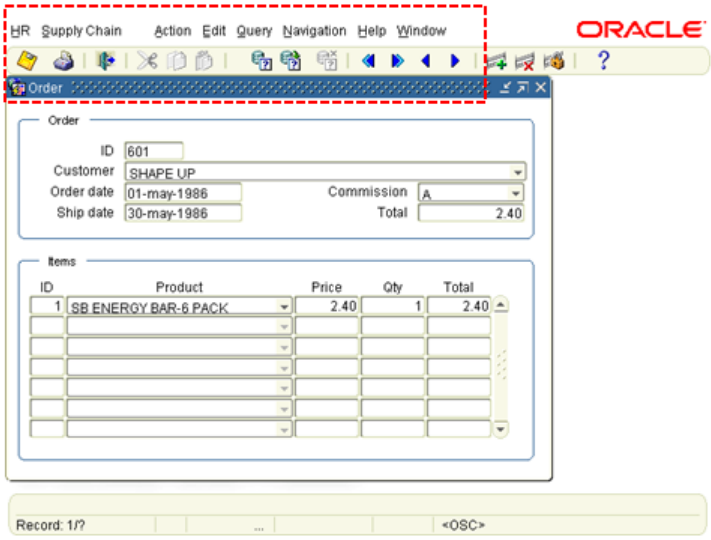
This will be demonstrated in a number of steps. Let's begin with a default Oracle Forms applet as it runs in a web browser:



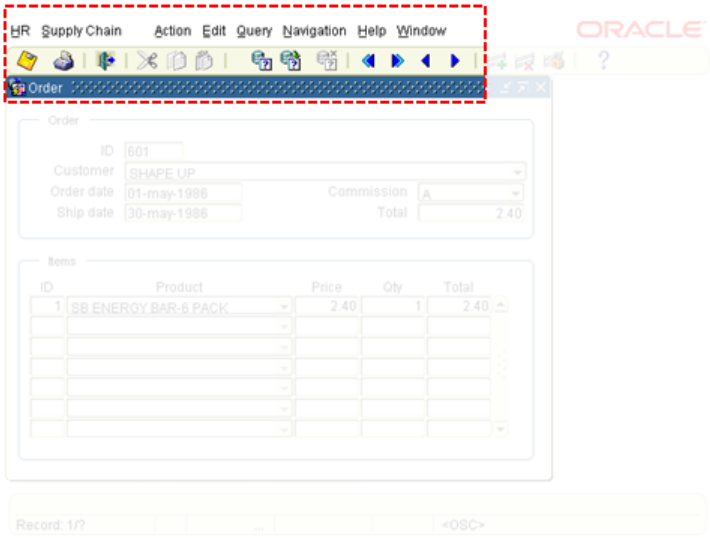
Next we put the Oracle Forms applet in a surrounding box:



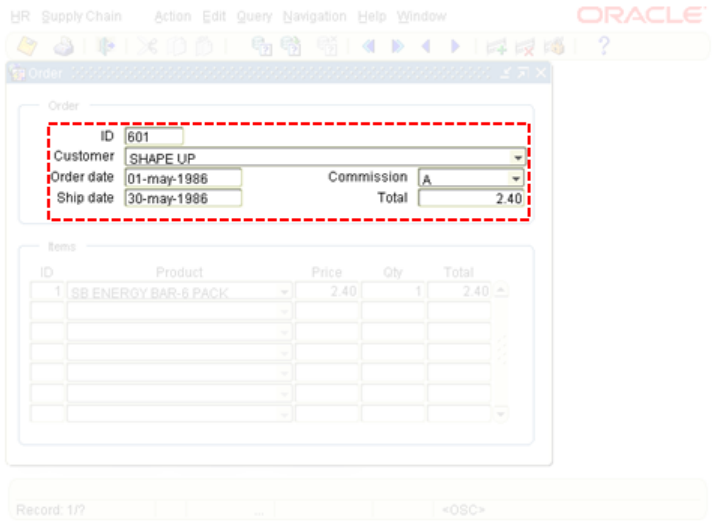
Next we reduce the size of the surrounding box:



Then we set CSS properties to instruct the box to not show anything outside its limits. In this example the areas cut off are still grayed out for demonstration purposes.



We then shift the Oracle Forms applet to the left and top by setting the X- and Y-position to negative values:



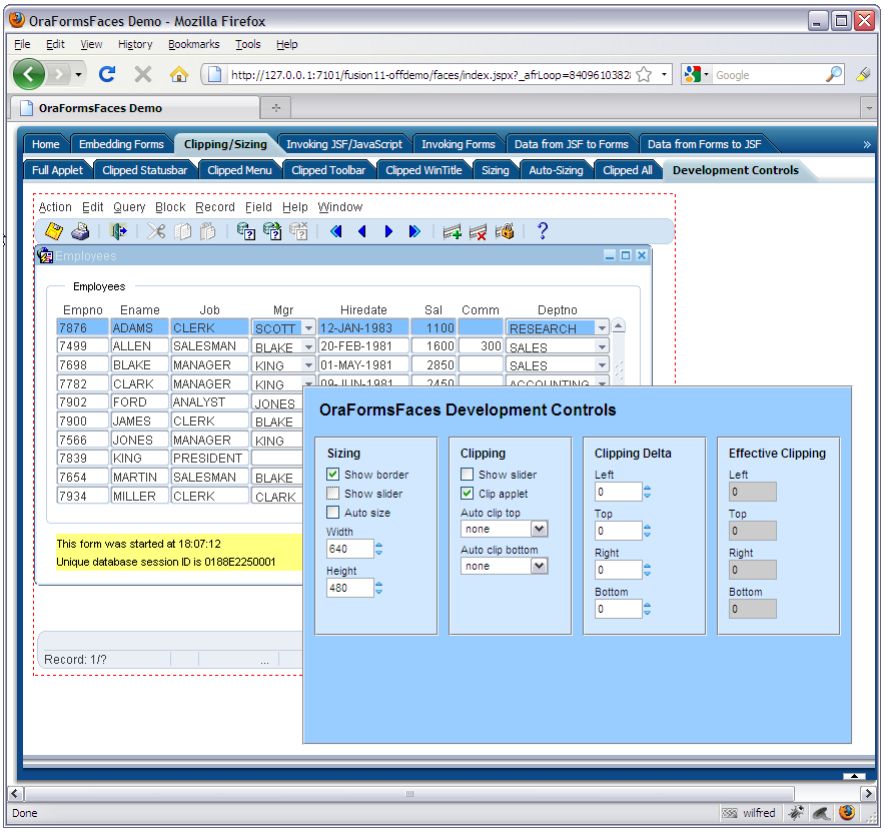
If we really hide the areas that are cut off, we get:

And in the end result we obviously do not use red dotted lines:

12.4. Using Development Controls

For most users the Auto-Sizing and Auto-Clipping features as described earlier will be the easiest way to size and clip the applet. If you want to specify the sizing or clipping manually, finding the correct values for the `Width`, `Height`, and – if using Clipping – `ClipLeft`, `ClipTop`, `ClipRight`, and `ClipBottom` can be a daunting task. It is not possible, nor desirable, to run the Oracle Forms applet in the JDeveloper JSP visual editor. So, you need another way to find the correct values for these settings.

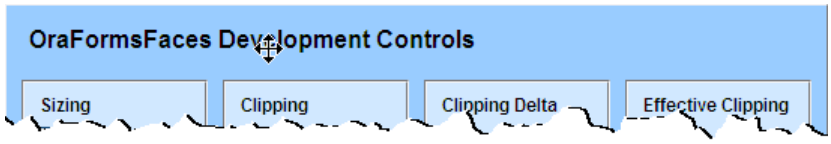
Luckily, OraFormsFaces comes with Development Controls to do just that. You can set the `ShowDevControls` property of the Form component to `true` in JDeveloper. If you then run your web page, you will be presented with controls to change these settings at runtime. You can use this with or without the Clipping feature, but the development controls are most valuable when using clipping:



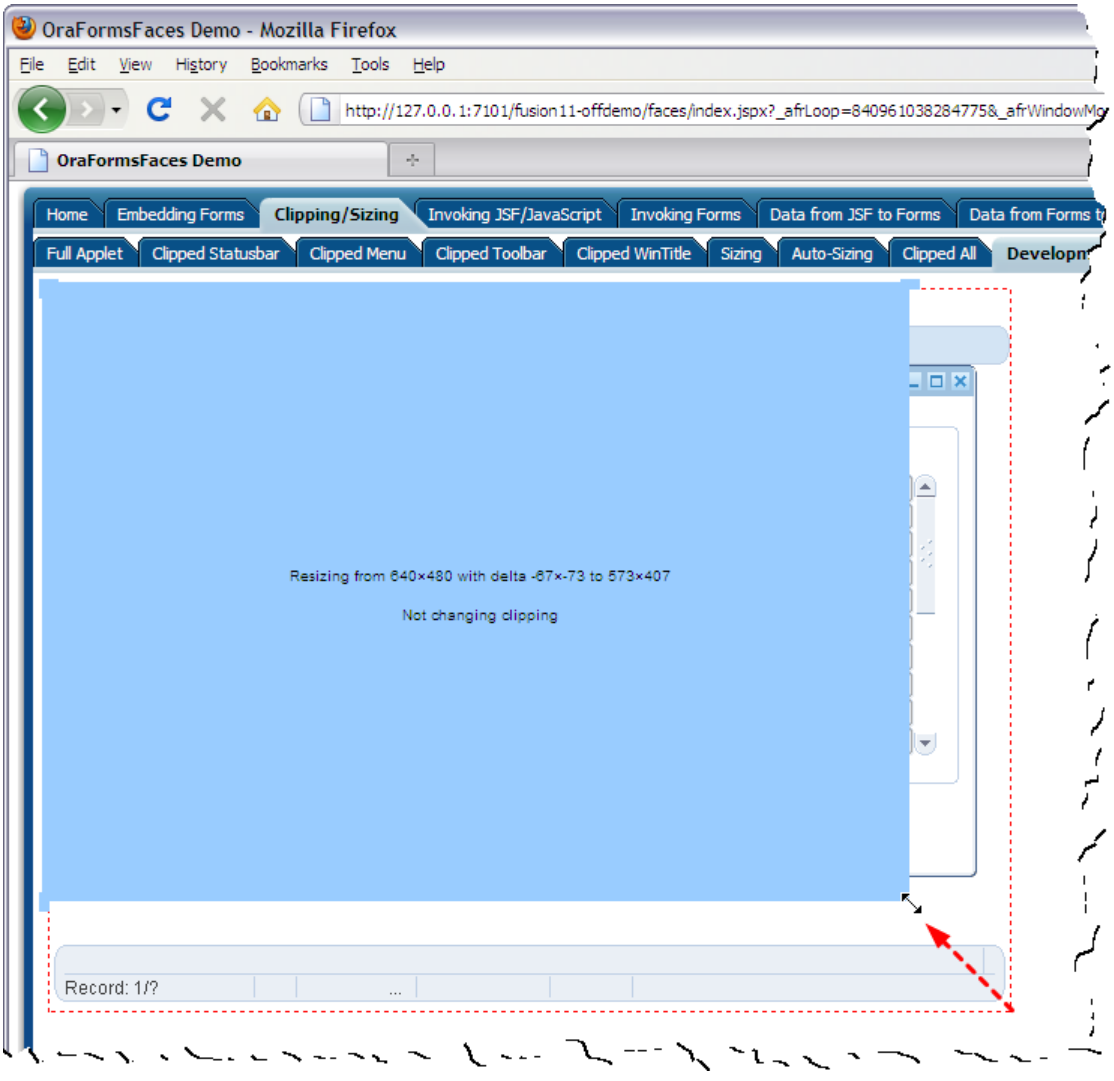
> **Note**: Development Controls might not work with certain versions of Microsoft Internet Explorer. If you are experiencing problems please use Mozilla Firefox while working with the development controls.

As you can see in the previous example, a number of development controls are available:

- **OraFormsFaces Development Controls title bar** – You can drag-and-drop the title bar to move the Development Controls around and move them out of the way:



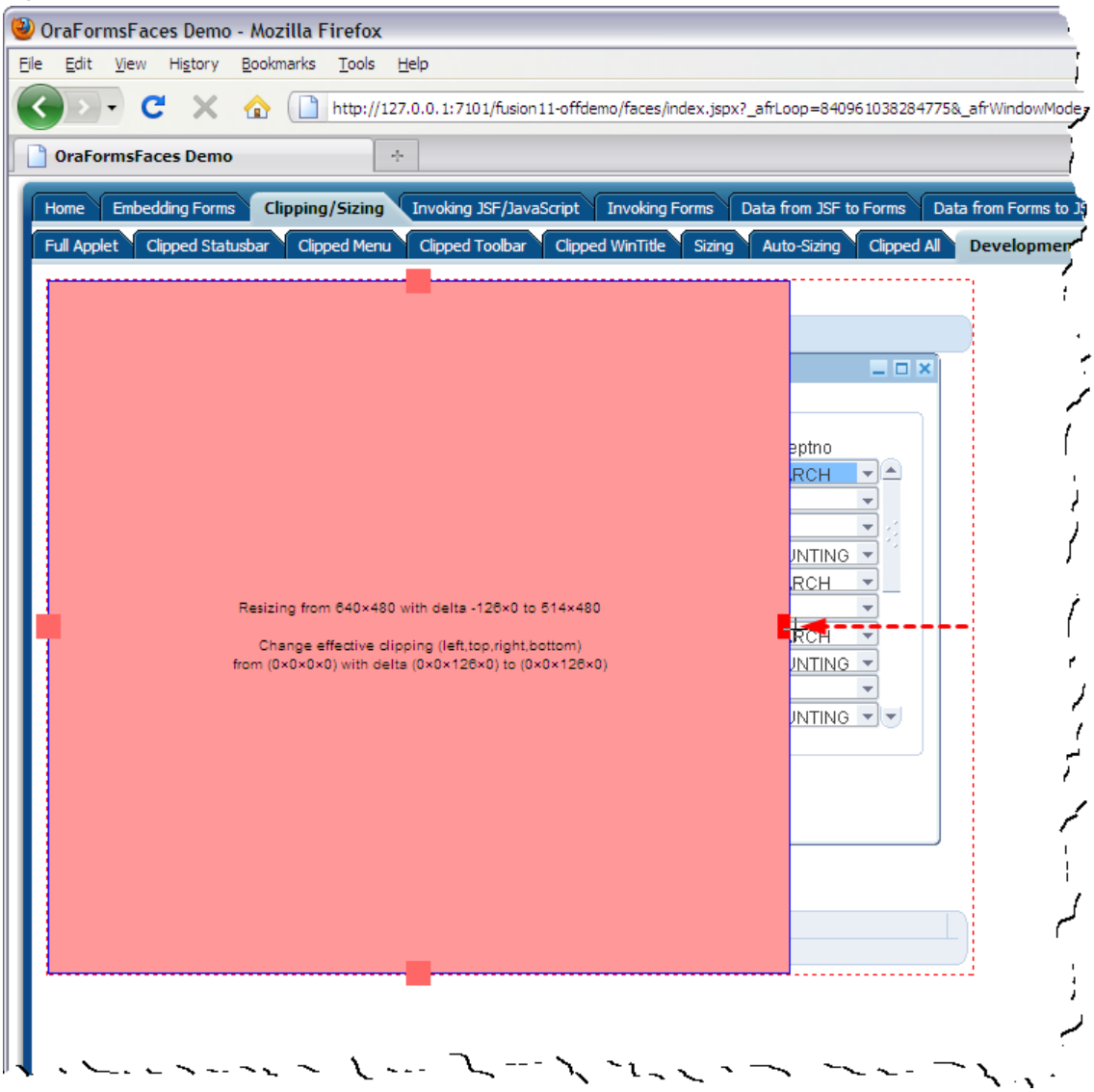
- **Sizing Controls**
  - **Show Border** – This checkbox indicates whether a red dotted border should be displayed around the OraFormsFaces Form component. This can be used to clearly distinguish the area that is occupied by the Form component.
  - **Show Slider** – This enables the sliders to resize the applet by dragging and dropping the corners or edges:



  - **Auto Size** – This checkbox indicates whether the auto sizing feature should be enabled. When this feature is enabled the `Width` and `Height` items become read-only and will show the size as it's been calculated by the auto-sizing feature. You can set these values for width and height at design time to prevent the applet area from resizing once the applet finishes its startup as that can cause items to shift at runtime.

- **Width** and **Height** – When the Auto Sizing feature is disabled, these controls can be used to set the width and height of the viewport. You can use the little arrow buttons to increase or decrease the values with steps of 1, or you can type in new values. These new values will be enforces once you leave the input box.

- **Clipping Controls**
  - **Show Slider** – This enables the sliders to change the applet clipping by dragging and dropping the edges:



  - **Clip Applet** – This checkbox indicates whether the clipping feature should be enabled.
  - **Auto Clip Top** – This dropdown list sets which items to clip from the top of the applet.
  - **Auto Clip Bottom** – This dropdown list sets which items to clip from the bottom of the applet

- **Clipping Delta Controls** – These controls can be used to specify the number of pixels to remove from the left, top, right, and bottom of the Oracle Forms applet. When you increase/decrease the clipping properties, the sizing properties (width and height) will automatically change accordingly. For example, increasing the `ClipLeft` by 10 will decrease the `Width` property by 10 as well. When the auto-clipping feature is enabled these values will be added/subtracted from the automatically determined values. This means you can use these as a correction to the automatically calculated clipping to clip (slightly) more or less.

- **Effective Clipping Controls** – These read only controls show the effective number of pixels that are clipped from the applet. This is a combination of the automatically calculated values from the auto-clipping features and the values from the `ClipLeft`, `ClipTop`, `ClipRight`, and `ClipBottom` properties.

Tip:	Use the development controls to quickly play around with the different features at runtime without the need to change the JSF source and restart the page each time. Once you are satisfied with the settings you can change the JSF source once to the appropriate values.
Tip:	When using Auto-Sizing you can use the development controls to find out the actual size calculated by this feature. If you set the width and height of the Form component to the same values in the JSF page than you won't have the applet change its size when it completed startup which can cause items to (slightly) shift in the page.

12.4.1. Best Practice use of Development Controls

For most users the Auto-Sizing and Auto-Clipping features as described earlier will be the easiest and best way to size and clip the applet. If you want to specify the sizing or clipping manually, the development controls make it much easier to find the correct values for the different properties.

But even with the development controls, care must be taken to follow the correct approach. You want to keep the clipping properties as low as possible to ensure Oracle Forms doesn't show ListOfValues, Alerts, or other "pop ups" that are partially in the clipped area. See ***12.5 Error and Message Handling with Applet Clipping*** for an example of this.

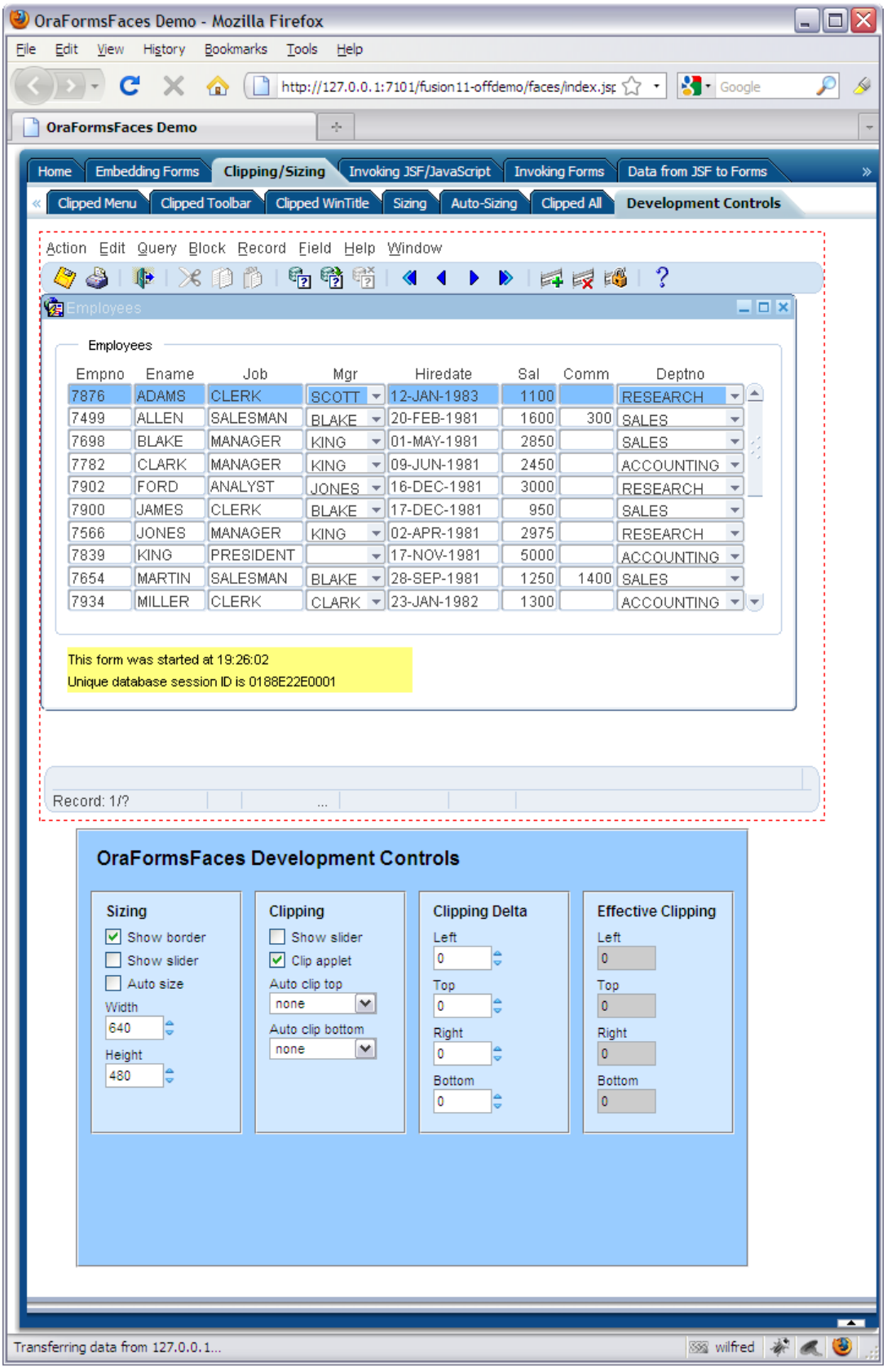
For demonstration purposes let's say you are not using the auto-clipping feature but still want to remove the Oracle Forms status bar from the bottom of the applet. From running the page with DevControls and AutoClipBottom set to statusbar we can see that setting `ClipBottom` to 44 is sufficient for that. But what happens if you set `ClipBottom` to 250? Visually, nothing changes. But if you look at the explanation in ***12.3 How Applet Clipping works***, there is an important difference. With `ClipBottom` set to 44, the actual Forms applet might be 394 pixels high (350 pixels viewing area plus the 44 pixels from the clipping). With `ClipBottom` set to 250, the actual Forms applet will be extended to 700 pixels high (350 pixels viewing area plus the 250 pixels from the clipping).

As said before, this does not make a visual difference. But it does make a difference when Oracle Forms want to display an alert, list-of-values, or any other type of popup. You or Oracle Forms might have code to center that popup in the middle of the Forms applet. But if you're clipping 250 pixels from the bottom, the physical center of the Forms applet might be (partially) outside of your viewport. See ***12.3 How Applet Clipping works*** for an example of how an applet can be cut off.

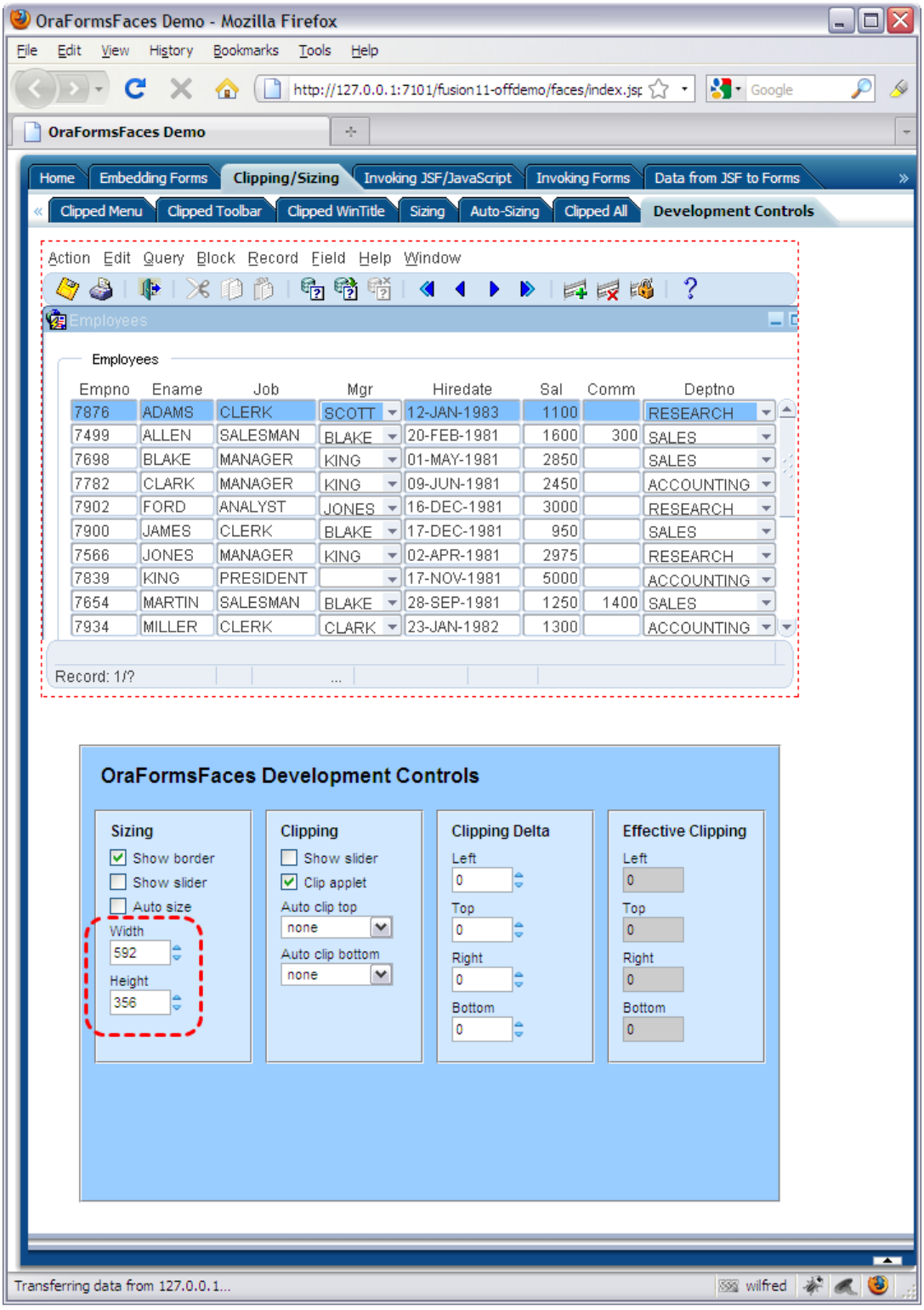
So, it is important to keep the clipping properties set to a minimal value. The following example shows the advised steps to clip the applet to only show the data entry part. Again, using the auto-clipping feature is the preferred approach but if there is a reason not to use the auto-clipping feature, the steps below are the way to go.

- Start the JSF page with Development Controls enabled
- Check the checkbox to show the red dotted border so you can clearly see the amount of screen real estate used by OraFormsFaces

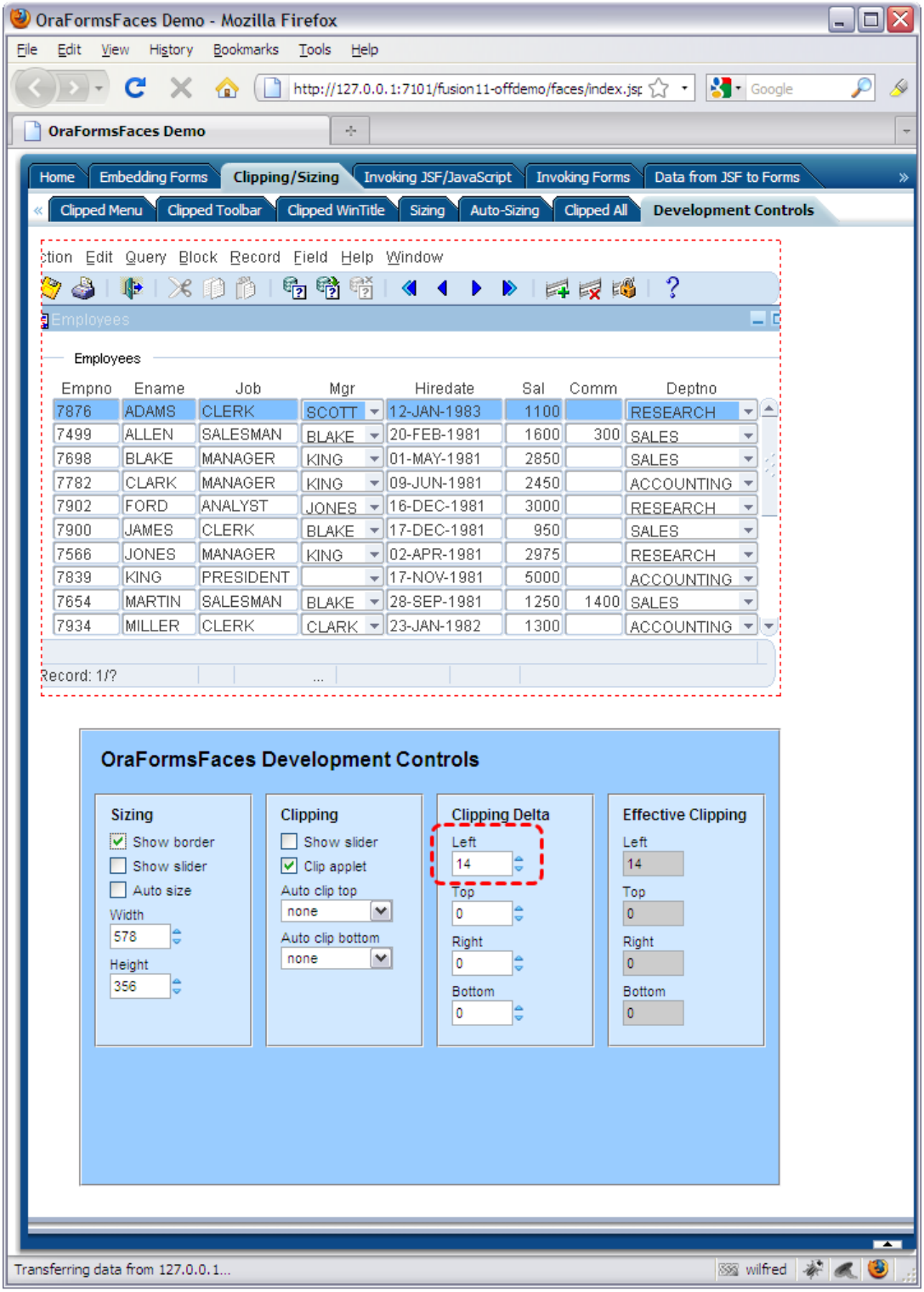
- Set all the clipping properties to 0:



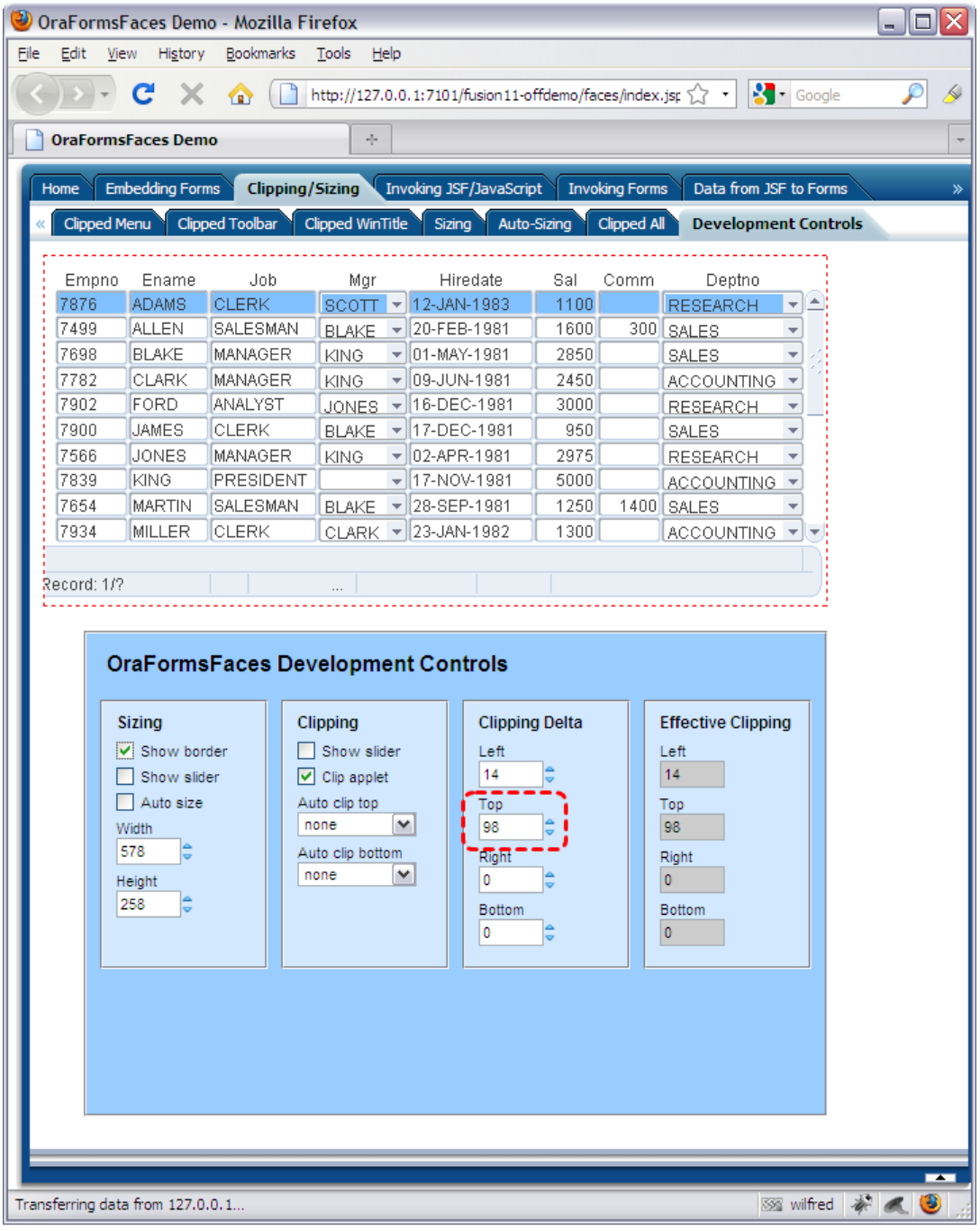
- Set the `Width` and `Height` properties as low as possible, without hiding parts of your form you want to display in the end result. This ensures the applet is as small as possible requiring minimal clipping reducing the risk of user interface elements being (partially) clipped. This example uses a width of `592` and a height of `356`:



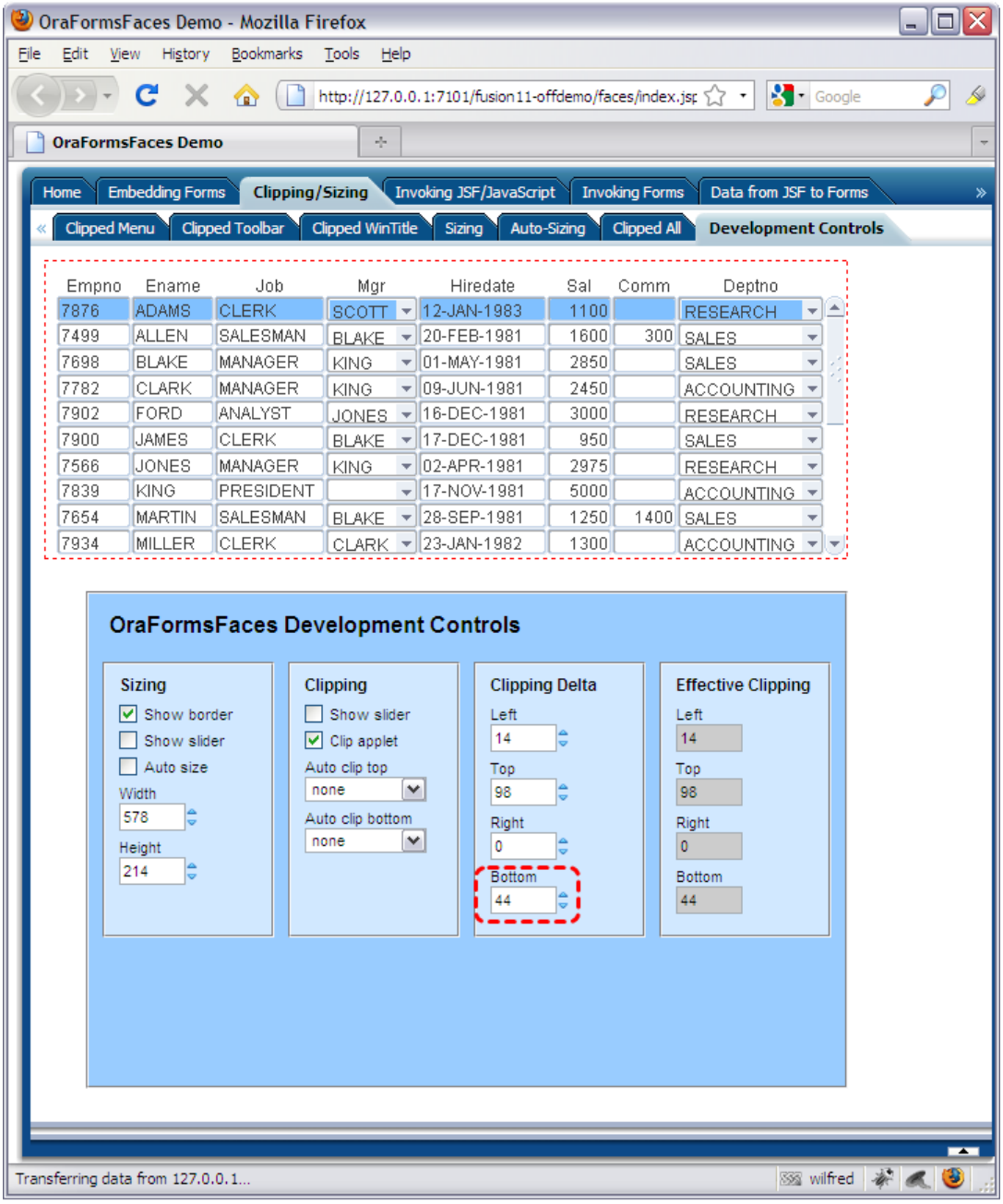
- Next, increase the `ClipLeft` until everything you want to clip from the left is ***just*** removed. Be sure not to increase the `ClipLeft` property any further than necessary. Remember, we want to keep the clipping values as low as possible. In this example it means increasing the `ClipLeft` to `14`:



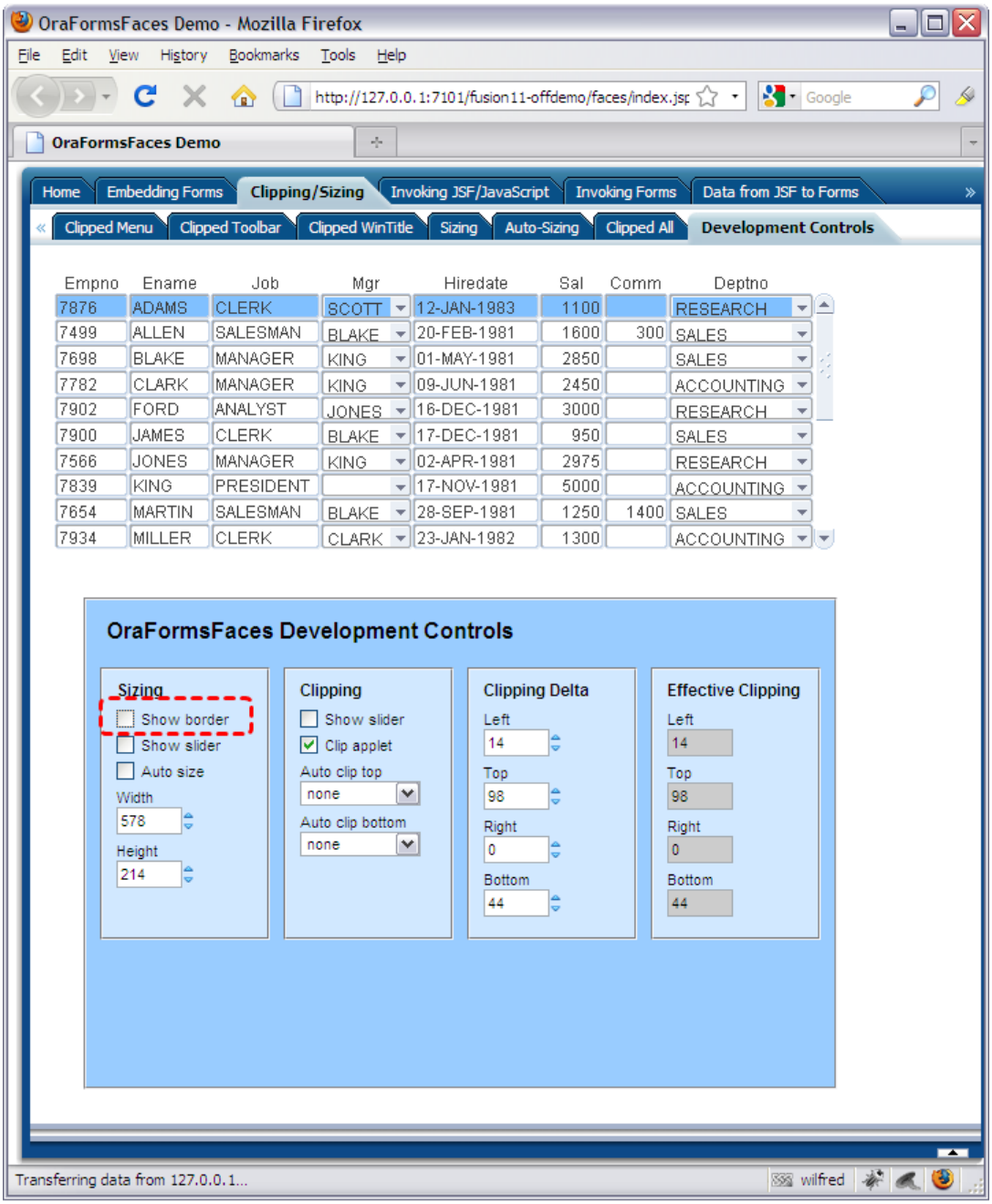
- Next, increase the `ClipTop` until everything you want to clip from the top is ***just*** removed. Again, be sure not to increase the `ClipTop` property any further than necessary since we want to keep the clipping values as low as possible. In this example we remove the menu bar, the tool bar, and the item group border by setting `ClipTop` to `98`:



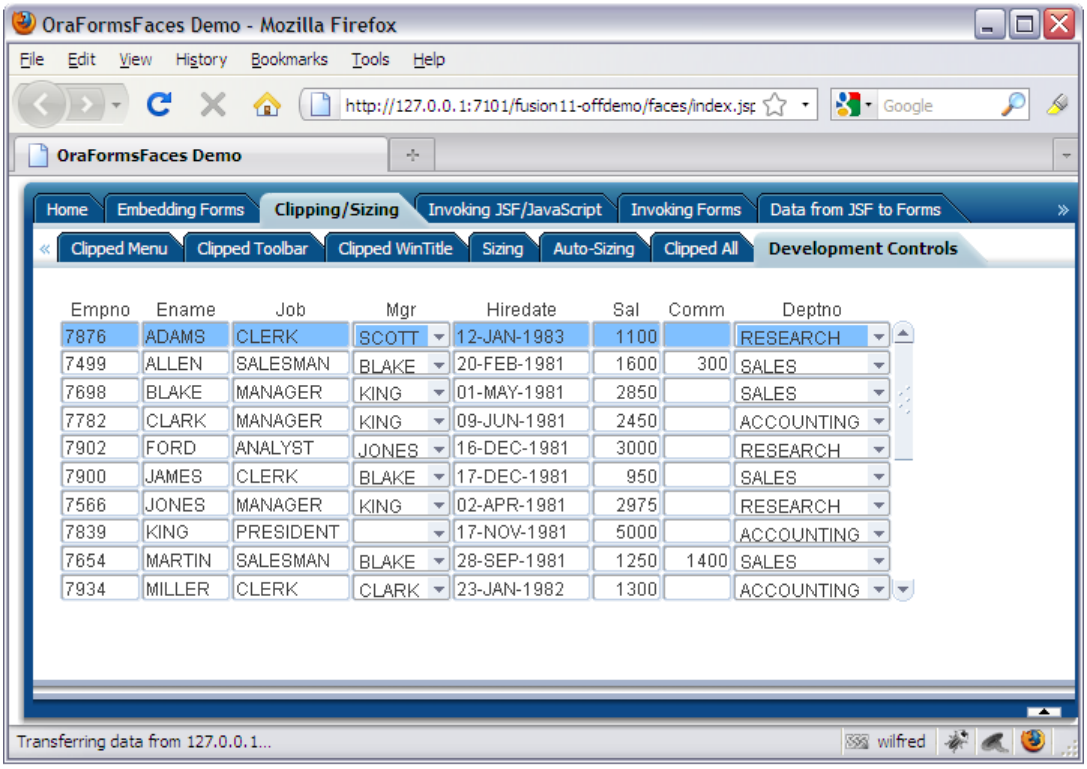
- In our example there is no need to increase `ClipRight` since we already removed all excess chrome by setting the `width`. Just increase `ClipBottom` until the status bar is invisible. In this example it required a value of `44`:



- As a final step disable the red dotted border and see if the end result is satisfactory:



- If everything looks fine, note the number used for the `Width`, `Height`, and the four clipping properties. Then put those numbers in the OraFormsFaces `Form` component in the source of your JSF page. Finally, you can run your application again to see if the initial sizing of the applet is satisfactory. If so, you can revert the `ShowDevControls` property of the `Form` component to `false` to remove the development controls from the end user application, with the following end result:

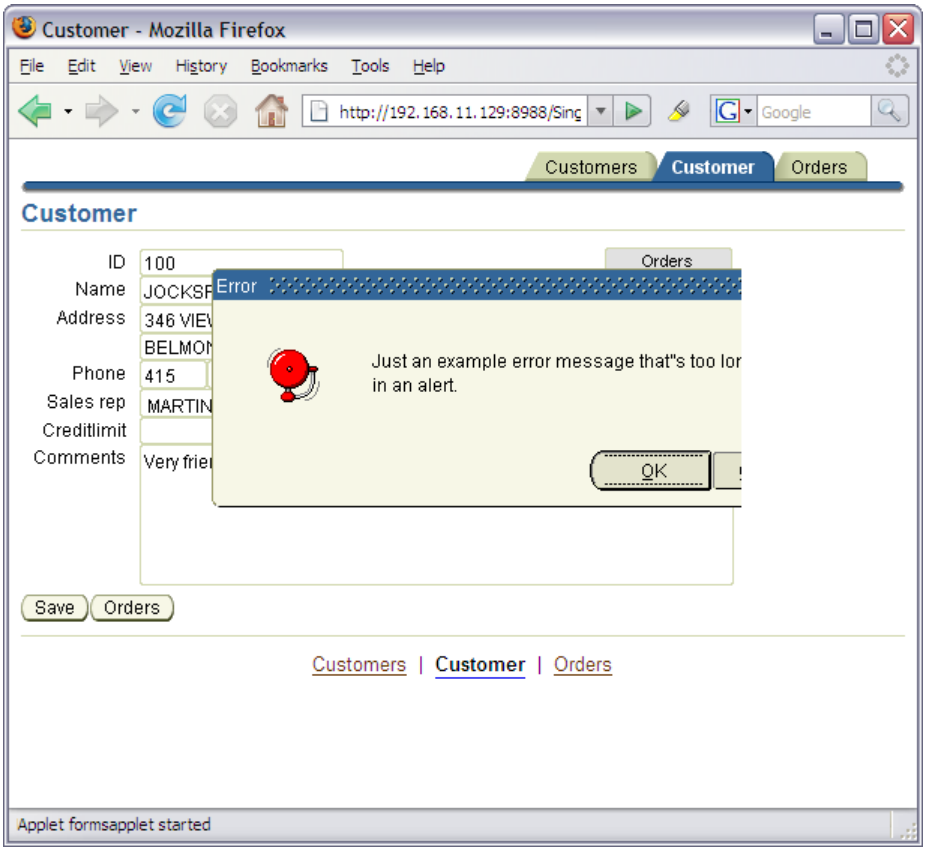


12.5. Error and Message Handling with Applet Clipping

If you use the Applet Clipping feature, error and message handling might get a bit more complicated. By default, Oracle Forms displays messages only in the status bar at the bottom of the Forms applet. With the clipping feature it is perfectly possible, and likely, that you remove the status bar from the viewport. But if Oracle Forms just continues to put warning messages in the status bar, they will be invisible for the end user.

You could add `ON-MESSAGE` and `ON-ERROR` triggers to your form and handle any messages raised by Oracle Forms. Many people already have something similar in a generic PL/SQL library to catch these messages and throw an Oracle Forms alert to show the message. You could use this in combination with OraFormsFaces Applet Clipping, but using Oracle Forms alerts also has its own risks. By default, Oracle Forms will display an alert centered in the middle of the Forms applet. If you clip an extreme number of pixels from the Forms applet,

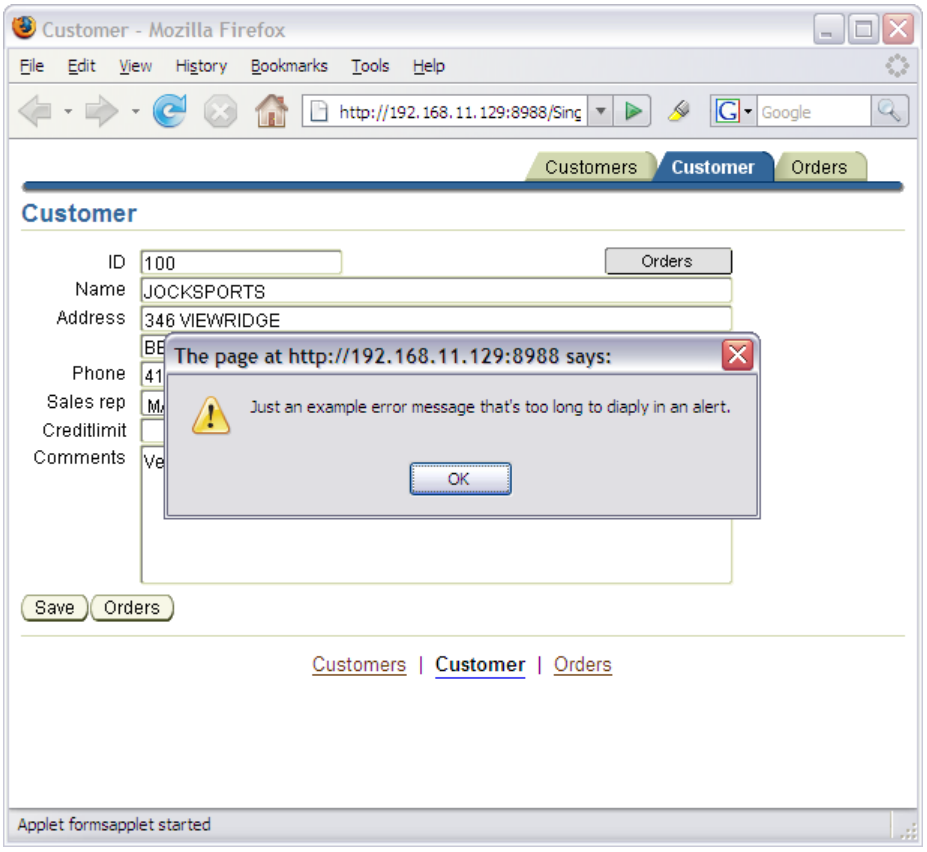
an alert might actually fall (partly) outside of the viewport, for example:



Currently there is no way to reposition Oracle Forms alerts to ensure they fit within the clipping viewport. Alternatively, you can prevent the usage of Oracle Forms alerts all together. OraFormsFaces comes with a supporting PL/SQL library. That library has a package `offJavaScript` that has a procedure to show an alert using JavaScript. The procedure is called `offJavaScript.jsAlert` and can be used as follows:

```
offJavaScript.jsAlert('some message');
```

This would result in a JavaScript alert being shown by the web browser:



You can use a call to `offJavaScript.jsAlert` in your own error and message handling routines, or add an `ON-ERROR` trigger that calls `offTriggers.onError` and an `ON-MESSAGE` trigger that calls `offTriggers.onMessage`. These two library functions are defined in the `offTriggers` package in `OFF_LIB.PLL`. They offer default handling of getting the message and showing it in a JavaScript alert. You can take a look at the code if you want to integrate it with your existing error- and message handlers.

> **Note:** In Oracle Forms prior to version 11, the JavaScript alert is executed from an asynchronous thread. This ensures the Forms applet in the background keeps running and keeps sending the periodic heartbeat message to the Forms server ensuring the session does not time out. This has the disadvantage that the PL/SQL code calling `offJavaScript.jsAlert` will continue to run the PL/SQL after the call to `jsAlert` even before the user dismisses the dialog box.
> If you want to change this behavior, just pass `false` as a second argument to the `jsAlert` call which will execute the JavaScript alert in the main thread blocking any PL/SQL processing until the alert is dismissed by the user. Be sure to set appropriate time-out values on the Forms server to prevent the server from timing out the session if the user is not quickly in dismissing the dialog.

13. Forms Java Applet Instance Reuse

OraFormsFaces uses a Java applet to show the Oracle Forms. The default behavior of web browser is to discard a Java applet when the user refreshes the page or navigates to another page. With small and trivial applets this is not a problem. However, with Oracle Forms this causes some issues. The startup time of an Oracle Forms applet is just too long to endure on every page where you want to embed an Oracle Form.

Luckily there is a Sun JRE (Java Runtime Environment) feature to change this behavior. When enabling this legacy lifecycle feature the applet is not discarded when leaving or refreshing the page. Instead it is just kept running in the background. Whenever the same or another page embeds an applet with the exact same properties, Sun JRE will restore the applet instead of starting a new applet. This obviously is much faster than starting a brand new applet on each page.

OraFormsFaces is configured to enable this feature and help you fully exploit it. The complicating factor is that the applet has to be 100% identical on each and every page to ensure an applet instance is reused. This causes some difficulties on how to pass the name of the form module and parameters to the Forms applet. OraFormsFaces makes sure these settings are passed through JavaScript so the applet declaration itself is kept 100% identical on each page.

The good thing about the legacy-lifecycle feature is that a single applet instance is reused on every page. This causes the end user to only endure the applet startup time once for an entire web session. However, it also introduces an extra challenge. By default, the applet will be restored in the exact same state it was in when it was suspended on the previous page. This is almost always undesirable. It could be that the user is on a different web page and should see a completely different form. Even when the user returned to the same web page it is likely she expected the data in the form to refresh when visiting the same page again.

To tackle this challenge, OraFormsFaces has enhanced the Forms applet to introduce two new events. These events are called `when-applet-suspended` and `when-applet-activated`. The `when-applet-suspended` event gets called when leaving a web page and the applet is suspended into the background cache. The `when-applet-activated` event gets called when the applet is restored from the background cache to be displayed (again) to the user.

> **Note**: You can disable the legacy lifecycle feature completely from the formsweb.cfg file. Be aware that the Forms applet instance and associated Forms and database session will be destroyed each time the Forms applet is removed from the user's view and a fresh instance is started each time again.

13.1. Suspending an Applet Instance

OraFormsFaces' default handling of the `when-applet-suspended` event is just to perform a database rollback. This ensures that any database locks are released. This is important since the applet could remain in the background cache for an extended period of time. It is undesirable that the applet is holding any database locks for this entire period. Due to the web browser architecture it is important that nothing is done in the handling of this event that can cause the display to change. Since the applet is already (being) placed in the background cache when this event is raised it is no longer possible for the applet to change its appearance. In fact, if the applet tries to change its appearance it might crash the Java Runtime Environment in some versions and browsers. This is why the default handling of OraFormsFaces for this event is kept to a bare minimum.

13.2. Resuming an Applet Instance

OraFormsFaces' default handling of the `when-applet-activated` event is to exit all running forms without saving any changes or doing any validation. Since all forms are actually started from a special OraFormsFaces

"landing form" this will return control to this landing form. That special landing form will then determine which form should be shown in the applet on this current page and call that particular form.

OraFormsFaces keeps track of all windows being opened and to which Form modules these windows belong. This is important since an applet can contain multiple forms through the use of `CALL_FORM` or `OPEN_FORM`. OraFormsFaces will use this information to close all running forms and will do so in the correct order.

13.2.1. Preventing OraFormsFaces from Closing Forms during Resume

As explained, OraFormsFaces will normally close all open Forms until control is returned to the `OFF_LAND` landing form. However, there are scenarios where you might not want to close a Form but simply keep it running.

As an example, let's say you have a JSF application using OraFormsFaces on a JSF page to embed the `Customer.FMX` form. The ID of the customer to query and edit in the form is determined by JSF and passed to Forms using an OraFormsFaces `FormParameter` component. The user navigates to this page to review and edit the details of customer 12345 and navigates away from this page to another JSF page. Later in the session the user returns to the same JSF page and OraFormsFaces will resume the old applet. If the user still has the same customer selected in the JSF application you might want the old applet to simply resume with any outstanding changes still showing. On the other hand, if the user has selected a different customer you do want the default handling of closing all Forms and opening a fresh new Form automatically querying the correct customer.

OraFormsFaces supports this (and other) scenarios through a hook in the `offCustom` PL/SQL package. When the applet resumes OraFormsFaces will perform the following steps:

1. The applet will detect it is resuming and will fire the `when-applet-activated` event.
2. The `when-applet-activated` event is handled by `offTriggers.whenAppletActivated`.
3. This internal OraFormsFaces procedure will call the custom function `offCustom.shouldFormClose` function. The implementation of `offCustom.shouldFormClose` can be customized by the application developer and determines if the current form should exit.
4. If `offCustom.shouldFormClose` returns true:
    a. The active Form is exited using `exit_form(no_validate)`
    b. If another custom Form is running, we return to step 2
    c. If the OraFormsFaces landing form (`off_land.fmx`) is the only remaining running Form it will start the forms module specified in the Form JSF component's `FormModuleName` property.
5. If `offCustom.shouldFormClose` returns false:
    a. Control is returned to the end user and no (more) forms are closed or started.

The `offCustom.shouldFormClose` function has four arguments: all properties of the previous `Form` JSF component that used this applet instance, all properties of the `FormParameter` JSF components associated with the previous `Form` JSF component as well as all properties and `FormParameter` properties of the current JSF components reusing this applet instance. This information can be used by the `shouldFormClose` function to determine whether or not the current form should be closed. It can check if the previous and current JSF components want to show the same Form module and/or if they use the same parameters and parameter values.

13.3. Preventing Applet Instance Reuse Between Specific Pages

As described above the default behavior of OraFormsFaces is to try to reuse a single Forms applet instance across all pages of an entire web application session. There might be scenarios where you do not want this behavior and want to ensure different (groups of) web pages use distinct Oracle Forms applet instances. For example, you might have a web page showing a very specific order entry form using Oracle Forms and OraFormsFaces. The

user can navigate away from the page and when she later returns you might use the technique from ***13.2.1 Preventing OraFormsFaces from Closing Forms during Resume*** to decide whether to close and reload the order entry form or not. It could be that you want the old form to remain untouched with perhaps pending changes from the last time the user visited the page. However if the user visited different web pages that use OraFormsFaces in the meantime, the applet instance would have been reused and is now showing a different form making it impossible to restore the order entry form as it was originally.

In situations like this you might want to use a dedicated Forms applet instance for the order entry page and a different Forms applet instance for the other pages. You can achieve this by setting the **UniqueAppletKey** property of the OraFormsFaces Form component in JSF. Only OraFormsFaces Form components with the same value for this property are allowed to share a Forms applet instance. So you can setup the order entry page to use a specific key. This would ensure the order entry page uses a dedicated Forms applet instance and all the other pages would be allowed to share a single Forms applet instance. You can use this technique to force distinct Forms applet instances for (groups of) pages.

Be aware that each Forms applet instance will consume resources at the client computer as well as the Forms server and database server as each applet instance also requires a Forms server session and thus a database session.

14. System Administration

14.1. Configuration Files

OraFormsFaces has an impact on a number of configuration files in the Oracle Forms and JSF environment. This section describes these configuration files and what changes OraFormsFaces requires.

14.1.1. Oracle Forms formsweb.cfg File

Most of Oracle Forms' configuration is stored in the `formsweb.cfg` file. Oracle Forms 10gR2 stores this file in `ORACLE_HOME/forms/server/formsweb.cfg` while Oracle Forms 11g has a more complex directory structure and stores the file at `DOMAIN_HOME/config/fmwconfig/servers/WLS_FORMS/applications/formsapp_11.1.1/config` where DOMAIN_HOME refers to `FMW_HOME/user_projects/domains/ClassicDomain/`

Each Forms application can have its own configuration section in the `formsweb.cfg` file. OraFormsFaces has added its own configuration section to this file specifying all properties relevant to OraFormsFaces. Each property has extensive documentation in the configuration file itself. Any properties not specified in a specific section are taken from the `[default]` section.

14.1.2. Oracle Forms Servlet Base Template Files

The `formsweb.cfg` file specified the base HTML file to use with the `baseHTML` and `baseHTMLjpi` parameters. The Oracle Forms Servlet handles all incoming HTTP requests to start an Oracle Forms application. The base HTML files that ship with Oracle Forms contain the HTML required to start the Oracle Forms applet.

OraFormsFaces comes with its own base HTML files that do not actually contain HTML. Instead they contain JavaScript. The call to this JavaScript is included in the JSF page that uses an OraFormsFaces Form component. Using JavaScript allows OraFormsFaces to be embedded in other pages instead of serving a complete HTML page itself. On the other hand it still uses a call to the Forms Servlet which is vital to set some Forms server side variables and to remain compatible with Oracle Forms Single Sign On.

The base files that ship with OraFormsFaces include the JavaScript to construct the applet and all its parameters. It is possible to add additional parameters in a similar way. An example of this approach is included in section ***17.3 Disabling Parts of OraFormsFaces***.

14.1.3. JSF Application web.xml File

Each Java web application contains a web.xml which contains configuration information about the application. OraFormsFaces uses a number of environment entries in this web.xml for its configuration. The `com.commit_consulting.oraformsfaces.FORMS_SERVLET_URL` environment entry is mandatory and specifies the URL to the Forms Servlet. See ***4.1 Preparing the JDeveloper Workspace*** for setting this up.

OraFormsFaces also has a number of optional environment entries that can be specified in this web.xml file. These are explained at ***14.2.2 Custom Java Class to Determine Database Credentials*** and ***17.3 Disabling Parts of OraFormsFaces***.

The default values of these environment entries are specified in the web.xml file that is bundled with the web application before being deployed to an application server. It could be that the values of these entries should differ for different deployments. The same web application might be deployed to testing, staging and production environments each requiring its own settings. Luckily most application servers offer a management console or other means to override these web.xml values during or after deployment. Refer to the documentation of your application server on how to achieve this, possibly using deployment plans.

14.2. Database Sessions and Credentials

Each Forms applet instance in the client relates to a single dedicated session at the Forms server which has its own dedicated session at the Oracle database. This session is independent of any session (pool) used by the JSF server to connect to the same or another database.

As described in ***3.5.1 Configuring Forms Server*** the default configuration of OraFormsFaces uses a fixed username and password to connect to the database. This is the easiest to setup and is similar to what most web applications use. Most web applications will use some sort of database session pooling and all sessions in that pool will use the same credentials to logon to the database regardless which web user is currently using the session from the pool.

Nonetheless, it might not be desirable to have Oracle Forms use the same credentials for each end user. It could be that some of the database PL/SQL logic or authorization is based on the database session being from the actual end user. If this is a requirement there are a number of alternatives:

- Do not specify any value for the `userid` parameter in `formsweb.cfg`. This will prompt the user for the database credentials when the Forms applet is initially started. The credentials do not have to be re-entered when the applet is suspended and resumed on other pages. Nonetheless this might not be a desirable solution since the user probably already had to logon to the JSF application and logging on twice can be confusing and is not very user friendly.
- Configure Oracle Forms to use Oracle Signe Sign On as described in ***14.2.1 Using OraFormsFaces with Single Sign On***.
- Use a custom Java class to determine the Forms Database credentials based on the JSF application state as described in ***14.2.2 Custom Java Class to Determine Database Credentials***.

14.2.1. Using OraFormsFaces with Single Sign On

Oracle Single Sing On can be used to protect both the JSF application and the Oracle Forms Server. This will prompt the user for credentials only once when the JSF application is first started. The Oracle Forms applet used by OraFormsFaces will detect this existing Single Sign On session and will retrieve the database credentials for the Forms session from the Oracle Internet Directory (LDAP) server.

For a detailed description on how Single Sign On works with Oracle Forms and how to configure it when not using OraFormsFaces see the Oracle Application Server Forms Services Deployment Guide at http://download.oracle.com/docs/cd/B25016_04/doc/dl/web/B14032_03/toc.htm

Configuring your OraFormsFaces to use Single Sign On is very similar to setting up a normal Oracle Forms application for SSO as described in the Oracle documentation. However, there are a few things to keep in mind:

- You need to set `ssoMode` to true in the OraFormsFaces specific configuration section of the `formsweb.cfg` or set it to true in the default section enabling SSO for all configurations
- Remove the default `userid` setting from the OraFormsFaces specific configuration section. With SSO enabled, the Forms server will get the necessary credentials by looking them up in the Oracle Internet Directory based on the currently logged on SSO user.
- It is important to keep the `ssoDynamicResourceCreate` property set to `false` as it was originally shipped with OraFormsFaces. When setting this to `true` the user would be forwarded to a page to create a new OID resource when none exists. This would not work with OraFormsFaces as the result from the Forms Servlet call is not displayed to the end user but is interpreted as JavaScript.

- Having `ssoDynamicResourceCreate` set to `false` means your system administrator has to setup resources for all authorized users prior to giving them access to the application. As described in the Oracle documentation you could also setup a shared resource to be used by everyone but that would have little added benefit over just disabling SSO and using a fixed username and password just as OraFormsFaces was originally configured.
- You could set the `ssoErrorUrl` parameter to a URL that returns a JavaScript to show a meaningful alert to the end user. This is also a feature that could be added to OraFormsFaces if desirable. If you are in this situation please contact us.
- Be sure to also protect your JSF application with SSO. This ensures that the user is already logged into SSO by the time she requests a page that uses OraFormsFaces/Oracle Forms. This will cause the Forms Servlet to just revalidate the SSO session and serve the requested JavaScript. If the user does not have a valid SSO session the Forms Servlet would redirect to the SSO logon page. This would cause problems as the client browser is expecting a JavaScript response from the Forms Servlet, not a logon page. By protecting the JSF application with SSO you make sure the user already has a valid SSO session and you prevent this situation. In development environments this may cause a challenge as your embedded OC4J running the JSF application is not capable of participating in a SSO session. If you still want to use SSO protected Forms in this configuration make sure to logon to whatever SSO enabled page before requesting the OraFormsFaces JSF page from your local OC4J.

14.2.2. Custom Java Class to Determine Database Credentials

OraFormsFaces version 3.0 introduced a new option for determining the database credentials for the Forms session. It allows a custom Java class at the JSF server to determine the credentials. These credentials are encrypted before they are transmitted to the Forms server. A custom `ON-LOGON` trigger in the OraFormsFaces `off_land.fmx` landing form will decrypt this information at the Forms server and use the decrypted information to setup a database session.

This setup allows each user to use its own credentials to logon to the database which is similar to traditional Oracle Forms. This allows full use of database roles, auditing, journaling and other database side functions that require individual user accounts.

Follow the next steps to setup this feature:

- Run the utility as described in ***11.1 KeyGenerator*** to generate a random AES encryption key.
- In the JSF application create a Java class that extends `FormsCredentialsImpl` and uses the randomly generated AES encryption key:

```
package com.example.app

import com.commit_consulting.oraformsfaces.component.crypto.*;
import java.security.GeneralSecurityException;
import javax.crypto.Cipher;
import javax.faces.context.FacesContext;

public class SecureCredentials extends FormsCredentialsImpl {
    protected Cipher createCipher() throws GeneralSecurityException {
        // use your own randomly generate key here
        return createCipher("8226B77F27119CA2A7CC2BB777CBB425");
    }
    protected String getUsername(FacesContext facesContext) {
        // example returns fixed credentials, real implementation would
        // determine credentials at runtime
        return "scott";
    }
    protected String getPassword(FacesContext facesContext) {
```

```
        // example returns fixed credentials, real implementation would
        // determine credentials at runtime
        return "tiger";
    }
    protected String getConnectString(FacesContext facesContext) {
        // example returns fixed credentials, real implementation would
        // determine credentials at runtime
        return "orcl";
    }
}
```

- Implement the `getUsername`, `getPassword` and `getConnectString` methods to determine the database credentials. These methods will likely user information about the current user of the JSF application through `FacesContext`, `FacesFacesContext.getExternalContext`, `ExternalContext.getInitParameter`, `ExternalContext.getUserPrincipal`, `ExternalContext.isUserInRole`. This information can be used to lookup the credentials in some secure store like LDAP, JNDI, or a secured database table.

> **Note:** Ensure the Java class has a public no-argument constructor as the OraFormsFaces framework has to be able to initiate an object. This can also be accomplished by not specifying any constructor in the source code having the Java compiler create a single no-argument constructor.

- When using Oracle Forms 11 or newer in combination with Oracle Database 10.2 or newer, these methods can also return credentials to use Proxy Authentication. See the Oracle Database Security Guide for more information on Proxy Authentication. A (simplified) example of Proxy Authentication is below:

```
protected String getUsername(FacesContext facesContext) {
    // return the proxyuser and the "real" username between square brackets
    // this will let proxyuser proxy to the "real" user without ever needing
    // to know the password of the "real" user
    return "proxyuser[" + getJsfUsername() + "]";
}
protected String getPassword(FacesContext facesContext) {
    return "SecretPasswordOfProxyuser";
}
protected String getConnectString(FacesContext facesContext) {
    return "orcl";
}
```

- Register an environment entry in the `WEB-INF/web.xml` configuration file to specify the name of the class implementing the `FormsCredentialsProvider` interface.

```
<env-entry>
    <description>Name of the class to provide credentials for Forms database session</description>
    <env-entry-name>com.commit_consulting.oraformsfaces.FORMS_CREDENTIALS_CLASS</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>com.example.app.SecureCredentials</env-entry-value>
</env-entry>
```

- Create a database user with sufficient privileges to decrypt the information:

```
create user oraformsfaces identified by secret;
grant create session to secret;
grant execute on dbms_crypto to oraformsfaces;
```

- Edit the Forms server `formsweb.cfg` file and edit the `userid` parameter in the `[OraFormsFaces]` section to specify the credentials of the user allowed to decrypt the information as well as the secret AES key used for the encryption. The AES encryption key must be enclosed by brackets and be used as a prefix to the username:

```
userid={8226B77F27119CA2A7CC2BB777CBB425}oraformsfaces/secret@orcl
```

Here is what happens at runtime:

- When the Forms applet starts, it will always start the `off_land.fmx` OraFormsFaces landing form first.
- The OraFormsFaces landing form has an `ON-LOGON` trigger that checks the specified `userid` parameter. If it is empty or contains normal credentials, the `ON-LOGON` trigger will simply call the default Forms handling of using these credentials or prompting the user for credentials.
- If the `userid` starts with an encryption key enclosed in curly brackets, the specific OraFormsFaces feature to get the credentials will kick in.
- The `ON-LOGON` trigger will use the credentials from the `userid` parameter to start a temporary database session.
- `DBMS_CRYPTO.randomNumber` is called to generate a secure random number.
- The `ON-LOGON` trigger uses the JavaScript API to make an AJAX (Asynchronous JavaScript and XML) call back to the JSF server including the randomly generate token.
- The AJAX call is intercepted by an OraFormsFaces `PhaseListener` at the JSF server.
- The `PhaseListener` will construct the class specified in the web.xml environment entry and will ask the class for encrypted credentials. These credentials are returned to the calling `ON-LOGON` trigger of the Forms applet.
- The `ON-LOGON` trigger continues to run at the (secure) Forms server and uses `DBMS_CRYPTO.decrypt` to decrypt the credentials.
- The temporary database connection is closed and the decrypted credentials are used to setup to final database connection.
- The `ON-LOGON` trigger finished and lets the applet continue its normal initialization with the new database session.

14.3. Combining OraFormsFaces and Oracle WebUtil

OraFormsFaces can be used in conjunction with Oracle WebUtil. There are a number of remarks or changes to the normal procedure for installing and configuring WebUtil as described in the Oracle Forms documentation:

- Be sure to use a webutil version that is compatible with your Forms version and Forms patchlevel. Different patchsets of Oracle Forms might require different versions of WebUtil.
- Be sure to compile `webutil.pll` to `webutil.plx` as described in the Forms documentation. Failure to do so will trigger a Forms bug which tries to interpret the `forms/webutil` directory as a library file causing an ORA-06508 error at startup.
- Uncomment the lines in the OraFormsFaces section of the `formsweb.cfg` file related to WebUtil. The OraFormsFaces HTML template should not require any edits as the WebUtil parameters are already included.

- Edit the `webutil.cfg` file at the Forms Server and change the `install.syslib.location` parameter to an absolute URL, for example `http://server.example.com:8888/forms/webutil`. Failure to do so will cause the client to try downloading the necessary DLL files from the JSF server instead of the Forms server.
- On Windows Vista or Windows 7 the client browser needs to run with administrative privileges to enable it to store the downloaded DLL files in the JVM directory. See Oracle Support Note 783937.1 on how to change the client download directory to a non-privileged directory.

14.4. Deploying OraFormsFaces applications

14.4.1. Deploying the JSF Application to a 10.1.2 OC4J or Application Server

It is possible to develop your OraFormsFaces application with JDeveloper 10.1.3.x, but still deploy it to a 10.1.2 application server. This can be particularly interesting if you run Oracle Forms 10gR2 which uses Oracle Application Server 10.1.2 and you do not have a 10.1.3 Oracle Application Server available (yet).

Use these steps to change your JDeveloper 10.1.3 project so it can be deployed to a 10.1.2 Application Server:

- Create a new OC4J instance in the Oracle Application Server. You can do this from the Application Server Management Console or see the administrator's guide for details.
- Disable ADF Runtime for this OC4J instance by modifying its `OC4J_INSTANCE_HOME/config/application.xml`. Comment out the following lines:

```
<library path="../../../BC4J/lib"/>
<library path="../../../jlib/ojmisc.jar"/>
<library path="../../../ord/jlib/ordim.jar"/>
<library path="../../../ord/jlib/ordhttp.jar"/>
<library path="../../../jlib/jdev-cm.jar"/>
```

  and

```
<library path="../../../uix/taglib"/>
```

- If necessary, set up Data Sources and other configuration settings for OC4J.
- Make sure the new OC4J container is started.
- In the **Model** project properties, go to **Libraries**. Note all the checked libraries.
- In the **`ViewController`** project properties, go to **`Libraries`** as well and add the libraries that where checked with the **`Model`** project.
- When compiling, deploying, or running your project later on you might discover that additional libraries are needed. `BC4J Runtime` and `BC4J HTML` are likely candidates.
- In the **ViewController** project, create a new WAR Deployment Profile (**File** > **New** > **General** > **Deployment Profiles** > **WAR File**).
- In the deployment profile properties, go to **WEB-INF/lib Contributors**, and select all the libraries (except `JSP Runtime`). This ensures that all necessary JAR files are included in the WAR/EAR file.
- In the deployment profile properties, go to **WAR Options**, and select **Compress Archive**.
- If you use Subversion, go to **all Filters** categories, then to **Patterns**, and exclude `**/.svn/`
- In **ViewController/Web Content/WEB-INF/web.xml**, replace

```
<web-app
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee">
```

with

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
```

- If the `web.xml` file has a `<jsp-config>` tag remove it, as it is not valid in the 2.3 file format.
- If the **ViewController** project already contains pages, you need to change them to use JSP version 1.2. For every page replace

```
<?xml version='1.0' encoding='windows-1252'?>
```

with

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

For every namespace declaration in the `<jsp:root>` tag add a `<%@ taglib>` tag just after the `<!DOCTYPE>` tag. This would mean that

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://xmlns.oracle.com/adf/faces" prefix="af"%>
<%@ taglib uri="http://xmlns.oracle.com/adf/faces/html" prefix="afh"%>
<%@ taglib uri="http://commit-consulting.com/OraFormsFaces/tags"
prefix="off"%>
```

would be added before a `<jsp:root>` tag that looked like

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces"
          xmlns:afh="http://xmlns.oracle.com/adf/faces/html"
          xmlns:off="http://commit-consulting.com/OraFormsFaces/tags">
```

Now remove the entire `<jsp:root>` tag and be sure to also remove the closing `</jsp:root>` tag at the end of the page.

Remove the `<jsp:output>` tag

If available, remove the `<jsp:directive.page>`. If it specified a `contentType`, add a `<%@page>` tag with the same `contentType`. This would mean that

```
<jsp:directive.page contentType="text/html;charset=windows-1252"/>
```

is replaced by

```
<%@ page contentType="text/html;charset=windows-1252"%>
```

- On your development PC be sure JDK 1.4 is available (for example use the one that ships with Forms 10.1.2 in `FORMS_HOME/jdk`)
- In the **ViewController Project Properties**, go to **Libraries**.
- Click the **Change button** for the J2SE Version.
- Choose if you want to define the new J2SE definition at Project level (every deployer needs to have JDK 1.4 in same path), or User level (every deployer needs to create own J2SE definition).
- Select the Project or User folder and click the **New** button to create a new J2SE Definition.
- Browse to your `[JDK1.4Home]/bin/java.exe` for the J2SE Executable.
- When asked to install OJVM, click **OK**.
- Accept the default settings for name, class path, source path, and doc path.
- Be sure the new 1.4 JDK is selected before dismissing the dialog.
- Save the settings.
- **Warning**: after this step (usage of J2SE 1.4), do not try to run Embedded OC4J (see below how to enable Embedded OC4J again).
- In the **Model Project Properties**, do the same (define a similar J2SE 1.4 Version) at Project or User level.
- In the **ViewController Project Properties**, go to **Compiler** and set both **Source** and **Target** to `1.4`.
- Repeat these steps for the **Model** Project, to also set it to Java 1.4 syntax.
- Empty your `classes` folder (the output directory for the compiler) to be sure any 1.5 compiled classes are removed. You can do that in JDeveloper by selecting the **ViewController** project and then via the menu option **Run** - **Clean ViewController.jpr**.
- Empty the `classes` folder for the **Model** project as well.
- Right-click the deployment profile and choose **Deploy to EAR**. The log tells you where it creates the EAR file.
- Deploy the EAR file to the new OC4J instance. You can either do this from the Application Server Enterprise Manager or directly from JDeveloper. Deploying from the Enterprise Manager or command line is more informative than JDeveloper if something goes wrong. Setting up a connection to a 10.1.2 application server from JDeveloper can be a bit tricky. See Oracle Support Note 293029.1 for details. One of the most common issues is that the Oracle home path is case sensitive, even on Windows. To get the correct case on Window see how the Oracle home directory is specified in the windows registry on the Application Server.
- When running the application be sure to use a URL that includes the `/faces/` directory as that triggers the necessary Faces listener. The URL should probably look something like `http://server.example.com/application/faces/page.jsp`

Note:	If you want to run the project with the JDeveloper Embedded OC4J be sure to change back to J2SE 1.5. Also be sure to clear all projects when switching J2SE version.

14.4.2. Deploying the JSF Application to a 10.1.3 OC4J or Application Server

An OraFormsFaces enabled JSF application can be deployed to a 10.1.3 Oracle Application Server just as any other JSF application. You just need to be sure the OraFormsFaces JAR file is included in the WAR/EAR file when building the application.

Refer to the JDeveloper and OC4J documentation for a more detailed explanation of building, packaging, and deploying J2EE applications.

14.4.3. Deploying the JSF Application to a 10.3 WebLogic Server

An OraFormsFaces enabled JSF application can be deployed to a 10.3 WebLogic Server just as any other JSF application. You just need to be sure the OraFormsFaces JAR file is included in the WAR/EAR file when building the application.

Refer to the JDeveloper, Fusion Middleware and WebLogic documentation for a more detailed explanation of building, packaging, and deploying J2EE applications.

15. OraFormsFaces as an Oracle Forms Migration Strategy

The applet clipping feature can be used for users that do not want the feature rich user interface of Oracle Forms, or it can be used as a method for a smooth and gradual migration from Oracle Forms to true JSF web development. You could convert an Oracle Forms application consisting of 300 Forms to a JSF application consisting of 300 web pages that each embeds a single form. Over time, you could migrate individual forms to 100% JSF/ADF web pages. This migration can be done on a one-form-at-a-time basis, whenever there is a business need to change or adopt the form. This way of migrating from Oracle Forms to JSF/ADF has a number of advantages:

- From day one, the JSF application embedding Oracle Forms has the intuitive user experience users nowadays expect from applications.
- You can migrate from Oracle Forms to JSF/ADF at your own pace.
- There is no need for a big-bang rewrite of your entire application in a new technology.
- You can develop new functionality in JSF/ADF immediately. There is no "lock in" into Oracle Forms. It's perfectly possible to have a hybrid application consisting of pure JSF/ADF and JSF pages that embed Oracle Forms. You can pass context (like selected customer ID) and events between the JSF application and the Oracle Forms modules.

This is just a very brief overview of the approach to Oracle Forms Migration and some of the advantages. This subject will be discussed in more detail in future versions of the Developer's Guide and online at http://www.commit-consulting.com/

16. Integration with other web technologies

OraFormsFaces is a component library based on Java Server Faces (JSF) technology. It can be used in a standards compliant JSF environment or in combination with ADF Faces, Oracle's implementation of the JSF standard. See all the other sections in this manual on how to work with OraFormsFaces in a JSF environment.

But with a little extra effort OraFormsFaces can also be used in combination with other Oracle, or non-Oracle web technologies. The most notable ones are discussed in this chapter.

16.1. Oracle WebCenter

Oracle WebCenter is an additional-fee option for Oracle's Application Server. It is a bundle of different technologies, including components to consume and produce JSR-168 and WSRP 2.0 compliant portlets. With Oracle WebCenter you can consume portlets in a JSF application, but more importantly you can create standards compliant JSF based portlets.

With this technology it is possible to use Oracle WebCenter, JSF, and OraFormsFaces to create a portlet that embeds Oracle Forms. That portlet can be consumed in any JSR-168 or WSRP 2.0 portal environment, which could be another WebCenter application or any other portal.

16.2. Oracle Portal

Oracle Portal, a member of the Oracle Fusion Middleware family of products, offers a complete and integrated framework for building, deploying, and managing enterprise portals. Oracle Portal can consume portlets and show them in an integrated portal environment. If we can create a portlet that uses OraFormsFaces to embed an Oracle Forms form, we could consume it in an Oracle Portal environment.

The problem is how to create a portlet that uses OraFormsFaces and that is compatible with the standards supported by Oracle Portal. You could use the aforementioned Oracle WebCenter product, but this is an expensive Application Server option if all you're going to do is create an OraFormsFaces portlet.

There is a more low-tech, low-cost alternative. Using JDeveloper, JSF, and OraFormsFaces you can create a very simple webpage that embeds a form using the OraFormsFaces components. This web page can be consumed in an Oracle Portal portlet that uses an IFRAME to include the JSF, and OraFormsFaces based web page.

You can end up with an Oracle Portal application that embeds Oracle Forms in one or more portlets. In this scenario, OraFormsFaces offers a key feature. Only the very first portal page that embeds an Oracle Form starts a Java applet instance. The startup time of the Oracle Forms Java applet can be several seconds. You don't want this startup time on each portal page that embeds a form. With OraFormsFaces, the forms applet is not destroyed when navigating to another page. The applet is kept running in the background and is reused on any other Oracle Portal page that embeds a form. This eliminates the Java applet startup time on all subsequent portal pages that embed a form.

It is possible to pass parameters to the Oracle Portal portlet, which can pass the parameters on to the URL in the SRC attribute of the IFRAME. This passes the parameters to the call to the JSF page on the JSF web server. That JSP/JSF page can use the HTTP request parameters in the OraFormsFaces components.

If you are interested in this solution, please contact Commit Consulting for more details and assistance. There have been companies that implemented this solution to integrate Oracle Forms into Oracle Portal. Also, see the section "Third party web technologies" on page 133 for a more technical description of the IFRAME based solution.

16.3. Third party web technologies

OraFormsFaces can even be used to integrate Oracle Forms with any other web technology, for instance PHP or .Net. The easiest way to use the same IFRAME solution as described in the Oracle Portal section.

A PHP (or any other web technology) page could include an IFRAME tag. That tag creates an inline frame that contains another document. The URL of this other document is specified with the SRC property of the IFRAME. That SRC property can point to a URL on the JSF web server. The page on the JSF server can use the OraFormsFaces component library to create a HTML document that embeds a form. Even parameters can be passed from the PHP page, to the IFRAME, to the JSF page, to the OraFormsFaces component.

Below is a simplified example of such a scenario. It all begins with a simple PHP page in which you want to embed an Oracle Forms form:

```
<html>
  <body>
    <?php
      echo "<iframe src='http://example.com/offaces.jspx?custid=$custid'/>";
    ?>
  </body>
</html>
```

This will result in the following HTML being sent to the client browser:

```
<html>
  <body>
    <iframe src='http://example.com/oraformsfaces.jspx?custid=1234'/>
  </body>
</html>
```

As a result the web-browser will request the URL specified in the SRC attribute of the IFRAME tag. That is a request to a JSF/JSP page on the web server. Its source might look something like:

```
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns: off="http://commit-consulting.com/OraFormsFaces/tags">
  <f:view>
    <html>
      <body>
        <h:form>
          <off:form weight="480" height="240" formModuleName="customer.fmx">
            <off:FormParameter value="#{param.custid}"/>
          </off:form>
        </h:form>
      </body>
    </html>
  </f:view>
</jsp:root>
```

As you can see this is a very clean JSF page that just returns a page with a single OraFormsFaces Form component and a **FormParameter** component. The value of the **FormParameter** component is retrieved from the HTTP request parameter `custid`, which is part of the URL requested by the client browser.

16.4. Integration without JSF components

The IFRAME approach as described in the previous section still uses a JSF compliant web server to host the content of the IFRAME. If this is undesirable you could also opt for an approach that completely eliminates the JSF server from the architecture.

The most important OraFormsFaces features are implemented in the supporting JavaScript, the applet and PJC enhancements and the supporting PL/SQL code. These are all technologies that can be used without using the JSF components themselves. The JSF components are merely a very convenient way of delivering these capabilities to the web application.

Assuming you are technically proficient, you can build a simple JSF based application and inspect the HTML code it is rendering to the client web browser. You could make sure you generate the same HTML from your custom PHP, RubyOnRails, .Net, or whatever other technology you are using. There's just some more work involved to get this up and running and it requires additional work when upgrading to a newer version of OraFormsFaces. With newer versions of OraFormsFaces the HTML and JavaScript being included in a rendered page is likely to change. With this approach you would have to change your custom code to render the new HTML as well.

17. Troubleshooting OraFormsFaces

17.1. Common Issues

This section lists common error message and issues you may encounter when using OraFormsFaces.

17.1.1. Resuming the OraFormsFaces applet crashes the Forms Server process with Assertion Failed @ ifrlf.c Line 898

This is due to Oracle Forms bug 4502472. This bug causes the Forms server to lose track of the correct current item when the applet is suspended and later resumed by using legacy_lifecycle like OraFormsFaces does.

Oracle Forms bug 4502472 was filed against Forms version 9.0.4.2 and is also present in version 10.1.2.0.2. Install the latest patch set and/or the latest "Forms Focus Super Merge" patch (see Oracle Support Note 730581.1) to resolve the issue.

17.1.2. FRM-40105: Unable to resolve reference to item OFF_LAND_BLOCK.OFF_LAND_DUMMY_ITEM

This is most likely the same issue as reported under 17.1.1.

17.1.3. java.lang.ClassNotFoundException: com.commit_consulting.oraformsfaces.extension.CommunicatorBean when running OraFormsFaces prepared Forms in traditional environment

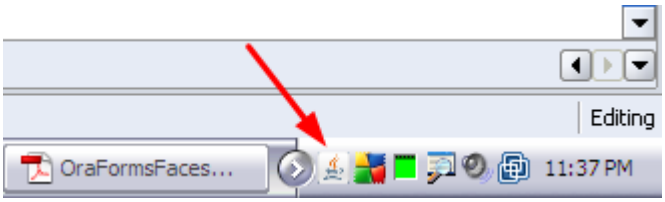
OraFormsFaces adds an object group to each of your Oracle Forms modules. When using Oracle Forms prior to version 11 this includes a Pluggable Java Component called a `CommunicatorBean`. When running such a prepared form in a traditional Forms application (without using the JSF components) an exception might be printed to the Java console at the client complaining about the missing `com.commit_consulting.oraformsfaces.extension.CommunicatorBean` class. This not only clutters the console log but also causes an additional network roundtrip to the forms server each time a Form starts since the applet will try to download the .class file.

By default the OraFormsFaces JAR file is only added to the archive parameters of the specific OraFormsFaces sections in the `formsweb.cfg` configuration file. You can safely add the same JAR file to the other sections as well. This makes the JAR file also available when running Forms in a traditional setup so the class can be found. The `CommunicatorBean` has been programmed to not perform any tasks when running outside of a full OraFormsFaces setup so there is no risk in having it available.

17.2. Diagnosing Issues

OraFormsFaces uses a number of different technologies. This can make it challenging to pinpoint a problem when things aren't working. This chapter will describe how to get more details on the working of the different components and some tips as to where to look.

The first important thing you have to determine is if the client web browser is at least trying to start a Java applet when you access an OraFormsFaces enabled page. The easiest way to detect this is whether an icon for the Java Runtime Environment appears in the system tray. For example:



The above screenshot is from a Sun JRE version 6 icon. When you're using version 5 or 1.4 the icon will look differently. If you see this icon you at least know that the browser is trying (or perhaps even succeeding) to start a Java applet. If this is the case, there are a lot of things you can rule out. Keep this in mind when looking at the next sections.

17.2.1. Inspecting the client side HTML

If the browser does not try to start a Java applet the HTML rendered to the client is probably wrong. The first thing to check is the source of the HTML page. Run the page and then view the HTML source in your browser. See if you can find the part about OraFormsFaces. It should look something like:

```
<!-- start of OraFormsFaces v3.0.3d20100110 Oracle Forms applet frm_test -->
<!-- Forms servlet URL: http://localhost:8889/forms/frmservlet?config=OraFormsFaces -->
```

If you cannot see this HTML but you can find the string `<off:form` then it means the OraFormsFaces JSP tags where never interpreted by the server. The most likely cause is that the OraFormsFaces JAR file is not included in the project or deployment.

It can also be you cannot find the comment tags, nor the `off:form` tag. This happens when OraFormsFaces is not part of the initial page load but is added to the page dynamically through something like ADF Rich Client components or other Ajax-style components. In these scenarios you need to inspect the HTML as it is currently running in the browser after all dynamic manipulation, not the HTML version as it was initially downloaded from the server. To inspect the runtime HTML use the Firebug add-on for Mozilla Firefox or the Developer Toolbar for Internet Explorer 7 or the built-in Developer Tools with Internet Explorer 8. With these tools you should be able to find the OraFormsFaces comment tag or the unparsed `off:form` tag.

The OraFormsFaces HTML also includes a `<script>` tag referring to the Forms Servlet URL you setup in the web.xml of the JSF project. For example:

```
<script src="http://localhost:8889/forms/frmservlet?config=OraFormsFaces
             &amp;oraformsfaces_baseURL=http%3A%2F%2Flocalhost%3A8889
             &amp;sess=T5FdLMFSV…2546408"
        id="oraformsfacesScript_frm_test"
        type="text/javascript">
</script>
```

You can copy the URL of this script tag and paste it in the address bar of the browser to see if it returns a JavaScript. This should be the baseOraFormsFaces JavaScript file you installed at the Forms server with all the %-placeholders replaced with their appropriate values by the Forms Servlet:

```
[[ lines removed for readability
/* get offJSONForm to construct */
var offBaseJSONForm = window.offConstructQueue.pop();
/* create new JSON object for Forms Applet */
var offBaseJSONApplet = {};
offBaseJSONForm.JSONApplet = offBaseJSONApplet;
/* set fixed properties of OraFormsFacesForm object */
offBaseJSONApplet.appletWidth  = htmlDecode("100%");
offBaseJSONApplet.appletHeight = htmlDecode("100%");
offBaseJSONApplet.ieClassId    = htmlDecode("...");
offBaseJSONApplet.ieCodeBase   = htmlDecode("http...");
offBaseJSONApplet.ieDelay      = htmlDecode("2500");
[[ lines removed for readability ]]
offBaseAppletParameters.push({name:"serverArgs",  value:offBaseServerArgs});
offBaseAppletParameters.push({name:"src",         value:htmlDecode("")});
offBaseAppletParameters.push({name:"pluginspage", value:htmlDecode("http...")});
[[ lines removed for readability ]]
```

This JavaScript injects the actual applet HTML into the web page at runtime. This means the applet HTML is not included in the initial HTML as send out by the JSF server. This also means you cannot see this HTML when viewing the HTML source of the page. If you do this, the browser will show the HTML as it was received from the JSF server, not as it currently is in memory. Use the aforementioned Firebug plug-in for Firefox, or the IE Developer Toolbar or IE Developer Tools to see the HTML as it currently is stored in the browser memory.

For Internet Explorer download and install the Internet Explorer Developer Toolbar. Once you did that you can view the runtime HTML by showing the Developer Toolbar and then View > Source > DOM (Page). If you use Mozilla Firefox you can download the Web Developer plugin or the Firebug plugin. With the Web Developer plugin you get a toolbar at the top of your Firefox window where you can select View Source > View Generated Source. When using Firebug there is a little icon in the Firefox system tray you can click to open Firebug. Then you can select the HTML tab to see the HTML as it currently is being displayed.

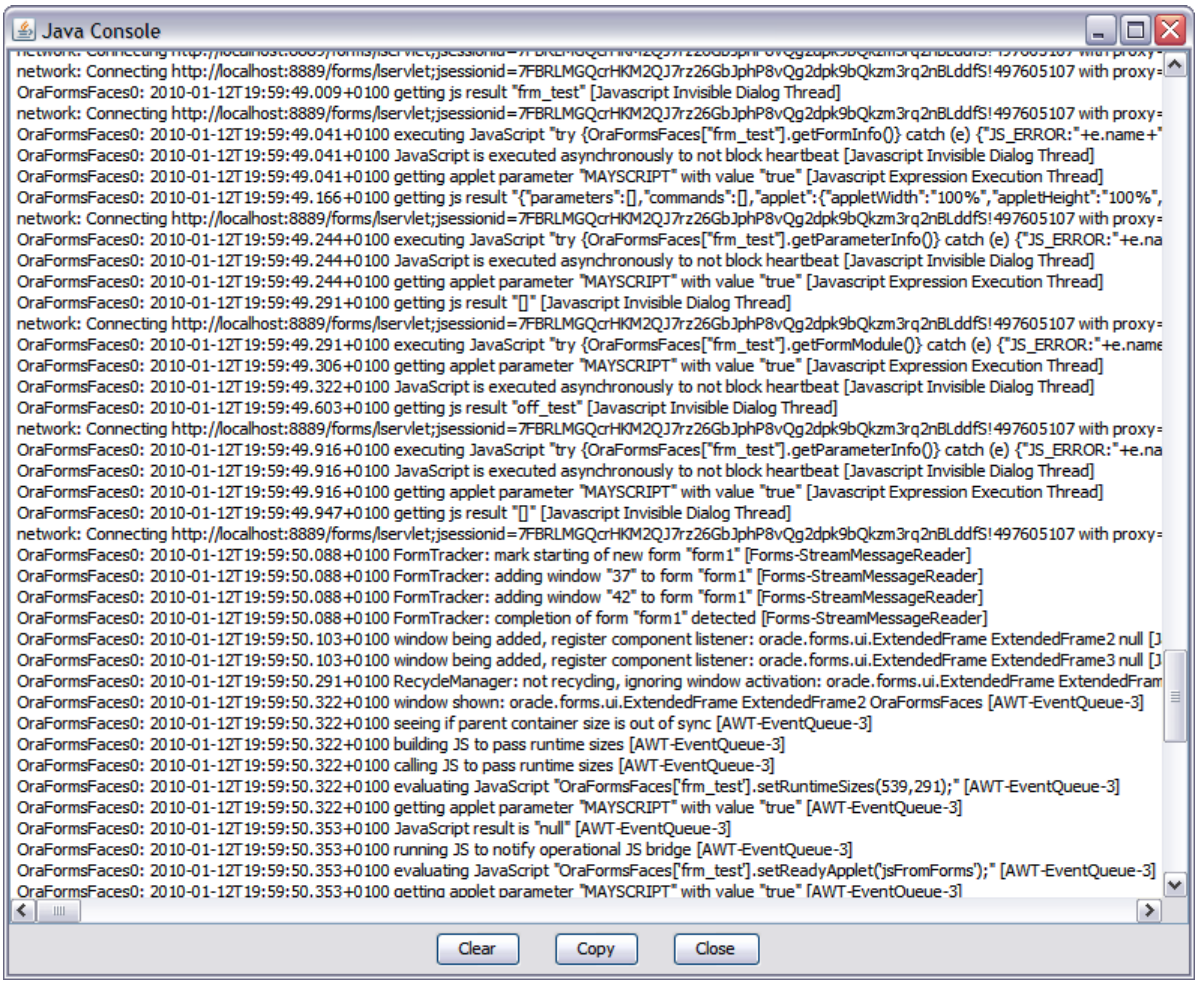
With these techniques you can see the HTML after the Forms JavaScript injected the applet tag. On Internet Explorer you should now see the `<object>` tag whereas Firefox uses an `<embed>` tag. You could check this HTML to see if all parameters have logical values. The main suspects are settings that are specific to your environment such as the different URLs.

17.2.2. Enabling OraFormsFaces Applet Logging

If the OraFormsFaces applet is at least trying to start or is actually succeeding in starting but there are other applet related problems, you could enable client side applet logging to get a better insight on what is happening and what might be causing problems.

To do this edit the `formsweb.cfg` file holding all Oracle Forms configuration. Set the `oraformsfaces_logging_parameter` in the `[OraFormsFaces]` specific config section to a value of `5`. This will set the logging to the highest level.

Once that is done close all browser windows and start a fresh browser. This ensures a new session is started with the logging enabled. Right click the Java icon in the system tray and enable the Java console. This will open a new Java console window with all the logging information as it is written by the client side OraFormsFaces code. While you're in the console you can also press the 5-key to set the logging of Sun's Java Runtime Environment to the highest level. This will cause Sun JRE to also write all sorts of detailed logging information to the console. An example of such a Java console:



Note:	Be sure to disable logging once you diagnosed the problem as having this logging enabled can have a negative impact on performance.

17.2.3. Enabling Client Side JavaScript Tracing

If you suspect the OraFormsFaces client side JavaScript to be causing problems, you can enable tracing for this as well. This logging leverages the logging capabilities of the Firebug add-on for Mozilla Firefox. If your issues are specific to Internet Explorer, OraFormsFaces will try to use its own logging window offering similar (but limited) logging capabilities to Firebug. It is best to try to reproduce any issues you might be having in Firefox and use the Firebug add-on for tracing.

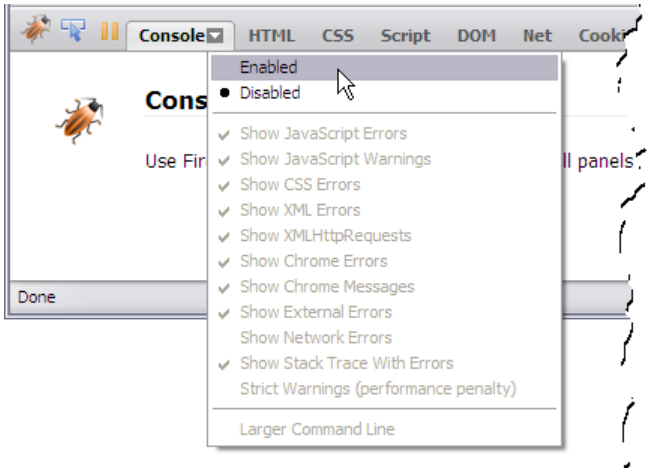
- Enable tracing in your JSF page by setting the JavaScript variable `ORAFORMSFACES_LOGGING` to `true`.
  - In plain JSF and ADF 10g applications use

```
<script type="text/javascript">ORAFORMSFACES_LOGGING=true</script>
```

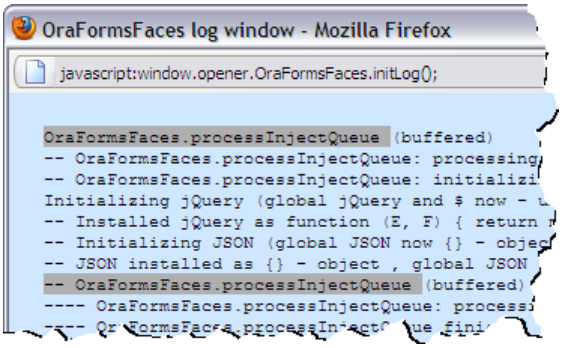
- In ADF 11 applications use

```
<af:resource type="javascript">ORAFORMSFACES_LOGGING=true;</af:resource>
```

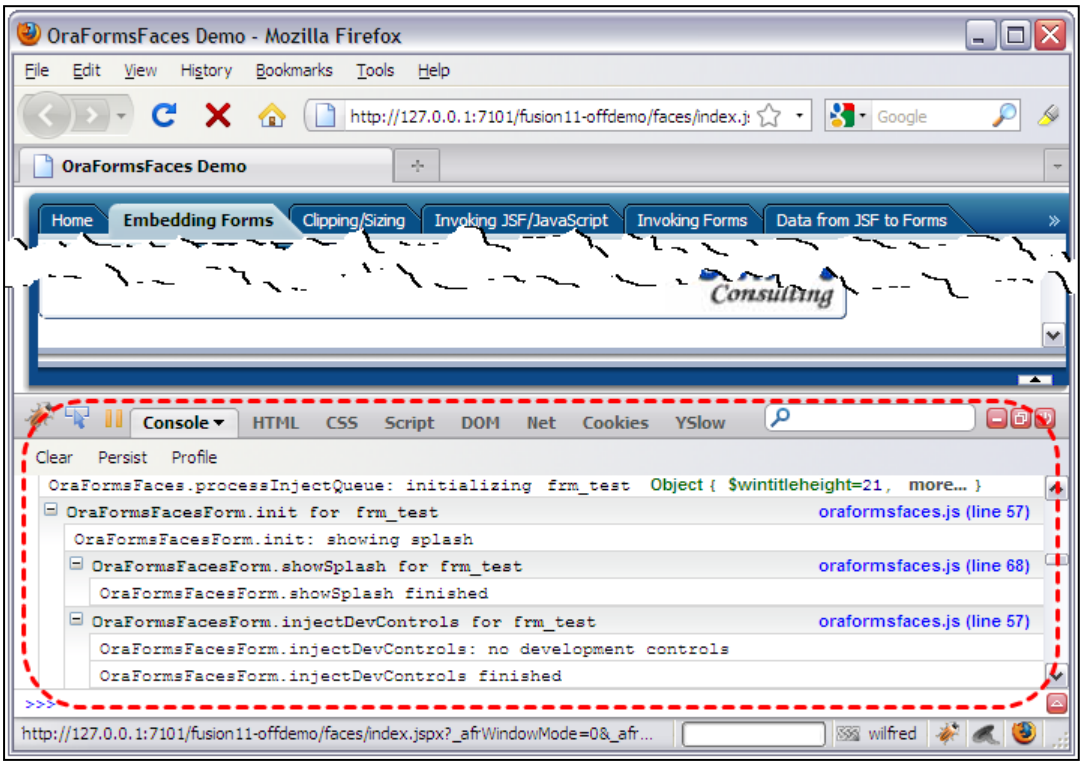
- To use the Firebug add-on in Firefox:
  - Download and install the Firebug add-on for your Firefox browser.
  - Start your application in Firefox.
  - Be sure to activate Firebug for this application before navigating to a page that uses OraFormsFaces:



  - If Firebug is not enabled for your application a popup window will appear showing the OraFormsFaces logging:



- If Firebug is installed and enabled the console will show detailed tracing information:



- To use OraFormsFaces tracing in Internet Explorer or in Firefox without Firebug simply start the application with the JavaScript to enable tracing included. If Firebug is not detected OraFormsFaces will open a popup window to show the tracing information. This is not as advanced as the Firebug console but it probably is sufficient for most scenarios.

> **Note:** Be sure to disable JavaScript tracing by removing the <script> tag once you diagnosed the problem as having this tracing enabled can have a negative impact on performance.

17.2.4. Enabling JSF Server Side Tracing

The OraFormsFaces JSF classes are heavily instrumented with Apache Commons logging. This means you can enable Apache Commons based logging on your JSF server to see detailed information on the JSF server side processing of your requests.

To enable this with the Embedded OC4J of JDeveloper 10.1.3, take the following steps:

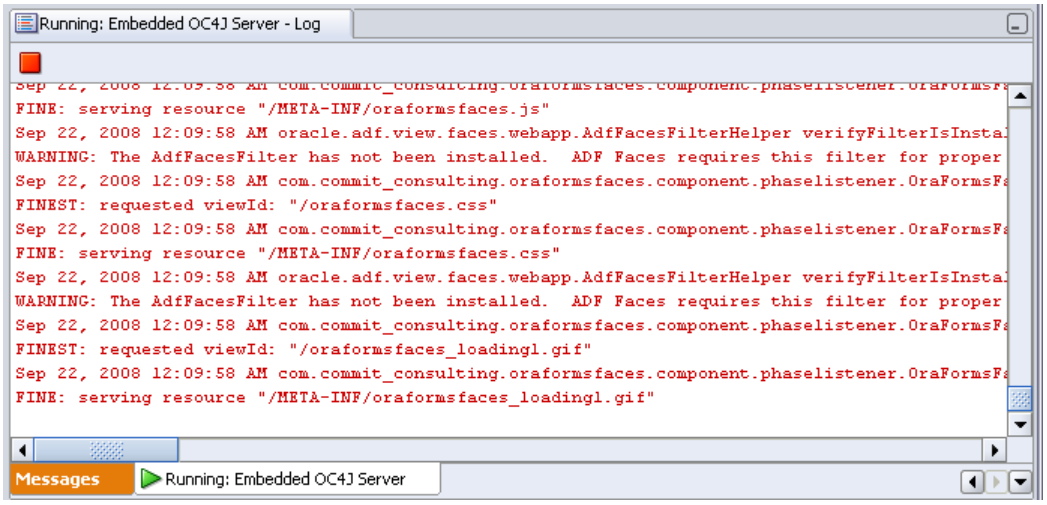
- Edit `JDEV_HOME/jdk/jre/lib/logging.properties`
  - Add the following line to print all levels of message from the OraFormsFaces classes:

```
com.commit_consulting.level = ALL
```

  - Be sure to set `java.util.logging.ConsoleHandler.level` to `ALL` and not the default of `INFO`, otherwise the logging of more detailed levels would still not be written to the console.
- If you stop any running embedded OC4J and then start your application again you should see detailed logging in the Embedded OC4J console in JDeveloper. If not, it might be that Apache Commons is using a different logger. To force it to use JDK logging add the following line to `JDEV_HOME/jdev/bin/jdev.conf` (and restart JDeveloper):

```
AddVMOption -Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.Jdk14Logger
```

- Once the logging works you should see something like the following example:



See Oracle Fusion Middleware Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server for details on how to setup Apache Commons logging with Oracle WebLogic server.

17.2.5. Enabling Forms Tracing

If you suspect an error at the Forms PL/SQL side of the application or OraFormsFaces it might be a good idea to enable Forms Tracing. See Oracle Fusion Middleware Forms Services Deployment Guide for information on setting up Forms Tracing.

17.3. Disabling Parts of OraFormsFaces

If the different tracing options from the previous sections could not give enough information to eliminate a problem and a bug or incompatibility is expected in OraFormsFaces you can disable parts of OraFormsFaces to eliminate the possibility of a bug or incompatibility in that component.

OraFormsFaces has a number of switches to disable parts of its functionality. Try to disable these one at a time to trim down on the possibilities. If you did disable a number of them and managed to circumvent the issue you are having, please try re-enabling most of the parts while still eliminating the issue before reporting an issue.

17.3.1. Web.xml Variables

Section ***14.1.3 JSF Application web.xml File*** explains how OraFormsFaces uses environment entries from the JEE application's web.xml for its configuration. Below are the additional parameters that can be specified in the `web.xml` file to disable parts of OraFormsFaces.

`com.commit_consulting.oraformsfaces.UNIQUE_APPLET_PER_SESSION`

Normally OraFormsFaces will include the ID of the JSF session to the applet declaration in the page. This ensures that a new JSF session always gets a fresh applet instance and is not allowed to reuse a suspended applet from the previous JSF session. Add this parameter to the `web.xml` file with a value of `false` to disable this behavior and allow applets to be reused across JSF sessions.

`com.commit_consulting.oraformsfaces._INITIAL_JS_TO_FORMS_CALL`

Normally OraFormsFaces will immediately send a (fake) JavaScript event to the applet at startup. This is especially meaningful for Firefox browser with older version of Java as the first JavaScript-to-Java call is slow in these situations. It's better to endure this delay at startup than when the user first tries to interact with the applet through JavaScript.

Add this parameter to the `web.xml` file with its value set to `false` to disable sending this initial JavaScript event.

com.commit_consulting.oraformsfaces._DISPLACE_DURING_INIT

Normally OraFormsFaces positions the applet out of sight while it is initializing. This prevents issues with some Firefox version showing a black area that even breaks through the loading animated image. Once the applet is fully initialized it is returned to its normal position.

Add this parameter to the `web.xml` file with its value set to `false` to disable the (re)positioning of the applet during initialization.

com.commit_consulting.oraformsfaces._LOADING_IMAGE

Normally OraFormsFaces will cover the applet with an (animated) image while the applet is initializing. Once the applet is fully initialized the image is removed and the applet shows. This is a better user experience than showing the applet while it is being constructed, moved, resized, and while it is closing and starting Forms.

Add this parameter to the `web.xml` file with its value set to `false` to disable the covering of the applets with an (animated) image during startup.

com.commit_consulting.oraformsfaces._REPAINT

Normally OraFormsFaces will force a redraw of the applet area after it finishes initialization. This resolves issues in some Firefox versions intermittently showing a black area instead of the applet or Internet Explorer showing a displaced version of the applet.

Add this parameter to the `web.xml` file with its value set to `false` to disable the redrawing of the applet at startup.

com.commit_consulting.oraformsfaces._REPAINT_DELAY

Normally OraFormsFaces will force a redraw of the applet area after it finishes initialization. This resolves issues in some Firefox versions intermittently showing a black area instead of the applet or Internet Explorer showing a displaced version of the applet.

The redraw is forced by shifting the applet 1 pixel and subsequently shifting it back. If these changes are performed too quickly the browser can detect the net result is nothing and will not perform a redraw. Add this parameter to the `web.xml` file to set the number of milliseconds between shifting the applet and shifting it back.

com.commit_consulting.oraformsfaces._VERSION_CHECK

Normally OraFormsFaces checks if all components are from the same OraFormsFaces version. This includes the JAR file at the Forms server, the JAR file at the JSF server and the PLL file at the Forms server.

Add this parameter to the `web.xml` file with its value set to `false` to disable this version checking and allow OraFormsFaces to run with an incompatible set of files.

17.3.2. Forms Configuration Variables

Section ***14.1.2 Oracle Forms Servlet Base Template Files*** described the base template files that ship with OraFormsFaces. These files contain numerous calls to `offBaseAppletParameters.push` to add parameters to the applet, for example:

```
offBaseAppletParameters.push({name:"colorScheme",value:htmlDecode("%colorScheme%")});
offBaseAppletParameters.push({name:"serverApp",  value:htmlDecode("%serverApp%")});
offBaseAppletParameters.push({name:"logo",       value: htmlDecode("%logo%")});
offBaseAppletParameters.push({name:"imageBase",  value:htmlDecode("%imageBase%")});
```

You can add additional calls to `offBaseAppletParameters.push` to add additional parameters that will disable parts of OraFormsFaces, for example:

```
offBaseAppletParameters.push({name:"XXX_oraformsfaces_applet_activated_event",
                              value:"false"});
```

XXX_oraformsfaces_applet_suspended_event

Normally OraFormsFaces will detect when an applet is removed from the page by navigating away or reloading. OraFormsFaces will send a `when-applet-suspended` event to the applet to let it know it is suspending into the legacy lifecycle cache. See ***13 Forms Java Applet Instance Reuse*** for more information.

Adding this parameter to the applet with is value set to `false` will prevent OraFormsFaces from sending the `when-applet-suspended` event. If you feel the event itself is not the issue but only the timing at which the event is triggered, you can add a HTML or JSF button to the page and use the JavaScript API to send the event manually. Just send an event "`when-applet-suspended`" with an empty payload to the applet as described in `7` ***Invoking PL/SQL Events from JavaScript***.

XXX_oraformsfaces_applet_activated_event

Normally OraFormsFaces will detect when an applet is resumed from the legacy lifecycle cache and is being reused on a HTML page. OraFormsFaces will send a `when-applet-activated` event to the applet to let it know it is resuming from the legacy lifecycle cache. See ***13 Forms Java Applet Instance Reuse*** for more information.

Adding this parameter to the applet with is value set to `false` will prevent OraFormsFaces from sending the `when-applet-activated` event. If you feel the event itself is not the issue but only the timing at which the event is triggered, you can add a HTML or JSF button to the page and use the JavaScript API to send the event manually. Just send an event "`when-applet-activated`" with an empty payload to the applet as described in `7` ***Invoking PL/SQL Events from JavaScript***.

XXX_oraformsfaces_startrunform_from_edt

Normally OraFormsFaces will check if the applet is initializing from the Event Dispatcher Thread. To prevent locking issues all manipulation of UI elements in an applet have to be performed from this thread. Some versions of Oracle Forms do not initialize the applet itself from this thread. To prevent deadlocks, OraFormsFaces detects these situations and ensure initialization is performed from the EDT.

Adding this parameter to the applet with its value set to `false` will prevent OraFormsFaces from detecting this and will keep the original Oracle Forms code initializing in the main thread.

XXX_oraformsfaces_communicate_window_size

Normally the OraFormsFaces applet will detect each time a Forms window is being opened or activated. It will then call an OraFormsFaces JavaScript function to let the JavaScript code know what the size of the Forms window is. This can be used in auto-sizing and auto-clipping as described in ***12 Visual Integration***.

Adding this parameter to the applet with its value set to `false` will prevent the OraFormsFaces applet from making this JavaScript call and will therefore disable any auto-sizing.

XXX_oraformsfaces_sync_container_size

Some older Sun JRE and browser versions will sometimes miscalculate the size of the container object in the browser containing the Java applet. If this happens the applet will shift upwards or downwards unexpectedly during initialization. As a workaround OraFormsFaces checks the size of this container during initialization and resets its size if it is detected to be wrong.

Adding this parameter to the applet with its value set to `false` will prevent OraFormsFaces from detecting a wrongly sized container. This might cause the applet to be misplaced in some older Java or browser versions.

XXX_oraformsfaces_initial_forms_to_js_call

Normally the OraFormsFaces makes a call to JavaScript during initialization to let the OraFormsFaces JavaScript objects know the applet has initialized and is capable of sending JavaScript events.

Adding this parameter to the applet with its value set to `false` will prevent the OraFormsFaces applet from sending this initial JavaScript message during initialization.