Enterprise Java and ADF For Forms Developers: Taking it to the next level

An Oracle White Paper
July 2006

ORACLE

Enterprise Java and ADF For Forms Developers: Taking it to the next level

Enterprise Java and ADF For Forms Developers: Taking it to the next level

INTRODUCTION

For many in the I.T world the opportunities opened up by the enterprise Java platform: SOA enablement (Service Oriented Architecture), standards based, wireless applications, rich Internet application, etc.; are see as exciting. However, the platform and development experience are perceived by some, to say the least, to be challenging.

Oracle has been meeting this challenge head-on through the development of JDeveloper and the Oracle Application Development Framework (ADF). Many of the features in JDeveloper and ADF provide design-time and runtime productivity services not dissimilar to those found when developing using 4GL tools such as Oracle Forms; and as such, when it comes to building Java EE applications with JDeveloper and ADF, many of the development techniques and application requirements are met through familiar development gestures.

This paper goes beyond the basic CRUD (Create, Remove, Update and Delete) set of operations already provided for you, and shows how you can supplement and extend the basic behavior by adding your own customized operations to perform enhanced record creation and deletes, validation, conditional page flow and UI processing; exactly as you did with Oracle Forms.

FOCUSSING ON THE BUSINESS LOGIC

At the heart of any application, and specifically those interacting with a database, is the business logic that implements the specific features of your business atop a more generic CRUD (Create, Retrieve, Update & Delete) set of features. As you do with Oracle Forms, you allow the runtime framework to perform the common CRUD actions, and those actions you complement with validation and business rules code. In some cases you also choose to tailor the default transactional features of the runtime to meet your application needs.

ADF provides the same core features in the same way as the Forms runtime: a whole range of built-ins/APIs to manipulate your application objects, and

numerous events/triggers where you can complement or override the runtime functionality.

Introducing your framework

The architecture of ADF is outside the scope of this paper so additional reading material can be found at http://www.oracle.com/technology/products/adf/index.html.

However to briefly recap on the main elements referenced in this paper:

ADF Business Components

ADF Business Components provides a database focus to building business services in ADF. As a Forms developer you tend to build you Forms based on an existing database schema with SQL queries defining how data is manipulated from the database. ADF Business Components gives this same view of application development and does so using three main building blocks:

Entity Object

The entity object is, in nearly all cases, a one to one mapping to a database table. It implements the Object/Relational (O/R) mapping between a relational table and an object in the application. Each entity object typically has a number of attributes, each mapping to a column in the underlying database table.

View Object

The view object is a SQL query that selects a view of data from one or many database tables and can be thought of in a similar way to a Forms block or a database view, except within context of an application.

Application Module

The application module is a container of one or many view objects that define a user transaction: in the same way as a Forms module is usually related to a defined unit of work (e.g. maintain employees or check order status)

This splitting of entity objects, view objects and application modules allows the reuse and modularization of business services.

Code exposure, not generation

Before embarking on a discussion on how to extend the functionality of ADF Business Components its worth re-emphasizing that JDeveloper and ADF do not work simply via mass code generation. The act of going through the ADF Business Components dialogs does not result in the automatic generation of thousands of lines of Java code. What happens is that XML meta data is generated and this meta data is read at runtime by ADF to implement the desired behavior. In theory, you can develop an ADF application without actually seeing a single line of Java code. Figure 1 shows an example of the meta data created for a *Customers*

object. Information such as the attribute name, column name to which it maps, type and validation information (such as *MaritalStatus* must be **single** or **married**) are held in this meta data format and read by the framework to implement the desired behavior.

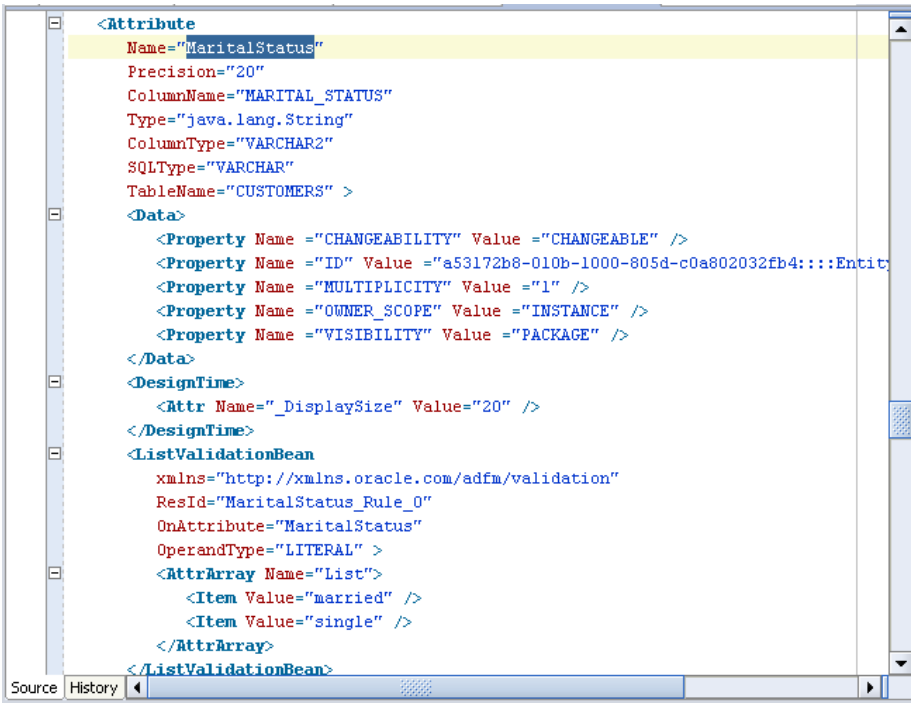

Figure 1 – XML Meta data to implement a Customers Entity Object

There are many benefits associated with this approach, for example:

- A change to metadata does not require re-compilation
- Less code means less testing and fewer bugs
- Runtime flexibility
- Iterative exposure to code as developers become more familiar with the framework

So, when you want to write Java code (as you wrote PL/SQL to extend your Oracle Forms) this is where JDeveloper and ADF give you a helping hand. By simply setting a check box (Figure 2), you can indicate to JDeveloper that you want to expose the Java methods that the framework will use to, for example, create a record.

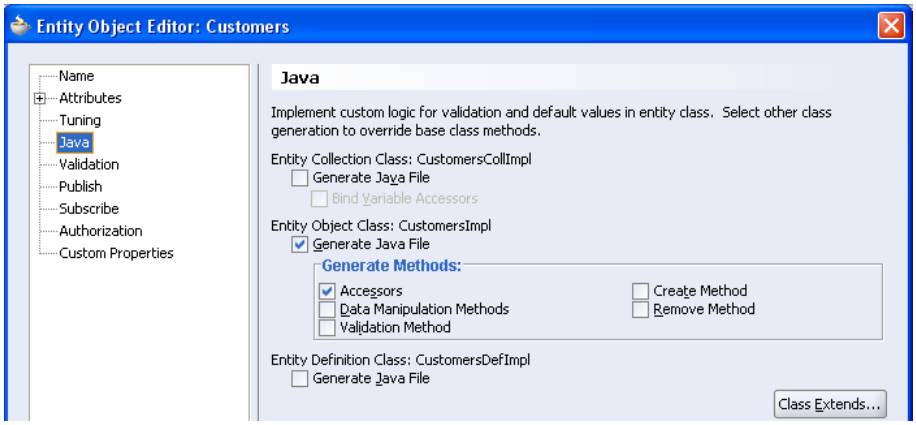

Figure 2 – Indicating what Java methods to expose on an Entity Object

To be strictly correct, JDeveloper is not actually exposing the code the framework calls, it is providing you with a custom class that extends the basic functionality, and in that extended class you can override the default functionality while still calling the base class to "do the right thing"

.

Once this code is exposed you can simply extend and augment the code to fit your needs.

Extending the default behavior of ADF Business Components

So, now that you know you can expose the code behind the framework, lets look at some of the most common business actions for which you would want to extend the framework.

Item level validation – When-Validate-Item

The most common feature in any data entry application will be the validation of the data entered. Oracle Forms provides a *when-validate-item* trigger where the default validation of a Forms field can be supplemented with PL/SQL code. While ADF Business Components provides a much richer declarative validation framework even than Forms, it is expected that at some point you will want to validate a data field based on some complex scenario that requires code.

Business case

Consider that for your application you need to validate that the email address for a customer is valid. As an absolute minimum the email address should contain the "@" symbol.

Implementation

ADF Business Components provides declarative validation on an entity object and one facet of this validation is the ability to call a method validator for a specific attribute (Figure 3).

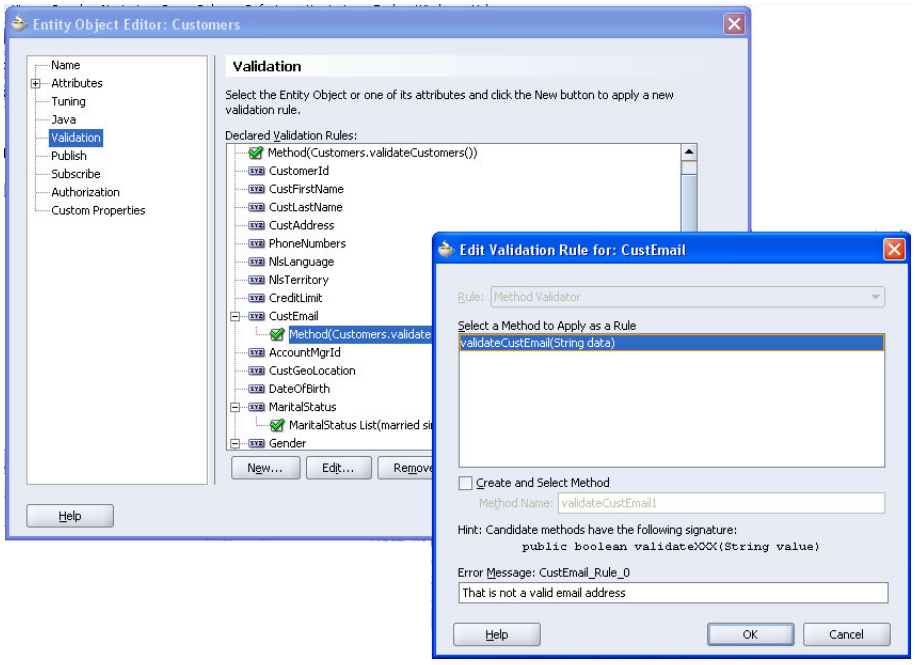

Figure 3 – Adding a method validator to an entity object

Again, one of the advantages of ADF Business Components is that validation code, such as demonstrated here, is easily reused throughout the application.

In the entity object editor, select the *CustEmail* attribute and add a method validator rule. JDeveloper will default the name of the method for you as *validateCustEmail*. To this method you can add your custom validation code (in this case, checking to see if the email address contains the "@" symbol).

.

```
/**Validation method for CustEmail
 */
public boolean validateCustEmail(String data) {
    // Add custom code to validate CustEmail
    return data.indexOf("@") != -1;
}
```

Note that for the business scenarios used in this paper, you may choose to implement them differently in your application. The scenarios serve only as a means to demonstrate the underlying techniques of using the framework.

Record level validation – When-Validate-Record

You may also wish to compare different fields of data entered against each other. Hence, while a particular field value may be within valid limits, it needs to be validated against the value of another field. In Forms the *when-validate-record* trigger performs this action.

Business case

When entering the details for a customer, any customer with a credit limit greater than 4000 must be assigned to a specific manager.

Implementation

The ADF Business Components declarative validation on an entity object also provides for a method validator to be called on the entity itself. In the entity object editor add a method validator. JDeveloper defaults the name of the method to *validateCustomers*. Here you can add code to check that if a customer's credit limit is greater than 4000 then the account manager must be account manager 145.

```
public boolean validateCustomers() {
    // Add custom code to validate Customers
    if ((getCreditLimit().intValue() > 4000) &&
        (getAccountMgrId().intValue() != 145)) {
        return false;
    }
    return true;
}
```

Transactional triggers

When it comes to transactions, the framework will automatically take care of performing the create, delete, insert and update operations for you. However, it may be that you want to supplement this default behavior with your own application specific code.

Business case

For your application you need to keep an audit trail of every new customer that was added into the system, and who inserted that customer. This is the equivalent of the Forms *post-insert* trigger, which will fire on the successful insertion of a record.

Implementation

On the *Customers* entity object bring up the entity object editor and for the Java node, set that Data Manipulation Methods are exposed (Figure 4).

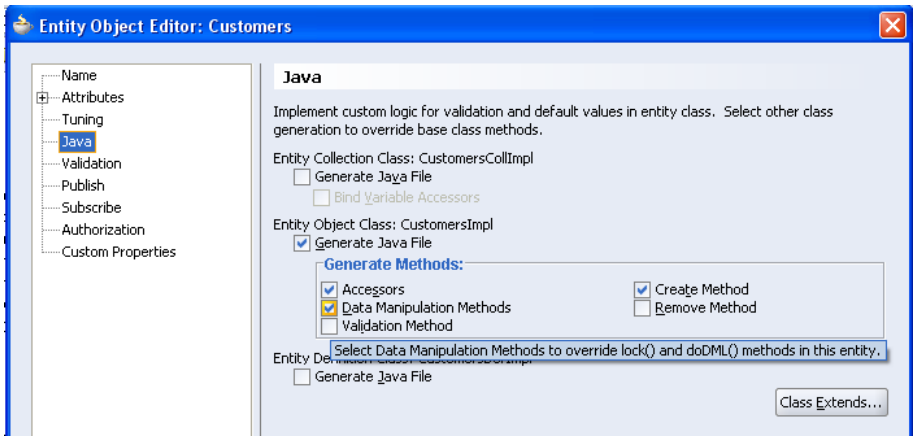

Figure 4 – Generating Data Manipulation Methods on an entity object

This will expose a method *doDML*, that will perform the DML operation. Thus, any code added before the DML operation mimics a Forms *pre*-transaction trigger, and any code added after the DML operation mimics a *post*-transaction operation.

The phrase "do the right thing" is one that should be familiar to Forms developers. It is used when writing a trigger (usually an on- trigger) where you call Forms to "do the behavior you were going to do before I overrode you".

```
    protected void doDML(int operation, TransactionEvent e) {

        //pre-transaction code added here

        super.doDML(operation, e);

        //post-transaction code added here

        if (operation == DML_INSERT) {

            System.out.println("That record was inserted by " +
e.getDBTransaction().getConnectionMetadata().getUserName());

        }

  }
```

When-Create-Record

At the point you create a completely new record in the application, you may want to do some manipulation of the record, for example, the default values for a specific field. On the ADF Business Components entity object you can expose a method that is called on creation of a new record (Figure 5).

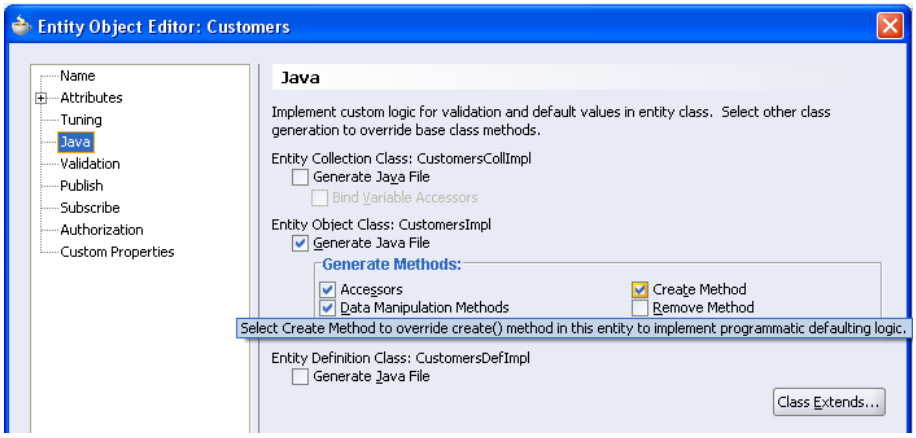

Figure 5 – Generating Create Method on an Entity Object

Business case

On the creation of a new customer, you want to default the value in the *CustomerId* field from a database sequence.

Implementation

Once you have generated the create method, add the following code to read a sequence from the database, set the *customerId* and then call the create method in the super class to "do the right thing".

```
    protected void create(AttributeList attributeList) {
        ApplicationModule am =
getDBTransaction().getRootApplicationModule();
        DBSequence ds = new DBSequence("CUSTOMER_SEQUENCE",
am);
        setCustomerId(ds.getSequenceNumber());
        super.create(attributeList);
    }
```

When-Remove-Record

Similarly, to when you added code to supplement the creation of a new record, you may want to do some processing when the use attempts to remove a record (when it is removed from the application as opposed to the point when the operation is committed to the database). This is the equivalent of the *when-remove-record* trigger in Forms.

Business case

You may want to prevent a user removing a particular data record. For example, if a customer order was not created on-line then you may want to prevent its removal from the on-line system.

Implementation

In the *Orders* entity object you can expose the code called when a record is removed in a similar manner to before (Figure 6).

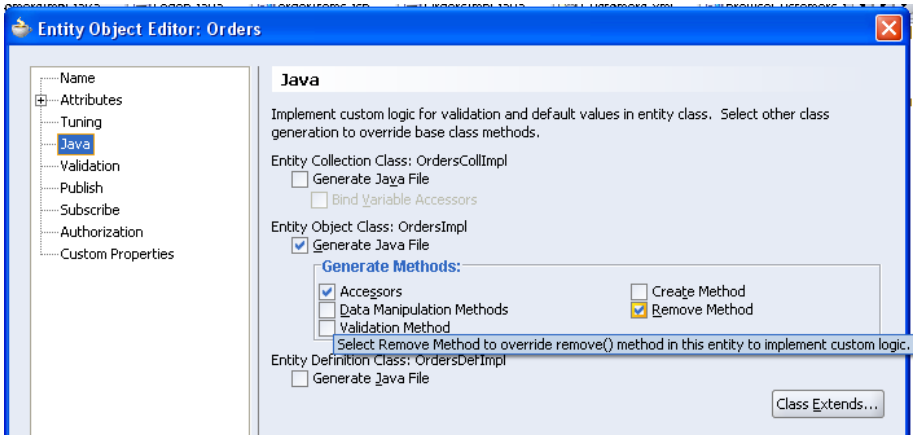

Figure 6 – Generating Remove Method on an entity object

Having exposed a remove method on your entity object, you can add the following code to check that if the order was placed directly then the record cannot be removed:

```
    /**Add entity remove logic in this method.
     */
    public void remove() {
    if (getOrderMode().equalsIgnoreCase("direct")){
        throw new JboException("Cannot remove a record that was
not ordered on line");
    }
        super.remove();
    }
```

SEARCH FORMS AND QUERY BY EXAMPLE

As with any programming model, there are numerous different ways of achieving similar functionality in an application. For searching and querying data in Oracle Forms you could set the *where* clause on a block then *execute_query* or even use the sophisticated "query by example" functionality in Forms to filter the data in a block.

ADF and ADF Business Component provide similarly sophisticated features for the application developer.

Search form

Oracle Forms provided you with the ability to put your form into "find mode" then execute a query on any data you supplied. ADF Business Components provides the same functionality.

Business case

You want to provide some "free form" query behavior for your application users such that the euser can query on an almost unlimited set of user criteria e.g. all single, male employees with credit limit >3040.

Implementation

When you drag data from the component palette onto a page, one of the options presented is to create an ADF Search Form (Figure 7). This will provide functionality to enter "free form" query criteria such as ">3050" "Smi%" etc.

You can of course customize this search form by removing fields or changing the default input text fields to other UI controls such as check boxes or drop down lists.

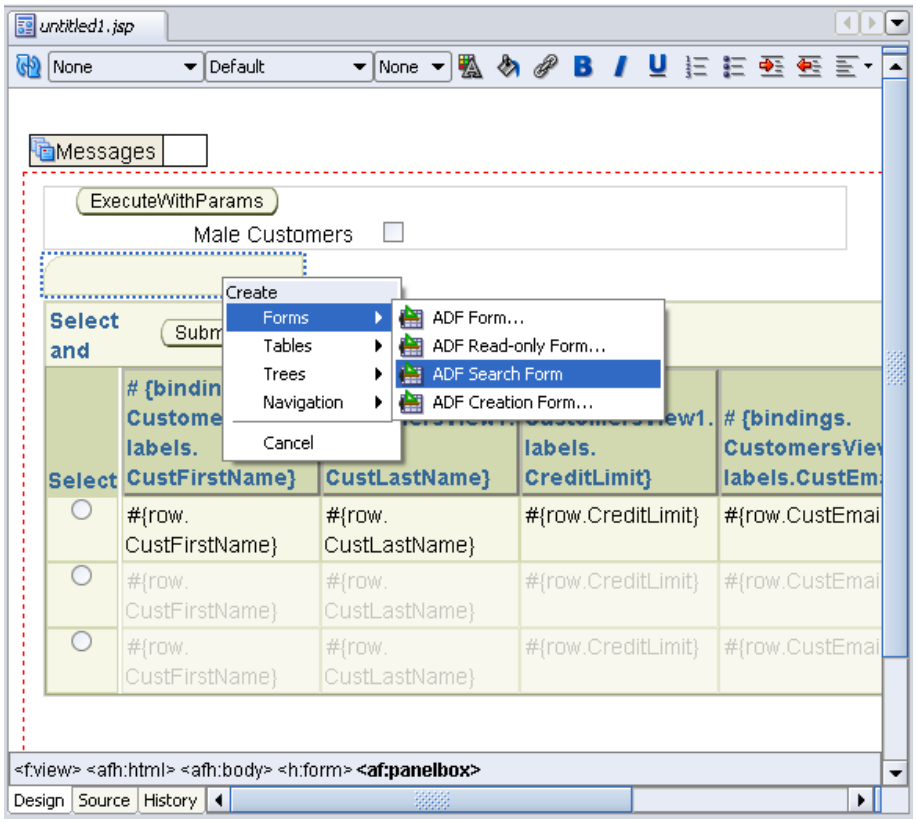

Figure 7 – Creating a search form

Execute with parameters

Within a particular view of data you may specify bind parameters to be resolved at runtime. ADF presents these runtime bind parameters as fields in the data model that can be dragged onto any webpage as a visual control (such as a text field). So in effect you are allowing the query to be re-executed with the bind parameter values coming from a field on your page.

Business case

In order that you can do some focused marketing, you want to quickly filter the list of customers to show only male (or female) customers.

Implementation

On the customers view object add a bind parameter to the where clause of the query. This bind parameter appears as *executeWithParams* in the data model. Drag this node from the component palette onto your page and display the bind parameter as an input field or as a check box. The functionality that will actually re-perform the query appears as a button (Figure 8).

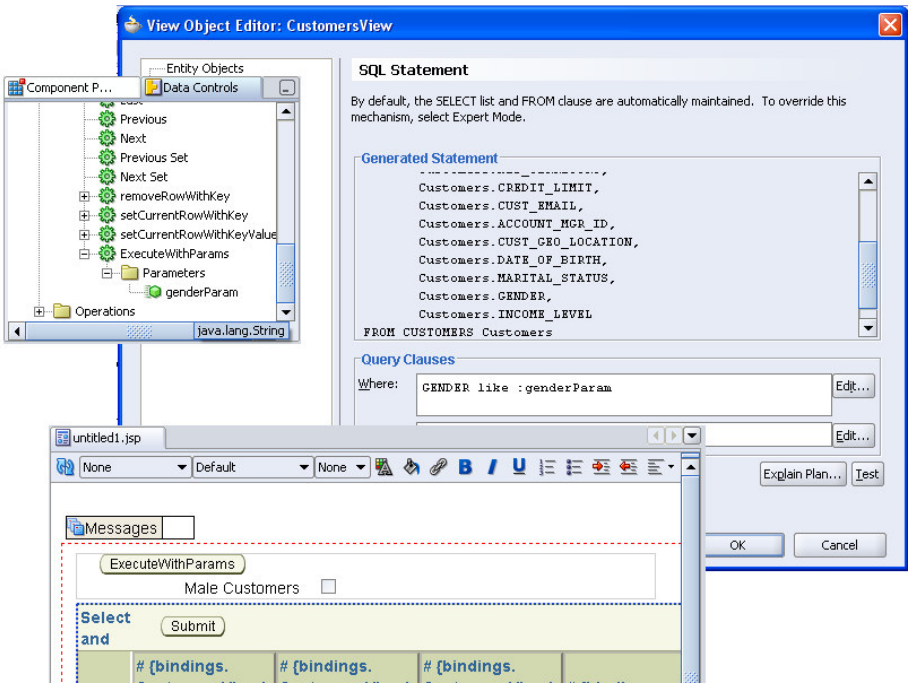

Figure 8 – Setting a bind parameter for executeWithParams

Execute query

As stated before, there are numerous ways of achieving this desired functionality. For a fuller explanation see section 5.8 Filtering Results Using Query By Example View Criteria in the Developers Guide for Forms/4GL Developers at http://download.oracle.com/docs/html/B25947_01/toc.htm

Finally for search functionally, you can simply code up the equivalent of the Oracle Forms *set_block_property* to define the *where* clause then *execute_query*. ADF provides various APIs for manipulating the objects of your applications. At the most basic level, you can add business level filtering to your application module and call those methods wherever required in your application.

Business case

Part of your business application is to focus on "Gold" customer (i.e. ones with a particular profile e.g. high credit limit). You therefore want to add the criteria to view your Gold customers to your business model.

Implementation

The application module in ADF Business Components is the container for the business transaction. For this sample, you will add two business rules to the application: one to filter your view of customers to show only those with a high credit limit, and a second to show all customers.

This is achieved by adding two methods to the application module and exposing those methods to the client.

```
public void goldCustomers() {
    ViewObject vo = findViewObject("CustomersView1");
    vo.setWhereClause("CREDIT_LIMIT > 4000");
    vo.executeQuery();
}

public void allCustomers() {
    ViewObject vo = findViewObject("CustomersView1");
    vo.setWhereClause(null);
    vo.executeQuery();
}
```

There methods can either be called directly or, once exposed to the client, dragged from the component palette onto the page as buttons.

FOCUSSING ON THE USER INTERFACE

Unlike Oracle Forms where there is no separation of the business logic and the UI, ADF naturally separates the business logic from the UI. We will now look at customizing some of the common UI behavior.

Auto submit

When using web technologies like JSP or JSF to deliver an HTML UI, the underlying technology means that communication from the client to the application only happens when the whole page is posted. This may be viewed as both an advantage and a disadvantage.

In terms of performance, only communicating with the application when the page is posted means that the application is much less "chatty". Excessive client/application communication is the source of one criticism of Oracle Forms when web deployed on a high latency network.

The emerging Ajax technology is moving to provide rich HTML pages by communicating in an asynchronous manner to retrieve information to update the page. Oracle's ADF Faces components are taking advantage of this technology as it becomes available, combining the ease of use of the JSF programming model with the power and interactivity of Ajax technologies.

However, this client/application traffic does bring a certain "richness" to the application generally not associated with HTML applications.

ADF, and specifically, ADF Faces, allows the application developer to indicate that a specific UI item can trigger the posting of a page back to the application. This allows a balance to be struck in adding richer UI interaction while still keeping the traffic to a minimum. This behavior is presented as the *AutoSubmit* and *PartialTrigger* attributes on ADF Faces components.

Business case

One of the pages of your application shows the line item for an order. The information for a line order item is essentially the *productId* and a read only *description* and *productName* (which is actually a look up to another table using the *productId*). The problem is that if the user changes the *productId*, the *productName* and *description* associated with the *productId* are not changed until the page is submitted. Hence, we want to submit the page to update the *productName* and *description* when the *productId* is changed.

Implementation

Select the item on which you want a change to submit the page (in this case *productId*). Set *AutoSubmit* to true and set an *id* for the field. This will mean that when *productId* is changed the whole page will be submitted.

The next step is to register the fields you want to change in response to the posting of the page. This is achieved by setting the *PartialTriggers* attribute for the fields (*description* and *productName*) that should reflect the change in the *productId*. (Figure 9)

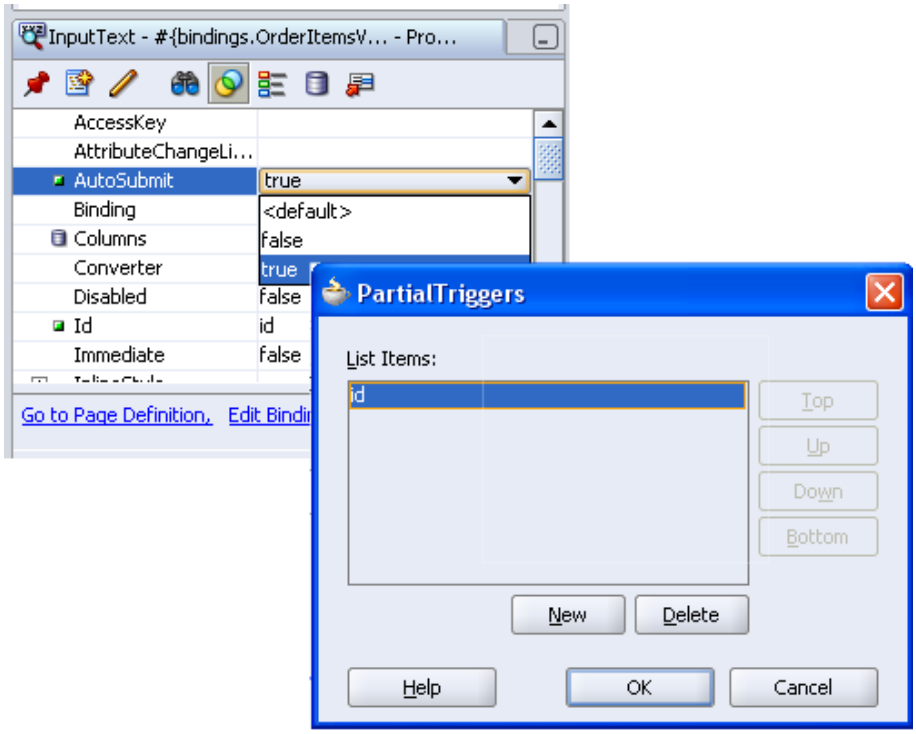

Figure 9 – Setting AutoSubmit and PartialTriggers

Now when the *productId* field is changed the *productName* and *description* immediately reflect the new value.

The stacked canvas

There are two features of ADF Faces that make the implementation of the Forms equivalent of a stacked canvas simple:

- The JSF Expression Language (EL)
- ADF Faces Switcher component

EL allows a simple scripting language to be associated with each component – and the value of an attribute in a component can be resolved from EL. Which makes EL very powerful not only for binding the UI component to the underlying model, but also resolving the values of e.g. background-color depending on a runtime expression defines in EL.

The switcher component

The ADF Faces switcher component works by having a number of facets and each facet can be dynamically managed by the switcher component. Thus, each facet could represent a different "stacked canvas" and depending on an expression a different facet is displayed.

Business case

If, as a result of entering some search criteria, the result returned shows no rows, you want to display a message indicating no rows rather than displaying an empty table.

Implementation

To achieve this behavior drop a switcher component on to the page. Add two facets to the switcher (right click on the switcher – Insert inside af:switcher -> JSF Core -> Facet). Name them "**CustTable**" and "**EmptyMessage**". Under the **CustTable** facet drag and drop the existing customer table. Under the **EmptyMessage** facet add an output field with a "no records returned" message. (Figure 10)

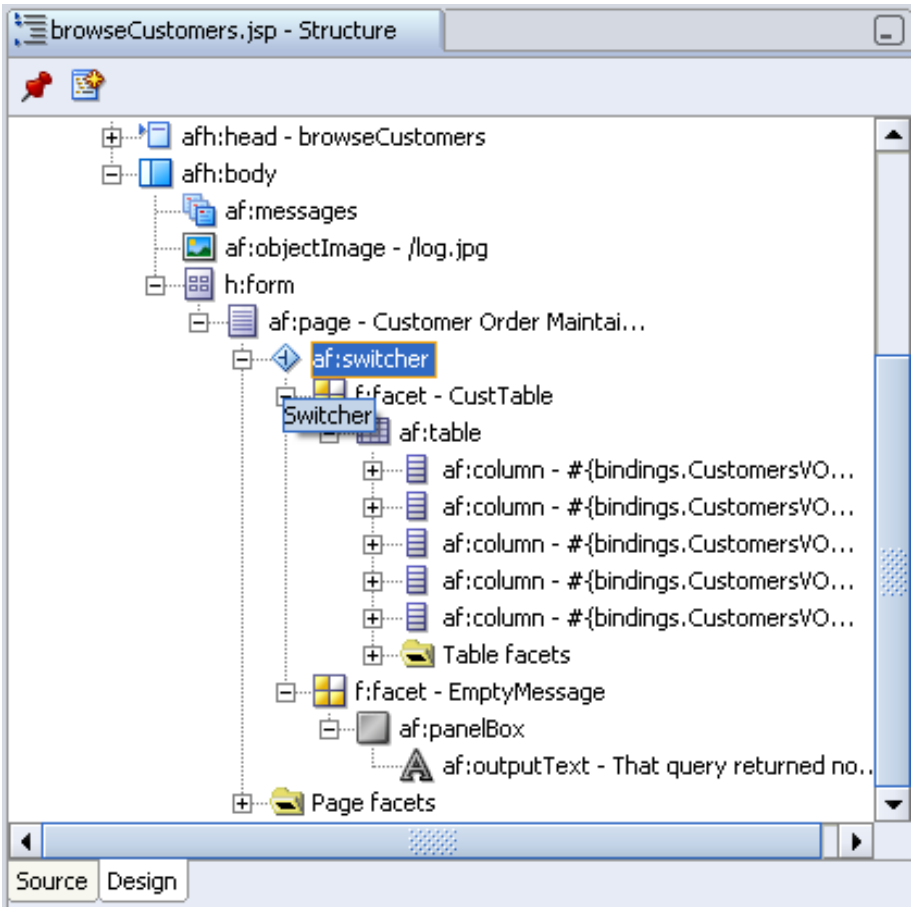

Figure 10– Two facets of a switcher component

This defines two named facets ("stacked canvas"). You now have to add the code to evaluate which facet is rendered. This is achieved by setting the *FacetName* property to the name of the facet to be displayed. In this case an EL to check the iterator's *estimatedRowCount* and then to return **CustTable** or **EmptyMessage**.

```
#{bindings.CustomersVO1Iterator.estimatedRowCount > 0 ?
"CustTable" : "EmptyMessage"}
```

Conditional Page Flow

With Oracle Forms, once you have developed your form it will probably be hooked into a number of other forms to create the flow to your application. So, the navigation of one form to another using Call_Form or Open_Form via a *when-button-pressed* trigger is a common task you will code into your application.

Coincidentally, the same functionally of conditional page flow is easy to achieve with JDeveloper and ADF.

Business case

Consider the situation that depending on some determinable factor; you want to conditionally navigate to one page or another. For example, if you fail to correctly log into your application you should be navigated back to the logon page, otherwise you will be taken to the first page of the application.

In actual fact, the logging onto an application would be handled in a completely different way to the business case proposed here but this is simply a way of demonstrating the underlying functionality of conditional page flow.

Implementation

With ADF, the act of pressing a button returns a string indicating an *action*. This *action* is defined on the JSF navigation diagram. So, as figure 10 shows, the logon screen results in two possible page flow actions: **logon** or **fail**. (Figure 11)

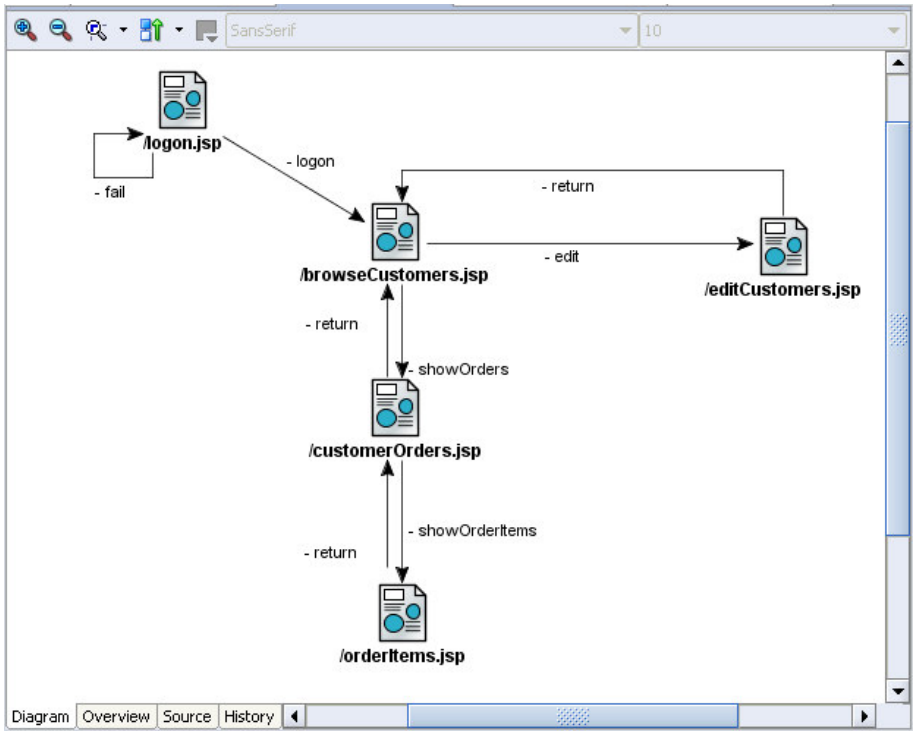

Figure 11 – Conditional Page Flow

You can now create the logon page. When you do this ensure you expose the UI components in a backing bean (this is one of the options in the wizard), drop two input fields and a button on to the page and when you double click on the button JDeveloper will automatically create a method for you (the equivalent of your *when-button-pressed* trigger). In this method add the following code:

```
public String commandButton1_action() {

// Add event code here...
```

```
        if (getInputText1().getValue().equals("scott") &&
            (getInputText2().getValue().equals("tiger"))) {
            return "logon";
        }
        return "fail";
    }
```

This simple case will navigate you to the first screen if you correctly enter "scott" and "tiger" into the two fields.

Application Look and Feel

Any business will undoubtedly want to "brand" their application with a specific look and feel. In the Oracle Forms world the use of visual attributes was key to achieving a consistent and easily maintainable application façade.

ADF Faces provides the same feature through the use of *skins*. ADF Faces provides a number of pre-configured *skins*; and also allows you to create your own custom *skins*.

Business case

Your company has a corporate standard set of fonts and color scheme for its websites and collateral. You therefore need to implement the corporate look and feel on your web application (Figure 12).

Implementation

Skins are implemented by the use of cascading style sheets (CSS). There are essentially three steps for creating a customer skin.

- Create a CSS to define the style for the various components
- Register your new skin in the `adf-faces-skins.xml` file
- Configure the application to use the new skin through the `adf-faces-config.xml` file

For full details on creating custom skins, refer to the JDeveloper online help.

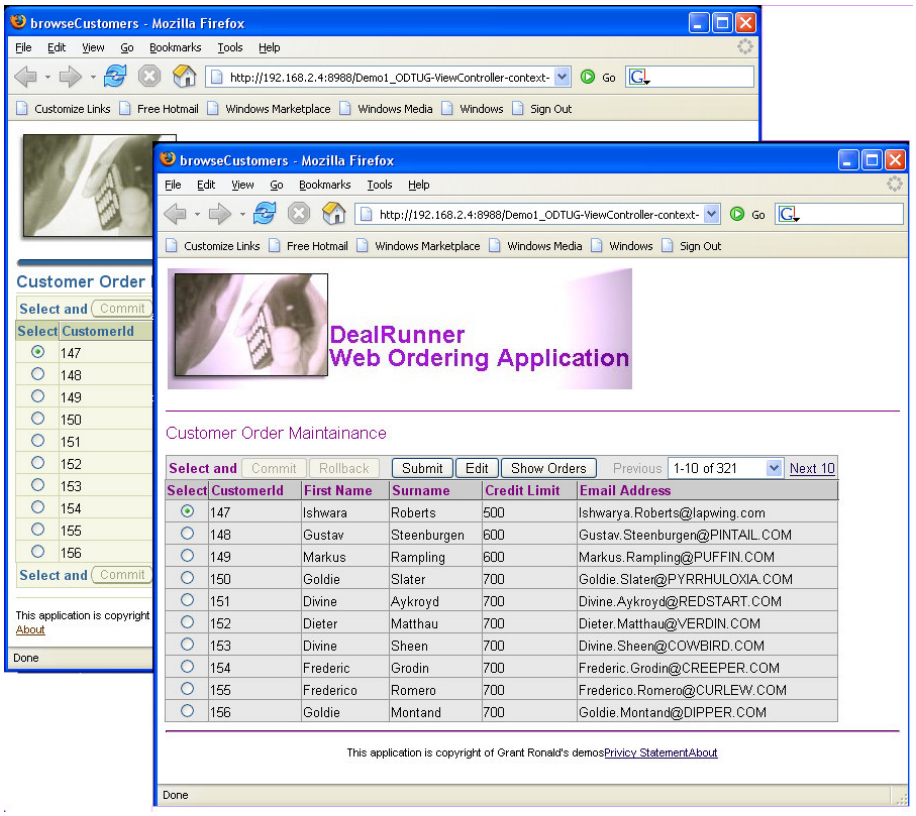

Figure 12 – Applying a custom skin to an application

CONCLUSION

JDeveloper and Oracle Application Development Framework (ADF) give you a whole range of design-time and runtime helper features and services for building your enterprise Java applications. The productivity features go way beyond the simple CRUD operations and give you the power and flexibility to customize and augment the default functionality in a similar fashion to Oracle Forms. Using similar development paradigms you can extend the validation, business services, transactional processing and UI behavior; allowing you to recognized the full power of the platform.

Enterprise Java and ADF for Forms Developers: Taking it to the next level
July 2006
Author: Grant Ronald
Contributing Authors:

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

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