A Case Study of Redeveloping a Forms Application in Oracle ADF

Grant Ronald
Oracle Corporation UK Ltd

Oracle Forms migrate ADF JDeveloper

Executive Overview

For many Oracle customers, Oracle Forms is the cornerstone of their business applications. However, as business and technologies grow and evolve, opportunites for change are likely to occur. There are many different ways these changes can be embraced: possibly through modernization within the existing technology stack, or through the adoption of new technologies.

Before embarking on any considerable change, we recommend refering to the following documents:

- For the statement of direction Oracle tools go to http://www.oracle.com/technetwork/issue-archive/2010/toolssod-3-129969.pdf
- For Oracle Forms modernization options go to http://www.oracle.com/technetwork/developer-tools/forms/forms-modernization-092149.html
- On the challenges of an Oracle Forms migration, go to http://www.oracle.com/technetwork/developer-tools/forms/documentation/formsmigration-133693.pdf

While the above links explain how you can continue with Oracle Forms as an IT investment, and the options you have for modernization, this paper focuses on the scenario of redeveloping a typical Oracle Forms application using Oracle JDeveloper and Oracle ADF. The paper documents many of the challenges, development techniques, decision process, guidelines and ultimately, the technicalities of redevelopment. The implementation documented herein shouldn't be interpreted as the only way to redevelop an application that was originally written in Oracle Forms. Still, the documented experiences and recommendations apply to the broad range of decision points in applications and can serve as a blueprint for your own custom redevelopment initiatives.

Introduction

The goal of this project was to show the experiences, design decisions, and possibilites available when redeveloping an Oracle Forms application using Oracle ADF and also to serve as a reference, or blueprint, for others embarking on a similar redevelopment effort. In order to ensure a "real world" experience, we wanted to avoid a simple Emp/Dept application, or one which was written with the express purpose of demonstrating ease-of-redevelopment. However, we also wanted to balance a real-world case with something which was succint enough to be easily conceptualised.

We decided that the "Summit" sporting goods application, developed many years ago as part of an Oracle Forms training course, served as a good example of such an application. As well as providing many common and typical Forms features and scenarios, it provided challenges where we had to make architectural decisions on how to best evolve a particular function, rather than simply implementing a mirror copy of the original implementaion.

In Forms, the term "form" for the .fmb file is used. This can be loosely translated to "view" or "page", meaning the .jspx or .jsff page that defines the artifact that is displayed in the browser. Similarly, the term "field" or "item" is used in Forms, because there is tight coupling between a UI element and it's corresponding data model value (block.item). In Java EE applications, there is cleaner separation between the data model and view layers and therefore the data model object is referred to as an "attribute" and the view component is referred to by the UI element that represents the attribute value on the client such as "input text".

The Forms Summit application also exhibited limitations in the original design as well as the business functions it was developed to demonstrate. This gave us further opportunities to change the application and demonstrate where redevelopment could also be a catalyst for improvement. This is something we saw as a common scenario for customers as well: where the redevelopment offers scope for improvement to correct any issues in the original design, but also to show how the dollars spent on redeveloping could improve not only the application, but the business as well.

In this paper we have documented the experience and design process, as well as some suggested best practices for the redevelopment of an Oracle Forms application using Oracle ADF. While there are infinite possibilites on how an application can be built based on many factors that might be particular to your situation, we feel our documented experiences represent a guide which addresses the broad challenges of this type of redevelopment, and demonstrates a path which will help in many redvelopment cases.

Initial Project Considerations

Before starting the project we had to define the parameters of what we were looking to achieve. While essential for a case study like this, this is also a typical requirement when embarking on a proof of concept, or even as a full redevelopment.

Business Decisions

In this case study we had to wear many different hats, but initially we had to think as a business manager and to decide what was right for the business. Here are some of the business decisions we made

- Similar application function – We assumed that the core use cases of the application, an online storefront for a fictional sporting good supplier, would remain the same.
- While implementing these core use cases, we should not be constrained by the mantra of "we always did it like this in Forms". This was particularly important when considering the UI design
- Redevelopment should be seen as an opportunity for change and to embrace new business scenarios. For example, we decided that rather than being a back-office only application, we would design the application to address both back-office users and self-service users. Again, this mirrors a common requirement we see with Oracle Forms and Oracle ADF customers.
- Some features in the Forms application weren't fully complete or production quality; primarily because the application was originally developed as a demo. We took the opportunity to fix, or enhance, some of these limitations

Technology Decisions

Technology decisions can be based on a myriad of factors, and it would be impossible for us to document a decision tree to cover all possibilities. Instead we document the high-level decisions we made and our reasoning.

- Oracle JDeveloper and Oracle ADF – The choice of development tool is a hotly debated topic, often influence by partisan personal preferences. Given Oracle's own use of JDeveloper and Oracle ADF for Fusion Applications, and the fact that many of these original applications were built using Oracle products, it was a natural choice to use this set of technologies.
- Oracle Fusion technology stripe – Given the previous point, we chose the same Oracle ADF technology stripe as Fusion Applications, namely, ADF Business Components, ADF Model, ADF Controller, ADF Faces Rich Client. Additionally, our final aim was to closely follow the Fusion Application design and UI interaction patterns although these would be phased into the application to avoid too many dramatic changes to the UI functionality.
- Easiest mapping of skills and concepts – It is true to say that since this redevelopment project was being undertake by Oracle ADF product managers there is an unavoidable slant in technology choice. However, from Oracle's own experience of Fusion Applications, and those of our customers and many industry commentators, understanding the complexities of the myriad of supporting technologies, APIs and changing flavors of the "framework of the day" is a considerable challenge. If you couple the fact that many Oracle customer with strong PL/SQL and Forms background are rarely hardcore Java experts, then the challenge just gets even more difficult. That is why we felt that that an environment and unified productivity framework such as Oracle ADF is the only real choice that provided any sort of realistic mapping of skills and concepts.

- Iterative development cycle – we decided to deliver the application over multiple iterations, each one building on the previous. This means the earlier iterations would focus on providing like-for-like business functionality while those later cycles would provide additional business functions or use more advanced technology features. While designing for the future, our initial development efforts were not focused on leveraging other products outside the Oracle ADF core (e.g. WebCenter, BPM, SOA, etc. etc.)

Understanding the Application Functionality

One of the biggest challenges of redeveloping an Oracle Forms application is firstly to understanding what that application does. Given a typical Forms application may have evolved and been upgraded over 5, 10, 15 or ever 20 years, it means that there are often very few people inside the business who understand fully what the application is capable of delivering. You not only have to understand the business function it delivers, but also the finer grained details of the application: such as how users search for data or how navigation is performed throughout the application. Understanding this is further complicated since those application functions might be spread across triggers, libraries or procedures in the database; or may even be a codeless feature of the Forms runtime.

Strategies for understanding application functionality

The scope of this redevelopment project was small enough that we were able to map the functionally between the Forms and SummitADF applications by both stepping through the Forms triggers to check the code had been implemented, and also by comparing the final applications side-by-side. Of course, this becomes a much more significant challenge when faced with a typical Forms application consisting of hundreds of Forms modules and libraries.

To help gain an insight into the functionality of an enterprise-level Forms application we would suggest evaluating some of the tools on the market that can introspect a Forms application and produce reports on the implemented functionality.

As noted, because of the scope of this application we didn't require any formal method to document the exisiting Forms functioality, however, Oracle partner, PITSS, generated a number of sample reports from their PITSS.CON tool which demonstrated to us that this kind of tooling could be a valuable aid to a redevelopment project.

Database Setup

When redeveloping a legacy application one key element will typically remain fixed: the data. For many businesses their data IS their business and so, as is characteristic with many Oracle Forms applications, the database and schema primarily drive the application. We decided that in this application the existing data and its structure must be preserved. We would therefore strictly limit changes to the database data and objects. The general exception to this rule was where we saw obvious limitations in the schema as a result of it being a demo schema rather that a production-level design.

Schema Overview

The Summit Schema is based on the concept of a sports goods supplier involving customers, orders and products. The goal of the project was to closely mimic the functionality of the existing application and so the existing database and data as used. We did, however, find limitations in the schema and took the opportunity to made appropriate changes

Changes to the existing schema

We used JDeveloper to create a diagram of the schema, allowing us to quickly visualize the tables and relationships. In the most part, our changes were to normalize the structures, although we also took the opportunity to make some corrections to the data and introduce sequences and triggers for the assignment of primary keys. We were able to do this to the offline database objects and then generate SQL scripts for the changes. The full SummitADF schema can be generated by running the build_summit_schema.sql script.

Project Standards and Methodology

Team Structure

The development team primarily comprised of two developers. Given the limited nature of the application and the team, we didn't assign specific development roles although you may choose to allocate different developers to different areas of the application, e.g. business service developer or UI developer.

Managing Source Code

We used a Subversion repository for SummitADF source control and checked in, checked out and merged files directly from JDeveloper. During the development we went through various stages of heavy refactoring and simultaneous development. In nearly all cases JDeveloper's integration with Subversion was seamless and we hit very few issues. Only in one or two incidents did we revert to Tortoise to address and issue which JDeveloper wasn't able to handle.

These were limitations that we regarded as bugs and so bugs were logged and we would expect these issues to be fixed in a later release of JDeveloper.

Application Structure

Application

For this initial release of SummitADF, we primarily developed the application within a single application workspace. Given the choice of technologies for the application, we chose a Fusion Web Application template. We did however, create a second application: SummitLib, in which we added

common helper code that might typically be shared across many different projects. For example, we used this application to define a library of database access features.

For any enterprise development effort you have to design for reuse and these database access helper classes served as a typical example of where you would refactor out code, and this is what we did, that might typically be shared across different applications.

Towards the end of the development cycle we also refactored the database scripts into a separate application workspace, Summit_Schema, so that they could be easily used by other applications/samples.

Project

Given a Fusion Web Application template was chosen for the application workspace, JDeveloper automatically created two projects: Model and ViewController. We used these default projects as a physical partitioning of business service and UI code.

Packages

Packages allow further partitioning of application artifacts. As a naming convention, we used "oracle.summit" as the package prefix. At this top level we had two packages "oracle.summit.base", which was used for base framework classes, and "oracle.summit.model" to partition the more general business service artifacts. Given the potential for a large number of ADF Business Component artifacts that would be created we wanted to further partition oracle.summit.model into packages that compartmentalized each of these artifact groups. Thus we created packages for entity objects, view objects, application modules and business components diagrams. The full package structure is as follows:

- oracle.summit.base
- oracle.summit.model
    - diagram
    - entities
        - assoc
    - views
        - links
        - readonly
    - services

Naming Standards

In any project it is important to ensure consistency in the framework objects being created. The rules we followed are explained below.

Entity Objects

Typically, entity objects have the same name as the underlying database object. However, the Summit database schema had the table names prefixed with "S_". The use of this prefix has no benefit and introduces an element of redundancy with respect to alphabetization of the objects in the Application Navigator. We therefore used the tables name as is, but without the "S_" prefix. Note also that the Summit schema is unusual in that the database tables are named in the singular.

We also decided to use the suffix "EO" on the entity objects. It could be argued that the fact they are entity objects is reflected in the package structure, but for the want of two characters it would help make it clear, whether through the Application Navigator or in code, that the object was an entity object.

Database Table	*Entity Object Name*
S_CUSTOMER	CustomerEO
S_ORD	OrdEO
S_ITEM	ItemEO

Associations

For this project we used the default names generated associations by the ADF Business Component wizard.

View Objects

For default view objects based on entity objects, the view object name would follow the naming of the primary entity object, but with an extension of "VO" instead of "EO". Given the Summit schema tables are named in the singular, and hence the entity objects and view objects as well, read only view objects were named in the singular as well to retain consistency.

For any read only view objects based on static data or a select statement, we used a name that identified the purposed of the view object with the suffix VO.

Read Only View Object	*Purpose*
YesNoVO	Static view of strings for Y/N
CreditRatingVO	Read only view of credit ratings
PaymentTypeVO	Read only view of payment types

View Links

For this project we used the default names generated for view links by the ADF Business Component wizard. For any manually created view links we chose a name that was descriptive of the relationship;

generally of the form "<*destinationCollection*><*conjunction*><*sourceData*>Link" where the conjunction could be something like "in" or "for". For example CustomersInCountryLink.

View Object Instances

Within an application module it is possible to have multiple instance of a view object, with each instance representing different data views. By default, JDeveloper would name these instances as per the view object but with a rolling number. Given that the names defined here are those that will be exposed through the Data Controls panel, we felt it was important that they should clearly reflect the data collection and so decided to explicitly name each. For example, SalesPeople is an instance of EmpVO where the instance applies a view criteria such that the data collection only represented those employees who are in a sales role.

We also named the view object instances in the plural. We felt this best reflected the fact that they are data collections, rather than single data values. This is also helpful for UI developers to identify collections in the Data Controls panel, as that may be their only exposure to the underlying data model.

Application Module

Each application module represents a use case and so is named, using camel case, to reflect the use case it implements.

Building the Business Service

Building Custom Framework Classes

Oracle recommends a best practice of implementing your own custom classes that extend the base framework classes. Even if you have no initial plans to put code into these custom classes they provide a "buffer" that gives you the ability to easily change base framework behavior.

Summit Model Custom Framework Classes

In our initial build of the application we decide to create custom framework classes for only the most commonly used ADF Business Components framework classes. These base classes were created in a separate package called "oracle.summit.base".

Framework Class	*Summit Framework Class*
CustomerVO	oracle.summit.base.SummitEntityImpl
OrdVO	oracle.summit.base.SummitViewRowImpl
OrdVO	oracle.summit.base.SummitViewObjectImpl
OrdVO	oracle.summit.base.SummitApplicationModuleImpl

We used the preferences feature of JDeveloper to define that these custom classes would automatically be used when generating the appropriate framework class.

One example where this practice was rewarded was where a method nextValSequence was added to the SummitEntityImpl class allowing all entity objects to easily read the next value from a specified database sequence.

First Cut Business Service

Given that database tables are the primary driver for the Oracle Forms version of the application, and that JDeveloper provides the ADF Business Components From Tables wizard, we were able to make a good first cut of the business service by using this wizard. This first cut also gave us the opportunity to add some of the more obvious, but easy to implement, features such as control hints.

Business Components from Tables

Since the number of tables involved in the database schema were relatively small, and that we could see that nearly all tables were directly used in the application, we felt it easier to use the ADF Business Components from Tables wizard to generate default entity and view objects for each tables. The effort of removing any business components objects that might subsequently be identified as surplus to requirement was deemed to be trivial.

The power of the ADF Business Components from Tables wizard, and ADF Business Components in general, is best utilized when the underlying schema is defined with the appropriate primary/foreign key relationships. These form the basis of associations and view links which are invaluable aids in implementing application behavior such as master detail coordination, data look ups, validation and list of values. While associations and view links can be expliciately created without the underlying database primary/foreign key relationships, we first of all ensured these were in place before generating our first cut business service.

Control hints

ADF Business Components provides a simple and easy way to define translatable strings for features such as labels, tooltip text and format masks. With ADF Business Components you have the ability to define control hints on the view object or the entity object. We chose the view object because we felt the view object represents the data as the user will view it, plus the view object may utilize attributes from different entity objects and their use, as reflected by a control hint, may be different in different usages.

We used the attribute control hint properties *Label Text*, *Tooltip Text*, *Format Type* and *Format* to define labels, help text and currency and date formatting. These mapped directly to the Oracle Forms item properties *Prompt*, *Hint*, and *Format Mask*. We also used the *Display Hint* to hide attributes that would not, by default, appear on the UI. This generally related to look up attributes used as part of the view object look ups (explained later).

Initial Values

When creating a new data record, some fields should automatically set to a default value. Oracle Forms has the *Initial Value* property and similarly, in JDeveloper you can set the initial value for an attribute. We did this to the view object for the following attributes:

View Object and Attribute	*Initial Value*
OrdVO.DateOrdered	adf.CurrentDate
OrdVO.PaymentType	CASH
OrdVO.OrderFilled	N
ItemVO.Quantity	0
ItemVO.QuantityShipped	0

Attribute property

In most cases, attribute properties such as *Queryable* and *Updatable* were left as the default values. In some cases it made sense to specifically override for some attributes, for example in OrdVO DateOrdered had the property *Updatable* set to "While New" to control that once and order was taken, the order date couldn't be changed.

Refining the Business Service

Having created a first cut of the business service and tested it through the ADF Business Components tester, we next moved on to refining that business service to implement application business logic.

Sequence Management

For most applications based on relational database tables, the assignment of unique values as primary keys is a principal requirement. The original Forms Summit application was not consistent in automatically assigning primary key values; in some cases the user was responsible for entering these values, which is obviously not desirable behavior in a production application. We took the opportunity to rationalize this behavior by creating database sequences for the relevant primary keys and assigning a value from the sequence through a database trigger.

To manage the sequence within the application, we set the relevant entity object attribute *Type* to "DBSequence" (which also automatically sets the properties *Updateable* to "Never" and *Refresh After* to "Insert.") This indicates that the value will be populated on the insertion of a new row and that the attribute is automatically refreshed with the new value.

This functionality allowed us to replace the following S_ORD block level Pre-Insert trigger.

```
SELECT S_ORD_ID.nextval
INTO :S_ORD.id
```

```
FROM SYS.DUAL;
EXCEPTION
  WHEN OTHERS THEN
    MESSAGE('Failed to assign Order Id');
    RAISE form_trigger_failure;
```

.

Entity Object and Attribute	*Database Trigger*	*Sequence*
CustomerEO.Id	NEW_CUSTOMER_TRIGGER	S_CUSTOMER_ID
OrdEO.Id	NEW_ORDER_TRIGGER	S_ORD_ID
ItemEO.ItemId	NEW_ITEM_TRIGGER	S_ITEM_ID

We did deviate from database trigger assigned primary keys in one place: ItemEO.ItemId. We found that the implementation of a click-to-edit table (which was the UI component on which order items would be edited) was easier to implement when a new row was added and the primary key for that row assigned when added. This also gave us the opportunity show a different method of implementing the assignment of primary key values. For ItemEO.ItemId, the value of a new row was assigned in the ItemsEOImpl create method.

```
protected void create(AttributeList attributeList) {
    super.create(attributeList);
    setItemId(nextValSequence("S_ITEM_ID", getDBTransaction()));
}
```

The Oracle Forms implemention used a composite of OrdId and a rolling ItemId which was calculated from the current highest ItemId + 1. This is obviously open to issues with concurrancy and we decided that for S_ITEM, the ItemId would be a unique number across all items.

Data Validation

At the core of the application is business logic that validates the data values input by the user. Here we describe how we implemented data and business logic validation.

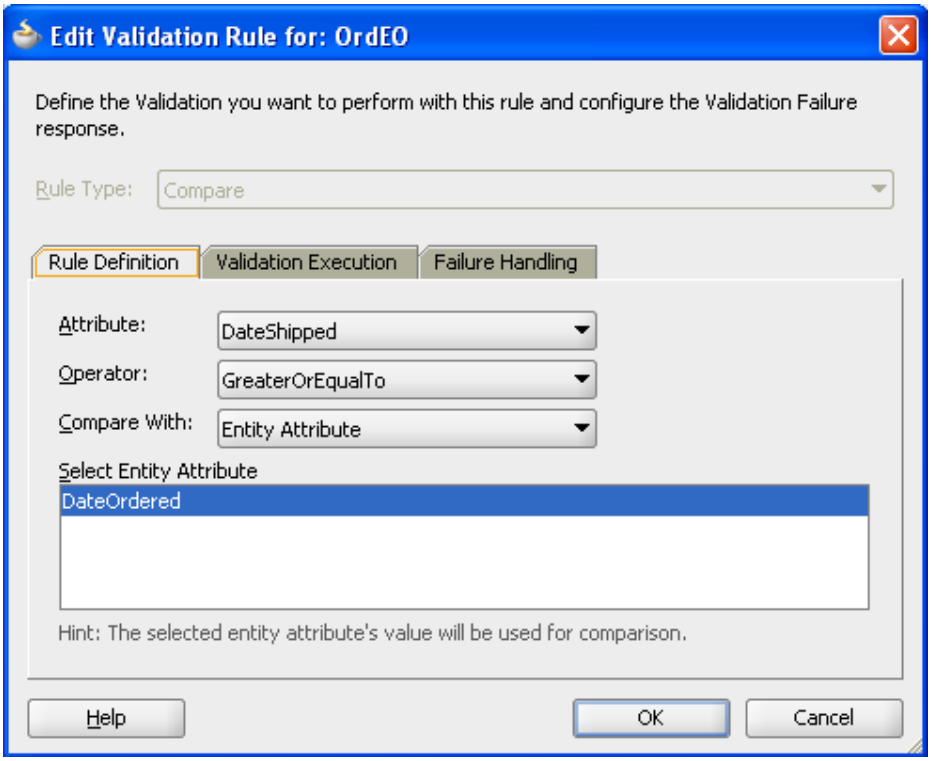
Validation of Shipping Dates

The application implements a validation rule that ensures that shipping date cannot be before the order data. This was originally implemented in a block level When-Validate-Item trigger:

```
IF :S_ORD.date_shipped < :S_ORD.date_ordered THEN
  MESSAGE('Ship date is before order date!');
  RAISE FORM_TRIGGER_FAILURE;
```

END IF;

In the Oracle ADF version of the application, we implemented the same validation rule declaratively using an entity object level compare validator:



Ensuring an order is for a valid customer

The original application did not specifically check that an order is assigned to a valid customer. It may be that this not a check that is required if we assume that an order cannot be moved to a different customer, and that an order can only be created for a customer that exists. However, we took the opportunity to implement this business rule through a key exists validation rule.

Check on Payment Type

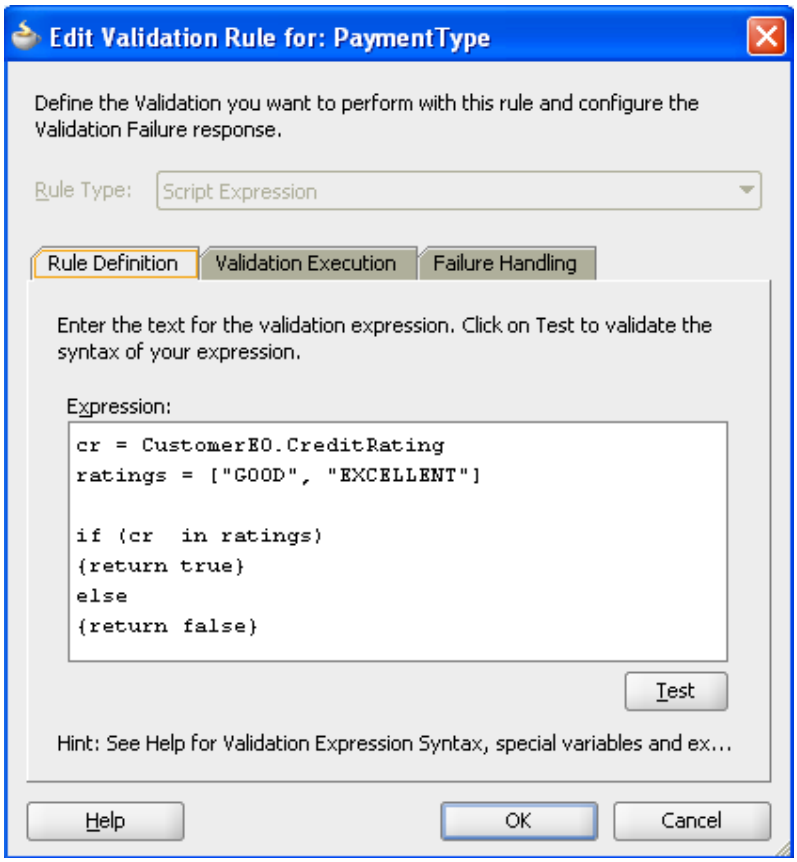
The application implements a business rule that the payment method for an order can only be on credit if the customer has the appropriate credit rating.

In Oracle Forms this was implemented using a When-Radio-Changed trigger on the Payment_Type field:

```
DECLARE
  N NUMBER;
  v_credit S_CUSTOMER.credit_rating%type;
```

```
BEGIN
  IF :S_ORD.payment_type = 'CREDIT' THEN
    SELECT credit_rating
    INTO v_credit
    FROM S_CUSTOMER
    WHERE :S_ORD.customer_id = id;
    IF v_credit NOT IN ('GOOD', 'EXCELLENT') THEN
      :S_ORD.payment_type:= 'CASH';
      n:=SHOW_ALERT('Payment_Type_Alert');
    END IF;
  END IF;
END;
```

The same functionality is implemented in SummitADF using a Groovy expression:



This uses an association accessor to access CustomerEO.CreditRating from the OrdEO.PaymentType attribute. If the payment type is one of the higher credit ratings then the validation is successful. The

rule itself is only executed for payment types that are credit. For cash transactions, the customer's credit rating is not relevant.

Changing a line item product

The application has a business use case where, for any order item, the user could choose to select a different product. In this case, the application has to ensure the product exists and that the line item price is now based on the wholesale price of that product.

In Oracle Forms this was implemented using a When-Validate-Item trigger on the S_ITEM.PRODUCT_ID:

```
SELECT name, suggested_whlsl_price
INTO :S_ITEM.description, :S_ITEM.price
FROM S_PRODUCT
WHERE :S_ITEM.product_id = id;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    MESSAGE('Invalid Product Id!');
    RAISE FORM_TRIGGER_FAILURE;
```

In SummitADF the same functionality was implemented through a key exists validation on the attribute and through augmenting the setProductId method exposed through the ItemEOImpl class. This allowed us to write code that would update of the item price to reflect the suggested wholesale price of the new product. We also generated ProductEOImpl so that we had type safe getters and setters for accessing the wholesale price.

```
    public void setProductId(Number value) {
        setAttributeInternal(PRODUCTID, value);


        //Code added to set the item price to be the wholesale price of the new product.
        ProductEOImpl prodInfo = (ProductEOImpl)getProductEO();
        Number x = prodInfo.getSuggestedWhlslPrice();
        setPrice(prodInfo.getSuggestedWhlslPrice());
    }
```

Shaping Data Views

Order By

In Oracle Forms, data order is usually implemented through the *Order* property of a block. In the SummitADF application we applied we set *Order By* on a view object to define the data order.

View Object	*Order by Clause*
CustomerVO	NAME
OrderVO	DATE_ORDERED desc
OrdVO	ITEM_ID

Lookups

Within the Summit schema, tables are related through primary and foreign key relationships, for example Customers has a column SalesRepId that relates to the Id column of the Employees table. These relationships are introspected by the ADF Business Components from Tables wizard. However, the information that is more relevant in the application would be the name of the sales rep referenced by this Id, and not the Id itself. In Oracle Forms, this would typically be implemented using a Post-Query trigger. For example, in Oracle Forms the block level Post-Query trigger for the S_CUSTOMER block has the code

SELECT last_name

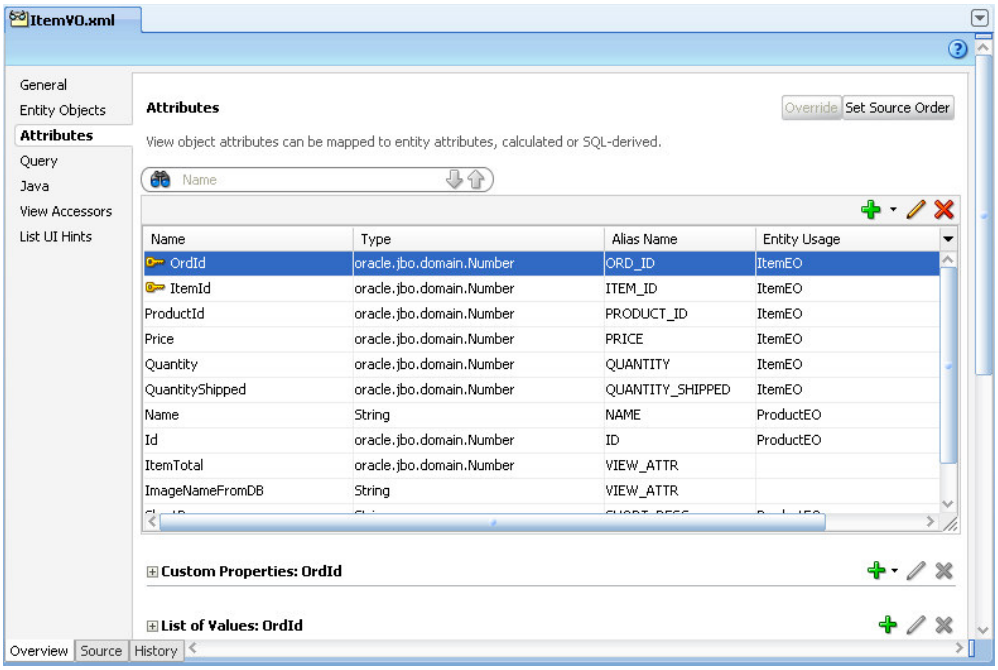
INTO :s_customer.sales_rep_name

FROM S_EMP

WHERE id = :s_customer.sales_rep_id;

The same functionality was implemented in SummitADF using view object attribute look ups. This was specifically implemented for:

View Object	*Lookup Usage*
CustomerVO	EmpVO.LastName
ItemVO	ProductVO.Name
OrdVO	EmpVO.LastName
	CustomerVO.Name



Read Only View Objects

To help facilitate list of values and validation, we identified data sets we regarded as read-only and defined them as view objects. This is a great way to ensure reusability; organizations can define read-only view objects that are used in list-of-values scenarios and other auxiliary functions of an application and package those view objects separately so that they can be shared by any application that shares the schema. This is especially beneficial for schemas that include many standardized codes that are used throughout the schema. All of the codes might not be used in just one application, but they can easily be reused across applications using ADF Libraries; additionally, any changes that occur in the logic or query for these view objects can be modified in just one source object, rather than having to implement the change for every application. In the initial release of SummitADF we kept these view objects as part of the application rather than refactoring into a separate workspace.

List of countries

Within the S_CUSTOMER table, there is a column named Country. Ideally the list of valid countries might be a held in a database table, but this didn't exist in the original Summit schema. We therefore, amended the schema to include a table of countries and created the appropriate entity and view object. The read only view object CountryVO, allowed the possibility to define a list of values or to form the basis of a validation rule.

Credit rating

Each customer also has a credit rating. The original Summit schema didn't provide a look up table of values; instead the value was a VARCHAR in the Customers table. Rather than allow free-form entry into this field, we added a table S_CREDIT_RATING and defined a read-only view object CreditRatingVO to hold the values.

Yes No

Each order has an attribute, OrderFilled, which indicates if the order is complete. While the underlying data is "Y" or "N" it would be more meaningful for the user to display "Yes" or "No". We therefore created a read only view object to map the data values to display values.

Defining a static view object in this way is a narrow use case and could be considered an anti-pattern because the values are hardcoded and more difficult to maintain. However, in this case, the existing schema doesn't provide for these codes and it was deemed a fair use of a static view object as a one-off solution to the business rule.

Lists of values

In the Forms Summit application there were a number of ways that a list of data could be associated with an attribute. The block attribute could be displayed as list item in which case list elements could be defined for that list item. Alternatively, a list of values could be created and associated with an attribute through the *List of Values* property, or called from a button press using code such as:

SHOW_LOV('sales_rep_lov')

In the SummitADF application we took the opportunity to rationalize the use and implementation of list of values by utilizing the list of values view object feature, and by defining view objects to represent the list data. Specifically we implemented list of values for:

View Object	*View Object attribute*	*List*
CustomerVO	CreditRating	CreditRatingVO
CustomerVO	SalesRepId	EmpVO utilizing bind variable to show only sales reps
OrdersVO	OrderFilled	YesNoVO
ItemVO	ProductId	ProductVO

Calculated Fields

A number of fields in the application have their data values calculated automatically via the application. This application had a number of such features.

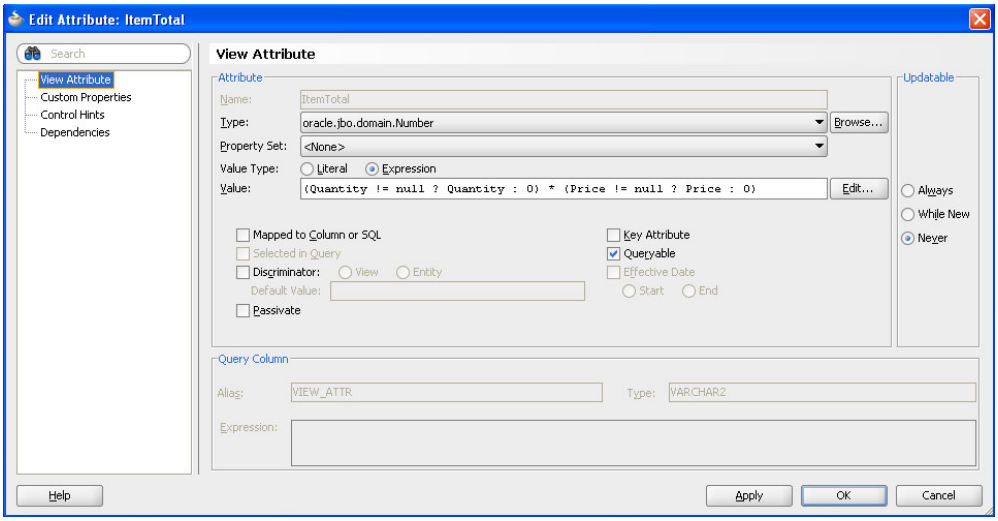
Calculating a line item total

For each order item, a line total is calculated which represents the product of line item price and the number of items shipped. In the Oracle Forms application this was implemented using the *Formula* property with the expression ":S_ITEM.quantity_shipped * :S_ITEM.price".

We felt that a more correct representation of the line total would be the product of quantity and price rather then the number of shipped items. So in SummitADF we added a transient attribute LineTotal based on the Groovy expression:

(Quantity != null ? Shipped : 0) * (Price != null ? Price : 0)

This expression also handled the case where a new line item was created but the relevant attribute may not yet be set.



Calculating an order total

The S_ORD table includes a column, Total, for persisting the total value of the order. While this exists as a database column, it's obvious that this value is dependant on the value of the order items. This functionality was not implemented in the original Oracle Forms application but we were able to implement this by updating the value of OrdEO.Total whenever an order item was changed, specifically whenever the price or quantity shipped was changed.

To implement this we generated ItemEOImpl and added code to the setPrice and setQuantityShipped methods to force a recalculation of the order total. We also generated OrdEOImpl to give us type safe getters and setters to the order total.

The SummitADF application automatically recalculates the order total based on any changes to the line item. Of course, the assumption is that the order total is already the correct sum of line items, which in the original data is not always the case.

Constructing the product image.

The Forms Summit application uses a database function to read the filename of the image of a product. In SummitADF we could (and probably would) have implemented this using a view object based on

the S_IMAGE table. Instead, we wanted to demonstrate that logic in the database could be easily reused in an ADF application.

To call a database procedure or function, we made use of a utility class that is available under GNU GPL license (see http://adf-tools.blogspot.com/2010/03/adf-plsql-procedure-or-function-call.html). This gave us a generic facility for calling database functions and stored procedures with various different parameter profiles.

We added a new transient attribute, ImageNameFromDB to ItemVO and set its value to the Groovy expression:

adf.object.readImageNameFromDB(new oracle.jbo.domain.Number(ProductId))

This then called a method in ItemVORowImpl, which called the appropriate utility functions to return the image filename. This filename was then used to read an image from the file system.

Application Module Design

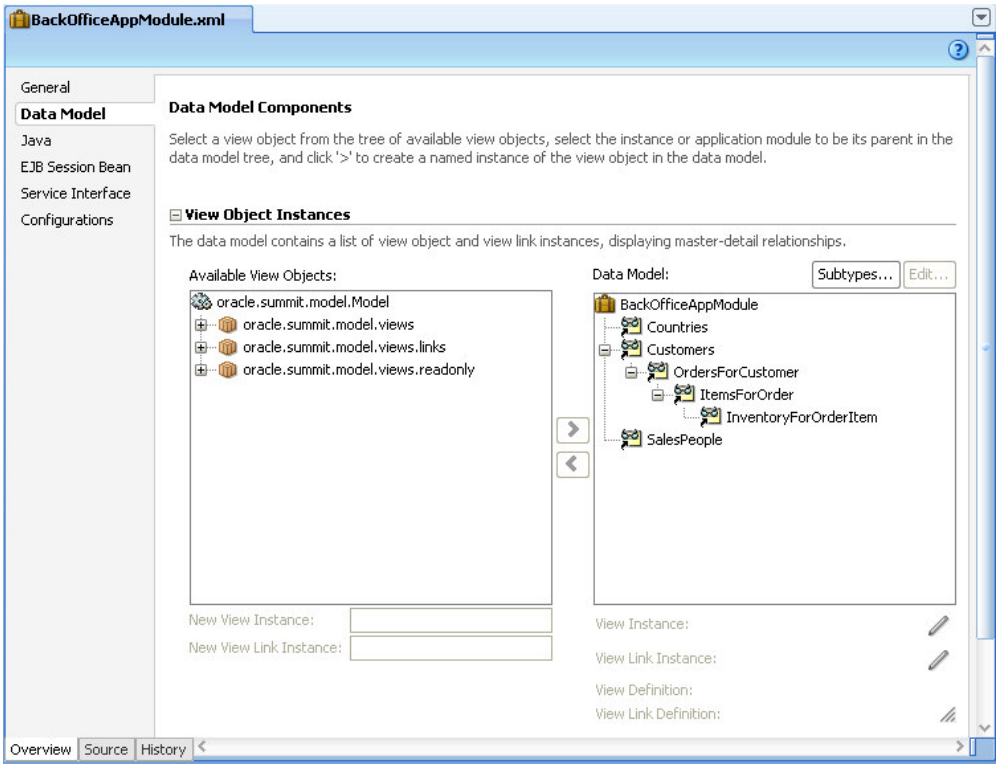
Each application module typically maps to a use case or discrete group of application work. In a Forms application, this might be considered analogous to a single Forms module, since each Form is generally a container for a discrete piece of application functionality. Furthermore, each top-level application module defines a database connection. We therefore wanted to ensure the correct granularity of application module without going as far as to have a top-level application module for every use case; which would obviously be expensive in terms of database connections.

Application Module Partitioning

Given that we envisaged SummitADF to expand on the original implementation, we decided that each of the broad channels of use of SummitADF would be top-level application modules. So, we still required a back office system to replace the existing Forms application, but might also consider a customer self-service web version. This gave us two top-level application modules: BackOfficeAppModule and CustomerSelfServiceAppModule. The former would be the focus for this development effort with the latter being in a subsequent development iteration.

View Object Instances

Given the existing business service model of Customers, Orders, Order Items we were able to create the following application module



As noted earlier, we renamed each view object instance in a way that more closely related to its function.

Identifying Business Service Functions

The initial Forms application was heavily based on CRUD operations on the underlying database tables. While this is typical of a Forms application, with an Oracle ADF application you have the ability to define a more process-based approach to both the UI and the business service. For example, rather than deleting a customer, instead a customer might be "archived". This business function could involve closing all existing orders that are currently open, setting the customer state to indicate they are archived and then dispatching an email to the customer to inform them their account is now suspended.

These more process focused business functions can be defined at a number of different levels in ADF Business Components. In ItemVO we defined a row-level method deleteOrderItem that, rather than simply deleting and order item, will first of all set the quantity to zero, thus resetting the order total before then removing the record. Thus, the service function for deleting an order item is not the default delete but deleteOrderItem

Building Application Flow

In building the application flow we decided that we first of all had to understand the main "tasks" that the application performed since these would conceptually map to ADF task flows. These were

broadly identified as maintenance of customers and maintenance of orders, which were then identified as bounded task flows.

Given the original Forms application was heavily based on simple CRUD actions, there were no other obvious business tasks that might also be candidates for task flows. However, we discussed possible scenarios that could be implemented as task flows in a future release (for example, "Ship Order" might involve navigation of a number of pages to confirm the order, stock, levels, confirm the packaging and delivery to a courier etc., rather than simply setting the OrderFilled attribute and committing the changed record – as happens now.

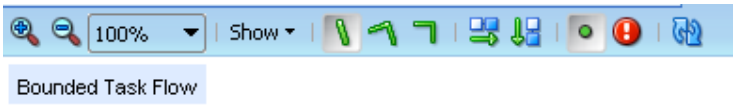
Top Level Unbounded Task Flow

We made the decision that the execution of these tasks would be performed within a SummitADF "shell" which is analogous to the MDI Window of the Forms application, but more importantly provided a container in which we built the application pages using page fragments. We created the "index" page, which would be the top-level page of the application, and this was the only entry in the adfc-config unbounded task flow.

Because the various tasks flows would be embedded inside this shell, each bounded task flow was created with page fragments. These bounded task flows were then embedded as regions.

Customer Task Flow

Given the relatively simplicity of the existing Forms functionality, the resultant task flows were also very simple. The customer task flow: customers-task-flow-definition, was simply a single view activity representing the customers region of the page.



Bounded Task Flow

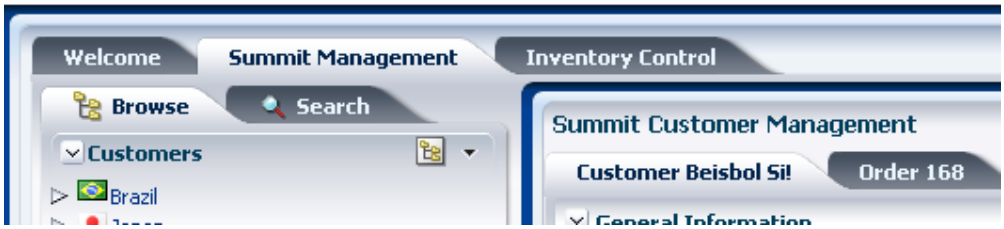
Orders Task Flow

Similarly, the bounded task flow we defined for managing an order was also a singe view activity implemented as a page fragment

Building the User Interface

High Level UI Design

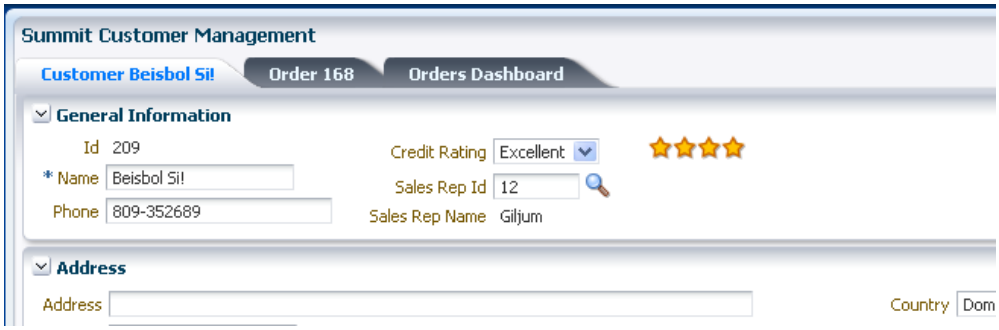
The vision was that different user for different business functions could use a single application. For example a back office data-entry clerk would use different screens to a warehouse employee who is more interested in stock levels; but it would still be the same application they were using. For this reason we designed the pages with a number of top-level tabs relating to business function.



In this implementation Summit Management represents the main screen for staff to manage the orders for any customer. In future releases of the application Inventory Control and other high-level and distinct business functions could be added.

For each of these top-level tabs there exist pages for carrying out that business function. We took the approach of having a page that was split with the left hand side providing functions, search and navigation with the primary interaction area being in the center. This resembles the general design layout used in many of Oracle's own Fusion applications.

Given that the management of a customer involves customer information as well as order information, we split this information across separate tabs. This allowed a user to quickly move between a customer and their orders without navigating off of the current page.



Page Design

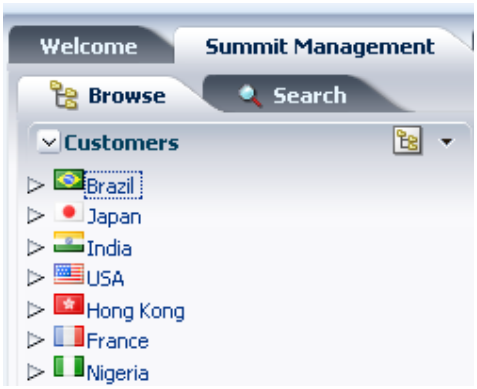
In order to provide an immediate "Fusion" look to the application we based the top-level application page, index.jspx, on the Oracle three-column template. We also felt that this would give us the scope to adopt the UIShell in a later iteration. The index.jspx page would serve as a top-level "frame" to the application and the various application pages would be embedded in this page as regions/fragments.

Customers Page Fragment

The Customers page is the main page in which a user is able to view information about a customer and to maintain that customer data. The page was implemented as a page fragment that was placed into the shell of the index page. This page comprised of the following main features.

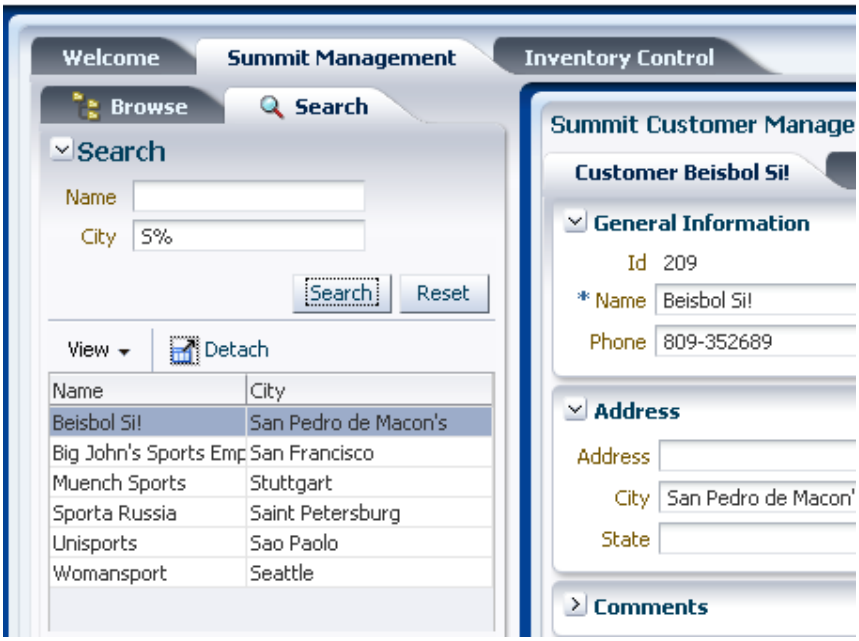
Searching

The Forms application provided two main mechanisms for browsing and searching for data. We were able to quite easily mimic the same functionality in SummitADF and exposed the functionality through two separate tabs within the left hand function/navigation panel.



Search Customers

We decided that there would be two attributes on which the customers might search: customer name and their location (city). We therefore implemented a view criteria on CustomerVO that provided this functionality. This was then exposed via a query panel as shown.



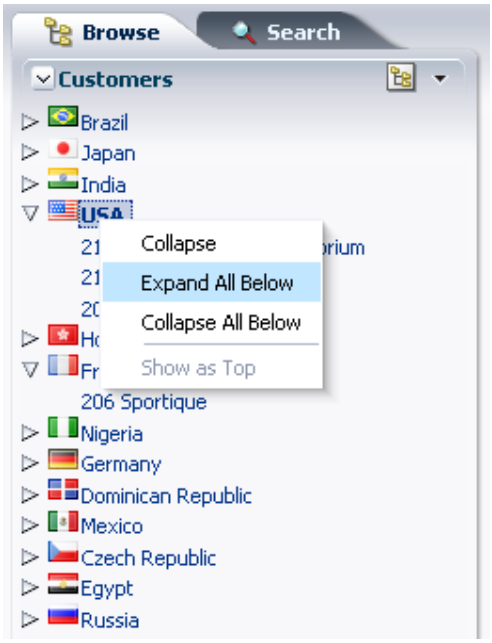
We implemented a feature in that when switching back to the browse tab, the view criteria would be unapplied. This was implemented through the use of the disclosure listener of the browse tab. This would call a backing bean that would call an action binding on a view object client method, which was responsible for clearing the view criteria. It is important to note that the code in this backing bean method would only interact with the binding layer. This meant that if the functionality of clearing a view criteria changed (for example, rather than totally clearing the view criteria to show all customers it still applied a view criteria to only show customers who are in your geographic region) then the UI is protected from this change given that it is fundamentally a business service decision.

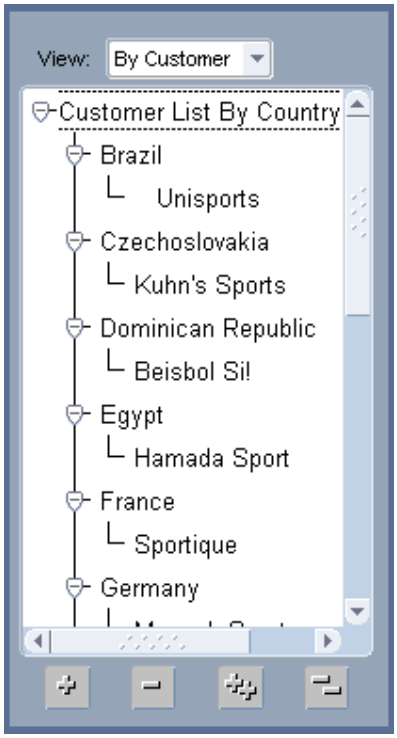
Browse Customers

The original Forms application allowed the user to be able to view the hierarchy of customers by country or by sales rep. Each tree control uses an instance of its underlying data control (SalesPeople and Countries) as the top-level data collection. The next level data collection is Customers and the tree binding uses the property *EL Expression* set to "${bindings.CustomersIterator}" to indicate that when selecting a customer in the tree control, the indicated iterator, in this case the CustomersIterator should also be refreshed. This ensures the tree control and the form are kept in synch.

In order to allow the user to switch between these two components, each was placed as a facet of an af:switcher component. The switcher component then uses its FacetName property to define which of the two faces should be displayed. In this implementation we decided to use view scope to hold the value of which facet should be displayed. So *FacetName* was set to "#{viewScope.customerTree}." We then added a context menu that that when selected would set #{viewScope.customerTree} to either "rep" or "country."

The tree control that shows countries and the customers in those countries also displays images of flags next to the flag name. This is implemented by adding an af:image component to the *nodeStamp* facet and setting its *Source* property to "/images/flags/#{node}.png". This means that the component will look for a png file with the same name as the country name.

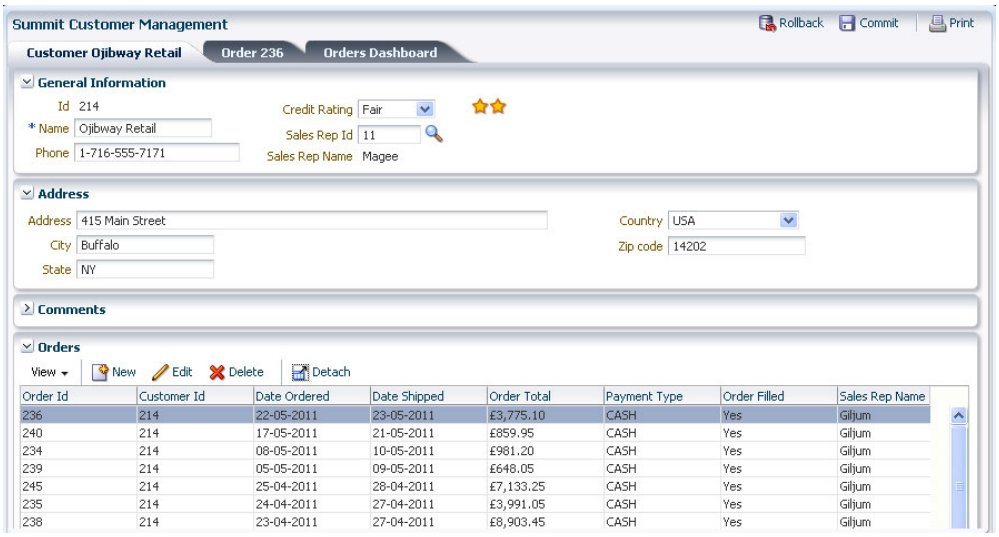




The ADF Faces tree component also included the default functionality to expand and collapse all the tree needs. In the Forms application this had to be implemented using four different buttons.

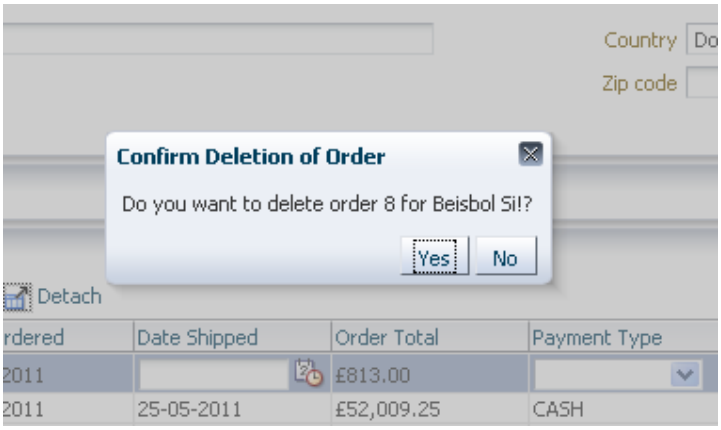
Customers Form

The primary maintenance of customer information takes places within this tabbed panel. We used panelBox components to allow the user to maximize screen estate by collapsing down groups of information that might be less relevant for the current action.



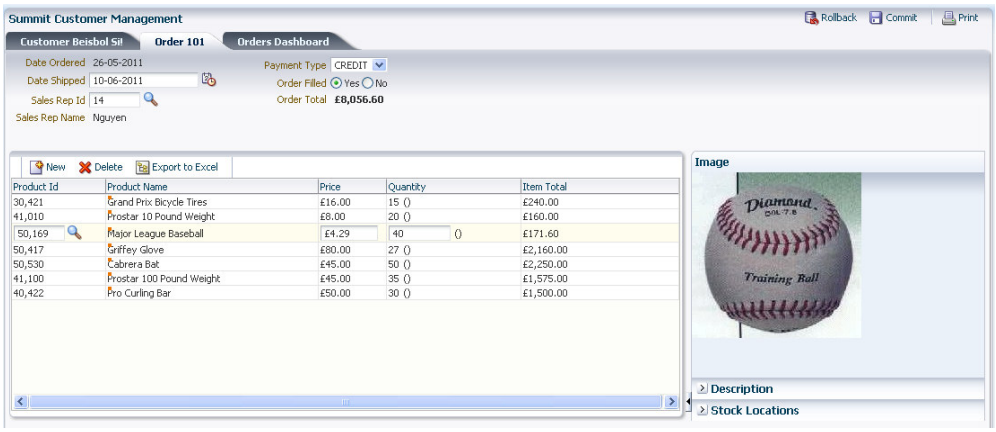
We leveraged the use of model driven LOVs to provide look-ups for key information such as Credit Rating and Sales Rep Id. Furthermore, by using an iterator component, we could stamp out a gold star rating for the customer. This is a UI gesture that will be familiar to users of sites such as Amazon where the user can immediately ascertain a rating level based on the number of stars.

The customer's tab contains a table that shows the orders for that customer. The user can click the edit button to edit an order or can click the orders tab, which indicates the currently selected order. Clicking the new button will create a new order and navigate to the order tab. On selecting the delete order button, the user is presented with a confirmation dialog.



Orders Page Fragment

This is the page in which the maintenance of a customer's order is undertaken. This consists of data related to the currently selected order, and an editable table or order items.



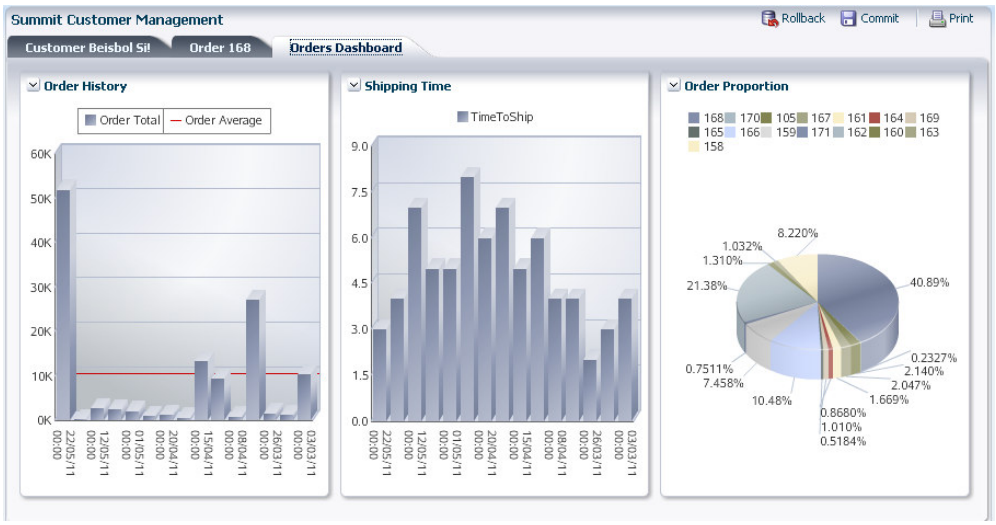
This page included the following functionality:

- Display image of the currently selected order item
- Show detail description of currently selected order item
- A pie graph depicting the stock levels and proportions for the selected order item
- Context info on Product Name to see further details on the order item
- List of values for selecting an order item
- Order Total maintained as items added or quantities changed.

Orders Dashboard

The third tab within SummitManagement is an orders dashboard. This provides various graphs to allow the user to visualize data trends for the selected customer. We found it relatively easy to build graphs based on various data collections and in the initial release demonstrated visual representations

of order history (with a reference line showing average order), shipping time, and a pie graph of the proportion of order totals for customer.



Each panel box in the dashboard can be moved and reordered within the dashboard.

Conclusion

There were a number of key points that arose from this project. Firstly, understanding what the existing application does, even before you consider opening JDeveloper, is a considerable undertaking in itself.

Secondly, embracing change was key to the relative ease of redevelopment. The relatively CRUD approach of the original Forms application could be developed directly in Oracle ADF, but with ADF's ability to defined service methods in the application module and view object, it felt natural to start thinking "what business action is being performed" rather that "data maintenance".

We also found that by "building the ADF right way" we were able to very easily add new features like a graphical dashboard, because it was simply a different view onto the same business services.

Finally, it is worth noting that while this was of course only a proof of concept, we wrote very little code. Other that using an existing library for calling database stored procedures, any code we ended up writing was in the region of 4 or 5 lines for each issues we were addressing.

In conclusion, Oracle ADF provided very similar concepts that could be mapped back to Forms. Furthermore, given the range of features such as model driven LOVs, view criteria, lookups etc, it was relatively easy to build comparable functionality in very little time. While we made a conscious effort not to simply re-implement the same application in a like-for-like manner, the sheer range of features and power of ADF meant we gravitated towards embracing the ADF way of building application and so it often felt quite natural to want build things differently from the original Forms application.

Appendix

Future tasks and next steps

The following were identified as areas where we could further enhance the application

- Internationalization/localization, including web services for currency conversion
- ADF Security
- Customization/Personalization with MDS
- Skinning
- More process driven user actions such as
    - Ship Order
    - Cancel Order
    - Create Customer
- Dependent LOV for state lookup by region
- Unit testing
    - Application module pooling

Contact address:

Grant Ronald
Oracle
Oracle Parkway, Thames Valley Park.
GU51 1EL, Reading, Berkshire

Phone: +44(0)1189249124
Fax:
Email grant.ronald@oracle.com
Internet: blogs.oracle.com/grantronald