CREATING A DEVELOPMENT ENVIRONMENT USING ORACLE'S FUSION MIDDLEWARE

Dr. Paul Dorsey, Dulcian, Inc.

Since 2005, Oracle Corporation has been buying up other IT companies and technologies. For the first time in Oracle's history, it has decided that creating a first-rate application development environment is the core critical success factor for the corporation. This decision has tremendous impact. That said, it is going to take several years for Fusion to get up to speed. Consequently, it is likely that there will be unprecedented support for the Oracle Fusion technology stack.

What is this technology stack? How can we take advantage of it? Fusion Middleware includes a subset of JDeveloper. Throughout its evolution, JDeveloper has tried to support a wide view of building applications. For example, even though Fusion uses Oracle's Application Development Framework—Business Components (ADF BC) to support connections to the database, JDeveloper also supports TopLink (another Oracle acquisition) as well as developers who create their own persistence using Enterprise JavaBeans (EJBs). JDeveloper supports the EJB3 technology as well. On the client side, even though Fusion uses ADF Faces (Oracle's proprietary, next generation of UIX-type extensions to the JavaServer Faces environment), JDeveloper also supports JSPs (with or without Struts), thick Java client (with or without ADF extensions), and generic JavaServer Faces development.

So, what development environment should you use to build Web applications? Should you jump on Oracle's large and well-funded bandwagon and build applications in exactly the same way that Oracle is building its internal applications? The simple answer is "Of course." Oracle's ADF Faces promises to be a highly efficient and reasonably rich development environment. However this development environment has a very steep learning curve if you need to go beyond the standard capability of the framework. Oracle seems to be building a 4GL wrapper for a 3GL world. At this point, it is somewhat immature and unforgiving, although this will likely be improved as the technology matures. Currently, it is difficult to create custom applications with features not explicitly supplied by ADF Faces.

This paper discusses some of the issues involved in creating a development environment for Oracle Fusion Middleware to build applications.

Elements of the Fusion Middleware Environment

The Oracle development team is still refining its architectural vision. The decision about which parts of the technology stack will actually be used, and how they will be used, is still in flux. Oracle's entire architecture is built using a Service Oriented Architecture (SOA) backbone, designed to support inter-system connectivity. This approach may not be appropriate for standard application development. Trying to apply SOA tools (such as BPEL) inappropriately to normal application development may create a system architecture that does not provide adequate performance for many systems. There are several different parts of Fusion Middleware that Oracle will be using internally. These are discussed in the following sections.

1. The Oracle Application Server (OAS)

Oracle's Application Server (OAS) has been around for many years. It is a solid enterprise application server that is appropriate for any organization's use when working in the Java EE environment. OAS is tightly integrated with the Fusion Suite. This tight integration with the rest of Fusion Middleware is a huge benefit which can save an organization many thousands of dollars.

Building applications with JDeveloper using the Application Development Framework (ADF) allows you to deploy your system to OAS with a single button click. For organizations not yet working in a Java EE environment, it may be difficult to understand this huge benefit. Typically deploying to an application server is an entirely separate project all by itself. It can take weeks of building and debugging deployment scripts to support an application deployment. Each time an application is significantly modified, the deployment script must be refactored, prior to redeployment. Using OAS and ADF eliminates most of these difficulties.

2. The Application Development Framework – Business Components (ADF BC)

Interacting with a relational database is the aspect of a system in development that most commonly causes projects to fail. Most Java developers reject the idea of using anyone else's framework and instead try to create their own. Since database transactions are often not taken into account, this approach usually results in terrible system performance. Even worse, such frameworks often provide the illusion of project progress, allowing prototypes to be demonstrated early on in the development cycle. Only later does it become evident that the underlying architecture is unsound with respect to scalability and performance.

Oracle's Application Development Framework – Business Components (ADF BC), which evolved out of the earlier Business Components for Java (BC4J) architecture, is the most sophisticated persistence framework in the industry. It has greater functionality, richness, and database awareness than any other framework, including Oracle's own TopLink framework which was purchased several years ago. The ADF BC framework has gone through a number of significant revisions. Each time, the framework emerged richer and more robust than before. Unfortunately for those of us building production applications using this framework as it evolved, applications built using each version had to be refactored every time. This led to some amount of frustration with the environment.

For those coming "late to the party" the ADF BC framework is stable, mature, and not likely to be redesigned in the next several years. Object-Oriented developers are likely to resist using the ADF BC framework (mainly because they didn't write it). It does require some careful study to understand and fully exploit its architecture. There is no more important decision that an organization can make with regard to a systems project than to build on a solid, stable, persistence environment. There should be compelling reasons for an organization not to choose the ADF BC framework. It is the best framework in the industry.

3. ADF Faces

ADF Faces consist of Oracle's extensions to the basic Faces components. They are the user interface elements that appear in applications, are richer from a UI perspective (some are AJAX-like in their sophistication) and can be bound automatically to ADF BC components. ADF Faces components are the next generation of Oracle's UIX components which were used in the last iteration of the E-Business development environment. Many of their design characteristics mirror those of their UIX predecessors. These components are still evolving and are certainly less mature than the ADF BC portion of the overall ADF framework. Some of the components, such as the tree control, are poorly architected and do not scale well for sophisticated applications. However, most of the ADF Faces components have a rich look-and-feel, are easy to use, and well designed. Oracle is working very hard to provide extraordinary richness in this part of the development environment.

A new area of the architecture that many initially overlooked is the WebCenter, which is an impressive architectural achievement. Oracle has made its own Oracle Portal product obsolete since any ADF Faces page can act as a portal, and any portlet can be wrapped as an ADF Faces component. It is this level of architectural depth that Oracle continually demonstrates which should make everyone look very closely at ADF as a development framework. Oracle is allocating more resources to this framework than any competitor and, as a result, is expanding its functionality more rapidly than that of any other environment.

Although ADF faces is still somewhat immature and not without its challenges to extend, using ADF Faces for development is a good choice. Any of the core Faces components can be bound to ADF BC and extended as needed. Choosing ADF Faces as the core structure does not limit developers to the functionality that it provides. It is possible to either alter the component or start with the core Faces control and extend it using hand-coding techniques in a traditional environment.

The evolution of ADF Faces is still in progress. The Controller layer in prior versions was the standard Faces controller. Since it lacked even the functionality of the Struts controller, Oracle is now replacing it with a proprietary controller based upon its earlier experience with the UIX controller. This controller is included in the JDeveloper 11*g* release.

4. Business Process Execution Language (BPEL) & Service Oriented Architecture (SOA)

The Business Process Execution Language (BPEL) and Service Oriented Architecture (SOA) Suite are some of the hot buzzword technologies du jour. BPEL is used to implement complex process flows with SOA architectures. Each node in the BPEL process has both the flexibility as well as the handicap of making a call to a web service. The problem is that web services or any services supporting calls from one system to another are inherently resource-intensive and usually slow. If the process has only a small number of nodes and does not require executing more than a few nodes at a time, BPEL architecture is adequate. However, in a complex application with thousands of logical nodes that are database intensive (which is not unusual) and could be implemented entirely within PL/SQL, using BPEL results in a routine that may run hundreds, if not

thousands, of times slower. BPEL is popular with developers who do not understand the power of logic executed in the database.

Oracle's SOA Suite consists of all of SOA-related parts of ADF including JDeveloper, Oracle Business Rules, Oracle BPEL Process Manager, Oracle Business Activity Monitoring, Oracle Web Services Manager, and the Oracle Enterprise Service Bus.

JDeveloper is not only an SOA tool. You can build applications that have little or no SOA orientation with JDeveloper. JDeveloper is the main development environment for building any Java EE application.

Oracle has delivered an excellent implementation of BPEL. A first rate drawing tool to describe the BPEL process is included. The execution is as efficient as possible, given the weakness of the underlying architecture, and is suitable for applications where a SOA orientation is appropriate. The concern lies with the suitability of the BPEL architecture in general. Most systems do not have the luxury of ignoring performance considerations. Making hundreds or thousands of round trips between the database and server for a single transaction is usually enough to render a system unusable. The SOA suite is built around managing service-oriented processes.

Most systems neither need, nor would they benefit from, this type of architecture. Applying an enterprise solution to a database-centric application will result in spending more money for hardware and still getting inadequate performance. It is important to carefully evaluate to what extent a SOA architecture makes sense. If taking disparate systems and helping them interact in something other than an ad hoc fashion is the goal, then a SOA approach makes sense. In order to demonstrate the need for SOA and BPEL, typically vendors discuss applications for which these architectures make no sense (usually a sale or purchase order processing). Describing a system where a SOA architecture is appropriate is fairly complex and does not lend itself to a short conference paper. Frequently, people building SOA products are not, themselves, the same ones who build end user systems. As a result, they may not explain the best way in which to use the tools they are developing.

The SOA-specific parts of the SOA suite include Oracle Business Activity Monitoring, the Oracle Web Services Manager, and the Oracle Enterprise Service Bus. Unless you committed to the SOA architecture, you can hold off on utilizing these tools.

5. Oracle Business Rules

Oracle Business Rules seems to have finally found a place in the Oracle world. However, I have still not encountered anyone outside of Oracle who has made productive use of this product. I think it best fits within the context of a BPEL flow, so unless you are already creating complex BPEL flows, you can safely ignore this product.

I have to admit a clear bias here. My company has its own business rules engine (BRIM®) and my bias is towards executing business logic in the database rather than in the application server wherever possible.

The Oracle Business Rules grammar seems to be clumsy and inconvenient. To date, I have not found any project where I would use it. I do have a colleague (not associated with me or my company) who spent three months trying to use the product before abandoning it. A web search for white papers about Oracle Business Rules yielded a few people who are excited about the prospects for the product, but were unable to get very far due to its immaturity.

At this point, the safest course is to adopt a "wait and see" attitude. Even if Oracle Business Rules will someday be a valuable part of the technology stack, that day has not yet arrived.

Concepts to Keep in Mind

To discuss how to use Fusion Middleware for custom application development requires understanding several important concepts:

1. **Use a "thick database" approach.** As stable as this architecture is, it is still evolving. Anything created anywhere other than the database is at risk of having the underlying architecture itself evolve, thus requiring changes. Software built in the middle tier may be architecturally out of date fairly quickly and need to be rewritten. The only defense against an evolving architecture is to use it as little as possible. Place as much of the business logic as possible into the database, thereby decreasing reliance on the evolving technology. Given the current state of this technology, you can expect the following benefits:
    - Write half as much code.
    - Have more efficient code.

PL/SQL that interacts heavily with the database is much more efficient than Java and will run much faster (10 times). Due to PL/SQL's performance efficiencies, network traffic between the application server and the database will be reduced by 99%. The database itself will only be utilized half as much, even though there is more logic because it is so much more efficient to perform all of the processing within the database, rather than in many separate retrievals and updates of information from the application server.

2. **Use ADF BC**. If most of the system logic is in the database with appropriate complex views, creating ADF BC images of these objects is a simple task. However, there are instances where all of the capability of ADF BC must be utilized. A full discussion of ADF BC is beyond the scope of this paper. (See Roy-Faderman, Koletzke & Dorsey, *Oracle JDeveloper 10g Handbook* (2004, Oracle Press) and OTN white papers for more information.

3. **Learn how to use ADF Faces appropriately.** This is also a very large topic. See Kolezke & Mills *Oracle JDeveloper 10g for Forms & PL/SQL Developers* (2006, Oracle Press) and OTN white papers for more information.

4. **Be very careful about the controller decision**. There is a brand new controller in JDeveloper 11g. How will it perform? How buggy will it be? In our organization, we are using a very simple controller layer and allowing the controller logic to be handled using a server-side routine. Over the last few years, the recommended controller of choice has shifted from the Oracle UIX controller, to the Struts controller to the Faces controller and finally to a new proprietary controller. Perhaps some level of caution is appropriate in this area.

The Thick Database Development Process

One of the real strengths of the thick database approach is that the two portions of an application can be coded independently. Once the interface between these two parts of the application is defined, the teams can work in isolation until substantive portions are working.

In practice, this approach works very well. Usually the first version of the user interface is built within a few days so it can be used as the first version testing environment for the database team and feedback can be received from users.

To maximize the efficiency of this approach, an Agile process is used rather than a linear set of development steps. Minimal design work is done to produce a partially working system. Additional functionality is then created in an iterative design process.

Working in the BRIM® environment simplifies some things for us that might be harder in a non-BRIM® environment. For example, it is not necessary to worry about workflow or data validation business rules as part of the UI design since the BRIM-Object functionality takes care of this part of the process.

The basic steps of the thick database development process are as follows:

1. UI Design

   Screens are designed on paper and white boards are used for page flows. Real screen mock-ups are usually a waste of time. A careful diagram on a piece of paper suffices for the initial UI design.

2. Interface Design

   Once the UI design is complete, the exact views required are determined along with the APIs that will be called.

3. Interface Stubbing

   The first coding performed is stubbing out the code for the views and APIs. The views can be as simple as `select <values> from dual`. The APIs are nothing more than functions that return a correct value (usually hard-coded). The idea is to write the stubs as quickly as possible so that the UI developers can get started.

   Interfaces will change as the application matures. Sometimes what is returned changes, additional columns are needed, or even whole new views or APIs can be added. In each case, even if the development time is fairly short, they should be stubbed out first so that the UI team can incorporate them while they are being developed.

4. UI and Database Development

UI and database development take place at the same time. The UI team takes the APIs and incorporates them into the application while the database team makes them work.

How thick is too thick?

What would happen if the thick database approach were followed completely and 100% of all UI logic was placed in the database? Placing all of the logic in the database and requiring round trips for everything implies the following:

- Tabbing out of a field requires a round trip to find out whether or not the field is required.
- Opening an LOV requires populating it from the database.
- The database is queried every time you leave a page to determine the next page in the navigation sequence.

This would be a pathologically complete way to implement the thick database approach. Clearly a system built this way would be sub-optimal.

At Dulcian, an application using exactly this approach was created for a web-based customer order entry system that had a very large number of business rules. For example, based upon the value in a given field, some fields would be enabled while others were disabled.

All rules for the application were stored in a database repository. The easiest way to build the first version of the system was to leave all of the rules in the database and access them at runtime. Surprisingly, this version of the system worked. Users complained that the screens "flickered" as data was entered, but they were otherwise satisfied with the applications and did not request additional changes to improve performance. Since the screens have minimal logic in them, screen refresh is sub-second over a normal internet connection.

This application demonstrates an amazing proof of concept and supports the idea that using a 100% thick database approach is viable. The system mentioned above was created with no logic whatsoever coded in the UI. Every UI behavior required a trip to the database to determine what should happen next. Despite this, the system still performed reasonably well and the database was not overwhelmed. However, this approach is clearly not optimal since it requires at least one potentially unnecessary round trip on each UI action and there are some extra unnecessary database queries and executions being performed. However, these round trips are limited and the superfluous database execution is minimal. Systems do not fail because of a few round trips between the application server and the database. Systems fail because of single screens requiring hundreds of round trips or due to passing poorly formed SQL to the database to execute.

Based on the results described here, it is possible to create a thick database system without fear of performance issues. Marginal performance improvements can be made by reducing trips to the database where appropriate, but this step is not essential to creating a viable system. This does not mean that poorly written code is acceptable as long as you use the thick-database approach. A slowly running query will run slowly no matter where it is executed. Thousands of unnecessary queries will bring a system to a halt even if they are all initiated from within a database-side procedure. But a 100% thick database approach is possible.

How thin is too thin?

Can a skilled team successfully build applications that are 100% database "thin"? In order to have any hope of success, building a system with no reliance on the database will have to be done by a highly skilled team.

UI developers must understand how to minimize round trips to the database to avoid performance problems. Under the best of circumstances, this can be a very difficult problem. Consider the requirement of displaying a customer information screen with data taken from many different sources. Many round trips to the database will be needed unless the developer is very clever and uses nested SELECT statements like the following:

```
Select
    (Select … from … ) value1,
    (Select … from … ) value2,
    (Select … from … ) value3
From dual
```

Use of other middle tier technologies such as BPEL can also be a performance killer. If a large percentage of the nodes in a BPEL flow require access to the database, then many round trips will be needed. BPEL users should try to cache all necessary data required by a BPEL process in the middle tier prior to executing the flow.

A first rate team can usually survive ignoring the database, but this is a very difficult way to develop.

Stateless Programming

Client/server development was inherently stateful. Developers counted on user sessions being persistent, used persistent global variables and assumed that transactions were only being committed to the database on demand.

Web development usually needs to be stateless. Typically there is a reusable set of sessions in a session pool, which means that you cannot rely on session-specific information to support an application.

Database developers tend to strongly resist moving into a stateless environment. Stateful coding techniques are very hard to give up. However, developers must abandon a stateful development style in order to successfully build web applications.

There are times when you can use a stateful web development style. You must ensure that the connection pool is larger than the maximum number of concurrent users. For internal applications, this can be an option, but you are really just postponing the inevitable shift to stateless programming.

Building a User Interface Using ADF Faces

Until recently, high quality user interfaces have been the exclusive domain of client/server applications. Web application user interfaces have traditionally been slower and of significantly lower quality than their client/server counterparts. JSP/Struts applications came nowhere near the quality of Oracle Forms or PowerBuilder applications. Development that could be accomplished in C++ or Java (using Swing components) could not even be imagined on the web. What could be done, took much longer than in the client/server environment.

Oracle's ADF Faces has significantly closed the gap between client/server technology and web technology. ADF Faces applications can easily rival Oracle Forms applications, both in quality of the user interface (UI) as well as in performance.

Perhaps more significant is that the "development gap" has also narrowed. It is now possible to build applications just as quickly in ADF Faces as it was in products such as Oracle Forms. If developers use a "thick-database" approach and place most of the logic in the database, then applications can actually be built faster than using a traditional client/server-based method.

Conclusions

In spite of JDeveloper becoming easier to use with each release, there is still a large learning curve. Most of the discussion surrounding Fusion centers on the SOA Suite and urges a SOA orientation. Organizations that have blindly leapt off of the SOA cliff in search of the "next new thing" have encountered the same challenges and problems faced by most early adopters of any new technology. Even with the most stable portion of Fusion Middleware (ADF BC), the critical success factor in effective utilization of this technology is leaving much of the logic in the database. Throwing all of the business logic into ADF BC can easily generate working applications, but these applications usually experience unacceptable performance.

ADF Faces provides a very nice look-and-feel and a brand new Controller layer in JDeveloper 11*g*. Developers should create a few test applications and thoroughly test network performance and scalability prior to embarking on a large development project using this technology.

The entire SOA orientation should be approached with even more caution. When system interaction is required, this interaction should be confined to simple web services calls. Developers should learn the basics of the technology and determine some of the pros and cons of its use before adopting the entire complex set of tools and utilities that are admittedly appropriate to support Oracle's massive Project Fusion effort. These tools will likely overwhelm smaller organizations coming from a traditional Oracle development environment. Even organizations with significant Java talent have frequently made incorrect system architectural decisions resulting in project failure through inappropriate utilization of SOA architecture. After a honeymoon period where SOA was heralded as the next "silver bullet," recognition of the complexity of its architecture that is not without challenges and traps has become more widespread.

About the Author

Dr. Paul Dorsey is the founder and president of Dulcian, Inc. an Oracle consulting firm specializing in business rules and web based application development. He is the chief architect of Dulcian's Business Rules Information Manager (BRIM®) tool. Paul is the co-author of seven Oracle Press books on Designer, Database Design, Developer, and JDeveloper, which have been translated into nine languages as well as the Wiley Press book *PL/SQL for Dummies*. Paul is an Oracle ACE Director. He is President Emeritus of NYOUG and the Associate Editor of the International Oracle User Group's SELECT Journal. In 2003, Dr. Dorsey was honored by ODTUG as volunteer of the year, in 2001 by IOUG as volunteer of the year and by Oracle as one of the six initial honorary Oracle 9*i* Certified Masters. Paul is also the founder and Chairperson of the ODTUG Symposium, currently in its eighth year. Dr. Dorsey's submission of a Survey Generator built to collect data for The Preeclampsia Foundation was the winner of the 2007 Oracle Fusion Middleware Developer Challenge and Oracle selected him as the 2007 PL/SQL Developer of the Year.