PRACTICAL ADF APPLICATION DEPLOYMENT FOR FUSION MIDDLEWARE ADMINISTRATORS

Simon Haslam, Veriton Limited

Paper submitted to: ODTUG Kaleidoscope 2008 conference

Abstract

Oracle's Application Development Framework (ADF) and Business Intelligence Publisher are expected to become the tools of choice for Oracle-based organisations wishing to replace their tried and trusted Forms/Reports enterprise applications.

This paper takes a less common view of ADF: how to effectively manage applications running on ADF, rather than just how to build them. It talks the administrator through key ADF components, deployment techniques, behavior under load and suggests numerous best practices. The scope is restricted to ADF BC, ADF Faces and JSF, namely those technologies being used by Oracle Fusion Applications and many custom development projects.

1 Introduction

Oracle's Application Development Framework (ADF) is a rapidly maturing product that helps organizations to build modern web applications using Java technology. ADF is very flexible and provides many options – technologies for the user interface and business logic layer, different deployment architectures right through to support for various vendors' application servers and databases.

Despite such a broad technology choice, as time goes by, applications built with a Java Server Faces (JSF)/ADF Faces user interface (UI), and ADF Business Components (ADF BC) layer for business logic, are expected to become increasingly dominant. This is consistent with Oracle's own development efforts, whether you look at upcoming Fusion Applications or Oracle Consulting's JHeadstart application generator. This paper focuses on these technologies and, whilst it pertains primarily to the production 10.1.3 release, it will mostly still apply to 11g too.

There are plenty of Oracle manuals, white papers, forum conversations and JDeveloper help pages describing ADF, but the author has found it difficult to find easy-to-use documentation detailing just what the administrator needs to know, without having to become a fully-fledged ADF developer. This paper aims to redress this balance and will consider techniques that use the standard management tools included with Oracle Application Server, rather than JDeveloper.

The author firmly believes administrators should have a healthy interest in destructive testing so the paper also talks about ADF behaviour under load. This topic is worthy of a paper in its own right but the intention here is to indicate areas where there are performance 'pinch-points' and to encourage the reader to investigate further.

For the purposes of this paper we consider the 'Administrator' to be the person responsible for deploying and managing a production Application Server environment, plus possibly the pre-production and/or quality assurance (QA) testing environments. The Administrator's key responsibilities therefore are ensuring that the implemented applications run in a reliable, consistent and well-performing manner for their users.

Finally, 'best practices' outlines here are based on the author's experience and those of others, the Oracle documentation and research/discussions. However please treat these within the context that they are intended, namely an ADF Faces/BC application with perhaps 500 or so users, deployed in web mode on Oracle Application Server: the further your circumstances deviate from this scenario, the more you will need to make your own judgements. As with most things to do with system implementation, one size rarely fits all.

2 Anatomy of ADF

2.1 ADF Business Components Architecture

An ADF application is a regular J(2)EE application so its deployment and tuning will be familiar to those who have run other Java applications on OC4J. The framework is driven by a combination of web content files (such as JSPs), XML configuration files and java beans; all supported by a set of ADF class libraries.

Many of us are familiar with the overall Model-View-Controller architecture of ADF. Less commonly discussed is how, in a typical implementation, the actual ADF objects relate to each other, the java container and the database.

To cope with the stateless nature of HTTP the application server maintains information for every user that is preserved between requests. This information covers user security details, what s/he is doing, pending data changes they have made, etc. In fact, state management is covered in two parts – the JEE session management features and JSF controller which, using ADF Faces components, handles the state of the UI, and the ADF BC Application Module which maintains the state of data controls, etc.

We'll consider the key components and choices for the Administrator in the following sections.

2.2 Application Modules

For the administrator the Application Module (AM) is at the centre of the ADF runtime environment as every call from client web browser to pages with data controls (the vast majority in a typical ADF application) is connected through to an AM instance. By means of the other ADF BC objects, it also stores information about the session and data controls: the current rowset (though not actually data), updates to underlying tables that have not yet been posted to the database, transactional state, etc.

However, if every user's session had to have its own dedicated AM instance, this could prove costly and inefficient in terms of memory resources (java heap space), particularly where the users have a relatively long 'think time' between page requests (keying in customer data for instance). To improve scalability Oracle have implemented a pooling mechanism, whereby the same AM instances are shared by different sessions and the user's state information is saved away as required. Writing a snapshot of the AM instance state to the persistent store Oracle calls 'passivation' and retrieving it is called 'activation'[1].

Ideally every request from a client would get routed back to the same Application Module instance – if this is not possible, e.g. due to the number of active users, then a different Application Module instance is taken from the pool, re-activated[*] and only then made available to the ADF data controls.

This passivation/activation workload is particularly important when an application is under stress as, depending on the application complexity, it can contribute significantly to overall system load. It does, however, increase scalability as it means resources requirements to not scale linearly with number of user sessions.

Why else should the administrator care about application modules? Well, in ADF BC's default configuration the number of AM instances determines the number of database connections required and so sessions in the database, another precious resource that the administrator will want to conserve.

2.3 Other ADF BC Objects

There are other Business Components that may appear on the administrator's 'tuning radar,' for instance, certain View Objects may need to be tuned using SQL and indexes as per traditional DBA/developer activities. Again this is an interesting area and the reader is directed to Steve Meunch's blog[2] which has many relevant articles.

2.4 Database Access

ADF BC offers two types of database connection specifications: JDBC URL and (JNDI named) data sources. With a JDBC URL ADF uses the JDBC driver to connect directly to the database, though note there is an optional database connection pooling mechanism available *within* ADF BC.

[*] actually, this behaviour depends on the failover mode – by default the AM instance will also have to passivate (save the previous session's state to the persistent store) and reset prior to activation for another user's session

A JDBC URL requires database name/credentials hard-coding within the ADF configuration, but there should be no need for the administrator to use this method. Even if you need to use a JDBC driver without pooling this can still be done by setting up a 'native' data source within Oracle Application Server and referring to that in ADF instead.

The author's preferred method is to use a container-managed JDBC data source as it allows the connection pool to be easily managed and monitored by the administrator, as well as supporting Oracle's newer features such as Fast Connection Failover (which reduces the impact on the application of node failure in a RAC database cluster).

2.5 Deployment Modes

Returning to the overall architecture, an ADF application is deployed across several tiers:

- *Client tier* which interacts directly with client (e.g. web browser, Swing applet)
- *Web tier* that generates UI on App Server (e.g. JSP, JSF faces servlet)
- *Business tier* providing business logic (e.g. app modules, entity/view objects)
- *Database tier* running the backend database

Each of these could be on different (or the same) machines. In ADF terms, for a JSF web application (rather than, say, a Swing client) there are two deployment modes for us to consider:

- *Local Mode:* both the web tier (JSF/ADF Faces) and business tier (ADF BC) are deployed as a web module in the same java container on OAS
- *EJB Mode:* the web tier (JSF/ADF Faces) is deployed as a web module in one java container and the business tier (ADF BC) is deployed in an EJB container (which may be in a different instance or server).

The EJB Mode allows for distributed processing and separation of security, i.e. you could have a firewall between web and EJB tiers. The author thinks security is typically the biggest driver for EJB Mode, although it may also be suitable for a more complex environment, where you might have other EJBs and be integrating with other systems via web services. The price you pay for this separation is in performance (the 'remoting' overhead). For the moderately-sized ADF application local mode is fine and can provide scalability by adding more servers (fronted by a load balancer). Also bear in mind that most reasonably straightforward ADF implementations today are local mode deployments and so this mode has the benefit of 'safety in numbers'.

3 Practical Deployment Approaches

As an administrator you primarily have control over 3 key areas:

- data sources and schema,
- the Java environment,
- the security provider.

In the following sections we will address each of these areas, followed by how you actually receive and deploy software delivered by your development team.

3.1 Data Source Management

You have now hopefully decided that to use pooled data sources for your ADF application. ADF BC accesses two types of data: firstly that in your application schema; secondly, ADF BC-specific internal tables[3] ('Persistent Collections') storing AM state information[†]. It is best practice to have two separate data sources for each of these: the application data source only needs DML access (SELECT, INSERT, UPDATE, DELETE) to your application's schema, whereas the account used for the BC internal source needs privileges to CREATE TABLE, CREATE INDEX and CREATE SEQUENCE (assuming you don't re-create the tables beforehand).

Note that by default JDeveloper will only set use one database connection for both application and internal data access, so you should ask your developers to change the **`jbo.server.internal_connection`** parameter in **`bc4j.xcfg`** to use your separate data source for BC-specific internal tables.

You can call data sources whatever you like (within the JNDI naming rules), but it is advisable to set up a standard naming convention for data sources across all your applications. For example, **`jdbc/<appname>DS`** for a main application DML and **`jdbc/<appname>InternalDS`** for the internal one (sometimes known as **`<appname>CoreDS`**). These resources can be mapped to different data source names during deployment, but for simplicity it is a good idea to try and keep them the same.

When it comes to deploying across different environments you have plenty of flexibility, depending on how many servers (or VMs) you have and how much independence you need between environments. For example, here is a naming scheme that would be suitable for a moderately-sized system with Production, QA and System Test environments (where @ indicates the database service name or SID) that would work whether you had 2-5 application servers:

Environment	**jdbc/appDS**	**jdbc/appInternalDS**	**Context root**
Production	**`app_user@PROD1`**	**`app_adf@PROD1`**	**`/app`**
QA / Pre-Production	**`app_user@TESTQA`**	**`app_adf@TESTQA`**	**`/app_qa`**
System Test 1A	**`app_user_a@TEST1`**	**`app_adf@TEST1`**	**`/app_test01a`**
System Test 1B	**`app_user_b@TEST1`**	**`app_adf@TEST1`**	**`/app_test01b`**
System Test 2A	**`app_user_a@TEST2`**	**`app_adf@TEST2`**	**`/app_test02a`**

It is perfectly safe for multiple environments to share the same ADF BC internal schema (as the primary key on the **`PS_TXN`** table is generated from a database sequence), though clearly the closer you get to the production environment the greater level of isolation you will need to assure valid testing.

3.2 How Many Database Connections Do I Need?

Within your pooled data sources you should decide the maximum number of connections you are going to allow to the database (rather than leaving them as unlimited) – this is a good practice, since if your ADF application is getting a little out

[†] 'Row Cache Spill-Over' allows excessive rows retrieved by a View Object to be written to the database for caching purposes, rather than using java heap space. These days the use of this feature is frowned upon for performance reasons and it is disabled by default. If you see tables in the BC internal schema called **`PS_AppModuleName(_number)`** and **`PS_AppModuleName_ky`** then talk to your developers to see if their usage is really necessary.

of control, you want to be able to indentify and manage the problem closer to its source rather than impacting other systems that may also be using the database tier.

As with traditional applications, when approaching configuration for a new ADF application you will have to make some initial assumptions based on knowledge of the target user community. You can then refine the configuration using data gathered during simulated load testing and, finally, by monitoring behavior in the production environment.

The actual values will depend on a number of factors: how demanding your service-level agreement is for application response times, the number of active users at peak times and hardware resources/software licenses available to you.

You will need to monitor the system using the usual application server and database techniques. In particular, you can look at the state of the ADF BC layer using various methods:

- using the ADF BC pages of Oracle Enterprise Manager Grid Control[4],
- by incorporating Steve Meunch's **`DumpPoolStatisticsServlet`**[5] into your environment,
- historically, Oracle's BC4J Admin Tool.

There are no hard and fast rules for ADF BC tuning as much depends on how your application is built and the nature of your users; as a starting point you should refer to the ADF Developer's Guide (chapter 29)[6] and various notes on OTN[7].

3.3 ADF BC Internal Schema Housekeeping

When using the database as a BC persistence store, ADF normally cleans up its state table (**`PS_TXN`**) as it is running. However over time you will still get some rows building up (due to certain failures and some timeouts) so you need a method to clear it out periodically.

When you create a new ADF BC internal database user you should also run the **`bc4jcleanup.sql`** script from the **`BC4J`** directory of the JDeveloper installation. This creates a PL/SQL package called **`BC4J_CLEANUP`**. It is a good idea to set up a regular **`DBMS_JOB`** (or use **`DBMS_SCHEDULER`**) to run **`BC4J_CLEANUP.SESSION_STATE(24*60)`** – this will clear out any **`PS_TXN`** data over a day old (assuming the system is configured so that a user will not manage to keep a session active for over a day!).

Note that if you are using a file system as the BC persistence store then, instead, you should periodically run an operating system script to delete the old XML files based on the file creation time.

3.4 Java Environment and ADF BC Configuration

Setting up the Java environment, such as heap size and garbage collection, for ADF is no different to other Java applications and so is not discussed further here.

You can however set ADF BC properties at various levels. For example, it is a good idea to use java system properties for data source related parameters since this makes them more easily visible to the administrator. However beware that precedence rules mean that any values set by the developer have highest priority so ADF BC changes need to be coordinated with the development team. Normally you would agree changes in **`bc4j.xcfg`** and a developer checks them into source control.

3.5 Security

Your developers will most likely have built the ADF application around a number of security roles, using 'container manager' security. This means that the java application server itself is responsible for authenticating the user and telling the application what security roles the user belongs to. These roles are capable of being nested and can be as fine or coarse are your organization requires. A user will belong to one[‡] or more roles at login time.

Java Authentication and Authorization Services (JAAS) is the API that defines how security works, and JAZN is Oracle's JAAS implementation, as used within OC4J.

[‡] strictly speaking a user could have no roles, but then would probably not have any useful access either

The configuration of JAZN is external to the application itself and can be different between environments. It can also be set at either the container level, for all deployed applications, or on an application by application basis. This gives the administrator flexibility to separate or share user credentials in different environments as required, minimizing management effort.

For example a developer may have their own small set of test users on their PC, stored in a small XML file. When deployed, however, the application will require a more substantial user store. For ADF applications this usually means:

1) users and user-role tables in a database (e.g. using the DBTableOraDataSource login module[8]): this suits the smaller, standalone ADF system and has the advantages of being low cost and very easy to set up.
2) Oracle Internet Directory (OID): more suitable for applications that need a higher degree of integration with existing systems and more sophisticated user provisioning facilities.
3) Oracle Internet Directory with Oracle Single Sign-On Server (SSO) which works particularly well when your users access ADF applications alongside existing Forms applications and/or Oracle Portal. Also in a Microsoft Windows security environment, if OID is integrated with Active Directory, Oracle SSO allows 'zero sign on,' i.e. no separate logon step is required once the user has logged into the Windows domain.

These are the typical approaches, but there are plenty of other options available for more extensive security requirements, including those where your ADF application has web service interfaces. For example, you may also have a requirement to control users at the database, as well as application-level, and can choose to implement proxy users, virtual private databases, etc.

The principal advantage here is that once you have defined the groups within your application, the actual security enforcement mechanism can be changed at deployment time. This can give the administrator a high degree of isolation and the flexibility to change to different security providers later.

3.6 Software Packaging and Release Procedure

As an administrator what you want is one, or a small number, of deliverables from your development team that can be easily deployed across all environments. As we saw earlier, there is some BC-specific configuration which may need amending, but mostly the only differences between environments are the data sources and context root, both of which can be set by the Administrator during deployment.

When we talk about application installation you will hear various 'deployment'-related terms:

- *Deployment Descriptors* are configuration files used by the application server – there are J(2)EE standard files (**`application.xml`**, **`data-sources.xml`**) plus vendor-specific ones (in our case **`orion-application.xml`** and **`jazn-data.xml`**). OAS will need all of these files for your application and they may either be supplied by the development team in the EAR file or by the administrator during deployment.
- *Deployment Profile* is a JDeveloper term for a file that tells it how to deploy your application (whether to create an EAR file, the context root, dependencies, etc). This file has a `.deploy` extension (except for the `.bcdeploy` which is an additional file for ADF BC deployments). You developers need at least one profile for your Application Server but they may have more for alternative environments (e.g. standalone OC4J).
- *Deployment Plan* is a J(2)EE term for a file that tells an application server how to deploy an application or web module. On OC4J this file typically has a `.dat` extension. Deployment Plans (part of JSR-88) have greatly simplified application deployment for administrators and are well supported within the Application Server Control console.

There are several ways to deploy a java application to OAS including Enterprise Manager, OC4J's **`admin_client.jar`**, Apache Ant, and by a direct connection from JDeveloper.

Assuming that we are deploying in Local Mode, it is simplest to have a WAR (web application archive) file from your developers and then set up the appropriate deployment descriptors. It also prevents you accidentally loading security data and data sources that may have been supplied by your developers in an EAR (enterprise archive) file. You can then save a Deployment Plan which you can use to reload the WAR (or EAR) file again later. However for more complicated systems it will be more likely that the developers will deliver an EAR file containing multiple web modules, EJBs etc.

It is best for the administrator to use Enterprise Manager (Application Server Control or Grid Control) to deploy WAR or EAR files. This has several benefits: it is easy to use, allows automatic deployment across clusters and provides a central point of management. A well-run development environment (using, say, Subversion and Ant) will provide consistent builds which, when coupled with deployment plans, can make application release a very reliable process.

3.7 ADF Libraries and Dependencies

All ADF applications are of course dependent on the ADF libraries – these support the runtime environment and are installed on the application server.

Whilst Oracle Application Server comes ready installed with the ADF libraries current at its time of release, it is a good idea to get used to installing ADF libraries separately using the ADF Installer[9]. This is a small java program that installs, updates or removes all the ADF java libraries using the correct directories for your type of application server. If you set up multiple OC4J instances on your application servers you will need to do this anyway, but you can expect ADF updates in line with JDeveloper releases which may be more frequent than those for the Oracle Application Server itself.

ADF Faces also has its own requirements on various Sun and Apache libraries (as will the ADF Faces Rich Client in 11g) so administrator should familiarize themselves which the required versions.

Your development team may also be using other libraries – perhaps log4j, Apache Axis, etc. They may choose to bundle them as part of the main application EAR file, though it is probably preferable for them to specify their requirements (e.g. in the Release Note) and ask the administrator to install the required JAR files instead.

Finally, at some point as an administrator you will probably have to contend with multiple versions of libraries (ADF and others) to support different applications on the same application server. The class loader allows you to finely control this and is well worth studying.

4 Conclusion

We have seen that ADF fits comfortably within the standard J(2)EE environment allowing Administrators to install applications easily, whilst still offering the flexibility to enforce security to enterprise standards. Deploying and managing ADF applications does not need detailed expertise in ADF or JDeveloper, however a good understanding of application modules and Oracle data sources will help the Administrator maintain optimum system performance even under heavy load.

For Further Information

Thank you for your interest in this work – if you have any questions you can contact the author at SimonH@veriton.co.uk or via his new Fusion Middleware Administration blog at http://simonhaslam.co.uk.

References

[1] Oracle® Application Development Framework Developer's Guide For Forms/4GL Developers
10g Release 3 (10.1.3.0), Ch. 28
http://download.oracle.com/docs/html/B25947_01/bcampool.htm#sm0299

[2] Dive into Oracle ADF, Steve Meunch
http://radio.weblogs.com/0118231/

[3] Overview of Temporary Tables Created By BC4J: Meunch (April 2002)
http://www.oracle.com/technology/products/jdev/htdocs/bc4j/bc4j_temp_tables.html

[4] Oracle® Enterprise Manager Oracle Application Server Metric Reference Manual 10*g* Release 2 (10.2), Ch. 1
http://download.oracle.com/docs/cd/B16240_01/doc/em.102/b25987/oracle_bc4j.htm#BDCDJEGF

[5] Dump Application Module Pooling Statistics Servlet [10.1.3]
http://otn.oracle.com/products/jdev/tips/muench/dumppoolstatistics/DumpPoolStatisticsExample.zip

[6] Oracle® Application Development Framework Developer's Guide For Forms/4GL Developers
10g Release 3 (10.1.3.0), Ch. 29
http://download.oracle.com/docs/html/B25947_01/bcampool.htm#sm0299

[7] How to Performance Tune an ADF BC Application: Andersen & Gantman (September 2004)
http://www.oracle.com/technology/products/jdev/howtos/10g/adfbc_perf_and_tuning.html

[8] Oracle® Containers for J2EE Security Guide 10g (10.1.3.1.0), Ch. 9
http://download.oracle.com/docs/cd/B31017_01/web.1013/b28957/loginmod.htm#BABCDDAI

[9] The ADF library installer can be downloaded from:
http://www.oracle.com/technology/software/products/jdev/htdocs/adfinstaller.html