Sun Java™ System

Application Server 8
Migrating and Redeploying Server Applications Guide

March 2004

Contents

About This Guide

This guide describes how Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications are migrated from the Sun ONE Application Server, J2EE Reference Implementation (RI) Application Server, Sun Java System Application Server 7, and several third party application servers to the Sun Java System Application Server 8 product line.

This preface contains information about the following topics:

- Who Should Use This Guide
- Using the Documentation
- How This Guide Is Organized
- Documentation Conventions
- Contacting Sun

Who Should Use This Guide

The intended audience for this guide is the system administrator, network administrator, application server administrator, and web developer who has an interest in migration issues.

This guide assumes you are familiar with the following topics:

- HTML
- Application Servers
- Client/Server programming model
- Internet and World Wide Web

- Windows 2000 and/or Solaris™ operating systems

- Java programming

- Java APIs as defined in specifications for EJBs, Java Server Pages (JSP)

- Java Database Connectivity (JDBC)

- Structured database query languages such as SQL

- Relational database concepts

- Software development processes, including debugging and source code control

Using the Documentation

The Sun Java System Application Server Platform Edition manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML).

The following table lists tasks and concepts described in the Sun Java System Application Server manuals.

Table 1   Sun Java System Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of supported hardware, operating system, JDK, and JDBC/RDBMS.	*Release Notes*
Installing the Sun Java System Application Server software and its components, such as sample applications, the Administration Console, and the high-availability components. Instructions for implementing a basic high-availability configuration are included.	*Installation Guide*
Creating and implementing Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications intended to run on the Sun Java System Application Server that follow the open Java standards model for J2EE components and APIs. Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules. A comprehensive Sun Java System Application Server glossary is included.	*Developer's Guide*
Using J2EE 1.4 platform technologies and APIs to develop J2EE applications and deploying the applications on the Sun Java System Application Server.	*J2EE 1.4 Tutorial*

Table 1   Sun Java System Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Information and instructions on the configuration, management, and deployment of the Sun Java System Application Server subsystems and components, from both the Administration Console and the command-line interface. Topics include cluster management, the high-availability database, load balancing, and session persistence. A comprehensive Sun Java System Application Server glossary is included.	*Administration Guide*
Editing the Sun Java System Application Server configuration file, `domain.xml`.	*Reference*
Migrating your applications to the new Sun Java System Application Server programming model, specifically from Sun ONE Application Server 6.x/7.x and from Netscape Application Server 4.0. Includes a sample migration.	*Migrating and Redeploying Server Applications Guide*
Information on solving Sun Java System Application Server problems.	*Troubleshooting Guide*
Utility commands available with the Sun Java System Application Server; written in manpage style.	*Utility Reference Manual*
Using the Sun™ Java System Message Queue 3.5 software.	The Sun Java System Message Queue documentation at: http://docs.sun.com/db?p= prod/s1.s1msgqu

How This Guide Is Organized

This guide is organized as follows:

- *Chapter 1, "Understanding Migration,"* describes the process of migrating applications developed for earlier versions of application servers from Sun Microsystems, Inc. and other third party vendors.

- *Chapter 2, "Migrating from EJB 1.1 to EJB 2.0,"* describes modifications that must be made within the source code of components when migrating from EJB 1.1 to EJB 2.0.

- *Chapter 3, "Migrating from Sun ONE Application Server 6.x/7.x to Sun Java System Application Server Platform Edition 8,"* describes the considerations and strategies that are needed when moving J2EE applications from Sun ONE Application Server 6.x and Sun Java System Application Server 7 to the Sun Java System Application Server Platform Edition 8.

- *Chapter 4, "Migrating a Sample Application - an Overview,"* describes the process for migrating the main components of a J2EE application to Sun Java System Application Server Platform Edition 8. Uses iBank sample application as the example to demonstrate the steps.

- *Chapter 5, "Migration Tools and Resources,"* describes migration tools that help automate the migration process from earlier versions of Sun ONE Application Server, Sun Java System Application Server 7, Netscape Application Server (Kiva), NetDynamics Application Server, and competitive application servers to Sun Java System Application Server Platform Edition 8.

- *Appendix A, "iBank Application Specification,"* provides an in-depth description of the iBank sample application.

Documentation Conventions

This section describes the types of conventions used throughout this guide:

- General Conventions
- Conventions Referring to Directories

General Conventions

The following general conventions are used in this guide:

- **File and directory paths** are given in UNIX® format (with forward slashes separating directory names). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.

- **URLs** are given in the format:

  http://*server.domain/path/file*.html

  In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename. Italic items in URLs are placeholders.

- **Font conventions** include:

  ❍ The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.

  ❍ *Italic* type is used for code variables.

  ❍ *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

- ○ **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.

- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in "Conventions Referring to Directories" on page 12.

  By default, the location of *install_dir* on **most** platforms is:

  - ○ Solaris and Linux file-based installations, non-root user:

    user's home directory/SUNWappserver

  - ○ Solaris and Linux file-based installations, root user:

    /opt/SUNWappserver

  - ○ Windows, all installations:

    system drive:\Sun\AppServer

  For the platforms listed above, *default_config_dir* is identical to *install_dir*. See "Conventions Referring to Directories" on page 12 for exceptions and additional information.

- **Domain root directories** are indicated by *domain_dir* in this document, which by default is an abbreviation for the following:

  install_dir/domains/*domain_dir*

  However, for package-based installations, the directory containing all the domains can be changed from *install_dir*/domains/ to another directory during installation. In configuration files, you may see *domain_dir* represented as follows:

  ${com.sun.aas.instanceRoot}

- **UNIX-specific descriptions** throughout this manual apply to the Linux operating system as well, except where Linux is specifically mentioned.

Conventions Referring to Directories

By default, when using the Solaris package-based or Linux RPM-based installation, the application server files are spread across several root directories. This guide uses the following document conventions to correspond to the various default installation directories provided:

- *install_dir* refers to `/opt/SUNWappserver`, which is the default location for the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.

- *default_config_dir* refers to `/var/opt/SUNWappserver/domains`, which is the default location for any domains that are created.

Contacting Sun

You might want to contact Sun Microsystems in order to:

- Give Us Feedback
- Obtain Training
- Contact Product Support

Give Us Feedback

If you have general feedback on the product or documentation, please send this to `appserver-feedback@sun.com`.

Obtain Training

Application Server training courses are available at:

`http://training.sun.com/US/catalog/enterprise/web_application.html/`

Visit this site often for new course availability on the Sun Java System Application Server.

Contact Product Support

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:

  http://www.sun.com/supporttraining/

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation

- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem. Here are some of the commonly used commands:

  - **Solaris:** pkginfo, showrev

  - **Linux:** rpm

  - **All:** asadmin version --verbose

- Detailed steps on the methods you have used to reproduce the problem

- Any error logs or core dumps

- Configuration files such as:

  - *domain_dir*/config/domain.xml

  - a web application's web.xml file,
    when a web application is involved in the problem

- For an application, whether the problem appears when it is running in a cluster or standalone

Contacting Sun

Understanding Migration

This chapter addresses the following topics:

- J2EE Component Standards

- J2EE Application Components and Migration

- Migration and Redeployment

J2EE Component Standards

Sun Java System Application Server Platform Edition 8 (hereafter called Application Server) is a J2EE v1.4-compliant server based on the component standards developed by the Java community for Java Servlets (servlets) , Java Server Pages (JSPs), and Enterprise JavaBeans (EJBs).

By contrast, Sun Java System Application Server 7 is a J2EE v1.3-compliant server and Sun ONE Application Server 6.x is a J2EE v1.2-compliant server. Between the three J2EE versions, there are considerable differences with the J2EE application component APIs.

The following table characterizes the differences between the component APIs used with the J2EE v1.4-compliant Sun Java System Application Server Platform Edition 8, the J2EE v1.3-compliant Sun ONE Application Server 7, and the J2EE v1.2-compliant Sun ONE Application Server 6.x.

Table 1-1   Application Server Version Comparison of APIs for J2EE Components

Component API	Sun ONE Application Server 6.x	Sun Java System Application Server 7	Sun Java System Application Server Platform Edition 8
JDK	1.2.2	1.4	1.4
Servlet	2.2	2.3	2.4

Table 1-1   Application Server Version Comparison of APIs for J2EE Components

JSP	1.1	1.2	2.0
JDBC	2.0	2.0	2.1, 3.0
EJB	1.1	2.0	2.0
JNDI	1.2	1.2	1.2.1
JMS	1.0	1.1	1.1
JTA	1.0	1.01	1.01

In addition, the two products support a number of technologies connected with XML standards and Web Services which are not part of the J2EE specification.

J2EE Application Components and Migration

J2EE simplifies development of enterprise applications by basing them on standardized, modular components, providing a complete set of services to those components, and handling many details of application behavior automatically, without complex programming. J2EE v1.4 architecture includes several component APIs. Prominent J2EE components include:

- Servlets

- Java Server Pages (JSPs)

- EJBs, including Message Driven Beans (MDBs)

- Java Database Connectivity (JDBC)

- Java Transaction Service (JTS)

- Java Naming and Directory Interface (JNDI)

- Java Message Service (JMS)

J2EE components are packaged separately and bundled into a J2EE application for deployment. Each component, its related files such as GIF and HTML files or server-side utility classes, and a deployment descriptor are assembled into a module and added to the J2EE application. A J2EE application is composed of one or more enterprise bean(s), Web, or application client component modules. The final enterprise solution can use one J2EE application or be made up of two or more J2EE applications, depending on design requirements.

A J2EE application and each of its modules has its own deployment descriptor. A deployment descriptor is an XML document with an .xml extension that describes a component's deployment settings. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. Because deployment descriptor information is declarative, it can be changed without modifying the bean source code. At run time, the J2EE server reads the deployment descriptor and acts upon the component accordingly.

A J2EE application with all of its modules is delivered in an Enterprise Archive (EAR) file. An EAR file is a standard Java Archive (JAR) file with an .ear extension. The EAR file contains EJB JAR files, application client JAR files and/or Web Archive (WAR) files. The characteristics of these files are as follows:

- Each EJB JAR file contains a deployment descriptor, the enterprise bean files, and related files

- Each application client JAR file contains a deployment descriptor, the class files for the application client, and related files

- Each WAR file contains a deployment descriptor, the Web component files, and related resources

Using modules and EAR files makes it possible to assemble a number of different J2EE applications using some of the same components. No extra coding is needed; it is just a matter of assembling various J2EE modules into J2EE EAR files.

The migration process is concerned with moving J2EE application components, modules, and files.

For more information on migrating various J2EE components please refer to Chapter 3.

For more background information on J2EE, see the following references:

- J2EE tutorial - http://java.sun.com/j2ee/tutorial/

- J2EE overview - http://java.sun.com/j2ee/overview.html

- J2EE topics - http://java.sun.com/j2ee

Migration and Redeployment

This section describes the need to migrate J2EE applications and the particular files that will need to be migrated. Following successful migration, a J2EE application can be redeployed to the Application Server. Redeployment is also described in this section.

The following topics are addressed:

Why is Migration Necessary?

Although J2EE specifications broadly cover requirements for applications, it is nonetheless an evolving standard. It either does not cover some aspects of applications or leaves implementation details as the responsibility of application providers.

These product implementation-dependent aspects manifest as differences in the way application servers are configured and also in the deployment of J2EE components on application servers. The array of available configuration and deployment tools for use with any particular application server product also contribute to the product implementation differences.

The evolutionary nature of the specifications itself presents challenges to application providers. Each of the component APIs in turn are separately evolving. This leads to a varying degree of conformance by products. In particular, an emerging product, such as the Application Server, has to contend with differences in J2EE application components, modules, and files deployed on other established application server platforms. Such differences require mappings between earlier implementation details of the J2EE standard such as file naming conventions, messaging syntax, and so forth.

Moreover, product providers usually bundle additional features and services with their products. These features are available as custom JSP tags or proprietary Java API libraries. Unfortunately, using these proprietary features will render these applications non-portable.

What Needs to be Migrated

For migration purposes, the J2EE application consists of the following file categories:

- Deployment descriptors (XML files)
- JSP source files that contain Proprietary APIs
- Java source files that contain Proprietary APIs

Deployment descriptors (XML files)

Deployment is accomplished by specifying deployment descriptors (DDs) for standalone enterprise beans (EJB JAR files), front-end Web components (WAR files) and enterprise applications (EAR files). Deployment descriptors are used to resolve all external dependencies of the J2EE components/applications. The J2EE specification for DDs is common across all application server products. However, the specification leaves several deployment aspects of components pertaining to an application dependent on product-implementation.

JSP source files

J2EE specifies how to extend JSP by adding extra custom tags. Product vendors include some custom JSP extensions in their products, simplifying some tasks for developers. However, usage of these proprietary custom tags results in non-portability of JSP files. Additionally, JSP can invoke methods defined in other Java source files as well. The JSPs containing proprietary APIs needs to be rewritten before they can be migrated.

Java source files

The Java source files can be EJBs, servlets, or other helper classes. The EJBs and servlets can invoke standard J2EE services directly. They can also invoke methods defined in helper classes. Java source files are used to encode the business layer of applications, such as EJBs.Vendors bundle several services and proprietary Java API with their products. The usage of proprietary Java APIs is a major source of non-portability in applications. Since J2EE is an evolving standard, different products may support different versions of J2EE component APIs. This is another aspect that migration will address.

What is Redeployment?

Redeployment refers to deploying a previously deployed application from an earlier version of Sun ONE Application Server 6.x, Sun Java System Application Server 7, or from applications that were previously deployed, but migrated, from third party application server platforms.

The act of redeploying an application typically refers to using the standard deployment actions outlined in the *Sun Java System Application Server Platform Edition 8 Administration Guide.* However, when migration activities are performed with automated tools, such as the Migration Tool for Sun Java System Application Server Platform Edition 8 (for J2EE applications) or the Sun ONE Migration Toolbox (for NetDynamics and Netscape Application Servers), there might be post-migration or pre-deployment tasks that are needed (and defined) prior to deploying the migrated application.

See Migration Tools and Resources for more information about migration tools that are available.

Migrating from EJB 1.1 to EJB 2.0

Although the EJB 1.1 specification will continue to be supported in Sun Java System Application Server Platform Edition 8, the use of the EJB 2.0 architecture is recommended to leverage its enhanced capabilities.

To migrate EJB 1.1 to EJB 2.0 a number of modifications will be required, including within the source code of components.

Essentially, the required modifications relate to the differences between EJB 1.1 and EJB 2.0, all of which are described in the following topics.

- EJB Query Language
- Local Interfaces
- EJB 2.0 Container-Managed Persistence (CMP)
- Defining Persistent Fields
- Defining Entity Bean Relationships
- Message-Driven Beans

EJB Query Language

The EJB 1.1 specification left the manner and language for forming and expressing queries for finder methods to each individual application server. While many application server vendors let developers form queries using SQL, others use their own proprietary language specific to their particular application server product. This mixture of query implementations causes inconsistencies between application servers.

The EJB 2.0 specification introduces a query language called *EJB Query Language*, or *EJB QL* to correct many of these inconsistencies and shortcomings. EJB QL is based on SQL92. It defines query methods, in the form of both finder and select methods, specifically for entity beans with container-managed persistence. EJB QL's principal advantage over SQL is its portability across EJB containers and its ability to navigate entity bean relationships.

Local Interfaces

In the EJB 1.1 architecture, session and entity beans have one type of interface, a remote interface, through which they can be accessed by clients and other application components. The remote interface is designed such that a bean instance has remote capabilities; the bean inherits from RMI and can interact with distributed clients across the network.

With EJB 2.0, session beans and entity beans can expose their methods to clients through two types of interfaces: a *remote interface* and a *local interface*. The 2.0 remote interface is identical to the remote interface used in the 1.1 architecture, whereby, the bean inherits from RMI, exposes its methods across the network tier, and has the same capability to interact with distributed clients.

However, the local interfaces for session and entity beans provide support for lightweight access from EJBs that are local clients; that is, clients co-located in the same EJB container. The EJB 2.0 specification further requires that EJBs that use local interfaces be within the same application. That is, the deployment descriptors for an application's EJBs using local interfaces must be contained within one `ejb-jar` file.

The local interface is a standard Java interface. It does not inherit from RMI. An enterprise bean uses the local interface to expose its methods to other beans that reside within the same container. By using a local interface, a bean may be more tightly coupled with its clients and may be directly accessed without the overhead of a remote method call.

In addition, local interfaces permit values to be passed between beans with pass by reference semantics. Because you are now passing a reference to an object, rather than the object itself, this reduces the overhead incurred when passing objects with large amounts of data, resulting in a performance gain.

Setting up a session or entity bean to use a local interface rather than a remote interface is simple. The local interface through which the bean's methods are exposed to clients extends `EJBLocalObject` rather than `EJBObject`. Similarly, the bean's home interface extends `EJBLocalHome` rather than `EJBHome`. The implementation class extends the same `EntityBean` or `SessionBean` interface.

In EJB 2.0, a bean that is destined to be remote extends `EJBObject` in its remote interface and `EJBHome` in its home interface, just as it did in EJB 1.1.

EJB 2.0 Container-Managed Persistence (CMP)

The EJB 2.0 specification expanded CMP to allow multiple entity beans to have relationships among themselves. This is referred to as *Container-Managed Relationships* (CMR). The container manages the relationships and the referential integrity of the relationships.

The EJB 1.1 specification presented a more limited CMP model. The EJB 1.1 architecture limited CMP to data access that is independent of the database or resource manager type. It allowed you to expose only an entity bean's instance state through its remote interface; there is no means to expose bean relationships. The EJB 1.1 version of CMP depends on mapping the instance variables of an entity bean class to the data items representing their state in the database or resource manager. The CMP instance fields are specified in the deployment descriptor, and when the bean is deployed, the deployer uses tools to generate code that implements the mapping of the instance fields to the data items.

You must also change the way you code the bean's implementation class. According to the EJB 2.0 specification, the implementation class for an entity bean that uses CMP is now defined as an abstract class.

Defining Persistent Fields

The EJB 2.0 specification lets you designate an entity bean's instance variables as CMP fields or CMR fields. You define these fields in the deployment descriptor. CMP fields are marked with the element `cmp-field`, while container-managed relationship fields are marked with the element `cmr-field`.

In the implementation class, note that you do not declare the CMP and CMR fields as public variables. Instead, you define `get` and `set` methods in the entity bean to retrieve and set the values of these CMP and CMR fields. In this sense, beans using the 2.0 CMP follow the JavaBeans model: instead of accessing instance variables directly, clients use the entity bean's `get` and `set` methods to retrieve and set these instance variables. Keep in mind that the `get` and `set` methods only pertain to variables that have been designated as CMP or CMR fields.

Defining Entity Bean Relationships

As noted previously, the EJB 1.1 architecture does not support CMRs between entity beans. The EJB 2.0 architecture does support both one-to-one and one-to-many CMRs. Relationships are expressed using CMR fields, and these fields are marked as such in the deployment descriptor. You set up the CMR fields in the deployment descriptor using the appropriate deployment tool for your application server.

Similar to CMP fields, the bean does not declare the CMR fields as instance variables. Instead, the bean provides `get` and `set` methods for these fields.

Message-Driven Beans

Message-driven beans are another new feature introduced by the EJB 2.0 architecture. Message-driven beans are transaction-aware components that process asynchronous messages delivered through the Java Message Service (JMS). The JMS API is an integral part of the J2EE 1.3 and J2EE 1.4 platform.

Asynchronous messaging allows applications to communicate by exchanging messages so that senders are independent of receivers. The sender sends its message and does not have to wait for the receiver to receive or process that message. This differs from synchronous communication, which requires the component that is invoking a method on another component to wait or block until the processing completes and control returns to the caller component.

Migrating EJB Client Applications

This section includes the following topics:

- Declaring EJBs in the JNDI Context
- Recap on Using EJB JNDI References

Declaring EJBs in the JNDI Context

In Sun Java System Application Server Platform Edition 8, EJBs are systematically mapped to the JNDI sub-context "*ejb/*". If we attribute the JNDI name "*Account*" to an EJB, then Sun Java System Application Server Platform Edition 8 will automatically create the reference "*ejb/Account*" in the global JNDI context. The clients of this EJB will therefore have to look up "*ejb/Account*" to retrieve the corresponding home interface.

Let us examine the code for a servlet method deployed in Sun ONE Application Server 6.x.

The servlet presented here calls on a stateful session bean, BankTeller, mapped to the root of the JNDI context. The method whose code we are considering is responsible for retrieving the home interface of the EJB, so as to enable a BankTeller object to be instantiated and a remote interface for this object to be retrieved, in order to make business method calls to this component.

```
/**
   * Look up the BankTellerHome interface using JNDI.
   */
private BankTellerHome lookupBankTellerHome(Context ctx)
    throws NamingException
{
    try
    {
      Object home = (BankTellerHome) ctx.lookup("ejb/BankTeller");
      return (BankTellerHome) PortableRemoteObject.narrow(home,
BankTellerHome.class);
    }
    catch (NamingException ne)
    {
      log("lookupBankTellerHome: unable to lookup BankTellerHome" +
         "with JNDI name 'BankTeller': " + ne.getMessage() );
      throw ne;
    }
}
```

As the code already uses `ejb/BankTeller` as an argument for the lookup, there is no need for modifying the code to be deployed on Sun Java System Application Server Platform Edition 8.

Recap on Using EJB JNDI References

This section summarizes the considerations when using EJB JNDI references. Where noted, the consideration details are specific to a particular source application server platform.

Placing EJB References in the JNDI Context

It is only necessary to modify the name of the EJB references in the JNDI context mentioned above (moving these references from the JNDI context root to the sub-context "*ejb/*") when the EJBs are mapped to the root of the JNDI context in the existing WebLogic application.

If these EJBs are already mapped to the JNDI sub-context `ejb/` in the existing application, no modification is required.

However, when configuring the JNDI names of EJBs in the deployment descriptor within the Forté for Java IDE, it is important to avoid including the prefix `ejb/` in the JNDI name of an EJB. Remember that these EJB references are *automatically* placed in the JNDI `ejb/` sub-context with Sun Java System Application Server Platform Edition 8. So, if an EJB is given to the JNDI name "*BankTeller*" in its deployment descriptor, the reference to this EJB will be "translated" by Sun Java System Application Server Platform Edition 8 into `ejb/BankTeller`, and this is the JNDI name that client components of this EJB must use when carrying out a lookup.

Global JNDI context versus local JNDI context

Using the global JNDI context to obtain EJB references is a perfectly valid, feasible approach with Sun Java System Application Server Platform Edition 8. Nonetheless, it is preferable to stay as close as possible to the J2EE specification, and retrieve EJB references through the local JNDI context of EJB client applications. When using the local JNDI context, you must first declare EJB resource references in the deployment descriptor of the client part (`web.xml` for a Web application, `ejb-jar.xml` for an EJB component).

Migrating CMP Entity EJBs

This section describes the steps to migrate your application components from the EJB 1.1 architecture to the EJB 2.0 architecture.

In order to migrate a CMP 1.1 bean to CMP 2.0, we first need to verify if a particular bean can be migrated. The steps to perform this verification are as follows.

1. From the `ejb-jar.xml` file, go to the `<cmp-fields>` names and check if the optional tag `<prim-key-field>` is present in the `ejb-jar.xml file` and has an indicated value. If it does, go to next step.

   Look for the `<prim-key-class>` field name in the `ejb-jar.xml`, get the class name and get the `public instance variables` declared in the class. Now see if the signature (name and case) of these variables matches with the `<cmp-field>` names above. Segregate the ones that are found. In these segregated fields, check if some of them start with an upper case letter. If any of them do, then migration cannot be performed.

2. Look into the bean class source code and obtain the java types of all the `<cmp-field>` variables.

3. Change all the `<cmp-field>` names to lowercase and construct accessors from them. For example if the original field name is `Name` and its java type is `String`, the accessor method signature will be:

   ```
   Public void setName(String name)
   Public String getName()
   ```

4. Compare these accessor method signatures with the method signatures in the bean class. If there is an exact match found, migration is not possible.

5. Get the custom finder methods signatures and their corresponding SQLs. Check if there is a 'Join' or 'Outer join' or an 'OrderBy' in the SQL, if yes, we cannot migrate, as EJB QL does not support 'joins', 'Outer join' and 'OrderBy'.

6. Any CMP 1.1 finder, which used `java.util.Enumeration`, should now use `java.util.Collection`. Change your code to reflect this. CMP2.0 finders cannot return java.util.Enumeration.

"Migrating the Bean Class," explains how to perform the actual migration process.

Migrating the Bean Class

This section describes the steps required to migrate the bean class to Sun Java System Application Server Platform Edition 8.

1. Prepend the bean class declaration with the keyword *abstract*. For example if the bean class declaration was:

```
Public class CabinBean implements EntityBean // before
modification

abstract Public class CabinBean implements EntityBean // after
modification
```

2. Prefix the accessors with the keyword abstract.

3. Insert all the accessors after modification into the source(.java) file of the bean class at class level.

4. Comment out all the `cmp` fields in the source file of the bean class.

5. Construct protected instance variable declarations from the `cmp-field` names in lowercase and insert them at the class level.

6. Read up all the `ejbCreate()` method bodies (there could be more than one `ejbCreate`). Look for the pattern '`<cmp-field>=`*some value or local variable*', and replace it with the expression 'abstract mutator method name (*same value or local variable*)'. For example, if the `ejbCreate` body (before migration) is like this:

```
public MyPK ejbCreate(int id, String name)
    {
this.id = 10*id;
Name = name;//1
return null;
    }
```

The changed method body (after migration) should be:

```
public MyPK ejbCreate(int id, String name)
    {
setId(10*id);
setName(name);//1
return null;
    }
```

Note that the method signature of the abstract accessor in `//1` is as per the Camel Case convention mandated by the EJB 2.0 specification. Also, the keyword '*this*' may or may not be present in the original source, but it *must be removed* from the modified source file.

7. All the protected variables declared in the `ejbPostCreate()` methods in step 5 must be initialized. The protected variables will be equal in number with the `ejbCreate()` methods. This initialization will be done by inserting the initialization code in the following manner:

```
    protected String name;                      //from step 5
    protected int id;                           //from step 5
        public void ejbPostCreate(int id, String name)
        {
name /*protected variable*/ = getName();    /*abstract accessor*/
//inserted in this step
id /*protected variable*/ = getId();        /*abstract accessor*/
//inserted in this step
            }
```

8. Inside the `ejbLoad` method, you must set the protected variables to the beans' database state. To do so, insert the following lines of code:

```
        public void ejbLoad()
            {
            name = getName();         //inserted in this step
            id = getId();             //inserted in this step
                ……..             //already present code
            }
```

9. Similarly, you will have to update the beans' state inside `ejbStore()` so that its database state gets updated. But remember, you are not allowed to update the setters that correspond to the primary key outside the `ejbCreate()`, so do not include them inside this method. Insert the following lines of code:

```
    public void ejbStore()
        {
        setName(name);        //inserted in this step
//  setId(id);                //Do not insert this if it is a
                                  part of the primary key
            ……………..        //already present code
        }
```

10. As a last change to the bean class source (`.java`) file, examine the whole code and replace all occurrences of any `<cmp-field>` variable name with the equivalent protected variable name (as declared in step 5).

   If you do not migrate the bean, at the minimum you need to insert the `<cmp-version>1.x</cmp-version>` tag inside the `ejb-jar.xml` file at the appropriate place, so that the unmigrated bean still works on Sun Java System Application Server Platform Edition 8.

Migration of ejb-jar.xml

To migrate the file `ejb-jar.xml` to Sun Java System Application Server Platform Edition 8, perform the following steps:

1. In the `ejb-jar.xml`, convert all `<cmp-fields>` to lowercase.

2. In the `ejb-jar.xml` file, insert the tag `<abstract-schema-name>` after the `<reentrant>` tag. The schema name will be the name of the bean as in the `< ejb-name>` tag, prefixed with "ias_".

3. Insert the following tags after the `<primkey-field>` tag:

    `<security-identity><use-caller-identity/></security-identity>`

4. Use the SQL's obtained above to construct the EJB QL from SQL.

5. Insert the `<query>` tag and all its nested child tags with all the required information in the `ejb-jar.xml`, just after the `<security-identity>` tag.

Custom Finder Methods

The custom finder methods are the `findBy...` methods (other than the default `findByPrimaryKey` method), which can be defined in the home interface of an entity bean. Since the EJB 1.1 specification does not stipulate a standard for defining the logic of these finder methods, EJB server vendors are free to choose their implementations. As a result, the procedures used to define the methods vary considerably between the different implementations chosen by vendors.

Sun ONE Application Server 6.x uses standard SQL to specify the finder logic.

Information concerning the definition of this finder method is stored in the enterprise bean's persistence descriptor (`Account-ias-cmp.xml`) as follows:

```
<bean-property>
  <property>
    <name>findOrderedAccountsForCustomerSQL</name>
    <type>java.lang.String</type>
    <value>
        SELECT BRANCH_CODE,ACC_NO FROM ACCOUNT where CUST_NO = ?
    </value>
    <delimiter>,</delimiter>
  </property>
</bean-property>
<bean-property>
  <property>
    <name>findOrderedAccountsForCustomerParms</name>
    <type>java.lang.Vector</type>
    <value>CustNo</value>
    <delimiter>,</delimiter>
  </property>
</bean-property>
```

Each `findXXX` finder method therefore has two corresponding entries in the deployment descriptor (SQL code for the query, and the associated parameters).

In Sun Java System Application Server Platform Edition 8 the custom finder method logic is also declarative, but is based on the EJB query language EJB QL.

The EJB-QL language cannot be used on its own. It has to be specified inside the file `ejb-jar.xml`, in the `<ejb-ql>` tag. This tag is inside the `<query>` tag, which defines a query (finder or select method) inside an EJB. The EJB container can transform each query into the implementation of the finder or select method. Here's an example of an `<ejb-ql>` tag:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
    <ejb-name>hotelEJB</ejb-name>
      ...
      <abstract-schema-name>TMBankSchemaName</abstract-schema-name>
      <cmp-field>...
      ...
      <query>
        <query-method>
          <method-name>findByCity</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
        <ejb-ql>
          <![CDATA[SELECT OBJECT(t) FROM TMBankSchemaName AS t
            WHERE t.city = ?1]]>
        </ejb-ql>
      </query>
    </entity>
  ...
  </enterprise-beans>
...
</ejb-jar>
```

Migrating CMP Entity EJBs

Migrating from Sun ONE Application Server 6.x/7.x to Sun Java System Application Server Platform Edition 8

This chapter describes the considerations and strategies that are needed when moving J2EE applications from Sun ONE Application Server 6.x and Sun Java System Application Server 7 to the Sun Java System Application Server Platform Edition 8 product line.

The sections that follow describe issues that arise while migrating the main components of a typical J2EE application from Sun ONE Application Server 6.x/7.x to Sun Java System Application Server Platform Edition 8.

The migration issues described in this section are based on an actual migration that was performed for a J2EE application called *iBank*, a simulated online banking service, from Sun ONE Application Server 6.x to Sun Java System Application Server Platform Edition 8. This application reflects all aspects that comprise a traditional J2EE application.

The following sensitive points of the J2EE specification covered by the iBank application include:

*   Servlets, especially with redirection to JSP pages (model-view-controller architecture)

*   JSP pages, especially with static and dynamic inclusion of pages

*   JSP custom tag libraries

*   Creation and management of HTTP sessions

*   Database access through the JDBC API

*   Enterprise JavaBeans: Stateful and Stateless session beans, CMP and BMP entity beans.

- Assembly and deployment in line with the standard packaging methods of the J2EE application

The iBank application is presented in detail in *Appendix A - iBank Application Specification*

This section also describes specific migration tasks at the component level.

The following topics are addressed:

- "Migrating Deployment Descriptors"

- Migrating J2EE Components

- Migrating Web Applications

- Migrating Enterprise EJB Modules

- Migrating Enterprise Applications

- Migrating Proprietary Extensions

- Migrating UIF

Migrating Deployment Descriptors

The following table summarizes the deployment descriptor migration mapping.

Source Deployment Descriptor	Target Deployment Descriptor
ejb-jar.xml – 1.1	ejb-jar.xml – 2.0
ias-ejb-jar.xml	sun-ejb-jar.xml
<bean-name>-ias-cmp.xml	sun-cmp-mappings.xml
web.xml	web.xml
ias-web.xml	sun-web.xml
application.xml	application.xml

The J2EE standard deployment descriptors `ejb-jar.xml`, `web.xml` and `application.xml` are not modified significantly. However, the `ejb-jar.xml` deployment descriptor is modified to make it compliant with EJB 2.0 specification in order to make the application deployable onSun Java System Application Server Platform Edition 8.

Majority of the information required for creating `sun-ejb-jar.xml` and `sun-web.xml` comes from `ias-ejb-jar.xml` and `ias-web.xml` respectively. However, there is some information that is required and extracted from the home interface (java file) of the CMP entity bean, in case the `sun-ejb-jar.xml` being migrated declares one. This is required to build the `<query-filter>` construct inside the `sun-ejb-jar.xml`, which requires information from inside the home interface of that CMP entity bean. If this source file is not present during the migration time, the `<query-filter>` construct will get created, but with lots of missing information (which will manifest itself in the form of "REPLACE ME" phrases in the migrated `sun-ejb-jar.xml`).

Additionally, if the `ias-ejb-jar.xml` contains a `<message-driven>` element, then information from inside this element is picked up and used to fill up information inside both `ejb-jar.xml` and `sun-ejb-jar.xml`. Also, inside the `<message-driven>` element of `ias-ejb-jar.xml`, there is an element `<destination-name>`, which holds the JNDI name of the topic or queue to which the MDB should listen to. In Sun ONE Application Server 6.5, the naming convention for this jndi name is "`cn=<SOME_NAME>`". Since a JMS Topic or Queue with this name is not deployable on Sun Java System Application Server Platform Edition 8, changes this to "`<SOME_NAME>`", and insert this information in the `sun-ejb-jar.xml`. This change must be reflected for all valid input files, namely, all `.java`, `.jsp` and `.xml` files. Hence, this change of JNDI name is affected globally across the application, and in case of non availability of some source files that contain reference to this jndi-name, you need to make the change manually in them so that the application becomes deployable.

Migrating J2EE Components

The following migration processes are described in this section:

- Migrating JDBC Code

- Migrating Java Server Pages and JSP Custom Tag Libraries

- Migrating Servlets

- Obtaining a Data Source from the JNDI Context

- EJB Migration

- EJB Changes Specific to Sun Java System Application Server Platform Edition 8

Migrating JDBC Code

With the JDBC API, there are two methods of database access:

- Establishing Connections Through the DriverManager Interface

  (JDBC 1.0 API), by loading a specific driver and providing a connection URL. This method is used by other Application Servers, such as IBM's WebSphere 4.0

- Using JDBC 2.0 Data Sources

  The `DataSource` interface (JDBC 2.0 API) can be used via a configurable connection pool. According to J2EE 1.2, a data source is accessed through the JNDI naming service

---

NOTE   Sun Java System Application Server Platform Edition 8 does not support the Native Type 2 JDBC drivers bundled with Sun ONE Application Server 6.x. You must manually migrate code that uses the Type 2 drivers to use third party JDBC drivers.

Establishing Connections Through the DriverManager Interface

Although this means of accessing a database is not recommended, as it is obsolete and is not very effective, there may be some applications that still use this approach.

In this case, the access code will be similar to the following:

```
public static final String driver =
"oracle.jdbc.driver.OracleDriver";
public static final String url =
"jdbc:oracle:thin:tmb_user/tmb_user@iben:1521:tmbank";
Class.forName(driver).newInstance();
Properties props = new Properties();
props.setProperty("user", "tmb_user");
props.setProperty("password", "tmb_user");
Connection conn = DriverManager.getConnection(url, props);
```

This code can be fully ported from Sun ONE Application Server 6.x to Sun Java System Application Server Platform Edition 8, as long as the Application Server is able to locate the classes needed to load the right JDBC driver. In order to make the required classes accessible to the application deployed in the Application Server, you should place the archive (JAR or ZIP) for the driver implementation in the `/lib` directory of the Application Server installation directory.

Modify the *CLASSPATH* by setting the path for the driver through the Admin Console GUI. Click the server instance "server1" and then click the tab "JVM Settings" from the right pane. Now click the option Path Settings and add the path in the classpath suffix text entry box. Once you make the changes, click "Save" and then apply the new settings. Restart the server to modify the configuration file, `server.xml`.

Using JDBC 2.0 Data Sources

Using JDBC 2.0 data sources to access a database provides performance advantages such as transparent connection pooling, enhances productivity by simplifying code and implementation, and provides code portability.

Using a data source in an application requires an initial configuration phase followed by a registration of the data source in the JNDI naming context of the application server. Once the data source is registered, the application will easily be able to obtain a connection to the database by retrieving the corresponding *DataSource* object from the JNDI context. The actions are described in the following topics:

- Configuring a Data Source

- Looking Up the Data Source Via JNDI To Obtain a Connection

Configuring a Data Source

In Sun ONE Application Server 6.0 data sources and their corresponding JDBC drivers are configured from the server's graphic administration console. Connection pools are managed automatically by the application server, and the administration tool can be used to configure their properties. With integrated type 2 JDBC drivers, the connection pooling properties are defined on a per-driver basis, and common to all data sources using a given driver.

On the other hand, for third-party JDBC drivers, connection pool properties are defined on a per-data source basis. Third-party JDBC drivers can be configured either from the administration tool, or from a separate utility (`db_setup.sh` in Sun Solaris, and `jdbcsetup` in Windows NT/2000). Moreover, the command line utility `iasdeploy` can be used to configure a data source from an XML file describing its properties. These utilities are all located in the `/bin/` sub-directory of the Application Server installation root directory.

In Application Server, data sources can be configured from the server's graphic administration console or through the command line utility *asadmin*. The command line utility asadmin can be invoked by executing the asadmin script in Solaris, available in Application Server installation's bin directory. At the asadmin prompt, use the following commands to create connection pool and JNDI resource.

The syntax for calling the asadmin utility to create a connection pool is as follows:

```
asadmin>create-jdbc-connection-pool -u username -w password -H
hostname -p adminport [-s] --datasourceclassname classname
[--steadypoolsize=8] [--maxpoolsize=32] [--maxwait=60000]
[--poolresize=2] [--idletimeout=300]
[--isconnectvalidatereq=false] [--validationmethod=auto-commit]
[--validationtable tablename] [--failconnection=false]
[--description text] [--property (name=value)[:name=value]*]
connectionpoolid
```

For example:

```
asadmin>create-jdbc-connection-pool -u admin -w password -H cl1
-p 4848 --datasourceclassname
oracle.jdbc.pool.OracleConnectionPoolDataSource --property
(user-name=ibank_user):(password=ibank_user) oraclepool
```

Here JDBC connection pool 'oraclepool' for oracle database is created using database schema having the username 'ibank_user' and password 'ibank_user'.

The syntax to create a JDBC resource is as follows:

```
asadmin>create-jdbc-resource -u username -w password -H hostname
-p adminport [-s] --connectionpoolid id [--enabled=true]
[--description text] [--property (name=value)[:name=value]*]
jndiname
```

For example:

```
asadmin>create-jdbc-resource -u admin -w password -H cl1 -p 4848
--connectionpoolid oraclepool jdbc/IBANK
```

Here a JDBC resourcewith the JNDI name jdbc/IBANK is created for the connection pool created above.

Here is the procedure to follow when registering a data source in Application Server through graphical interface.

1. Register the data source classname

   a. Place the archive (JAR or ZIP) for the data source class implementation in the /lib directory of the Application Server installation directory.

b. Modify the CLASSPATH by setting the path for the driver through the Admin Console GUI. Click at the server instance "server1" and then click at tab "JVM Settings", now click at path settings and add the path at the classpath suffix column. Once you make the changes save it and then apply these new settings. Restart the server, which would modify the configuration file, `server.xml`.

2. Register the data source

In Application Server, data sources and their corresponding JDBC drivers are configured from the server's graphic administration interface.

The left pane is a tree view of all items you can configure in the Application Server. Click on the item Connection pool at the left pane, the right pane would display the page associated with it where the relevant entries can be made.

Similarly now click at the item Data source, right pane would show the entries required for data source setup.

The Application Server-specific deployment descriptor `sun-web.xml` has to be modified accordingly.

For example if a new data source is configured for the iBank application, the `sun-web.xml` file would contain the following entries.

```
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN' 'Http://localhost:8000/sun-web-app_2_3.dtd'>
<sun-web-app>
   <resource-ref>
      <res-ref-name>jdbc/iBank</res-ref-name>
      <jndi-name>jdbc/iBank</jndi-name>
      <default-resource-principal>
          <name>ibank_user</name>
          <password>ibank_user</password>
      </default-resource-principal>
   </resource-ref>
</sun-web-app>
```

Looking Up the Data Source Via JNDI To Obtain a Connection

To obtain a connection from a data source, the process is as follows:

1. Obtain the initial JNDI context.

    To guarantee portability between different environments, the code used to retrieve an InitialContext object (in a servlet, in a JSP page, or an EJB), should be simply, as follows:

    ```
    InitialContext ctx = new InitialContext();
    ```

2. Use a JNDI lookup to obtain a data source reference.

   To obtain a reference to a data source bound to the JNDI context, look up the data source's JNDI name from the initial context object. The object retrieved in this way should then be cast as a DataSource type object:

   ```
   ds = (DataSource)ctx.lookup(JndiDataSourceName);
   ```

3. Use the data source reference to obtain the connection.

   This operation is very simple, and requires the following line of code:

   ```
   conn = ds.getConnection();
   ```

Sun ONE Application Server 6.x and Application Server both follow the above technique for obtaining a connection form data source. So to summarize migration does not require any modification to be made to the code.

Migrating Java Server Pages and JSP Custom Tag Libraries

Sun ONE Application Server 6.x complies with the JSP 1.1 specification and Application Server complies with the JSP 2.0 specification.

JSP 2.0 specification contains many new features as well as corrections and clarifications of areas that were not quite right in JSP 1.1 specification.

These changes are basically enhancements and are not required to be made, while migrating JSP pages from JSP 1.1 to 2.0.

The implementation of JSP custom tag libraries in Sun ONE Application Server 6.x complies with the J2EE specification. Consequently, migration of JSP custom tag libraries to the Sun Java System Application Server Platform Edition 8 does not pose any particular problem, nor require any modifications to be made.

Migrating Servlets

Sun ONE Application Server 6.x supports the Servlet 2.2 API. Sun Java System Application Server Platform Edition 8 supports the Servlet 2.4 API.

Servlet API 2.4 actually leaves the core of servlets relatively untouched; most changes are concerned with adding new features outside the core.

The most significant features are:

- Servlets now require JDK 1.2 or later

- A filter mechanism has been created

- Application lifecycle events have been added

- New internationalization support has been added

- New error and security attributes have been added

- The HttpUtils class has been deprecated

- Several DTD behaviors have been expanded and clarified

These changes are basically enhancements and are not required to be made while migrating servlets from Servlet API 2.2 to 2.4.

However, if the servlets in the application use JNDI to access resources of the J2EE application (such as data sources, EJBs, and so forth), some modifications may be needed in the source files or in the deployment descriptor.

These modifications are explained in detail in the following sections:

- Obtaining a Data Source from the JNDI Context

- Declaring EJBs in the JNDI Context

One last scenario may mean modifications are required in the servlet code, naming conflicts may occur with Sun ONE Application Server if a JSP page has the same name as an existing Java class. In this case, the conflict should be resolved by modifying the name of the JSP page in question, which may then mean editing the code of the servlets that call this JSP page. This issue is resolved in Application Server as it uses new class loader hierarchy as compared to Sun ONE Application Server 6.x. In this new scheme, for a given application, one class loader loads all EJB modules and another class loader loads web module. As these two loaders do not talk with each other, there would be no naming conflict.

Obtaining a Data Source from the JNDI Context

To obtain a reference to a data source bound to the JNDI context, look up the data source's JNDI name from the initial context object. The object retrieved in this way should then be *cast* as a DataSource type object:

```
ds = (DataSource)ctx.lookup(JndiDataSourceName);
```

For detailed information, refer to section "Migrating JDBC Code" in the previous pages.

Declaring EJBs in the JNDI Context

Please refer to section Declaring EJBs in the JNDI Context from "Migrating from EJB 1.1 to EJB 2.0" on page 21."

EJB Migration

As mentioned in Understanding Migration, while Sun ONE Application Server 6.x supports the EJB 1.1 specification, Application Server also supports the EJB 2.0 specification. The EJB 2.0 specification introduces the following new features and functions to the architecture:

- Message Driven Beans (MDBs)

- Improvements in Container-Managed Persistence (CMP)

- Container-managed relationships for entity beans with CMP

- Local interfaces

- EJB Query Language (EJB QL)

Although the EJB 1.1 specification will continue to be supported in the Application Server, the use of the EJB 2.0 architecture is recommended to leverage its enhanced capabilities.

For detailed information on migrating from EJB 1.1 to EJB 2.0, please refer to Chapter 2, "Migrating from EJB 1.1 to EJB 2.0."

EJB Changes Specific to Sun Java System Application Server Platform Edition 8

Migrating EJBs from Sun ONE Application Server 6.x to the Application Server is done without making any changes to the EJB code. However, the following DTD changes are required.

Session Beans

- The `<!DOCTYPE>` definition should be modified to point to the latest DTDs with J2EE standard DDs, such as `ejb-jar.xml`.

- Replace `ias-ejb-jar.xml` file with the modified version of this file, named `sun-ejb-jar.xml`, created manually according to the DDs. For more details, see the URL
  http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2_0-0.dtd

- In the `sun-ejb-jar.xml` file, the JNDI name for all the EJBs should prepend 'ejb/' in all the JNDI names. This is required as, in Sun ONE Application Server 6.5, the JNDI name of the EJB could only be ejb/*<ejb-name>* where *<ejb-name>* is the name of the EJB as declared inside the `ejb-jar.xml` file.

  In the Application Server, a new tag has been introduced in the `sun-ejb-jar.xml`, where the JNDI name of the EJB can be declared.

NOTE	To avoid changing JNDI names throughout the application, we recommend that the JNDI name of the EJB should be declared as ejb/*<ejb-name>* inside the *<jndi-name>* tag.

Entity Beans

- The `<!DOCTYPE>` definition should be modified to point to the latest DTDs with J2EE standard DDs, such as `ejb-jar.xml`.

- Insert `<cmp-version>` tag with the value 1.1 for all CMPs in the `ejb-jar.xml` file.

- Replace all the *<ejb-name>*`-ias-cmp.xml` files with the manually created `sun-cmp-mappings.xml` file. For more information, see URL
  http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mapping_1_0.dtd

- Generate `dbschema` by using the `capture-schema` utility in the Application Server installation's bin directory and place it above `META-INF` folder for Entity beans.

- `ias-ejb-jar.xml` should be replaced with its new version, named `sun-ejb.jar.xml`, in Application Server.

- In Sun ONE Application Server 6.5, the finders sql was directly embedded inside the *<ejb-name>*`-ias-cmp.xml`. In Application Server, this has changed such that, now mathematical expressions are used to declare the `<query-filter>` for the various finder methods.

Message Driven Beans

Application Server provides seamless Message Driven Support through the tight integration of Sun Java System Message Queue with the Application Server, providing a native, built-in JMS Service.

This installation provides Application Server with a JMS messaging system that supports any number of Application Server instances. Each server instance, by default, has an associated built-in JMS Service that supports all JMS clients running in the instance.

Both container-managed and bean-managed transactions as defined in the Enterprise JavaBeans Specification, v2.0 are supported.

Message Driven Bean support in iPlanet Application Server was restricted to developers, and used many of the older proprietary APIs. Messaging services were provided by iPlanet Message Queue for Java 2.0. An LDAP directory was also required under iPlanet Application Server to configure the `Queue Connection Factory` object.

The `QueueConnectionFactory`, and other particulars required to configure Message Driven Beans in Application Server should be specified in the `ejb-jar.xml` file.

For more information on the changes to deployment descriptors, see "Migrating Deployment Descriptors." For information on Message Driven Bean implementation in Sun Java System Application Server Platform Edition 8, see *Sun Java System Application Server Platform Edition 8, Developer's Guide to Enterprise Java Bean Technology*.

Migrating Web Applications

Sun ONE Application Server 6.x support servlets (Servlet API 2.2), and JSPs (JSP 1.1). Sun Java System Application Server Platform Edition 8 supports Servlet API 2.4 and JSP 2.0.

Within these environments it is essential to group the different components of an application (servlets, JSP and HTML pages and other resources) together within an archive file (J2EE-standard Web application module) before you can deploy it on the application server.

According to the J2EE specification, a Web application is an archive file (WAR file) with the following structure:

- a root directory containing the HTML pages, JSP, images and other "static" resources of the application.

- a `META-INF/` directory containing the archive manifest file (MANIFEST.MF) containing the version information for the SDK used and, optionally, a list of the files contained in the archive.

- a `WEB-INF/` directory containing the application deployment descriptor (`web.xml` file) and all the Java classes and libraries used by the application, organized as follows:

  - A `classes/` sub-directory containing the tree-structure of the compiled classes of the application (servlets, auxiliary classes), organized into packages

  - A `lib/` directory containing any Java libraries (JAR files) used by the application

Migrating Web Application Modules

Migrating applications from Sun ONE Application Server 6.x to Sun Java System Application Server Platform Edition 8 does not require any changes in the Java/JSP code. The following changes are, however, still required.

- `web.xml`

  The Application Server adheres to J2EE 1.4 standards, according to which, the `web.xml` file inside a WAR file should comply with the revised DTD available at URL `http://java.sun.com/dtd/web-app_2_3.dtd`. This DTD fortunately, is a superset of the previous versions' DTD, hence only the `<! DOCTYPE` definition needs to be changed inside the `web.xml` file, which is to be migrated. The modified `<! DOCTYPE` declaration should look like:

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

- `ias-web.xml`

  In Sun Java System Application Server Platform Edition 8, the name of this file is changed to `sun-web.xml`.

  This XML file is required to declare the Application Server-specific properties/resources that will be required by the Web application.

  See "Potential Servlets and JSP Migration Problems," for information about important inclusions to this file.

If the `ias-web.xml` of the Sun ONE Application Server 6.5 application is present and does declare Sun ONE Application Server 6.5 specific properties, then this file needs to be migrated to Application Server standards. The DTD file name has to be changed to `sun-web.xml`. For more details, see URL http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2_3-0.dtd.

Once the `web.xml` and `ias-web.xml` files are migrated in the above-mentioned fashion, the Web application (WAR file) can be deployed from the Application Server's deploytool GUI interfaceor from the command line utility `asadmin`, where the deployment command should mention the type of application as web.

Invoke the `asadmin` command line utility by running `asadmin.bat` file or the `asadmin.sh` script in the Application Server's `bin` directory.

The command at the `asadmin` prompt would be:

```
asadmin> deploy -u username -w password -H hostname -p adminport
--type web [--contextroot contextroot] [--force=true] [--name
component-name] [--upload=true] filepath
```

Potential Servlets and JSP Migration Problems

The actual migration of the components of a Servlet / JSP application from Sun ONE Application Server 6.x to Application Server will not require any modifications to be made to the component code.

If the Web application is using a server resource, a DataSource for example, then the Application Server requires that this resource to be declared inside the `web.xml` file and, correspondingly, inside the `sun-web.xml` file. To declare a DataSource called `jdbc/iBank`, the `<resource-ref>` tag in the `web.xml` file would be as follows:

```
<resource-ref>
   <res-ref-name>jdbc/iBank</res-ref-name>
   <res-type>javax.sql.XADataSource</res-type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Corresponding declaration inside the `sun-web.xml` file will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE FIX ME: need confirmation on the DTD to be used for
this file
```

```
<sun-web-app>
   <resource-ref>
      <res-ref-name>jdbc/iBank</res-ref-name>
      <jndi-name>jdbc/iBank</jndi-name>
   </resource-ref>
</sun-web-app>
```

Migrating Enterprise EJB Modules

Sun ONE Application Server 6.x supports EJB 1.1; the Application Server supports EJB 2.0. Thereby, both can support:

*   Stateful or stateless session beans

*   Entity beans with bean-managed persistence (BMP), or container-managed persistence (CMP)

EJB 2.0, however, introduces a new type of enterprise bean, called a message-driven bean(MDB) in addition to the session and entity beans.

J2EE 1.4 specification dictates that the different components of an EJB must be grouped together in a JAR file with the following structure:

*   `META-INF/` directory with an XML deployment descriptor named `ejb-jar.xml`

*   The `.class` files corresponding to the home interface, remote interface, the implementation class, and the auxiliary classes of the bean with their package

Sun ONE application servers observe this archive structure. However, the EJB 1.1 specification leaves each EJB container vendor to implement certain aspects as they see fit:

*   Database persistence of CMP EJBs (particularly the configuration of mapping between the bean's CMP fields and columns in a database table).

*   Implementation of the custom finder method logic for CMP beans.

As we might expect, Sun ONE Application Server 6.x andApplication Server diverge on certain points, which means that when migrating an application certain aspects require particular attention. Some XML files have to be modified:

*   The `<!DOCTYPE` definition should be modified to point to the latest DTD url in case of J2EE standard DDs, like `ejb-jar.xml`.

- Replace the `ias-ejb-jar.xml` file with the modified version of this file (for example, file `sun-ejb-jar.xml`, which is created manually according to the DTDs). For more information, see URL http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2_0-0.dtd.

- Replace all the <*ejb-name*>`-ias-cmp.xml` files with one `sun-cmp-mappings.xml` file, which is created manually. For more information, see URL http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mapping_1_0.dtd.

- Optionally, for CMP entity beans, use the capture-schema utility in the Application Server's `bin` directory to generate dbschema. Then place it above the `META-INF` directory for the entity beans.

Migrating Enterprise Applications

According to the J2EE specifications, an enterprise application is an EAR file, which must have the following structure:

- a `META-INF/` directory containing the XML deployment descriptor of the J2EE application called `application.xml`

- the JAR and WAR archive files for the EJB modules and Web module of the enterprise application, respectively

In the application deployment descriptor, we define the modules that make up the enterprise application, and the Web application's context root.

Sun ONE Application server 6.x and the Application Server support the J2EE model wherein applications are packaged in the form of an enterprise archive (EAR) file (extension `.ear`). The application is further subdivided into a collection of J2EE modules, packaged into Java archives (JAR files, which have a `.jar` file extension) for EJBs and Web archives (WAR files,which have a `.war` file extension) for servlets and JSPs.

It is essential to follow the steps listed here before deploying an enterprise application:

1. Package EJBs in one or more EJB modules.

2. Package the components of the Web application in a Web module.

3. Assemble the EJB modules and Web modules in an enterprise application module.

4. Define the name of the enterprise application's root context, which will determine the URL for accessing the application.

The Application Server uses a newer class loader hierarchy than Sun ONE Application Server 6.x does. In the new scheme, for a given application, one class loader loads all EJB modules and another class loader loads Web modules. These two are related in a parent child hierarchy where the JAR module class loader is the parent module of the WAR module class loader. All classes loaded by the JAR class loader are available/accessible to the WAR module but the reverse is not true. If a certain class is required by the JAR file as well as the WAR file, then the class file should be packaged inside the JAR module only. If this guideline is not followed it could lead to class conflicts.

Application Root Context and Access URL

There is one particular difference between Sun ONE Application Server 6.x and the Application Server, concerning the applications access URL (root context of the application's Web module. If `AppName` is the name of the root context of an application deployed on a server called `hostname`, the access URL for this application will differ depending on the application server used:

- With Sun ONE Application Server 6.x, which is always used jointly with a Web front-end, the access URL for the application will take the following form (assuming the Web server is configured on the standard HTTP port, 80):

    ```
    http://<hostname>/NASApp/AppName/
    ```

- With the Application Server, the URL will take the form:

    ```
    http://<hostname>:<portnumber>/AppName/
    ```

The TCP port used as default by Application Server is port 8080.

Although the difference in access URLs between Sun ONE Application Server 6.x and the Application Server may appear minor, it can be problematic when migrating applications that make use of absolute URL references. In such cases, it will be necessary to edit the code to update any absolute URL references so that they are no longer prefixed with the specific marker used by the Web Server plug-in for Sun ONE Application Server 6.x.

Migrating Proprietary Extensions

A number of classes proprietary to the Sun ONE Application Server 6.x environment may have been used in applications. Some of the proprietary Sun ONE packages used by Sun ONE Application Server 6.x are listed below:

- `com.iplanet.server.servlet.extension`

- `com.kivasoft.dlm`
- `com.iplanetiplanet.server.jdbc`
- `com.kivasoft.util`
- `com.netscape.server.servlet.extension`
- `com.kivasoft`
- `com.netscape.server`

These APIs are not supported in the Application Server. Applications using any classes belonging to the above package will have to be rewritten to use standard J2EE APIs. Applications using custom JSP tags and UIF framework also needs to be rewritten to use standard J2EE APIs.

For a sample migration walkthrough using the iBank application, see Migrating a Sample Application - an Overview.

Migrating UIF

The Application Server does not support the use of Unified Integration Framework (UIF) API for applications. Instead, it supports the use of J2EE Connector Architecture (JCA) for integrating the applications. However, the applications developed in Sun ONE Application Server 6.5 use the UIF. In order to deploy such applications to the Application Server, you need to migrate the UIF to J2EE Connector Architecture. This section discusses the prerequisites and steps to migrate the applications using UIF to Application Server.

Before migrating the applications, you need to make sure that the UIF is installed on Sun ONE Application Server 6.5. To check for the installation, you can follow any of the following two approaches:

Approach 1: Checking in the registry files

UIF is installed as a set of application server extensions. They are registered in the application server registry during the installation. Search for the following strings in the registry to check whether UIF is installed.

Extension Name Set:

- Extension DataObjectExt-cDataObject
- Extension RepositoryExt-cLDAPRepository
- Extension MetadataService-cMetadataService

- Extension RepoValidator-cRepoValidator

- Extension BSPRuntime-cBSPRuntime

- Extension BSPErrorLogExt-cErrorLogMgr

- Extension BSPUserMap-cBSPUserMap

The registry file on Solaris Operating Environment can be found at the following location:

AS_HOME/`AS/registry/reg.dat`

Approach 2: Checking for UIF binaries in installation directories

UIF installers copy specific binary files in to the application server installation. A successful find of these files below indicate that UIF is installed.

The location of the following files on Solaris and Windows is:

AS_HOME`/AS/APPS/bin`

List of files to be searched on Solaris:

- `libcBSPRlop.so`

- `libcBSPRuntime.so`

- `libcBSPUserMap.so`

- `libcDataObject.so`

- `libcErrorLogMgr.so`

- `libcLDAPRepository.so`

- `libcMetadataService.so`

- `libcRepoValidator.so`

- `libjx2cBSPRuntime.so`

- `libjx2cDataObject.so`

- `libjx2cLDAPRepository.so`

- `libjx2cMetadataService.so`

List of files to be searched on Windows:

- `cBSPRlop.dll`

- `cBSPRuntime.dll`

- `cBSPUserMap.dll`

- `cDataObject.dll`

- `ErrorLogMgr.dll`

- `cLDAPRepository.dll`

- `cMetadataService.dll`

- `cRepoValidator.dll`

- `jx2cBSPRuntime.dll`

- `jx2cDataObject.dll`

- `jx2cLDAPRepository.dll`

- `jx2cMetadataService.dll`

Before migrating the UIF to Application Server, make sure that the UIF API is being used in applications. To verify its usage:

- Check for the usage of `netscape.bsp` package name in the Java sources

- Check for the usage of `access_cBSPRuntime.getcBSPRuntime` method in the sources. You must call this method to acquire the UIF runtime.

Contact `appserver-migration@sun.com` for information about UIF migration to the Application Server.

Migrating Rich Clients

This section describes the steps for migrating RMI/IIOP and ACC clients developed in Planet Application Server 6.x to the Application Server.

Authenticating a Client in Sun One Application Server 6.x

iPlanet Application Server provides a client-side callback mechanism that enables applications to collect authentication data from the user such as the username and the password.The authentication data collected by the iPlanet CORBA infrastructure is propagated to the Application Server via IIOP.

If ORBIX 2000 is the ORB used for RMI/IIOP, portable interceptors implement security by providing hooks, or interception points, which define stages within the request and reply sequence.

Authenticating a Client in Sun Java System Application Server Platform Edition 8

The authentication is done based on JAAS (Java Authorization and Authentication System API) and the client can that implement a `CallBackHandler`. If a client does not provide a `CallbackHandler`, then the default `CallbackHandler`, called the `LoginModule`, will be used by the ACC for obtaining the authentication data.

For detailed instructions on using JAAS for authentication, see the *Sun Java System Application Server Platform Edition 8 Developer's Guide to Clients.*

Using ACC in Sun ONE Application Server 6.x and Sun Java System Application Server Platform Edition 8

In Sun ONE Application Server 6.x, no separate appclient script is provided. You are required to place the `iasacc.jar` file in the classpath instead of the `iascleint.jar` file. The only benefit of using the ACC for packaging application clients in 6.x is that the JNDI names specified in the client application are indirectly mapped to the absolute JNDI names of the EJBs.

In case of Sun ONE Application Server 6.x applications, a stand-alone client would use the absolute name of the EJB in the JNDI lookup. That is, outside an ACC, the following approach would be used to lookup the JNDI:

```
initial.lookup("ejb/ejb-name");
initial.lookup("ejb/module-name/ejb-name");
```

If your application was developed using Sun ONE Application Server 6.5 SP3, you would have used the prefix "`java:comp/env/ejb/`" when performing lookups via absolute references.

```
initial.lookup("java:comp/env/ejb/ejb-name");
```

In Sun Java System Application Server Platform Edition 8, the JNDI lookup is done on the `jndi-name` of the EJB. The absolute name of the ejb must not be used. Also, the prefix, `java:comp/env/ejb` is not supported in Sun Java System Application Server Platform Edition 8. Replace the `iasclient.jar`, `iasacc.jar`, or `javax.jar` JAR files in the classpath with `appserv-ext.jar`.

If your application provides load balancing capabilities, in Sun Java System Application Server Platform Edition 8, load balancing capabilities are supported only in the form of S1ASCTXFactory as the context factory on the client side and then specifying the alternate hosts and ports in the cluster by setting the `com.sun.appserv.iiop.loadbalancingpolicy` system property as follows:

```
com.sun.appserv.iiop.loadbalancingpolicy=roundrobin,host1:port1,host2:port2,...,
```

This property provides you with a list of host:port combinations to round robin the ORBs. These host names may also map to multiple IP addresses. If you use this property along with `org.omg.CORBA.ORBInitialHost` and `org.omg.CORBA.ORBInitialPort` as system properties, the round robin algorithm will round robin across all the values provided. If, however, you provide a host name and port number in your code, in the environment object, that value will override any such system property settings.

The Provider URL to which the client is connected in Sun ONE Application Server 6.5 is the IIOP host and port of the CORBA Executive Engine (CXS Engine). In case of Sun Java System Application Server Platform Edition 8, the client needs to specify the IIOP listener Host and Port number of the instance. No separate CXS engine exists in Sun Java System Application Server Platform Edition 8.

The default IIOP port is 3700 in Sun Java System Application Server Platform Edition 8; the actual value of the IIOP Port can be found in the `server.xml` configuration file.

Migrating a Sample Application - an Overview

This chapter describes the process for migrating the main components of a typical J2EE application from Sun ONE Application Server 6.x to Sun Java System Application Server Platform Edition 8. This chapter highlights some of the problems posed during the migration of each type of component and suggests practical solutions to overcome such problems.

For this migration process, the J2EE application presented is called *iBank* and is based on the actual migration of the iBank application from Sun ONE Application Server 6.x to the Application Server. iBank simulates an online banking service and covers all of the aspects traditionally associated with a J2EE application.

The sensitive points of the J2EE specification covered by the iBank application are summarized below:

- Servlets, especially with redirection to JSP pages (model-view-controller architecture)

- JSP pages, especially with static and dynamic inclusion of pages

- JSP custom tag libraries

- Creation and management of HTTP sessions

- Database access through the JDBC API

- Enterprise JavaBeans: Stateful and Stateless session beans, CMP and BMP entity beans

- Assembly and deployment in line with the standard packaging methods of the J2EE application

The iBank application is presented in detail in Appendix A, "iBank Application Specification."

Preparing for Migrating the iBank Application

Before you start with the migration process learn about the differences in the deployment descriptors. For detailed information, see "Migrating Deployment Descriptors" on page 34.

Choosing the Target

You must choose the migration target server as Sun Java System Application Server Platform Edition **8**. After choosing the target server, install the server on your migration environment. For step-by-step instructions to install the software, see the *Sun Java System Application Server Platform Edition 8 Installation Guide.*

If you are using Migration Tool for Sun Java System Application Server Platform Edition **8** to migrate the components, you must install the tool. The Migration Tool can be downloaded from the following location:

http://java.sun.com/j2ee/tools/migration

For information on how to use the Migration Tool for Sun Java System Application Server Platform Edition **8**, see the Migration Tool online help. The iBank application is bundled with the tool.

Identifying the Components of the iBank Application

The iBank application has the following directory structure:

```
iBank
/docroot
/session
/entity
/misc
```

- `/docroot` contains HTML, JSP's and Image files in its root. It also contains the source files for servlets and EJBs in the sub-folder `WEB-INF\classes` following the package structure `com.sun.bank.*`. A war file is generated through the contents of this directory.

- `/session` contains the source code for the session beans following the package structure `com.sun.bank.ejb.session`. This directory forms the EJB module for the session beans.

- `/entity` contains the entity beans following the package structure `com.sun.bank.ejb.entity`. This directory would form the EJB module for entity beans.

- `/misc` contain the sql scripts for the database setup.

Manual Steps in the iBank Application Migration

Most of the migration is done by the Migration Tool. There are some aspects of migration that must be done manually. These steps are documented in the Migration Tool's user's guide and the documentation for the iBank sample application, which is included in the bundle.

Assembling Application for Deployment

Application Server primarily supports the J2EE model wherein applications are packaged in the form of an enterprise archive (EAR) file (extension .ear). The application is further subdivided into a collection of J2EE modules, packaged into Java archives (JAR, extension .jar) for EJBs and web archives (WAR, extension .war) for servlets and JSPs.

All the JSPs and Servlets should be packaged into WAR file, all EJBs into the JAR file and finally the WAR and the JAR file together with the deployment descriptors in to the EAR file. This EAR file is a deployable component.

Using the asadmin Utility to Deploy the iBank Application on Sun Java System Application Server Platform Edition 8

The last stage is to deploy the application on Sun Java System Application Server Platform Edition 8. The process for deploying an application is described below:

The Sun Java System Application Server Platform Edition 8 asadmin command includes a help section on deployment that is accessible from the Help menu.

The command line utility `asadmin` can be invoked by executing `asadmin.bat` file in Windows and `asadmin` file in Solaris Operating Environment that is stored in Application Server's installation's `bin` directory.

At `asadmin` prompt, the command for deployment would be:

```
asadmin> deploy -u username -w password -H hostname -p adminport
absolute_path_to_application
```

After you restart the Application Server, open a browser and go to the following URL to test the application:

```
http://<machine_name>:<port_number>/IBank
```

When prompted, enter one of the available user names and passwords. The main menu page of the iBank application should display.

Migration Tools and Resources

This chapter describes migration tools that help automate the migration process from earlier versions of Sun ONE Application Server, Sun Java System Application Server 7, Netscape Application Server (Kiva), NetDynamics Application Server, and competitive application servers to Sun Java System Application Server Platform Edition 8.

Migration Tool for Sun Java System Application Server Platform Edition 8

The Migration Tool for Sun Java System Application Server Platform Edition 8 (hereafter called Migration Tool) migrates J2EE applications from other server platforms to **Sun Java System Application Server Platform Edition 8**.

For **Sun Java System Application Server Platform Edition 8** the following source platforms are supported:

- Sun ONE Application Server 6.*x*, 7.0

- J2EE Reference Implementation Application Server (RI) 1.3, 1.4 Beta1

- Sun ONE Web Server 6.0

- WebLogic Application Server (WLS) 5.1, 6.0, 6.1

- WebSphere Application Server (WAS) 4.0

Migration Tool for Sun Java System Application Server Platform Edition 8 automates the migration of J2EE applications to Sun Java System Application Server Platform Edition 8, without much modification to the source code.

The key features of the tool are:

- Migration of application server-specific deployment descriptors

- Runtime support for selected custom JavaServer Pages (JSP) tags and proprietary APIs

- Conversion of selected configuration parameters with equivalent functionality in Application Server

- Automatic generation of Ant based scripts for building and deploying the migrated application to the target server, Application Server

- Generation of comprehensive migration reports after achieving migration

You can download the Migration Tool from the following location:

http://java.sun.com/j2ee/tools/migration/index.html

For detailed information on how to install and use the tool, consult its online help.

The Migration Tool specifications and migration process change from time to time, so the sample migration using the tool is not included in this guide. The migration process of a sample application is discussed in the documentation for this tool.

Redeploying Migrated Applications

Most of the applications that are migrated automatically through the use of the available migration tools will utilize the standard deployment tasks described in the *Sun Java System Application Server Platform Edition 8 Administration Guide*.

In some cases, the automatic migration will not be able to migrate particular methods or syntaxes from the source application. When this occurs in the case of the Migration Tool, you are notified of the steps that will be needed to complete the migration. Once you complete the post-migration manual steps, you will be able to deploy the application in the standard manner.

Sun ONE Migration Toolbox for Applogic and NetDynamics

Sun ONE Migration Toolbox (formerly called the iPlanet Migration Toolbox or iMT) is used to migrate applications built on NetDynamics or Kiva/NAS platforms to Sun ONE Application Server 6.x. In some cases, you might be able to use the Migration Tool for Sun Java System Application Server Platform Edition 8 to complete a migration to Sun Java System Application Server Platform Edition 8.

The Sun ONE Migration Toolbox is available upon request. Please contact appserver-migration@sun.com.

J2EE Application Verification Kit

The Java Application Verification Kit (AVK) for the Enterprise helps you build and test your applications for correct use of J2EE APIs and migrate to other J2EE compatible application servers using specific guidelines and rules.

You can download the Java Application Verification Kit (AVK) from the following location:

`http://java.sun.com/j2ee/verified/`

More Migration Information

This section provides references to additional migration documents.

Migrating from KIVA/NAS/NetDynamics Application Servers

For information about migrating your KIVA/NAS/NetDynamics applications to Sun ONE Application Server 6.0, see the *Sun ONE Application Server Migration Guide* at the following URL:

`http://docs.sun.com/db/doc/816-5780-10`

For information about migrating your KIVA/NAS/NetDynamics applications to Sun ONE Application Server 6.5, see the *Sun ONE Application Server 6.5 Migration Guide* at the following URL:

`http://docs.sun.com/db/doc/816-5793-11`

For information about migrating your KIVA/NAS/NetDynamics applications to Sun Java System Application Server 7, see *Sun Java System Application Server 7 Migrating and Redeploying Server Applications Guide* at the following URL:

`http://docs.sun.com/db/doc/817-2158-10`

More Migration Information

iBank Application Specification

The iBank application is used as the migration sample. This application simulates a basic online banking service with the following functionality:

- log on to the online banking service

- view/edit personal details and branch details

- summary view of accounts showing cleared balances

- facility to drill down by account to view individual transaction history

- money transfer service, allowing online transfer of funds between accounts

- compound interest earnings projection over a number of years for a given principal and annual yield rate

The application is designed after the MVC (Model-View-Controller) model where:

- EJBs are used to define the business and data model components of the application

- Java Server Pages handle the presentation logic and represent the View.

- Servlets play the role of Controllers and handle application logic, taking charge of calling the business logic components and accessing business data via EJBs (the Model), and dispatching processed data for display to Java Server Pages (the View).

Standard J2EE methods are used for assembling and deploying the application components. This includes the definition of deployment descriptors and assembling the application components within the archive files:

- a WAR archive file for the Web application including HTML pages, images, Servlets, JSPs and custom tag libraries, and ancillary server-side Java classes.

- EJB-JAR archive files for the assembling of one or more EJBs, including deployment descriptor, bean class and interfaces, stub and skeleton classes, and other helper classes as required.

- an EAR archive file for the packaging of the enterprise application module that includes the Web application module and the EJB modules used by the application.

The use of standard J2EE assembling methods will be useful in pointing out any differences between Sun ONE Application Server 6.x/7.x and Sun Java System Application Server Platform Edition 8, and any issues arising thereof.

Database Schema

The iBank database schema is derived from the following business rules:

- The iBank company has local branches in major cities.

- A Branch manages all customers within its regional area.

- A Customer has one or more accounts held at their regional branch.

- A customer Account is uniquely identified by the branch code and account number, and also holds the number of the customer to which it belongs. The current cleared balance available is also stored with the account.

- Accounts are of a particular Account Type that is used to distinguish between several kinds of accounts (checking account, savings account, etc.).

- Each Account Type stores a number of particulars that apply to all accounts of this type (regardless of branch or customer) such as interest rate and allowed overdraft limit.

- Every time a customer receives or pays money into/from one of their accounts, the transaction is recorded in a global transaction log, the Transaction History.

- The Transaction History stores details about individual transactions, such as the relevant branch code and account number, the date the transaction was posted (recorded), a code identifying the type of transaction and a complementary description of the particular transaction, and the amount for the transaction.

- Transaction types allow different types of transactions to be distinguished, such as cash deposit, credit card payment, fund transfer between accounts, and so on.

Figure A-1, the entity-relationship diagram shown below, illustrates these business rules.

Figure A-1   Database Schema
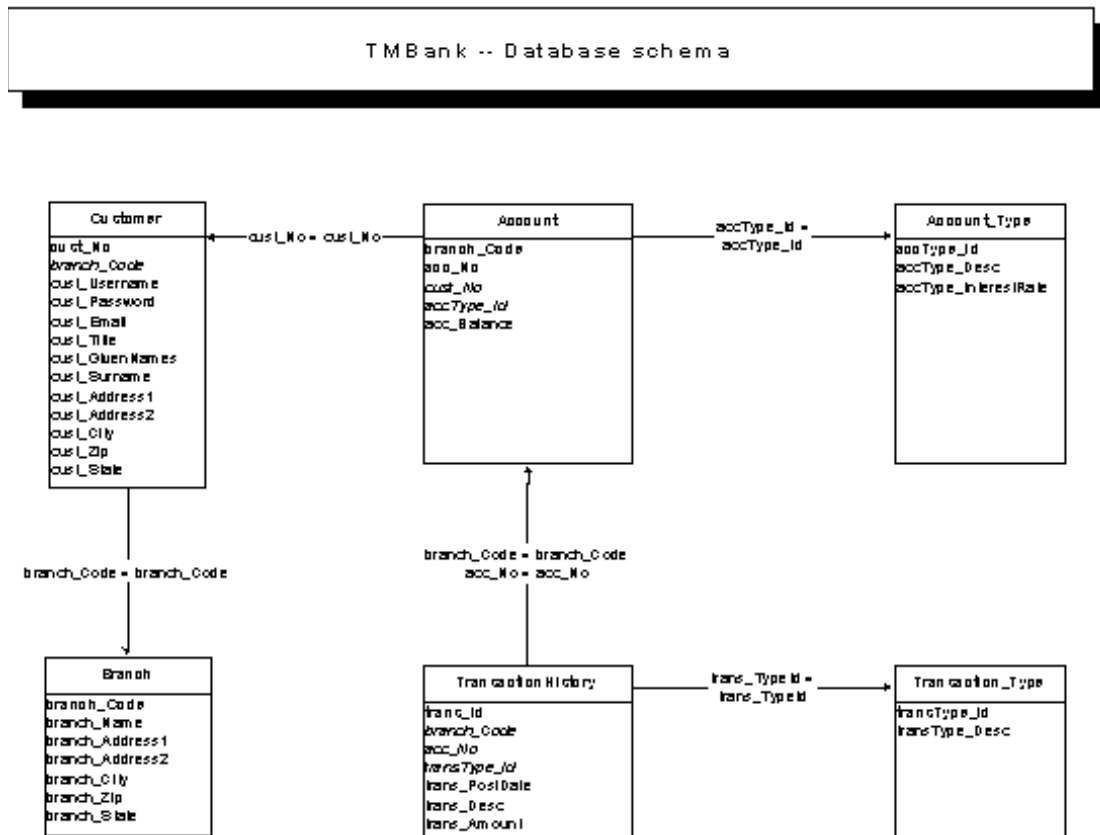
The database model translates as a series of table definitions below, where primary key columns are printed in bold type, while foreign key columns are shown in italics.

BRANCH			
BRANCH_CODE	CHAR(4)	NOT NULL	4-digit code identifying the branch
BRANCH_NAME	VARCHAR(40)	NOT NULL	Name of the branch

BRANCH_ADDRESS1	VARCHAR(60)	NOT NULL	Branch postal address, street address, 1st line
BRANCH_ADDRESS2	VARCHAR(60)		Branch postal address, street address, 2nd line
BRANCH_CITY	VARCHAR(30)	NOT NULL	Branch postal address, City
BRANCH_ZIP	VARCHAR(10)	NOT NULL	Branch postal address, Zip code
BRANCH_STATE	CHAR(2)	NOT NULL	Branch postal address, State abbreviation

CUSTOMER			
CUST_NO	INT	NOT NULL	iBank customer number (global)
BRANCH_CODE	CHAR(4)	NOT NULL	References this customer's branch
CUST_USERNAME	VARCHAR(16)	NOT NULL	Customer's login username
CUST_PASSWORD	VARCHAR(10)	NOT NULL	Customer's login password
CUST_EMAIL	VARCHAR(40)		Customer's e-mail address
CUST_TITLE	VARCHAR(3)	NOT NULL	Customer's courtesy title
CUST_GIVENNAMES	VARCHAR(40)	NOT NULL	Customer's given names
CUST_SURNAME	VARCHAR(40)	NOT NULL	Customer's family name
CUST_ADDRESS1	VARCHAR(60)	NOT NULL	Customer postal address, street address, 1st line
CUST_ADDRESS2	VARCHAR(60)		Customer postal address, street address, 2nd line
CUST_CITY	VARCHAR(30)	NOT NULL	Customer postal address, City
CUST_ZIP	VARCHAR(10)	NOT NULL	Customer postal address, Zip code
CUST_STATE	CHAR(2)	NOT NULL	Customer postal address, State abbreviation

ACCOUNT_TYPE			
ACCTYPE_ID	CHAR(3)	NOT NULL	3-letter account type code
ACCTYPE_DESC	VARCHAR(30)	NOT NULL	Account type description

ACCTYPE_INTERESTRATE	DECIMAL(4,2)	DEFAULT 0.0	Annual interest rate

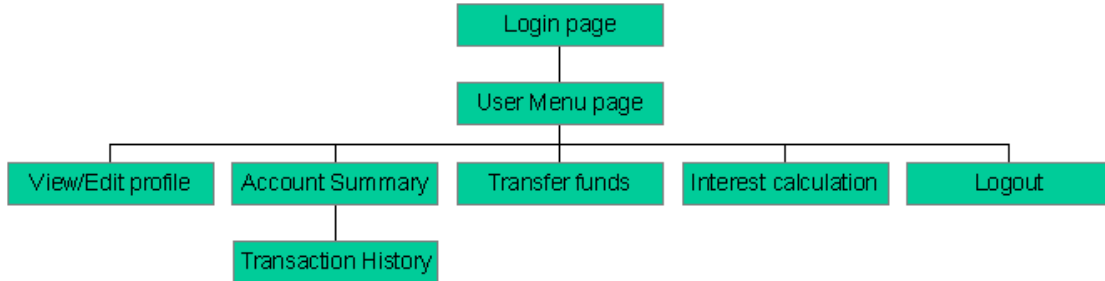| **ACCOUNT**			
BRANCH_CODE	CHAR(4)	NOT NULL	branch code (primary-key part 1)
ACC_NO	CHAR(8)	NOT NULL	account no. (primary-key part 2)
CUST_NO	INT	NOT NULL	Customer to whom accounts belongs
ACCTYPE_ID	CHAR(3)	NOT NULL	Account type, references ACCOUNT_TYPE
ACC_BALANCE	DECIMAL(10,2)	DEFAULT 0.0	Cleared balance available

TRANSACTION_TYPE			
TRANSTYPE_ID	CHAR(4)	NOT NULL	A 4-letter transaction type code
TRANSTYPE_DESC	VARCHAR(40)	NOT NULL	Human-readable description of code

TRANSACTION_HISTORY			
TRANS_ID	LONGINT	NOT NULL	Global transaction serial no
BRANCH_CODE	CHAR(4)	NOT NULL	key referencing ACCOUNT part 1
ACC_NO	CHAR(8)	NOT NULL	key referencing ACCOUNT part 2
TRANSTYPE_ID	CHAR(4)	NOT NULL	References TRANSACTION_TYPE
TRANS_POSTDATE	TIMESTAMP	NOT NULL	Date & time transaction was posted
TRANS_DESC	VARCHAR(40)		Additional details for the transaction
TRANS_AMOUNT	DECIMAL(10,2)	NOT NULL	Money amount for this transaction

Application Navigation and Logic

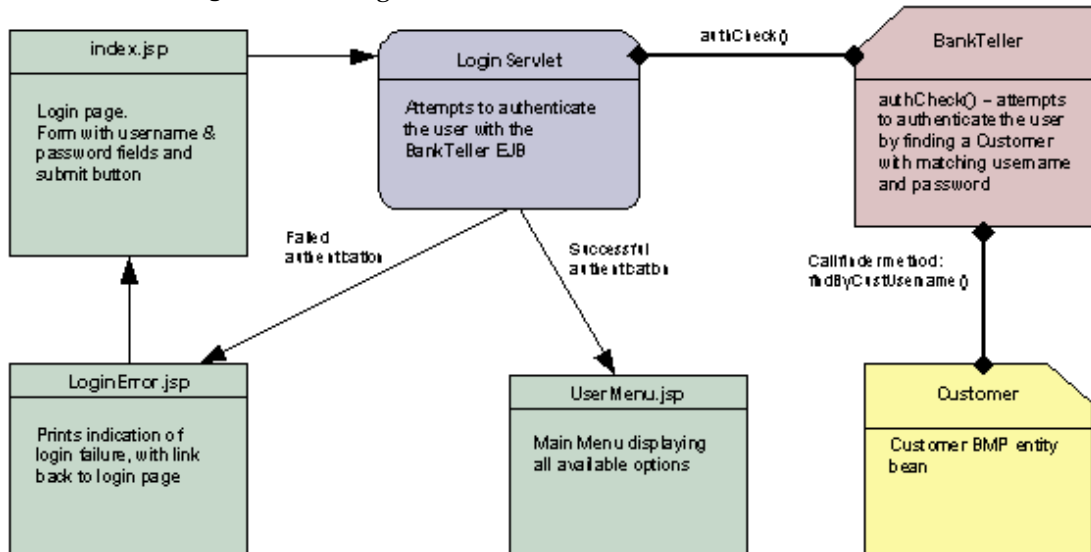Figure A-2 provides a high-level view of application navigation.

Figure A-2   Application Navigation and Logic

Login Process

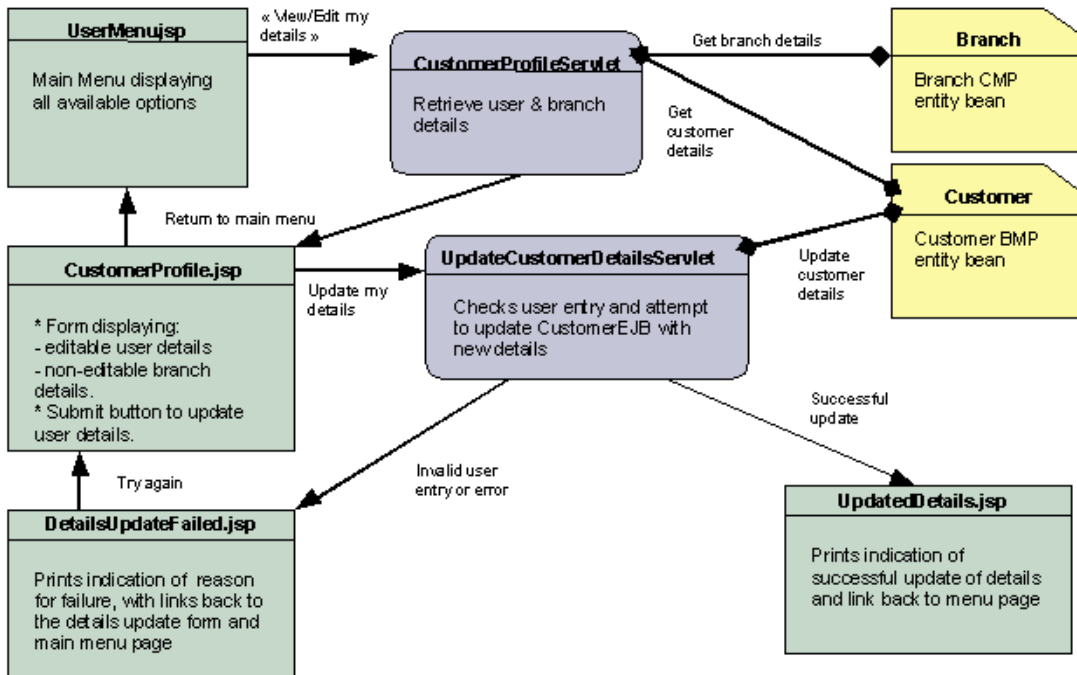Figure A-3 shows the login process used in the iBank application.

Figure A-3   Login Process

View/Edit Details

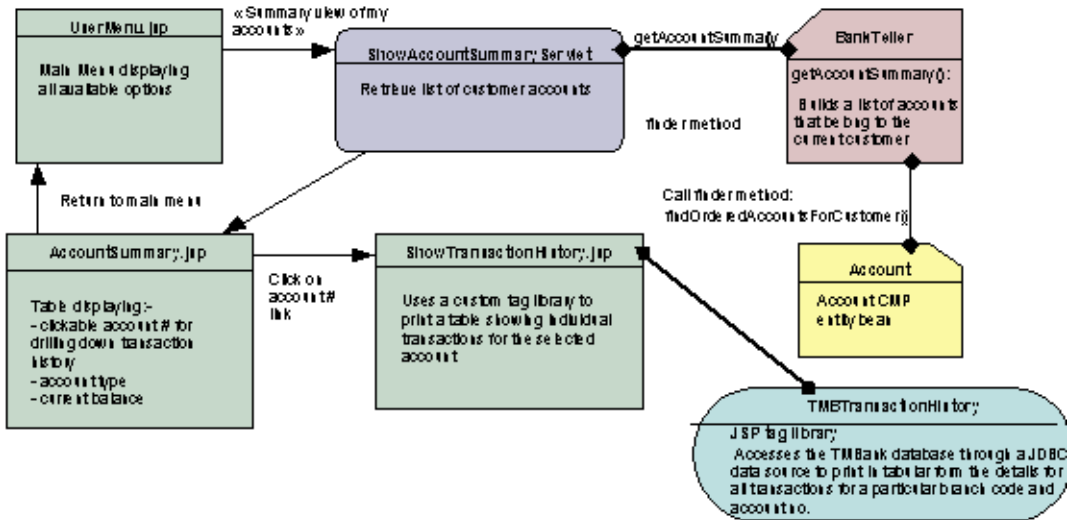Figure A-4 shows the view/edit details process used in the iBank application.

Figure A-4   View ⁄ Edit Details Process

Account Summary and Transaction History

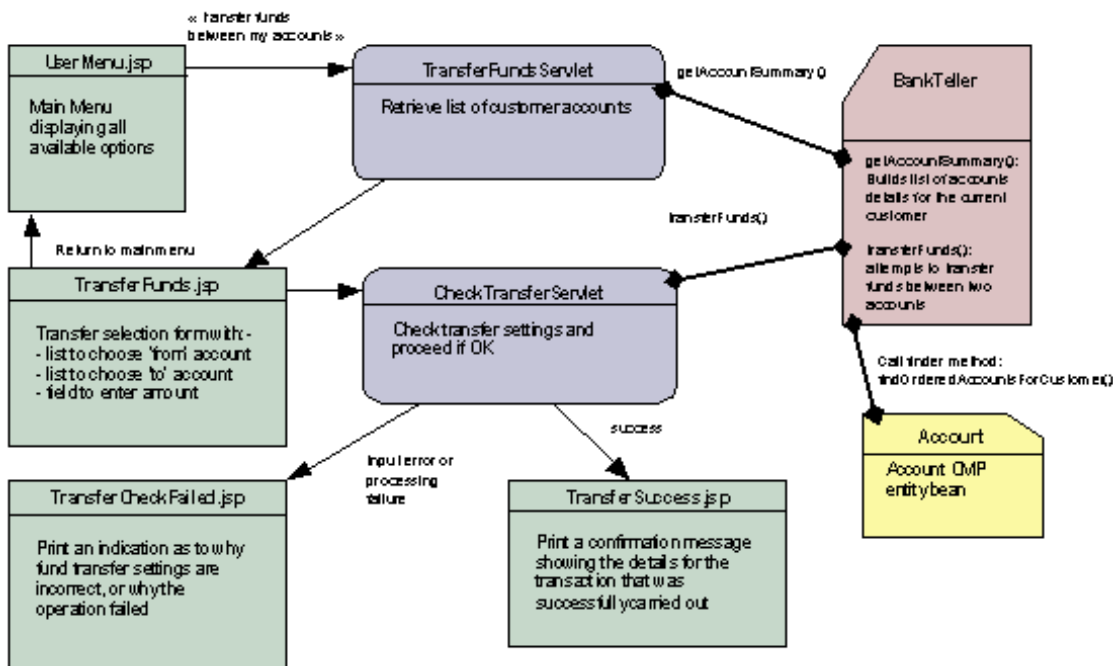Figure A-5 shows how the account summary and transaction history work in the iBank application.

Figure A-5 Account Summary and Transaction History

Fund Transfer

Figure A-6 shows how funds are transferred in the iBank application.

Figure A-6   Fund Transfer

Interest Calculation

Figure A-7 shows how interest is calculated in the iBank application.

Figure A-7   Interest Calculation
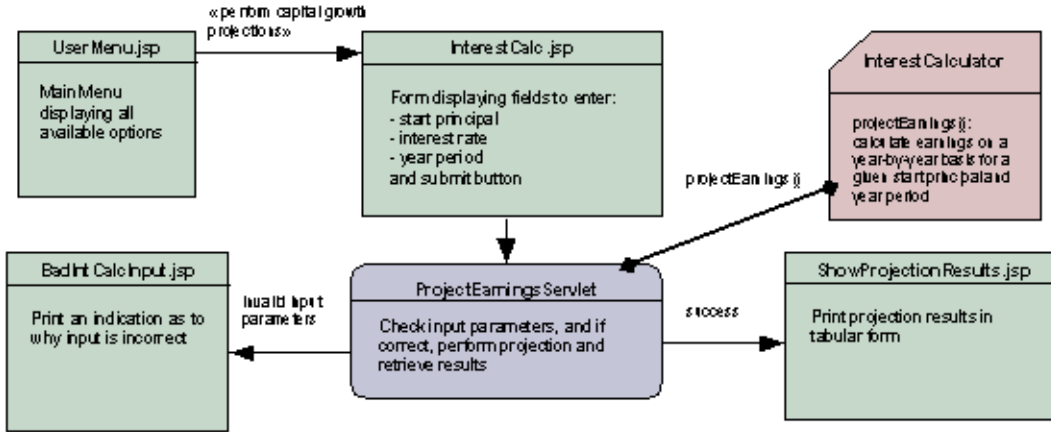
Application Components

Data Components

Each table in the database schema is encapsulated as an entity bean:

Entity Bean	Database Table
Account	ACCOUNT table
AccountType	ACCOUNT_TYPE table
Branch	BRANCH table
Customer	CUSTOMER table
Transaction	TRANSACTION_HISTORY table
TransactionType	TRANSACTION_TYPE table

All entity beans use container-managed persistence (CMP), except `Customer`, which uses bean-managed persistence (BMP).

Currently, the application only makes use of the `Account`, `AccountType`, `Branch`, and `Customer` beans.

Business Components

Business components of the application are encapsulated by session beans.

The `BankTeller` bean is a stateful session bean that encapsulates all interaction between the customer and the system. `BankTeller` is notably in charge of the following activities:

- Authenticating a customer through the `authCheck()` method

- Giving the list of accounts for the customer through the `getAccountSummary()` method

- Transferring funds between accounts on behalf of the customer through the `transferFunds()` method

The `InterestCalculator` bean is a stateless session bean that encapsulates financial calculations. It is responsible for providing the compound interest projection calculations, through the `projectEarnings()` method.

Application Logic Components (Servlets)

Component name	Purpose
LoginServlet	Authenticates the user with the BankTeller session bean (`authCheck()` method), creates the HTTP session and saves information pertaining to the user in the session. Upon successful authentication, forwards request to the main menu page (`UserMenu.jsp`)
CustomerProfileServlet	Retrieves customer and branch details from the Customer and Branch entity beans and forwards request to the view/edit details page (`CustomerProfile.jsp`).
UpdateCustomerDetailsServlet	Attempts to effect customer details changes amended in `CustomerProfile.jsp` by updating the Customer entity bean after checking validity of changes. Redirects to `UpdatedDetails.jsp` if success, or to `DetailsUpdateFailed.jsp` in case of incorrect input.
ShowAccountSummaryServlet	Retrieves the list of customer accounts from the BankTeller session bean (`getAccountSummary()` method) and forwards request to `AccountSummary.jsp` for display.

TransferFundsServlet	Retrieves the list of customer accounts from the BankTeller session bean (`getAccountSummary()` method) and forwards request to `TransferFunds.jsp` allowing the user to set up the transfer operation.
CheckTransferServlet	Checks the validity of source and destination accounts selected by the user for transfer and the amount entered. Calls the `transferFunds()` method of the BankTeller session bean to perform the transfer operation. Redirects the user to `CheckTransferFailed.jsp` in case of input error or processing error, or to `TransferSuccess.jsp` if the operation was successfully carried out.
ProjectEarningsServlet	Retrieves the interest calculation parameters defined by the user in InterestCalc.jsp and calls the `projectEarnings()` method of the InterestCalculator stateless session bean to perform the calculation, and forwards results to the `ShowProjectionResults.jsp` page for display. In case of invalid input, redirects to `BadIntCalcInput.jsp`

Presentation Logic Components (JSP Pages)

Component name	Purpose
`index.jsp`	Index page to the application that also serves as the login page.
`LoginError.jsp`	Login error page displayed in case of invalid user credentials supplied. Prints an indication as to why login was unsuccessful.
`Header.jsp`	Page header that is dynamically included in every HTML page of the application
`CheckSession.jsp`	This page is statically included in every page in the application and serves to verify whether the user is logged in (i.e. has a valid HTTP session). If no valid session is active, the user is redirected to the `NotLoggedIn.jsp` page.
NotLoggedIn.jsp	Page that the user gets redirected to when they try to access an application page without having gone through the login process first.
UserMenu.jsp	Main application menu page that the user gets redirected to after successfully logging in. This page provides links to all available actions.
`CustomerProfile.jsp`	Page displaying editable customer details and static branch details. This page allows the customer to amend their correspondence address.

`UpdatedDetails.jsp`	Page where the user gets redirected to after successfully updating their details.
`DetailsUpdateFaile d.jsp`	Page where the user gets redirected if an input error prevents their details to be updated.
`AccountSummaryPage .jsp`	This page displays the list of accounts belonging to the customer in tabular form listing the account no, account type and current balance. Clicking on an account no. in the table causes the application to present a detailed transaction history for the selected account.
`ShowTransactionHis tory.jsp`	This page prints the detailed transaction history for a particular account no. The transaction history is printed using a custom tag library.
`TransferFunds.jsp`	This page allows the user to set up a transfer from one account to another for a specific amount of money.
`TransferCheckFaile d.jsp`	When the user chooses incorrect settings for fund transfer, they get redirected to this page.
`TransferSuccess.js p`	When the fund transfer set-up by the user can successfully be carried out, this page will be displayed, showing a confirmation message.
`InterestCalc.jsp`	This page allows the user to enter parameters for a compound interest calculation.
`BadIntCalcInput.js p`	If the parameters for compound interest calculation are incorrect, the user gets redirected to this page.
`ShowProjectionResu lts.jsp`	When an interest calculation is successfully carried out, the user is redirected to this page that displays the projection results in tabular form.
`Logout.jsp`	Exit page of the application. This page removes the stateful session bean associated with the user and invalidates the HTTP session.
`Error.jsp`	In case of unexpected application error, the user will be redirected to this page that will print details about the exception that occurred.

Fitness of Design Choices with Regard to Potential Migration Issues

While many of application design choices made are certainly debatable especially in the "real-world" context, care was taken to ensure that these choices enable the sample application to encompass as many potential issues as possible as one would face in the process of migrating a typical J2EE application.

This section will go through the potential issues that you might face when migrating a J2EE application, and the corresponding component of iBank that was included to check for this issue during the migration process.

With respect to the selected migration areas to address, this section specifically looks at the following technologies:

Servlets

The iBank application includes a number of servlets, that enable us to detect potential issues with:

- The use of generic functionality of the Servlet API

- Storage/retrieval of attributes in the HTTP session and HTTP request

- Retrieval of servlet context initialization parameters

- Page redirection

Java Server Pages

With respect to the JSP specification, the following aspects have been addressed:

- Use of JSP declarations, scriptlets, expressions, and comments

- Static includes (`<%@ include file="…" %>`): notably tested with the inclusion of the `CheckSession.jsp` file in every page)

- Dynamic includes (`<jsp:include page=… />`): this is catered for by the dynamic inclusion of `Header.jsp` in every page

- Use of custom tag libraries: a custom tag library is used in the file `ShowTransactionHistory.jsp`

- Error pages for JSP exception handling: the `Error.jsp` page is the application error redirection page

JDBC

The iBank application accesses a database via a connection pool and the data source, both programmatically (BMP entity bean, BankTeller session bean, custom tag library) and declaratively (with the CMP entity beans).

Enterprise Java Beans

The iBank application uses a variety of Enterprise Java Beans.

Entity Beans

Bean-managed persistence (`Customer` bean): allows us to test the following:

- JNDI lookup of initial context
- Pooled data source access via JDBC
- Definition of a BMP custom finder ("`findByCustUsername()`")

Container-managed persistence ("`Account`" and "`Branch`" beans): allow us to test the following:

- Object/Relational mapping with the development tool and within the deployment descriptor
- Use of composite primary keys (`Account`)
- Definition of custom CMP finders (with the "`Account`" bean, and its `findOrderedAccountsForCustomer()` method). This is the occasion to look at differences in declaring the query logic in the deployment descriptor, and also to have a complex example returning a collection of objects.

Session Beans

Stateless session beans: `InterestCalculator` allows us to test the following:

- Using and deploying a stateless session bean
- Calling a business method for calculations

Stateful session beans: `BankTeller` allows us to test the following:

- Looking up various interfaces using JNDI and initial contexts

- Using JDBC to perform database queries

- Using various transactional attributes on bean methods

- Using container-demarcated transactions

- Maintaining conversational state between calls

- Business methods acting as front-ends to entity beans (e.g., the "`getAccountSummary()`" method)

Application Assembly

The iBank application is assembled by following the J2EE standard procedures. It contains the following components:

- A Web application archive file for the Web application module, and EJB-JAR archives for the EJBs

- An enterprise application archive file (EAR file) for the final packaging of the Web application and EJB modules

Index

M

O

R

S

W