

Migrating and Redeploying Server Applications Guide

Sun ONE Application Server

Version 7

816-7148-10
October 2002

Copyright © 2002 Sun Microsystems, Inc. Some preexisting portions Copyright © 2002 Netscape Communications Corporation. All rights reserved.

Sun, Sun Microsystems, and the Sun logo, iPlanet, and the iPlanet logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Netscape and the Netscape N logo are registered trademarks of Netscape Communications Corporation in the U.S. and other countries. Other Netscape logos, product names, and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries.

This product includes software developed by Apache Software Foundation (<http://www.apache.org/>). Copyright (c) 1999 The Apache Software Foundation. All rights reserved.

This product includes software developed by the University of California, Berkeley and its contributors. Copyright (c) 1990, 1993, 1994 The Regents of the University of California. All rights reserved.

Federal Acquisitions: Commercial Software—Government Users Subject to Standard License Terms and Conditions

The product described in this document is distributed under licenses restricting its use, copying, distribution, and decompilation. No part of the product or this document may be reproduced in any form by any means without prior written authorization of the Sun-Netscape Alliance and its licensors, if any.

THIS DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2002 Sun Microsystems, Inc. Pour certaines parties préexistantes, Copyright © 2002 Netscape Communication Corp. Tous droits réservés.

Sun, Sun Microsystems, et the Sun logo, iPlanet, and the iPlanet logo sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et d'autre pays. Netscape et the Netscape N logo sont des marques déposées de Netscape Communications Corporation aux Etats-Unis et d'autre pays. Les autres logos, les noms de produit, et les noms de service de Netscape sont des marques déposées de Netscape Communications Corporation dans certains autres pays.

Le produit décrit dans ce document est distribué selon des conditions de licence qui en restreignent l'utilisation, la copie, la distribution et la décompilation. Aucune partie de ce produit ni de ce document ne peut être reproduite sous quelque forme ou par quelque moyen que ce soit sans l'autorisation écrite préalable de l'Alliance Sun-Netscape et, le cas échéant, de ses bailleurs de licence.

CETTE DOCUMENTATION EST FOURNIE “EN L'ÉTAT”, ET TOUTES CONDITIONS EXPRESSES OU IMPLICITES, TOUTES REPRÉSENTATIONS ET TOUTES GARANTIES, Y COMPRIS TOUTE GARANTIE IMPLICITE D'APTITUDE À LA VENTE, OU À UN BUT PARTICULIER OU DE NON CONTREFAÇON SONT EXCLUES, EXCEPTÉ DANS LA MESURE OÙ DE TELLES EXCLUSIONS SERAIENT CONTRAIRES À LA LOI.

Contents

About This Guide	5
What You Should Know	5
How This Guide is Organized	6
Documentation Conventions	6
Chapter 1 About Sun ONE Application Server 7	9
Sun ONE Application Server 7 Architecture	9
J2EE Component Standards	12
Development Environments	13
Sun ONE Application Server 6.0/6.5 Development Environment	13
Sun ONE Application Server 7 Development Environment	14
Administration Tools	15
Sun ONE Application Server 6.0 Administration Tools	15
Sun ONE Application Server 6.5 Administration Tools	16
Sun ONE Application Server 7 Administration Tools	17
Database Connectivity	19
Database Support in Sun ONE Application Server 6.0	19
Database Support in Sun ONE Application Server 6.5	20
Database Support in Sun ONE Application Server 7	20
J2EE Application Components and Migration	21
Migration and Redeployment	22
Why is Migration Necessary	23
What Needs to be Migrated	23
What is Redeployment	24
Chapter 2 Migration Considerations and Strategies	27

About Sun ONE Application Server 6.0/6.5	27
Migration Issues From Sun ONE Application Server 6.x to 7	29
Migrating JDBC Code	30
Establishing Connections Through the DriverManager Interface	30
Using JDBC 2.0 Data Sources	32
Migrating Java Server Pages and JSP Custom Tag Libraries	36
Migrating Servlets	37
Obtaining a Data Source from the JNDI Context	38
Declaring EJBs in the JNDI Context	38
EJB Migration	39
EJB Changes Specific to Sun ONE Application Server 7	39
Migrating Web Applications	40
Migrating Web Application Modules	41
Particular setbacks when migrating servlets and JSPs	42
Migrating Enterprise EJB Modules	43
Migrating Enterprise Applications	44
Application root context and access URL	45
Migrating Proprietary Extensions	46
Migrating Example: iBank	46
Manual Migration of iBank Application	47
Web application changes	48
EJB Changes	49
Assembling Application for Deployment	68
Deploying iBank application on Sun ONE Application Server 7 using the asadmin utility ..	68
Migrating iBank using Sun ONE Studio for Java 4.0	69
Creating a Web application module in Sun ONE Studio for Java	72
Converting CMP Entity EJBs from 1.1 to 2.0	78
Creating an EJB module in Sun ONE Studio for Java	90
Creating an enterprise application in Sun ONE Studio for Java	110
Deploying an application in Sun ONE Application Server 7	112
Migration from BEA WebLogic Server v6.1 and IBM WebSphere v4.0	113
Chapter 3 Migration from KIVA/NAS 4.1 to Sun ONE AS 7	115
Introduction	115
Migration Preparation	115
Migration Process Overview	115
Preparing your Working Environment	117
Preparing a Project for Automated Migration	118
Preparing the GXR file	119
Before Running the Extraction Tool	119
Migrating OnlineBankSample	120
Running the Migration Toolbox	120

Create a Toolbox	120
Chapter 4 Migration from NetDynamics to Sun ONE AS 7	143
Introduction	143
Migration Preparation	144
Migration Process Overview	144
Preparing your Working Environment	145
Preparing a Project for Automated Migration	146
Migrating ToolBox Sample Application	148
Running the Migration Toolbox	148
Create a Toolbox Builder	148
Chapter 5 Automating Migration	163
Sun ONE Migration Tool for Application Servers	163
Sun ONE Migration Toolbox (formerly iPlanet Migration Toolbox)	164
Redeploying Migrated Applications	164
Appendix A	165
iBank Application specification	165
Tools used for the development of the application	166
Database schema	166
Application navigation and logic	171
Application Components	174
Fitness of design choices with regard to potential migration issues	177
Appendix B	181
Sun ONE Migration Toolbox	181
Supported Platforms	181
Migration	181
Toolbox Builder	182
Kiva Migration Toolbox Builder	182
NetDynamics Migration Toolbox Builder	186
Tools and Toolboxes	192
Creating New Tools	192
Cloning Tools	192
Deleting Tools	192
Importing & Exporting Tools	193
Toolbox Merging	193
Troubleshooting	193
Toolbox Installation & Configuration	193

Extraction	194
Translation	196
Post-Migration	196
Appendix C	199
Migrating from EJB 1.1 to EJB 2.0	199
EJB Query Language	199
Local Interfaces	200
EJB 2.0 Container-Managed Persistence (CMP)	201
Defining Entity Bean Relationships	202
Message-Driven Beans	202
Migrating EJB Client Applications	202
Declaring EJBs in the JNDI Context	202
Recap on Using EJB JNDI References	203
Migrating CMP Entity EJBs	204
Migrating the Bean Class	205
Migration of ejb-jar.xml	208
Custom Finder Methods	208
Index	211

About This Guide

This *Migrating and Redeploying Server Applications Guide* describes how J2EE applications are migrated from earlier versions of the Sun ONE Application Server (formerly known as 'iPlanet Application Server') to Sun ONE Application Server 7.

In addition, this guide describes how NetDynamics applications and applications from the Netscape Application Server (NAS) are migrated to the Sun ONE Application Server 7.

This manual is intended for system administrators, network administrators, application server administrators and web developers who have an interest in migration issues.

What You Should Know

Before you begin, you should already be 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

How This Guide is Organized

This guide is organized as follows:

- *About Sun ONE Application Server 7* - describes the architecture of the Sun ONE Application Server 7 and the differences between J2EE standards and application components implemented with this version of the Sun ONE Application Server versus previous versions.
- *Migration and Redeployment* - describes those application components that need to be migrated and why, as well as the redeployment process for migrated applications.
- *Migration Considerations and Strategies* - describes considerations and strategies for migrating applications from competing platforms and from previous versions of the Sun ONE Application Server. There are also sample migration applications included that provide an end-to-end description of the migration process.
- *Automating Migration* - describes the available automation tools used to migrate applications from competing platforms and earlier versions of the Sun ONE Application Server.
- *Redeploying Migrated Applications* - describes how migrated applications are redeployed to the Sun ONE Application Server.

Documentation Conventions

File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except forward slashes are used instead of backslashes to separate directories.

This guide uses URLs of the form: `http://server.domain/path/file.html`, where:

- `server` is the name of the server where you are running the application.
- `domain` is your internet domain name.
- `path` is the directory structure on the server.
- `file` is an individual filename.

The following table shows the typographic conventions used throughout Sun ONE documentation

Table 1 Typographic Conventions

Typeface	Meaning	Examples
Monospaced	The names of files, directories, sample code, and code listings; and HTML tags	Open <code>Hello.html</code> file. <HEAD1> creates a top level heading.
<i>Italics</i>	Book titles, variables, other code placeholders, words to be emphasized, and words used in the literal sense	See Chapter 2 of the <i>Migrating and Redeploying Server Applications Guide</i> . Enter your <i>UserID</i> . Enter <i>Login</i> in the Name field.
Bold	First appearance of a glossary term in the text	Templates are page outlines.

About Sun ONE Application Server 7

This chapter describes the architecture of the Sun ONE Application Server 7 and the J2EE components that are integral to the server environment. In addition, the differences between the Sun ONE Application Server 7 environment and earlier Sun ONE Application Server environments are described.

The following topics are addressed:

- Sun ONE Application Server 7 Architecture
- J2EE Component Standards
- Development Environments
- Administration Tools
- Database Connectivity
- J2EE Application Components and Migration
- Migration and Redeployment

Sun ONE Application Server 7 Architecture

Application servers provide the framework for a client to connect to a backend source, execute the application logic, and return the result to the client. The application server occupies the middle-tier in the three-tier computing model.

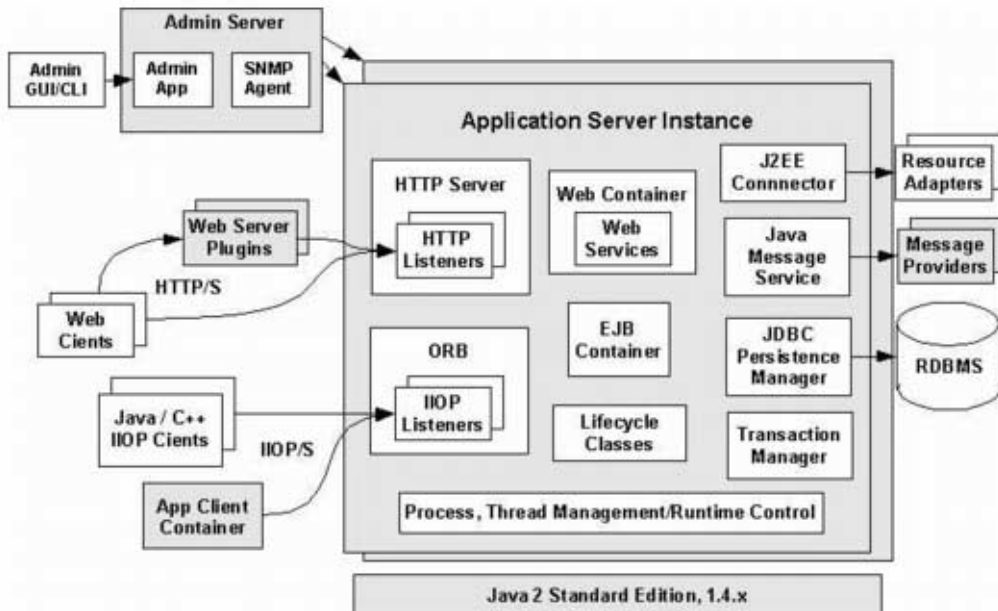
The Sun ONE Application Server 7 is a Java application server and is fully compliant with the Java 2 Enterprise Edition (J2EE™) specifications. J2EE provides a complete, secure foundation and describes a rich set of standards for security, development, deployment, code re-use and portability that allows the enterprise to create applications that are portable and vendor independent.

The Sun ONE Application Server 7 provides a robust J2EE platform for the development, deployment, and management of e-commerce application services to a broad range of servers, clients, and devices.

Sun ONE Application Server 7 is a J2EE 1.3 compliant application server.

The key goals of this architecture are horizontal and vertical scalability, high availability, reliability, performance, and standards compliance. Sun ONE Application Server 7 is also a significant architectural departure from the first generation of Sun ONE application server products. By combining existing and strong Sun ONE products and technologies with the J2EE 1.3 standards, Sun ONE Application Server 7 architecture is built upon a proven framework of technologies.

Sun ONE Application Server 7 Architecture



The Sun ONE Application Server architecture is graphically represented in the figure “Sun ONE Application Server 7 Architecture”. The architecture shows the Sun ONE Application Server components, sub-systems, access paths and how external entities interface with the core server.

Sun ONE Application Server 7 architecture, is highly componentised which results in a very highly manageable architecture. All the services required by the J2EE specification are present with well-defined standard interfaces to invoke them from within applications.

The web user interface, new in Sun ONE Application Server 7, provides for easy remote server management. In fact, the server is designed such that one administration server could be used to administer multiple numbers of administered servers. The task of creating a new administered server instance has been greatly simplified in this new version.

Support for the type 2 JDBC drivers bundled along with the earlier versions of Sun ONE Application Server has been withdrawn. As a result of this, the platform has moved towards a more standardized approach to JDBC resource management.

By using the JDK 1.4 for the server operation, Sun ONE Application Server utilizes the enhanced abilities of this newer version of JDK to its advantage.

A typical J2EE application is composed of an *n*-tier system in which a client obtains processed information from a Web server or an application server. The servers in turn access the information from enterprise systems such as RDBMS or ERP, process them by using contained business logic, and deliver the processed information to the client in an appropriate format. These layers can be designated as client layer (Web browser or rich Java client), middle layer (Web servers and application servers), and the back-end layer or data layer (enterprise systems such as databases).

The J2EE application model within the Sun ONE Application Server allows developers to focus on the business logic while J2EE components handle all the low level details. Therefore, applications and services can be easily enhanced and rapidly deployed, allowing business to quickly react to competitive changes. By providing an open standard architecture through the J2EE Platform, Sun ONE Application Server solves the problem of the cost and complexity in developing multi-tiered services that are scalable, highly available, secure and reliable.

J2EE Component Standards

Sun ONE Application Server 7 is a J2EE v1.3 compliant server based on the component standards developed by the Java community for Servlets, Java Server Pages (JSPs), and Enterprise JavaBeans (EJBs).

In contrast to Sun ONE Application Server 7, Sun ONE Application Server 6.0/6.5 is a J2EE v1.2 compliant server. Between the two 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 v1.3 compliant Sun ONE Application Server 7 and the J2EE v1.2 Sun ONE Application Server 6.0/6.5.

Application Server Version Comparison of APIs for J2EE Components

Component API	Sun ONE Application Server 6.0/6.5	Sun ONE Application Server 7
JDK	1.2.2	1.4
Servlet	2.2	2.3
JSP	1.1	1.2
JDBC	2.0	2.0
EJB	1.1	2.0
JNDI	1.2	1.2
JMS	1.0	2.0
JTA	1.0	1.01

In addition, the two products support a number of technologies connected with XML standards and Web Services which, while not part of the J2EE specification, are mentioned in the following table due to the increasing usage of these standards in enterprise applications.

Additional Application Server Supported Technologies

Technology	Sun ONE Application Server 6.0/6.5	Sun ONE Application Server 7
XML document processing (API and XML parser)	JAXP 1.0, Apache Xerces	JAXP 1.1
SOAP/Java support for Web Services	SOAP 1.1 (IBM SOAP4J framework)	Apache SOAP 2.2, JAX-RPC 1.0, JAXM 1.1, JAXR 1.0

Development Environments

This section characterizes the differences between the development environments for the Sun ONE Application Server 6.0/6.5 and the Sun ONE Application Server 7. The following topics are described:

- Sun ONE Application Server 6.0/6.5 Development Environment
- Sun ONE Application Server 7 Development Environment

Sun ONE Application Server 6.0/6.5 Development Environment

Sun ONE Application Server 6.0/6.5 offers an evaluation version of Sun ONE Studio for Java, which is especially geared towards application development for this version of the Sun ONE Application Server.

It is a very complete development environment in Java, based on the NetBeans platform. This IDE provides an extremely rich range of features for designing and developing Java applications and EJB components. It also integrates through a plug-in with Sun ONE Application Server for assembly, deployment, and debugging of the various J2EE components of an application. It is available in both Windows and Solaris.

Of the third-party vendor solutions available on the market, the recently released *Borland JBuilder 6 Enterprise* is an extremely mature, comprehensive product, with the added advantage of being available on several platforms (Windows, Solaris, Linux, and MacOS X). In addition to its Java development features (servlets, JSP pages, EJB components, graphic applications), JBuilder also caters for UML design,

unit testing, collaborative development, and XML development. Moreover, JBuilder integrates perfectly with mainstream application servers (including the Sun ONE Application Server) for assembly, deployment and debugging of Web applications and EJB components.

Sun ONE Application Server 7 Development Environment

The availability of a fully integrated development solution is key to the success of the Sun ONE Application Server 7. Sun ONE Studio for Java Enterprise Edition 4 is the Sun ONE strategic tool for Sun ONE application development.

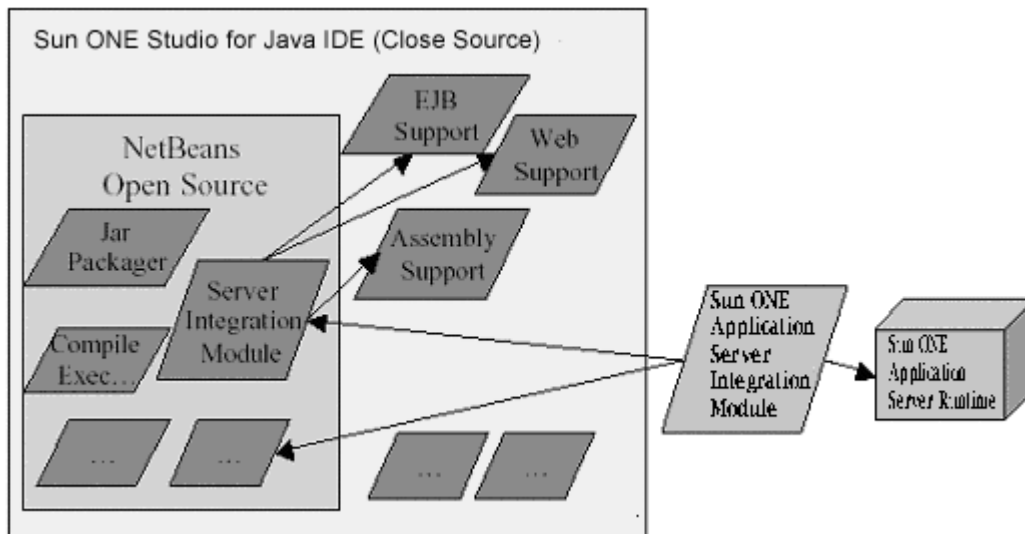
Sun ONE Studio for Java 4 is provided with Sun ONE Application Server.

Some of the key features of Sun ONE Studio for Java Enterprise Edition 4 are:

- Ability to build EJBs quickly and easily
- Ability to assemble applications from EJBs and package applications for deployment
- Application server integration for deployment
- Ability to develop and publish web services
- Sun ONE studio for java enterprise service presentation toolkit
- Ability to integrate with the Sun ONE Application Server 7

As shown in the figure “Sun ONE Studio Enterprise Edition and Sun ONE Application Server 7 Integration”, the Sun ONE Application Server 7 integration module relies upon the NetBeans Open Source modules that are implemented from the Sun ONE Studio Close Source.

Sun ONE Studio Enterprise Edition and Sun ONE Application Server 7 Integration



Administration Tools

This section characterizes the differences between the administration tools for the Sun ONE Application Server 6.0, Sun ONE Application Server 6.5, and the Sun ONE Application Server 7. The following topics are described:

- Sun ONE Application Server 6.0 Administration Tools
- Sun ONE Application Server 6.5 Administration Tools
- Sun ONE Application Server 7 Administration Tools

Sun ONE Application Server 6.0 Administration Tools

Sun ONE Application Server 6.0 features a full set of graphical administration tools, which cover all the aspects of server management and administration

- **Sun ONE Console** - the main administration control panel. Sun ONE console gives fast access to the Administration Server Console, the Directory Server, and the Administration Tool.

- **Administration Server Console** - used to define event-logging options and to create SSL security certificates.
- **Sun ONE Directory Server Console** - used for administration of the Sun ONE Directory Server. The Directory Server is used to administer the two main information directory trees, the *user directory* (user and organizational unit administration), and the *configuration directory* (server configuration).
- **Sun ONE Administration Tool** - used to administer one or more instances of Sun ONE Application Server 6.0, along with the applications deployed. It also enables JDBC drivers and data sources to be configured.
- **Sun ONE Registry Editor** (*kregedit*) - is a graphical tool similar to the windows registry editor (*regedit*). It is used to adjust certain parameters specific to the Sun ONE Application Server, stored in a specific registry.

Sun ONE Application Server 6.5 Administration Tools

Sun ONE Application Server 6.5 can be administered using integrated Administration Tool, Sun ONE registry editor and command line tools, which are described below:

- **Sun ONE Application Server Administration Tool** - a stand-alone java application with a graphical user interface that allows you to administer one or more instances of Sun ONE Application Server along with administering application components.
- **Command line tools** - can be run from the command-line prompt on Windows and the shell prompt on Solaris. You can perform a variety of tasks using the command line tools, right from basic configuration to deploying an application. To get a complete description of any command-line tool, type `[command] -help` at the command prompt. For ease of use, most of the command-line tools have been integrated with the Sun ONE Application Server Administration Tool and the Sun ONE Application Server Deployment Tool.
- **Sun ONE Registry Editor** (*kregedit*) - a stand-alone GUI tool similar to the Windows Registry editor (*regedit*). It can display and edit registry information for Sun ONE Application Server.

Sun ONE Application Server 7 Administration Tools

The Administration Server in Sun ONE Application Server 7 is a special instance of the Server that serves the Administrative interface and controls some global settings common to all server instances. It is a web-based server that contains the forms used to configure the Sun ONE Application Server.

This graphical tool allows you to manage your application server including viewing error and access logs, monitoring server usage, creating and editing virtual servers, apply configuration changes and start or stop server instances.

When you installed the Sun ONE Application Server, you chose a port number for the Administration Server, or used the default port of 4848. To access the Administrative interface, in a web browser type:

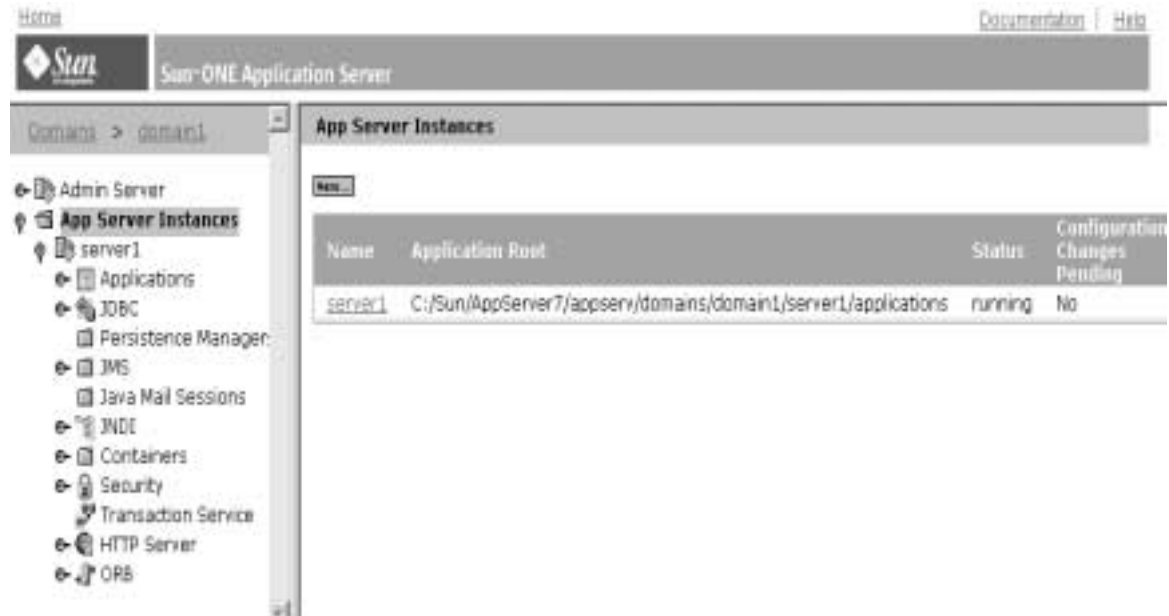
<http://hostname:port/admin>

You are prompted for the configured user name and password. Upon entering this information and clicking the OK button, the home page of the Administrative interface is displayed, as shown in the figure "Administrative Interface Home Page".

The left pane is a tree view of all items you can configure in the Sun ONE Application Server. To use the Administrative interface, click an item in the left pane. The right pane displays the page associated with that item.

You can access help for any page in the Administrative interface by clicking the Help button in the banner at the top of the Administrative interface. The online help describes the use of the page you are accessing and gives information about what to enter in the fields on the page.

Administrative Interface Home Page



Sun ONE Application Server 7 contains a command line interface. You can use a utility and commands to perform the same set of tasks as you can perform in the Administrative interface. You can use these commands either from a command prompt in the shell, or you can call them from other scripts and programs. Using these commands you can automate administration tasks that otherwise might become repetitive.

Sun ONE Application Server 7 has a command line utility *asadmin*, which can be run from the command-line prompt on Windows and the shell prompt on Solaris. The *asadmin* utility has a set of commands used to perform administrative tasks. You can use these commands to perform all the same tasks that are performed from the Administrative Interface, from basic configuration to deploying an application. To get a complete description of any command, type *help* after entering the *asadmin* utility.

You can run *asadmin* either in singlemode or multimode. In singlemode you run one command at a time from the command prompt. In multimode you can run multiple commands without needing to reenter environment-level information.

Database Connectivity

This section describes type of drivers included in the Sun ONE Application Server 6.0, Sun ONE Application Server 6.5 and Sun ONE Application Server 7. This section also describes the database(s) supported by each type of driver.

The following topics are included:

- "Database Support in Sun ONE Application Server 6.0"
- "Database Support in Sun ONE Application Server 6.5"
- "Database Support in Sun ONE Application Server 7"

Database Support in Sun ONE Application Server 6.0

Sun ONE Application Server 6.0 includes a series of type 2 JDBC drivers (which require installation of native client libraries for access to the corresponding DBMSs), which provide connectivity to the following main market database back-ends:

- DB2 6.1, 7.1
- Informix 7.3, 9.1.4, 9.2
- Oracle 8.0.5, 8i, 9i
- Sybase 11.9.2, 12
- Microsoft SQL Server 7
- PointBase 3.5

It is possible to use third-party Type 4 JDBC drivers, by declaring them via the Sun ONE Application Server Administration Tool, or via a specific, separate utility: *db_setup.sh* in *Solaris*, *jdbcsetup* in *Windows*.

JDBC data sources and connection pool properties can be added and configured from the Sun ONE Application Server Administrative interface, or from the *iasdeploy* command line utility. For the latter, an XML file is passed which defines the properties of the data source to be defined.

Database Support in Sun ONE Application Server 6.5

Sun ONE Application Server 6.5 provides a JDBC type 2 driver which supports a variety of databases, including:

- DB2 5.1 and 6.1 and client version 7.1
- Informix 7.3, 9.1.4, 9.2 and client version SDK 2.40
- Oracle 8i, 9i
- Sybase 12
- Microsoft SQL Server 7

Configuration of native JDBC drivers on Solaris can be done via a specific utility, *db_setup.sh*. On Windows, native drivers are automatically configured during installation if the database client libraries are present in your machine. If you install a database client library after Sun ONE Application Server installation, then restart Sun ONE Application Server to automatically configure the native drivers.

It is possible to use third-party Type 4 JDBC drivers, by declaring them via the Sun ONE Application Server Administration Tool, on Solaris as well as on Windows.

Sun ONE Application Server allows you to adjust database connectivity through connection parameters via the Sun ONE Application Server Administrative interface. The connection parameters are grouped in the following categories:

- Connection
- Threads, and
- Database cache

Database Support in Sun ONE Application Server 7

Sun ONE Application Server 7 has Type 2 and Type 4 XA capable JDBC 2.0 style drivers, which provide connectivity to the main market database back-ends:

- DB2 v7
- Oracle 8.1.7
- Sybase v11

- PointBase version 4.2RE

All external JDBC compliant drivers are supported by Sun ONE Application Server.

JDBC data sources and connection pool properties can be added and configured from the Sun ONE Application Server Administration interface, or from the *asadmin* command line utility.

For details on configuring JDBC Data sources and connection pools, refer to the section "Using JDBC 2.0 Data Sources".

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 v1.3 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 2, section "Migration Issues From Sun ONE Application Server 6.x to 7".

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 Sun ONE Application Server. Redeployment is also described within this section.

The following topics are addressed:

- Why is Migration Necessary
- What Needs to be Migrated
- What is Redeployment

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 Sun ONE 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.

Usage of such proprietary features 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 API's

- Java source files that contain Proprietary API's

Deployment descriptors (XML files)

Deployment is accomplished by specifying deployment descriptors (DDs) for EJBs (ejb-jar), front-end web components (war) and enterprise applications (ear). 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 JSP's containing proprietary API's needs to be rewritten before they can be migrated.

Java source files

The Java source files can be Servlets, EJBs or other helper classes. The Servlets and EJBs 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 API* is the 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.

Files within the above file categories need to be migrated to Sun ONE Application Server. The details on how to migrate each of the indicated file categories are provided in Migration Issues From Sun ONE Application Server 6.x to 7.

What is Redeployment

Redeployment refers to deploying a previously deployed application from an earlier version of Sun ONE Application Server, or from applications that were previously deployed, but migrated, from a competing application server platform.

The act of redeploying an application typically refers to using the standard deployment actions outlined in the Sun ONE Application Server *Administrator's Guide*. However, when migration activities are performed with automated tools, such as the *Sun ONE Migration Tool for Application Servers* (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.

For more information about the available migration tools, refer to Automating Migration.

Migration Considerations and Strategies

This chapter describes the considerations and strategies that are needed when moving J2EE applications from Sun ONE Application Server 6.0 and 6.5 to Sun ONE Application Server 7.

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

The following topics are addressed:

- About Sun ONE Application Server 6.0/6.5
- Migration Issues From Sun ONE Application Server 6.x to 7
- Migrating Example: iBank

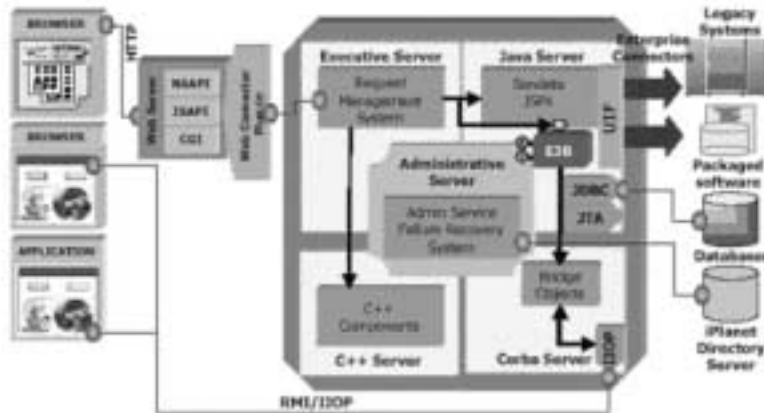
About Sun ONE Application Server 6.0/6.5

Sun ONE Application Server version 6.0 is a multi-platform application server based entirely on the J2EE 1.2 specification. Supported platforms include Windows NT and 2000, Solaris, AIX, and HP-UX.

In addition, Sun ONE Application Server 6.0 integrates with many Web servers through specific Web connector plug-ins that it ships with. These connectors enable it to be coupled with Sun ONE Web Server, Microsoft IIS, or Apache.

The Sun ONE Application Server 6.0/6.5 architecture is shown in the following figure.

Sun ONE Application Server 6.0/6.5 Architecture



As shown in the figure "Sun ONE Application Server 6.0/6.5 Architecture", there are four internal servers, which are often called engines or processes. These processes are responsible for all the processing in the Sun ONE Application Server. The four internal servers of the Sun ONE Application Server 6.0/6.5 are:

Executive Server - provides most system services (some services are managed by the Administrative Server).

Administrative Server - provides system services for Sun ONE Application Server Administration and failure recovery.

Java Server - provides services to java applications.

C++ Server - components written in C++ are hosted in C++ server.

When a web server forwards requests to Sun ONE Application Server 6.0/6.5, the requests are first received by the Executive Server process (KXS). The KXS process forwards the request either to a Java Server process (KJS) or to a C++ Server process (KCS). A KJS process runs Java programming logic, whereas a KCS process runs C++ programming logic. Each KJS and KCS process maintains a specified number of threads and runs the programming logic to completion on those threads. The results are returned to the web server and sent on to the client browser.

Migration Issues From Sun ONE Application Server 6.x to 7

This section describes the issues that will arise while migrating the main components of a typical J2EE application from Sun ONE Application Server 6.0 and 6.5 to Sun ONE Application Server 7.

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 6.0 and 6.5 to Sun ONE Application Server 7. 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*.

The following migration processes are described:

- Migrating JDBC Code
- Migrating Servlets
- Migrating Java Server Pages and JSP Custom Tag Libraries
- Obtaining a Data Source from the JNDI Context
- EJB Migration
- EJB Changes Specific to Sun ONE Application Server 7

- Migrating Web Applications
- Migrating Enterprise EJB Modules
- Migrating Enterprise Applications

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 *Data Source* 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

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@ibcn:1521:tmbank";

Class.forName(driver).newInstance();

Properties props = new Properties();

props.setProperty("user", "tmb_user");

props.setProperty("password", "tmb_user");

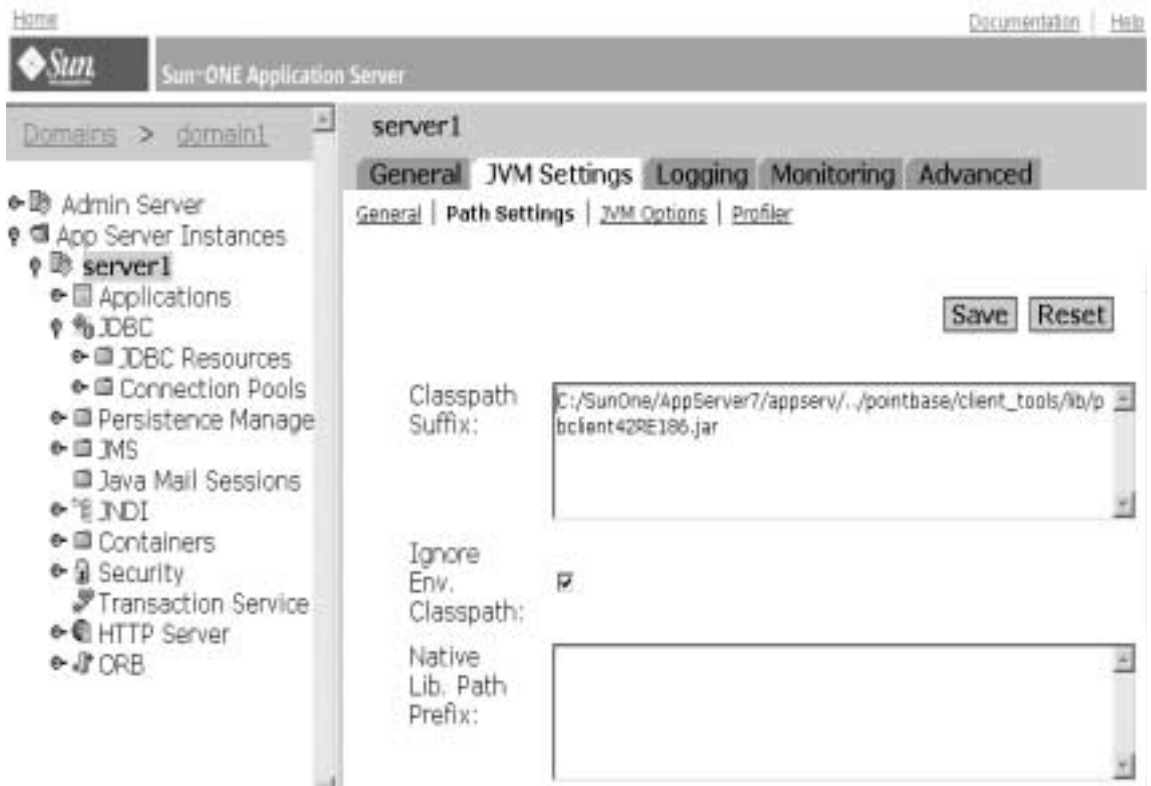
Connection conn = DriverManager.getConnection(url, props);
```


This code can be fully ported from Sun ONE Application Server 6.0/6.5 to Sun ONE Application Server 7, as long as Sun ONE 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 Sun ONE Application Server 7, you should:

- Place the archive (JAR or ZIP) for the driver implementation in the */lib* directory of the Sun ONE Application Server 7 installation directory.
- Modify the *CLASSPATH* by setting the path for the driver through the GUI of the admin server. 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`.

The figure "Using the JVM Settings to Set the Classpath Suffix" shows adding the path of the driver in the classpath suffix through GUI.

Using the JVM Settings to Set the Classpath Suffix



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 `jdbctestup` 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 Sun ONE Application Server installation root directory.

In Sun ONE Application Server 7, 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 `asadmin.bat` in windows and `asadmin` file in Solaris kept at Sun ONE Application Server 7 installation's `bin` directory. Then on the `asadmin` prompt, following commands would 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] [--instance instancename]
--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 c11
-p 4848 -instance server1 --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] [--instance instancename] --connectionpoolid id  
[--enabled=true] [--description text] [--property  
(name=value)[:name=value]*] jndiname
```

For example:

```
asadmin>create-jdbc-resource -u admin -w password -H c11 -p 4848  
--instance server1 --connectionpoolid oraclepool jdbc/IBANK
```

Here jdbc resource is created for the connection pool created above with the JNDI name 'jdbc/IBANK'.

Here is the procedure to follow when registering a data source in Sun ONE Application Server 7 through graphical interface.

1. *Register the data source classname*

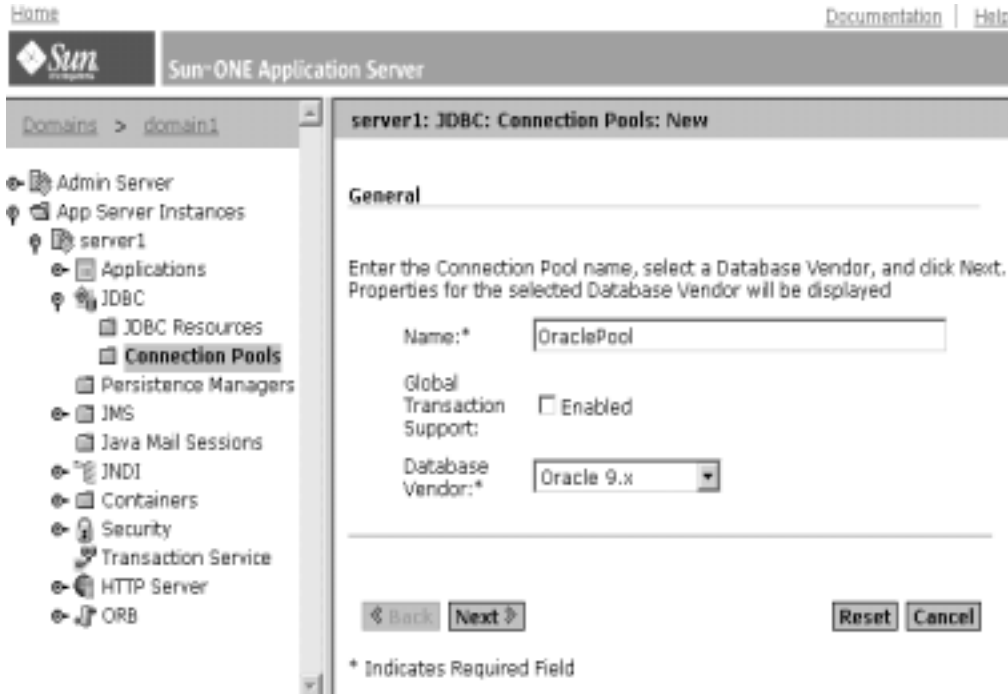
- a. Place the archive (JAR or ZIP) for the data source class implementation in the */lib* directory of the Sun ONE Application Server 7 installation directory.
- b. Modify the CLASSPATH by setting the path for the driver through the GUI of the admin server. Click at the server instance "server1" and 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 Sun ONE Application Server 7, 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 Sun ONE 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.

Configuring Connection Pool through GUI



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

Sun ONE Application Server 7 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* would have 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:

- Obtain an initial JNDI context
- Obtain a reference to the data source by using a JNDI lookup
- Obtain a connection using this referen

1. Obtaining 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. Obtaining 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. Obtaining the connection

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

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

Sun ONE Application Server 6.0/6.5 and 7 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 6.0/6.5 complies with the JSP 1.1 specification and Sun ONE Application Server 7 complies with the JSP 1.2 specification.

JSP 1.2 specification contains many new features as well as corrections and clarifications of areas that were not quite right in JSP 1.1 specification.

The most significant changes are

- JSP 1.2 is based on Servlet 2.3 and Java 2. JSP 1.2 applications will not run on platforms that only support JDK 1.1. JSP 1.2 is backward compatible with JSP 1.1, so JSP 1.1 application should run without any tweaking in a JSP 1.2 compliant container.
- The definition of XML syntax for a JSP page has been finalized. So a JSP 1.2 compliant container must accept files in both JSP 1.1 format and the new XML format called as JSP Document.
- Tag libraries can make use of Servlet 2.3 event listeners.
- A new type of validation has been added, for the tag libraries, which validates JSP pages.
- New options for tag library distribution and deployment have been added.

These changes are basically enhancements and are not required to be made, while migrating JSP pages from JSP API 1.1 to 1.2.

The implementation of JSP custom tag libraries in Sun ONE Application Server 6.0 and 6.5 complies with the J2EE specification. Consequently, migration of JSP custom tag libraries to Sun ONE Application Server 7 does not pose any particular problem, nor require any modifications to be made.

Migrating Servlets

Sun ONE Application Server 6.0 and 6.5 support the Servlet 2.2 API whereas Sun ONE Application Server 7, supports the Servlet 2.3 API.

Servlet API 2.3 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 2.3.

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 Sun ONE Application Server 7 as it uses new class loader hierarchy as compared to Sun ONE Application Server 6.0/6.5. 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 Appendix C.

EJB Migration

As mentioned in "About Sun ONE Application Server 7", while Sun ONE Application Server 6.0 and 6.5 support the EJB 1.1 specification, Sun ONE Application Server 7 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 Sun ONE Application Server 7**, the use of the EJB 2.0 architecture is recommended to leverage its enhanced capabilities.

To migrate EJB 1.1 to EJB 2.0, please refer to "Appendix C".

EJB Changes Specific to Sun ONE Application Server 7

Migrating EJB's from Sun ONE Application server 6.0/6.5 to Sun ONE Application Server 7 would not require any changes in the EJB code. The following DTD changes are required.

Session Beans:

- 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` with modified version of this file, named `sun-ejb-jar.xml` created manually according to the DDs. See url

```
<!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7 EJB 2.0//EN'
'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2
_0-0.dtd'>
```

for details.

- In `sun-ejb-jar.xml`, the JNDI name for all the EJB's, 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 `ejb-jar.xml`. In Sun ONE Application Server 7 this has changed as a new tag has been introduced in `sun-ejb-jar.xml` inside which the JNDI name of the EJB can be declared. Because of this flexibility provided by Sun ONE Application Server 7 we advice that the JNDI name of the EJB should be declared as "ejb/<ejb-name>" inside the <jndi-name> tag to avoid changing JNDI names throughout the application.

Entity Beans:

- The <!DOCTYPE definition should be modified to point to the latest DTD url in case of J2EE standard DDs, like `ejb-jar.xml`.
- Insert <cmp-version> tag with value 1.1 for all CMPs in `ejb-jar.xml`.
- Replace all the <ejb-name>-ias-cmp.xml files with one `sun-cmp-mappings.xml` file, which is created manually. See url

```
<!DOCTYPE sun-cmp-mappings PUBLIC "-//Sun Microsystems, Inc.//DTD  
Sun ONE Application Server 7 OR Mapping //EN"  
'http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mappi  
ng_1_0.dtd'>
```

for details.

- Generate `dbschema` by using the *capture-schema* utility in the Sun ONE Application Server 7 installation's bin directory and place it above `META-INF` folder for the Entity beans.
- `ias-ejb-jar.xml` should be replaced with its new version named `sun-ejb.jar.xml` in Sun ONE Application Server 7.
- In Sun ONE Application Server 6.5, the `finders.sql` was directly embedded inside the <ejb-name>-ias-cmp.xml, in Sun ONE Application Server 7 this has changed such that now mathematical expressions are used to declare the <query-filter> for the various finder methods.

Migrating Web Applications

Sun ONE Application Server 6.0 and 6.5 support servlets (Servlet API 2.2), and JSPs (JSP 1.1). Sun ONE Application Server 7 on the other hand supports servlets (Servlet API 2.3) and JSPs (JSP 1.2).

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 1.3 specification, a Web application is an archive file (.WAR file) with the following structure:

- a root directory containing the HTML pages, JSP pages, 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 6.0/6.5 to Sun ONE Application Server 7 would not require any changes in the Java/JSP code. The following changes are, however, still required.

- **web.xml**

Sun ONE Application Server 7 adheres to J2EE 1.3 standards, according to which, the `web.xml` file inside a WAR should adhere to the revised DTD available at 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`, 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 ONE Application Server 7 the name of this file is changed to `sun-web.xml`.

This XML file is required to declare the Sun ONE Application Server 7 specific properties/resources that will be required by the web application.

Note: See the next section for some 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 Sun ONE Application Server 7 standards. The file name has to be changed to `sun-web.xml` and other details are available at

```
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7 Servlet 2.3//EN"
'http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2
_3-0.dtd'>
```

Once the `web.xml` and `ias-web.xml` are migrated in the above-mentioned fashion, the Web application (.WAR archive) can be deployed from the Sun ONE Application Server 7's GUI interface of the admin server or from the command line utility ***asadmin***, where the deployment command should mention the ***type of application as web***.

The command line utility `asadmin` can be invoked by running `asadmin.bat` file kept at Sun ONE Application Server 7 installation's bin directory.

The command at *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] [--instance instancename]
filepath
```

Deployment can also be done from the Sun ONE Studio development environment as explained in section "Deploying an application in Sun ONE Application Server 7".

Particular setbacks when migrating servlets and JSPs

The actual migration of the components of a Servlet / JSP application from Sun ONE Application Server 6.0/6.5 to Sun ONE Application Server 7 will not require any modifications to be made to the component code.

In case if the web-application is using a server resource, for example, a `DataSource`, then Sun ONE Application Server 7 requires that this resource be declared inside the `web.xml` and correspondingly inside `sun-web.xml`. For declaring a `DataSource` called `jdbc/iBank`, the `<resource-ref>` tag as declared inside the `web.xml` would look like this:

```
<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` 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 6.0 and 6.5 support the EJB 1.1 API whereas Sun ONE Application Server 7 supports the EJB 2.0 API. Thereby, both can support:

- Stateful or Stateless Session Beans.
- Entity beans with bean managed persistence (BMP), or container managed persistence (CMP).

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

J2EE 1.3 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 6.0 or 6.5 and Sun ONE Application Server 7 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` with modified version of this file, i.e., `sun-ejb-jar.xml` created manually according to the DTDs. See url

```
<!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7 EJB 2.0//EN'
'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. See url

```
<!DOCTYPE sun-cmp-mappings PUBLIC '-//Sun Microsystems, Inc.//DTD
Sun ONE Application Server 7 OR Mapping //EN'
'http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mappi
ng_1_0.dtd'>
```

- Only for CMP entity beans: Generate `dbschema` by using the `capture-schema` utility in the Sun ONE Application Server 7 installation's bin directory and place it above `META-INF` folder 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 6.0/6.5 and 7 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.

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

- Package EJBs in one or more EJB modules,
- Package the components of the Web application in a Web module,
- Assemble the EJB modules and Web modules in an enterprise application module
- Define the name of the enterprise application's root context, which will determine the URL for accessing the application.

Note: Sun ONE Application Server 7 uses a new class loader hierarchy as compared to Sun ONE Application Server 6.0/6.5. In the new scheme of things, 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. Hence all classes loaded by the JAR class loader are available/ accessible to the WAR module but the reverse is not true. Hence, suppose there is a certain class which is required by the JAR as well as the WAR, then it should be packaged inside the JAR module only. If this guideline is not followed it would lead to class conflicts hence ClassCastException.

Application root context and access URL

There is one particular difference between Sun ONE Application Server 6.0/6.5 and Sun ONE Application Server 7, 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, then the access URL for this application will differ depending on the application server used:

- With Sun ONE Application Server 6.0 or 6.5, 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 Sun ONE Application Server 7, the URL will take the form:

```
http://hostname:port/AppName/
```

The TCP port used as default by Sun ONE Application Server 7 is port 80.

Although the difference in access URLs between Sun ONE Application Server 6.0/6.5 and Sun ONE Application Server 7 may appear minor, it can however be problematical 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 6.0/6.5.

Migrating Proprietary Extensions

A number of classes proprietary to the Sun ONE Application Server 6.0/ 6.5 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.ipplanet.server.servlet.extension
- com.kivasoft.dlm
- com.ipplanetiplanet.server.jdbc
- com.kivasoft.util
- com.netscape.server.servlet.extension
- com.kivasoft
- com.netscape.server

These APIs are not supported in Sun ONE Application Server 7. Applications using any classes belonging to the above package will have to be re written such that the applications use standard J2EE APIs. Applications using Custom JSP tags and UIF framework also needs to be rewritten to use standard J2EE API.

Migrating Example: iBank

In this section we describe the process for migrating the main components of a typical J2EE application from Sun ONE Application Server 6.0 and 6.5 to Sun ONE Application Server 7. For each aspect we highlight any problems posed by migration, and suggest practical solutions to overcome these.

For this migration process, the J2EE application presented is called 'iBank' and is based on the actual migration of the iBank application from the Sun ONE Application Server 6.0 and 6.5 versions to Sun ONE Application Server 7. 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*.

The iBank Application can be migrated to Sun ONE Application Server 7 by manually changing the deployment descriptors or using Sun ONE Studio or using Sun ONE Migration Tool. The recommended process among the above three is the Sun ONE Migration Tool. If the migration has to be carried out without converting CMP's to 2.0, then follow the section "Manual Migration of iBank Application" or use Sun ONE Migration Tool.

In this guide the Manual Migration process and the migration using Sun ONE Studio are discussed. The Automatic migration procedure, using Sun ONE Migration Tool for iBank example, is discussed in the documentation provided with the Migration Tool itself.

Manual Migration of iBank Application

The manual migration does not require any major changes in the source code as Sun ONE Application Server 7 supports CMP 1.1. However manual migration of the application would require a few changes to be made in the following aspects:

Web application changes

Migrating iBank from Sun ONE Application server 6.0/6.5 to Sun ONE Application Server 7 would not require any changes in the web application part of the iBank application. Delete the `ias-web.xml` file from the source directory, as there is no information in this file that can go inside its counterpart in the Sun ONE Application Server 7 Deployment descriptor, the `sun-web.xml` file. The `web.xml` requires no changes.

However, generically speaking, if there is some information inside the `web.xml` that needs to be mapped to the Server specific resources, then a declaration in `sun-web.xml` would have been required in that case. For example, if the `web.xml` file had declared a `javax.sql.DataSource` type resource reference, it would be mandatory to map it to the JNDI name of the actual `DataSource` on the Sever, inside the `sun-web.xml`.

The migrator needs to create the new `sun-web.xml`. The creation process is outlined below:

1. Create a new XML file which has the following DOCTYPE definition on top:

```
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7 Servlet 2.3//EN"
'http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2
_3-0.dtd'>
```

Save this file as “`sun-web.xml`”.

2. The root tag of this XML file, as evident from the DOCTYPE definition, is `sun-web`. In the DTD, this element is defined as

```
<!ELEMENT sun-web-app (security-role-mapping*, servlet*,
session-config?, resource-env-ref*, resource-ref*, ejb-ref*,
cache?, class-loader?, jsp-config?, locale-charset-info?,
property*)>
```

From the above declaration it is clear that all tags are optional so a default `sun-web.xml` would look something like:

```
<!DOCTYPE sun-web-app SYSTEM
"http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2
_3-0.dtd">
</sun-web-app>
```

3. For declaring any resource references, the element declaration would be:

```
<!ELEMENT resource-ref (res-ref-name, jndi-name, default-resource-principal?)>
where the sub elements are:
```

```
<!ELEMENT res-ref-name (#PCDATA)>
<!ELEMENT default-resource-principal (name, password)>
<!ELEMENT jndi-name (#PCDATA)>
```

In case of `ibank`, resource reference details, `sun-web.xml` would be:

```
<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>
```

EJB Changes

Migrating `iBank` from Sun ONE Application server 6.5 to Sun ONE Application Server 7 would not require any changes in the EJB code.

Session Beans:

In `ejb-jar.xml`: The `<!DOCTYPE` definition should be modified to point to the latest DTD url in case of `ejb-jar.xml`. This new definition should look like this:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" 'http://java.sun.com/dtd/ebj-jar_2_0.dtd'>
```

In `ias-ebj-jar.xml`: The `ias-ebj-jar.xml` in Sun ONE Application server 6.5 has been replaced by `sun-ebj-jar.xml` in Sun ONE Application server 7. Since the DTDs for these two XML files are radically different, the migrator needs to create the new `sun-ebj-jar.xml` by extracting relevant information from the `ejb-jar.xml` and `ias-ebj-jar.xml`. The creation process is outlined below:

1. Create a new XML file which has the following DOCTYPE definition on top:

```
<!DOCTYPE sun-ebj-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7 EJB 2.0//EN"
'http://www.sun.com/software/sunone/appserver/dtds/sun-ebj-jar_2
_0-0.dtd'>
```

Save this file as “`sun-ebj-jar.xml`”, along with the modified `ejb-jar.xml`.

2. The root tag of this XML file, as evident from the DOCTYPE definition, is `sun-ejb-jar`. In the DTD, this element is defined as

```
<!ELEMENT sun-ejb-jar (security-role-mapping*,enterprise-beans)>
```

The `security-role-mapping` tag is meant for mapping the security roles declared in the `ejb-jar.xml`. As in the iBank application, there is no security declared in the `ejb-jar.xml` file, we will skip the `security-role-mapping` optional tag and focus on the `enterprise-beans` tag. Right now, the `sun-ejb-jar.xml` file should look like.

```
<sun-ejb-jar>
  <enterprise-beans>
</enterprise-beans>
</sun-ejb-jar>
```

NOTE: We have not included the header part of the document, namely the XML declaration and DOCTYPE definition, here for brevity.

3. The *enterprise-beans* element is defined in the DTD as

```
<!ELEMENT enterprise-beans (name?, unique-id?, ejb*,
pm-descriptors?, cmp-resource?)>
```

The optional *name* element should contain the canonical name of the *enterprise-beans*. You may give it some name. We will skip this tag.

The *unique-id* element is used by the Sun ONE Application Server and is *inserted by the Application Server automatically at the time of application deployment*. We will skip this tag.

The *EJB* element is the tag in which we are interested. This is the element describing runtime bindings for a single EJB. It is defined in the DTD as

```
<!ELEMENT ejb (ejb-name, jndi-name?, ejb-ref*, resource-ref*,
resource-env-ref*, pass-by-reference?, cmp?, principal?,
mdb-connection-factory?, jms-durable-subscription-name?,
jms-max-messages-load?, ior-security-config?,
is-read-only-bean?, refresh-period-in-seconds?, commit-option?,
gen-classes?, bean-pool?, bean-cache?)>
```

In our case, the *ejb* element will contain the *ejb-name* element. The *ejb-name* element will contain the canonical name of the EJB. This name should be the same as declared inside the *ejb-name* element of the `ejb-jar.xml` for that EJB. It will also contain the *jndi-name* of the EJB. One of the differences between Sun

SUN ONE Application Server 6.5 and 7 is the flexibility of the latter in providing freedom to the bean developer to have different *ejb-name* and *jndi-name* of an EJB. In Sun ONE Application Server 6.5, the *jndi name* of an EJB by default was `ejb/<ejb-name>`.

To allow for smooth migration, we need to keep the *jndi-names* of the EJB and all other resources to be same as they were on Sun ONE Application Server 6.5. Hence, we declare the *ejb-name* of all the EJBs' to be `ejb/<ejb-name>`.

Using the logic described above, the `sun-ejb-jar.xml` now should look like

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>BankTeller</ejb-name>
      <jndi-name>ejb/BankTeller</jndi-name>
    </ejb>
    <ejb>
      <ejb-name>InterestCalculator</ejb-name>
      <jndi-name>ejb/InterestCalculator</jndi-name>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

4. For each `<ejb-ref>` element inside the `ejb-jar.xml`, there should be a corresponding `<ejb-ref>` element inside the `sun-ejb-jar.xml`. The `<ejb-ref>` element inside the `ejb-jar.xml` is used to declare all the EJBs referenced from inside the bean class of that EJB. While the bean class code will reference the EJB by using its `<ejb-ref-name>`, this `<ejb-ref-name>` has to be mapped to the actual `<jndi-name>` of the bean on the Application Server. Hence, this serves as a mechanism to add a layer of abstraction between the name referenced by the EJB implementation and the actual JNDI name of the bean.

Using the logic explained above, let us examine the BankTeller EJB. In the `ejb-jar.xml`, there are two `<ejb-ref>` declarations inside this EJB. The first one is for the Customer EJB (an entity bean in the Entity Bean module). As we have explained in point #3 above, the JNDI names of all EJBs will be kept as `ejb/<ejb-name>`, we will add this declaration inside the `sun-ejb-jar.xml`

```

<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>BankTeller</ejb-name>
      <jndi-name>ejb/BankTeller</jndi-name>
      <ejb-ref>
        <ejb-ref-name>Customer</ejb-ref-name>
        <jndi-name>ejb/Customer</jndi-name>
      </ejb-ref>
    </ejb>
    <ejb>
      <ejb-name>InterestCalculator</ejb-name>
      <jndi-name>ejb/InterestCalculator</jndi-name>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>

```

Similarly, we will add a similar `<ejb-ref>` tag for Account EJB. Since the InterestCalculator bean does not have a `<ejb-ref>` tag inside the `ejb-jar.xml`, it is not required inside the `sun-ejb-jar.xml` also. By now, the `sun-ejb-jar.xml` should look like this

```

<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>BankTeller</ejb-name>
      <jndi-name>ejb/BankTeller</jndi-name>
      <ejb-ref>
        <ejb-ref-name>Customer</ejb-ref-name>
        <jndi-name>ejb/Customer</jndi-name>
      </ejb-ref>
    <ejb-ref>
      <ejb-ref-name>Account</ejb-ref-name>
    </ejb-ref>
  </enterprise-beans>
</sun-ejb-jar>

```

```

        <jndi-name>ejb/Account</jndi-name>
    </ejb-ref>
</ejb>
<ejb>
    <ejb-name>InterestCalculator</ejb-name>
    <jndi-name>ejb/InterestCalculator</jndi-name>
</ejb>
</enterprise-beans>
</sun-ejb-jar>

```

5. The `ejb` element would contain element `pass-by-reference` `<!ELEMENT pass-by-reference (#PCData)`.

`pass-by-reference` element controls use of Pass by Reference semantics. The EJB specification requires pass by value, which will be the default mode of operation. This can be set to true for non-compliant operation and possibly higher performance. It can apply to all the enclosed EJB modules. Allowed values are true and false. Default will be false.

6. The `ejb` element would also have element `bean-cache`.

```

<!ELEMENT bean-cache (max-cache-size?,
is-cache-overflow-allowed?, cache-idle-timeout-in-seconds?,
removal-timeout-in-seconds?, victim-selection-policy?)>

```

This element is used only for stateful session beans and entity beans. In iBank, only *BankTeller* session bean would have this entry.

In this tag, *max-cache-size* defines the maximum number of beans in the cache. *cache-idle-timeout-in-seconds* specifies the maximum time that a stateful session bean or entity bean is allowed to be idle in the cache. After this time, the bean is passivated to backup store. This is a hint to server. Default value for *cache-idle-timeout-in-seconds* is 10 minutes.

The amount of time that the bean remains passivated (i.e. idle in the backup store) is controlled by *removal-timeout-in-seconds* parameter. Note that if a bean was not accessed beyond *removal-timeout-in-seconds*, then it will be removed from the backup store and hence will not be accessible to the client. The Default value for *removal-timeout-in-seconds* is 60min.

With the above entries, `sun-ejb-jar.xml` should look like this:

```

<sun-ejb-jar>
    <enterprise-beans>

```

```

<ejb>
  <ejb-name>BankTeller</ejb-name>
  <jndi-name>ejb/BankTeller</jndi-name>
  <ejb-ref>
    <ejb-ref-name>Customer</ejb-ref-name>
    <jndi-name>ejb/Customer</jndi-name>
  </ejb-ref>
  <ejb-ref>
    <ejb-ref-name>Account</ejb-ref-name>
    <jndi-name>ejb/Account</jndi-name>
  </ejb-ref>
  <pass-by-reference>false</pass-by-reference>
  <bean-cache>
    <cache-idle-timeout-in-seconds>
      0
    </cache-idle-timeout-in-seconds>
    <removal-timeout-in-seconds>
      0
    </removal-timeout-in-seconds>
  </bean-cache>
</ejb>
<ejb>
  <ejb-name>InterestCalculator</ejb-name>
  <jndi-name>ejb/InterestCalculator</jndi-name>
  <pass-by-reference>false</pass-by-reference>
</ejb>
</enterprise-beans>
</sun-ejb-jar>

```

7. The element used only for Stateless session bean and message-driven bean pools is bean-pool.


```
<!ELEMENT bean-pool (steady-pool-size?, resize-quantity?,
max-pool-size?, pool-idle-timeout-in-seconds?,
max-wait-time-in-millis?)>
```

steady-pool-size specified the initial and minimum number of beans that must be maintained in the pool.

resize-quantity specifies the number of beans to be created or deleted when the pool is being serviced by the pool manager.

max-pool-size specifies the maximum pool size. Valid values are from 0 to MAX_INTEGER.

max-pool-size specifies the maximum pool size.

pool-idle-timeout-in-seconds specifies the maximum time that a stateless session bean or message-driven bean is allowed to be idle in the pool.

Finally the `sun-ejb-jar.xml` would have the following shape:

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>BankTeller</ejb-name>
      <jndi-name>ejb/BankTeller</jndi-name>
      <ejb-ref>
        <ejb-ref-name>Customer</ejb-ref-name>
        <jndi-name>ejb/Customer</jndi-name>
      </ejb-ref>
      <ejb-ref>
        <ejb-ref-name>Account</ejb-ref-name>
        <jndi-name>ejb/Account</jndi-name>
      </ejb-ref>
      <pass-by-reference>false</pass-by-reference>
      <bean-cache>
        <cache-idle-timeout-in-seconds>
          0
        </cache-idle-timeout-in-seconds>
```

```

        <removal-timeout-in-seconds>
            0
        </removal-timeout-in-seconds>
    </bean-cache>
</ejb>
<ejb>
    <ejb-name>InterestCalculator</ejb-name>
    <jndi-name>ejb/InterestCalculator</jndi-name>
    <pass-by-reference>false</pass-by-reference>
    <bean-pool>
        <pool-idle-timeout-in-seconds>
            0
        </pool-idle-timeout-in-seconds>
    </bean-pool>
</ejb>
</enterprise-beans>
</sun-ejb-jar>

```

Entity Beans:

In `ejb-jar.xml`: The `<!DOCTYPE` definition should be modified to point to the latest DTD url in case of `ejb-jar.xml`. This new definition should look like this:

```

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN"
'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

```

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

Entry for entity bean would look like:

```

<entity>
    <description>Account CMP entity bean</description>
    <ejb-name>Account</ejb-name>
    <home>com.sun.bank.ejb.entity.AccountHome</home>

```

```

<remote>com.sun.bank.ejb.entity.Account</remote>
<ejb-class>com.sun.bank.ejb.entity.AccountEJB</ejb-class>
<persistence-type>Container</persistence-type>
<prim-key-class>
  com.sun.bank.ejb.entity.AccountPK
</prim-key-class>
<reentrant>False</reentrant>
<cmp-version>1.x</cmp-version>
<cmp-field>
  <field-name>branchCode</field-name></cmp-field>
<cmp-field>
  <field-name>accTypeId</field-name></cmp-field>
<cmp-field>
  <field-name>accBalance</field-name></cmp-field>
<cmp-field>
  <field-name>custNo</field-name></cmp-field>
<cmp-field>
  <field-name>accNo</field-name></cmp-field>
</entity>

```

similarly all the CMP beans would have this entry.

Similar to Session Beans, the `ias-ejb-jar.xml` in Sun ONE Application server 6.5 has been replaced by `sun-ejb-jar.xml` in Sun ONE Application server 7. Since the DTDs for this two XML files are radically different, the migrator needs to create the new `sun-ejb-jar.xml` by extracting relevant information from the `ejb-jar.xml` and `ias-ejb-jar.xml`. The creation process is outlined below:

1. Create a new XML file which has the following DOCTYPE definition on top:

```

<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7 EJB 2.0//EN"
'http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2
_0-0.dtd'>

```

Save this file as “`sun-ejb-jar.xml`”, along with the modified `ejb-jar.xml`.

2. The root tag of this XML file, as evident from the DOCTYPE definition, is `sun-ejb-jar`. In the DTD, this element is defined as

```
<!ELEMENT sun-ejb-jar (security-role-mapping*, enterprise-beans)
>
```

The `security-role-mapping` tag is meant for mapping the security roles declared in the `ejb-jar.xml`. As in the iBank application, there is no security declared in the `ejb-jar.xml` file, we will skip the `security-role-mapping` optional tag and focus on the `enterprise-beans` tag. Right now, the `sun-ejb-jar.xml` file should look like.

```
<sun-ejb-jar>
  <enterprise-beans>
  </enterprise-beans>
</sun-ejb-jar>
```

NOTE: We have not included the header part of the document, namely the XML declaration and DOCTYPE definition, here for brevity.

3. The `enterprise-beans` element is defined in the DTD as

```
<!ELEMENT enterprise-beans (name?, unique-id?, ejb*,
pm-descriptors?, cmp-resource?)>
```

The optional `name` element should contain the canonical name of the `enterprise-beans`. You may give it some name. We will skip this tag.

The `unique-id` element is used by the Sun ONE Application Server and is inserted by the Application Server automatically at the time of application deployment. We will skip this tag.

The `ejb` element is the tag in which we are interested. This is the element describing runtime bindings for a single EJB. It is defined in the DTD as

```
<!ELEMENT ejb (ejb-name, jndi-name?, ejb-ref*, resource-ref*,
resource-env-ref*, pass-by-reference?, cmp?, principal?,
mdb-connection-factory?, jms-durable-subscription-name?,
jms-max-messages-load?, ior-security-config?,
is-read-only-bean?, refresh-period-in-seconds?, commit-option?,
gen-classes?, bean-pool?, bean-cache?)>
```

In our case, the *ejb* element will contain the *ejb-name* element. The *ejb-name* element will contain the canonical name of the EJB. This name should be the same as declared inside the *ejb-name* element of the `ejb-jar.xml` for that EJB. It will also contain the *jndi-name* of the EJB. One of the differences between Sun ONE Application Server 6.5 and 7 is the flexibility of the latter in providing freedom to the bean developer to have different *ejb-name* and *jndi-name* of an EJB. In Sun ONE Application Server 6.5, the JNDI name of an EJB by default was `ejb/<ejb-name>`.

To allow for smooth migration, we need to keep the *jndi-names* of the EJB and all other resources to be same as they were on Sun ONE Application Server 6.5. Hence, we declare the *ejb-name* of all the *ejb*'s to be `ejb/<ejb-name>`.

Using the logic described above, the `sun-ejb-jar.xml` now should look like

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name> Account</ejb-name>
      <jndi-name> ejb/Account</jndi-name>
    </ejb>
    <ejb> --- </ejb>
    <ejb> --- </ejb>
                                other ejb's
    <ejb> --- </ejb>
    <ejb> --- </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

4. The *ejb* element would contain element `pass-by-reference` `<!ELEMENT pass-by-reference (#PCData)`.

pass-by-reference elements control use of Pass by Reference semantics. EJB spec requires pass by value, which will be the default mode of operation. This can be set to true for non-compliant operation and possibly higher performance. It can apply to all the enclosed EJB modules. Allowed values are true and false. Default will be false.

5. In case of CMP entity beans, element `cmp` is declared, which describes runtime information for a CMP EntityBean object for EJB1.1 and EJB2.0 beans.

```
<!ELEMENT cmp (mapping-properties?, is-one-one-cmp?,
one-one-finders?)>
```

In this *mapping-properties* contains the location of the persistence vendor specific O/R mapping file. *is-one-one-cmp* field is used to identify CMP 1.1 with old descriptors. This contains the boolean true if it is CMP 1.1. *one-one-finders* contains the finders for CMP 1.1.

This root element *finder* contains the finder for CMP 1.1 with a method-name and query parameters.

```
<!ELEMENT finder (method-name, query-params?, query-filter?,
query-variables?)>
```

Element *method-name* contains the method name for the query field. Element *query-params* contains the query parameters for CMP 1.1 finder.

query-filter is an optional element which contains the query filter for CMP 1.1 finder.

After making the above entries in iBank, `sun-ejb-jar` would look like:

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name> Account</ejb-name>
      <jndi-name> ejb/Account</jndi-name>
      <pass-by-reference>>false</pass-by-reference>
      <cmp>
        <mapping-properties>
          META-INF/sun-cmp-mappings.xml
        </mapping-properties>
        <is-one-one-cmp>>true</is-one-one-cmp>
        <one-one-finders>
          <finder>
            <method-name>
              findOrderedAccountsForCustomer
```

```

        </method-name>
        <query-params>int custNo</query-params>
        <query-filter>
            custNo == custNo
        </query-filter>
        </finder>
    </one-one-finders>
</cmp>
</ejb>
<ejb> --- </ejb>
<ejb> --- </ejb>
                                other ejb's
<ejb> --- </ejb>
<ejb> --- </ejb>
</enterprise-beans>
</sun-ejb-jar>

```

Account is the only entity bean having a finder other than primary key. So the finder entry shown above would only be in the case of *Account* bean.

6. The `<!ELEMENT commit-option (#PCDATA)>` specifies option for committing.
7. The `ejb` element would also have an element `bean-cache`.

```

<!ELEMENT bean-cache (max-cache-size?,
is-cache-overflow-allowed?, cache-idle-timeout-in-seconds?,
removal-timeout-in-seconds?, victim-selection-policy?)>

```

This element is used only for stateful session beans and entity beans. In this tag, *max-cache-size* defines the maximum number of beans in the cache. *cache-idle-timeout-in-seconds* specifies the maximum time that a stateful session bean or an entity bean is allowed to be idle in the cache. After this time, the bean is passivated to backup store. This is a hint to server. Default value for *cache-idle-timeout-in-seconds* is 10 minutes.

The amount of time that the bean remains passivated (i.e. idle in the backup store) is controlled by *removal-timeout-in-seconds* parameter. Note that if a bean was not accessed beyond *removal-timeout-in-seconds*, then it will be removed from the backup store and hence will not be accessible to the client. The Default value for *removal-timeout-in-seconds* is 60min.

With the above entries, `sun-ejb-jar.xml` should look like this:

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name> Account</ejb-name>
      <jndi-name> ejb/Account</jndi-name>
      <pass-by-reference>>false</pass-by-reference>
      <cmp>
        <mapping-properties>
          META-INF/sun-cmp-mappings.xml
        </mapping-properties>
        <is-one-one-cmp>>true</is-one-one-cmp>
        <one-one-finders>
          <finder>
            <method-name>
              findOrderedAccountsForCustomer
            </method-name>
            <query-params>int custNo</query-params>
            <query-filter>
              custNo == custNo
            </query-filter>
          </finder>
        </one-one-finders>
      </cmp>
      <commit-option>C</commit-option>
      <bean-cache>
```



```

        <max-cache-size>60</max-cache-size>
    <cache-idle-timeout-in-seconds>
        0
    </cache-idle-timeout-in-seconds>
</bean-cache>
</ejb>
<ejb> --- </ejb>
<ejb> --- </ejb>
                                other ejb's
<ejb> --- </ejb>
<ejb> --- </ejb>
</enterprise-beans>
</sun-ejb-jar>

```

8. In `<!ELEMENT enterprise-beans (name?, unique-id?, ejb*, pm-descriptors?, cmp-resource?)>`

Element *pm-descriptors* would be `<!ELEMENT pm-descriptors (pm-descriptor+, pm-inuse)>` Persistence Manager descriptors contain one or more pm descriptors, but only of them must be in use at any given time.

pm-descriptor describes the properties for the persistence manager associated with entity bean. *pm-identifier* element describes the vendor who provided the PM implementation. *pm-version* further specifies which version of PM vendor product to be used. *pm-config* specifies the vendor specific config file to be used. *pm-class-generator* specifies the vendor specific concrete class generator. This is the name of the class specific to a vendor. *pm-mapping-factory* specifies the vendor specific mapping factory. This is the name of the class specific to a vendor. *pm-insue* specifies whether this particular PM must be used or not.

Element *cmp-resource* contains the database to be used for storing CMP beans in an `ejb-jar`. `<!ELEMENT cmp-resource (jndi-name, default-resource-principal?)>`

Element *jndi-name* Specifies the JNDI name string. Element *default-resource-principal* has element name and password to be used when none are specified while accessing a resource.

```
<!ELEMENT default-resource-principal ( name, password)>
```

Finally `sun-ejb-jar.xml` would look like:

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name> Account</ejb-name>
      <jndi-name> ejb/Account</jndi-name>
      <pass-by-reference>>false</pass-by-reference>
      <cmp>
        <mapping-properties>
          META-INF/sun-cmp-mappings.xml
        </mapping-properties>
        <is-one-one-cmp>>true</is-one-one-cmp>
        <one-one-finders>
          <finder>
            <method-name>
              findOrderedAccountsForCustomer
            </method-name>
            <query-params>int custNo</query-params>
            <query-filter>
              custNo == custNo
            </query-filter>
          </finder>
        </one-one-finders>
      </cmp>
      <commit-option>C</commit-option>
      <bean-cache>
        <max-cache-size>60</max-cache-size>
        <cache-idle-timeout-in-seconds>
```

```

        0
        </cache-idle-timeout-in-seconds>
    </bean-cache>
</ejb>

<ejb> --- </ejb>
<ejb> --- </ejb>

                                other ejb's
<ejb> --- </ejb>
<ejb> --- </ejb>
<pm-descriptors>
    <pm-descriptor>
        <pm-identifier>IPLANET</pm-identifier>
        <pm-version>1.0</pm-version>
        <pm-class-generator>
            com.iplanet.ias.persistence.
                internal.ejb.ejbcode.JDCodeGenerator
        </pm-class-generator>
        <pm-mapping-factory>
            com.iplanet.ias.cmp.NullFactory
        </pm-mapping-factory>
    </pm-descriptor>
    <pm-inuse>
        <pm-identifier>IPLANET</pm-identifier>
        <pm-version>1.0</pm-version></pm-inuse>
</pm-descriptors>
<cmp-resource>
    <jndi-name>jdo/pmf</jndi-name>
</cmp-resource>
</enterprise-beans>
</sun-ejb-jar>

```

Generate dbschema by using the *capture-schema* utility in the Sun ONE Application Server 7 installation's bin directory. Execute capture-schema.bat file kept in bin directory and specify the valid inputs for the database URL, username, password and specify the tables for which schema has to be generated. By default, schema has to be generated for all the tables used by the application. In case of iBank, there are five tables for which schema has to be generated. Name this schema file as myschema.dbschema. The tables used in iBank are:

```
ACCOUNT
ACCOUNT_TYPE
BRANCH
CUSTOMER
TRANSACTION_HISTORY
TRANSACTION_TYPE
```

Place this file myschema.dbschema above META-INF folder for the Entity beans.

In <ejb-name>-ias-cmp.xml: Replace all the <ejb-name>-ias-cmp.xml files in Sun ONE Application Server 6.0/6.5 with one sun-cmp-mappings.xml file. This file maps (at least one) set of beans to tables and columns in a specific db schema. Since the DTDs for this two XML files are radically different, the migrator has to actually create a new file following the steps given below:

1. Create a new XML file which has the following DOCTYPE definition on top:

```
<!DOCTYPE sun-cmp-mappings PUBLIC "-//Sun Microsystems, Inc.//DTD
Sun ONE Application Server 7 OR Mapping //EN"
'http://www.sun.com/software/sunone/appserver/dtds/sun-cmp_mappi
ng_1_0.dtd'>
```

Save this file as "sun-cmp-mappings.xml".

2. The root tag of this XML file, as evident from the DOCTYPE definition, is sun-cmp-mappings. In the DTD, this element is defined as:

```
<!ELEMENT sun-cmp-mappings ( sun-cmp-mapping+ ) >
```

Element sun-cmp-mapping would be :

```
<!ELEMENT sun-cmp-mapping ( schema, entity-mapping+ ) >
```

Here element schema is the path name to the schema file.

A cmp bean has a name, a primary table, one or more fields, zero or more relationships, and zero or more secondary tables, plus flags for consistency checking. Element `entity-mapping` has following elements

```
<!ELEMENT entity-mapping (ejb-name, table-name,
cmp-field-mapping+, cmr-field-mapping*, secondary-table*,
consistency?)>
```

Element *ejb-name* is the EJB name from standard EJB-jar DTD. Element *table-name* is the name of the database table. A *cmp-field-mapping* has a field, one or more columns that it maps to *cmr-field mapping*. A *cmr* field has a name and one or more column pairs that define the relationship. Element *secondary-table* is for secondary table used. In case of iBank, no secondary table is used.

Right now, the `sun-cmp-mappings.xml` file with entries for Account entity bean should look like:

```
<sun-cmp-mapping>
  <schema>mySchema</schema>
  <entity-mapping>
    <ejb-name>Account</ejb-name>
    <table-name>ACCOUNT</table-name>
    <cmp-field-mapping>
      <field-name>custNo</field-name>
      <column-name>CUST_NO</column-name>
    </cmp-field-mapping>
    <cmp-field-mapping>
      <field-name>branchCode</field-name>
      <column-name>BRANCH_CODE</column-name>
    </cmp-field-mapping>
    <cmp-field-mapping>
      <field-name>accTypeId</field-name>
      <column-name>ACCTYPE_ID</column-name>
    </cmp-field-mapping>
    <cmp-field-mapping>
```

```
<field-name>accNo</field-name>
<column-name>ACC_NO</column-name>
</cmp-field-mapping>
<cmp-field-mapping>
  <field-name>accBalance</field-name>
  <column-name>ACC_BALANCE</column-name>
</cmp-field-mapping>
</entity-mapping>
</sun-cmp-mapping>
```

NOTE: We have not included the header part of the document, namely the XML declaration and DOCTYPE definition, here for brevity.

Entries for all the CMP entity beans have to be made.

The above changes can be referenced in file `iBankWithCMP1.1.zip` provided with this guide.

Assembling Application for Deployment

Sun ONE Application server 7 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.

So 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.

Deploying iBank application on Sun ONE Application Server 7 using the *asadmin* utility

The last stage is to deploy the application on an instance of Sun ONE Application Server 7. The process for deploying an application is described below:

The Sun ONE Application Server 7 *asadmin* 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 kept at Sun ONE Application Server 7 installation's bin directory. i.e., `<Install_dir>/AppServer7/appserv/bin`.

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

```
asadmin> deploy -u username -w password -H hostname -p adminport [--type
application | ejb | web | client | connector] [--contextroot contextroot] [--force=true]
[--name component-name] [--upload=true] [--instance instancename] filepath
```

Restart the server instance and then test the application on the browser by typing the url '*http://<machine_name>:<port_number>/iBank*'. Test by giving one of the available user name and password, say username as 'jatkings' and password as 'Monday'. This should show the main menu page of the iBank application.

Migrating iBank using Sun ONE Studio for Java 4.0

The sample application we defined is called 'iBank' and 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 major steps to be followed for migrating the iBank application would be as follows:

- The first and the foremost requirement of this migration is to install Sun ONE Application Server 7 and Sun ONE Studio.
- Extract the application, which is in a zip format in a local directory.

The source for the iBank application (*iBank65.zip*) can be found at the migration site <http://www.sun.com/migration/sunonetools.html>. Unzipping the file "*iBank65.zip*" would create following directory structure:

It would have 4 sub directories 'Docroot', 'SessionContent', 'EntityContent' and 'Scripts'.

- 'Docroot' would contain Html, Jsp's and Image files in its root. It would also contain the source files for servlets, EJBs, etc in the sub-folder WEB-INF\classes following the package structure com.sun.bank.*. War file would be generated through the contents of this directory.
- 'SessionContent' would contain the source code for Session beans following the package structure com.sun.bank.ejb.session. This directory would form the EJB module for session beans.
 - I. 'EntityContent' would contain the Entity beans following the package structure com.sun.bank.ejb.entity. This directory would form the EJB module for Entity beans.
- 'Scripts' contain the sql scripts for the database setup.
- Setup the schema for iBank application by executing the sql scripts provided in the 'Scripts' folder. These scripts are for oracle database. These scripts would create user, create tables and insert data into the tables. Execute the scripts in the following order
 - 01_iBank_CreateUser.sql
 - 02_iBank_CreateTables.sql
 - 03_iBank_InsertData.sql

Manual migration would involve following steps:

- a. Migrate Servlets, JSPs and JSP Custom tag libraries.
- b. Migrate Session Beans.
- c. Migrate Entity Beans.
- d. Migrate JDBC code.

These steps have to be carried out manually and is explained as and when required in the following sections. If migration tool is used as an option, it has to be carried out at this point itself. If manual approach is followed then changes have to be done as and when specified in following sections.

- Prepare Sun ONE Studio for assembling and deploying sample application 'iBank'

Sun ONE Studio can be invoked through the `runide.exe` file (`runide.sh` in case of Solaris) kept in the `<Sun ONE App Server ROOT>/<AppServ>/<SUN ONE STUDIO FOR JAVA_ROOT>/bin` directory.

(Note: Sun ONE Application Server 7 should be up and running before following the steps below)

- In the explorer window,
- Click at the Runtime tab
- Click 'Server Registry'
- Click 'installed servers'
- Choose Sun ONE Application Server.
- Setup admin server by right clicking at the Sun ONE Application Server and then selecting 'Add Admin Server'
- Enter details for host (local machine name), port number (by default its 4848), username and password.
- Once the admin server is setup, click on it to get the server instance installed.
- Set the server instance as default server by right clicking on the server instance and selecting option for setting it as default.
- Create web module by following the instructions given in "Creating a Web application module in Sun ONE Studio for Java".
- Migrate EJBs manually if migration tool is not used as an option for migrating the application. Follow the section on "EJB Migration", for the manual migration. This step can be carried out by opening the source code for the EJB's in Sun ONE Studio and modifying it.
- Migrate the JDBC code if migration tool is not used as an option to migrate the application by following the section on "Migrating JDBC Code".
- iBank application has Entity Beans with CMP 1.1, so they have to be converted to CMP 2.0 by following the manual procedure explained in the section on "Migrating CMP Entity EJBs" if the application is not migrated using the tool.
 If application is migrated through the tool, all the entity beans are migrated except one, i.e., 'Account' entity bean as it has Enumeration used in its code. The code for this has to be changed manually following the instructions given in section, "Migrating CMP Entity EJBs". Refer section, "Converting CMP Entity EJBs from 1.1 to 2.0" for an example of changes to be carried out for converting CMPs from 1.1 to 2.0.
- Create separate EJB modules for the Entity Beans and the Session Beans by following the instructions given in section, "Creating an EJB module in Sun ONE Studio for Java".

- Create Enterprise application by following the instructions given in section, "Creating an enterprise application in Sun ONE Studio for Java", which would include the web module as well as the EJB modules. The final output of this step would be .ear file, which can be deployed.
- Deploy .ear file on Sun ONE Application server 7 by following the instructions given in section, "Deploying an application in Sun ONE Application Server 7".

Creating a Web application module in Sun ONE Studio for Java

To create a Web application module in Sun ONE Studio for Java, follow the procedure below:

1. Mount the directory containing the source files i.e, 'Docroot' in the Sun ONE Studio for Java "FileSystems Explorer" window by right clicking at the Filesystem and choosing option for mount.
2. Create an empty directory, say 'WarContent' for the web module in the root directory structure containing the source files.
3. Mount the newly created directory 'WarContent' in the Sun ONE Studio for Java "FileSystems Explorer" window by right clicking at the Filesystem and choosing option for mount.
4. Mount the other directories containing the EJBs in the source file directory structure. i.e., 'EntityContent' and 'SessionContent'.
5. Convert the FileSystem (WarContent) into a Web Module by right clicking at the folder name and then selecting tools where there is an option for converting it into WebModule.
6. Copy the source JSP, HTML and image files to the web application root. i.e., to the directory 'WarContent' from the directory 'Docroot'.
7. Copy servlets and auxiliary class sources to the `WEB-INF/classes` directory. i.e., copy the sub folder 'com' in the 'Docroot' directory to the `WEB-INF/classes` directory of 'WarContent' directory.
8. Copy the tag library present in the `WEB-INF` of the 'Docroot' directory to the `WEB-INF` of 'WarContent' directory.
9. Edit the source code wherever required to migrate it to Sun ONE Application Server 7 (if it has not been modified through the migration tool), by following the steps below:
 - Figure out the JSPs that have to be changed.
 - Figure out if any custom JSP tags are used in the application.

- Open the selected JSP code in Sun ONE Studio by right clicking at the file and selecting option to open.
 - Follow the steps given in section "Migrating Java Server Pages and JSP Custom Tag Libraries" to modify the source.
 - Similarly migrate the servlets by following the details in section, "Migrating Servlets".
- 10.** Assemble the application and fill in the deployment descriptor `web.xml` (in the `WEB-INF/` directory). Click on the `web.xml` file and edit the properties of it, i.e. During this assembly phase, configure each servlet, JSP page and JSP tag library, as well as the EJB or data source references used in the Web application.

The following screen shots illustrate how this assembly phase is carried out using Sun ONE Studio for Java:

Configuring a Servlet

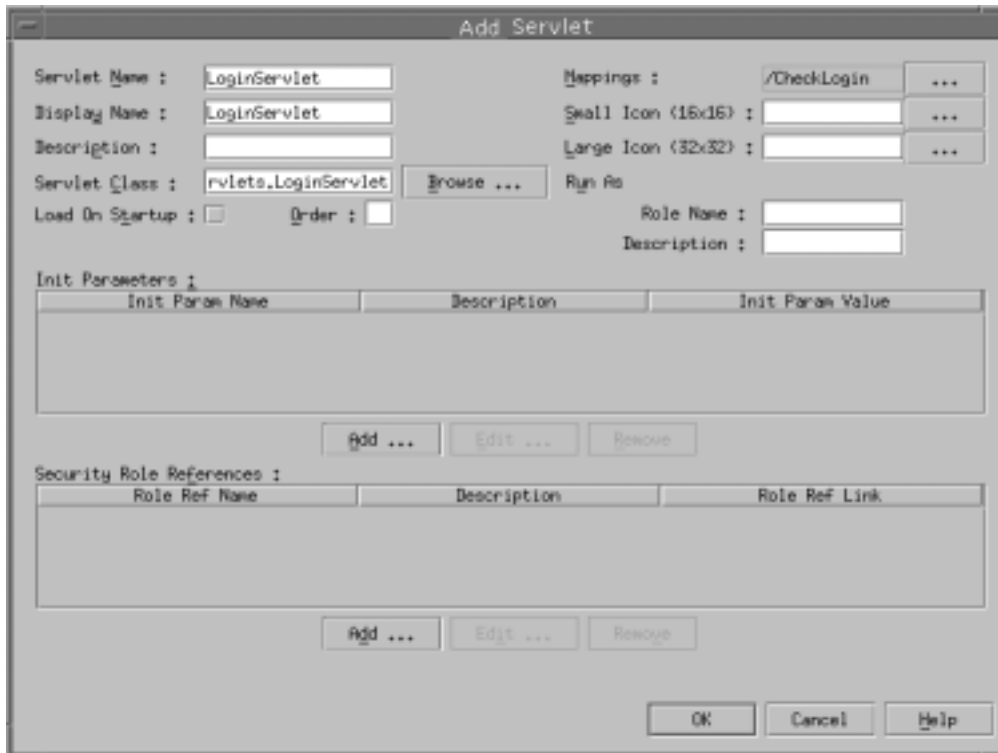
In the web module, click on `web` and then view the properties window.

Click at the *deployment* tab of the properties window of `web.xml`. Further click on the servlets for configuring servlets.

A property editor is displayed, click at 'Add' button to add new servlet.

For each servlet in the Web application, you specify the name of the servlet, the full name of its implementation class by clicking at the 'Browse' button, the mapping elements for the servlet by clicking at 'Mappings', and any initialization parameters.

Configuring Servlet



The list of servlets and their mappings in iBank application are:

Servlets and Mappings

Servlet Name	Display Name	Mapping
LoginServlet	LoginServlet	/CheckLogin
CheckTransferServlet	CheckTransferServlet	/CheckTransfer
CustomerProfileServlet	CustomerProfileServlet	/CustomerProfile
DataSourceTestServlet	DataSourceTestServlet	/DataSourceTest
HelloWorldServlet	HelloWorldServlet	/HelloWorld
LookUpDataSourceTestServlet	LookUpDataSourceTestServlet	/LookUpDataSourceTest

Servlets and Mappings

Servlet Name	Display Name	Mapping
ProjectEarningsServlet	ProjectEarningsServlet	/ProjectEarnings
ShowAccountSummaryServlet	ShowAccountSummaryServlet	/ShowAccountSummary
TestContextServlet	TestContextServlet	/TestContext
TransferFundsServlet	TransferFundsServlet	/TransferFunds
UpdateCustomerDetailsServlet	UpdateCustomerDetailsServlet	/UpdateCustomerDetails

All the above servlets have to be configured such that `web.xml` has entries for all of them.

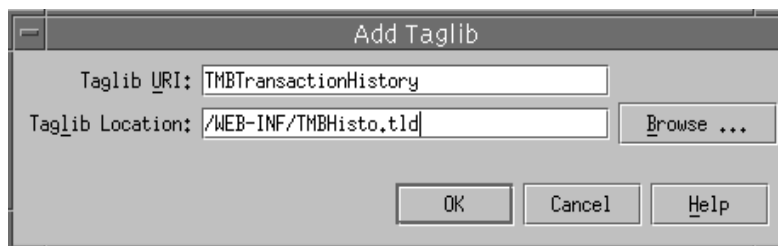
Finally the 'deployment' tab should show 11 servlets mappings and 11 servlets.

Configuring a JSP tag library

Click on the Deployment tab of the `web.xml` properties window. Click at the Tag Libraries to set the Tag lib.

To define a JSP tag library in the Web application deployment descriptor, specify the URI of the library (the identifier which the JSP pages will use to access it), and the path to the library's deployment descriptor (`.tld` file).

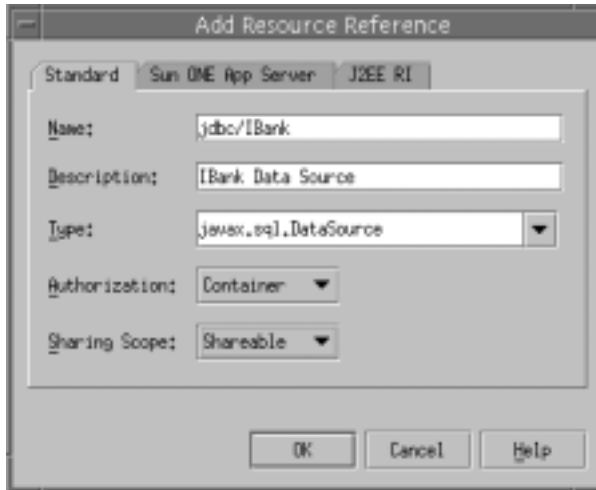
In iBank, there is one JSP Tag library `TMBHisto.tld`. The deployment descriptor is kept in `WEB-INF`. Following entries have to be made.



Configuring Tag lib

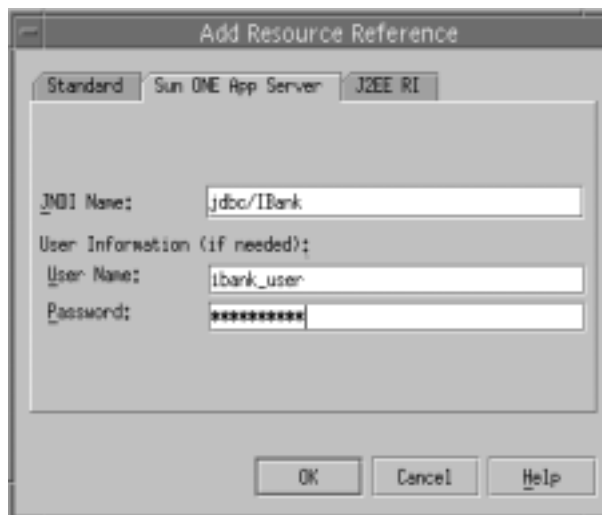
Add Resource Reference

Click at the references tab of the `web.xml` Properties window. Click at the Resource Reference to add a new resource. Following screen shot shows adding a new Resource for Data source in iBank i.e., `jdbc/iBank`



Adding Resource Reference

Click at the Sun ONE App Server tab and set the JNDI name as '`jdbc/iBank`' and also set the User name and Password depending on the database schema you are using.

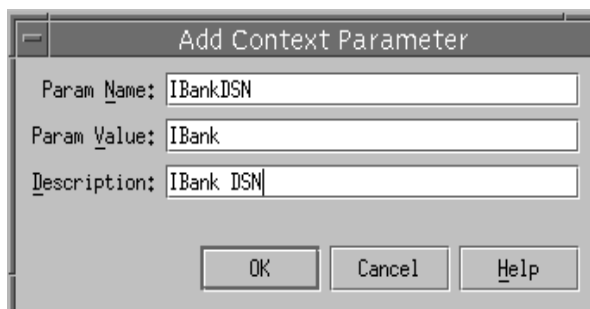


Adding Resource Reference entry for Sun ONE Application Server

Add Context Param

Add entry for context parameter for the JNDI name to lookup iBank data source.

Following screen shot shows the entry for context param, which can be done by clicking at the context param in the Properties window at the Deployment tab of web.xml.



Adding Context Parameter

Specify the Welcome File

Specify the welcome file in the properties window by clicking at the Welcome Files.

In case of iBank, `index.jsp` is the welcome file so that has to be mentioned.

Converting CMP Entity EJBs from 1.1 to 2.0

Referring to the manual process described in section, "Migrating CMP Entity EJBs", Here is an example of Account Entity bean being converted from CMP 1.1 to CMP 2.0.

The related files for Account bean are:

`Account.java`

`AccountEJB.java`

`AccountHome.java`

`AccountPK.java`

The details of the changes done are described below:

- **Account.java:**

There are no major changes in the code except for commenting out the setters for the primary key. The other setters are kept as it is.

Following is the code before modification:

```
public String getBranchCode()
    throws RemoteException;
public void setBranchCode(String branchCode)
    throws RemoteException;
public String getAccNo()
    throws RemoteException;
public void setAccNo(String accNo)
    throws RemoteException;
-----
-----
-----other getters and setters-----
```

After commenting the setters for the primary keys, i.e., `branchCode` and `accNo`, the same code would be:


```

public String getBranchCode()
    throws RemoteException;

/* public void setBranchCode(String branchCode)
    throws RemoteException; */

public String getAccNo()
    throws RemoteException;

/* public void setAccNo(String accNo)
    throws RemoteException; */
    -----
    -----
-----other getters and setters-----

```

- **AccountEJB.java :**

The changes incorporated in the bean class are as follows:

- Prepend the bean class declaration with the key word **abstract**.

Before modification:

```

public class AccountEJB implements EntityBean
{
--
--
}

```

After modification:

```
public abstract class AccountEJB implements EntityBean
{
--
--
}
```

- o Comment all the cmp fields and Prefix the accessor methods with the keyword abstract, thus the line of code in the methods would be commented and postfix the methods with a semicolon. Thus replace the given code below Before modification with the code given below After modification.

Before modification:

```
public String branchCode;
public String accNo;
public int custNo;
public String accTypeId;
public double accBalance;
public String accTypeDesc;
public double accTypeInterestRate;
private EntityContext context;

public String getBranchCode() {
    return(branchCode);
}

public void setBranchCode(String branchCode) {
    this.branchCode = branchCode;
}

public String getAccNo() {
    return(accNo);
}
```

```
public void setAccNo(String accNo) {
    this.accNo = accNo;
}

public int getCustNo() {
    return(custNo);
}

public void setCustNo(int custNo) {
    this.custNo = custNo;
}

public String getAccTypeId() {
    return(accTypeId);
}

public void setAccTypeId(String accTypeId) {
    this.accTypeId = accTypeId;
}

public BigDecimal getAccBalance() {
    return new BigDecimal(accBalance);
}

public void setAccBalance(BigDecimal accBalance) {
    this.accBalance = accBalance.doubleValue();
}
```

After modification:

```

private EntityContext context;
public abstract void setBranchCode(String branchCode);
public abstract String getBranchCode();
public abstract void setAccNo(String accNo);
public abstract String getAccNo();
public abstract void setCustNo(int custNo);
public abstract int getCustNo();
public abstract void setAccTypeId(String accTypeId);
public abstract String getAccTypeId();
public abstract void setAccBalance(BigDecimal accBalance);
public abstract BigDecimal getAccBalance();

```

- o **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)`'. Hence the code changes would be:**

Before modification :

```

public void setEntityContext(EntityContext ec) {
    context = ec;
}

public void unsetEntityContext() {
    this.context = null;
}

public void ejbActivate() {

    this.branchCode =
((com.sun.bank.ejb.entity.AccountPK)

```

```
context.getPrimaryKey()).branchCode;

        this.accNo = ((com.sun.bank.ejb.entity.AccountPK)
            context.getPrimaryKey()).accNo;
    }

    public void ejbPassivate() {
    }

    public void ejbLoad() {
    }

    public void ejbStore() {
    }

    public AccountPK ejbCreate(String branchCode,
        String accNo, int custNo, String accTypeId,
        BigDecimal accBalance) {
        this.branchCode = branchCode;
        this.accNo      = accNo;
        this.custNo     = custNo;
        this.accTypeId  = accTypeId;
        this.accBalance = accBalance.doubleValue();
        return null;
    }

    public void ejbPostCreate(String branchCode,
        String accNo, int custNo, String accTypeId,
        BigDecimal accBalance) {
    }
}
```

```
public void ejbRemove() {  
    }  
}
```

After modification:

```
public void setEntityContext(EntityContext ec) {  
    context = ec;  
}  
  
public void unsetEntityContext() {  
    this.context = null;  
}  
  
public void ejbActivate() {  
}  
  
public void ejbPassivate() {  
}  
  
public void ejbLoad() {  
}  
  
public void ejbStore() {  
}  
  
public AccountPK ejbCreate(String branchCode,  
    String accNo, int custNo, String accTypeId,  
    BigDecimal accBalance) {  
    setBranchCode(branchCode);  
    setAccNo(accNo);  
}
```

```

        setCustNo(custNo);
        setAccTypeId(accTypeId);
        setAccBalance(accBalance);
        return null;
    }

    public void ejbPostCreate(String branchCode,
        String accNo, int custNo, String accTypeId,
        BigDecimal accBalance) {
    }

    public void ejbRemove() {
    }

```

- **AccountPK.java**

No changes required in this file.

- **AccountHome.java**

In the home interface of the bean, changes are required to be made only if the return type of any finder methods is of type `java.util.Enumeration`. In case of Account bean, the home interface has a finder `findOrderedAccountsForCustomer` which has a return type as `Enumeration`. In such cases, the return type has to be changed to `Collection` and also the code affected by this change, i.e, the code in the session bean which uses this finder method has to be changed such that it has provision to accept the result of this finder method in a `Collection`.

The changes done in the home interface is shown below:

Before Modification:

```

public interface AccountHome extends EJBHome
{
    public Account findByPrimaryKey(AccountPK key)
        throws FinderException, RemoteException;
}

```

```

        public Enumeration findOrderedAccountsForCustomer(int
        custNo)
            throws FinderException, RemoteException;
    }

```

After Modification:

```

public interface AccountHome extends EJBHome
{
    public Account findByPrimaryKey(AccountPK key)
        throws FinderException, RemoteException;

    public Collection findOrderedAccountsForCustomer(int
    custNo)
        throws FinderException, RemoteException;
}

```

Due to the above changes, Session bean BankTeller which accesses this finder method also needs to incorporate changes to accept the result of the finder method in a Collection.

Following code snippet shows the changes made to the BankTellerEJB.java

Consider method `getAccountSummary` which uses finder method *findOrderedAccountsForCustomer*

Before modification:

```

public AccountSummary getAccountSummary()
    throws EJBException
{
    int custNo          = 0;
    Enumeration accEnum = null;
    AccountSummary accSum = new AccountSummary();
    -----
}

```



```

----
try
{
    AccountHome home=(AccountHome) PortableRemoteObject.
        narrow(accHomeHandle.getEJBHome(),
AccountHome.class);

    AccountTypeHome accTypeHome = (AccountTypeHome)
PortableRemoteObject.narrow(accTypeHomeHandle.getEJBHome(),
AccountTypeHome.class);

    accEnum = (Enumeration) home.
        findOrderedAccountsForCustomer(this.custNo);
    AccountTypePK accTypePK = new AccountTypePK();
    Account    accRef      = null;
    AccountType accTypeRef = null;
    String      accTypeDesc = null;
    int i = 0;
    while(accEnum.hasMoreElements())
    {
        i++;
        accRef = (Account) accEnum.nextElement();
        accTypePK.accTypeId = accRef.getAccTypeId();
        accTypeRef = (AccountType) PortableRemoteObject.
narrow(accTypeHome.findByPrimaryKey(accTypePK),
        AccountType.class);
        accTypeDesc = accTypeRef.getAccTypeDesc();
        accSum.addElement(
            accRef.getBranchCode(),
            accRef.getAccNo(),
            accRef.getAccBalance(),
            accTypeDesc

```

```

        );
    }
}
-----
----
}

```

After Modification:

```

public AccountSummary getAccountSummary()
throws EJBException
{
    int custNo          = 0;
    //Enumeration accEnum = null;
    Collection accEnum = null;
    AccountSummary accSum = new AccountSummary();
    ---
    ---

    try
    {
        AccountHome home = (AccountHome) PortableRemoteObject.
        narrow(accHomeHandle.getEJBHome(), AccountHome.class);
        AccountTypeHome accTypeHome = (AccountTypeHome)
        PortableRemoteObject.narrow(accTypeHomeHandle.
        GetEJBHome(), AccountTypeHome.class);
        // accEnum = (Enumeration) home.
        // findOrderedAccountsForCustomer(this.custNo);
        accEnum = (Collection) home.
        findOrderedAccountsForCustomer(this.custNo);
        AccountTypePK accTypePK = new AccountTypePK();

```

```

Account      accRef      = null;
AccountType  accTypeRef  = null;
String       accTypeDesc = null;
int i = 0;

Iterator iterator = accEnum.iterator();
// while(accEnum.hasMoreElements())
while(iterator.hasNext())
{
    i++;
    // accRef = (Account) accEnum.nextElement();
accRef = (Account) PortableRemoteObject.
        narrow(iterator.next(), Account.class);
accTypePK.accTypeId = accRef.getAccTypeId();
accTypeRef = (AccountType) PortableRemoteObject.

narrow(accTypeHome.findByPrimaryKey(accTypePK),
        AccountType.class);
accTypeDesc = accTypeRef.getAccTypeDesc();
accSum.addElement(
        accRef.getBranchCode(),
        accRef.getAccNo(),
        accRef.getAccBalance(),
        accTypeDesc
);
}
}
-----
-----
}

```

Creating an EJB module in Sun ONE Studio for Java

The procedure described below explains how to create an EJB module in Sun ONE Studio for Java, using existing source files:

Creating Module for Session Beans

1. Directory for Session Beans 'SessionContent' has following in it.

There would be bean class and interfaces for the following Session Beans:

BankTeller

InterestCalculator

In addition to this, it will also contain Exception classes.

2. Create the new EJBs from existing source files.

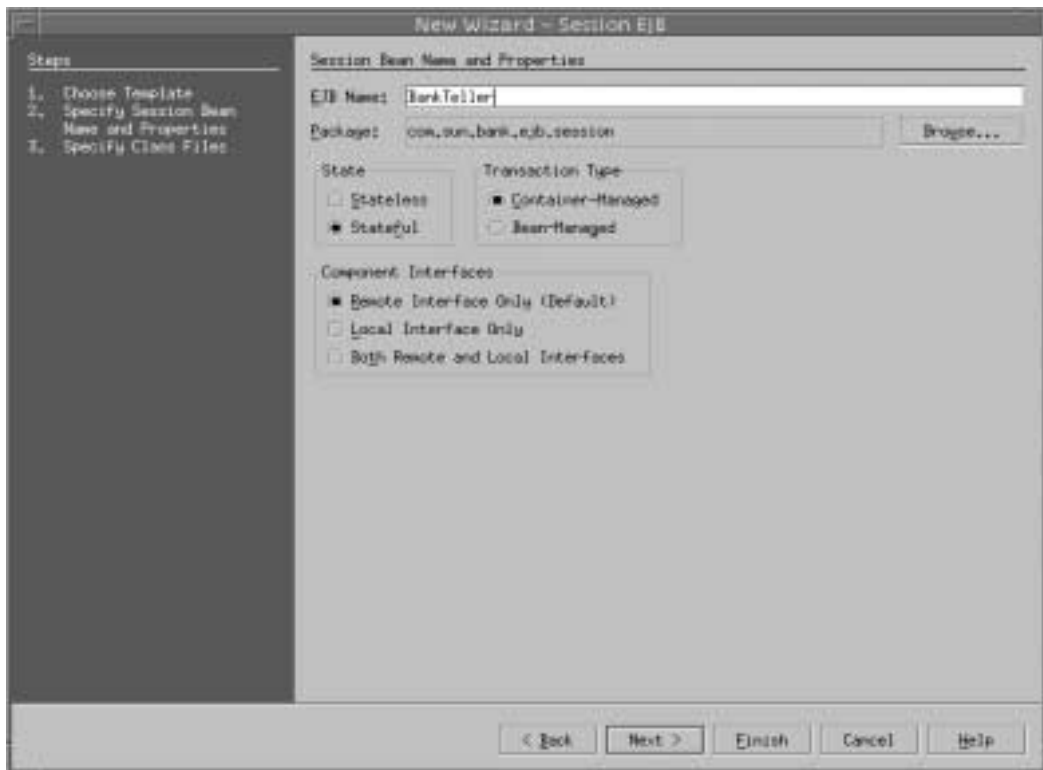
In Sun ONE Studio for Java, it is possible to create an EJB from existing source files.

Select mounted directory 'SessionContent', walk through the sub folders there to finally reach the package 'session', right click here and select option for new J2EE and finally click at 'Session EJB', which shows a new EJB wizard.

After specifying the main characteristics of the EJB (i.e., session, stateful or stateless), and defining the name and package for the EJB, you match the existing source files and the different components of the EJB: implementation class, home and remote interfaces. In order to make the match with existing source files, use the "Modify" button in the dialog box and select "Select an existing source file."

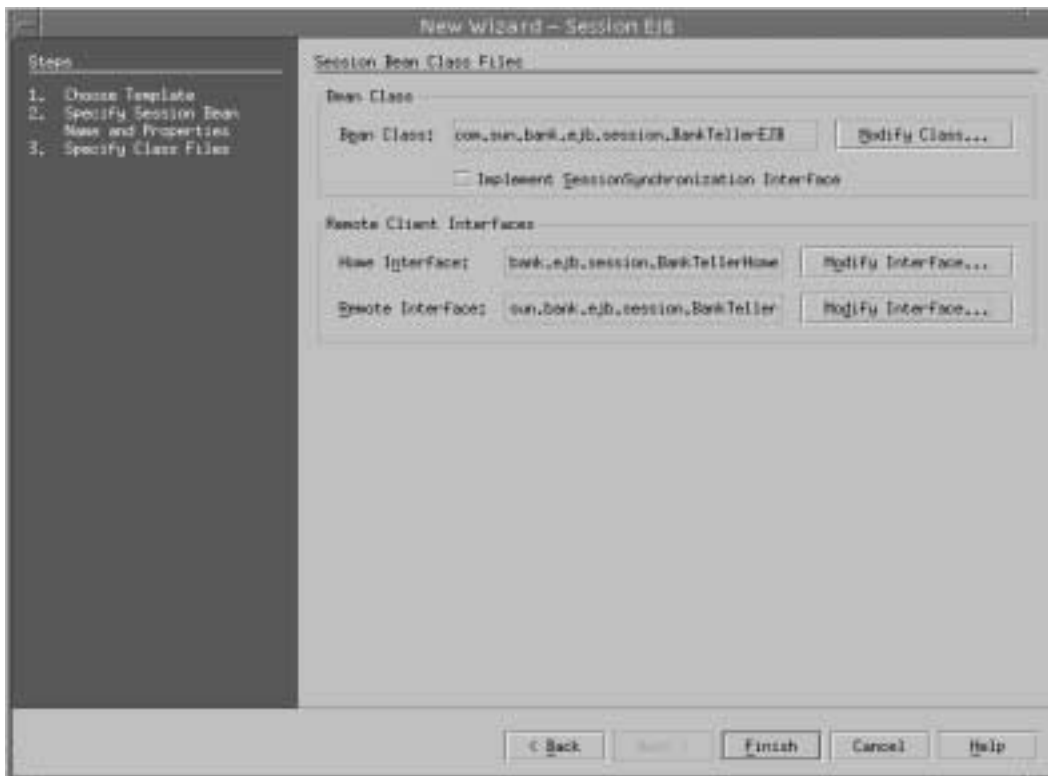
All the session beans have to be created in similar fashion.

Following screen shot shows the creation of Session Bean *BankTeller* which is a Stateful Session bean. So the State specified should be Stateful whereas *InterestCalculator* session bean is Stateless, so while creating *InterestCalculator* bean, the state specified should be stateless. Click at the browse button to specify the package.



Creation of new Session Bean

Following screen shot (*click 'Next>' when you are done*) shows specifying the bean class, the home interface and the remote interface. Clicking on the modify button and selecting option for using existing class would show up the existing files, which can be selected.

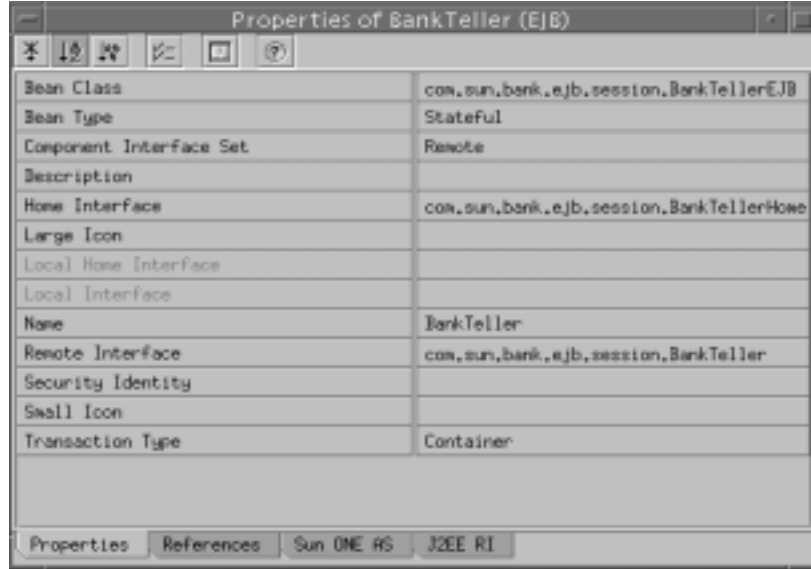


Specifying the Bean class, Home Interface and the Remote Interface

Create the *InterestCalculator* session bean in similar fashion.

3. Edit the properties of the EJBs

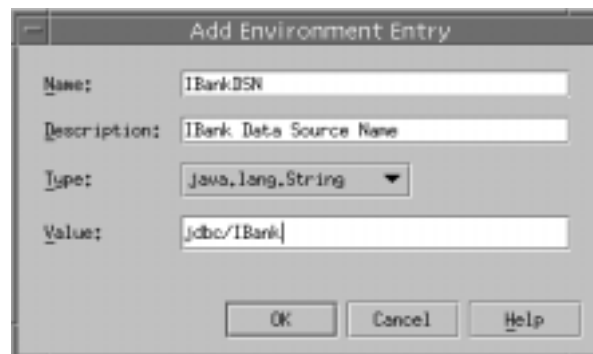
By editing the properties of an EJB, you can declare the EJB Resource references; specify an EJB's environment entries.



Properties window of the Session Bean BankTeller

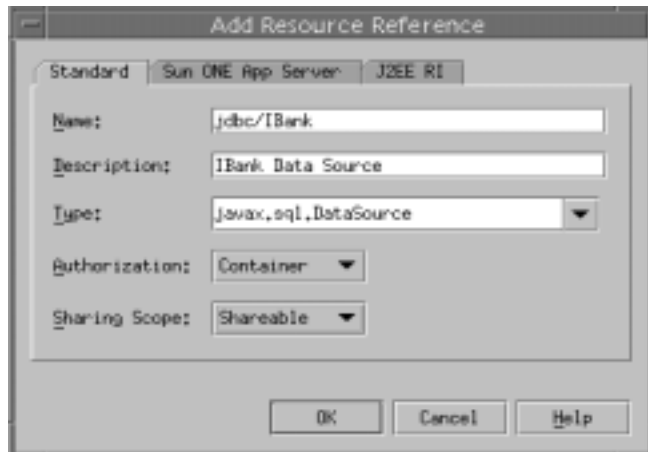
The following screenshot shows the declaration of an environment entry for the BankTeller session bean. InterestCalculator bean does not require this entry.

Click at the Environment Entries in the 'References' tab and then click on Add to add new entry for the DSN.



Adding Environment Entry to BankTeller Session Bean

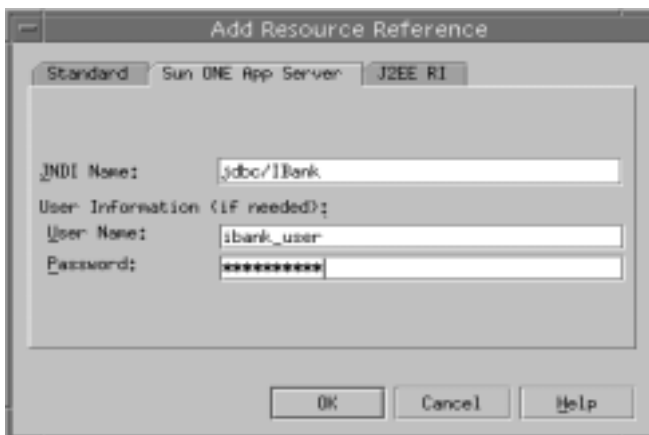
At the references tab of the Properties window for *BankTeller* Session bean, click at the Resource Reference to add a new resource. Following screen shot shows adding a new Resource for Data source in iBank i.e., jdbc/iBank.



Adding Resource Reference

Click at the Sun ONE App Server tab to set the JNDI name as 'jdbc/iBank' and username and password depending on the database schema used.

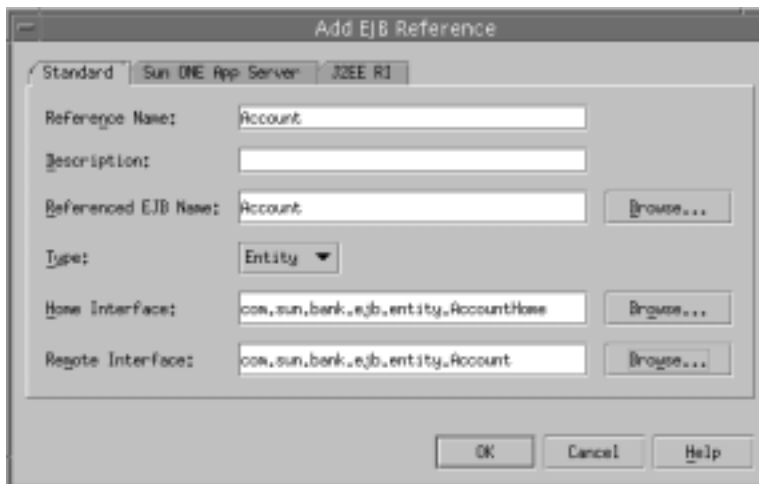
InterestCalculator bean does not require this entry.



Adding Resource Reference for Sun ONE Application Server

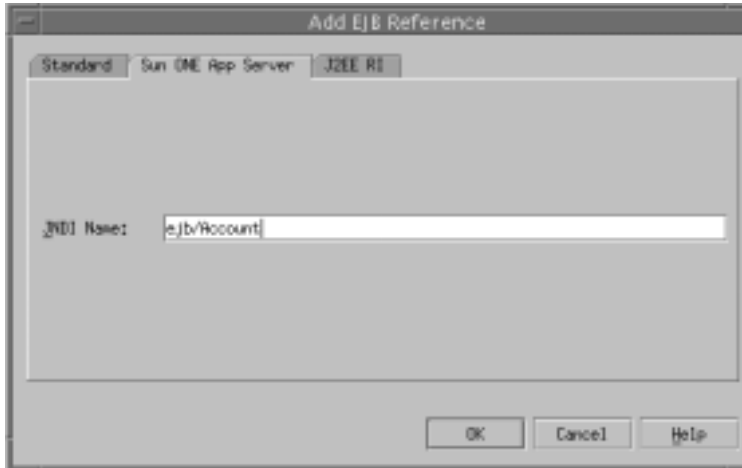
At the references tab of the Properties window, click at the Ejb Reference to addEJBReferences. Following screen shot shows adding EJB Reference for the BankTeller session bean. BankTeller session bean has references to Entity bean 'Account' and 'Customer'. So entries have to be made for both the entity beans.

Home and Remote interfaces have to be specified by clicking at the modify button and then selecting existing source for the beans.



Adding EJB Reference

Now click at the 'Sun ONE App Server' tab in the EJB Reference, to specify the JNDI name. Following screen shot shows the JNDI entry to be made for the Account entity bean i.e., 'ejb/Account'. Similarly whenEJBreference for 'Customer' bean is added the JNDI name specified at the Sun ONE App Server tab would be 'jndi/Customer'.

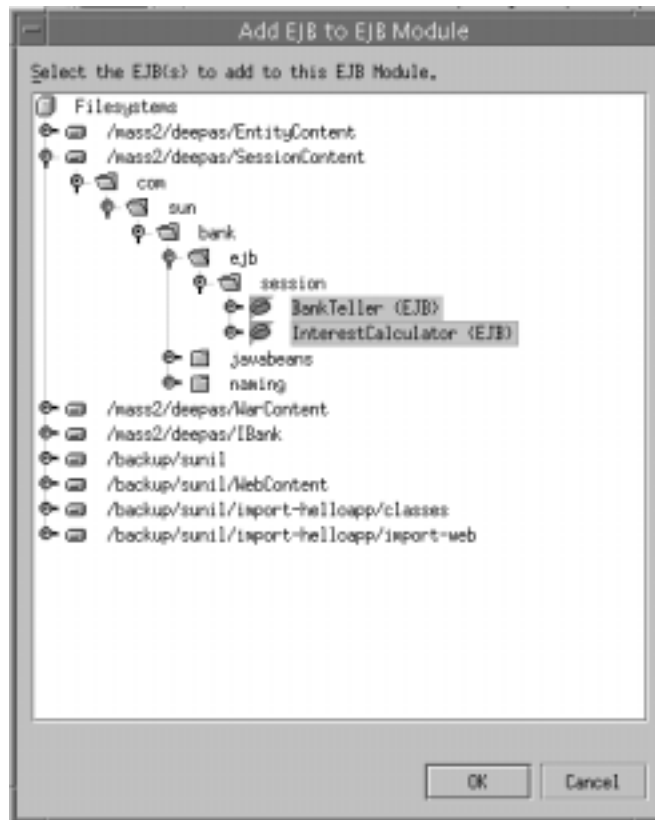


Adding EJB Reference entry for Sun ONE Application Server

4. Compile the source files
5. Create an EJB module and assemble the EJBs within it.

In accordance with the J2EE 1.2 specification, in Sun ONE Application Server 7 you must group EJBs together in an EJB module. Create new EJB Module i.e., SessionModule at the root directory i.e., 'SessionContent' by right clicking the folder and selecting option for New and then selecting J2EE and then finally selecting New EJB Module. After creation add the Session EJB's into it.

The screen shot below shows how the BankTeller and InterestCalculator EJBs are added to an EJB module SessionModule.



Adding Session Beans to EJB Module

Creating Module for Entity Beans

1. Directory for Entity Beans would contain following.

Bean class, Remote and Home interface for the following Entity Beans:

- a. Account
- b. AccountType
- c. Branch
- d. Customer
- e. Transaction

f. TransactionType

Customer entity bean is Bean managed and others are Container managed.

2. Configure the JDBC driver

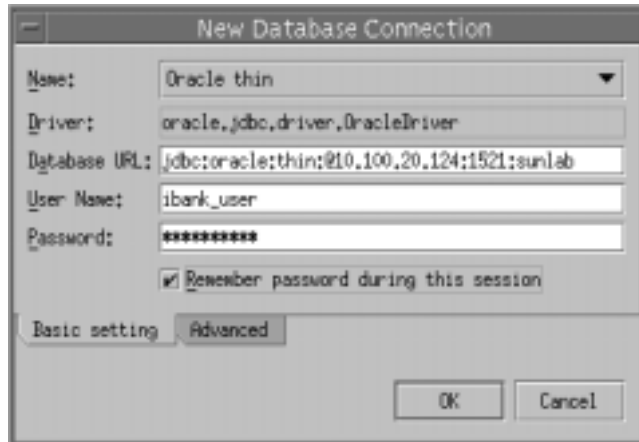
In the *Runtime* view of the Explorer, in *Databases / Drivers / Add Driver*: specify the driver name, implementation class, and the prefix of the relevant URL. The corresponding JAR or ZIP for the driver must be accessible to Sun ONE Studio for Java, and must therefore be copied into the <SUN ONE STUDIO FOR JAVA_ROOT>/lib/ext directory.

To place the driver classes in the appropriate Sun ONE Studio for Java directory in Solaris, run the following command from the shell (sh or ksh):

```
cp $ORACLE_HOME/jdbc/lib/classes12.zip <SUN ONE STUDIO FOR  
JAVA_ROOT>/lib/ext
```

3. Define the database connection properties

In the Runtime view of the Explorer, in *Databases / Add Connection...* indicate the driver used, the full connection URL, the user name, the related password, and lastly the appropriate database schema:

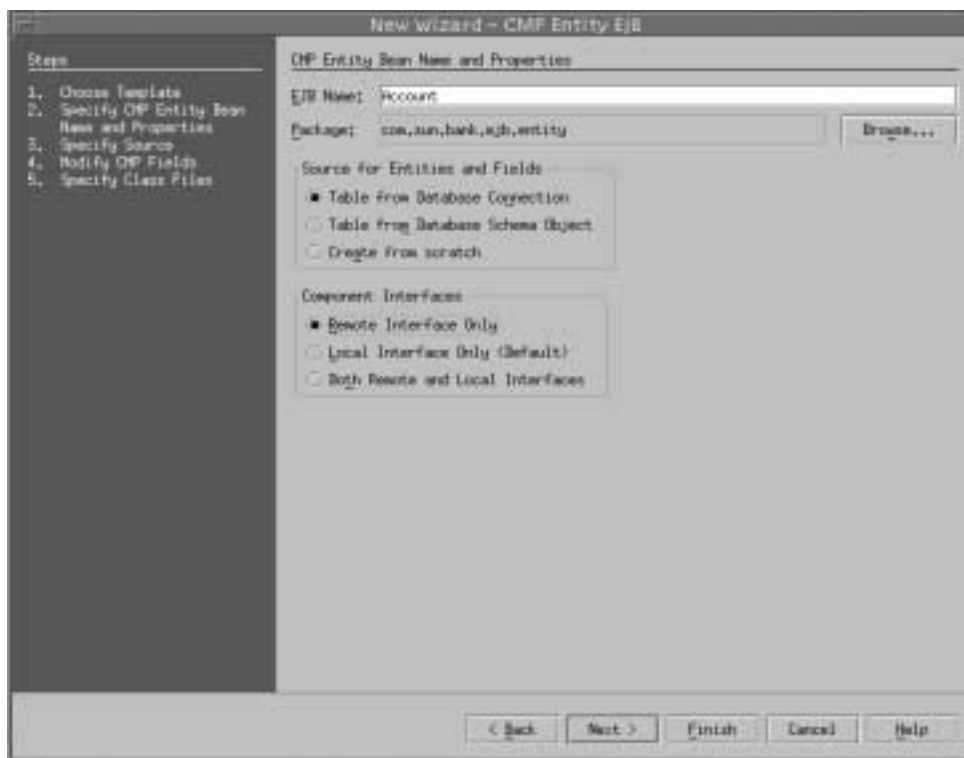


Configuring a database connection (Oracle) in Sun ONE Studio for Java

4. Create the new EJBs from existing source files.

In Sun ONE Studio for Java, it is possible to create an EJB from existing source files. Select the mounted directory 'EntityContent', walk through the directory till you reach 'entity' sub-folder. Right click and select option for new J2EE and finally click at 'Entity EJB(CMP/BMP)', which shows a new EJB wizard.

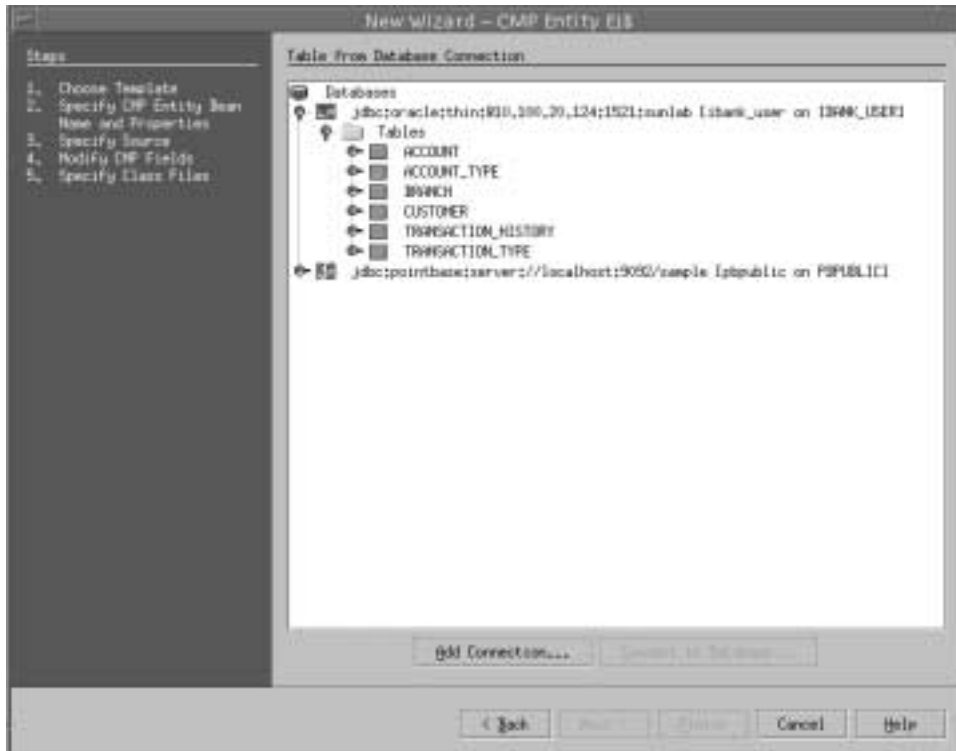
After specifying the main characteristics of the EJB (entity, BMP or CMP), and defining the name and package for the EJB, you match the existing source files and the different components of the EJB: implementation class, home and remote interfaces. In order to make the match with existing source files, use the "Modify" button in the dialog box and select "Select an existing source file." Entity beans require an extra step of specifying the mappings of the cmp fields with the table. In the Explorer Filesystems view, after selecting the option New CMP Entity Bean, Select option, table from Database connection in order to be able to specify the database table to be used for persistence of the EJB fields:



Creation of an Entity bean with container-managed persistence.

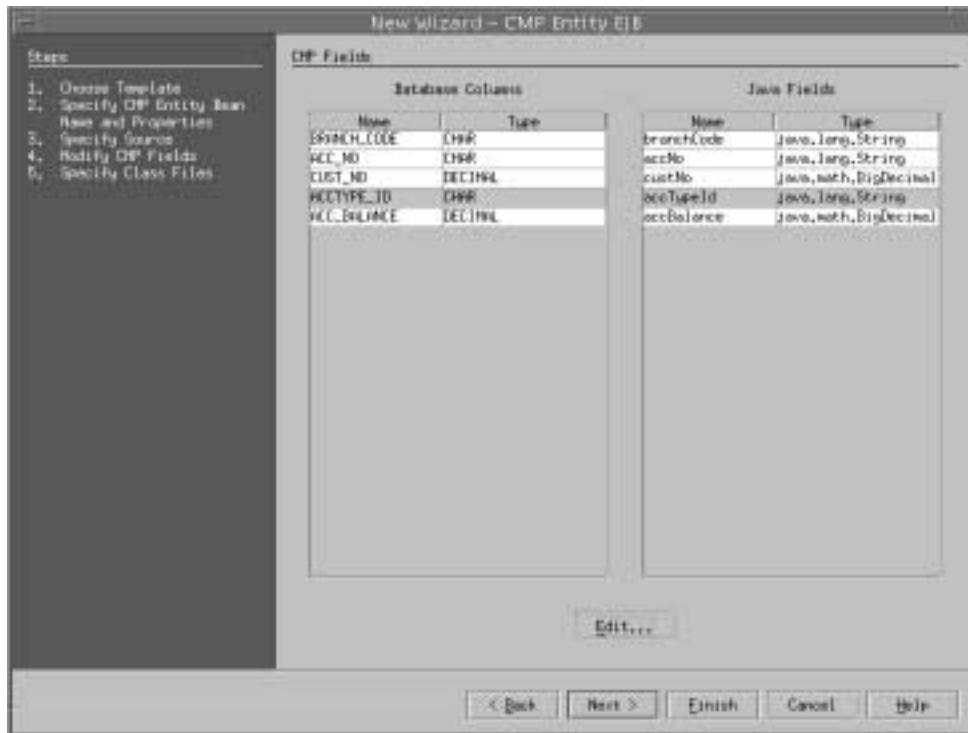
The following screen enables you to select the right connection from the list of database connections defined.

Once the connection is selected, list of tables accessible from this connection are shown, and select the appropriate table:



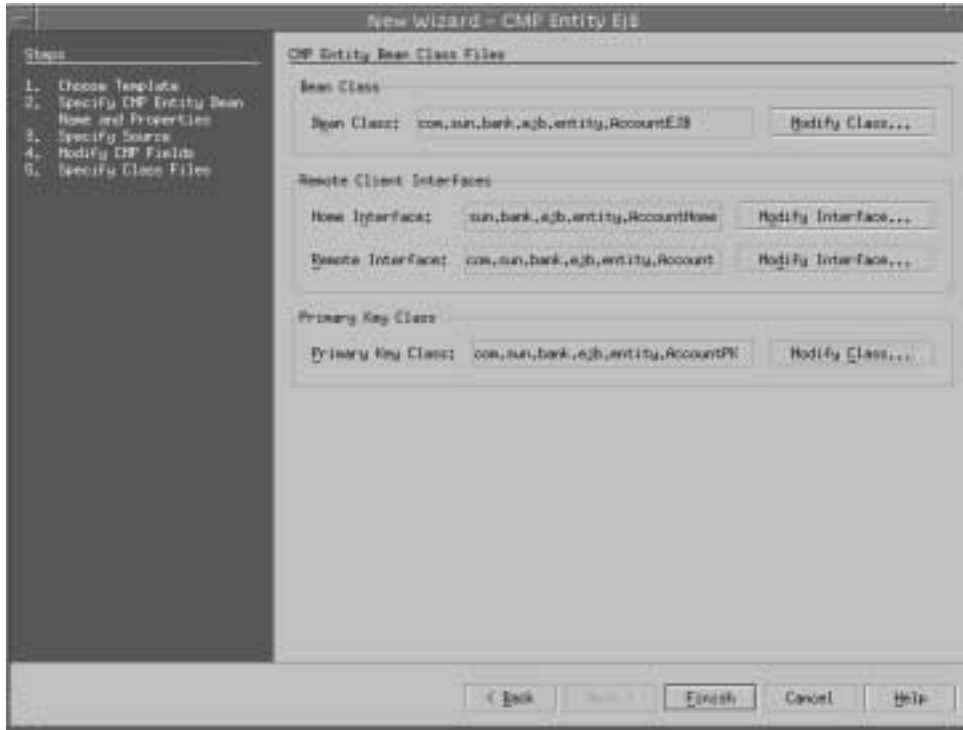
Choosing a table for mapping CMP bean fields.

The next screen is used to configure mapping between the columns of the table selected and the CMP fields of the bean. Particular care should be taken to correctly indicate the names of the bean fields and associated Java types.



Mapping between table columns and CMP fields of the bean

The next screen shot shows, specifying the source files for the Entity Bean.

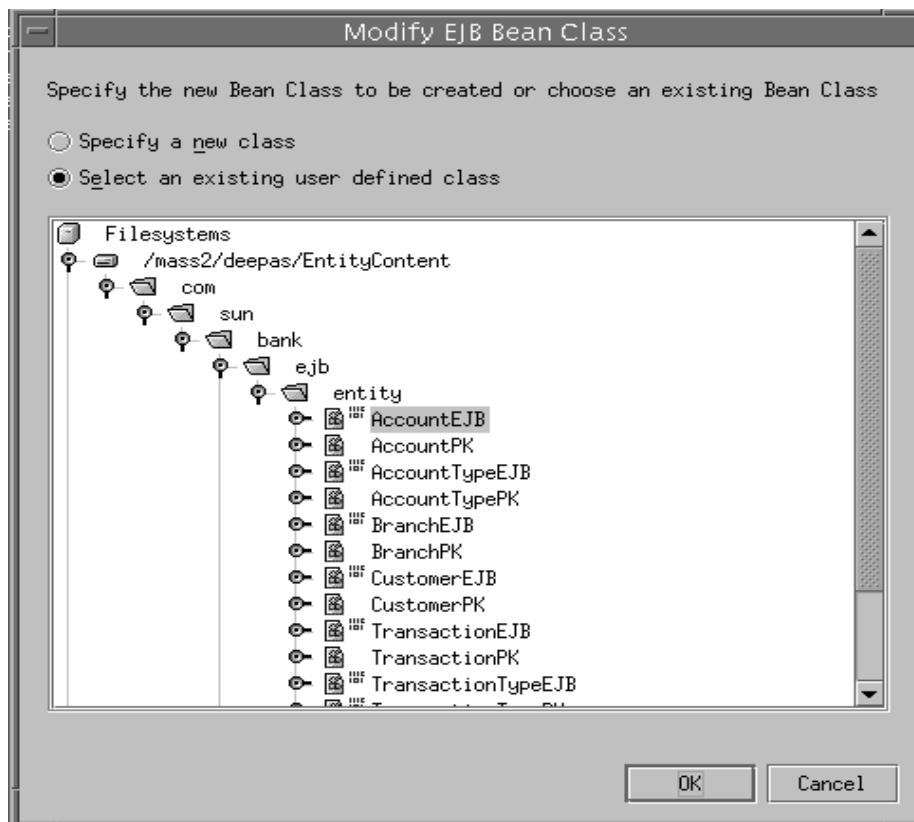


Specifying the source files for the Entity Bean

The next step involves informing Sun ONE Studio for Java that you want to create the EJB from existing source files, which can be specified by clicking at the 'Modify Class' button.

If you get any error while pointing to the existing source files, it may have caused because you made a mistake in the previous steps or the source is not migrated properly. Such errors should be handled by making changes as and when reported.

The next screen shot shows selecting existing source file for EJB bean class.



Specifying EJB Bean class by selecting option for Existing Source files

The next stage involves editing the properties of the new EJB wherever required.

All the entity beans have to be created in similar fashion.

(Note: This might give some errors giving option to select the existing class or using another one, click on to 'using same class'. Sun ONE Studio might show some unexpected results, in such condition, exit Sun ONE Studio and then reload it again.)

5. Edit the properties of the EJBs.

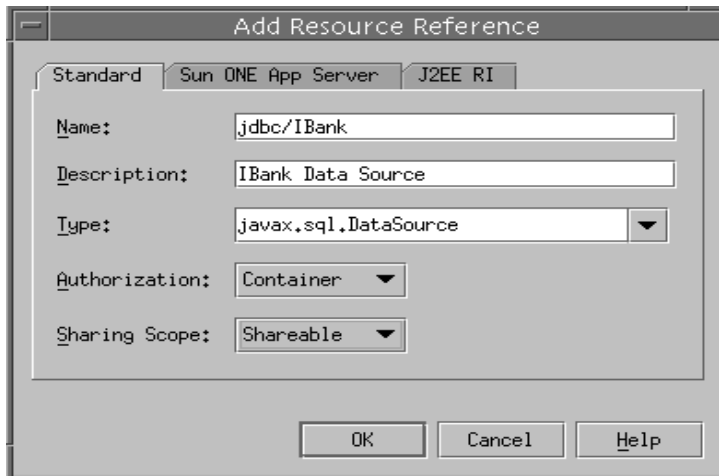
Select the new EJB in the explorer window so that its properties are displayed in the properties inspector.

In the properties window, select the *References* tab, click on the text zone to the right of the "*Resource References*" label, then on the button showing suspension points ("...") on the right hand edge of this text zone.

Following properties have to be set for the entity bean *Customer* only.

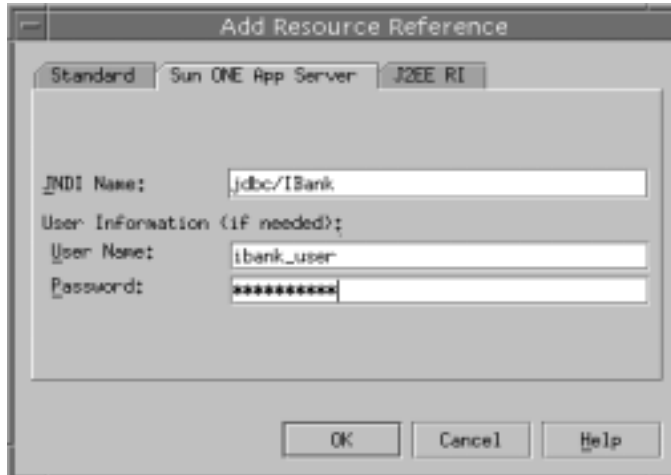
Following screen shot shows adding Resource reference for the Entity Bean *Customer*.

In the "Standard" tab, give the full name of the data source ("*jdbc/DataSourceName*"), the resource type (`javax.sql.DataSource`), and select "*Container*" from the drop-down list of options for managing access to this resource ("*Authorization*").



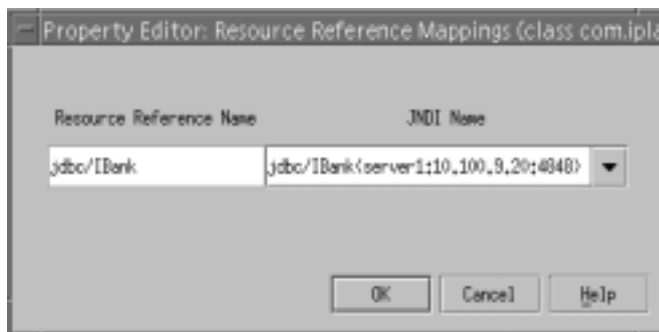
Adding Resource Reference

Once the declaration has been made, select the "*Sun ONE App Server*" tab, and specify the JNDI name of the data source "*jdbc/iBank*" in the JNDI Name column of the entry that corresponds to the resource reference defined previously. Also specify the username and password.



Editing Resource Reference

In the properties window select the 'Sun ONE AS' tab Click on the 'Reference Resource Mapping' and choose the data source i.e, `jdbc/IBank` on the server instance which has to be used. Following screen shot would depict the same

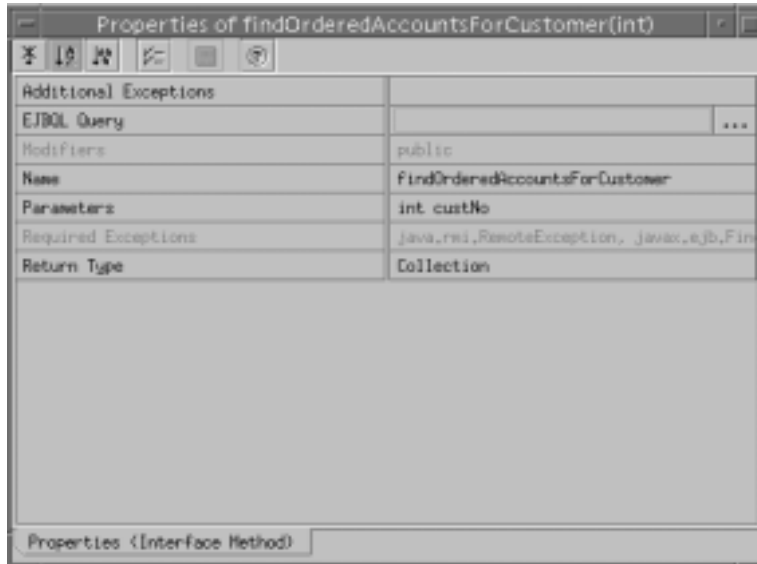


Resource Reference Mapping for Sun ONE Application Server

6. Set EJB QL for finders other than *findByPrimaryKey* method.

EJB QL has to be specified for finders. As per the CMP 2.0 specification, the finders will use EJB QL.

In iBank application the entity bean that would require this type of editing is Account bean. Select the *AccountEJB* node in the Sun ONE Studio explorer window and expand the finder methods in it. Click on any finder method other than the *findByPrimaryKey* to open its properties window:



Properties of Finder Method

Click at the EJBQL Query to enter the query. Following screen shot shows the query entered:



Editing EJB QL for the Finder

7. Create an EJB module and assemble the EJBs within it.

Create new EJB module named *EntityModule* and add all Entity beans into this module by right clicking on the EJB module and selecting the option to add EJB's. As per the J2EE 1.2 specification, you must group EJBs together in a EJB module.

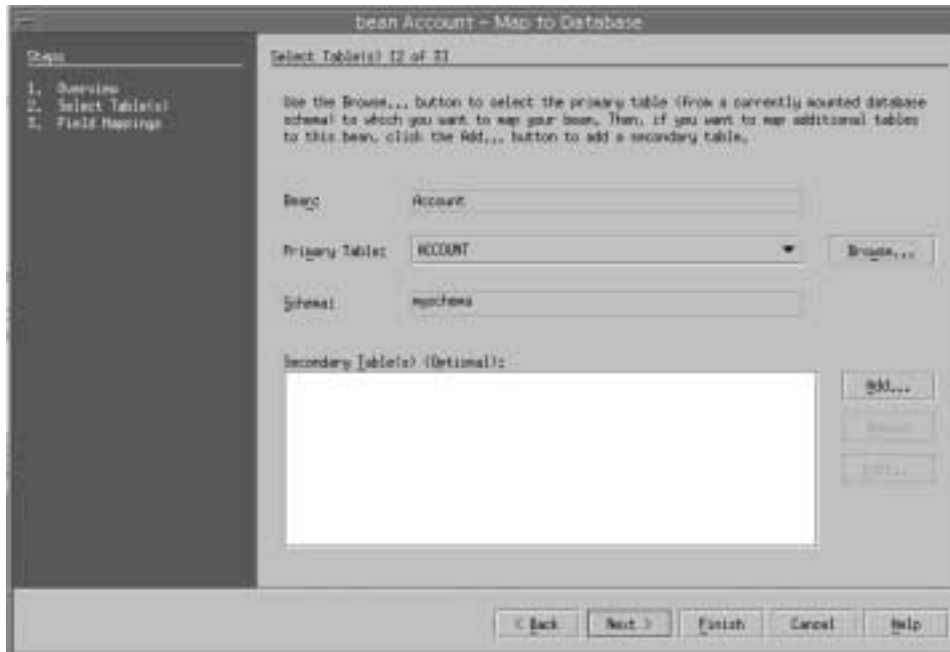
8. Create new Database Schema

From the file menu click new and then select new Database Schema. Provide the connection information for the database from which schema has to be captured.

9. Map the database entries for Sun ONE Application Server 7.

Select a EJB node in the EJB module, right click the node to choose the properties window and select Sun ONE AS tab. Specify the database schema and primary table name for this particular entity bean. Repeat the process for other Entity Beans in the EJB Module.

Following screen shot shows selection of primary table for the entity bean Account



Database Mapping

Click on 'Next >' for specifying the mappings for the cmp fields of bean with the table fields.

Now select the Sun ONE Mapping Tab from the properties window and re-enter the mappings.

Following screen shot shows mappings for the Account EJB

Property	Database Field
acctBalance	ACCOUNT_ACC_BALANCE
acctNo	ACCOUNT_ACC_NO
acctTypeid	ACCOUNT_ACCTYPE_ID
branchCode	ACCOUNT_BRANCH_CODE
custNo	ACCOUNT_CUST_NO

Properties of entity bean 'Account'

Similarly mappings for all the Entity beans have to be set.

See Appendix A for the details on the mapping of particular Entity bean to corresponding database table field.

10. Add CMP resource

Select EntityModule and view its properties, click at Sun ONE AS tab, and now click at CMP Resource to configure the Persistence manager factory.

Following screen shot shows the configuration:

Adding CMP Resource

Creating an enterprise application in Sun ONE Studio for Java

After creating the Web application and EJB files, the next step is to create an enterprise application, which groups all the modules together. The process for creating an enterprise application is as follows:

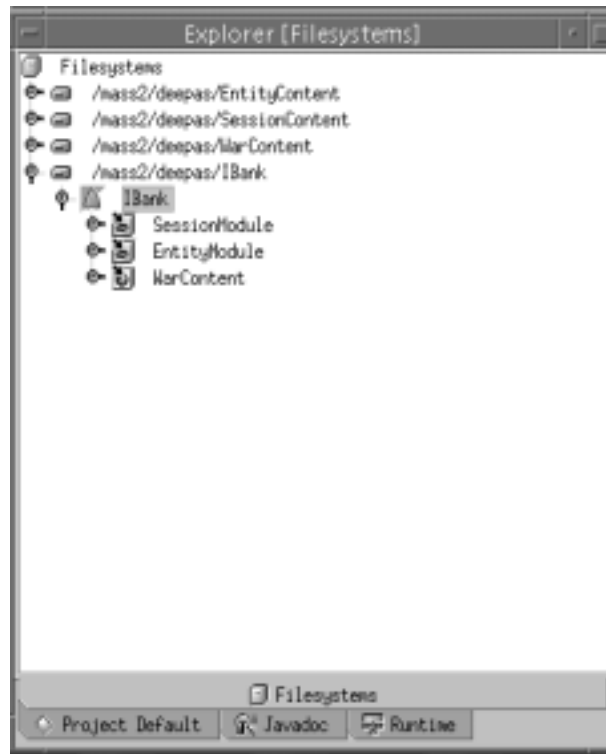
1. Create a new enterprise application module in a new directory say 'iBank' under the same package available for the source.
2. Add the Web module and EJB modules to the enterprise application module

The following screen shots show an enterprise application called iBank, containing a Web module called WarContent and EJB module called SessionModule and EntityModule.



Adding Modules to the Application

Following screen shot shows Application iBank having 3 modules in it.



File System showing Application 'iBank' having different modules

3. Edit the enterprise application properties.

The property editor allows you to set the different properties of the enterprise application module. In particular, this is where the root context name is defined for the Web module of the enterprise application:



Specifying the Web Context

4. Export EAR file.

Export EAR file by right clicking the Enterprise application and selecting option for exporting EAR file. This EAR file would contain JAR files, WAR file and XML files. This EAR file has all the Sun ONE specific XML files required for the deployment on Sun ONE Application Server 7. This EAR file can now be deployed.

Deploying an application in Sun ONE Application Server 7

The last stage is to deploy the application on an instance of Sun ONE Application Server 7. The process for deploying an application is described below:

1. **Deploying an application on Sun ONE server 7 instance from Sun ONE Studio for Java**

Right click on the EAR file and select option '*Deploy*'. This would deploy the application on the default server instance. Restart the server instance and then test the application.

2. Deploying an application on an Sun ONE Application Server 7 instance using Sun ONE Application Server 7 *asadmin* utility

An alternative to using Sun ONE Studio for Java to deploy enterprise applications on an Sun ONE server instance is to use the Sun ONE Application Server 7 *asadmin* utility, after creating and exporting the application EAR archive from Sun ONE Studio for Java.

For instructions on deploying the iBank application using the *asadmin* deployment utility, please refer to "Deploying iBank application on Sun ONE Application Server 7 using the *asadmin* utility" section under "Manual Migration of iBank Application" topic.

Migration from BEA WebLogic Server v6.1 and IBM WebSphere v4.0

The detailed J2EE application migration process and the sample application migration for BEA WebLogic v6.1 and IBM WebSphere v4.0 is part of another guide which can be found at the Migration Site.

Migration from KIVA/NAS 4.1 to Sun ONE AS 7

Kiva/NAS 4.1 Java AppLogic applications can be migrated to J2EE web modules using iPlanet Migration Toolkit (iMT 1.2.3). The resulting web modules leverage JATO and a thin KFC (Kiva Foundation Classes) adaption layer to support running the AppLogic code on any J2EE web container.

Introduction

Before starting the migration process, be sure you have read the release notes so that you are aware of the latest information and any issues that might be relevant to you and your environment. Also refer to `%MIGTBX_HOME%/bin/readme.txt` file. This file also describes proper installation and configuration of the Migration Toolbox and its environment, which must be complete before beginning the migration process described in later sections.

`%MIGTBX_HOME%` represents the directory in which you installed/unzipped the Sun ONE Migration Toolbox (S1MT).

Migration Preparation

Migration Process Overview

There are two main phases to full migration of a proprietary AppLogic application to its J2EE equivalent. These phases are the *automated migration phase* and the *manual migration phase*. The automated migration itself consists of two steps called *extraction and translation*.

Automated Migration Phase

This phase consists of preparing the AppLogic application source for migration and then using the S1MT to perform automated extraction and translation. The input to this phase is a user provided archive (JAR/ZIP) containing the original application source files (AppLogic files, GXR, query files, templates, static content and regular Java source and properties). This file is called the application extract archive. Using a standard Java archive (JAR/ZIP) to package the existing application and NOT requiring a NAS/iAS runtime environment allows the migration environment to be more flexible; migration may even be performed remote from the customer site since the runtime infrastructure (databases, web servers, app servers) is only needed during manual migration for unit testing. Essentially, this archive is just the targeted contents of the `./nas/APPS` directory of the application server and document root of the web server. The Extraction Tool for KIVA AppLogics will read this archive and create the application descriptor. iMT v1.2.3 now supports the automatic creation of the application extract archive (See the Kiva Migration Toolbox Builder 'Addin' from the 'Addin' menu).

The application descriptor (an XML file) is used to guide the Translation Tool on the disposition of each file in the archive. The migrator may need to adjust the application descriptor. See Technote on editing the application descriptor. After running the Translation Tool the result is a partially (or in some cases, fully) migrated application consisting entirely of J2EE-compliant components based on JATO and the KIVA Migration library composed in a web application archive (deployment descriptor, servlets, JSPs, Command, query files).

The output from the translation process entirely transforms HTML templates to JSPs and converts GX tags to new JSP tags used with the KIVA migration library. AppLogic source files are adjusted to use the KIVA migration library (minimal change mostly to import statements). The translation process also creates the web application infrastructure including all the components of the JATO application and direct command invocation module. However, the translation phase does not automatically port code written to proprietary KFC APIs which are "non-targeted" in the KIVA migration library. This porting will be the primary task during the manual migration phase. iMT v1.2.3 now supports the automatic migration of static documents specifically with help in fixing URLs. (See the Kiva Migration Toolbox Builder 'Addin' from the 'Addin' menu) and the new Kiva Document Translation Tool.

Manual Migration Phase

In general, the manual migration phase consists of reviewing the automatically migrated application output and porting non-targeted KFC API code to J2EE-specific code. Understand that this process does not typically require a redesign of the application or its architecture. In many cases, code which needs manual attention is clearly outlined in a deprecated compile using the MIGRATION version of the KIVA Migration library [`kivaMIGRATION.jar`]

Preparing your Working Environment

Before going further, ensure you've done the following:

1. Make sure you've installed the iPlanet Migration Toolbox
 - Unzip the distribution archive into the desired target directory. Follow the directions in the `readme.txt` file.
 - Test the installation by trying to start the Toolbox application. Run `toolbox.bat` in the `%MIGTBX_HOME%/bin` directory. An empty Toolbox should appear after a few moments. If nothing appears, check that the Migration Toolbox was installed properly and that all appropriate environment settings have been set in `%MIGTBX_HOME%/bin/setenv.bat`.
2. To avoid class version issues, it is strongly recommended that you remove all JAR files from your JDK's extension directory (`%JAVA_HOME%/jre/lib/ext`) while running the Toolbox application. All the classes necessary for running the Toolbox are included with the distribution. Please note that simply renaming the JAR files in the extension directory is not sufficient; you must move them to a different location.
3. Identify the AppLogic based application which is to be migrated.
4. Generate the application extract archive. In the simplest case, it is a ZIP or JAR file containing all files and directories under `./nas/APPS` which are related to the application. The iMT for AppLogics DOES NOT actually load or run any Java classes or libraries from your application. All extraction and translation is done at the source level so it is not a problem if the archive does not contain all dependent classes or libraries - these will only be needed while compiling after automated migration.
5. At this point, you may also want to install Sun ONE Application Server 7 (known as S1AS), Forte for Java 4.0 EE or another J2EE-compliant servlet/JSP container.
 - Follow the installation instructions for the server or container

- Test the installation by starting the server or container and trying to load the default home or index page. If an error occurs, troubleshoot the installation process before continuing

Preparing a Project for Automated Migration

Because AppLogic and the KFC allowed developers immense latitude and practically no prescription, there is no way for the iMT to account for all possible permutations. For this reason, it is strongly recommended that customers engage Sun Professional Services to assist in preparing projects for migration

iMT Kiva BETA customers discovered that during manual migration procedures portions of the existing code caused obstacles in compiling the code in the JDK 1.3.1 an J2EE environment. The following is a list of considerations and activities which should be performed before a migration is attempted.

You have to prepare the application code for J2EE environment before using the iMT. The code should be compiled against JDK 1.3 (or at least JDK 1.2.2) and the J2EE APIs. For instance, the iMT comes with the `kfcjdk11.jar` library for the KFC. This is provided so that customers may compile their existing application in an advanced J2EE capable IDE like Forte for Java (FFJ). A standard AppLogic application should be able to compile in FFJ by simply adding the `kfcjdk11.jar` to the classpath (FileSystems). Prior to compiling, the deprecate flag should be set (TRUE) to expose deprecated code.

When customers are already using JDBC it is highly recommended that the database services be re-factored for the latest third party drivers (JDBC, Oracle, and Sybase) as recommended by the vendors for the new JDK.

In order for exact migration tasks to be identified and sourced, all special considerations would need to be assessed first. In simpler terms, we need to identify anything “out of the ordinary” which may be in the code. This includes code patterns or use of Java services which conflict with a concurrent server pattern of the J2EE container contract with the developer. For instance, if the code used `java.lang.Thread` directly or shared resources, this code will need to be inspected for suitability in J2EE.

Some customers use other third party Java services which may themselves may not be ready for J2EE even though the customers code is. For instance, an old version of CORBA (e.g. Visibroker for Java, or Iona) may need to be upgraded.

J2EE has the requirement that logical applications shall be deployed into separate web applications as WARs. It is easier to isolate logical applications and common libraries before migration proceeds.

Customers need to prepare for the change of external URLs. No matter what technique is used to migrate to J2EE, URLs will change and therefore a strategy is required for bookmarks and previously published URLs. The iMT v1.2.3 release provides some support for automated migration of static documents URLs. Nevertheless, customers will need to survey the existing system to account for all the changes which will need to be managed.

Preparing the GXR file

In order for the extraction phase to perform accurately when generating the application descriptor, a GXR file is needed to identify the AppLogic files and the AppLogic names used during NameTrans and URLs. Most applications use at least one GXR file or at least one for each package in the application. The extraction phase requires one (1) single GXR file in the application extract archive. If you have more than one GXR file, combine them. If you do not have a GXR file you will need to compose one using proper GXR syntax; the source data can be acquired by dumping the KIVA registry (`./nas/bin/kreg -save temp.out SOFTWARE`). In short, the extraction tool uses the GXR file to determine which files in the application extract archive are AppLogic source files and also develops a mapping of GUID to AppLogic name to AppLogic class name.

Before Running the Extraction Tool

If your AppLogic application is entirely based on app server side Java and material (query files, HTML templates, AppLogic source, support Java source, etc.) then you can usually create the application extract archive by zipping up the relevant contents of the `./nas/APPS` directory. However, if the application also contains static content then you have some additional work to do. It is common and more efficient to have static content located on the Netscape Enterprise web server and leave the dynamic content on the application server side (AppLogics and templates). Depending upon your J2EE server vendor, you may benefit from this separation or it may be helpful to combine the static content and dynamic application resources. The static content may be added to the WAR during or after automated migration - this is usually the easiest.

There is one important consideration when migrating from original AppLogics application to J2EE JATO using the iMT. URLs which invoke AppLogics (POST/GET) are absolute URLs (e.g.

`http://host/cgi-bin/gx.cgi/AppLogic+HelloWorld`) After migration, the URLs become relative to a context defined by the ServletContext and therefore absolute URLs should be avoided. The transformation of URLs is different for static content and dynamic content (HTML templates). The iMT maps all

AppLogics to JATO Command implementations in a special JATO module called the direct command invocation module. Since all translated AppLogics are invoked from the same path within the ServletContext, the intra-AppLogic invocations (URLs) in the resulting HTML markup are the most predictable. Therefore, all AppLogic invocation URLs are translated as if intra-AppLogic is in order. When there is static content among the HTML templates in the application extract archive, the AppLogic URLs will need to be adjusted since the context of the static content will most likely NOT match the path in the ServletContext for the direct command invocation module (ModuleServlet). The OnlineBankSample application migration demonstrates the need to make this adjustment and utilizes the automatic translation of static documents using the Kiva Document Translation Tool.

Migrating OnlineBankSample

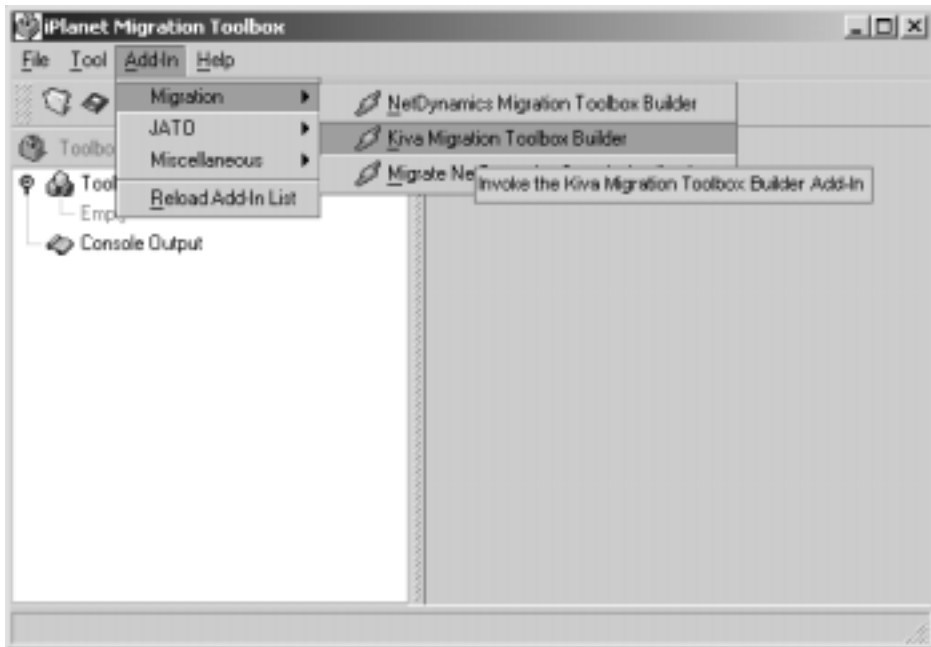
This section describes the automated and manual migration procedures of the onlineBankSample to J2EE.

Running the Migration Toolbox

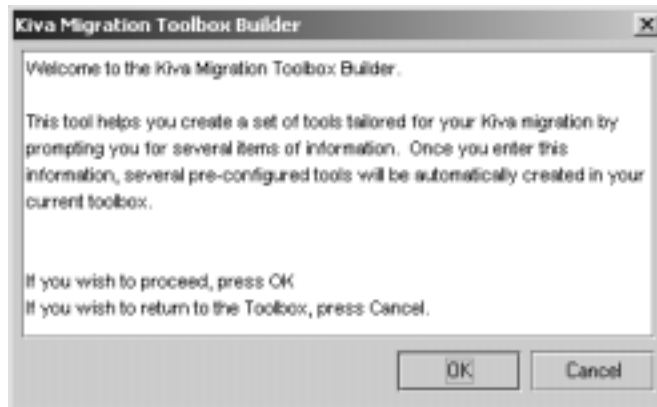
Install iMT 1.2.3, if you have not installed it already and refer to "Migration Preparation" section for details on installing and starting iMT. Make sure that you edit the `%MIGTBX_HOME%\bin\setenv.bat` to account for the installation location of the iMT and the JDK home dir.

Create a Toolbox

1. Select the Kiva Migration Toolbox Builder from the Addin:Migration menu.



A modal dialog wizard will appear.



Select OK to proceed to the first step of the wizard.



2. Automated iMT migration will produce some J2EE infrastructure including new Java JATO files. These new files must be assigned a package. Although existing Java source in the original application will retain packaging, we still need to assign a package for these new files. There is no restriction on the package name. The default value is provided for the OnlineBankSample application.

Enter a package and select OK to display the next step in the wizard.



3. Enter the directory where all materials generated by the iMT will be stored. The default is usually satisfactory and is used in this example. Select OK to display the next step in the wizard.

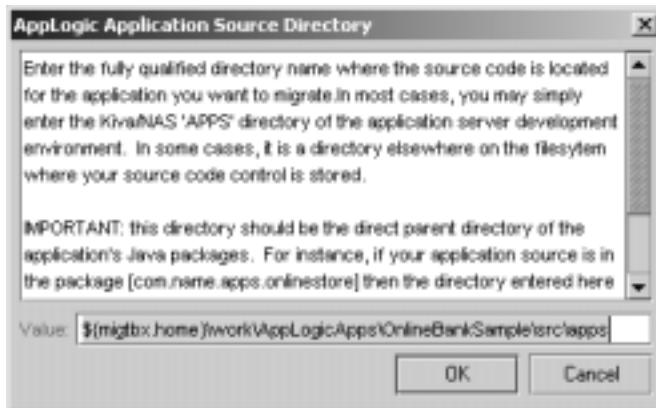


4. The Automatic Application Extract Archive wizard will help create tools to automatically build the archive. If you choose OK then proceed to Step (5) otherwise Cancel will show the Extract Archive selection dialog (see below)



which allows you to specify the manually created archive. This is useful if you already have invested time in the extract archive and you are just building a new Toolbox.

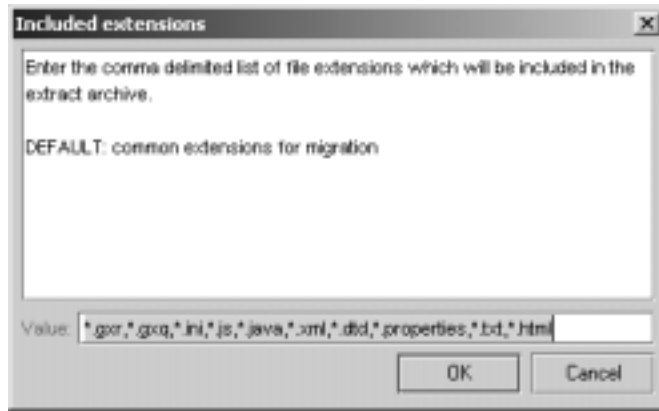
If you choose OK to the Automatic Application Extract Archive wizard you will see the following dialog:



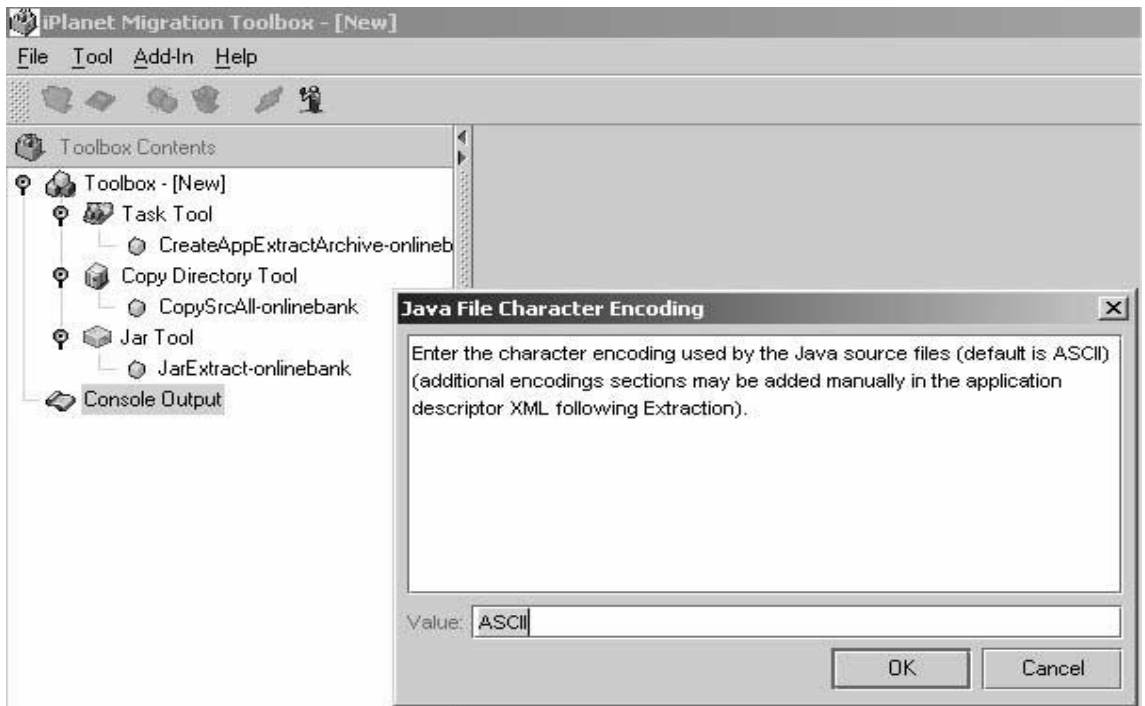
5. Select OK to accept the default and display the next step.



6. Select OK to accept the default BLANK list and display the next step in the wizard.



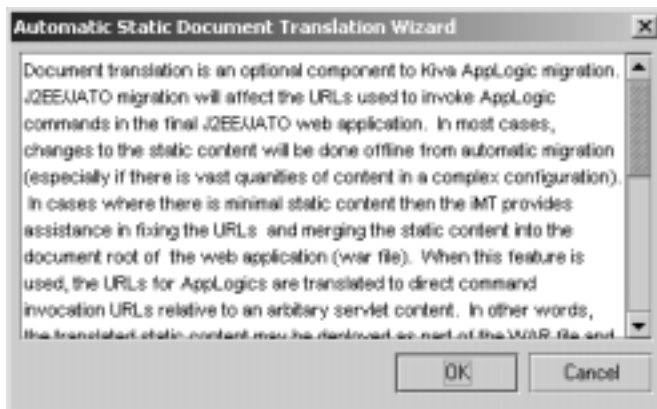
7. Select OK to accept the default, generate three new tools for the toolbox and display the next step in the wizard.



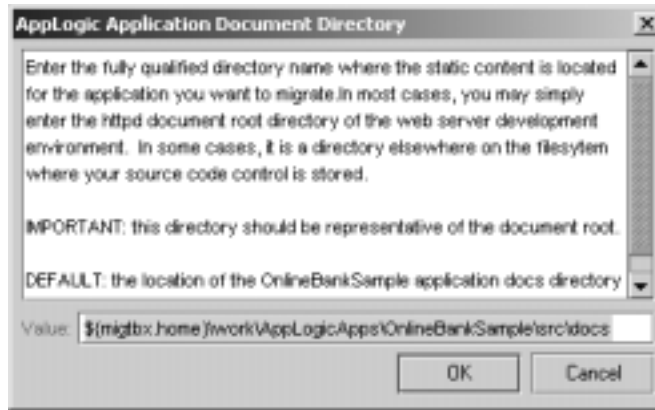
8. The OnlineBankSample only contains Java source which is ASCII encoded so accept the default. When you are migrating your own application, if you have Java source using another character encoding (e.g. Japanese Shift_JIS) then be sure to specify the encoding used. Select OK to display the next step in the wizard.



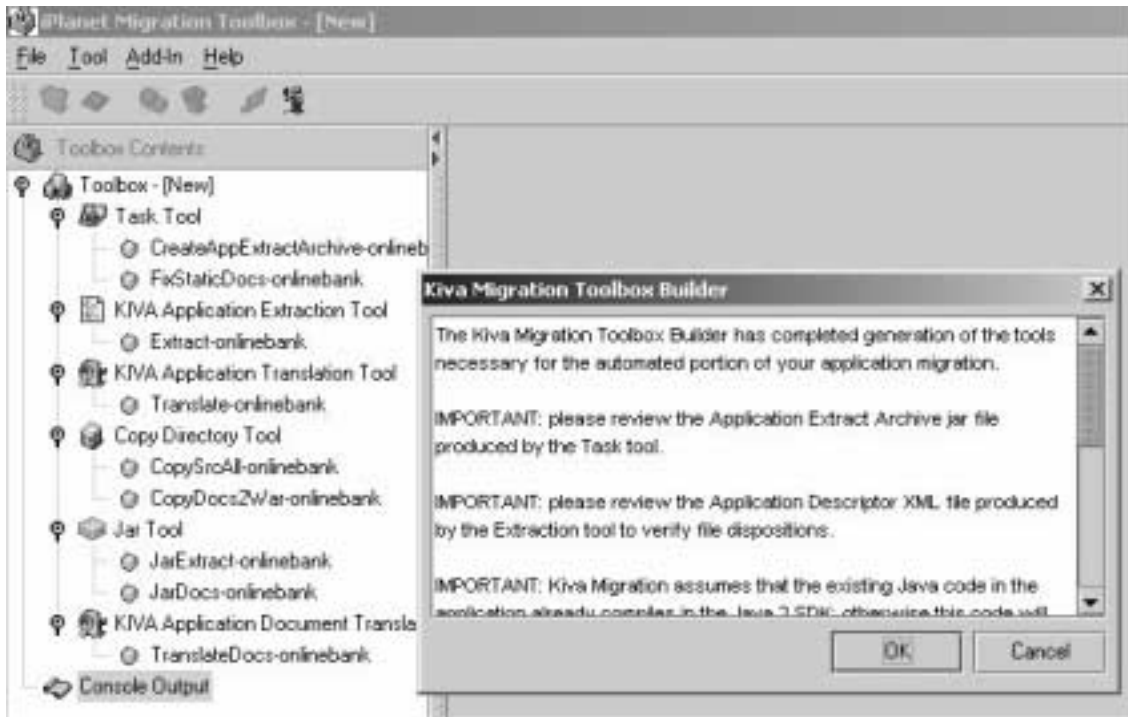
9. The OnlineBankSample only contains Query files which are ASCII encoded so accept the default. When you are migrating your own application, if you have Query files using another character encoding (e.g. Japanese Shift_JIS) then be sure to specify the encoding used. Select OK to create the Kiva Extraction and Translation tools in the toolbox and display the next step in the wizard.



10. iMT v1.2.3 now provides assistance automatically translating static HTML documents and combining them with the WAR file. If you choose the skip this feature the wizard exits and the toolbox is complete. For the OnlineBankSample, we will Select OK and use the automated feature.



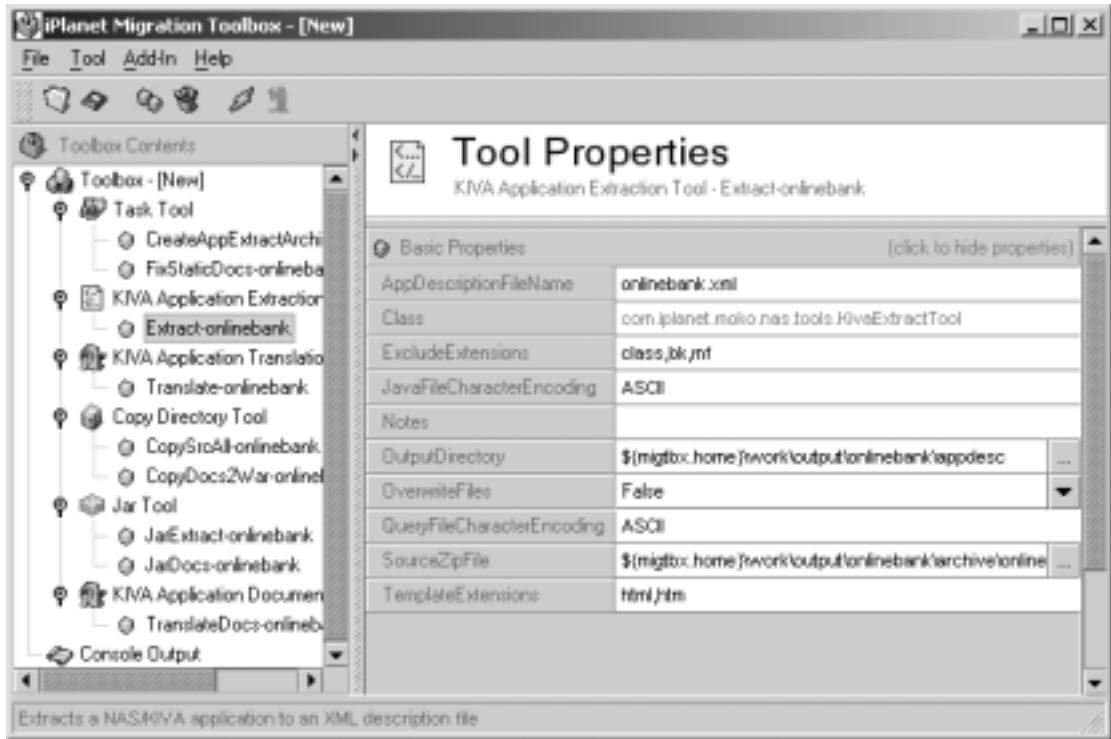
11. Select OK to accept the default location of the document directory for the OnlineBankSample and proceed to add four (4) new tools to the toolbox and exit the wizard.



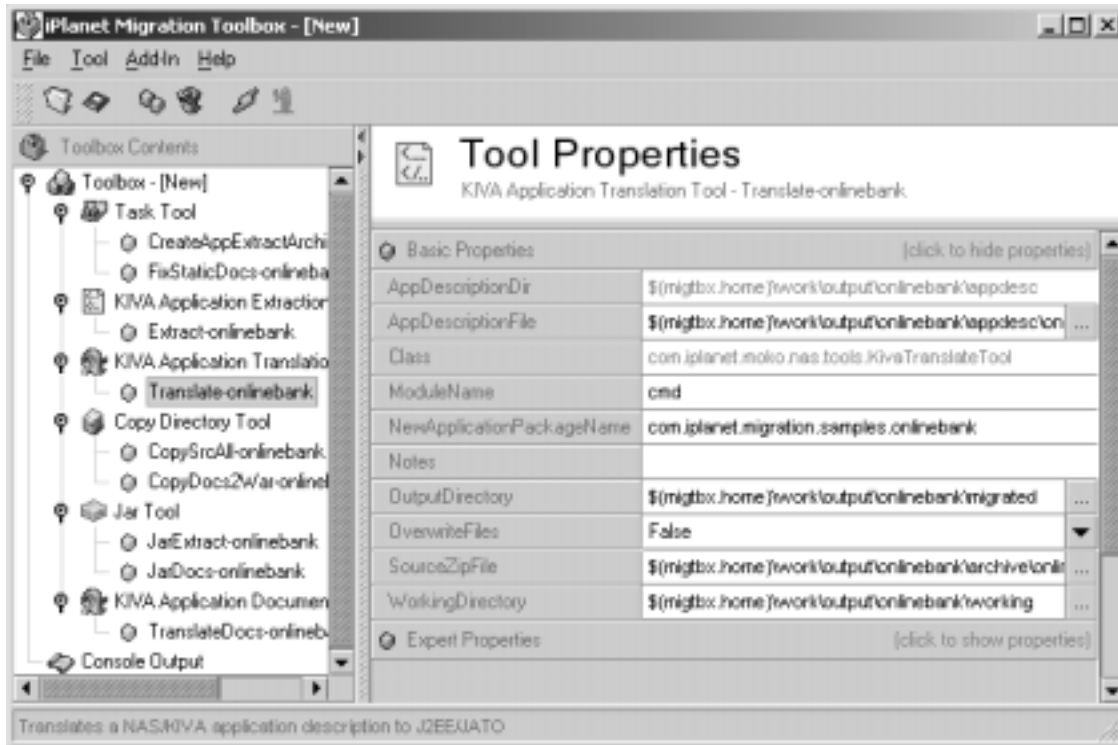
Click 'OK' to complete the generation of the necessary tools. The result of the Addin is a complete Toolbox consisting of an Extraction and Translation tool and the optional tools to automatically create the application extract archive and translate the documents. Please remember to select the 'branch' for each tool to display the detailed Help for each tool in the right frame. The Help explains each property in the tool. Click on each 'instance' of the tools to display the bean property panel in the right frame. Both the basic and expert properties may be edited.

The Task Tools simply cause a list of other tools to be executed in order. It is usually more informative to run the tools separately so that you can carefully watch the console output.

The Extraction tool properties are shown here:



The Translation tool properties are shown here:



12. Invoke the CreateAppExtractArchive-onlinebank Task tool. This tool runs the CopySrcAll-onlinebank and JarExtract-onlinebank tools one after the other to produce the application extract archive

```
%MIGTBX_HOME%\work\onlinebank\archive\onlinebankApps.jar
```

13. Invoke the Extract-onlinebank. This tool runs very quickly. The trace of the tool execution is shown in the Console frame. It will introspect the application extract archive, concentrating on GXR files to produce the application descriptor XML file. You must review the application descriptor and sometimes edit it so that the files are organized properly so that the Translation tool clearly understand the disposition of each file (including the proper encoding) in the archive. For iMT 1.2.3 the Extraction tool will automatically discern the encoding of the HTML templates. Please review the application descriptor to ensure that the proper encoding was selected for each template. The location of the application descriptor is

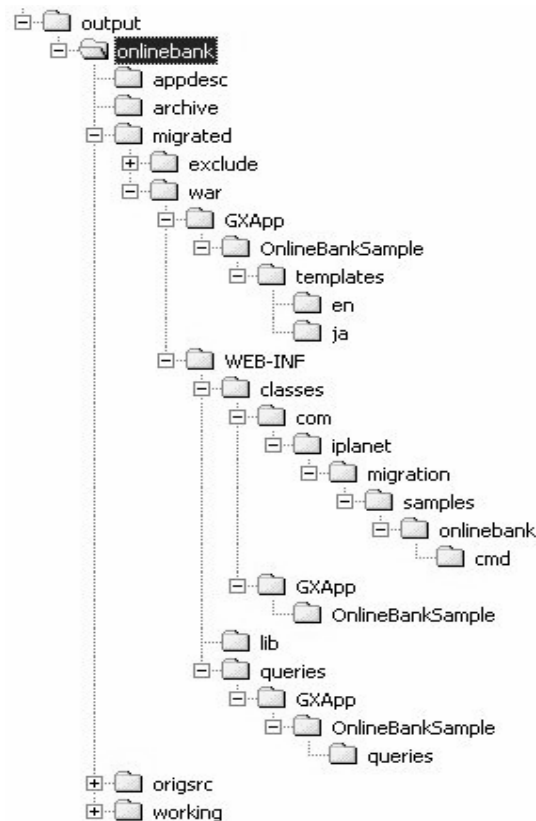
```
%MIGTBX_HOME%\work\onlinebank\appdesc\onlinebank.xml
```

and it is helpful to use an XML editor to navigate and edit this file carefully. Here is a view of a portion of this file in XML Spy.



- Invoke the Translate-onlinebank tool. It takes a little longer than extraction and the time will depend on the number of AppLogic source files, Java files and Html templates you have in the archive. ALWAYS review the Console output when Translating to see if errors are reached. The Translation tool will usually

skip past errors and continue to translate the rest of the application. It is easy to miss a WARNING or ERROR in the large trace output. You may change the expert properties to enable debugging and verbose tracing to see the real detail of the translation including use of the internal calls to Regular Expression mapping rules and element processing. The results of translation are placed in the 'migrated' directory under output directory



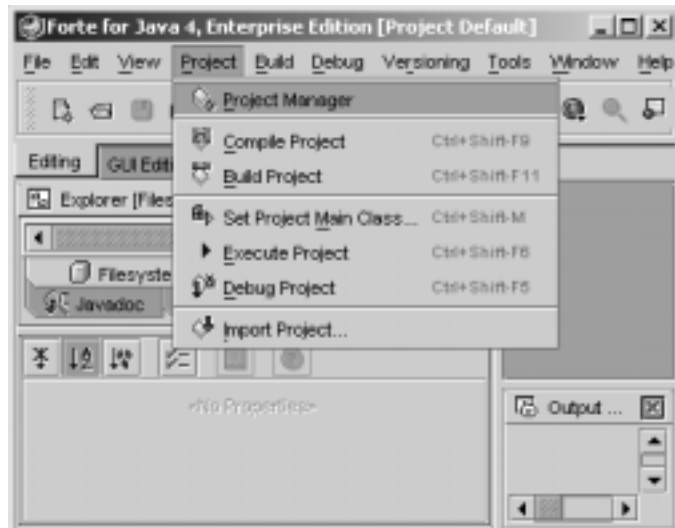
A complete J2EE JATO Web Application is created under 'migrated/war'.

15. Invoke the FixStaticDocs-onlinebank Task tool. This task will call in order the JarDocs-onlinebank, TranslateDocs-onlinebank and CopyDocs2War-onlinebank so that the static content URLs for AppLogics are fixed and the content is copied to the document root of the WAR.

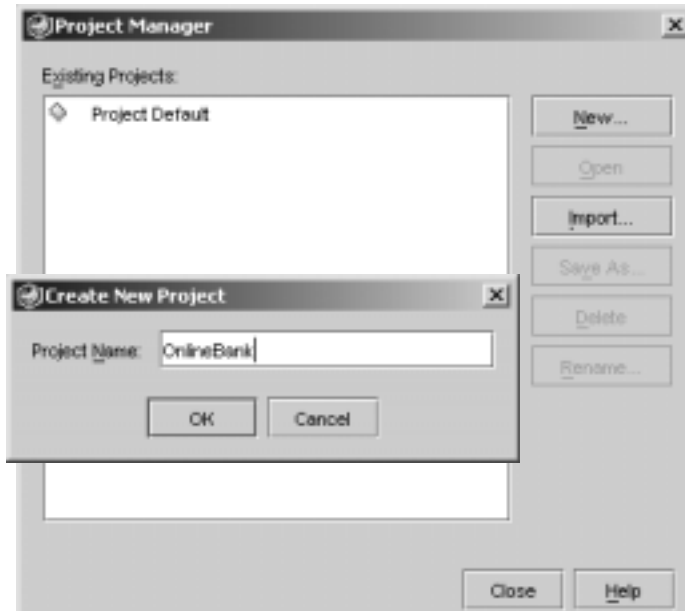
At this point, automated migration is complete and manual migration starts.

The easiest way to proceed in manual migration is to load the web application into a J2EE IDE. Forte for Java EE (FFJ) is used in this example.

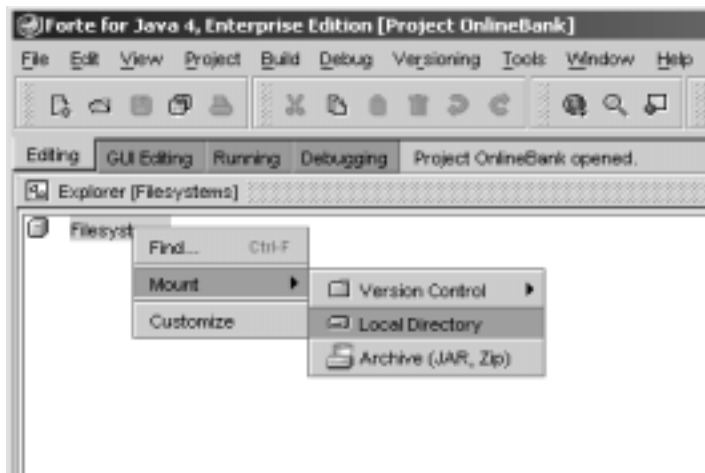
16. Start FFJ 4.0 and create a new Project called OnlineBank. Make sure there are no existing file systems in the new project. Select [Project] from menu and click [Project Manager].



17. On the Project Manager window, click New and put a project name (OnlineBank).

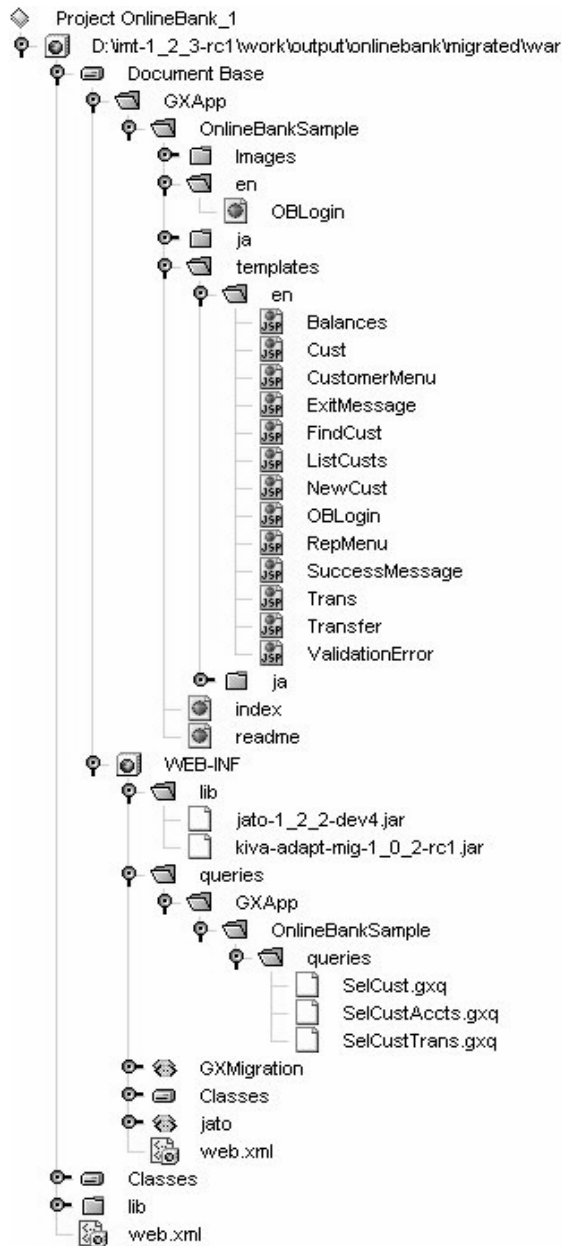


18. Right click [Filesystem] icon and select [Mount Directory] on the Explorer panel. Select `${migtbox_home}\work\output\onlinebank\migrated\war` and click OK. Forte should recognize this directory as a standard WAR directory and create a WAR view in the Explorer.

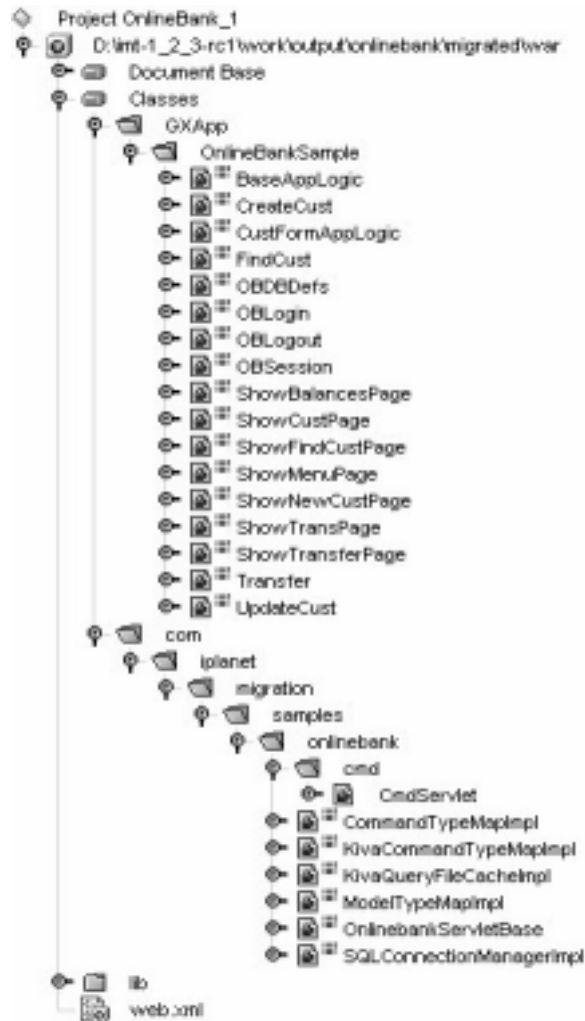


FFJ uses the term FILESYSTEM to refer to an entry in the CLASSPATH for a project. Upon mounting the WAR directory not only will the `./war/WEB-INF/classes` directory be AUTOMATICALLY part of the CLASSPATH because its a 'war' file, but each library under `./war/WEB-INF/lib` will also be added (ZIPs and JARs). See the Filesystems for the OnlineBank project below

Here is the document root of the new web application (see below). Notice that some static content has been translated to JSPs and the HTML templates have been translated to JSPs.



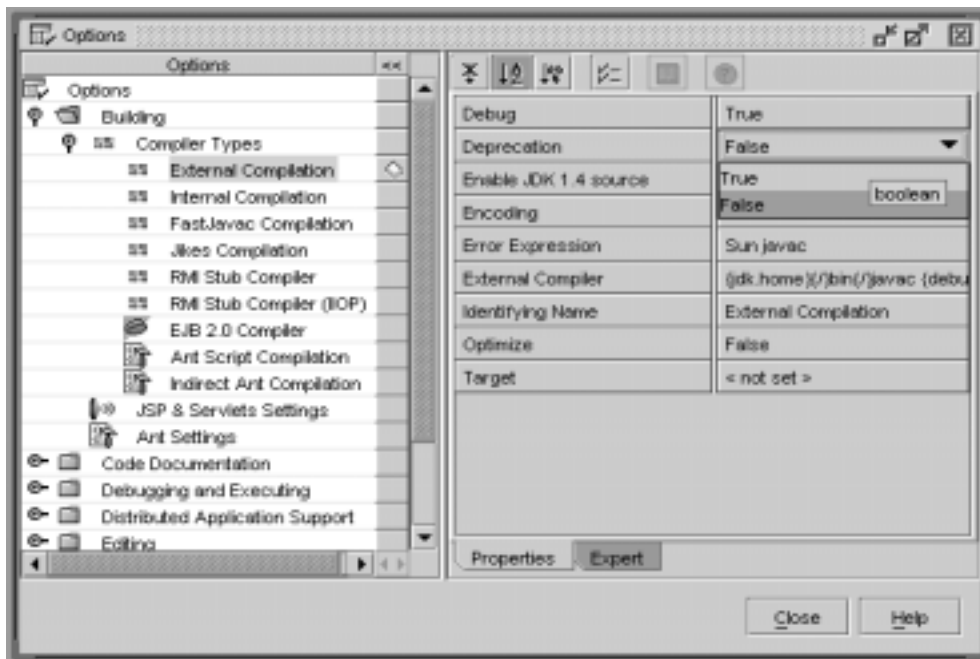
Here is the new layout of Java classes in the web application (see below). Notice that the original Java source retains original packaging. The AppLogics are translated to JATO commands and very little code is affected. The new JATO source files are placed in the new package specified as a property in the Translation tool.



The Java source will need to be compiled. It is very important to enable 'deprecation' flag in the compiler. The Translation tool automatically placed the debug or 'migration' version of the KFC adaption library in the WAR. When you compile your translated application using this library and the

'deprecation' flag is enabled, the compile will produce a report of each line of code which uses a 'non-targeted' API. The intention here is to reach a complete compilation as quickly as possible and produce a report on the tasks required for manual migration. Even if the application uses 'non-targeted' APIs, as long as it compiles it will run; although it may not function properly since non-targeted API are non-functional (e.g. return null or GXE.FAILURE). This is valuable because the migrator may incrementally migrate portions of the application and test these portions without being burdened with having to totally migrate the application. In other words, the migrated AppLogic JATO Commands may be tested one at a time. Another value proposition is that the deprecation report is a nice way to determine how much work there is to do.

19. Edit Project properties (Compiler: External Compiler:) and set deprecation to TRUE. Select [Tools] from menu and click [Options]. Expand the 'Building' and then 'Compiler Types' nodes and set [deprecation] as True for External Compilation on Options window as shown below:



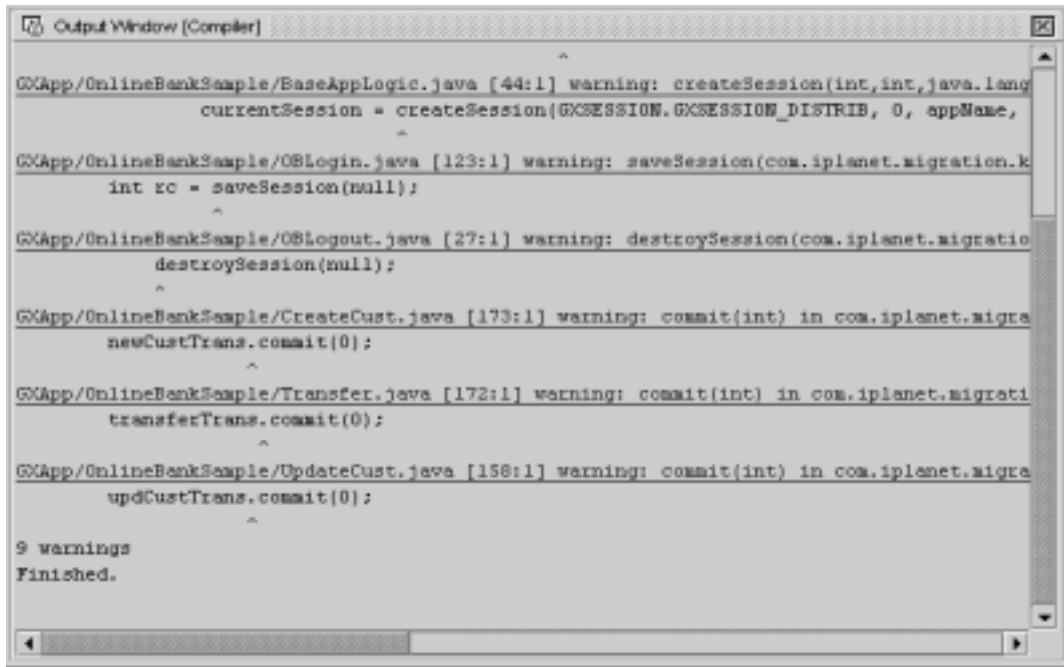
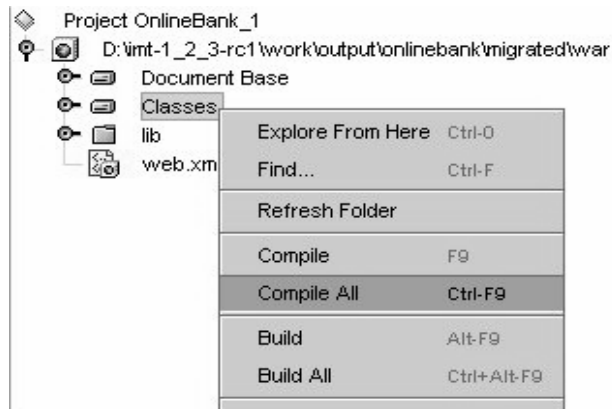
20. In the project view in the Explorer, select the Classes branch and right click to menu and choose Compile ALL. All the migrated code (AppLogics, etc.) in

```
{migtbox_home}\work\output\onlinebank\migrated\war\WEB-INF\classes\
```

and the new generated JATO infrastructure in

`${migtbox_home}\work\output\onlinebank\migrated\war\WEB-INF\classes\com`

Everything should compile immediately.



There are **six uses of NON-TARGETED SESSION KFC API's** in the OnlineBankSample and **three uses of the NON-TARGETED "commit" method of ITrans** in the version of the iMT.

The 'session' APIs are the most commonly found non-targeted APIs. In the KIVA Application Server and KFC, developers could optionally supply an ISessionIDGen reference to any of the 'session' APIs. This interface allowed the developer to control the Session ID and related behavior. There is no such capability in J2EE. Applications which used ISessionIDGen will need to manually redesign that portion of the application. Most developers chose not to use this feature by providing a 'null' object reference to the APIs. Nevertheless, since ALL the KFC 'session' APIs required this parameter and the ISessionIDGen type is non-targeted, ALL the KFC 'session' APIs are non-targeted too. There are alternative APIs provided for most of the non-targeted methods which do NOT require the ISessionIDGen parameter. The migrator will need to revise each case of non-targeted 'session' APIs so that the alternative APIs are used instead. Usually, these 'session' APIs are located in one or few places in the application so it should not be a costly manual change. Please note that there are two special cases in the 'session' APIs. The IAppLogic.saveSession(ISessionIDGen) does not provide an alternative method because there is no concept of 'saving or flushing' HttpSession in J2EE. This API is eliminated. The IAppLogic.createSession(int, int, String, String, ISessionIDGen) API provides an alternative API which takes zero parameters. Again, in J2EE, the Servlet API does not provide any control for the developer like the KFC API did; although the container vendor may provide value-added configuration or the HttpSession via deployment descriptor and app server configuration.

The single argument to the ITrans.commit method was never used by KIVA. We have eliminated this API for an Adapted API which takes zero arguments. You will need to remove the '0' value in the three commit methods in CreateCust.java Transfer.java and UpdateCust.java

21. In the OnlineBankSample application the 'session' APIs are used in BaseAppLogic.java, OBLogin.java, and OBLogout.java. The changes are shown below and are required to proceed.

BaseAppLogic.java LINE 38

```
ISession2 currentSession = getSession(); // getSession(0,  
appName, null);
```

BaseAppLogic.java LINE 44

```
currentSession = createSession();  
//createSession(GXSESSION.GXSESSION_DISTRIB, 0, appName,  
null, null);
```

OBSession.java LINE 52

```
// result = m_logic.saveSession(null);
```

OBLogin.java LINE 123

```
int rc = GXE.SUCCESS; // saveSession(null);
```

OBLogout.java LINE 27

```
destroySession(); // destroySession(null);
```

There will usually be manual modifications needed on the HTML source or even the HTML Template source (now JSPs). The modifications will be different for every application. The iMT alleviates most of the manual work for systematic tasks. Customers may find repeatable patterns and leverage the Regular Expression mapping tool to help automate their efforts. In most cases, the maintenance on the markup is in the area of URL paths. Links to static content from the dynamics content may suffer from invalid absolute paths caused by the addition of the web application context.

22. edit both parallel versions of the ExitMessage.jsp. The absolute reference to static content from the dynamic content are broken because we have moved the static content into the WAR file. These references would be correct if the content was deployed outside of the WAR file. Notice the [..] characters added to the beginning of the absolute URL. Because the ExitMessage.jsp is rendered from the context of the [/cmd] servlet mapping within the servlet context, we can get back to the document root of the servlet context by just moving one segment up in the path.

```
/GXApp/OnlineBankSample/templates/en/ExitMessage.jsp
```

```
/GXApp/OnlineBankSample/templates/ja/ExitMessage.jsp
```

LINE 15 (html -> jsp links and path) (see snippet below for English version)

```
href="../../GXApp/OnlineBankSample/en/OBLogin.html"> Back to Login  
Page </a><br>
```

23. *Optional* edit /WEB-INF/web.xml to allow for automatic startup when the root context is requested (see snippet below) You need to add welcome file elements between the servlet mappings and the taglib elements

```

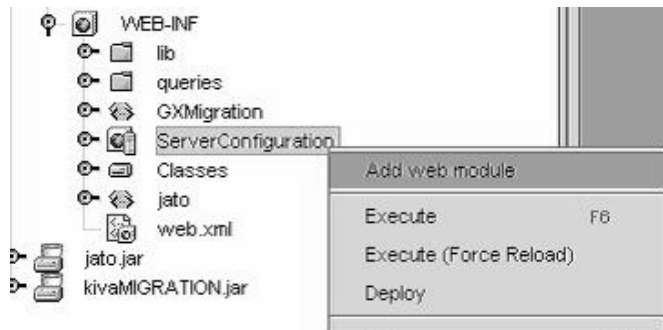
</servlet-mapping>
<welcome-file-list>
<welcome-file>
GXApp/OnlineBankSample/index.html
</welcome-file>
</welcome-file-list>
<taglib>

```

A major effort required during manual migration will be to verify URLs within the application. Links between static and dynamic content will usually need to be updated for the relative paths required for portable J2EE deployment. Also, JavaScript may need to be revised.

The manual migration effort is completed and the final web application may be deployed on any J2EE web container. In FFJ you may export a WAR file and deploy on iAS 6.5. You may also run the web application directly in FFJ using the built-in TomCat server.

24. Add a Server Module Group in FFJ. Right click on WEB-INF branch in Explorer, select [New]->[JSP&Servlet]->[Web Module Group] and add a server group. Accept the default on the wizard screen and simply chose 'Finish'. A new element under WEB-INF in the Explorer appears called 'ServerConfiguration'. Add the current web application by right clicking on [Server Configuration] branch in Explorer and select [Add Web Module]. Provide a servlet context name on [Add Web Module] window. For example "Demo".



25. Execute in FORTE by right clicking on [Server Configuration] branch in Explorer and select [Execute].

Migration from NetDynamics to Sun ONE AS 7

NetDynamics applications can be migrated to J2EE web modules using iPlanet Migration Toolkit (iMT 1.2.3). The resulting web modules can be deployed and executed on any J2EE web container.

Introduction

Before continuing, be sure you have read the `%MIGTBX_HOME%/bin/readme.txt` file so that you are aware of the latest information and any issues that might be relevant to you and your environment. The readme file also describes proper installation and configuration of the Migration Toolbox and its environment, which must be complete before beginning the migration process described in this document.

[`%MIGTBX_HOME%` represents the directory in which you installed/unzipped the iPlanet Migration Toolbox (iMT)].

This document covers the minimal process of migrating a NetDynamics application to J2EE. It is not intended to be an exhaustive reference for the migration process, in large part because there are only a few common aspects between any two migrations. Instead, this document provides the information necessary to understand the basic migration process using the iPlanet Migration Toolbox (iMT).

Migration Preparation

Migration Process Overview

There are two main phases to full migration of a NetDynamics proprietary project to its J2EE equivalent. These phases are the *automated migration phase* and the *manual migration phase*. The automated migration itself consists of two steps called *extraction and translation*.

Automated Migration Phase

This phase consists of manually preparing a NetDynamics project for migration and then using the iMT to perform automated extraction and translation. The input to this phase is a proprietary NetDynamics project or set of projects, and the result is a partially (or in some cases, fully) migrated application consisting entirely of non-proprietary J2EE-compliant components (servlets and JSPs).

The output from the translation process entirely replicates the component structure present in the original NetDynamics project. This process also uses the declarative property information present in the project's INTRP files to generate equivalent features in the migration application. However, the translation phase does not (currently) automatically port code written to the NetDynamics Spider API to its J2EE equivalent. This porting will be the primary task during the manual migration phase. The process does, however, place the original source code in the appropriate location in the new output. For example, code from the NetDynamics `onBeforeDisplay` event handlers is placed in the analogous event handler methods in the migrated application.

Manual Migration Phase

The degree of application migration accomplished in the automated phase is directly related to the amount of declarative versus API features used in the original application. In those rare cases where a project used entirely declarative features, that project can frequently be automatically migrated fully and be immediately deployable and runnable in a J2EE container without any manual work. Consequently, projects that use fewer declarative features will require more manual work to become functional as J2EE applications.

In general, the manual migration phase consists of reviewing the automatically migrated application output and porting Spider-API-specific code to J2EE-specific code. Understand that this process does not typically require a redesign of the application or its architecture; rather, it is largely a straightforward 1-to-1 mapping of API calls. This is possible because of the use of JATO, a powerful J2EE-compliant web application foundation targeted by the automated translation process.

Preparing your Working Environment

Before going further, ensure you've done the following:

1. Make sure you've installed the iPlanet Migration Toolbox.
 - Unzip the distribution archive into the desired target directory. Follow the directions in the `readme.txt` file.
 - Test the installation by trying to start the Toolbox application. Run `toolbox.bat` in the `%MIGTBX_HOME%/bin` directory. An empty Toolbox should appear after a few moments. If nothing appears, check that the Migration Toolbox was installed properly and that all appropriate environment settings have been set in `%MIGTBX_HOME%/bin/setenv.bat`.
2. To avoid class version issues, we strongly recommend that you remove all JAR files from your JDK's extension directory (`%JAVA_HOME%/jre/lib/ext`) while running the Toolbox application. We have included all the classes necessary for running the Toolbox with the distribution. Please note that simply renaming the JAR files in the extension directory is not sufficient; you must move them to a different location.
3. Copy the NetDynamics project(s) you wish to migrate into the `%MIGTBX_HOME%/work/NDProjects` directory (or any other convenient directory). This directory will be referred to as the *NetDynamics projects directory* below. This directory is not necessarily the actual project directory used by a NetDynamics installation on the same machine (although it could be). Instead, it is the directory in which you will place the NetDynamics projects to be migrated. Note that NetDynamics need not be installed on the machine running the Migration Toolbox. However, if NetDynamics is installed on the machine that will be used to run iMT, you must be sure that the installed NetDynamics does not interfere with the iMT. This will happen if the installed ND's classpath is referenced in the system environment variable called `CLASSPATH`. When iMT is started, it appends its own necessary classpaths to the end of the system classpath. If the installed ND's classpath is part of the system classpath, then the iMT will not operate properly.
4. At this point, you may also want to install Sun ONE Application Server 7 or another J2EE-compliant servlet/JSP container
 - Follow the installation instructions for the server or container
 - Test the installation by starting the server or container and trying to load the default home or index page. If an error occurs, troubleshoot the installation process before continuing

Preparing a Project for Automated Migration

Because NetDynamics allowed developers immense latitude (with both positive and negative consequences), there is no way for iMT to account for all possible project permutations. This is particularly true of projects that use non-standard portions of the NetDynamics Spider API, or use this API in an unorthodox or undocumented way. Therefore, some applications will require manual preparation before being migrated by the iMT. In some cases, this preparation may be significant if a particular problematic feature is widespread throughout a project or set of projects.

Of the two automated phases, you are more likely to encounter initial difficulties during project extraction. This is normal, and is simply a consequence of the issues noted above. The good news is that many projects will not encounter any difficulties during extraction, and once an application description has been extracted from a project, it should be translatable with little or no difficulty.

Differences Between the Project Extraction Runtime and NetDynamics Runtime Environments

The iMT uses an embedded NetDynamics Connection Processor (CP) to instantiate and then extract information from a project. From the project's perspective, it is being instantiated inside a normal NetDynamics 5.x server environment. However, the extraction runtime environment differs substantially from that present in a NetDynamics server. Specifically, the JDBC Service, the PE Service, and PACs are not available to applications instantiated within the iMT's embedded runtime, nor are they necessary to extract the necessary information.

We have found that some project objects perform tasks that depend on these runtime features in their constructors, static initializers, initialization events, or non-Spider threads. The iMT automatically suppresses the firing of the NetDynamics 4/5.x-style `onBeforeInit` and `onAfterInit` events so that customer code in those events will not execute during the initialization. However, other initialization-time methods, such as static initializers, overridden `init()` methods, and NetDynamics 3.x-style `onBeforeInit` and `onAfterInit` events may still execute. You may need to comment out code in these methods if that code attempts to perform behaviors that cannot complete successfully within the iMT runtime. (You may leave the code in the original location and it will be automatically moved to the correct target location during translation). One can normally identify these problematic cases most easily from error messages and exceptions generated by the *Extraction Tool*.

Before Running the NetDynamics Extraction Tool

For the reasons given above, we generally advocate running the *Extraction Tool* on your project with only minimal preparation. Although it is more likely that the extraction will fail with an error, doing so will typically save you time in the overall migration process, it is usually easier and faster to detect and rectify problems using the diagnostic error information than trying to find and fix potential problems preemptively (unless potential problems are well-known).

However, to avoid several other common sources of extraction difficulties, we recommend you perform the following tasks before running the NetDynamics Extraction Tool:

- We have found instances of NetDynamics projects that appeared normal when opened with the Studio or run in the NetDynamics server, but in reality contained corrupted references and project objects that were only detected upon closer inspection. In other cases, we have found corrupted class files that prevented the embedded NetDynamics runtime from loading the corresponding project object and caused it to throw seemingly unrelated exceptions. Therefore, we strongly recommend you follow these steps to prevent trouble before beginning migration:
 - If the project came from another source (such as a client or colleague), ensure the projects `links` directory is present and contains a number of `.sid` files. You may open several of these files in a text editor and ensure that the objects named in the file correspond to the names of the project objects. Also ensure that all necessary external classes were included with the project.
 - The project must have been converted to NetDynamics 5 using the Studio's automated conversion process. This process entails opening the project in the NetDynamics 5 Studio and following the upgrade prompts. During conversion, the Studio upgrades object properties and converts DataObjects to NetDynamics 5.x-compatible versions. **IMPORTANT:** Note that the project need not actually run under NetDynamics 5.xósimply using the Studio to convert the project is sufficient.
 - Open the projects you will be migrating in the NetDynamics 5.x Studio and inspect them for completeness and validity. Also inspect the project directory itself. For example, you should have one `<project>.spj` (or `<project>Project.spj`) and `<project>.class` file, one `<page>.spg`, `<page>.class`, and `<page>.html` file per NetDynamics page, and one `<dataobject>.sdo` and `<dataobject>.class` file per DataObject.

- Delete all `.class` and `.ser` files from the project directory and fully recompile the project. The project must be compiled against the NetDynamics 5.x binaries. The easiest way to do this is to use the "Compile All" command in the Studio. You may also be able to use the Java Compilation Tool in the Migration Toolbox application to compile a project using the NetDynamics 5 binaries, though this is not recommended and may require substantially more configuration.
- If possible, test run the project in NetDynamics 5.x. A project that runs successfully in the server is more likely to be migratable without trouble. If you have a running copy of NetDynamics, configure the CP to preload the projects you will be migrating. Use the Command Center to stop or remove the JDBC Service, the PE Service, and all PACs from the current configuration. Restart the CP. After the CP starts successfully, check the NetDynamics log and the Service Manager (SM) log to determine if any exceptions were thrown. Projects that throw exceptions at this point are likely to throw exceptions during extraction.

Migrating ToolBox Sample Application

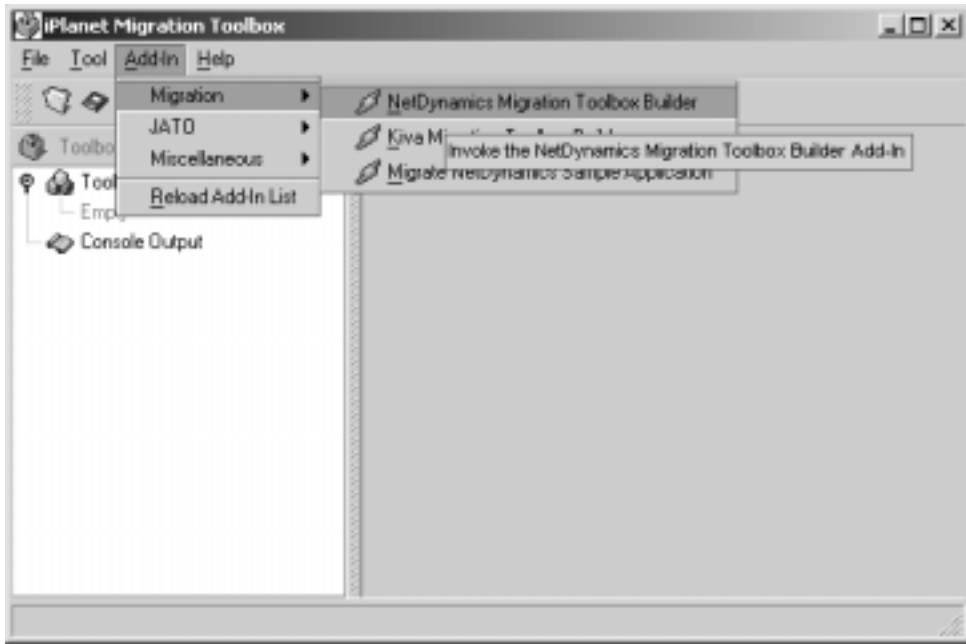
This section describes the automated and manual migration procedures of the ToolBox sample application.

Running the Migration Toolbox

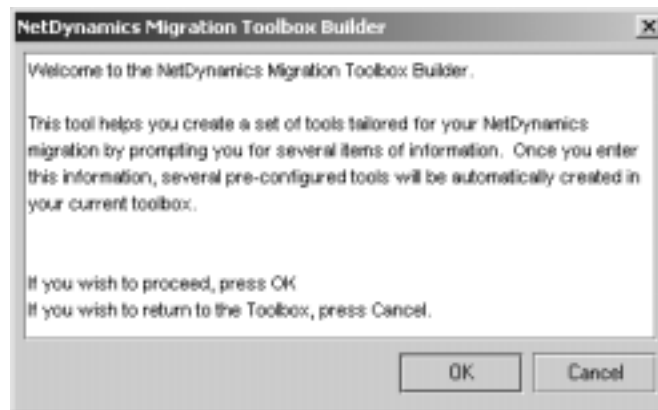
If you don't have the Toolbox application currently running, please follow the instructions given in section "Preparing your Working Environment" to setup your toolbox.

Create a Toolbox Builder

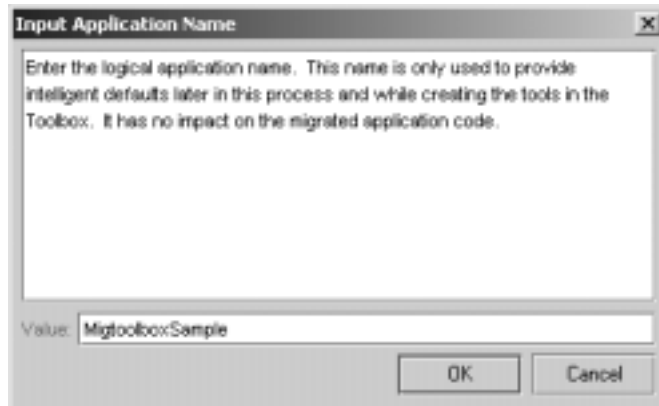
1. Start the toolbox and choose "Migrate an application" option in the Welcome dialog and press OK. With the Toolbox running, be sure that you have an empty (New) toolbox. Select the menu option `Add-In -> Migration -> NetDynamics Migration Toolbox Builder`.



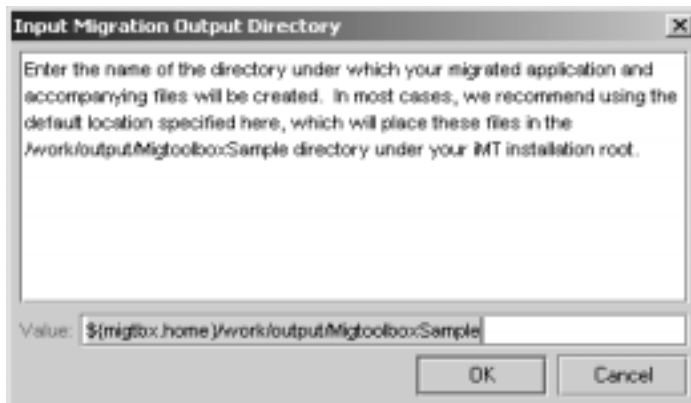
A modal dialog wizard will appear.



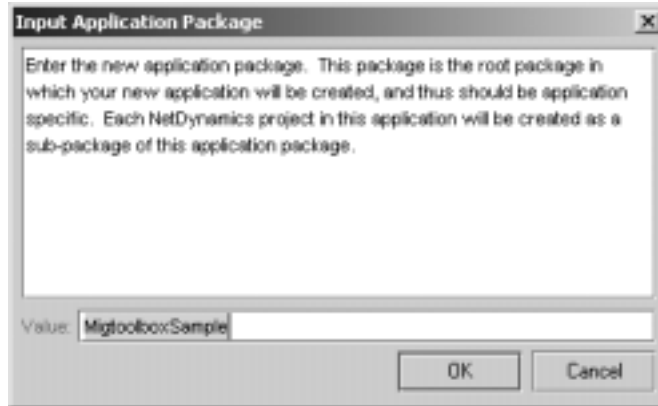
Select OK to proceed to the first step of the wizard.



2. Enter the name of the application you will be migrating in the “Input Application Name” dialog box, for e.g. ‘MigtoolboxSample’. Select OK to proceed to the next step.

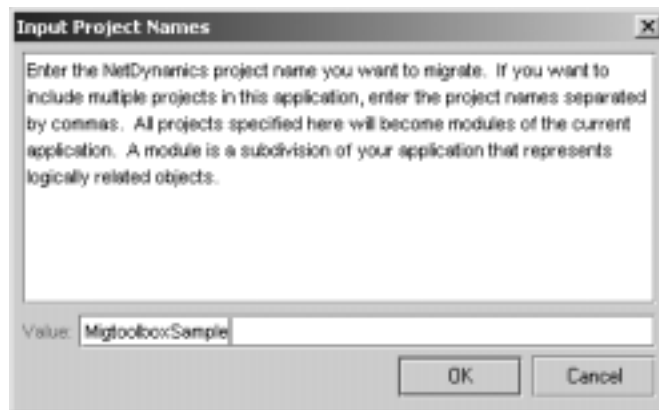


3. Enter the directory where all materials generated by the iMT will be stored. The default is usually satisfactory and is used in this example. Select OK to display the next step in the wizard

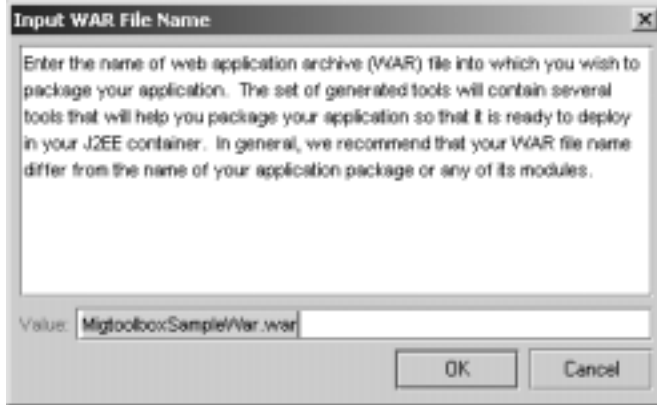


4. Automated iMT migration will produce some J2EE infrastructure including new Java JATO files. These new files must be assigned a package. Although existing Java source in the original application will retain packaging, we still need to assign a package for these new files. There is no restriction on the package name. The default value is provided for the MigtoolboxSample application.

Enter a package and select OK to display the next step in the wizard.



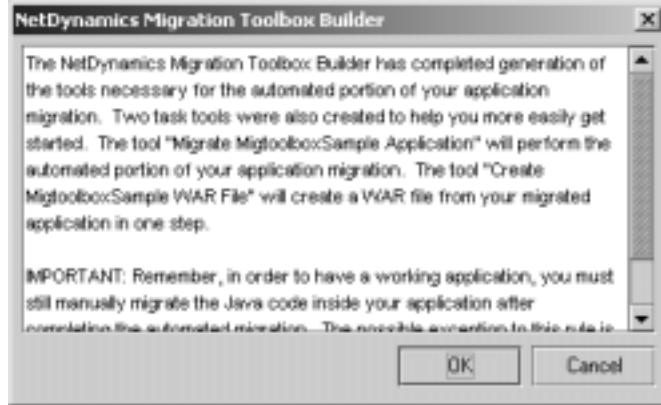
5. Enter the project name you want to migrate. This project should be located in '{MIGTBX_HOME}\work\NDProjects\' folder.



6. Enter the name of the web application archive (WAR) file into which you want to package your application. The default value is provided for this application. Select OK to proceed to the next step.

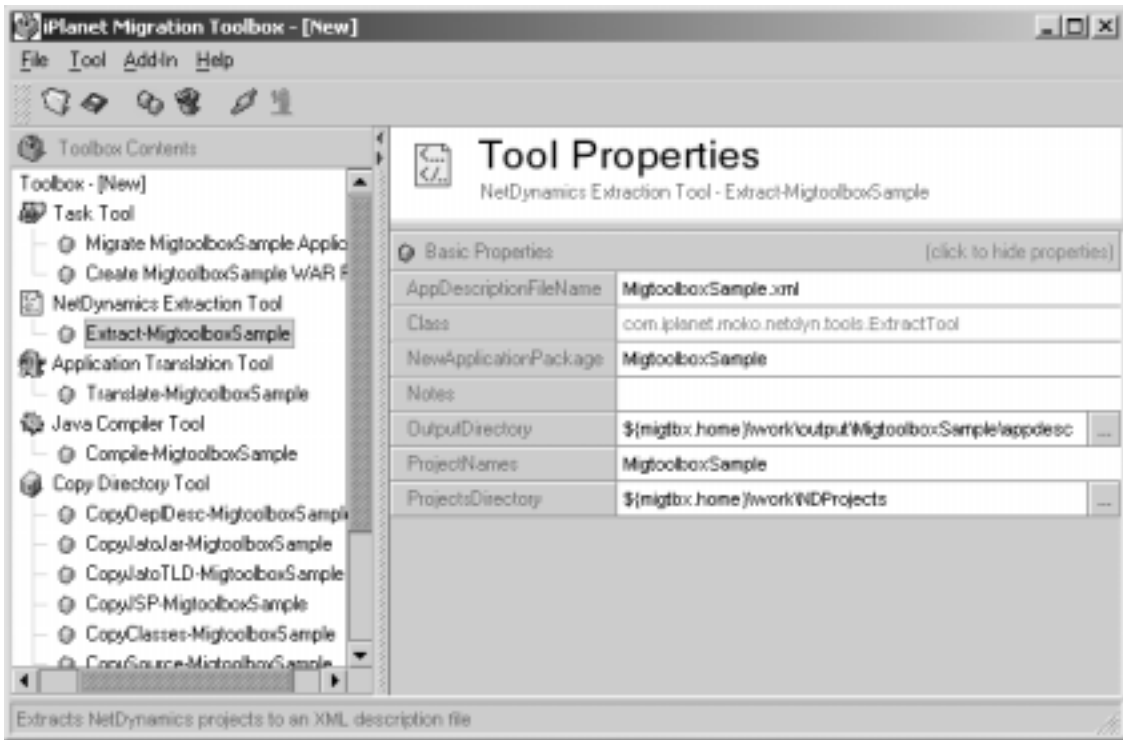


7. Enter the output directory name where iMT will generate the WAR file. When you select OK on the dialog box, the toolbox builder will generate a set of tools necessary for the automated portion of the application migration process.

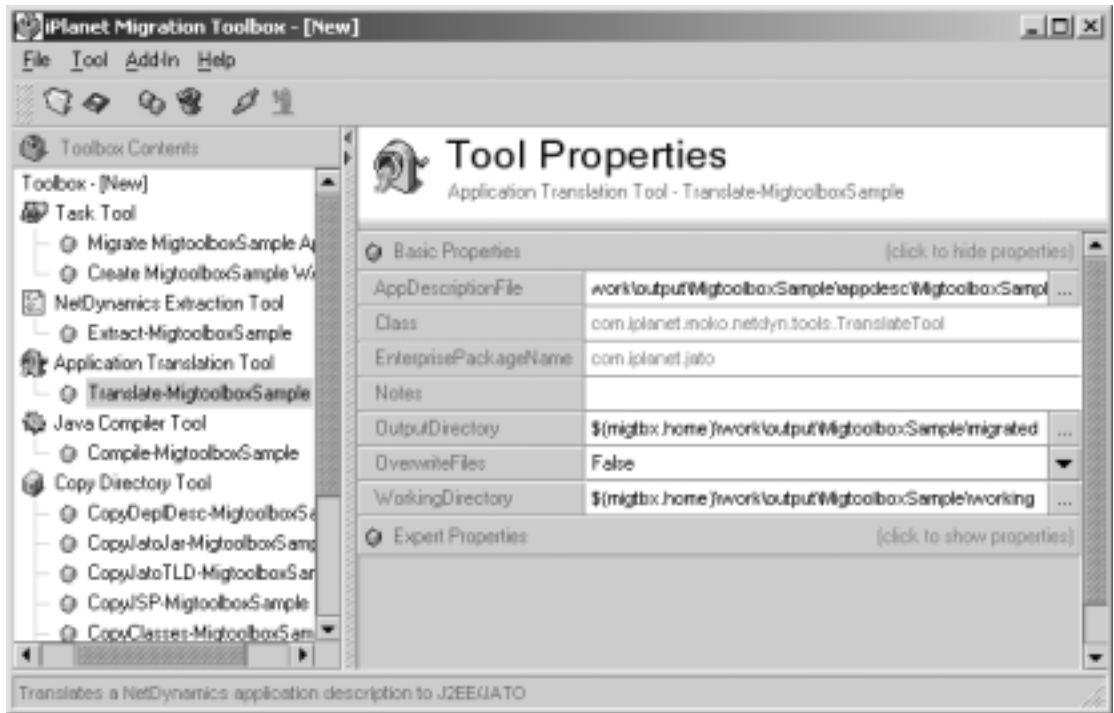


8. Select OK to exit the NetDynamics Migration Toolbox Builder wizard. The result of the Addin is a complete Toolbox consisting of an Extraction and Translation tool and the optional tools to automatically create the application extract archive and translate the documents. When you select the 'branch' for each tool on the left frame, it will display the detailed help for each tool in the right frame. The help explains each property in the tool. Click on each 'instance' of the tools to display the bean property panel in the right frame. Both the basic and expert properties may be edited.

The Task Tools simply cause a list of other tools to be executed in order. It is usually more informative to run the tools separately so that you can carefully watch the console output. The extraction tool properties are shown here:



The Translation tool properties are shown here:



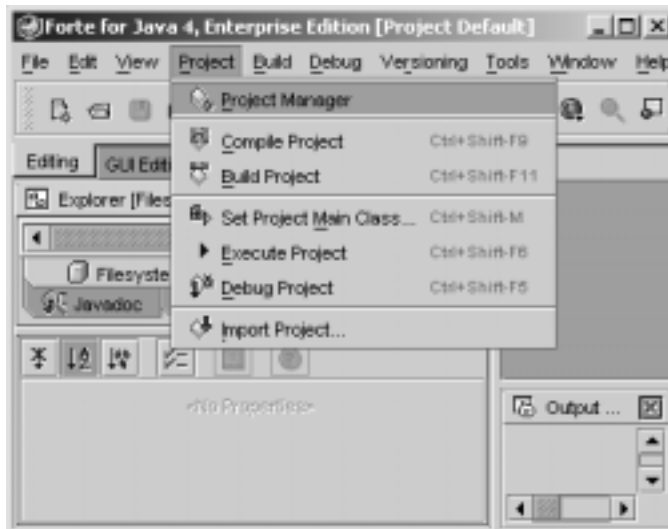
9. Invoke the Migrate MigtoolboxSample Application Task tool. This tool will in turn invoke Extract-MigtoolboxSample, Translate-MigtoolboxSample and MapSpider2JATO-MigtoolboxSample tools one after the other to produce the migrated code and the *application description* file (`MigtoolboxSample.xml`).
10. Invoke the Create MigtoolboxSample War File Task tool. This tool will invoke the following tools to produce a Web Application Archive (WAR) file to enable automatic deployment of the application to a J2EE container. This WAR file will be the only file you will need to deploy your application to the J2EE container.

CopyDeplDesc-MigtoolboxSample, CopyJatoJar-MigtoolboxSample, CopyJatoTLD-MigtoolboxSample, CopyJSP-MigtoolboxSample, CopyClasses-MigtoolboxSample, CopySource-MigtoolboxSample, JarWarFile-MigtoolboxSample
11. Invoke the Compile-MigtoolboxSample tool to compile the JATO Foundation classes and the new J2EE application components. This tool simply invokes the `javac` command line tool provided with the JDK.

At this point, automated migration is complete and manual migration if any starts.

The easiest way to proceed in manual migration is to load the web application into a J2EE IDE. Forte for Java EE (FFJ) is used in this example.

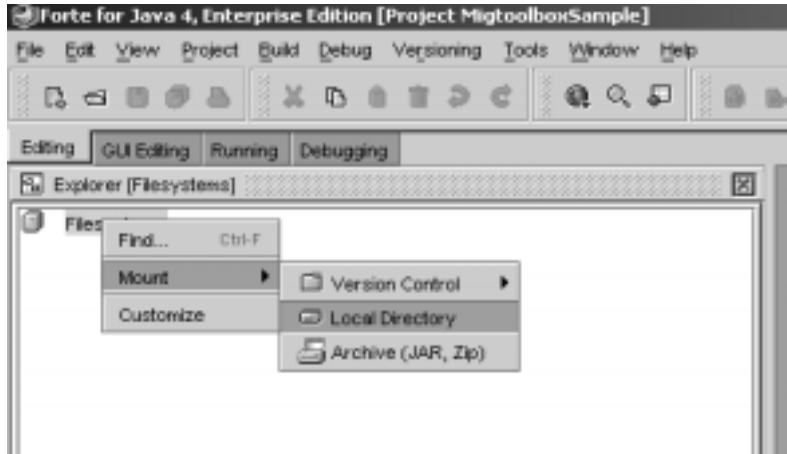
12. Start FFJ 4.0 and create a new Project called OnlineBank. Make sure there are no existing file systems in the new project. Select [Project] from menu and click [Project Manager].



13. On the Project Manager window, click New and put a project name (MigtoolboxSample).



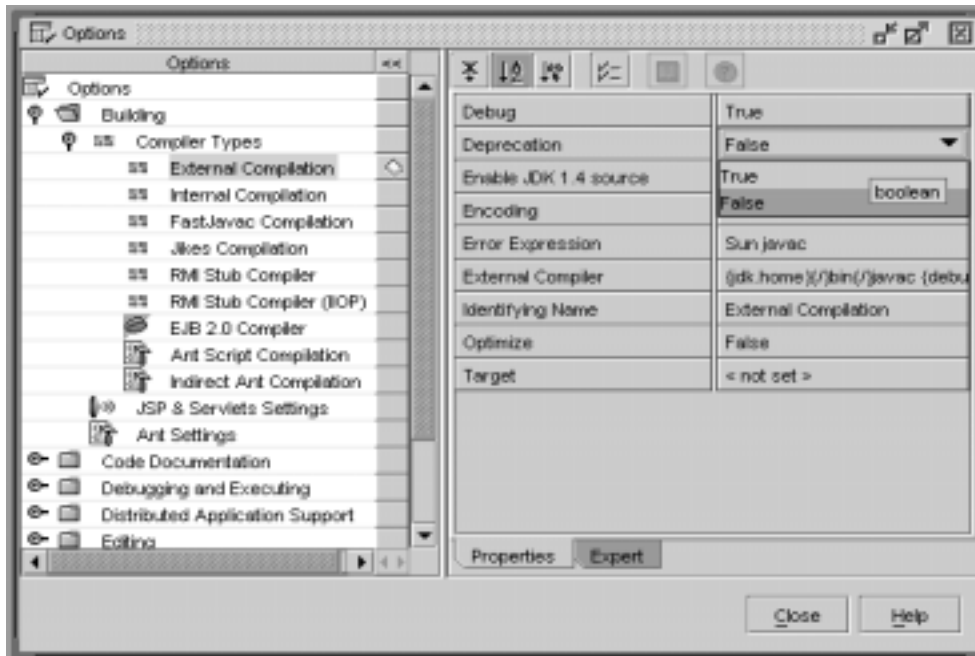
14. Right click [Filesystem] icon and select [Mount Directory] on the Explorer panel. Select `{migtoolbox_home}\work\output\MigtoolboxSample\war` and click OK. Forte should recognize this directory as a standard WAR directory and create a WAR view in the Explorer.



FFJ uses the term `FILESYSTEM` to refer to an entry in the `CLASSPATH` for a project. Upon mounting the WAR directory not only will the `./war/WEB-INF/classes` directory be AUTOMATICALLY part of the `CLASSPATH` because its a 'war' file, but each library under `./war/WEB-INF/lib` will also be added (ZIPs and JARs).

The Java source will need to be compiled. It is very important to enable 'deprecation' flag in the compiler. When you compile your translated application and the 'deprecation' flag is enabled, the compile will produce a report of each line of code which uses a 'non-targeted' API. The intention here is to reach a complete compilation as quickly as possible and produce a report on the tasks required for manual migration.

15. Edit Project properties (Compiler: External Compiler:) and set deprecation to TRUE. Select [Tools] from menu and click [Options]. Expand the 'Building' and then 'Compiler Types' nodes and set [deprecation] as True for External Compilation on Options window as shown below:



16. In the project view in the Explorer, select the Classes branch and right click to menu and choose Compile ALL. All the migrated code in

```

${migtoolbox_home}\work\output\MigtoolboxSample\war\WEB-INF\classes
\

```

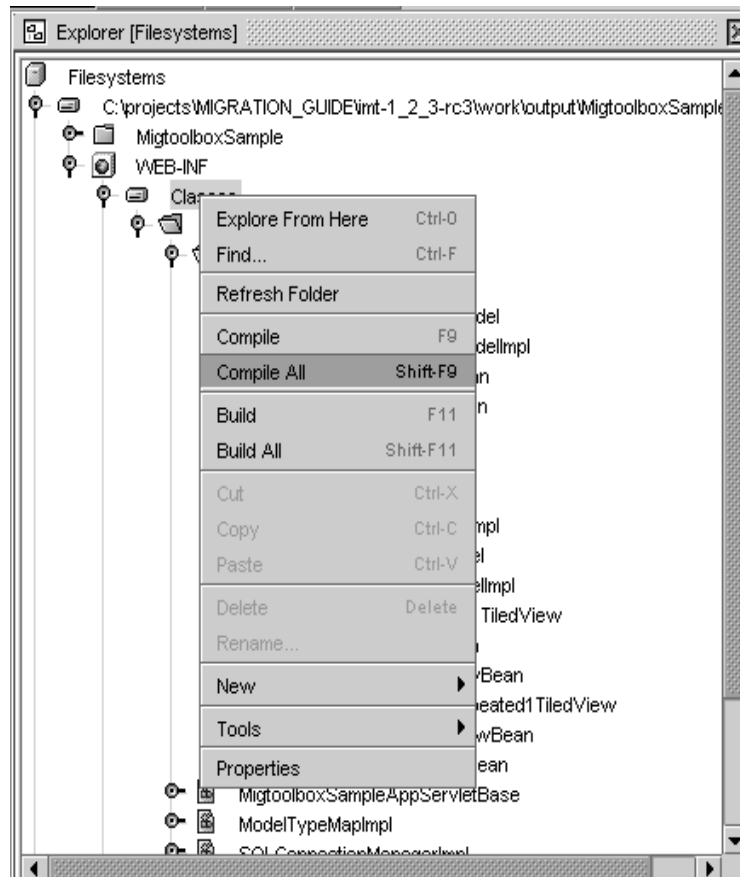
and the new generated JATO infrastructure in

```

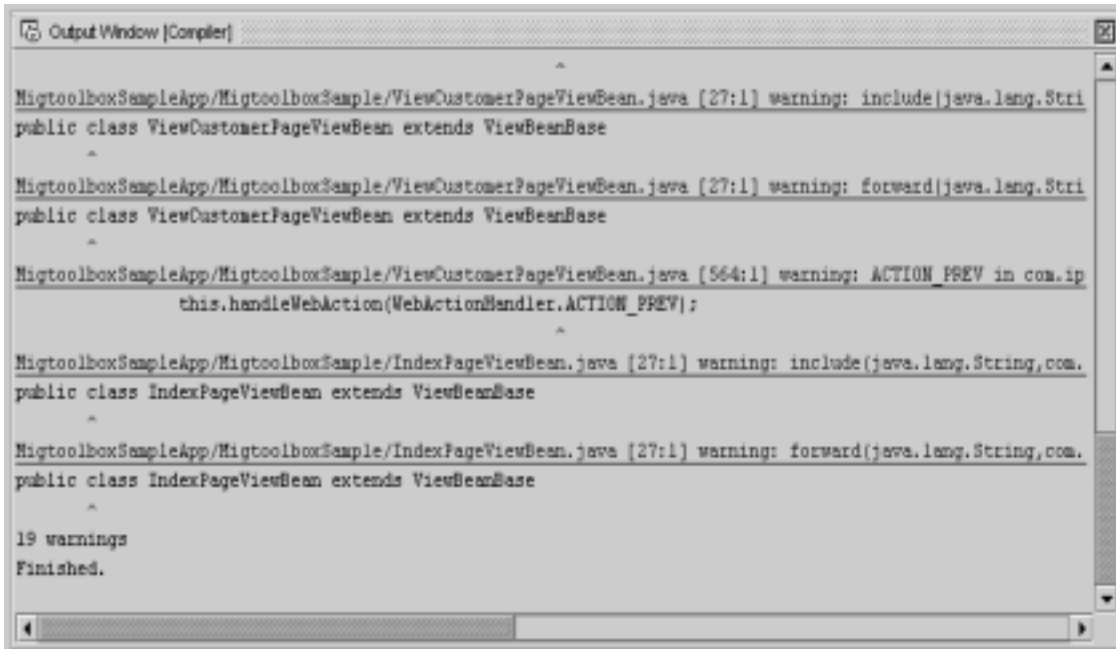
${migtoolbox_home}\work\output\MigtoolboxSample\war\WEB-INF\classes
\

```

Everything should compile immediately.



The compiler generates some warnings, they are shown here:



```
Output Window [Compiler]

^
MigtoolboxSampleApp/MigtoolboxSample/ViewCustomerPageViewBean.java [27:1] warning: include(java.lang.Stri
public class ViewCustomerPageViewBean extends ViewBeanBase
^
MigtoolboxSampleApp/MigtoolboxSample/ViewCustomerPageViewBean.java [27:1] warning: forward(java.lang.Stri
public class ViewCustomerPageViewBean extends ViewBeanBase
^
MigtoolboxSampleApp/MigtoolboxSample/ViewCustomerPageViewBean.java [564:1] warning: ACTION_PREV in com.ip
this.handleWebAction(WebActionHandler.ACTION_PREV);
^
MigtoolboxSampleApp/MigtoolboxSample/IndexPageViewBean.java [27:1] warning: include(java.lang.String,com.
public class IndexPageViewBean extends ViewBeanBase
^
MigtoolboxSampleApp/MigtoolboxSample/IndexPageViewBean.java [27:1] warning: forward(java.lang.String,com.
public class IndexPageViewBean extends ViewBeanBase
^
19 warnings
Finished.
```

17. These warnings should be fixed to complete the manual migration of the application. The final web application may be deployed on any J2EE web container. In FFJ you may export a WAR file and deploy on Sun ONE Application Server 7. You may also run the web application directly in FFJ using the built-in TomCat server.
18. Add a Server Module Group in FFJ. Right click on WEB-INF branch in Explorer, select [New]->[JSP&Servlet]->[Web Module Group] and add a server group. Accept the default on the wizard screen and simply chose 'Finish'. A new element under WEB-INF in the Explorer appears called 'ServerConfiguration'. Add the current web application by right clicking on [Server Configuration] branch in Explorer and select [Add Web Module]. Provide a servlet context name on [Add Web Module] window. For example "Demo".



19. Execute in FORTE by right clicking on [Server Configuration] branch in Explorer and select [Execute].

Automating Migration

This chapter describes the use of available migration tools that can be used to automate the migration process from both earlier versions of Sun ONE Application Server and from other application server providers.

The following migration tools are available:

- Sun ONE Migration Tool for Application Servers
- Sun ONE Migration Toolbox (formerly iPlanet Migration Toolbox)

Sun ONE Migration Tool for Application Servers

The Sun ONE Migration Tool for Application Servers migrates J2EE[tm] applications from other server platforms to Sun ONE Application Server (version 6.5 / 7).

For Sun ONE Application Server 6.5 the following source platforms are supported:

- WebSphere Application Server (WAS) 4.0
- WebLogic Application Server (WLS) 5.1

For Sun ONE Application Server 7 the following source platforms are supported:

- WebLogic Application Server (WLS) 5.1, 6.0, 6.1
- WebSphere Application Server (WAS) 4.0
- J2EE Reference Implementation Application Server (RI) 1.3

- Sun ONE Application Server 6.x
- Sun ONE Web Server 6.0

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. The latest version of the Sun ONE Migration Tool for Application Servers can be downloaded from Sun Download center. For the latest on Sun ONE Migration Tool please visit, <http://www.sun.com/migration/sunonetools.html>.

Sun ONE Migration Toolbox (formerly iPlanet Migration Toolbox)

For information on Sun ONE Migration Toolbox, please refer to “Appendix B”

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 ONE Application Server *Administrator's 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 Sun ONE Migration Tool for Application Servers, 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 described in the Sun ONE Application Server *Administrator's Guide*.

iBank Application specification

The sample application we defined is called 'iBank' and 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).

For packaging and deployment of application components, standard J2EE methods are used, and include definition of deployment descriptors, and packaging of application components within 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 packaging 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 packaging methods will be useful in pointing out any differences between Sun ONE Application Server 6.0/6.5 and Sun ONE Application Server 7, and any issues arising thereof.

Tools used for the development of the application

Sun ONE Studio Enterprise Edition for Java, Release 4.0

As Sun ONE Application Server 7 supports both the EJB 1.0 and EJB 1.1 standard, the other EJBs in the iBank application (2 session EJBs and the BMP entity bean) were developed with Sun ONE Studio for Java, and then packaged and deployed in Sun ONE Application Server 7 using the supplied Application Assembly Tool. This approach enabled us to test usage of a third-party IDE for developing 1.1 EJBs in Sun ONE Application Server 7. Moreover, the approach also gave us the chance to experiment with migrating 1.1 EJBs developed for Sun ONE Application Server 6.5 to Sun ONE Application Server 7.

The Sun ONE Studio for Java development environment was also used to migrate EJB components in the iBank application to Sun ONE Application Server (code adapted from EJB 1.0 standard to EJB 1.1, O/R mapping for CMP entity beans, configuration of deployment properties and packaging of the application's different modules).

Oracle 8i 8.1.6

The database was developed with Oracle 8i (version 8.1.6) and the JDBC driver used to access the database was the thin Oracle driver (type 4).

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 no., and also holds the no. 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 no., 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.

These business rules are illustrated in the entity-relationship diagram below:

TMBank -- Database schema

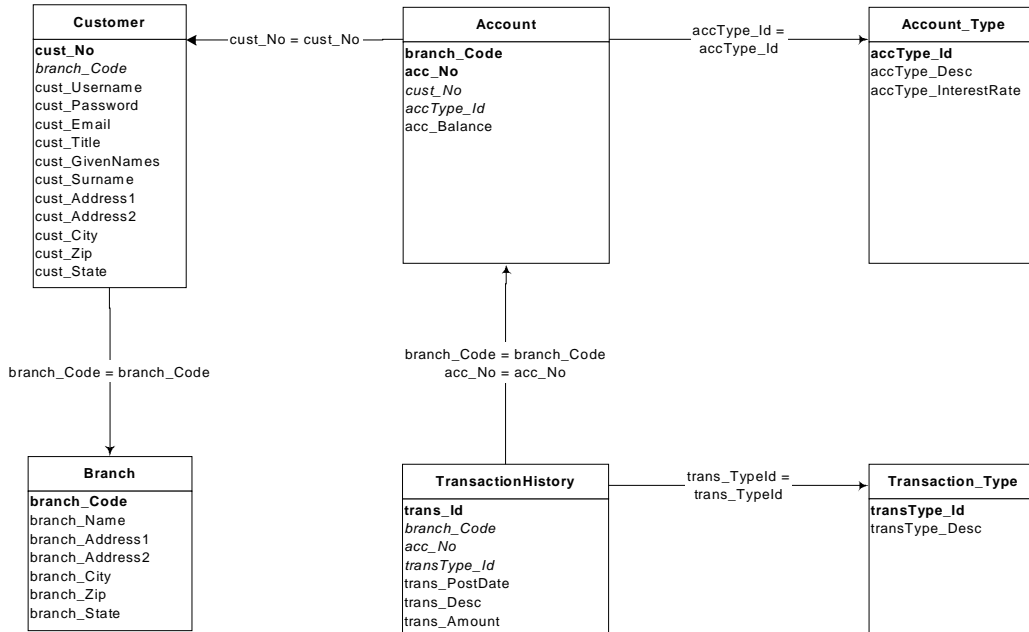


Figure 1 Database Schema

The database model translates as the 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

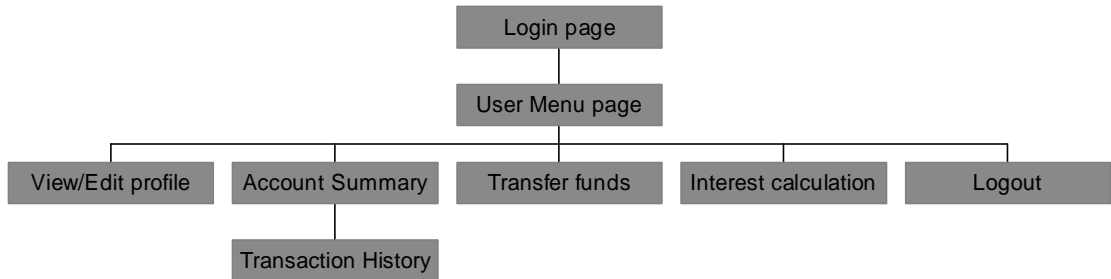
ACCOUNT			
BRANCH_CODE	CHAR(4)	NOT NULL	branch code (primary-key part 1)
ACC_NO	CHAR(8)	NOT NULL	account no. (primary-key part 2)
CUST_NO	INT	NOT NULL	Customer to whom accounts belongs
ACCTYPE_ID	CHAR(3)	NOT NULL	Account type, references ACCOUNT_TYPE
ACC_BALANCE	DECIMAL(10,2)	DEFAULT 0.0	Cleared balance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

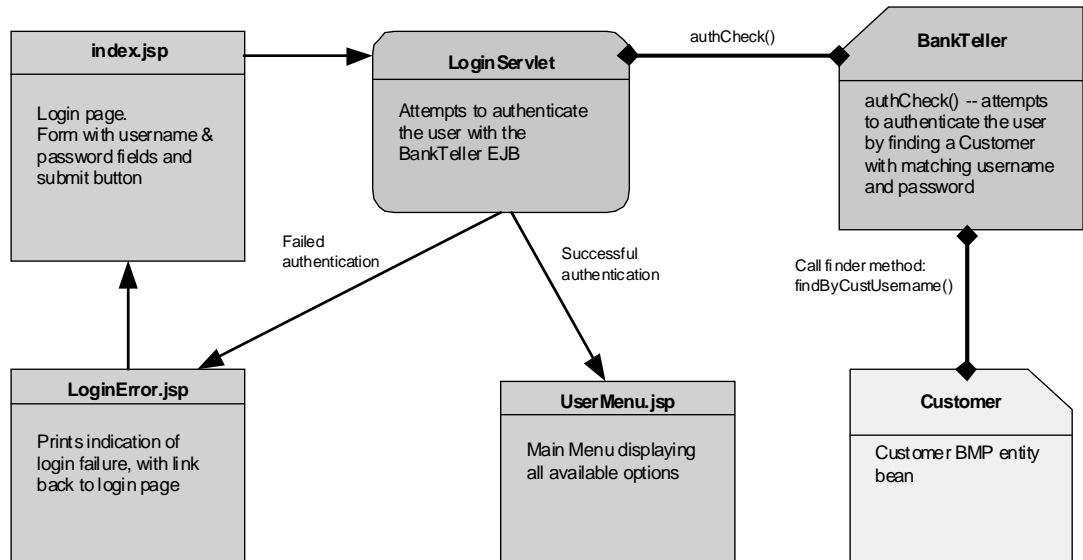
Application navigation and logic

High-level view of application navigation

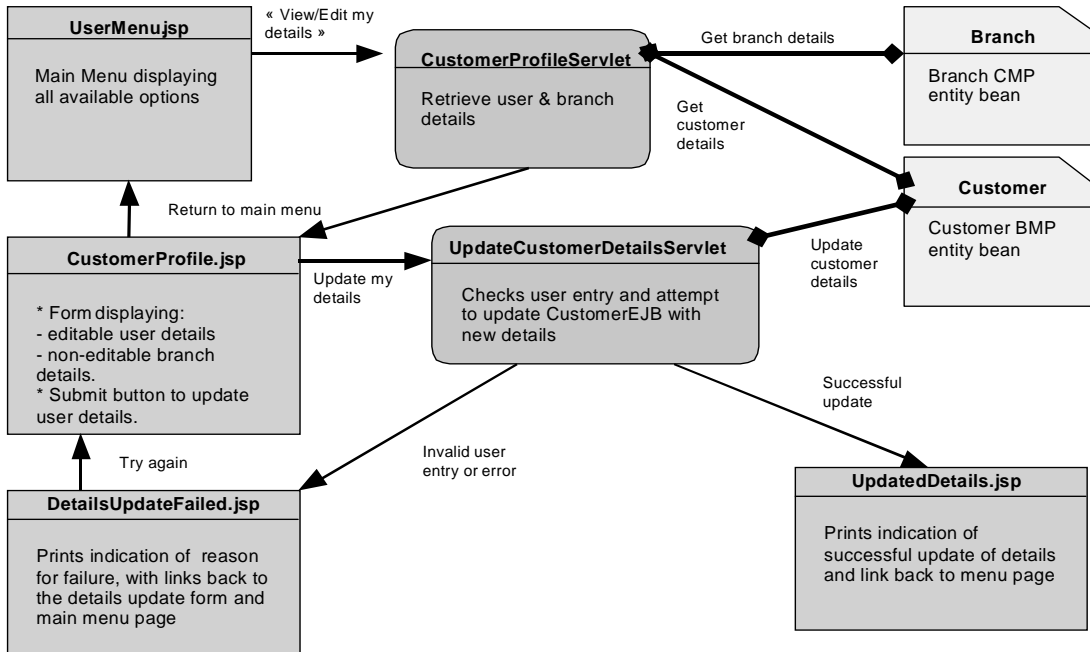


Detailed application logic

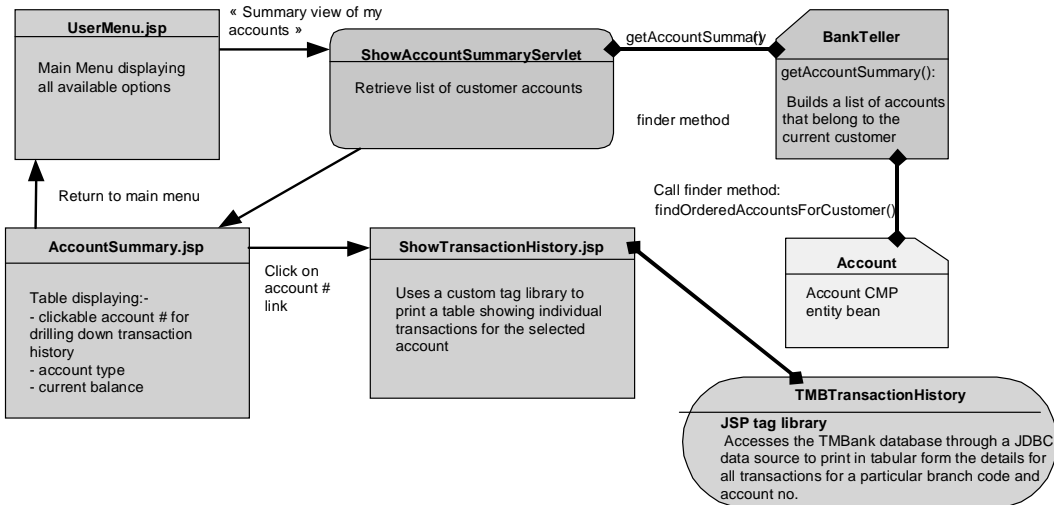
- **Login Process**



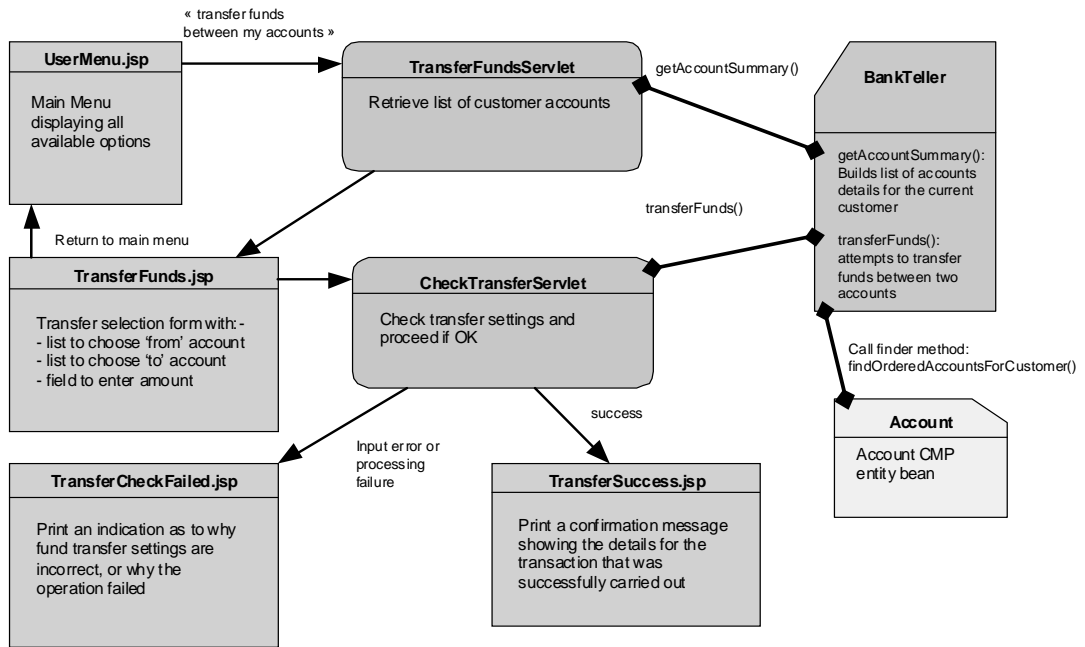
- **View / edit details**



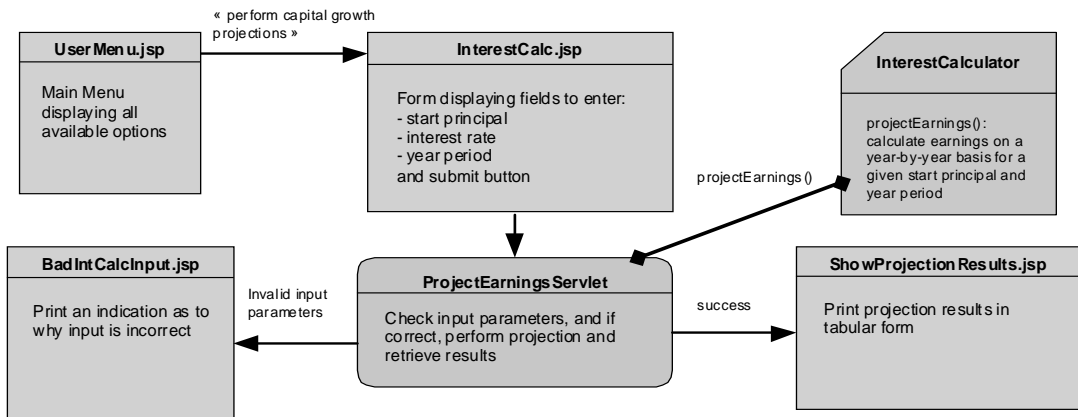
• Account summary and Transaction history



• **Fund Transfer**



• **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:

- 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 (<code>authCheck()</code> 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 (<code>UserMenu.jsp</code>)
CustomerProfileServlet	Retrieves customer and branch details from the Customer and Branch entity beans and forwards request to the view/edit details page (<code>CustomerProfile.jsp</code>)
UpdateCustomerDetailsServlet	Attempts to effect customer details changes amended in <code>CustomerProfile.jsp</code> by updating the Customer entity bean after checking validity of changes. Redirects to <code>UpdatedDetails.jsp</code> if success, or to <code>DetailsUpdateFailed.jsp</code> in case of incorrect input.
ShowAccountSummaryServlet	Retrieves the list of customer accounts from the BankTeller session bean (<code>getAccountSummary()</code> method) and forwards request to <code>AccountSummary.jsp</code> for display
TransferFundsServlet	Retrieves the list of customer accounts from the BankTeller session bean (<code>getAccountSummary()</code> method) and forwards request to <code>TransferFunds.jsp</code> 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 <code>transferFunds()</code> method of the BankTeller session bean to perform the transfer operation. Redirects the user to <code>CheckTransferFailed.jsp</code> in case of input error or processing error, or to <code>TransferSuccess.jsp</code> 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.
DetailsUpdateFailed.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
ShowTransactionHistory.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.
TransferCheckFailed.jsp	When the user chooses incorrect settings for fund transfer, they get redirected to this page.
TransferSuccess.jsp	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.jsp	If the parameters for compound interest calculation are incorrect, the user gets redirected to this page.
ShowProjectionResults.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 a "real-world" context, care was taken to ensure that these choices enabled 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 one can 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, the white paper specifically looks at the following technologies:

Servlets

- iBank 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 initialisation 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 `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 data source, both programmatically (BMP entity bean, BankTeller session bean, custom tag library) and declaratively (with the CMP entity beans).

Enterprise Java Beans

iBank uses a variety of Enterprise Java Beans:

Entity beans:

Bean-managed persistence ("*Customer*" bean): that allows us to test:

- 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): that allow us to test:

- 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:

- using and deploying a stateless session bean
- calling a business method for calculations

Stateful session beans: *BankTeller* allows us to test:

- 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 Packaging

iBank is packaged following J2EE standard procedures, using:-

- a Web application archive file for the Web application module, and EJB-JAR archives for EJBs.
- an Enterprise application archive file (EAR file) for the final packaging of the Web application and EJB modules.

Sun ONE Migration Toolbox

Sun ONE Migration Toolbox (S1MT) is used primarily to migrate applications built on NetDynamics or Kiva/NAS platforms to Sun ONE Application Server or any J2EE compatible containers. The main interface for the Sun ONE Migration Toolbox is what we call the *Toolbox application*, or the *Toolbox GUI*. This application can be invoked by running the `%MIGTBX_HOME%/bin/toolbox.bat` script (provided the `setenv.bat` file has been customized appropriately, see `README.txt` for more information).

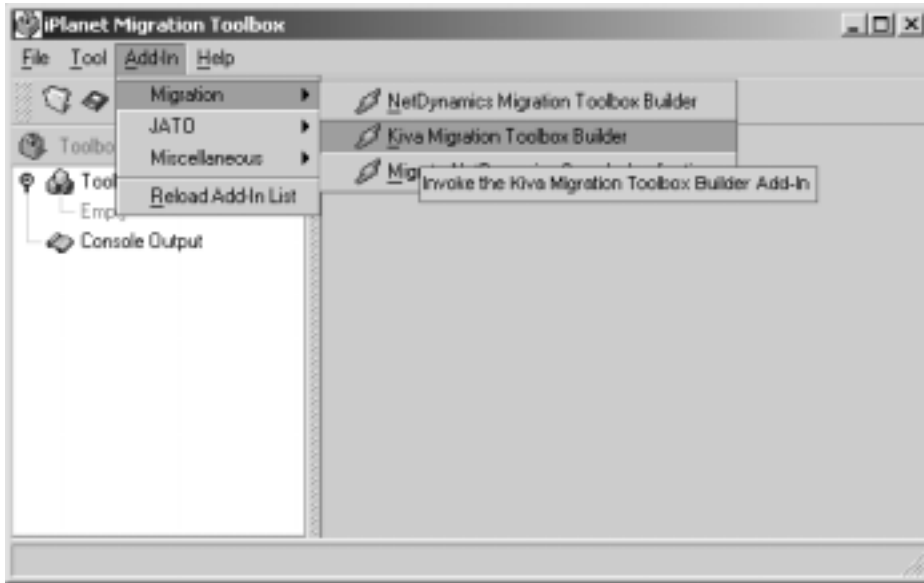
Supported Platforms

Microsoft Windows NT 4.0 and Windows 2000 currently support S1MT. Although it is expected that the application can be run on other Win32 platforms (Windows 95/98/Me), these platforms have not been tested and may require additional configuration beyond that specified in the S1MT installation documentation.

The Toolbox require atleast JavaSoft JDK 1.2.2 (JDK 1.3.1 has been tested) to run successfully.

Migration

The toolbox is set of tools which perform different aspects of migration. S1MT 1.2.3 support migration from NetDynamics and Kiva/NAS platforms. Each platform has its own Toolbox Builder which when executed will create a set of tools used to migrate a application. *Kiva Migration Toolbox Builder* creates tools for Kiva/NAS application migration and similarly *NetDynamics Migration Toolbox Builder* is used for migrating NetDynamics applications. The following figure shows you how to invoke a toolbox builder.



Toolbox Builder

You will use the same basic set of tools for each migration you perform, but each tool will need to be customized to that particular migration. Creating each of these tools can be a tedious task and prone to inconsistencies in naming conventions and layout of directory structures. Therefore, we've created a toolbox *add-in* (a pre-configured, ready-to-run tool) to simplify the process of creating these tools and setting their properties appropriately. Many of the tools have similar or even the same properties where consistency is important to the success of your migration.

Kiva Migration Toolbox Builder

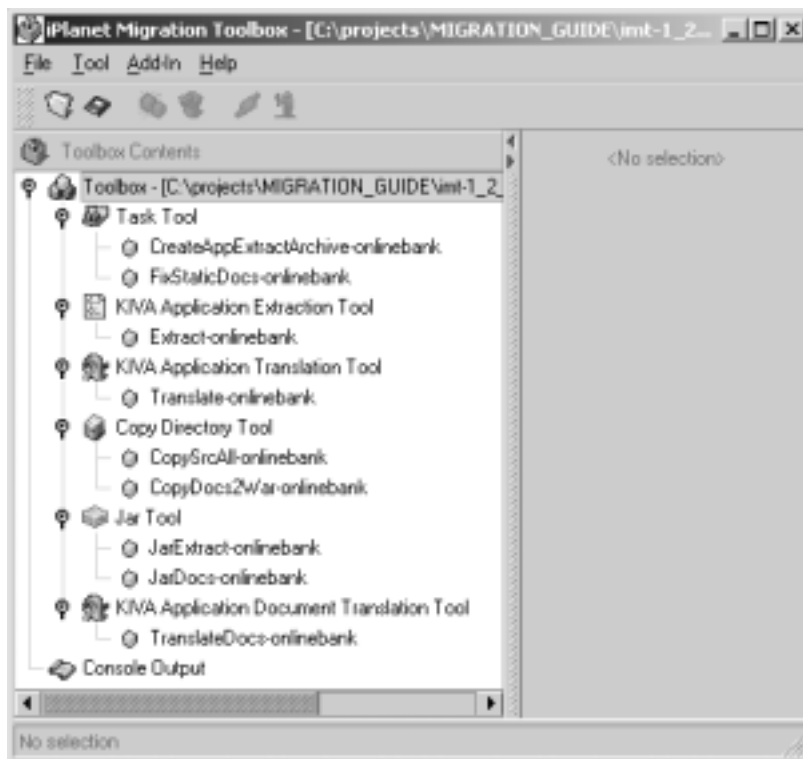
The following are the steps for creating a new toolbox using the Kiva Migration Toolbox Builder add-in:

1. If you don't have the Toolbox application currently running, start the Toolbox and select the menu option `Add-In|Migration|Kiva Migration Toolbox Builder`.

2. After a few moments, you will be prompted by a series of dialog boxes that will request some information. This information will be used by the Toolbox Builder to fill property values for the tools it generates. Some of the properties that you are not prompted for will contain defaults that may or may not need to be modified after the tools are created by the Toolbox Builder.
3. First, you are prompted for the package which the new JATO Application will be placed in. The best way to understand what this means is to run the OnlineBankSample migration and learn how a new package is created under `./war/WEB-INF/classes` to contain the JATO material. Although all the existing Java code is left in the same package, there is a need to create some new Java code for the JATO Application infrastructure. The new package is for this code. Please note that ALL Java source from the original application may remain in the same package. It is only the new Java source for the JATO resources which need a new package defined. No matter what package you choose (e.g. `com.ipplanet.migration.samples.onlinebank`), **the last name in the package** will be used as the default directory name for the migration results. You can override this directory location in the next panel; we recommend taking all the default values.
4. Next, you are provided the choice of using the Automatic Application Extract Archive Wizard. This wizard will help create tools for creating the application extract archive. If you choose Cancel then you are simply asked for the `application extract archive (ZIP/JAR)` path name. This is the name of the zip or JAR file which contains all the source for the application. In this case the archive must have been created manually beforehand and the wizard continues with encoding specifications.
5. If you choose OK for the Automatic Application Extract Archive Wizard then you are asked to enter the root directory to the application source (this is normally the `./nas/APPS` directory).
6. Next, you are asked to provide a list of top level packages in the application source directory pertinent to this migration. If all the source in the directory is included then you can skip specifying a value.
7. Next, you are asked to provide a list of file extensions which will be included in the Application Extract Archive.
8. If you choose OK for the Automatic Application Extract Archive Wizard then you will see a Task tool and Copy Directory tool and Java tool added to the toolbox.

9. There are two (2) panels which ask for the character encodings for Java source and query files. There are many customers who have Java source in an alternate character encoding (not ASCII). For instance, it is common for Asian developers/customers to use double-byte character source files. In a change from the S1MT BETA, only one (1) encoding value is allowed for file type. It is assumed that there is a common encoding standard within an application. If there are varying encodings then the application descriptor XML file may be edited accordingly after Extraction. Please note that S1MT 1.2.3 attempts to automatically discover character encoding of HTML templates by inspecting the `<meta>` tags in the source files. However, the migrator should carefully review the application descriptor XML file for encoding dispositions to ensure proper translation.
10. At this point the Kiva Extraction and Translation Tools are added to the toolbox.
11. Lastly, you are provided the choice of using the Automatic Static Document Translation Wizard. This wizard will help create tools for assembling the static document content and translating appropriate documents fixing the URLs for AppLogic invocation and copying the documents to the result WAR directory structure.
12. If you choose CANCEL then the builder exits. If you choose OK, you are asked to enter the location of the document root for the application and another Task tool, JAR tool, Document Translation tool and Copy Directory tool are added to the toolbox.
13. Save the toolbox to disk by selecting the menu option `File | Save` and give it a name.

Tools generated by Kiva Toolbox Builder are shown here:



Invoking the Tools

You are now ready to migrate your application by invoking the generated tools; extraction first and then translation. Before invoking each tool, inspect its properties first and make adjustments as needed. In general, if you've provided desirable initial values to the Toolbox Builder, none of the properties will need to be adjusted. (note: The Toolbox Builder created one Task Tool in your toolbox which you can use to invoke all of the other generated tools at once. However, we recommend invoking each tool individually until you have migrated one or two applications and become familiar with each tool's output.)

Tools Created by Kiva Migration Toolbox Builder

1. KIVA Application Extraction Tool

This tool reads a zip or JAR file called the application extract archive containing Kiva application files and creates an XML document called an application description file. The application description file contains high level information describing the application including disposition of each file found in the application extract archive.

This tool assumes, as input, the pre-existence of a zip file containing all of the original NAS/KIVA application source (i.e. templates, applogic java files, other application specific resources). The zip file need not contain the actual original class files since the migration effort will be altering the source files.

Creation of the application description file is the first step in the automated migration process. Although this file may be useful for other purposes, its main use is as input to the application translation process using the Kiva Application Translation Tool (com.ipplanet.moko.nas.tools.KivaTranslateTool).

2. KIVA Application Translation Tool

This tool reads both a zip or JAR file called the application extract archive containing Kiva application files and also an XML document called an application description file. This tool takes as input an application description file and uses it to generate a set of equivalent J2EE components and files. The application description file (an XML document) is produced as the result of using the Kiva Extraction Tool (com.ipplanet.moko.nas.tools.KivaExtractTool) to extract information from a set of source Kiva projects. Use of the translation tool is the second step in performing the automated migration of a Kiva application.

3. Copy Directory Tool

Copies the contents of a source directory to a target directory

4. JAR Tool

JARs all files in the source directory and all subdirectories

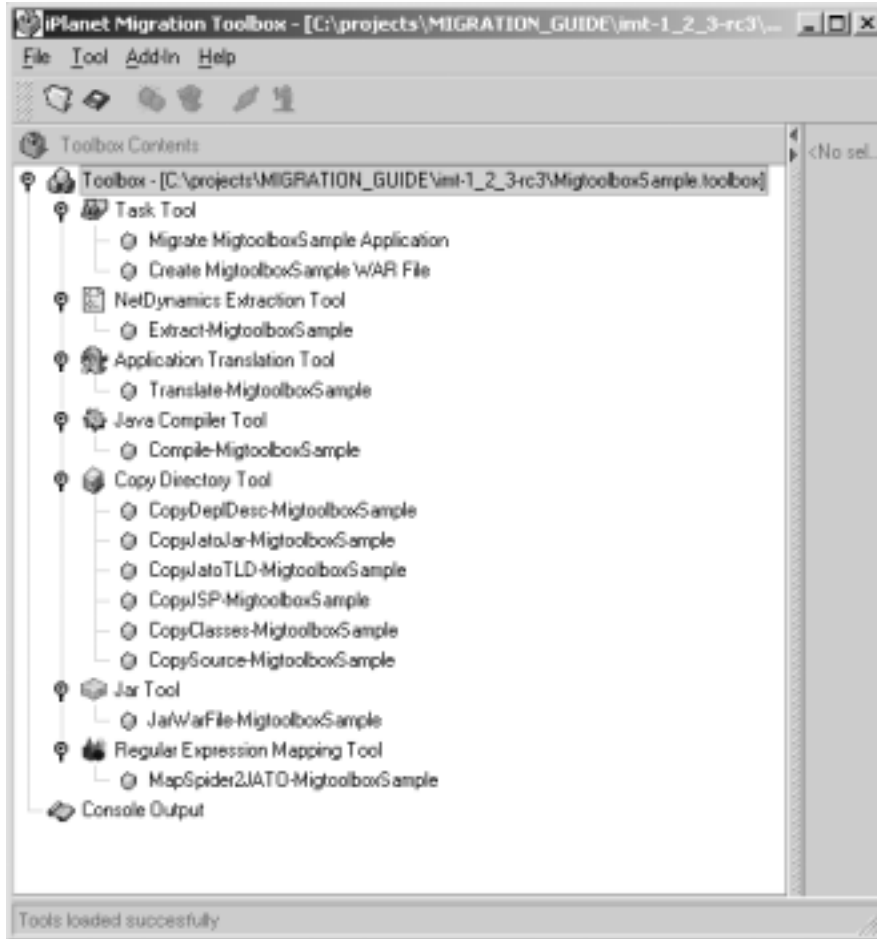
NetDynamics Migration Toolbox Builder

The following are the steps for creating a new toolbox using the NetDynamics Migration Toolbox Builder add-in:

1. If you don't have the Toolbox application currently running, start the Toolbox and choose the "Migrate an application" option in the Welcome dialog and press OK. With the Toolbox running, be sure that you have an empty (New) toolbox. Select the menu option `Add-In|Migration|NetDynamics Migration Toolbox Builder`.

2. After a few moments, you will be prompted by a series of dialog boxes that will request some information. This information will be used by the Toolbox Builder to fill property values for the tools it generates. Some of the properties that you are not prompted for will contain defaults that may or may not need to be modified after the tools are created by the Toolbox Builder.
3. **First prompt:** Enter the logical application name. This is the name of the entire application, which may include more than one NetDynamics project. If the application is only one project, then it is not recommended to use the project name as the application name. For example, if your project is called `foo`, then call your application `fooapp` rather than just plain `foo`. This will prevent confusion with other similar properties and avoid difficulties later during deployment.
4. After you've entered an application name, the Toolbox Builder will prompt you for more information, providing default values when possible. We recommend taking all the default values.
5. Once you have finished entering information, The Toolbox Builder will create several tools in your current toolbox. Save the toolbox to disk by selecting the menu option `File|Save` and give it a name. Using the *application name* (the value from the first prompt to `fooapp` in our example here) as the name of the toolbox is recommended.

Tools generated by NetDynamics Toolbox Builder are shown here:



Invoking the Tools

You are now ready to migrate your NetDynamics application by invoking several of the generated tools. Before invoking each tool, inspect its properties first and make adjustments as needed. In general, if you've provided desirable initial values to the Toolbox Builder, none of the properties will need to be adjusted. (**NOTE:** The Toolbox Builder created one or more Task Tools in your toolbox which you can use to invoke several of the other generated tools at once. However, we recommend invoking each tool individually until you have migrated one or two applications and become familiar with each tool's output.)

Tools Created by Kiva Migration Toolbox Builder

1. **NetDynamics Extraction Tool**

This tool gathers as much information as possible from the source NetDynamics project or projects and then writes this information to an XML file called the *application description* file. This application description will serve as the input to the *Application Translation Tool*.

Before invoking this tool, check the following properties for accuracy:

`ProjectsDirectory` is the path to the NetDynamics projects directory used during extraction. The default value is `%MIGTBX_HOME%/work/NDProjects`. We recommend placing all the NetDynamics projects you intend to migrate in this directory.

All other properties should be fine with their current values unless you made an error during the prompting stages of the Toolbox Builder add-in. The other properties will be discussed in detail later in this document.

Save the toolbox if you made any changes and invoke the tool. The XML output file (the application description) will be written to the location specified by the `OutputDirectory` property.

You may open and browse the application description file if you wish to understand the details of the project extraction. Using an XML browser like XML Spy is recommended. We highly discourage editing this file as mistakes introduced here may significantly affect the translation phase, causing it to fail completely or generate faulty output

2. **Application Translation Tool**

This tool uses the application description file generated by the *NetDynamics Extraction Tool* to output a set of J2EE-compliant components that accurately reflect the structure of the behavior of the original NetDynamics application.

All other properties should be fine with the current values unless you made an error during the prompting stages of the Toolbox Builder add-in. The other properties will be discussed in detail later in this document.

Save the toolbox if you made any changes and invoke the tool. The new J2EE components will be written to the location specified by the `OutputDirectory` property.

Additionally, this tool places a migration log file (`MigrationLog.csv`) in the translation output directory. This file indicates various items that were identified during translation as requiring additional or special migration attention. Our reason for generating this file is to alert migration developers to

those items that were not automatically handled by the translation, and to record information that was otherwise not carried forward during translation. This file generally serves as a minimal task list for the manual portion of the migration (there will likely be other tasks as well not related directly to the translation).

3. Regular Expression Mapping Tool

This tool, also known as the *Regex Tool*, uses a set of XML-specified match and replace specifications to effect changes within files (note, this tool uses the Perl 5 regular expression syntax). The Toolbox Builder generates a Regex Tool that is preconfigured to replace common Spider API Java constructs with equivalent JATO constructs in your migrated Java source files.

Before invoking this tool, check the following properties for accuracy:

`sourceDirectory` is the location of your migrated application code. Please make sure that this directory does not also contain the JATO source files, as the processing of those files may cause unexpected problems.

All other properties should be fine with the current values unless you made an error during the prompting stages of the Toolbox Builder add-in.

Save the toolbox if you made any changes and invoke the tool. The migrated source files will be processed and any changes that occur will be written to the console. Before any file is modified, the tool will backup the original file in its original location with a `.orig` file extension.

IMPORTANT: At this point, you have completed the automated migration phase, and must now port the Java code in the migrated application to use the J2EE/JATO API instead of the NetDynamics Spider API. The remaining tools described below will be useful for packaging and deploying your application once manual migration has been completed, with one exception: the migrated application should compile successfully at this point and minimally run if deployed (pages can be invoked); however, the application may not be functional if you've used any of the NetDynamics Spider API. Therefore, unless you want to simply make a sanity check or check the migration of non-Spider dependent features, we recommend porting at least part of the migrated application before continuing.

4. Java Compiler Tool

This tool is a convenient way to compile the JATO Foundation Classes and the new J2EE application components with one click. This tool simply invokes the `javac` command line tool provided with the JDK.

There should be no properties that need adjusting in this tool unless changes were made to the output directory properties of the previous tools. All of the properties will be discussed in detail later in this document.

Save the toolbox if you made any changes and invoke the tool. All of the java class source files (.java) under the directory specified by the `SourceDirectory` will be compiled.

5. Copy Directory Tools (Create WAR File Directory Structure)

This tool copies directories/files from one location to another with a file filter capability. The goal of the generated tools of this type is to create a "WAR file ready" directory structure. Running the first four *Copy Directory Tools* will copy the deployment descriptor, tag lib definition, JSPs, and Java classes into the appropriate directories so that the *Jar Tool* can be used to create a WAR file to be deployed in your J2EE container.

The instance of the *Copy Directory Tool* labeled `CopySource` is optional. The source files are not needed in your production WAR file, but you may find it helpful to keep a copy of the source files with your deployed application to ensure proper version control (these may also come in handy if a quick fix is necessary at the deployment site). These source files will **not** be visible to any application clients, and will therefore remain safe on your deployment server.

All of the properties should be fine with the current values unless you made changes to the output directory properties of the previous tools. All of the properties will be discussed in detail later in this document.

Save the toolbox if you made any changes and invoke the first four copy directory tools (`CopyDeplDesc`, `CopyTLD`, `CopyJSP`, `CopyClasses`). Invoke the fifth copy directory tool (`CopySource`) if this makes sense for your environment. Once these tools have been invoked, the appropriately filtered files will be written to the directory specified by each of the tools' respective `outputDirectory` property. The application is now ready to be "WAR'ed".

6. Jar Tool

This tool uses the JAR command line tool from the JDK to create a WAR file using the directory structure created by the previous copy directory tools. This WAR file will be the only file you will need to deploy your application to the J2EE container. (The iAS deployment procedure is discussed in the JATO Deployment Guide). Each container generally has its own deployment procedure; please follow the instructions for your container.

All of the properties should be fine with the current values unless you made changes to the output directory properties of the previous tools. All of the properties will be discussed in detail later in this document.

Save the toolbox if you made any changes and invoke the tool. The WAR file will be created and written to the location specified by the `OutputDirectory` property. The application is ready to be deployed.

Tools and Toolboxes

Toolboxes are persisted to disk in the format of a toolbox file (`.toolbox`). Individual tools of the toolbox are contained in the toolbox file in a serialized object format. These individual tools can exist outside of the toolbox file as a tools file (`.tools`) in a similar format. There are several menu commands that allow you to create, copy, delete, and merge two toolboxes together, as well as import and export individual or groups of tools.

Creating New Tools

To create a new instance of any tool, use the `Tool|New` menu option and select the type of tool you would like to create (Extraction, Translation, Compilation, etc.). You will notice the new tool will be added to currently opened toolbox in the toolbox tree. It will be grouped with other tools of its type and will have a default name of the form `<ToolType><##>`, like `CopyDirectoryTool7`. You can triple-click the tool name or press `F2` to rename it as you wish. Spaces are allowed in tool names.

Cloning Tools

To create a copy of a current tool, use the `Tool|Clone` menu option and a new tool of the same type will be created with the same properties as the original. Rename and adjust properties as needed.

Deleting Tools

To delete a tool, use the `Tool|Delete` menu option and the tool will be removed from the toolbox. You will be prompted verify your delete tool command, but there is no undo action. You may select several tools to delete at once by holding down the `Ctrl` or `Shift` keys while selecting additional tools.

Importing & Exporting Tools

You may have many different toolboxes (.toolbox files) that are focused on different NetDynamics application migrations. With the import and export commands, you can export a tool to a .tools file and then import it into another toolbox (.toolbox file).

To export a tool, open the toolbox with the tool you wish to export, select the tool or tools in the toolbox tree, then use the `File|Export` menu option and name the .tools file to export the tool. The tool will not be removed from the current toolbox.

To import the tool into another toolbox, open the toolbox you wish to be imported, then use the `File|Import` menu option, browse to the location of the .tools file you wish to import, then save the toolbox.

Toolbox Merging

If you have two separate toolboxes and would like to merge them into single toolbox you use the merge toolbox feature of the `Open Toolbox` menu option. To merge two toolboxes into one toolbox, open one of the toolboxes, and while it is open, open the other toolbox. You will be prompted to replace the existing toolbox, merge the new toolbox with the already-open toolbox, or cancel the operation.

Troubleshooting

IMPORTANT: Before continuing, make sure you have the latest S1MT patches available from the Sun ONE Migration Website. We will be releasing patches regularly as we discover and diagnose difficulties. We will release most of these patches to address problems found by users of the S1MT. Please submit any problems you encounter to the S1MT team so that we can diagnose the problem and issue a patch if necessary.

Toolbox Installation & Configuration

If you have difficulty running the Toolbox application, consult the following:

- Ensure that all the `%MIGTBX_HOME%/bin/setenv.bat` script is customized for your environment. Because of limitations of the JDK, you may not install the S1MT in a path containing directory names with spaces. For example, do not unpack the archive in your `C:\Program Files` directory. We recommend unpacking the archive either in `c:\iPlanet` or `c:\.`

- There are known problems using older versions of WinZip to unpack archives created with the JDK's zip/jar tools. Doing so will cause files to be truncated during unpacking, resulting in file lengths of zero bytes. Therefore, please ensure that you are using the latest version of WinZip when unpacking the S1MT archive (<http://www.winzip.com>).
- To avoid class version issues, we strongly recommend that you remove all JAR files from your JDK's extension directory (`%JAVA_HOME%/jre/lib/ext`) while running the Toolbox application. We have included all the classes necessary for running the Toolbox with the distribution. Please note that simply renaming the JAR files in the extension directory is not sufficient; you must move them to a different location.
- Because many development machines have several installed copies and/or versions of the JDK, be sure you know which copy of the JDK you are using. Set the `JAVA_HOME` environment variable in the `%MIGTBX_HOME%/bin/setenv.bat` file to ensure you are running the preferred copy with the Toolbox application.

Extraction

For the most part, as we've mentioned above, extraction of an application description is the most likely step in which you will encounter errors or difficulties. Also as we've already mentioned, this is frequently a **normal part of the migration process** and shouldn't discourage you if you are following the steps in previous sections. If you are having difficulties not covered above, consult the following tips.

General Issues

- During extraction, ensure that all external classes (non-NetDynamics project classes) are present on the Toolbox's classpath. The easiest way to make these classes available is to place JAR files or unpackaged classes in the `%MIGTBX_HOME%/lib/ext` directory. Classes and JARs in this directory will automatically be added to the Toolbox classpath upon startup. If this solution is unsatisfactory, you may either add the classes to your classpath or edit the `%MIGTBX_HOME%/bin/setclasspath.bat` file.
- Note the summary at the end of the output from the extraction and translation tools to determine if any project objects failed the automated process.
- Because of a limitation inherent in using the embedded NetDynamics runtime, exceptions thrown during extraction may not impact the reported tool status, and therefore the tool may report success when in fact the extraction failed. Therefore, we caution users to note and investigate all exceptions thrown during extraction. In some cases, we have seen seemingly innocuous

exceptions cause side effects which significantly impacted the fidelity of extracted project information. For example, during one extraction, we encountered a `ClassNotFoundException` from the NetDynamics runtime looking for a (seemingly) non-critical class. This exception later prevented certain `DataObject` properties from being extracted, resulting in a non-functional migrated application. Therefore, to ensure the best possible migration, always be sure to eliminate all sources of exceptions during the extraction phase before continuing.

- Note that because of a feature of the embedded NetDynamics CP, two copies of a project are instantiated during project extraction, one before extraction and one after. This is generally harmless, but if the project throws exceptions during instantiation, you will see two sets of stack traces in the Toolbox's console log.

Non-Fatal Error During Extraction

If only part of the automated migration succeeds (or fails), we recommend the following:

- Find and correct the cause of the failure using the tips in the above Sections and re-run the extraction or translation
- If a problem occurs with NetDynamics migration, create a new project in the NetDynamics Studio and import the problematic objects. Simplify them until you can get this project to run through the appropriate tool(s). Introduce these files back into the original, now-migrated project.
- Migrate the failed objects by hand. This is not as hard as it may sound. The JATO framework was also designed for manual application authoring. Using the templates in the `application` package, follow the example of a migrated object of the same type. Documentation has been created to assist in creating new JATO objects manually. Check the "Files" location of the JATO eGroups forum.
- Diagnose the problem as thoroughly as possible and consult the discussion forums or the S1MT team.

Fatal Error During Extraction

Ensure the following items are not factors in the failure (in approximate order of likelihood):

1. Incorrect environment settings. Check the settings of your `%MIGTBX_HOME%/bin/setenv.bat` file and ensure they are appropriate for your machine.
2. Missing external classes

3. Incorrect tool property settings. Ensure that the *Extraction Tool* has valid property settings
4. Use of non-existent runtime feature in a critical location (such as a class initializer or initialization of non-Spider threads to perform background tasks)
5. Non-present `links` directory or corrupted class files
6. Use of incorrect JDK version or platform
7. Conflicting class file versions in boot classpath (such as those present in the JDK's extension directory)

If none of the above items are discernable factors in the problem, you may have encountered a bug in the S1MT. We reiterate that because of the latitude NetDynamics allowed during project development, Sun ONE cannot anticipate all possibilities and thus ensure a trouble-free migration for all customers. However, the S1MT is committed to making the migration process as painless as possible. Please report any problems to the S1MT team and/or the discussion forums so that we may address them and issue patches as necessary.

Translation

If you encounter an error during application translation, do the following first:

- Ensure that your application description file looks complete and is valid XML. Use a tool like XMLSpy or Internet Explorer to open the document and view it.
- Ensure that the Translation Tool settings are correct
- Verify your environment settings in the `%MIGTBX_HOME%/bin/setenv.bat` file and ensure they are appropriate for your machine
- Ensure that you have a complete Toolbox installation

If none of the above items are discernable factors in the problem, you may have encountered a bug in the S1MT. We reiterate that because of the latitude NetDynamics allowed during project development, Sun ONE cannot anticipate all possibilities and thus ensure a trouble-free migration for all customers. However, the S1MT is committed to making the migration process as painless as possible. Please report any problems to the S1MT team and/or the discussion forums so that we may address them and issue patches as necessary.

Post-Migration

Some problems may arise after migration or during testing. In general, such problems will need to be posted to the discussion forums or discussed with the S1MT team. However, before contacting others, note the following:

- The module URLs for each servlet and display URLs for each view bean are set to certain defaults during project translation. These defaults will likely be correct for your deployment environment, but may not be in some cases. Please consult the JATO Deployment Guide or the discussion forums for information on how to configure these URLs differently for deployment.
- There are inconsistencies in the way JDBC drivers treat certain column types. JATO contains a number of options that may need to be modified in order for your application to work against your specific database. If you are having difficulty running the migrated application against your target database, please consult the Sun ONE Migration website and discussion forums for information on specific database-related tweaks.

Migrating from EJB 1.1 to EJB 2.0

Although the EJB 1.1 specification will continue to be supported in Sun ONE Application Server 7, 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.

NOTE A bean destined to be remote in EJB 2.0 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 has 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 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 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 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 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 ONE Application Server 7, EJBs are systematically mapped to the JNDI sub-context "*ejb*". If we attribute the JNDI name "*Account*" to an EJB, then Sun ONE Application Server 7 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 6.0/6.5,

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 ONE Application Server 7.

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 ONE Application Server 7. 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 ONE Application Server 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 ONE Application Server 7. 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` and has an indicated value, if yes, 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`.

The next topic, "*Migrating the Bean Class*", performs to migration process.

Migrating the Bean Class

This section describes the steps required to migrate the bean class to Sun ONE Application Server.

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

7. All the protected variables declared in the `ejbPostCreate()` methods in Step 5 have to 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 have to set the protected variables to the beans database state. 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` at the appropriate place, so that the unmigrated bean still works on Sun ONE Application Server.

Migration of `ejb-jar.xml`

To migrate the file `ejb-jar.xml` to Sun ONE Application Server perform the following steps:

1. In the `ejb-jar.xml`, convert all `<cmp-fields>` to become 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. As 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 6.0 and 6.5 use standard SQL to specify the finder logic.

Information concerning the definition of this finder method is stored in the EJB'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>findOrderedAccountsForCustomerParams</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 ONE Application Server 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>
```

Index

A

- About Sun ONE Application Server 6.0/6.5 27
- About Sun ONE Application Server 7 9
- About This Guide 5
 - How This Guide is Organized 6
 - What you should know 5
- Administration Server 17
- Administration Tool 16
- Administration Tools 15
 - Sun ONE Application Server 6.0 15
 - Sun ONE Application Server 7 17
- application client JAR 22
- AppLogic 115
- Architecture 9, 10
 - Sun ONE Application Server 6.0/6.5 architecture 27
 - Sun ONE Application Server 7 Architecture 9
- asadmin 18, 42, 68, 113
- Automated Migration Phase 116, 144
- automated tools 25
- Automating Migration 6, 163

B

- BEA WebLogic Server v6.1 113
- BMP 43

C

- CMP 39, 43
- CORBA 118

D

- data sources 32
- Database Connectivity 19
 - Database Support in Sun ONE Application Server 6.0 19
 - Database Support in Sun ONE Application Server 7 20
- db_setup.sh 19
- DB2 19
- Deploy 112
- Deployment descriptors 22, 23
- Development Environments 13
 - Sun ONE Application Server 6.0/6.5 13
 - Sun ONE Application Server 7 14
- DriverManager 30

E

- EAR files 22

- EJB 39
- EJB 1.1 to EJB 2.0
 - Defining Entity Bean Relationships 202
 - EJB 2.0 Container-Managed Persistence (CMP) 201
 - EJB Query Language 199
 - Message-Driven Beans 202
 - Migrating CMP Entity EJBs
 - Custom Finder Methods 208
 - Migrating the Bean Class 205
 - Migration of ejb-jar.xml 208
 - Migrating EJB Client Applications 202
 - Declaring EJBs in the JNDI Context 202
 - Migration of ejb-jar.xml 208
- EJB Changes Specific to S1AS 7 39
- EJB JAR 22
- EJB Migration 39
- EJB QL 39
- ejbCreate 82
- enterprise application 110
- Enterprise Applications 44
 - Application root context and access URL 45
 - Migrating Proprietary Extensions 46
- Enterprise EJB Modules 43
- Enterprise JavaBeans 12
- Entity Beans 40
- Extraction Tool 147
- Extraction tool 128

F

- format
 - URLs, in manual 6
- Forte for Java (FFJ) 118

G

- GXR 119

H

- home interface 91

I

- iasdeploy 19
- iBank 29, 46
 - Migrating iBank using Sun ONE Studio for Java 4.0 69
 - Converting CMP Entity EJBs from 1.1 to 2.0 78
 - Creating a Web application module 72
 - Creating an EJB module 90
 - Creating an enterprise application 110
 - Deploying the application 112
- iBank Application specification
 - Application Components 174
 - Application navigation and logic 171
 - Database schema 166
 - Fitness of design choices with regard to potential migration issues 177
 - Tools used for the development of the application 166
- IBM WebSphere v4.0 113
- Informix 19
- Iona 118

J

- J2EE 12
- J2EE Application Components and Migration 21
- J2EE applications
 - components 21
- J2EE Component Standards 12
- J2EE JATO 132
- JATO 122, 137
- JavaServer Pages 12
- JDBC Code 30
 - Using JDBC 2.0 Data Sources 32
 - Configuring a Data Source 33
 - Looking Up the Data Source Via JNDI 36

- JDBC drivers 19
- jdbcsetup 19
- JNDI context 36
- JSP 1.2 specification 37
- JSP's and JSP Custom Tag Libraries 36

K

- KFC (Kiva Foundation Classes) 115
- Kiva 115
 - automated migration phase 116
 - extraction 116
 - translation 116
 - manual migration phase 116
- Kiva Migration Toolbox Builder 181
- Kiva/NAS 4.1
 - Migration Preparation 115
 - Before Running the Extraction Tool 119
 - Migration Process Overview 115
 - Preparing a Project for Automated Migration 118
 - Preparing the GXR file 119
 - Preparing your Working Environment 117
- KIVA/NAS 4.1 to Sun ONE AS 7 115

M

- Manual Migration of iBank Application 47
 - Assembling Application for Deployment 68
 - EJB Changes 49
 - Web application changes 48
- Manual Migration Phase 117, 144
- MDB 39
- Migrating From S1AS 6.x to S1AS 7 29
- Migration and Redeployment 22
 - What is Redeployment 24
 - What Needs to be Migrated 23
 - Why is Migration Necessary 23
- Migration Considerations and Strategies 27

N

- NAS 4.1 115
- NetDynamics 144
 - automated migration phase 144
 - extraction 144
 - translation 144
 - Create a Toolbox Builder 148
 - Extraction Tool 146, 147
 - manual migration phase 144
 - Migrating ToolBox Sample Application 148
 - Migration Preparation 144
 - igration Process Overview 144
 - Preparing a Project for Automated Migration 146
 - Preparing your Working Environment 145
 - Running the Migration Toolbox 148
- NetDynamics Migration Toolbox Builder 181
- NetDynamics to Sun ONE AS 7 143

O

- Obtaining a Data Source from the JNDI Context 38
 - onAfterInit 146
 - onBeforeInit 146
- OnlineBankSample 120
 - Create a Toolbox 120
 - Running the Migration Toolbox 120
- Oracle 19

P

- PointBase 19
- Project Manager 133

R

- Registry Editor 16
- remote interface 91

S

- S1MT 115, 116
- Servlets 12, 37
- Session Beans 39
- setenv.bat 145
- SQL Server 19
- Sun ONE Console 15
- Sun ONE Migration Tool 25
- Sun ONE Migration Tool for Application Servers 163
- Sun ONE Migration Toolbox 25, 115, 181
 - Migration 181
 - Kiva Migration Toolbox Builder 182
 - NetDynamics Migration Toolbox Builder 186
 - Toolbox Builder 182
 - Supported Platforms 181
 - Tools and Toolboxes 192
 - Cloning Tools 192
 - Creating New Tools 192
 - Deleting Tools 192
 - Importing & Exporting Tools 193
 - Toolbox Merging 193
 - Troubleshooting 193
 - Extraction 194
 - Post-Migration 196
 - Toolbox Installation & Configuration 193
 - Translation 196
- Sun ONE Studio 14, 69
- Sybase 19

T

- Task Tools 128
- toolbox 125
- Toolbox application 181
- Toolbox GUI 181
- Translation tool 129
- type 2 19
- Type 4 19

U

- URLs
 - format, in manual 6

V

- Visibroker for Java 118

W

- WAR 22, 118
- Web Applications 40
 - Migrating Web Application Modules 41
 - Particular setbacks when migrating servlets and JSPs 42
- Web module 110
- web.xml 73
- WEB-INF 72, 73
- Welcome File 78