WebSphere software

IBM

Designing and Coding Applications for Performance and Scalability in WebSphere Application Server

Application design

Performance considerations

Best practices

Byron Braswell
Marcelo Manhães
Kon Samartzis
Eddy Tjandra

Redbooks

ibm.com/redbooks

International Technical Support Organization

Designing and Coding Applications for Performance and Scalability in WebSphere Application Server

January 2008

SG24-7497-00

Note: Before using this information and the product it supports, read the information in "Notices" on page xiii.

First Edition (January 2008)

This edition applies to Version 6, Release 1 of WebSphere Application Server.

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

- CICS®
- ClearCase®
- ClearCase MultiSite®
- ClearQuest®
- DB2®
- developerWorks®
- IBM®
- IMS™
- Purify®
- Rational®
- Rational Rose®
- Rational Unified Process®
- Redbooks®
- Redbooks (logo) ®
- RequisitePro®
- RUP®
- System p™
- System z™
- Tivoli®
- WebSphere®
- Workplace™
- z/OS®

The following terms are trademarks of other companies:

SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

Enterprise JavaBeans, EJB, Java, Java Naming and Directory Interface, Javadoc, JavaBeans, JavaScript, JavaServer, JavaServer Pages, JDBC, JDK, JNI, JRE, JSP, JVM, J2EE, J2SE, Solaris, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Expression, Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

In many WebSphere® Application Server environments, support personnel often get more questions on application performance and tuning than WebSphere Application Server tuning.

While there is much documentation and guidance for installing, monitoring, and tuning the performance and scalability aspects of WebSphere Application Server, not a lot of guidance and recommendations for performance and scalability considerations are given when designing and coding applications that execute in the WebSphere Application Server environment.

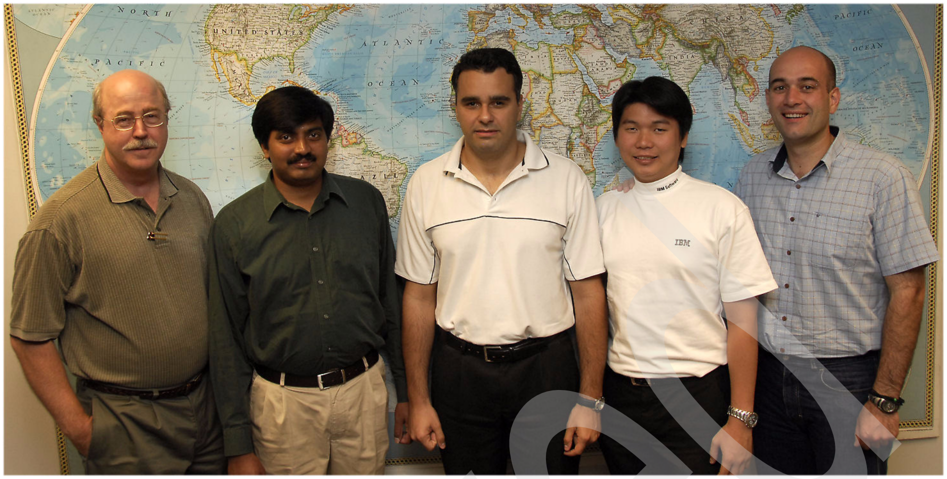
This IBM® Redbooks® publication provides performance and scalability considerations to keep in mind when developing and coding WebSphere Application Server applications. In this book, we take a layered approach to application development covering performance and coding considerations for each layer in a separate chapter.

In addition, various application development tools and strategies are compared within each layer along with best practices to keep in mind when designing and developing applications.

The target audience for this book includes the application development team, especially architects and developers. It also includes developers with experience using various application development techniques and tools for the different layers in the application architecture.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.



The team: Byron, James, Marcelo, Eddy, Kon

Byron Braswell is a Networking Professional at the ITSO, Raleigh Center. He received a Bachelor of Science degree in Physics and a Master of Computer Science degree in Computer Sciences from Texas A&M University. He writes extensively in the areas of networking, application integration middleware, and personal computer software. Before joining the ITSO, Byron worked in IBM Learning Services Development in networking education development.

Marcelo Manhães is a WebSphere IT Specialist working for IBM Global Services in Hortolândia-São Paulo-Brazil. He has ten years of experience in Web development and seven years in the WebSphere Software Platform field with support and development. He holds a degree in Bachelor in Computer from Paraná Federal University. He is certified for WebSphere Application Server 5 (Basic/Advanced Administration) and WebSphere Application Server 6 (Network Deployment Administration) and Sun™ certified for Java™ Programmer, Developer, Web Components, Business Components. His areas of expertise include EJB™ development and design, infrastructure development, and WebSphere Application Server support.

Kon Samartzis is a Senior IT Specialist in IBM Global Services, Australia. He has 13 years of experience in Web development and the systems integration field. He holds a degree of Bachelor in Computer Science from LaTrobe University in Melbourne. His areas of expertise include J2EE™ design and development, Enterprise Application Integration, Service Oriented Architecture, IBM WebSphere Application Server, WebSphere MQ, Web Services, and global technical team leadership.

Eddy Tjandra is an IT Specialist with IBM Software Lab Services in IBM Software Group Indonesia. He specializes in implementing IBM software that is based on J2EE technology such as WebSphere Application Server, WebSphere Portal Server, Workplace™ Software, and so on. He holds a Bachelor of Informatics Engineering degree from Sepuluh Nopember Institute of Technology (ITS), Surabaya, Indonesia.

Thanks to the following people for their contributions to this project:

James S. Pagadala is an IBM Global Business Services Associate Consultant, Package Solution Consultant. James participated in the planning and with the sample application setup during the project.

Margaret Ticknor
Carla Sadtler
Carolyn Briscoe
Linda Robinson
Robert Haimowitz
International Technical Support Organization, Raleigh Center

Richard M Conway
International Technical Support Organization, Poughkeepsie Center

Yvonne Lyon
International Technical Support Organization, San Jose Center

Thomas Alcott
Keys Botzum
Kyle Brown
Beena Hotchandani
Ruth Willenborg
IBM WebSphere Design/Performance

Richard Raszka
IBM Australia

Amrit Sanjeev
IBM India

Rudhi Wijanarko
Topik Hidayat
IBM Indonesia

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

1

Introduction

This chapter introduces the overall application development environment including the layered application design structure.

A review of design considerations is presented, followed by an introduction to various development and testing tools that are available.

The chapter is organized into the following major sections:

1.1 Introduction

This book discusses best practices and performance considerations for developing WebSphere Application Server based applications.

Before you start reading, you should be aware that the "best" in best practices is situational. There are certain situations where design decisions that are contrary to the "best" practices listed below are actually good. Just keep in mind that there are always several different factors that lead to a specific design decision, so being clear about the factors contributing to the decision is key.

We use the *Model View Controller* (*MVC*) paradigm as a basis for design guidelines, because this common architectural concept is used in nearly all J2EE-based applications running on an application server. See 2.2.2, "Model View Controller" on page 28 for more information on Model View controller.

Nevertheless, even with a good architecture, it is still possible to have a bad design. Therefore, we outline several different design alternatives that are based on this model. For each specific layer, we focus on existing design patterns, rather than "reinventing the wheel," but also cover new technologies that are available in IBM WebSphere Application Server V6.1. Throughout this book, we especially focus on performance-related aspects. In general, the best practices should help you in designing a scalable, high-performing application that can also be run on a highly available WebSphere cluster. In addition, we also cover some techniques and strategies that can assist you in optimizing the performance of your current application. However, they do not compensate for a poorly designed or architected application.

There can be a number of reasons for a poor application design, with limited scalability or several performance bottlenecks. These include a lack of focus on performance during requirement analysis and application design, a limited awareness of performance issues by the developers, or a desire to produce an elegant, extremely flexible, or highly object-oriented solution. It is important that performance targets be set at the beginning of the project and communicated to all team members involved in development. The implications of these targets on development work should also be analyzed early in the development cycle and made known. Note that there is often a trade-off between implementations that are elegant, flexible, and easy to read and maintain; and those that offer higher performance.

In general, it is a good idea to verify as soon as possible that the performance expectations are met. In general, a load-performance test is a good way to find possible architecture and design defects. So it might make sense to develop and then load-test a prototype in a very early stage of the project, just to verify your design decisions. Always keep in mind that design changes in a later phase of the software life cycle are very, very expensive! Therefore, ongoing performance testing activities should be undertaken throughout the whole project life cycle to ensure that the performance requirements are met.

To optimize this specific testing of performance under simulated loads, we recommend using a tool such as Rational® Performance Tester, OpenSTA, or Apache JMeter. The tool can also provide insight as to where performance optimization work should be targeted.

If your application is already in or near production and you notice performance problems, the following chapter also contains useful information, for example, about caching technologies or other performance optimizations whereby IBM WebSphere Application Server V6.1 can potentially improve your application performance without changing the overall application design. In these cases, profiling tools can also be extremely useful, such as those included with IBM Rational Application Developer and described in Chapter 15 of *WebSphere Application Server V6 Scalability and Performance Handbook*, SG24-6392.

These tools indicate which parts of the code are most frequently used, and also where the majority of the execution time is spent. Typically, most of the execution time is spent on a minimum of the code, as suggested by the 80/20 rule: 80% of the execution time is spent on 20% of the code. In many cases, the ratio might be even higher, such as 90/10 or more. Although it is important to be aware of best practices when performing all development work, extra care should be taken to optimize the most frequently used code sections.

More information about server-side tools for analyzing usage patterns of components of an application such as servlets and EJBs can be found in Chapter 14 of *WebSphere Application Server V6 Scalability and Performance Handbook*, SG24-6392.

1.2 Layered application design

Any application based on Java 2 Platform, Enterprise Edition is inherently a distributed application and thus it can be classified into a set of layers. During the implementation stage, we can group together related code under one layer. For example, all the code related to user interface (UI) elements for data input and for displaying information can be grouped together under the presentation layer.

A typical separation of layers in J2EE applications is:

- Presentation layer
- Controller layer
- Business facade layer
- Domain layer
- Data access layer

When the application is divided into layers, we can identify a horizontal separation of the functionality, as shown in Figure 1-1.

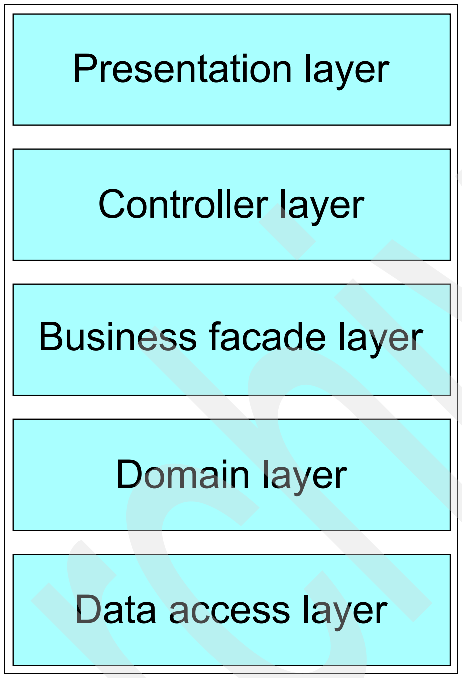


Figure 1-1 Horizontal separation of the application functionality

1.2.1 Presentation layer

The presentation layer is the user interface of a component. This layer includes any Web pages that use forms and other UI elements to allow the user to enter data, as well as any Web pages that use tables and other UI elements to display information. It is normally implemented using a combination of:

- HTML pages
- JavaServer™ Pages™(JSP™)
- JavaScript™
- Images and other multimedia files

1.2.2 Controller layer

The controller layer connects the presentation layer with the component business logic, which is implemented in the business facade layer. In effect, the controller layer accepts a request from the presentation layer, calls the appropriate method of the business facade layer, stores any results in the request object, and returns back to the presentation layer for displaying the results. It is normally implemented using a combination of:

- Java Servlets
- JavaBeans™
- Struts actions
- Struts forms

1.2.3 Business facade layer

The business facade layer plays the role of a bridge between the front-end and the back-end of the application. It is implemented using a manager class which exposes the business methods required from the implementation of the component. In effect, a method of the manager is called from one of the servlets or Struts actions of the controller layer with some parameters passed in the form of a Data Transfer Object (DTO). The manager creates an instance of the appropriate Data Access Object (DAO), executes it, and returns the results obtained from the database to the caller either as a single DTO or a collection of DTOs.

1.2.4 Domain layer

The domain layer consists of custom-designed JavaBeans that are used to encapsulate data transferred between layers of the application. These JavaBeans are called Data Transfer Objects (DTOs). To implement these DTOs, we used the functionality provided by WebSphere Studio and the Rational Software Development Platform to help us generate getter and setter methods.

1.2.5 Data access layer

The data access layer consists of custom designed JavaBeans that perform database operations using JDBC™. These JavaBeans are called Data Access Objects (DAOs). To implement these DAOs, we used the various data wizards provided by WebSphere Studio and the Rational Software Development Platform.

1.3 Naming conventions

Spending some extra time on application-related naming concepts quickly pays off in practice, because it can reduce the time spent on analyzing the source of issues during standard operations of future J2EE applications.

1.3.1 Naming for applications

Generally, some form of the version, release, modification, fix (VRMF) schema is used to organize code and builds, and commonly, a dotted number system such as 1.4.0.1 is used. In this way, code management systems can be certain of identifying, creating, and re-creating application builds accurately from the correct source code, and systems administrators and developers know exactly which version is used.

Append the version number to the enterprise archive (EAR) file name, such as in OrderApplication-1.4.0.1.ear.

Sometimes, the version number of included components, such as utility JAR files packaged in the EAR, can also have version numbers in their file names, but this can often cause problems. Consider a utility JAR with a version number in the file name, such as log4j-1.2.4.jar. If that is updated and the name is changed to log4j-1.2.5.jar, each developer has to update the class path settings in their workspace, which costs them time. It is then better to use an SCM system and label the new JAR file as being version 1.2.5, but keep the file name constant, such as just log4j.jar.

To keep track of all the versions of included components, it is a good idea to include a bill of materials file inside the EAR file itself. The file can be a simple text file in the root of the EAR file that includes versions of all included components, information about the tools used to build it, and the machine on which application was built. The bill of materials file can also include information about dependencies to other components or applications, as well as a list of fixes and modifications made to the release.

1.3.2 Naming for resources

When naming resources, preferably associate the resource to both the application using it and the physical resource to which it refers. As an example for our discussion, we use a data source, but the concept holds also for other types of resources such as messaging queue. Remember, if your company already has a naming convention for other environments (non-WebSphere) in place, it is probably a good idea to use the same naming convention in WebSphere.

Assume that you have a database called ORDER that holds orders placed by your customers. The obvious name of the data source would be Order and its JNDI name jdbc/Order.

If the ORDER database is used only by a single application, the application name can also be included to further explain the purpose of the resource. The data source would then be called Order_OrderApplication and its JNDI name jdbc/Order_OrderApplication.

Because the WebSphere administrative console sorts resources by name, you might want to include the name of the application first in the resource, such as in OrderApplication_Order. This gives you the possibility to sort your resources according to the application using them.

To group and sort resources in the WebSphere administrative console, you can also use the Category field, which is available for all resources in the administrative console. In this text field, you can enter, for example, a keyword and then sort your resource on the Category column. So instead of including the name of the application in the resource name, you enter the application name in the Category field instead.

If you have several different database vendors, you might also want to include the name of the database vendor for further explanation. The Category field is a good place to do that.

1.4 Source code management

In development, it is important to manage generations of code. Carefully organize and track application builds and the source code used to create them to avoid confusion. In addition to tracking the version of the source code, it is equally important to track the version of the build tools and which machine was used to generate a build. Not all problems are due to bugs in source code.

Developers produce code and usually use an integrated development environment (IDE) such as the Application Server Toolkit or Rational Application Developer to do that. Code in an IDE is stored in a workspace on the file system, usually locally on each developer’s machine. As the project continues, and perhaps new members join the team, the code grows and it becomes necessary to manage the code in a central master repository. This allows for:

- Development team collaboration (work on common code)
- Code versioning (managing which versions are in which releases)
- Wider team collaboration (access for project managers, testers)

Source code management (SCM) systems are used for these purposes.

Rational Web Developer and Rational Application Developer support Rational ClearCase® and CVS as SCM systems, while the Application Server Toolkit supports only CVS.

1.4.1 Rational ClearCase

Rational ClearCase organizes its code repositories as versioned object bases or VOBs. VOBs contain versioned file and directory elements. Users of Rational ClearCase are organized according to their role. Each user has their own view of the data that is in the VOB on which they are working. Rational ClearCase tracks VOBs and views and coordinates the checking in and checking out of VOB data to and from views.

As the role-based model suggests, Rational ClearCase is not just an SCM system but also a Software Asset Management (SAM) system. This means that it not only manages code but other assets. These further assets might be produced by the other Rational products with which Rational ClearCase integrates.

The Rational products with which ClearCase integrates are Rational Enterprise Suite Tools, the Rational Unified Process®, and, of course, Rational IDEs. Artifacts such as use cases generated by Rational RequisitePro® can be stored in Rational ClearCase. These can then be fed into a Rational Rose® design model and used to design Java components and generate Unified Modeling Language (UML) diagrams and documentation.

ClearCase can also be used to implement the Unified Change Management (UCM) process. This change management process can be enhanced by using Rational ClearCase in conjunction with Rational ClearQuest®, a change and defect tracking software.

The software is scalable. Rational ClearCase LT is a cut down version of Rational ClearCase for small-to medium-sized teams. It can be upgraded seamlessly to Rational ClearCase as a user's requirements change. Additionally, use a ClearCase MultiSite® add-on to support use of the software in geographically dispersed development teams.

In short, although ClearCase is an SCM system, it is also an integral part of the Rational toolset and RUP®.

For more information about Rational software, see:

http://www.ibm.com/software/rational

1.4.2 Concurrent Versions System

Concurrent Versions System (CVS) uses a branch model to support multiple courses of work that are somewhat isolated from each other but still highly interdependent. Branches are where a development team shares and integrates ongoing work. A branch can be thought of as a shared workspace that is updated by team members as they make changes to the project. This model enables individuals to work on a CVS team project, share their work with others as changes are made, and access the work of others as the project evolves. A special branch, referred to as HEAD, represents the main course of work in the repository (HEAD is often referred to as the trunk).

CVS has the following features:

- It is free to use under the GNU license.
- It is open source.
- It is widely used in the development community.
- Other SCM repositories can be converted to CVS.
- Many free client applications are available, for example, WinCVS.
- It can store text and binary files.
- It handles versioning and branching.
- It is a centralized repository.

For more information about Concurrent Versions System, see:

http://ximbiot.com/cvs/wiki

1.4.3 Which source code management to use

The obvious question arises: Which SCM should the team use? There is no simple answer to this question, because the answer depends on a number of factors.

Current software and processes

To some extent, the choice depends on what the existing situation is (if any) and what the SCM and development process requirements are now and in the future. If a team uses CVS and an existing, successful, development process, ClearCase might not be necessary, especially if the size and complexity of requirements is not likely to grow in the future. If this is not the case, Rational ClearCase LT or Rational ClearCase are a good choice so that the full integration of Rational and WebSphere products can be exploited now and in the future.

Team size

Rational ClearCase LT gives a sound starting place for smaller teams. Rational ClearCase LT can be upgraded to Rational ClearCase later if necessary. On very large development projects, Rational ClearCase and Rational ClearQuest have a MultiSite option that allows for easier development by geographically dispersed development teams.

Complexity of requirements

RUP provides a holistic approach to the end-to-end development life cycle. The use of the UCM process, which is part of the RUP, can shield the user from complex tagging and branching of code. CVS does not shield the user from this.

Cost

CVS is a possibly a cheaper option because it is free and has a large user base, which means cheaper skills. In terms of hardware, it is likely that hardware costs for hosting CVS itself are cheaper because of its smaller footprint. However, these might be false economies. The limitations of CVS can cause a team to migrate to Rational ClearCase later.

Change management process

If the development team uses CVS rather than Rational ClearCase, the team does not get a prescribed change management process for CVS such as the UCM. If their organization does not have its own change management process, such a process should be created and put into place.

Summary

In summary, the smaller the development team and the less complex the requirements, the more likely that CVS or Rational ClearCase LT are good choices. As team size and complexity grows, Rational ClearCase and then Rational ClearCase MultiSite become more attractive. Existing processes and software as well as the budget for new software, hardware, and training are likely to inform the decision further. In matters of cost, there might be false economies.

1.5 Automated build process

The major driver for implementing and maintaining an automated build process is to provide a simple and convenient method for developers to perform builds for development, test, and production environments.

These are some of the main problems you might run into when you do not have an automated process:

- Failures occur in your test or production environment because the code was not packaged correctly.
- The wrong code was deployed, causing the application to fail.
- The development team, testers, and even customers have to wait to get the code out to a test, staging, or production environment because the only person who has control over these is unavailable.
- You cannot reproduce a problem on production because you do not know what version of files are in production at the moment.

The time spent developing an automated build script pays for itself over time. After you have an automatic build process in place, you can virtually eliminate failures due to improper deployment and packaging, considerably reduce the turnaround time for a build, allow you to easily recreate what is in each of your environments, and ensure that the code base is under configuration management.

There are several tools on the market to help you develop a build script, including Apache Ant. Apache Ant is a Java-based build tool that extends Java classes and uses XML-based configuration files to perform its job. These files reference a target tree in which various tasks are run. Each task is run by an object that implements a particular Task interface. Ant has become a very popular tool in the Java world.

WebSphere Application Server provides a copy of the Ant tool and a set of Ant tasks that extend its capabilities to include product-specific functions. These Apache Ant tasks reside in the com.ibm.websphere.ant.tasks package. The Javadoc™ for this package contains detailed information about the Ant tasks and how to use them.

The tasks included with WebSphere Application Server enable you to:

- Install and uninstall applications.
- Run EJB deployment and JSP pre-compilation tools.
- Start and stop servers in a base configuration.
- Run administrative scripts or commands.

By combining these tasks with those provided by Ant, you can create build scripts that pull the code from the SCM repository, and then compile, package, and deploy the enterprise application on WebSphere Application Server. To run Ant and have it automatically see the WebSphere classes, use the **`ws_ant`** command.

For more detailed information about Ant, refer to the Apache organization Web site at:

http://ant.apache.org/index.html

1.6 Automated functional tests

Automating your functional tests might be a good idea, depending on your project size and how complex the requirements of the project are. Scripts execute much faster than people, but they are not automatically generated, so someone has to create them at least one time. It is possible to create a script to cover all function in your application, but it would be very complicated and costly. A good idea is to create scripts for the main features of the system and for those that do not change very much over time, so whenever a new build is published by an automated build tool or human personnel, you can be sure that the application still works properly.

IBM offers a rich set of software tools for implementing automated test solutions. These solutions solve many common problems and therefore reduce complexity and cost. For more information, see Rational Functional Tester at:

http://www.ibm.com/software/awdtools/tester/functional/

1.7 Test environments

Before moving an application into production, it is very important to test it thoroughly. Because there are many kinds of tests that have to be run by different teams, a proper test environment often consists of multiple test environments.

Tests cases must be developed according to system specification and use cases. Do this before the application is developed. These system specification and use cases must be detailed enough so that test cases can be developed. Test cases have to verify both functional requirements (such as application business logic and user interface) and non-functional requirements (such as performance or capacity requirements). After developing the test cases and enough functionality has been developed in the application, start testing.

Figure 1-2 shows an overview of a recommended test environment setup.

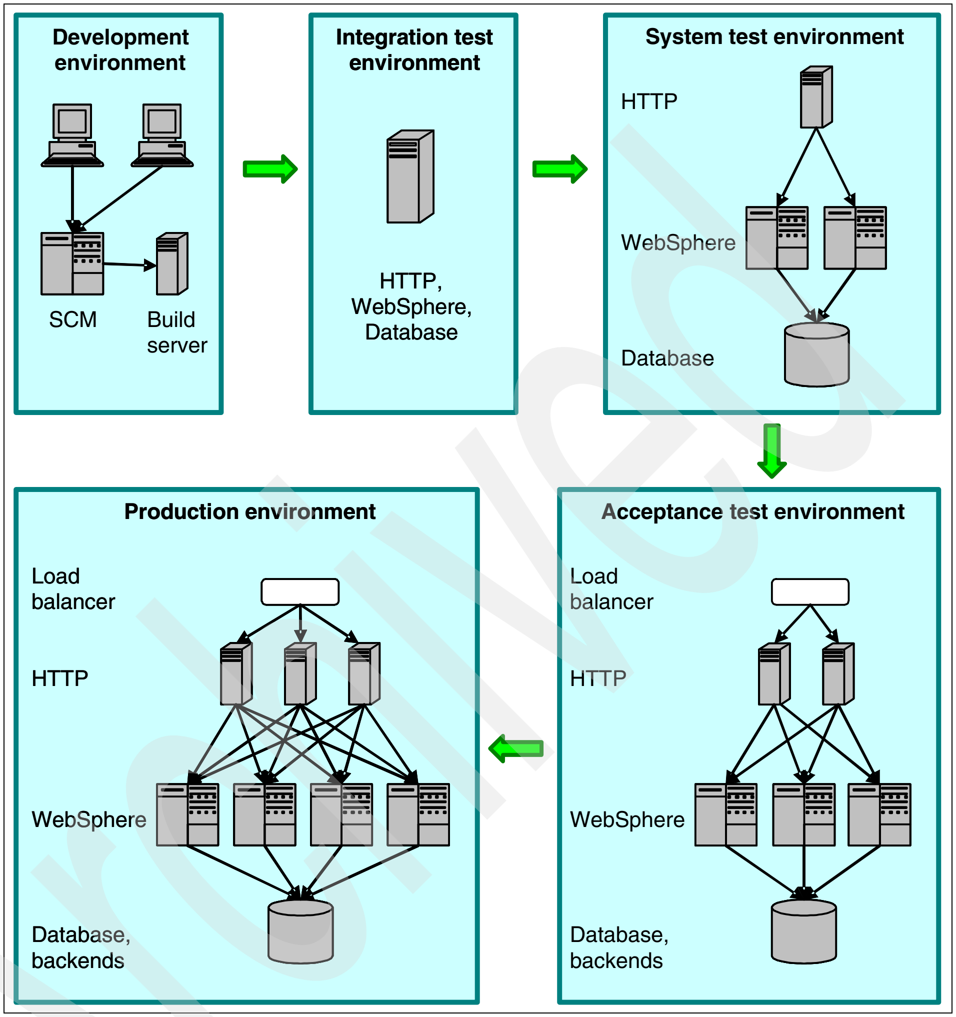


Figure 1-2 Test environments

Whether you choose to use some of these test environments, all of them, or even additional test environments, depends on the system being developed, project size, budget constraints, and so on.

Each environment is maintained as a separate cell in order to completely isolate the environments from each other. For smaller environments, a single application server profile is usually sufficient, while larger ones might require a deployment manager for that particular cell environment.

1.7.1 Development environment

Usually each developer has their own WebSphere test environment integrated in the development tool. This test environment is used for the developer's daily work and it is often active while the developer is coding. Whenever necessary, the developer can perform instant testing.

Because of the tight integration between WebSphere Application Server and the IBM development tools, the application server can run the application using the resources in the developer's workspace. This eliminates the necessity for developers to execute build scripts, export, or otherwise package the application into an EAR file and deploy that on a test server for every small change made. This capability makes it very easy and quick to test applications while developing them and increases developer productivity.

Each developer is also responsible for performing unit testing of their own code. Most tests performed for the system are executed in this environment, and the primary goal is to wash out obvious code bugs. The developers work against this environment and share code using the SCM system. The development environment is most often a powerful Windows® desktop machine.

When each developer has committed their code on to the integration stream in the SCM system, a development lead or integration team usually performs a clean build of the whole application, bringing together code developed by different developers. This is usually done on a special build server and is controlled by automatic build scripts (see 1.5, “Automated build process” on page 10). This server might require having a copy of the Application Server Toolkit or Rational Web Developer installed.

The development team should also create a Build Verification Test process (see 1.6, “Automated functional tests” on page 12), one where each new build is executed before making this build available to the team. A Build Verification Test covers test cases or scenarios that verify that critical paths through the code are operational. Build Verification Test scripts are often controlled by JUnit.

Another activity that is every developer's responsibility is to perform basic code profiling. By using the profiling tools in Rational Application Developer, a developer can discover methods that perform poorly, find memory leaks, or excessive creation of objects.

1.7.2 Integration test environment

After a successful build and regression test, the application is deployed to the integration test environment. This is the environment where the developers perform integration tests among all system components on a hardware and software platform that mirrors the production environment, although in a very small size.

Because the production environment is often not the same platform as the development environment, a guideline is to start testing on the target platform as early as possible in the test phase. This testing helps discover problems with incompatibilities between platforms, for example, hard coded folder paths (such as C:\ versus /usr). The integration test environment is usually the first environment suitable for that.

For small projects, the integration test environment can often be shared between different projects. But if the number of projects or developers is too large, it becomes difficult to manage. Usually no more that 5 to 10 developers should share a single integration test environment. If a developer has to perform tests that might damage the environment, a dedicated environment should be used.

As long as the machine has enough resources in terms of CPU and memory, using multiple WebSphere profiles can also be a good method to isolate different teams from each other. Using VMWare is another option.

The development team manages and controls the integration test environment.

1.7.3 System test environment

The purpose of the system test is to verify that the system meets both functional and non-function requirements. After the development team has tested the application in their own controlled environment, it is delivered to the system test team. When the application is delivered, the system test team deploys it using the instructions given.

If the tests in the previous test stages have been less formal, a key aspect of the system test is formality. The system test team is responsible for verifying all aspects of the system and ensuring that it conforms to the specifications. Functional requirements include things such as does the system execute the business rules defined, does the user interface show the right information, and so on. Non-functional requirements include capacity, performance, installation, backup, and failover requirements.

The system test team completely controls the system test environment. The environment is usually a cut-down version of the real production environment, but with all the important components in place. If the production environment is a highly available environment with WebSphere clusters, the system test should also be set up with clusters to verify both application functionality and deployment routines.

The system test environment can also be used by other teams. Perhaps the system administrators have to test new patch levels for the operating system, WebSphere, database, and so on before rolling them out in production. The system test environment is a good place to do that. If a patch is committed, it should also be applied to the other test environments to keep all environments in sync.

1.7.4 Acceptance test environment

The acceptance test environment is the last stage, where testing takes place before moving the application into production. The acceptance test environment is the one that most closely resembles the actual production environment. Hardware and software must be identical to the production environment.

Because of cost constraints, it is often not possible to have an acceptance test environment with identical capacity as the production environment. The acceptance test environment is, therefore, usually smaller than the production environment, but must contain all the same components, same brands, same software patch levels, and same configuration settings as the production environment.

The purpose of the acceptance test environment is to give the operations team a chance to familiarize themselves with the application and its procedures (such as installation, backup, failover, and so on). It also provides an opportunity to test unrelated applications together. The previous environments all focused on testing the applications independently of each other.

Often the acceptance test environment is where performance tests are run, because the acceptance test environment is the one most similar to the real production environment.

When doing performance tests, it is extremely important to have a representative configuration as well as representative test data. It is not unusual that projects perform successful performance tests where the results meet the given requirements, and then when the application is moved into production, the performance is bad. Often this can be because the production database is much larger than the databases used in the acceptance test environment.

Therefore, it is very important that the test databases have been populated with representative data. Ultimately, a copy of the production database should be used, but sometimes this is not possible because tests might involve placing orders or sending confirmation e-mails. Other causes for differences in performance between the successful performance tests and the production environment is, for example, that the performance tests ran without HTTP session persistence, while the production environment uses session persistence.

To get realistic results, the performance test environment and setup must be realistic, too.

1.8 New in WebSphere Application Server V6.1

The following list highlights the features added since WebSphere Application Server V6.1:

- Application Server Toolkit enhancements:

  The Application Server Toolkit has shipped with WebSphere Application Server since Version 5.1, but with Version 6.1, it has been significantly improved and is now a full-blown integrated development environment (IDE). It can be used to build, test, and deploy J2EE applications on a WebSphere Application Server V6.1 environment (but not on any previous release). It has support for all J2EE artifacts supported by WebSphere Application Server V6.1, such as servlets, JSPs, EJBs, XML, and Web Services, and also supports developing Java 5.0 applications.

- Portlet application support:

  The portlet container in WebSphere Application Server V6.1 provides the runtime environment for JSR 168 compliant portlets. Portlet applications are intended to be combined with other portlets to collectively create a single page of output. The portlet container takes the output of one or more portlets and generates a complete page that can be displayed.

  The primary development tool for portlets on WebSphere Application Server portlet applications is the Application Server Toolkit. You can also use Rational Application Developer, but you should review the following item in the WebSphere Information Center:

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.nd.doc/info/ae/ae/cport_portlets.html`

  Portlets are packaged in WAR files.

  Note that the portlet runtime does not provide the advanced capabilities of WebSphere Portal, such as portlet aggregation and page layout, personalization and member services, or collaboration features.

  For more information about JSR 168, see:

  `http://jcp.org/en/jsr/detail?id=168`

- Session Initiation Protocol (SIP) support:

  SIP applications are Java programs that use at least one Session Initiation Protocol (SIP) servlet written to the JSR 116 specification. SIP is used to establish, modify, and terminate multimedia IP sessions. SIP negotiates the medium, the transport, and the encoding for the call. After the SIP call has been established, the communication takes place over the specified transport mechanism, independent of SIP. Examples of application types that use SIP include voice over IP, click-to-call, and instant messaging.

  The Application Server Toolkit provides special tools for developing SIP applications. SIP applications are packaged as SIP archive (SAR) files and are deployed to the application server using the standard WebSphere Application Server administrative tools. SAR files can also be bundled within a J2EE application archive (EAR file), just like other J2EE components.

  For more information, see:

  – JSR 116 SIP Servlet API 1.0 Specification:

    http://www.jcp.org/aboutJava/communityprocess/final/jsr116/

  – RFC 3261:

    http://www.ietf.org/rfc/rfc3261.txt

- WebSphere Application Server V6.1 Feature Pack for Web Services extends the capabilities of Application Server V6.1 to enable Web Services messages to be sent asynchronously, reliably, and securely, focusing on interoperability with other vendors.

  For more information, see:

  http://www-1.ibm.com/support/docview.wss?rs=180&uid=swg21264563

1.9 Development and deployment tools

The WebSphere Application Server V6.1 environment comes with a rich set of development tools. All editions of WebSphere Application Server V6.1 include the Application Server Toolkit V6.1, which has been much improved since previous WebSphere releases and is now a full-blown J2EE development tool.

The Application Server Toolkit is targeted to support only the version of the WebSphere Application Server with which it ships. This means that Application Server Toolkit V6.1 supports all new features of WebSphere Application Server V6.1 and supports it as an integrated test environment. It does not, however, support any of the previous versions of WebSphere Application Server as integrated test environments.

1.9.1 Application Server Toolkit V6.1

Application Server Toolkit was first shipped with WebSphere Application Server V5.1 and was originally only for the assembly, deployment, and debugging of J2EE applications on WebSphere. In WebSphere Application Server V6.1, it has been significantly enhanced and is now a full-blown development tool that can be used also for developing J2EE applications.

Application Server Toolkit V6.1 is based on the Eclipse 3.1.2 platform and inherits much of its functionality from the Eclipse Web Tools Platform, which is a relatively new Eclipse project to which IBM has been a major contributor. The Web Tools Platform is what provides the Web and J2EE concepts to Eclipse and, thus, the Application Server Toolkit.

The Application Server Toolkit V6.1 provides the following features:

- Java 5.0 support
- Development of standard J2EE artifacts, such as servlets, JSPs, and EJBs complying with J2EE 1.2, 1.,3 and 1.4 specifications
- Web Services tools, including wizards to generate Web Services from Java beans, EJBs, and WSDL files and to consume Web Services; also includes UDDI test registry integration
- Development of static Web projects (HTML, CSS style sheets, JavaScript)
- SIP development, including support for JSR 116 SIP servlets
- Portlet development (JSR 168)
- XML tools to build and validate XML artifacts, including schemas, DTDs, and XML files
- Data tools for connecting to and interacting with various database vendors
- WebSphere Enhanced EAR support
- Support for annotation-based development (part of WebSphere rapid deployment)
- Support for WebSphere Application Server V6.1 test environments in either a local or remote configuration, but no support for any previous versions of WebSphere Application Server (such as 6.0 or 5.1)
- Jython script development, including script debugging capabilities
- Jacl to Jython script conversion tools (jacl2jython)
- Integration with Concurrent Versions System (CVS), which is a popular Source Code Management (SCM) repository (no integration with Rational ClearCase is provided)

To summarize, Application Server Toolkit V6.1 is a full-blown development environment that provides you with the tooling necessary to create, test, and deploy the various artifacts supported by WebSphere Application Server V6.1.

It does not, however, include the productivity-enhancing features and visual editors found in Rational Application Developer. It also does not include Rational ClearCase, Crystal Reports, UML modeling, Struts, or JSF support, and it does not support any of the previous releases of WebSphere Application Server (such as 5.1 or 6.0) as test environments.

1.9.2 Rational Application Developer V7.0

IBM Rational Application Developer for WebSphere Software V7.0 is an integrated development environment and platform for building Java Platform Standard Edition (Java SE) and Java Platform Enterprise Edition (Java EE) applications with a focus on applications to be deployed to IBM WebSphere Application Server and IBM WebSphere Portal.

There are many new features in Version 7. The objective of this section is to summarize the new features in Rational Application Developer V7.0:

- Specification versions: Full support is provided for Java EE V1.4, Java SE V5.0 and IBM WebSphere Application Server V6.1.
- Eclipse and IBM Rational Software Delivery Platform: Based on Eclipse 3.2.
- Application Developer V7.0 supports Java 5. There is tooling for such features as annotations, generics, enums, static import, and variable arguments.
- Web tooling:
  - The Web Diagram Editor is rewritten to leverage the Graphical Modeling Framework (GMF).
  - Drag and drop functionality (from the Palette) in the Web Diagram Editor updates the diagram and (behind the scenes) generates appropriate code (keeping diagram and code in-sync).
- JavaServer Faces (JSF):
  - Full support is provided for JSF 1.1.
  - There is a new version of the IBM JSF Widget Library (JWL), including AJAX-like behavior.
  - Support is provided for JSF portlet bridge.
  - Support is provided for standard JSF only mode (which excludes usage of IBM-specific JSF components) as well as support for third party JSF components.
  - Support is provided for multiple faces configuration files.

- Portal application development:
  - Application Developer V7.0 includes portal development tooling, as well as integrated test environments for WebSphere Portal V5.1 and V6.0.
  - Support is provided for Web Services for Remote Portlet (WSRP).
  - Wizard support is provided for cooperative portlets.
  - Enhanced credential vault support is provided.
  - Support is provided for business process portlet.
- Test server environments:
  - Test environments are included for WebSphere Application Server V6.1, V6.0, V5.1, WebSphere Portal V6.0, V5.1, and WebSphere Express V5.1.
  - Integration with IBM WebSphere Application Server V6.0 for deployment, testing, and administration is the same (test environment, separate install, and Network Deployment edition).
- XML
  - Updated support is provided for XML and XSLT tooling.
  - Updated support is provided for XML schema editing, including visual modeling.
  - Updated support is provided for XML schema to Java code generation.
- Web Services:
  - There is a series of usability improvements in Web Services development (improved skeleton merge for top-down Web Services creation, simplified editing of WSDL and XML schema, remote WSDL validation).
  - Complex schema support is provided with SDO.
  - Enhanced support is provided for XSD.
  - Support is provided for WSDL and XSD modeling.
- Model driven development:
  - Support is provided JET transformations.
  - Updates to general features are provided (such as Project Explorer, model import, and model template).
- Rational Unified Process (RUP) integration:
  - Process Browser provides search capability for RUP best practices and life cycle for development.
  - Process Advisor view displays RUP information specific to current task.
  - This feature is offered in Application Developer and Rational Software Architect.

- Debugging:
  - Support is provided for debugging WebSphere Jython administration scripts.
  - Support for DB2® V9.0 Stored Procedure Debug.
- Additional enhancements:
  - Database tools are provided.
  - There are many enhancements provided around DB2 V9.0.

2

Application planning and design

J2EE design encourages component portability across different operating systems and hardware architectures. More complex designs adapting to new technologies become an important consideration with component development, multi-tier enterprise applications. Questions were asked on how best to design across different application layers and J2EE components.

This chapter provides a starting point for the design patterns and other considerations one should have when designing enterprise applications.

The chapter is organized into the following major sections:

2.1 System capabilities and qualities

Many distributed systems developed eventually fail to deliver a performing solution that is scalable to the customer's requirements, due to any of the following possible limitations:

- Lack of focus on performance during requirement analysis and application design
- Lack of understanding in the intent of patterns in design
- Lack of skilled resources in the target technology to leverage the provided features
- Limited awareness of performance issues by the developers
- A desire to produce an elegant, extremely flexible or highly object-oriented solution

It is important that, at the beginning of the project, performance targets are defined and communicated to all team members. The implications of these targets on development work should also be analyzed early in the development cycle and made known. There is often a trade-off between implementations that are elegant, flexible, and easy to read and maintain; and those that offer higher performance.

Design changes in a later phase of the software life cycle are very expensive. Therefore it is a good idea to conduct load-performance modeling as soon as possible to identify early architecture and design issues that could degrade the performance expectations. It is good practice to verify your design decisions by developing and then load-testing a prototype, especially when adopting new technologies for the first time.

Considerations should be placed on a performance testing harness during the design of systems, and the tools should be developed or acquired in a timely manner to test the system components of concern. Tools such as Rational Performance Tester, OpenSTA, or Apache JMeter exist to simulate load and provide insight as to where performance optimization work should be targeted.

A well designed performing system should have the following main capabilities and qualities:

- Availability
- Scalability
- Flexibility
- Manageability

2.1.1 Availability

Availability is the ability for a system to be operational for the largest percentage of time possible. The most any one system can be operational is 24/7/365 (for 24 hours, 7 days a week, 365 days in a year). The greatest challenge to availability is surviving system instabilities, whether from hardware or software failures.

The ability to have multiple copies of an application working together is a technique known as clustering and is used to provide high availability and performance. Clustering can occur within a single host or, most effectively, across multiple hosts. In Figure 2-1, each copy is on a separate host and a load balancer is used to distributed requests across the two copies.

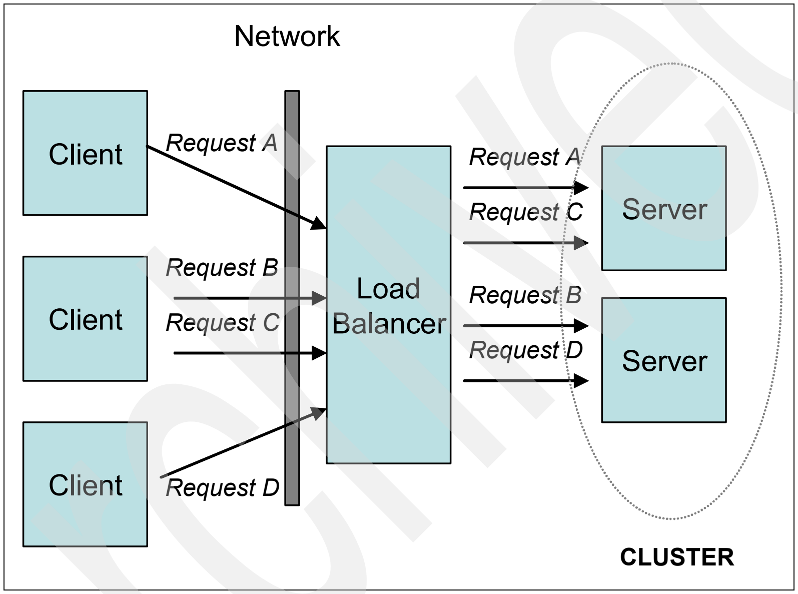


Figure 2-1 Deploying an application in a clustered environment

The risk of the system being unavailable from hardware or software failures is now reduced. If problems arise on one host, the other host is available to process the request. The client is unaware of the inactive server, as its request is seemlessly rerouted to the active server.

2.1.2 Scalability

Scalability is the ability of a system to cater for increases in load, remain available, and be able to perform by adding hardware. The approach adopted can be either or both of the following possibilities:

- Vertical scalability: Adding capacity such as memory and CPU to existing nodes/servers
- Horizontal scalability: Adding more nodes to the same system infrastructure

Here are some considerations to help achieve scalable systems:

- Design platform independent solutions, to cater for future server/node upgrades.
- Design with load balancing in mind, to avoid the reliance of subsequent requests processed by the same server as for the original. This situation arises when client data is maintained in the servicing application's memory.
- Leverage application layering architecture pattern to adequately deploy components to nodes that can best provide performance.

2.1.3 Flexibility

Flexibility is the ease of a system to address change to its environment. Flexible systems are component driven solutions that easily integrate with various technologies, not tied to a specific software vendor or hardware. A design should have the flexibility to allow application components or even entire layers to be deployed onto different application servers. Technology platform changes should have a minimum effect on business processes, as should changes in business process on the technology platforms. Such systems are very adaptable and portable, however, designs that provide greater flexibility are increasingly complex operationally.

2.1.4 Manageability

Manageability is the ability to ensure system integrity when faults occur and to react to them. Logging the occurrence of faults is the absolute minimum a system should provide. Other means of notification should also be considered when fault detection might not reflect the true state of a system. System heartbeat capabilities can be used to indicate that the system is in a functional state but idle. The lack of a heartbeat might indicate that a system's unresponsiveness could be due to errors that have not been logged, leaving it in a hung state.

System state notification is only as effective as the tools used to monitor it. Without enterprise monitoring tools, corrective action can only be taken after something more serious occurs, rather than when the system first begins to show problematic symptoms. Obviously, we want to know as soon as possible when a failure occurs. Enterprise monitoring products such as Tivoli® Management Solutions provide tools to monitor systems for health and fault tolerance and automatically notify administrators in response to specific events. These tools should directly integrate with the system's notification mechanism to provide a highly maintainable system.

2.2 Architectural patterns

The authors of "In Pattern-Oriented Software Architecture: A System of Patterns, by F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal, John Wiley and Sons, 1996, ISBN 0-471-95869-7" define an architectural pattern as follows:

> *"...expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."*

Next, we discuss the following architectural patterns:

- Three tier architecture
- Model View Controller
- Service Oriented Architecture

2.2.1 Three tier architecture

Many applications developed today adopt a three tier architecture. Figure 2-2 shows the basic layering approach adopted in this IBM Redbooks publication.

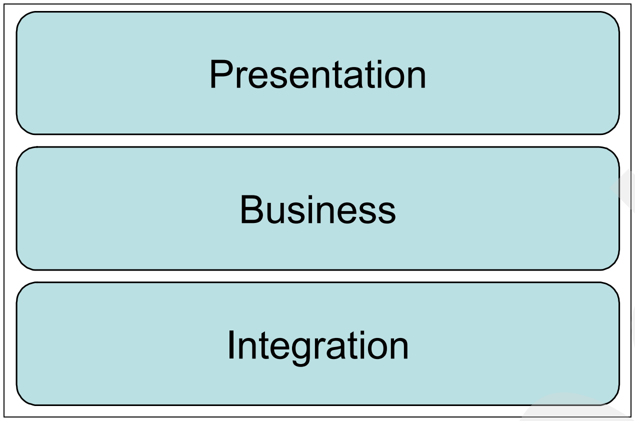


Figure 2-2 Typical Three Tier Architecture

- The Presentation tier is responsible for interacting with the user and submitting requests to the Business tier. Technologies like HTML, JSP, JSF and Servlets are used within this layer.
- The Business tier is responsible for defining the business services provided by the application. Depending on the design patterns adopted, a further subdivision of the layer can exist within.
- The Integration tier manages communication to external systems or resources. Technologies such as Entity Beans, JDBC and J2EE Connectors are used to communicate to data sources, legacy systems, and other enterprise systems.

Decomposing applications into tiers addresses each layer's specific design requirements. Designs can be more comprehensive and in context of what they are trying to achieve rather that focus on the application as a whole. This enables clear demarcation of roles and responsibilities and better placement of application components.

2.2.2 Model View Controller

Model View Controller (MVC) is a architectural design pattern for interactive applications. An interactive application is organized into three separate modules:

- The model, for the business logic and application data representation
- The view, to present the data to the user and accept input
- The controller, to control flow and dispatch requests

Figure 2-3 briefly describes the MVC-architecture: the Web-tier *controller* receives each incoming (HTTP) request and invokes the requested business logic operation in the application *model*. Based on the results of the operation and state of the model, the controller then selects the next view to display. Finally, the *view* renders the contents of the model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes.

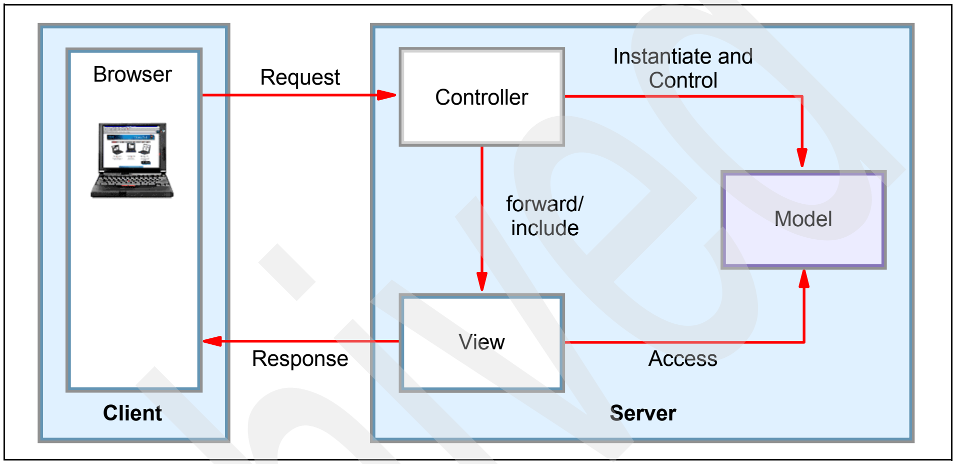


Figure 2-3 Model View Controller architecture

The main advantage is the clear separation between the different design concerns: data persistence, business logic, presentation, and control. This means that changes to the presentation (view) of the data are limited to this layer; the other layers (business logic, control, and data persistence) remain unaffected. The same applies to changes in all other layers. Another advantage is that this separation facilitates parallel application development.

2.2.3 Service Oriented Architecture

Service Oriented Architecture (SOA) is an architectural design pattern that transforms systems into what is essentially a collection of linked services that can be accessed when required over a network. A service is a repeatable business task performed by a service provider to achieve desired end results for a service consumer. In Figure 2-4, we illustrate the exposure of services in the Integration Layer to access our business domain. The Presentation Layer in this example still retains access directly to the Business Layer. For illustration purposes, only services are depicted in the Integration Layer, however, other entities such as those for database access could also exist.

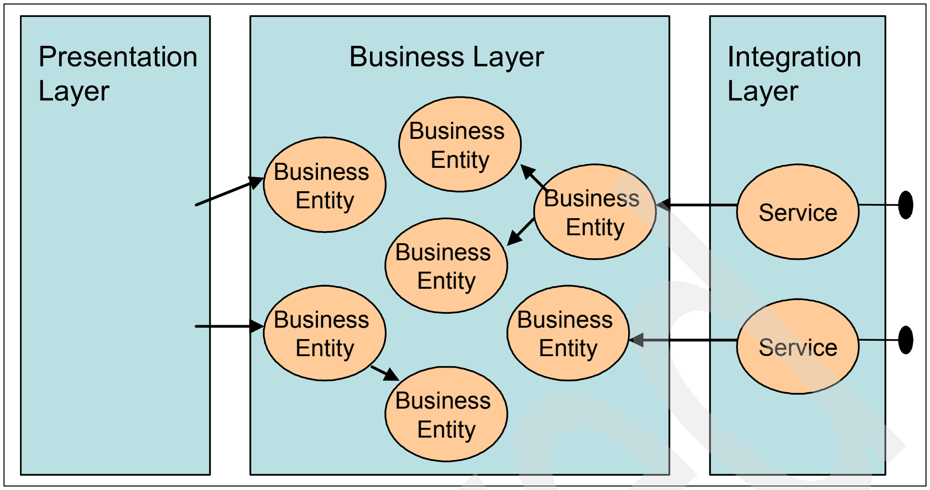


Figure 2-4 Exposing services in the Integration Layer

The goal of SOA is to achieve loose coupling among interacting producers and consumers and promote reusability. Figure 2-5 illustrates the interaction among the distributed SOA entities. This can also take place on a single node.

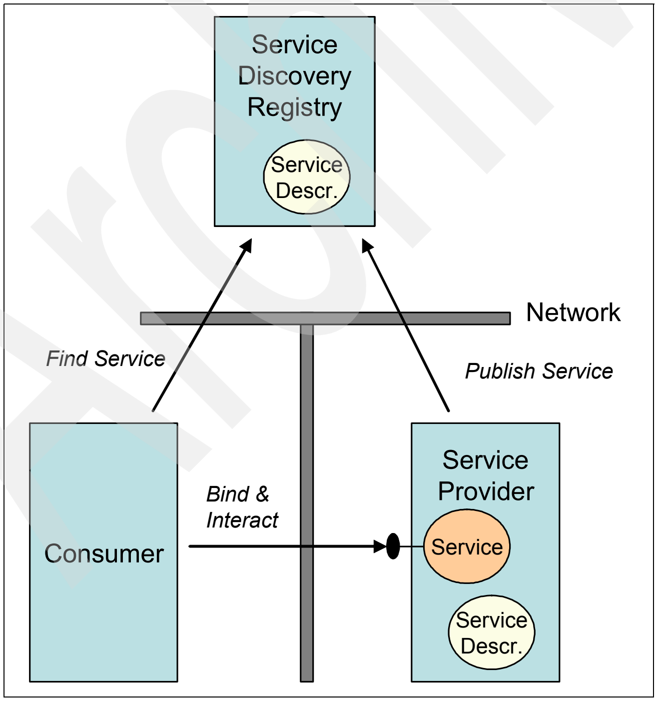


Figure 2-5 Systems composed of consumers and services providers

The service provider (that is, the producer) defines a service description for the Web service and publishes it to a requestor or service discovery registry. The consumers retrieves the service description either locally or from the service discovery registry and uses it to bind and invoke to the Web service.

2.3 Design patterns

Design patterns are an integral part of application design. They are recurring solutions to software design problems and promote the re-use of concepts, ideas, design templates and frameworks that are proven, rather than "reinventing the wheel." Design patterns help to improve software quality and reduce development time. Evolution of technology means that there is no one best design that caters for all scenarios. You have to assess which design patterns meet your application requirements and not develop applications for design patterns.

For each of the three architecture tiers, we focus on some existing design patterns for performance, scalability, and re-use of code. The bibliography lists references for further details on these and other design patterns.

2.3.1 Presentation Layer patterns

In the following sections we discuss the Presentation Layer patterns:

- "Front Controller" on page 31
- "View Dispatcher" on page 34

Front Controller

Use the Front Controller pattern to centralize incoming client requests.

Web applications often require the user to navigate through a set of screens. User requests can be managed in a centralized fashion, where all screen requests are initially serviced by a common component; or decentralized, where each screen deals with the generated requests individually. Figure 2-6 shows the decentralized approach of handling requests.

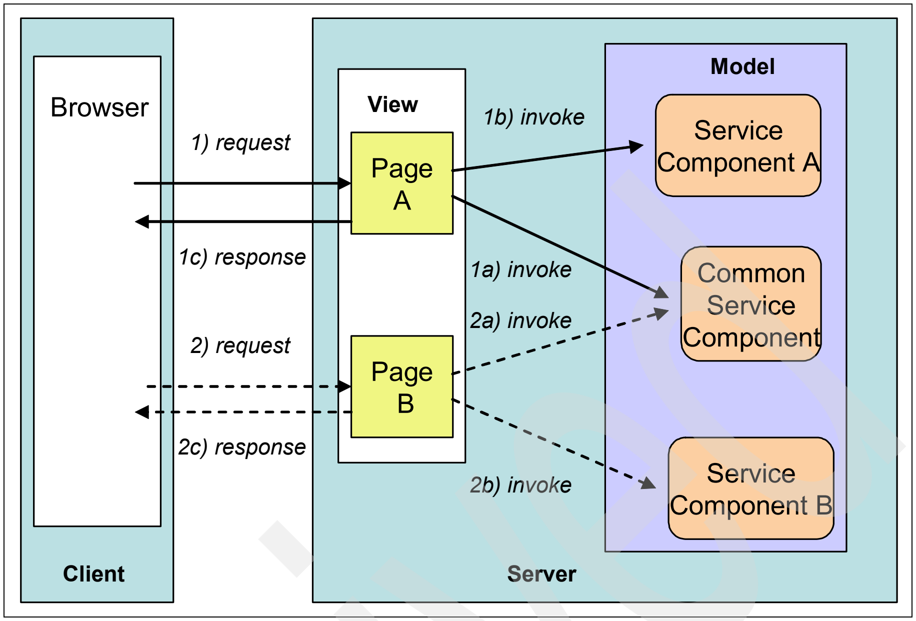


Figure 2-6 Decentralized page control

A request to display Page A is issued from the browser. For Page A to be displayed, Common Service Component and Service A Component have to be invoked. The Web browser then issues a request for Page B. Page B requires interaction with Common Service Component and Service Component B. In this example, the next view to display is determined either by hyperlinks selected in the current page displayed by the Web browser or from the requested URL.

Each page invokes its own model components to fulfill their request. This leads to duplication of business logic when dealing with services that are common across views, like security. When screen navigation is performed by each page, it makes it difficult to clearly understand the different navigation paths a user might take for the entire application.

The Front Control pattern focuses on centralizing control logic in the client layer, dispatching requests to the next page based on the requested URL, input parameters, and application state. It provides a central point for handling Web requests, removing the decision logic performed by each view. This enables services to be easily applied across all screens, such as the loss of a user's interaction across the site, or to handle error scenarios uniformly across the presentation layer.

Figure 2-7 shows how a request resulting in Page A is initially handled by the Front Controller.

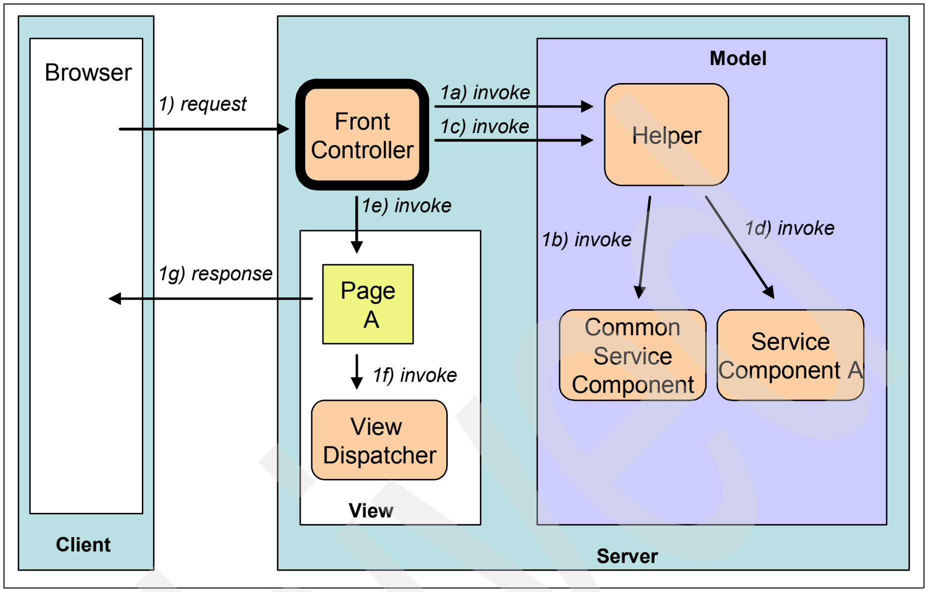


Figure 2-7 Centralized control logic for pages

A request for a dynamic page is issued by the Web browser to the Front Controller. The Front Controller interacts with the model and dispatches Page A to the browser. The same control logic is also used to handle a request resulting in Page B, as shown in Figure 2-8.

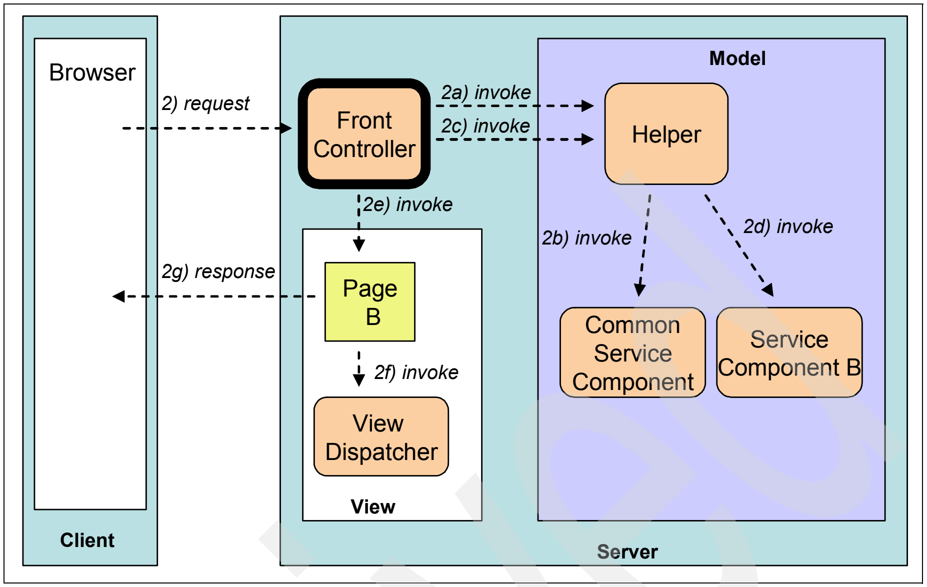


Figure 2-8 Centralized control logic for pages

All requests are initially handled by the Front Controller. The controller can be a servlet, which then delegates the business processing to a number of service components. The helper provides an interface to the Front Controller, hiding the business layer implementation. This can be a Business Delegate for a distributed service component, or lain Old Java Objects (POJO) Facade for non-distributed components. In 2.3.2, “Business Layer patterns” on page 35 we provide further details on the Business Delegate and Facade. Based on the outcome of the business processing, the controller determines the next screen to display. Like the business processing, this can be delegate by the controller to another class, responsible for screen navigation.

We can have multiple controller classes within an application, grouping the application into logical modules such as administration and normal business processing.

View Dispatcher

Use the View Dispatcher pattern to minimize code duplication in pages.

Presentation formatting code logic that can be reused in different screens should be provided as utilities/helpers. This would include JSP tags and HTML code, for example, to create tables. In doing so, code duplication can be minimized in the pages and reused, as it is not particular to the realization of a use case.

2.3.2 Business Layer patterns

In this section, we cover the following Business Layer patterns:

- "Application Service" on page 35
- "Session Facade" on page 36
- "Service Locator" on page 38
- "Business Delegate" on page 40
- "Transfer Object" on page 41
- "Page-by-Page Iterator" on page 42

Application Service

Use the Application Service pattern to centralize business logic.

Application development is based on a set of functional requirements, normally in the form of use cases. This defines not only the objects that are to exist in the domain, but also their interaction. The coordination of multiple Business Objects, Data Access Objects, and Services providing common utilities for realizing use cases should not reside within Business Objects, as this increases coupling and reduces cohesion. We also do not want the coordination to reside in the client, as this can increase dependencies between the presentation and business tiers.

The Application Service pattern encapsulates all object invocations and exposes a simplified coarse grain interface. The invoker of an Application Service can be other Application Services, helper objects, or a presentation tier component via a Session Facade, discussed later in this chapter. Figure 2-9 shows the Application Service interaction with objects in the business tier.

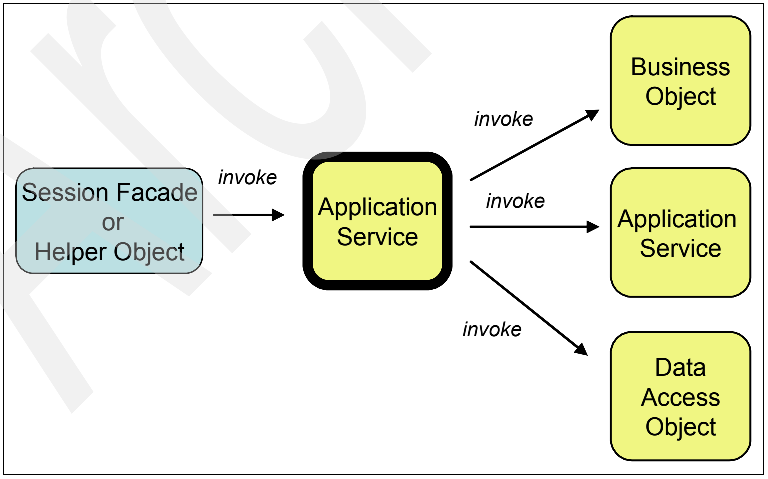


Figure 2-9 Application Service Interaction

Application Services can layered to according to the services provided:

- Common business logic that is not use case specific
- Use case specific business knowledge
- Client channel type specific business processing

Session Facade

Use the Session Facade pattern to provide a unified, workflow-oriented interface to presentation layer.

The Session Facade addresses clients interacting directly with Application Services, Business Objects, or Data Access Objects to perform a workflow. The client’s knowledge of the objects within the business layer, their interactions and dependencies tightly couples both layers together. Modifications to the objects within the business layer would also affect the clients invoking that object. Where objects in the business layer are distributed, each call can become a remote method invocation. Figure 2-10 illustrates an object in the client layer directly invoking the remote Business Objects participating in a workflow.

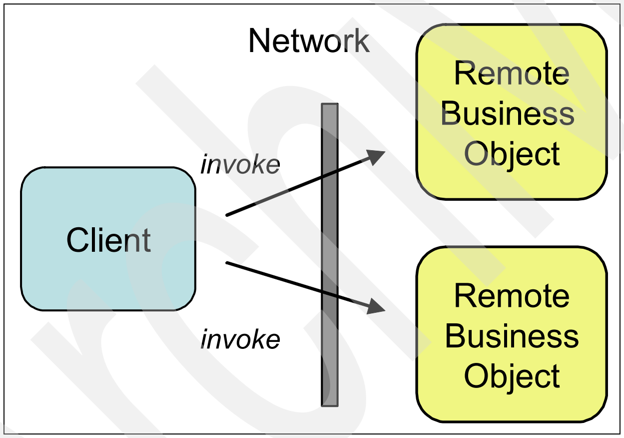


Figure 2-10 Client direct access to remote Business Objects

When interacting with remote objects, the client is placing extra load on the network and increasing processing time. Transaction management becomes more complicated, as each enterprise bean performs a separate transaction on the server side requiring synchronization of remote entity beans with the underlying data store. For cooperating beans to work in a single transaction, the client would require extra logic to cater for errored scenarios and undoing the work done by prior successful bean invocations. Another approach is to adopt the demarcation of a transaction boundary by the client with the usage of Java Transaction API. This would make each remote enterprise bean call perform under the same transaction.

As can be seen from the foregoing approach, this introduces high coupling between the client and business layer, slow performance due to high network usage, and poor transaction management. Figure 2-11 introduces the Session Facade as the mediator between the client and business layer.

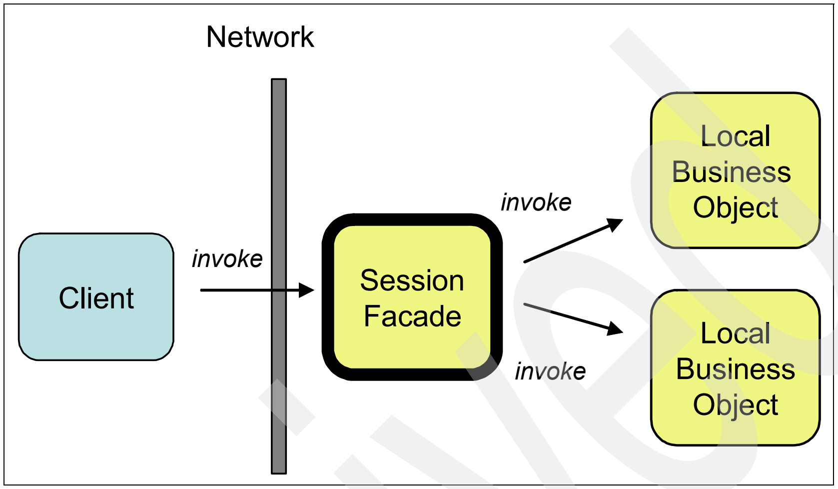


Figure 2-11 Client work delegated to Session Facade

The Session Facade encapsulates the interactions and dependencies between participants in the workflow and exposes to the client a simplified coarse grained set of operations. This removes the complexity of managing Business Objects from the client to the session facade. With a session bean behaving as the Session Facade, the number of remote calls made by the client can be minimized with the bean co-located together with the enterprise business objects.

It is good practice when employing a Session Facade to have the enterprise Business Objects behave as local beans rather than remote, to reduce network overhead. Transaction management is simplified with the session bean demarcating the transaction boundary, such that all updates performed by the enterprise beans within it are treated as a single transaction.

Having a Single Facade for the entire system would prove to be quite a maintenance and manageability headache for a non-trivial application. At the same time, one does not want to create too many facades by having a one-to-one mapping of Session Facades to use cases. It could be best to partition your system in a logical way that in turn defined your facades. Facades would therefore contain several operations, each to a particular workflow within that logical grouping.

For example, all interactions with a particular Business Object such as Purchase Order can be grouped into a PurchaseOrder Facade. Operations such as create, view, update, delete Purchase Order details can be defined within the one facade. This provides highly course-grained access and control to underlying components. A Session Facade works best when it has little or no business logic. The work flows exposed by a Session Facade should be implemented by Business Objects and Application Services with any additional work required to sequence them to be performed the Session Facade. Thus Session Facades can be seen as the remote interfaces for Application Services. For non-EJB applications, their applicability still remains as a controller, even though they are implemented as POJO Facades.

The Session Facade is a synchronous way of accessing the business logic and it should be used whenever immediate feedback is required. This kind of feedback is in general required for read operations. For writing or updating the business model, it might make sense to have a look at the Messaging Facade.

Service Locator

Use the Service Locator pattern to hide the JNDI lookup from the client.

Java Naming and Directory Interface™ (JNDI) is an interface service that associates names (bindings) with the location of services and information. The naming service provides a single location for locating machines, users, objects and services by applications.

Services are identifiable in JNDI by a binding name. When looking up for an object, you provide the name it is bound to. The naming server returns the object or, in some cases, a stub that can be used to interact with the object. Figure 2-12 shows how a naming service in general maintains the bindings to the Enterprise Bean Home object and JMS ConnectionFactory object.

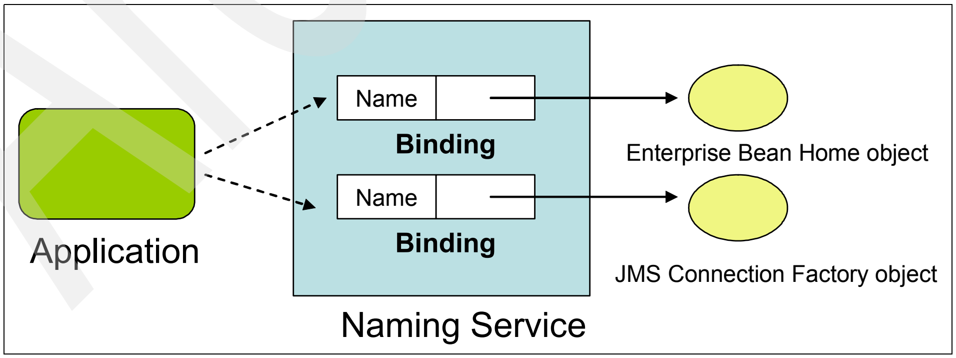


Figure 2-12 Logical view of Naming Service bindings

The Directory service is similar to the Naming service, except that it allows you to associate properties with the objects stored and provides tools for searching them.

The JNDI server is provided by application servers like WebSphere. The EJB container registers application session or entity EJBs with the local JNDI server. Access to the container objects is through the associated JNDI server.

The following tasks are done to perform a lookup of Enterprise Beans:

- Get the initial JNDI context to be used to look up the bean. The initial context contains the name bindings for the enterprise home beans it contains. Note that all JNDI lookups are performed relative to a context.
- The initial context provided by the JNDI service is used by the client to look up the EJB Home interface, based on the specified binding name.
- With the EJB Home interface, you can find, create, and remove the enterprise bean.

Lookup of a JMS QueueSender or QueueReceiver is very similar and is done as follows:

- Obtain the initial context.
- The initial context provided by the JNDI service is used by the client to obtain the ConnectionFactory associated with JNDI binding name.
- With the ConnectionFactory, you can get a QueueConnection.
- Using the QueueConnection, obtain the QueueSession
- With the QueueSession, obtain the QueueSender or QueueReceiver.

Rather than requiring each client to perform the same steps over and over again, we recommend that you introduce a Service Locator. The Service Locator is responsible for retrieving the EJB home interface or JMS Destination. This minimizes duplication of code in clients, as the code used to access the JNDI service is common across all. Creating a JNDI Initial Context and performing a lookup for a resource utilizes significant resources. Performance gains can be obtained by caching the same bean used by different clients to minimize lookups using the InitialContext. However, this can place restrictions on the application; see 2.4, “Cluster considerations” for further details.

In Figure 2-13, for an object, the client delegates JNDI access for the retrieval of the Enterprise Bean Home to the Service Locator. Once the client has acquired the Enterprise Bean Home, the client can invoke methods on it as with any other ordinary Java object.

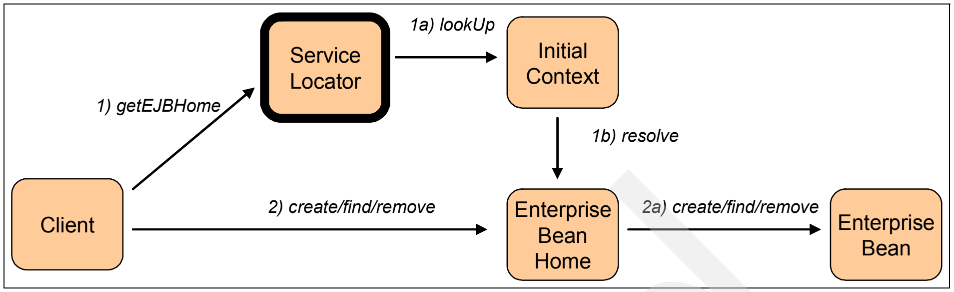


Figure 2-13 Service Locator sequence diagram

A single instance of the Service Locator should exist, and used by all the clients. The lookup of JNDI resources is done with the Initial Context. The Initial Context is only created once, on the first call to the Service Locator. Subsequent calls to the service locator all use the same initial context.

Business Delegate

Use the Business Delegate pattern to hide the complexities of a distributed business layer from the client.

Clients that interact with distributed business components have to look up and retrieve a remote reference before invoking it. They also have to cater for infrastructure and network exceptions that can arise, such as java.rmi.RemoteException or EJBException. The Business Delegate pattern is used to minimize exposure of the distributed business layer from the client. The client should not have to be concerned with the technologies implemented by the business layer such as EJB. The location of business services should be transparent to the client. In Figure 2-14, rather than invoke the Session Facade directly, the client uses the Business Delegate as the intermediatory.

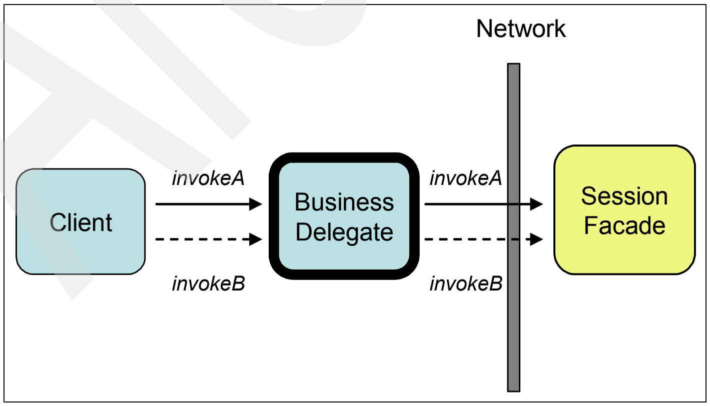


Figure 2-14 Business Delegate masking all remote invocations

The Business Delegate is a Plain Old Java Object (that is, POJO) that behaves like a proxy to the client, hiding the business services and their remoteness. For each remote business service, we have an equivalent business delegate. There is a direct one to one mapping between Business Delegate and business service methods. All remote method invocations are now performed by the business delegate. The client is made transparent of naming and lookup services and uses the business delegate to perform the business service invocations. This lookup service can be apart of Business Delegate, but it is best performed by the Service Locator described in "Service Locator" on page 38.

The Business Delegate can be designed such that it has the ability to retry operations in the business layer on failure. The client is unaware of any attempts for successful execution, only after the number of retry attempts have failed. The Business Delegate translates business service remote exceptions into application exceptions. This also helps shield the client from EJB and network-related implementation details.

Transfer Object

Use the Transfer Object pattern to efficiently transfer remote, fine-grained data by sending a coarse-grained view of the data.

Clients require access to server domain model and the ability to manipulate its contents. The domain model refers to the application mapping of real world entities into objects. In enterprise applications, these objects are developed as entity beans. Allowing client access to the domain model is not a recommended practice. Apart from the tight cohesion that is introduce between the two layers, in a distributed environment, accessing attributes in a remote entity bean can have performance degradation, as each call becomes a remote method invocation.

Using Session Facades resolves the issue with the client's direct invocation with enterprise beans, however, it does not address how the data in the client layer is structured. The Transfer Object pattern also knows, as Value Object provides the client layer with a copy of the domain object's data enabling localized data access and manipulation. In Figure 2-15, the client receives data from the business layer in the form of Transfer Object.

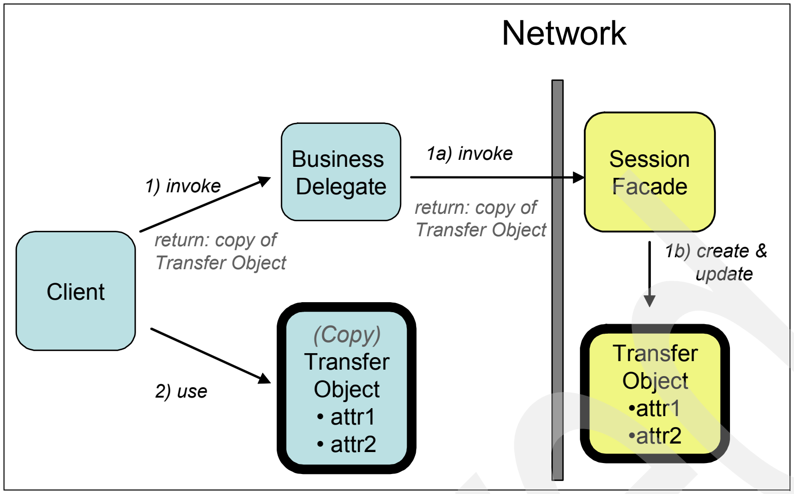


Figure 2-15 Data returned to Client layer in the form of Transfer Object

The client can now access the return data provided in the Transfer Object, decoupled from any of the Entity Beans it might represent. Entity Beans would contain an access method that allows for the session facade to retrieve a Transfer Object version of the bean. The Transfer Object being a POJO means that calls by the client to its access methods are local, thus reducing the network overhead. Modifications to the Transfer Object can be incorporated back into the Entity Bean by providing a method on the bean to copy the values from the Transfer Object to the beans corresponding attributes. Transfer Objects can also be created by the client, encapsulating the data that is to be passed to the Session Facade via the Business Delegate.

Important: As the Transfer Object is created and used across application layers, it has to be serializable. There can be many variations of the Transfer Object for a component, varying the degree of information contained within.

Page-by-Page Iterator

Use the Page-by-Page Iterator pattern to efficiently access a large, remote list by retrieving its elements one sublist of value objects at a time.

The Page-by-Page Iterator design pattern is used to manage large lists of data in response to a client's request. Returning the entire list has a performance impact on the application due to the extra load placed on the network. When the client does not require all the data on the first request, the list can be decomposed into smaller pages and returned one page at a time. The client can specify the size of the list to be returned, with each subsequent request retrieving the next portion of the list.

While this pattern provides efficient access to large lists, the number of data transfers/requests increases. It is important to note that data should not be duplicated, so copies of the lists are not contained by the application's business layer. To achieve this, the last entry in the returned page has to be either sent as a part of the request or maintain by the application. If the data list is concurrently altered by insertions or removals, this interferes with the page results returned to the client.

2.3.3 Integration Layer patterns

Next, we discuss the following Integration Layer patterns:

- "Data Access Object" on page 43
- "Message Facade" on page 44

Data Access Object

Use the Data Access Object pattern to decouple business logic from data access logic.

The Data Access Object (DAO) design pattern decouples the business logic from the data access logic. It provides accesses to the persistence medium, and by exposing uniform data access APIs, the business layer is made unaware of the type of persistence medium accessed/modified. Data sources such as RDBMS, LDAP, and flat files can be interchanged or co-exist without knowledge or impact to business layer code.

DAOs are implemented as stateless POJOs, and do not cache data retrieved from the data store. If caching is required, this should be done in the business layer of the application. Only DAO code is aware of the persistence API required, and how data is accessed/modified. Figure 2-16 illustrates a call made to the DAO to retrieve data from the RDBMS.

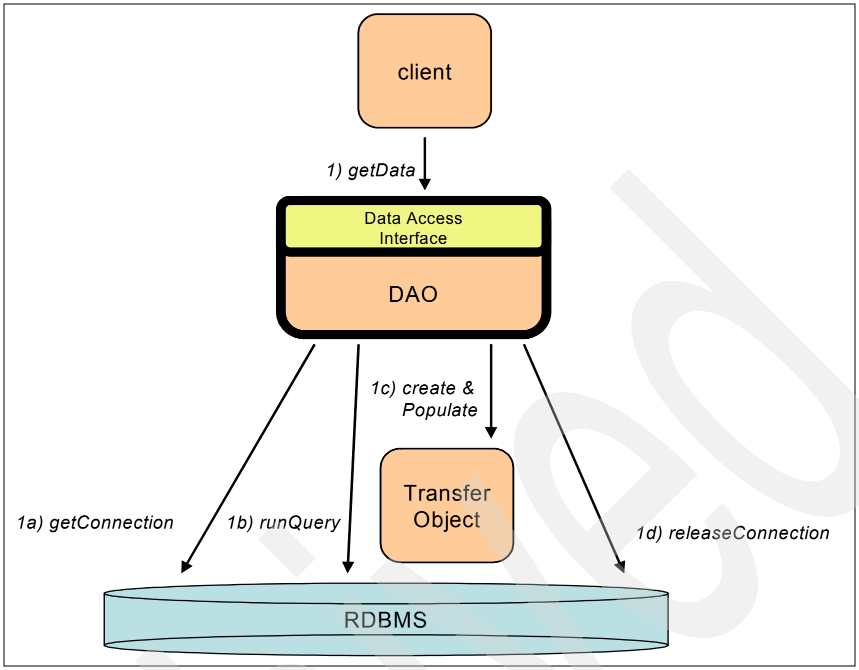


Figure 2-16 Using DAO patter to access RDBMS

Before the DAO retrieves data from the RDBMS, it is required to obtain a connection to the datastore. It is assumed for simplicity that the DAO already has a reference to the datastore. With the datastore connection, it uses JDBC API to run a query on the datastore with the data returned used to populate a Transfer Object. The Transfer Object is then returned to the caller, which it is acted upon. The client is not exposed to any JDBC API and data structures. DAO creation can be made highly flexible, using the Factory pattern as discussed in 2.3.4, “General patterns” on page 46.

Message Facade

Use the Message Facade pattern to provide asynchronous processing of requests.

Response times can be improved when use cases can be identified that executed separately from the client, without requiring the client to wait. This enables large applications to scale effectively, as the request can be queued and processed separately from the clients, allowing the client to progress through the system while their request is actioned by another part of the system. The Message Facade pattern is an asynchronous version of the Session Facade. The client is abstracted from the enterprise objects in the workflow, however, rather than blocking until all the participating enterprise objects in the facade have completed, control is returned to client once the request has been submitted. The facade processes the request independent of the client. To achieve this, the Message Facade is developed using a message-driven bean (MDB). Figure 2-17 shows how a client's request is processed asynchronously.

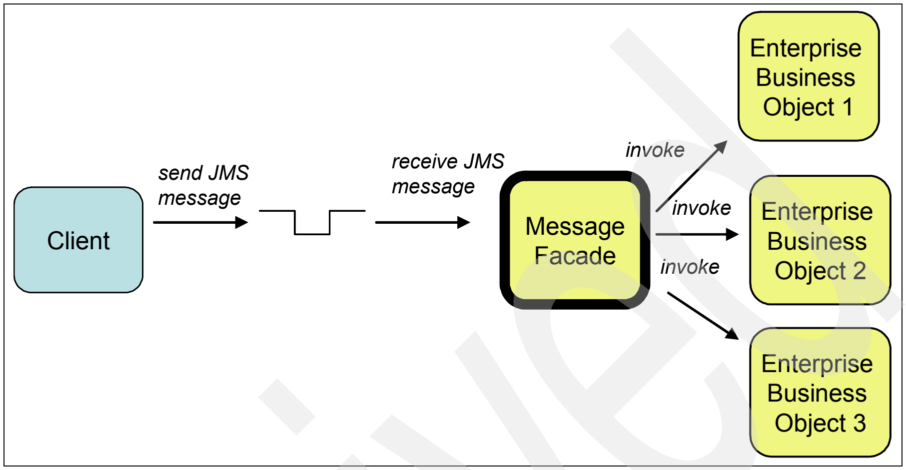


Figure 2-17 Asynchronous processing of a client's request

The client creates a Java Message Service (JMS) message containing the request details and sends it to the JMS destination, a queue in this example. Application control is returned to the client to continue processing. The message bean container allocates the next available Message Facade bean to process the JMS message. If one is not available in the pool of message beans, it waits without impacting the client response time. Aside from the asynchronous processing difference between Session and Message Facade, another and just as important distinction is the guarantee of execution. If the Message Facade transaction fails, for instance, due to the unavailability of an interface system, the transaction is rolled back and the JMS message is placed back on the queue for processing at a later time/date.

This retry is performed without the client's knowledge. As the client is not notified of the success of the Message Facade when it regains application control, a mechanism must be devised if it is required to communicate the result back to the client. This can achieved using the polling model, where the client with a given reference identifier can find the status of an asynchronous request. Another approach might be adopting the event model, where once the message facade completes processing, an event is raised to notified the result. This event might trigger a Web service call to the client, passing back the result. A drawback of using the Message Facade is the mapping of the request into a JMS message. The request object, if serializable, can be stored directly into the JMS message, and extracted by Message Facade actioning it.

2.3.4 General patterns

The following Integration Layer patterns are covered:

- “Singleton” on page 46
- “Factory” on page 46

Singleton

Use the Singleton pattern to provide a single instance of an object.

Some application resources are exclusive in that there is one and only one of this type of resource. A single instance of the class managing the resource might be required throughout the application, such as for logging, caching, thread pools, handling preferences, database access, and factory implementations. If more than one was to be instantiated, it might result in overuse of a resource or encounter problems with inconsistency.

One approach is to use global variables; in Java this would be a static Class attribute. The problem with this approach is that the object might be created when the application starts up. This could be a resource intensive operation in which the application might never use it. Another approach is to use the Singleton pattern, where an instance of the object is created when it is required.

The responsibility for creating an instance of the class is on itself, rather than the caller. The caller is unable to explicitly instantiate the class, it only requests an object of that class to be returned. Typical use of this pattern is in service-like classes.

Factory

Use the Factory pattern to create more than one kind of object.

Many of the design patterns encourage loose coupling. To understand this concept, it is easiest to talk about a struggle that many developers go through in large systems. The problem occurs when you change one piece of code and watch as a cascade of breakage happens in other parts of the system, parts you thought were completely unrelated. The problem is tight coupling. Functions and classes in one part of the system rely too heavily on behaviors and structures in other parts of the system’s functions and classes. You require a set of patterns that lets these classes talk with each other, but you do not want to tie them together so heavily that they become interlocked.

In large systems, lots of code relies on a few key classes. Difficulties can arise when you have to change those classes. Figure 2-18 shows an example of two classes tightly coupled, one using the other class to read from a file.

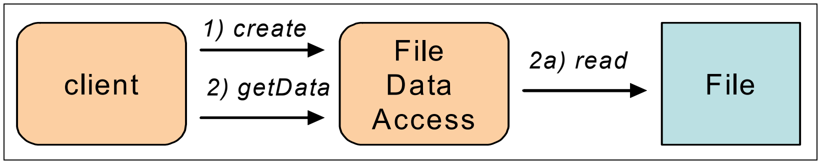


Figure 2-18 Tightly coupled classes

In this diagram, the client represents an object that requests a service. If you want to change the client to use a different class that reads from the database, all code references that read from a file must be removed. This is where the Factory pattern comes in handy.

The Factory pattern is a class that has methods that create objects for you. Instead of directly creating the data access class, you use the factory class to create the appropriate object. That way, if you want to change the types used by your client, you can change just the factory. All the code that uses the factory changes automatically.

An important to point to note with the factory implementation is that its access methods have an interface for their return type. The caller is not aware of the object types returned by the factory, only the interface they implement. Figure 2-19 shows how the factory is used to provide the appropriate object to the caller for retrieval of data.

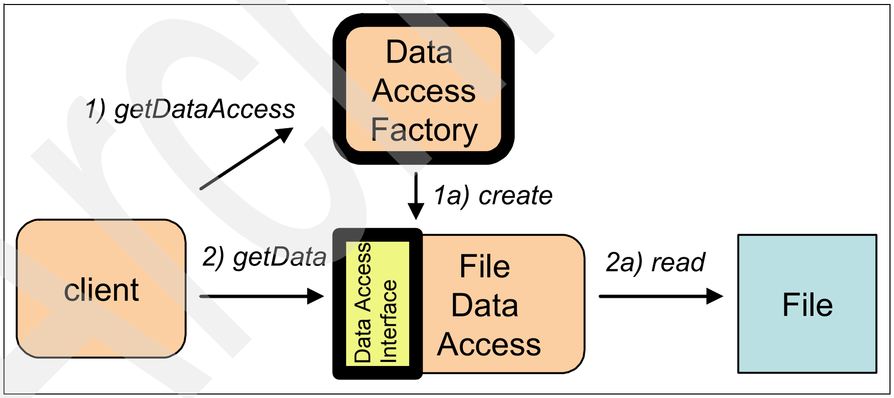


Figure 2-19 Factory returns an object to access a data file

The client is unaware that it is accessing an object of type file data access. Its only concern is that the data access factory has provided an object to service the request identified to the factory. The exposed interface must be implemented by the returned object, in this example, to retrieve data from the persistence medium. The interface clearly defines all the operations that are to be provided by the object returned from the factory. How the factory determines which object to return can be configured via a property file, database table, JNDI, and so on. Using the key provided by the client on the request of a data access class, it can obtain the mapped class, instantiate it, and return the object to the client.

Figure 2-20 shows how the factory can be modified to return an object that accesses the database.

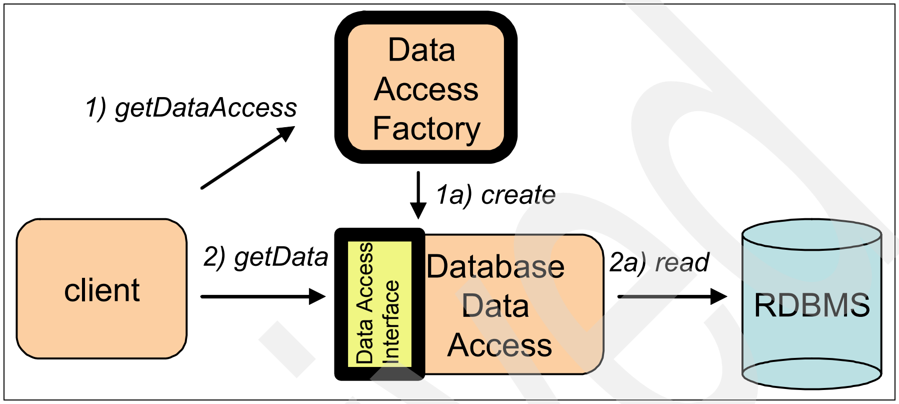


Figure 2-20 Factory returns an object to access a database

To simplify the example, we do not depict how the client obtains an instance of the factory. To limit the number of factory objects created within an application, one should adopt the Singleton pattern.

2.4 Cluster considerations

An important characteristic of any design is the ability for it to be able to scale under load. Application performance involves more than applying coding best practices, it is how one's design is able to leverage off the application server's capabilities, such as clustering provided by the IBM WebSphere Application Server Network Deployment Edition.

Utilizing clustering, a group of servers appears to a client as a single application server. From a client perspective, the existence of multiple servers is transparent. The client does not have to be aware that it is dealing with multiple copies of an application server (in a cluster,) rather than a single stand-alone application server. Application load can be shared across application servers improving performance, and fail over capabilities increase with more than one server available to handle requests. Most application servers allow you to utilize the clustering support without having to change the application, if considered during application design.

If an application is not designed to be clustered, functional or performance issues can arise when deploying your application to such an environment. A common mistake during design is expecting the application to run only on a single JVM™. This severely restricts an application's ability to scale when load is beginning to impact performance and application response times. This situation could be alleviated if clustering strategies were considered during design. With a clustered environment, multiple application instances are now executing on different JVMs on the same or across different hosts. Subsequent requests now can be processed by different instances of the application, so the sharing of resources and data between instances has to be considered.

Next, we discuss the following design considerations in further detail:

- Some objects cannot be shared across a cluster
- Avoid using local files to contain mutable configuration data
- Cater for concurrent data access
- Leverage the caching capabilities provided by application servers
- Use application server provided workload management and replication
- Keep user session state to a minimum

Some objects cannot be shared across a cluster

To enable any instance of an application server within a cluster to handle a request, most of them must share some objects. However, there are objects such as timed services and file services that are not going to be able to take advantage of being shared between instances of the application server, even if they are associated with objects that are shared across the cluster. The result is that each server is running its own instance. If this is an issue, then a checking mechanism has to be adopted to ensure that only one instance is ever running. This could be a flag set in the database shared by all instances in the cluster.

Avoid using local files to contain mutable configuration data

It is common to have properties files containing application configuration data. This data is normally contained in applications as Java objects accessed using static fields or a singleton. A single instance of the data is shared across the application and any changes are accessible to the entire application on the JVM. Even though the changes are persisted back in the property file, when deployed in a clustered environment, these updates are not reflected to all running copies of the application. A preferred practice is to use a database or LDAP server to keep the configuration data.

Cater for concurrent data access

With multiple copies of an application using the same shared data, concurrent data access becomes more significant. Application servers provide a level of concurrency, such as that Servlet threads never process the same HTTP request. However, applications require appropriate levels of locking in a transaction to ensure for instances records with a database are not altered concurrently. These mechanisms, even if they are considered during design, can still lead to data corruption or duplicate processing if not properly addressed for a clustered environment.

Locking mechanisms serializing modification or processing of shared data should not be restricted within an application's JVM. Maintaining a list of requests processed by an application in memory does not prevent an application copy from processing the same request. Using a database table in this instance to register requests being actioned would provide visibility to all copies of the application. With locking, caution must to be heeded to ensure that deadlocks are not possible.

Leverage the caching capabilities provided by application servers

Application servers can help to improve application performance with the caching capabilities provided for read-only EJBs, Java objects, Web Services, and servility/JSP/JSF tags. Some application servers support caching across application copies in a cluster. The policies governing when and how often updated data in a cache is reflected across the cluster should be taken into consideration during application design, to achieve the desired performance improvements and behavior. Two such policies are "End of Service" (EOS) and "Time Based Writes" (TBW).

The EOS policy is catered for regular cache updates, with synchronization occurring at the end of a request. The TBW policy synchronizes cache updates in batches at certain time intervals. This provides better performance, as the frequency of the updates is less, however, there is more data inconsistency, which is not ideal for requirements of data with high integrity.

Use application server provided workload management and replication

Before designing a solution, become familiar with services provided by the targeted application server. WebSphere provides capabilities for cluster scalability, including workload management and failover, HTTP session affinity, HTTP session and cache replication services. The provided services should simplify the design of your application, and if well designed, should not require code changes when deployed in a clustered environment.

Keep user session state to a minimum

HTTP is a stateless protocol requiring user state to be stored on the server. In a stand-alone application environment, all user requests are service by the one server with HTTP sessions maintaining user state. However in a cluster environment, different requests from the same user might go to different application servers. Application servers provide the ability to share HTTP session data across servers through workload routing and caching, eliminating the necessity for the server servicing the request to repopulate the session object from the database or shared memory.

To enable efficient session sharing, adhere to the following recommendations:

- Session objects should be small with little complex data structures as possible. Keep only key data.
- Ensure that the session data represents the user's state, not request history.
- The state data maintained in the session object should be specific for that user and not just application data.
- All Java objects stored in a session have to be serializable.
- Define object attributes that do not have to be persisted as transient.

2.5 Best practices

You should be aware that the "best" in best practices is situational. There are certain situations where design decisions that are contrary to the "best" practices listed below are actually good. Just keep in mind that there are always several different factors that lead to a specific design decision, so being clear about the factors contributing to the decision is key.

2.5.1 Set performance goals early

At the beginning of design, ensure that you have a clear understanding of the requirements for the system qualities and capabilities so designs and tests can incorporate them. Avoid ambiguous or incomplete goals that cannot be measured, such as "the application must run fast" or "the application must handle increased load." Make sure that your goals are measurable and verifiable.

Requirements to consider include response times, workload, throughput and resource. For example:

- How long should a particular request take?
- What is the peak load the application must handle?

- How many users does your application have?
- How critical is its availability?

2.5.2 Validate your architecture and design early

Ensure that the application architecture and design can support your performance goals up-front by validating the important decisions like deployment topology, load balancing, authentication and authorization strategies, database design, data access strategies, state management, and caching. Decisions that do not meet the performance goals would have to be revisited. Thus, ensure that the costs of specific design choices are considered and documented for future reference.

2.5.3 Always use the Model View Controller architecture pattern

Cleanly separate business logic (Java beans and EJB components) from controller logic (servlets/Struts actions) from presentation (JSP, XML/XSLT). Good layering can cover many coding problems.

This practice is so central to the successful adoption of Java EE that there is no competition for the MVC architecture. Model-View-Controller is fundamental to the design of good Java EE applications. It is simply the division of labor of your programs into the following parts:

1. Those responsible for business logic (the Model — often implemented using Enterprise JavaBeans™ or plain old Java objects).
2. Those responsible for presentation of the user interface (the View).
3. Those responsible for application navigation (the Controller — usually implemented with Java servlets or associated classes like Struts controllers).

There are a number of problems that can emerge from not following basic Model View Control architecture. Most problems occur from putting too much logic into the view portion of the architecture. Performing application flow control or database access within a JSP are relatively common in small-scale applications. These can cause issues in later development as JSPs become progressively more difficult to maintain and debug.

The same can also be noted for business logic containing view layer constructs, such as having XML parsing technologies used in the construction of views contained within the business layer. The business layer should operate on business objects, not on a particular data representation tied to the view.

However, just having the proper components does not make your application properly layered. It is quite common to find applications that have servlets, JSPs, and EJB components, where the majority of the business logic is done in the servlet layer, or where application navigation is handled in the JSP. You must be rigorous about code review and refactoring to ensure that business logic is handled in the Model layer only, that application navigation is solely the province of the Controller layer, and that your Views are simply concerned with rendering model objects into appropriate HTML and Javascript.

User interface technologies change rapidly, and tying business logic to the user interface makes changes to “just the interface” deeply impact existing systems. Just a few years ago, user interface developers for Web applications could choose from servlets and JSPs, Struts, and perhaps XML/XSL transformation. Since then, Tiles and Faces have become popular, and now AJAX is gaining a strong following. It would be a shame to have to redevelop an application's core business logic every time the preferred user interface technology changes.

2.5.4 Do not “reinvent the wheel”

Use common, proven frameworks such as Apache Struts, JavaServer Faces, and Eclipse RCP. Use proven patterns. Struts and JSF are not only well accepted in the Java community, but fully supported within the WebSphere runtimes and Rational tool suites as well. Likewise, in the rich client arena, the Eclipse Rich Client Platform (RCP) has also gained wide acceptance for building stand-alone rich clients. While not a part of the Java EE standard, these frameworks are now a part of the Java EE community, and should be accepted as such.

2.5.5 Develop to the specifications, not the application server

Know the specifications by heart and deviate from them only after careful consideration. Just because you can do something does not mean you should.

It is very easy to cause yourself grief by trying to play around at the edges of what Java EE enables you to do. Developers can dig themselves into a hole by trying something that they think might work “a little better” than what Java EE allows, only to find that it causes serious problems in performance, or in migration (from vendor to vendor, or more commonly from version to version) later. In fact, this is such an issue with migrations, that it calls this principle out as the primary best practice for migration efforts.

There are several places in which not taking the most straightforward approach can definitely cause problems. A common one today is where developers take over Java EE security through the use of JAAS modules rather than relying on built-in spec compliant application server mechanisms for authentication and authorization. Be very wary of going beyond the authentication mechanisms provided by the Java EE specification. This can be a major source of security holes and vendor compatibility problems. Likewise, rely on the authorization mechanisms provided by the servlet and EJB specs, and where you have to go beyond them, make sure you use the spec's APIs, such as getCallerPrincipal(), as the basis for your implementation. This way, you are able to leverage the vendor-provided strong security infrastructure and, where business has a greater requirement, then support more complex authorization rules.

Other common problems include using persistence mechanisms that are not tied into the Java EE spec (making transaction management difficult), relying on inappropriate Java Standard Edition facilities (like threading or singletons) within your Java EE programs, and creating your own solutions for program-to-program communication instead of staying within supported mechanisms like Java 2 Connectors, JMS, or Web Services. Such design choices cause no end of difficulty when moving from one Java EE compliant server to another, or even when moving to new versions of the same server.

Using elements outside of Java EE often causes subtle portability problems. The only time you should ever deviate from a spec is when there is a clear problem that cannot be addressed within the spec. For instance, scheduling the execution of timed business logic was a problem prior to the introduction of EJB 2.1. In cases like this, we might recommend using vendor-provided solutions where available (such as the Scheduler facility in WebSphere Application Server), or to use third-party tools where these are not available. Today, of course, the EJB specification now provides for time-based function, so we encourage the use of standard interfaces. In this way, maintenance and migration to later spec versions becomes the problem of the vendor, and not your own problem.

Be careful about adopting new technologies that are yet to be integrated into the rest of the J2EE specification, or into a vendor's product. Support is critical, where if your vendor does not directly support a particular technology proposed in a JSR that is not yet accepted into J2EE, you should probably not pursue it.

2.5.6 Employ iterative development

Iterative development allows you to gradually master all the moving pieces of J2EE, rather than doing everything at once. For a development team that is just starting with J2EE, it is difficult to try to learn it all at once. The key to success in this environment is to take J2EE on in small, controlled steps. This approach is best implemented through building small, vertical slices through your application. Once a team has built its confidence by building a simple domain model, back-end persistence mechanism (perhaps using JDBC), and thoroughly tested the model, they can then move on to mastering front-end development with servlets and JSPs that use that domain model. If a development team finds a requirement for EJBs, they could likewise start with simple Session Facades atop Container-Managed persistence EJB components or JDBC-based Data Access Objects (DAOs) before moving on to more sophisticated constructs such as Message-Driven beans and JMS.

Iterative development of each application layer fosters the application of appropriate patterns and best practices. If you begin with the lower layers of your application and apply patterns like Data Access Objects and Session Facades, you should not end up with domain logic in your JSPs and other View objects.

When you do development in thin vertical slices, it makes it easier to start early in performance testing your application. Delaying performance testing until the end of an application development cycle is a sure recipe for disaster.

2.5.7 Always use Session Facades whenever you use EJB components

Never expose entity beans directly to any client type. Only use local EJB interfaces for entity types. Using a session facade is one of the best-established practices for the use of EJB components. The lower the distribution "cross-section" of your application, the less time is wasted in overhead caused by multiple, repeated network hops for small pieces of data. The way to accomplish this is to create very large-grained facade objects that wrap logical subsystems and that can accomplish useful business functions in a single method call. Not only does this reduce network overhead, but within EJBs, it also critically reduces the number of database calls by creating a single transaction context for the entire business function.

EJB local interfaces provide performance optimization for co-located EJBs. Local interfaces must be explicitly called by your application, requiring code changes and preventing the ability to later distribute the EJB without application changes. Because the Session Facade and the entity EJBs it wraps should be local to each other, we recommend using local interfaces for the entity beans behind the Session Facade. However, the implementation of the Session Facade itself, typically a stateless session bean, should be designed for remote interfaces.

For performance optimization, a local interface can be added to the Session Facade. This takes advantage of the fact that most of the time, in Web applications at least, your EJB client and the EJB are co-located within the same JVM. If you use a remote interface (as opposed to a local interface) for your Session Facade, then you might also be able to expose that same Session Facade as a Web service in a J2EE 1.4 compliant way. (This is because JSR 109, the Web Services deployment section of J2EE 1.4, requires you to use the remote interface of a stateless session bean as the interface between an EJB

Web service and the EJB implementation.) Doing so is often desirable, since it can increase the number of client types for your business logic.

2.5.8 Acquire shared resources late and release early

Minimize the duration that shared and limited resources such as network and database connections are held onto.

2.5.9 Put the processing closer to the resources it requires

If your processing involves a lot of client-service interaction, you might have to push the processing closer to the client. If the processing interacts intensively with the data store, you might want to push the processing closer to the data.

2.5.10 Embrace Java EE, rather than faking it

Commit to building real Java EE applications that truly leverage Java EE function.

One of the most disturbing things we have observed, more than once, is an application that claims to "run in WebSphere" but is not really a WebSphere application. We have seen several examples where there is a thin piece of code (perhaps a servlet) in WebSphere Application Server and all of the remaining application logic is actually in a separate process; for example, a daemon process written in Java, C, C++ or whatever — but not using Java EE — does the real work.

That is *not* a real WebSphere Application Server application. Virtually all of the qualities of service that WebSphere Application Server provides are not available to such applications. This can be quite a rude awakening for folks who think that they have a WebSphere Application Server application.

3

General coding considerations

This chapter describes performance and scalability considerations when coding an application. Our discussion is not specific to any application layer — we leave those details to subsequent chapters. New features of Java 5 are highlighted at the end of this chapter.

The chapter is organized into the following major sections:

3.1 General considerations

This section describes a variety of techniques to improve performance of WebSphere applications, particularly through the efficient use of the core Java functionality. Here are some considerations:

- Although the reflection facilities in Java can be extremely useful and allow for elegant implementations, reflection is an expensive operation that should not be used indiscriminately. This is another case where a trade-off between performance and elegance of the solution should be made.
- Avoid creating excessively complicated class structures. There is a performance overhead in loading and instantiating these classes.
- Avoid excessive and repeated casting. Once an object has been cast, assign a variable of the correct type and reuse this reference.
- Use of "? :", where the equivalent "if" blocks simply assign one value or another, provides better performance for most JVMs.
- When iterating n items, iterating from n-1 to 0 instead of 1 to n is quicker for most JVMs. See Example 3-1.

Example 3-1 Iterating through a loop n times

```
for (int i=n-1;i>0;i--)
{
   // Do something in a loop.....
}
```

- Avoid repeatedly calling the same method within a loop if the result is the same every time. Instead, store the value in a variable prior to entering the loop and use this stored value for each loop iteration.
- Where possible, declare methods with the `final` modifier if they are not to be overridden. Final methods can be optimized by the compiler by method in-lining. The byte code of the method is included directly in the calling method. This avoids the overhead of performing a method call.
- When reading and writing small amounts of data, use of the Java Buffered I/O classes can significantly improve performance, by minimizing the number of actual I/O calls that should be made.
- It is best to avoid spawning of new threads. Spawned threads do not have access to J2EE resources such as JNDI, security, or transaction contexts. Rather than spawning a thread to act as a server to receive incoming messages, consider using message-driven beans (MDBs).

- In many applications, performance can be improved by performing some caching of data by the application. Note that if this is done, consideration must be given to periodically flushing the cache and the usage of SoftReferences to avoid it growing continuously. Also be careful with making assumptions about requests for a client always being served by a particular application server instance. Even if session affinity is used, in a failover situation, HTTP requests can be serviced by a different application server instance, which might not have the cached data. We recommend that the cache be implemented using a well-defined interface, and that data that is not in the cache be retrieved again, transparent to the rest of the application.

> **Note:** You can use the WebSphere Dynamic Cache service to intercept calls to cacheable objects and store their output in a Dynamic Cache. Refer to the content related to dynamic caching in the WebSphere InfoCenter at:
>
> http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/tdyn_dynamiccache.html

3.1.1 Reviews and testing

It is important to conduct reviews during software development. Coding should not commence until designs have been reviewed. The earlier that issues and concerns are detected within application development, the less effort is required to rectify them. It is imperative that reviews be conducted with team members who have adequate development experience to foresee potential performance and scalability problems within an application.

Performance testing should be done as the application components are developed, not only on the final version of the application. As issues arise, that application can be modified to ensure that it performs and is scalable before other components that might be affected are built. Performance testing the final version of the application might require running different scenarios over several days, even weeks. Application tuning should only be done using profiling tools such as those provided by IBM Rational Application Developer, Borland OptimizeIt, and Sitraka JProbe (by Quest Software). These tools can help you to find bottlenecks in your code, blocked threads, excess object creation, and unused objects that survive garbage collection.

It is also important to ensure the test environments mirror the production environment. Do not be guilty of improper testing due to cost considerations.

3.2 Garbage collection

Garbage collection has an important role when considering application performance. Design decisions such as object pooling are introduced to minimize the creation and, in turn, cleanup by the garbage collector of objects no longer in use.

Unlike other programming languages, Java does not require (or even allow) programmers to explicitly allocate and reclaim memory. The Java Virtual Machine (JVM) runtime environment allocates memory when a new object is created, and reclaims the memory once there are no more references to the object. This reduces the amount of coding required, as well as minimizing the potential for memory "leaks" caused by the programmer forgetting to deallocate memory once it is no longer required. Additionally, Java does not allow pointer arithmetic. Memory deallocation is performed by a thread executing in the JVM called the garbage collector (GC).

In the early days of Java technology, JVMs performed poorly in both memory allocation and garbage collection. There were many articles advising developers to avoid creating temporary objects unnecessarily because allocation and the corresponding garbage collection overhead was expensive. Approaches such as object pooling were adopted to avoid allocation, which used to be good advice, but is no longer generally applicable to all but the most performance critical situations. In fact, object pooling can now pose performance loss for all but the most heavyweight of objects, and even then it is tricky to get right without introducing concurrency bottlenecks.

The introduction of generational collectors in JDK™ 1.2 has enabled a much simpler approach to allocation, greatly improving performance. Efficient collection is made possible by focusing on the fact that a majority of objects "die young". A generational garbage collector divides the heap into multiple generations with most JVMs using two generations, a "young" and an "old" generation. Garbage collection occurs in each generation when a configurable amount of the generation fills up.

Objects are allocated in a generation for younger objects, or the "young" generation, and because of infant mortality, most objects die there. When the young generation fills up, it causes a minor collection. Minor collections can be optimized assuming a high infant mortality rate. A young generation full of dead objects is collected very quickly. The vast majority of objects in most Java applications become garbage before the next collection. The cost of a minor garbage collection is proportional to the number of live objects in the young generation, not the number of objects allocated since the last collection.

Because so few young generation objects survive to the next collection, the cost of collection per allocation is fairly small (and can be made even smaller by simply increasing the heap size, subject to the availability of enough memory). Some surviving objects are moved to an “old” generation if they survive past a certain number of garbage collections. They are considered “long lived” and get promoted into the old generation as they are less likely to be collected. When the “old” generation has to be collected, there is a major collection that is often much slower because it involves all live objects.

As the JVM can support many threads of execution, concurrent access to the shared heap can become a problem.Therefore, it must be protected by a resource lock so that one thread can complete updates to the heap before another thread is allowed in. Access to the heap for most JVMs is therefore single-threaded. Most JVMs use thread-local allocation blocks to avoid this, areas of the heap that are allocated as a single large object, marked non collectable, and allocated to a thread. Threads can now sub allocated from the thread local heap, objects that are below a defined size. No heap lock is required, so allocation is very fast and efficient. When a cache becomes full, a thread returns the thread local heap to the main heap and grabs another chunk for a new cache. As a result, the number of times a thread has to acquire the shared heap lock is greatly reduced, improving concurrency.

More details about garbage collection can be found in:

- “Garbage collection policies” by Mattias Persson

  http://www-128.ibm.com/developerworks/java/library/j-ibmjava2/index.html

- “Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory using JDK 1.4.1” by Nagendra Nagarajayya and J. Steven Mayer

  http://developers.sun.com/techtopics/mobility/midp/articles/garbagecollection2/

3.2.1 Vertical clustering

Most current garbage collections still require single threading at some point, as when performing major collection. JVMs have been optimized to try to keep this minimal, however, this causes all other program threads to stop, potentially increasing the response times experienced by users of the application. The length of each garbage collection call is dependent on numerous factors, including the heap size and number of objects in the heap. Thus as the heap grows larger, garbage collection times can increase, potentially causing erratic response times depending on whether a garbage collection occurred during a particular interaction with the server. The effect can be reduced by using vertical scaling and running multiple copies of the application on the same hardware.

Provided that the hardware is powerful enough to support vertical scaling, this can provide two benefits: first, the JVM for each member of the cluster only requires a smaller heap, and secondly, it is likely that while one JVM is performing garbage collection, the other one should be able to service client requests, because the garbage collection cycles of the JVMs are not synchronized in any way. However, any client requests that have been directed by Workload Management to the JVM (doing garbage collection) are affected.

3.2.2 Explicit garbage collection

A common mistake by developers is the use of System.gc() in their application code to trigger garbage collection. This call only suggests that the Java Virtual Machine spend effort towards garbage collection. Garbage collection occurs asynchronously when free memory reaches threshold values, and it cannot be explicitly scheduled programmatically. A call to the `System.gc()` method requests the JVM to perform garbage collection. However, this is not guaranteed to happen immediately or within any specified time period. Do not try to control the garbage collector or to predict what might happen in a given garbage collection cycle. You cannot do it. This unpredictability is handled by the JVM, and the garbage collector is designed to run well and efficiently inside these conditions. Therefore let the garbage collector run in the parameters that an application selects at start-up time. This approach nearly always produces best performance.

Forcing the garbage collector to run can severely degrade the JVM performance. The System.gc() triggers a full collection, which includes tracing all live objects in the heap and sweeping and compacting the old generation. This can be a lot of work. In general, it is better to let the system decide when it should collect the heap, and whether or not to do a full collection. Most of the time, a minor collection does the job. Worse, calls to System.gc() are often deeply buried where developers might be unaware of their presence, and where they might get triggered far more often than necessary. If you are concerned that your application might have hidden calls to System.gc() buried in libraries, you can invoke the JVM with the -XX:+DisableExplicitGC option to prevent calls to System.gc() and triggering a garbage collection.

Tools exist that parse verbose GC trace, analyze Java heap usage, and recommend key configurations for optimizing garbage collection. Consider the following tools:

- IBM Pattern Modeling and Analysis Tool for Java Garbage Collector:

  http://www.alphaworks.ibm.com/tech/pmat

- Diagnostic Tool for Java Garbage Collector:

  http://www.alphaworks.ibm.com/tech/gcdiag?open&S_TACT=105AGX59&S_CMP=GR

3.2.3 Lazy instantiation

Lazy instantiation is a technique used to refrain from creating certain resources until the resource is first required. Typically, this comes about for instance variable initialization. Rather than initialize explicitly in the constructor (or class static initializer), it is left until access time for the variable to be initialized, using a test for null to determine if it has been initialized. Example 3-2 is a simple illustration of lazy instantiation.

Example 3-2 Lazy instantiation of an attribute.

```
public class LazyInitialization
{
    private AttrType attr = null;
    ...
    public AttrType getAttr()
    {
        if (attr == null)
        {
            attr = initializeAttr();
        }

        return attr;
    }
}
```

With lazy instantiation, the null tests are always performed upon accessing the variable even after it has been initialized. The overhead generally is small and can be ignored, however, you should use lazy instantiation only when there is a defined merit in the design, or when identifying a bottleneck which is alleviated using lazy instantiation. When implementing lazy instantiation, it is important to take into consideration multi threading issues and synchronization. Refer to 3.3, “Synchronization” on page 77 for further detail on synchronization considerations.

A significant benefit in application performance can be gained when you have objects with complex initialization that might never be used by creating the objects when first required. Applications avoid exercising code that might never be run. Delaying object creation also has benefits with load distribution, where there are many and/or complex objects that should be created and initialized but not used immediately. To prevent an application’s performance been impacted with large hits from object creations, it can be useful to spread out the load of object creation and initialization. On the other hand, there can be situations where creation and initialization of expensive objects can be performed on application startup, to reduce the performance hit during normal processing.

Designing with lazy instantiation in mind can result in over-engineering of applications prior to performance tuning. Even though you might want to introduce this into the application from the beginning, it is best used as a performance tuning technique. Designs could take into consideration some object attributes that might never used, however, the impact might be small and additional complexity not warranted. But this is quite an easy change to make when performance profiling the application identifies this as an issue, usually affecting just the accessor for that object attribute.

3.2.4 Object pools

If objects of the same class are being repeatedly created and destroyed, it can be beneficial to create an object pool that allows the objects to be reused. When the object creation cost is high or the pooled object represents a limited and costly resource, then pooling is a good practice. Classes whose objects are to be used in a pool require an initializer, so that objects obtained from the pool have some known initial state. It is also important to create a well-defined interface to the pool to allow control over how it is used.

However, the number of situations where applying object pooling is beneficial is fairly small. Object pooling has some serious downsides. Because the object pool is generally shared across all threads, allocation from the object pool can be a synchronization bottleneck. Pooling also forces you to manage deallocation explicitly, which reintroduces the risks of dangling pointers. The pool size must be properly tuned to get the desired performance result. If it is too small, it does not prevent allocation; and if it is too large, resources that could get reclaimed, instead sit idle in the pool. By tying up memory that could be reclaimed, the use of object pools places additional pressure on the garbage collector. Writing an effective pool implementation is not simple.

> **Note:** IBM WebSphere Application Server V6 provides object pools for pooling application defined objects or basic JDK types. This benefits an application which tries to squeeze every ounce of performance gain out of the system.

3.2.5 Thread local variables

In multi threaded applications, sometimes you require only one object per thread, with a thread consistently using the same object throughout its processing. Since Java 1.2, the ThreadLocal class was introduced to provide thread local variables, where you can associate a variable value to a thread. This could be useful, for instance, in a servlet associating a user to the thread assigned by the servlet container to process the request. If you place the user associated with a servlet request into a thread local variable, you can refer to that user downstream, without passing the details along as a parameter. The ThreadLocalUser class in Example 3-3 allows you to do this.

Example 3-3 Class containing user id that allows each thread to have its own instance

```
public class ThreadLocalUser
{
    private static ThreadLocal id = new ThreadLocal();

    public static void setId(String newValue)
    {
        id.set(newValue);
    }

    public static String getId()
    {
        return (String) id.get();
    }
}
```

Thread local variables differ from their object attribute counterpart in the way they are declared, initialized, and accessed. They are typically private static fields and use the set and get access methods on the ThreadLocal class to modify and retrieve its value.

An instance of the class containing the thread local variable is therefore not required, and a call to set the userId on the ThreadLocalUser class can be made as follows:

```
ThreadLocalUser.setId("TEL12345");
```

Once the ThreadLocalUser userId has been set to "TEL12345", we can retrieve its value by calling the access method as shown below:

```
String userId = ThreadLocalUser.getId();
```

To help illustrate how separate threads create their own instance of the ThreadLocalUser attribute id, the test class ThreadLocalTestServlet in Example 3-4 was written.

Example 3-4 Usage of ThreadLocal class

```
public class ThreadLocalTestServlet extends HttpServlet
{
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                 throws ServletException, IOException
  {
```

```
    Thread currThread = Thread.currentThread();
    String threadDtls = "Thread id["+currThread.getName()+"]";

    // Display user id before setting its value.
    System.out.println(threadDtls +
              ",User id["+ThreadLocalUser.getId()+"]");

    // Set user id value with that passed in from the request.
    System.out.println(threadDtls+",Set user id...");
    ThreadLocalUser.setId(req.getParameter("UserId"));
    System.out.println(threadDtls+
             ",User Id["+ThreadLocalUser.getId()+"]");

    // Sleep for 10 seconds
    // This will cause other threads run concurrently for that time.
    System.out.println(threadDtls+",Sleep of 10 seconds...");
    try
    {
      Thread.sleep(10 * 1000);
    } catch (InterruptedException e)
    {
      // Do nothing for interrupt exception.
    }

    // Log the user id value, after the pause in processing.
    System.out.println(threadDtls+
                        ",End processing for ["+
                        ThreadLocalUser.getId()+"]...");

  }
}
```

The ThreadLocalTestServlet, on receipt of an HTTP GET request, performs the following actions:

- It logs the for the thread's ThreadLocalUser attribute id.
- It sets the ThreadLocalUser attribute id value to that passed in the request.
- It sleeps for a period of 10 seconds to allow others threads to run concurrently and access their own ThreadLocalUser attribute id.
- Finally, it logs the current value of ThreadLocalUser attribute id.

As each thread has its own copy of the ThreadLocalUser attribute id, we would expect the value of id to be null when first logged to indicate it has not been set.

Once a value has been assigned, other thread operations on that attribute would not affect the thread's value.

The following calls to the ThreadLocalTestServlet were made in separate browser windows, within seconds of each other:

```
http://localhost/Examples/ThreadLocalTestServlet?UserId=Bob
http://localhost/Examples/ThreadLocalTestServlet?UserId=Larry
http://localhost/Examples/ThreadLocalTestServlet?UserId=Bill
```

The output from the three consecutive calls to the test servlet logged the following details:

```
Thread id[WebContainer : 1],User id[null]
Thread id[WebContainer : 1],Set user id...
Thread id[WebContainer : 1],User Id[Bob]
Thread id[WebContainer : 1],Sleep of 10 seconds...
Thread id[WebContainer : 0],User id[null]
Thread id[WebContainer : 0],Set user id...
Thread id[WebContainer : 0],User Id[Larry]
Thread id[WebContainer : 0],Sleep of 10 seconds...
Thread id[WebContainer : 2],User id[null]
Thread id[WebContainer : 2],Set user id...
Thread id[WebContainer : 2],User Id[Bill]
Thread id[WebContainer : 2],Sleep of 10 seconds...
Thread id[WebContainer : 1],End processing for [Bob]...
Thread id[WebContainer : 0],End processing for [Larry]...
Thread id[WebContainer : 2],End processing for [Bill]...
```

Each thread holds an implicit reference to its copy as long as the thread is alive and the thread local variable is accessible. All of its copies are subject to garbage collection upon thread completion and no further references to the thread local variable.

3.2.6 String concatenations

Although strings are a simple and efficient data structure in many languages such as C/C++, there is overhead associated with the use of strings (`java.lang.String`) in Java. Java strings are immutable; once created, their value cannot be changed. Hence operations such as string concatenation (+) involve the creation of new strings with the data copied from the original strings, creating more work for the garbage collector as well. When performing string manipulation operations, the use of `java.lang.StringBuffer` can improve performance. Using String and StringBuffer classes together is an optimization performed by many Java compilers today. Code is written like that shown in Example 3-5.

Example 3-5 Sample code

```
String b;
// ... value of b is set at some stage in the application.
String abc = “a” + b + “c”;
```

This is automatically optimized by Java compilers to the code shown in Example 3-6.

Example 3-6 Java optimized sample code

```
String b;
// ... value of b is set at some stage in the application.
String abc = new StringBuffer().append(“a”)
                               .append(b)
                               .append(“c”)
                               .toString();
```

However, there are instances when it is best to use a StringBuffer class on the onset, rather than allow this optimization to be performed by the Java compiler. The code snippet shown in Example 3-7 performs the string concatenation several times, within the loop block.

Example 3-7 Code snippet

```
String nbrs1To9 = "";
for (int i=1; i<10;i++)
{
    nbrs1To9 +=i;
}
```

The Java compiler would transform this to the code shown in Example 3-8.

Example 3-8 Java optimized code snippet

```
String nbrs1To9 = "";
for (int i=1; i<10;i++)
{
    nbrs1To9 = new StringBuffer().append(nbrs1To9)
                                 .append(i)
                                 .toString();
}
```

In each loop iteration, a new StringBuffer is created and then discarded once the String value is returned to the assigned variable. It would have been better to have written the code shown in Example 3-9 from the beginning.

Example 3-9 Efficient code

```
String nbrs1To9 = "";
StringBuffer buffer = new StringBuffer();
for (int i=1; i<10;i++)
{
    buffer.append(i);
}
nbrs1To9 = buffer.toString();
```

This code is more efficient, with the StringBuffer only created once and updated within the loop block.

3.2.7 Canonicalize objects

Canonicalizing objects is the replacing of copies of an object with a single or pool of copies. When a single copy is to be used by the entire application, the restriction on object instantiation can enforce by either using static methods on the class or implementing the Singleton pattern.

Static classes

Static classes have no object representation. All methods are defined as static, and the visibility of their constructors is restricted to prevent instantiation, as shown in Example 3-10.

Example 3-10 Static class implementation

```
public class ClassicStaticClass
{
    private ClassicStaticClass()
    {
        // Restrict the ability to instanciate this class.
    }

    public static String doSomething(String inputVal)
    {
        String returnVal = null;

        // Perform something on the input String and
        // assign it to the returnVal variable.

        return returnVal;
    }
}
```

Rather than invoking a method on an instance, you access the class method directly as shown here:

```
String resultSt = MyStatic.doSomething(strval);
```

Singleton objects

Singleton objects follow the Singleton pattern, described in "Singleton" on page 46. Singletons are classes with a private constructor's and often with a getInstance() method that returns an instance of the class, as shown in Example 3-11.

Example 3-11 Implementation of a singleton

```
public class ClassicSingleton
{
    // Contain a single instance of this class.
    private static ClassicSingleton sInstance = new ClassicSingleton();

    private ClassicSingleton()
    {
        // Restrict the ability to instanciate this class.
    }

    /*
     * Exists only to provide access to a single instance of this class.
     */
    public static ClassicSingleton getInstance()
    {
        return sInstance;
    }


    /*
     * Instance method(s)
     */
    public void doSomething(String inputVal)
    {
        // Perform something on the input String.
    }
}
```

When coding singletons, it is important to keep synchronization in mind. Refer to 3.3, "Synchronization" on page 77, for further details on this subject.

To invoke a method on a singleton, you have to obtain an instance to the class as shown here:

```
ClassicSingleton.getInstance().soSomething(strval);
```

Static classes versus singletons

When should one use a static class and when a singleton class? There is no one correct answer to this question. The main difference between the two is that static classes have no instance level variables, while singleton classes can and probably do. Static classes are best suited for utility type classes like a Math class, having no state and a collection of methods that convert a value and return another value based on what was passed in. If a class has to hold some data and only one instance of the class to be available, a singleton class would be the preferred option.

Singleton classes create an object of the class and use that throughout the application. At a later stage, if you require more instances of a singleton class, you can modify the method that returns a single instance of the class to return either an instance from a pool of objects or a new instance on every call. Static classes do not create an object, therefore methods are referenced through class name.

A limitation with static classes is that static methods are not polymorphic. They cannot take advantage of inheritance nor implement interfaces. A true singleton is a factory. It is configurable to load different implementations at runtime, just as the Runtime class differs for each operating system. Use static classes if the details do not change over time — a utility class for parsing strings is a good example. They are not impacted by a change in context, and additionally, they gain very little from polymorphism. If the code contains business logic that is likely to change over time, the singleton provides the flexibility.

3.2.8 Array copy

Use the System.arraycopy() method for copying the contents of one array into another, instead of an iterative loop. For example, the following two arrays are to be copied into a third array:

```
int array1[] = {0,1,2,3,4};
int array2[] = {5,6,7,8,9};
int array3[] = new int[array1.length+array2.length];
```

Using loops, the contents of array1 and array2 can be copied as shown in Example 3-12.

Example 3-12 Copy 2 arrays into a third array

```
for (int i = 0; i < array1.length; i++)
{
    array3[i] = array1[i];
}
for (int i = 0; i < array2.length; i++)
```

```
{
    array3[array1.length + i] = array2[i];
}
```

However, a more efficient approach is to use the System.arraycopy method as shown in Example 3-13.

Example 3-13 Efficient copy technique

```
System.arraycopy(array1, 0, array3, 0, array1.length);
System.arraycopy(array2, 0, array3, array1.length, array2.length);
```

3.2.9 Collection sizing

Although the Java runtime environment dynamically grows the size of collections such as `java.util.Vector` or `Java.util.Hashtable`, it is more efficient if they are appropriately sized when created. Each time the collection size is increased, its size is doubled so when the collection reaches a stable size, it is likely that its actual size can become significantly greater than required. It is better to presize the collection to its largest potential size to reduce, rather than having the runtime environment managing this. The collection only contains references to objects rather than the objects themselves, which minimizes the overallocation of memory due to this behavior.

3.2.10 Static and final variables

When a value is used repeatedly and is known at compile time, it should be declared with the static and final modifiers. This ensures that it gets substituted for the actual value by the compiler. If a value is used repeatedly but can be determined only at runtime, it can be declared as static and referenced elsewhere to ensure that only one object is created. Note the scope of static variables is limited to the JVM. Hence if the application is cloned, care should be taken to ensure that static variables used in this way are initialized to the same value in each JVM. A good way of achieving this is the use of a *singleton* object. For example, an EJB initial context can be cached with a singleton using the code fragment shown in Example 3-14.

Example 3-14 Use of the singleton pattern to cache EJB initial context references

```
public class EJBHelper
{
   private static javax.naming.InitialContext initialContext= null;

   public javax.naming.InitialContext getInitialContext()
```

```
    {
       if (initialContext == null)
       {
          initialContext = new javax.naming.InitialContext();

       return initialContext
    }
}
```

Use the final modifier on instance-variable definitions to create immutable internally accessible objects.

3.2.11 Object references

Although memory does not have to be explicitly deallocated, it is still possible to effectively have "memory leaks" due to references to objects being retained even though they are no longer required. They do not only degrade performance, but could eventually cause the application to terminate prematurely. These objects are commonly referred to as *loitering objects*. Memory leaks tend to be very subtle and difficult to debug, and tend to result from objects either referenced by variables that never go out of scope, or once placed into collection, are never removed.

Object references should be cleared once they are no longer required, rather than waiting for the reference to be implicitly removed when the variable is out of scope. This can be as simple as assigning the object reference to null when the object is no longer required, allowing it to be reclaimed sooner. Care should be taken with objects in a collection, particularly if the collection is being used as a type of cache. In this case, some criteria for removing objects from the cache is required to avoid the memory usage constantly growing. A complete understanding of the problem domain is required.

Another common source of memory leaks in Java is due to programmers not closing resources such as IO streams, Java Database Connectivity (JDBC) (further discussed in 3.5, "Database access" on page 85), Java Message Service (JMS) and Java Connector Architecture (JCA) resources when they are no longer required, particularly under error conditions. In Example 3-15, the loadProperties() method is not catered to handle any exceptions encountered once the file handle has been acquire. The exceptions are propagated to the caller, with the stream not closed until the garbage collector runs its finalizer.

Example 3-15 Incorrectly acquiring, using and releasing a resource

```
public Properties loadProperties(String inputFileName)
                        throws IOException
```

```
{
    FileInputStream fistream = new FileInputStream(inputFileName);
    Properties properties = new Properties();

    properties.load(fistream);
    fistream.close();

    return props;
}
```

In Example 3-16, the file handle, once obtained, is released prior to returning from the method call regardless of whether an exception was encountered or not.

Example 3-16 Correctly acquiring, using and releasing a resource

```
public Properties loadProperties(String inputFileName)
                       throws IOException
{
    Properties properties = null;
    FileInputStream fistream = null;

    try
    {
        fistream = new FileInputStream(inputFileName);
        properties = new Properties();
        properties.load(fistream);
    }
    finally
    {
        if (fistream != null)
        {
            fistream.close();
        }
    }

    return properties;
}
```

Note, however, that the code in the finally block can throw an exception, in this case an I/O Exception, which would supersede any exception raised in the try block. This means that the original exception is lost and masking the real reason why an error was encountered. To aid in the debugging effort, any exceptions raised in the final block should be handled, as shown in Example 3-17.

Example 3-17 Handling exceptions raised in finally block

```
public Properties loadProperties(String inputFileName)
                       throws IOException
{
  Properties properties = null;
  FileInputStream fistream = null;

  try
  {
    fistream = new FileInputStream(inputFileName);
    properties = new Properties();
    properties.load(fistream);
  }
  finally
  {
      if (fistream != null)
      {
        try
        {
          fistream.close();
        }
        catch(Exception e)
        {
          // Do nothing as this is only trying
          // to close the file stream.
        }
      }
  }

  return properties;
}
```

Normally, exceptions relating to closing a resource can be ignored within the finally block, and if one chooses, can log a message to note that this has occurred.

It is also important that static references be explicitly cleared when no longer required, because static fields never go out of scope. Since WebSphere Application Server applications typically run for a long time, even a small memory leak can cause the JVM to run out of free memory. An object that is referenced but no longer required might in turn refer to other objects, so that a single object reference can result in a large tree of objects which cannot be reclaimed.

The profiling tool available in IBM Rational Application Developer can help to identify memory leaks. Other tools that can be used for this purpose include Rational Purify®, Sitraka JProbe (by Quest Software), and Borland OptimizeIt.

3.2.12 Finalizers

Objects with finalizers (those that have a non-trivial finalize() method) have significant overhead compared to objects without finalizers, and should be used sparingly. Finalizeable objects are both slower to allocate and slower to collect. At allocation time, the JVM must register any finalizeable objects with the garbage collector, and (at least in the HotSpot JVM implementation) finalizeable objects must follow a slower allocation path than most other objects. Similarly, finalizeable objects are slower to collect. It takes at least two garbage collection cycles (in the best case) before a finalizeable object can be reclaimed, and the garbage collector has to do extra work to invoke the finalizer. When the garbage collector runs, it determines which are the unreachable objects and normal those objects would be collected. However, objects with finalizer must have its finalizer run before it can be collected, therefore no finalized garbage can be collected in the cycle that actually finds it.

Finalizers are not run at any particular time. The garbage collector cannot run finalizers itself when it finds them. This is because a finalizer might run an operation that takes a long time, and the garbage collector cannot risk locking out the application while this operation is running. Finalizers must be collected into a separate thread for processing and this adds more overhead into the garbage collection cycle.

The sequence in which finalized objects are located by the garbage collector has no relationship to the sequence in which they were created nor to the sequence in which their objects became garbage. The garbage collector has no knowledge of what is in a finalizer, or how many finalizers exist, it tries to satisfy an allocation without having to process finalizers. If a garbage collection cycle cannot produce enough normal garbage, it might decide to process finalized objects. So it is not even possible to predict when a finalizer is run, if at all.

Finalizers should be used as an emergency clear-up of, for example, hardware resources. If you must use finalizers, there are a few guidelines you can follow that can help contain to minimize their impact. Limit the number of finalizeable objects, which can minimize the number of objects that have to incur the allocation and collection costs of finalization. Organize your classes so that finalizeable objects hold no other data, which can minimize the amount of memory tied up in finalizeable objects after they become unreachable, as there can be a long delay before they are actually reclaimed. In particular, beware when extending finalizeable classes from standard libraries.

For tidying Java resources, think about the use of a clean up routine. When you have finished with an object, call the routine to null out all references, deregister listeners, clear out hash tables, and so on. This is far more efficient than using a finalizer and has the useful side-benefit of speeding up garbage collection. The Garbage Collector does not have so many object references to chase in the next garbage collection cycle.

3.3 Synchronization

The mechanism by which access to shared resources by different threads is controlled is called *Synchronization*. While the synchronization functionality in Java is convenient and easy to use, it can introduce significant performance overhead. When a block of code is synchronized, only a single thread can execute it at any one time. There are two performance impacts of synchronization:

- Managing the *monitors*, the objects internal to the JVM that are used to control access to synchronized resources. Although they are not explicitly accessed by programmers, there is an overhead due to the management of the monitors by the JVM.
- Reduced concurrency, since threads have to wait for other threads to exit from synchronized blocks of code.

Thus the use of synchronization should be minimized and limited to cases where it is definitely necessary. It is also good practice to clearly document all assumptions relating to synchronization, because they might not be obvious to someone reading the design or code.

3.3.1 Synchronized keyword

Defining methods as sychronized (using the synchronized keyword) is one of the most common approaches to prevent other threads from accessing the same functionality until the thread performing the action is finished. Example 3-18 illustrates the usage.

Example 3-18 Using the synchronized keyword to obtain a lock

```
import java.util.*;

public class Jury
{
    ArrayList members;
    String juryReferenceNbr;
```

```
    public Jury(String referenceNbr)
    {
        members = new ArrayList(12);
        juryReferenceNbr = referenceNbr;
    }

    public String getJuryReferenceNbr()
    {
        return juryReferenceNbr;
    }

    /**
     * Add member to jury list.
     */
    public synchronized void addMember(String name)
    {
        members.add(name);
    }

}
```

Here, two threads cannot invoke addMember() function at the same time; one must block while the other is working. However, any number of threads can simultaneously access the jury reference number via the getJuryReferenceNbr() function, because getJuryReferenceNbr() is not a synchronized function and hence is independent of the associated locking. Consider the impact of adding another function in the Jury class with the following implementation shown in Example 3-19.

Example 3-19 Adding addAlt() function to Jury class

```
import java.util.*;

public class Jury
{
    ArrayList members;
    ArrayList alternates;
    String juryReferenceNbr;

    public Jury(String referenceNbr)
    {
        members = new ArrayList(12);
        alternates = new ArralyList(12);
        juryReferenceNbr = referenceNbr;
```

```
    }

    ...

    /**
     * Add member to alternate jury list.
     */
    public synchronized void addAlt(String name)
    {
        alternates.add(name);
    }

}
```

The function addAlt(), when accessed, blocks not only other invocations of that function, but also the addMember() function in the same instance of Jury class, because both functions access the same lock. If the two functions must not block each other, then the programmer must use synchronization at a lower level.

3.3.2 Synchronized object lock

Synchronization can also occur for a block of code using objects as the locking mechanism. For instance, to ensure that addMember() and addAlt() functions do not block each other, the Jury class can be rewritten as shown in Example 3-20.

Example 3-20 Object locking within Jury class methods addMember() and addAlt()

```
import java.util.*;

public class Jury {
    ArrayList members;
    ArrayList alternates;
    String juryReferenceNbr;

    public Jury(String referenceNbr)
    {
        members = new ArrayList(12);
        alternates = new ArrayList(12);
        juryReferenceNbr = referenceNbr;
    }

    public String getJuryReferenceNbr()
    {
```

```
        return juryReferenceNbr;
    }

    /**
     * Add member to jury list.
     */
    public void addMember(String name)
    {
        synchronized(members)
        {
            members.add(name);
        }
    }

    /**
     * Add member to alternate jury list.
     */
    public void addAlt(String name)
    {
        synchronized(alternates)
        {
            alternates.add(name);
        }
    }
}
```

Synchronizing using the keyword can cause different methods to be unnecessarily synchronized with each other, and hence reduce concurrency. Note that synchronizing on an object has a greater overhead than calling a synchronized method. However, synchronizing the method can result in significantly greater amounts of code being synchronized, again reducing the concurrency. So the trade-off between the synchronization overhead and reduced concurrency has to be evaluated on a case-by-case basis.

3.3.3 Synchronized method decomposition

When using synchronization, it is best to use specific lock objects to synchronize on. You should analyze and ensure that only the minimum amount of execution time is spent within a synchronized block. To illustrate this, in Example 3-21 we add the method retrieveAll() to the class Jury to retrieve a list of jury participants.

Example 3-21 Using members and alternates as synchronized objects

```
public ArrayList retrieveAll()
{
    ArrayList retval;

    synchronized(members)
    {
        synchronized(alternates)
        {
            retval = new ArrayList(members);
            retval.addAll(alternates);
        }
    }
    return retval;
}
```

However, that would have been inefficient because we are obtaining locks on members and alternates well before they would be required. The rewrite in Example 3-22 is a better example because it holds the lock for the least amount of time and also obtains only one lock at a time.

Example 3-22 Rewrite with better lock management

```
public ArrayList all()
{
    ArrayList retval;
    synchronized(members)
    {
        retval = new ArrayList(members);
    }
    synchronized(alternates)
    {
        retval.addAll(alternates);
    }
    return retval;
}
```

As seen before, synchronized methods acquire a lock on the object. If the method is being called frequently by different threads, the method becomes the bottleneck by limiting the parallelism and hence limiting the efficiency. Thus, as a general philosophy, synchronized functions should be kept as small as possible. Notwithstanding this philosophy, there are times where a single function might have to do some tasks that require locking an object, while also doing other tasks that consume a lot of time. Example 3-23 shows low efficiency code where two function calls unsafe1() and unsafe2() required synchronization.

Example 3-23 Original low-efficiency code

```
public synchonized void doWork()
{
    unsafe1();
    write_file();
    unsafe2();
}
```

In such situations, a dynamic lock-release-lock-release method can be employed. The next example shows code that could be transformed in this manner.

Example 3-24 Rewritten high-efficiency code

```
public void doWork()
{
    synchonized(this)
    {
        unsafe1();
    }
    write_file();
    synchonized(this)
    {
        unsafe2();
    }
}
```

First and third functions require the object to be locked, while the more time-consuming write_file() function does not. As you can see, by rewriting the function, the lock to the object is released after the first function is finished and then reacquired when necessary for the third function. As a result, any other method that is waiting for a lock on this object can run while the write_file() function is executing. Decomposing a synchronized method into such hybrid code can greatly improve the performance. However, you should be careful that no logical errors are introduced in such a code.

You should pay special attention to methods that access slow resources that can degrade your program's efficiency like files, directories, network sockets, and databases. Try to put access to such resources in a separate thread, preferably outside any synchronized code.

3.3.4 Double-checked locking

Double-checked locking was a practice introduced to reduce the necessity for synchronization of a shared attributed, in particular after its initialization. A common implementation that is flawed is shown in Example 3-25.

Example 3-25 How not to perform double-check locking

```
public class Singleton
{
    private static Singleton instance = null;

    private Singleton(){};

    public static Singleton getInstance()
    {
        if (instance == null)
        {
            synchronized(Singleton.class)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
        }

        return instance;
    }
}
```

Unfortunately, in Java versions 1.4 and earlier, there were issues where double-checking locking does not work. This has been rectified with Java 5 with the usage of the volatile keyword. Example 3-26 shows how the variable instance is declared as volatile, to insure that multiple threads handle the class variable correctly when it is being initialized.

Example 3-26 Double-check locking in Java 5

```
public class Singleton
{
    private volatile static Singleton instance = null;

    private Singleton(){};

    public static Singleton getInstance()
```

```
    {
        if (instance == null)
        {
            synchronized(Singleton.class)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
        }

        return instance;
    }
}
```

Using object locks within the getInstance() method or synchronizing the method as discussed earlier are working alternatives independent of the version of Java. Synchronization can be alleviated when instance attributes can be initialized on class load up as shown in Example 3-11 on page 70.

3.3.5 Implicit synchronization

In addition to the explicit use of synchronization in application code, synchronization can be used indirectly, as some of the commonly used core Java functionality uses synchronization. Here are some particular examples:

- For Java I/O libraries, it is best to minimize the use of `System.out.println()` for this reason. Use of a multithreaded logging library as discussed in 3.4, "Logging" on page 84 is suggested.
- Some of the Java collection classes, such as `java.util.Hashtable` and `java.util.Vector`, are synchronized. If only a single thread is accessing the data (or multiple threads are reading only), the synchronization overhead is unnecessary. Many of the newer collections introduced in Java 1.2, such as `java.util.ArrayList` are not synchronized and can provide better performance. However, care should be taken when accessing them from multiple threads.

3.4 Logging

Java I/O classes use synchronization. Hence, `System.out.println()` should not be used for logging purposes. If a lot of output using stdout is generated by an application in a UNIX® environment, the overhead can be avoided by redirecting stdout to /dev/null in the production environment. However, a better approach is to use logging framework such as the Jakarta Commons Logging. Jakarta Commons Logging provides a simple logging interface and thin wrappers for several logging systems. The logging interface enables application logging to be simple and independent of the logging libraries that the application uses. In addition it allows logging statements to be defined at a particular level, which can be dynamically changed at runtime. Thus the amount of logging in production environments can be reduced in comparison to development and test environments without requiring code changes, improving the performance of the application.

WebSphere Application Server supports Jakarta Commons Logging by providing a logger, a thin wrapper for the WebSphere Application Server logging facility. WebSphere Application Server is pre-configured to use Jakarta Commons Logging with application logging calls routed by default to the underlying WebSphere Application Server logging facility.

When logging, it is also good practice to guard log statements so that the parameters are not evaluated if the logging level is not on. The use of guard statements is shown in Example 3-27.

Example 3-27 Use of guard statements for logging

```
if (Log.isLogging(Log.WARN)
{
   Log.log(LOG.WARN, “This is a warning”);
}
```

3.5 Database access

The Java Database Connectivity (JDBC) API provides a vendor-independent mechanism to access relational databases from Java. However, obtaining and closing a connection to a database can be a relatively expensive exercise, so the concept of *connection pools* has been introduced. When a database operation is to be performed, a connection can be obtained from the pool, which contains a defined number of connections to the database that have already been established. When the connection is closed, it is returned to the pool and made available for reuse. Using connection pooling can significantly reduce the overhead of obtaining a database connection. However, the connection pool is accessed via a data source. References to the data source are obtained by performing a lookup via the Java Naming and Directory Interface (JNDI). This lookup is an expensive operation, so it is good practice to perform the lookup once and cache the result for reuse.

JDBC resources should always be released once they are no longer required. This includes `java.sql.ResultSet`, `java.sql.Statement` and `java.sql.Connection` objects, which should be closed in that order. The code to close the resources should be placed in a final block to ensure that it is executed even when an exception condition occurs, as shown in Example 3-28.

Example 3-28 Correctly releasing JDBC resources

```
public List retrieveUsers() throws SQLException
{
  Statement statement = null;
  ResultSet resultSet = null;
  Connection connection = null;
  ArrayList userList = new ArrayList();

  try
  {
    connection = getConnection();
    statement = connection.createStatement();
    resultSet = statement.executeQuery("SELECT * FROM user");

    // Use resultSet to populate the userList.
  }
  finally
  { // Release JDBC resources.

    // Release result set.
    try
    {
      if (resultSet != null)
      {
          resultSet.close();
      }
    }
    catch(Exception e)
    {
       // Do nothing with exceptions raised
       // from releasing a resource.

       // You can choose to log a message to
       // indicate what exception was raised
       // from releasing the resource.
    }

    // Release statement.
```

```
    try
    {
      if (statement != null)
      {
        statement.close();
      }
    }
    catch(Exception e)
    {
    }


    // Release connection.
    try
    {
      if (connection != null)
      {
        connection.close();
      }
    }
    catch(Exception e)
    {
    }

  } // Release JDBC resources.

  return userList;
}

private Connection getConnection()
{
  // Returns a database connection.
}
```

Failure to properly close resources can cause memory leaks, as well as slow response due to threads having to wait for a connection to become available from the pool. Since database connections in the pool are a limited resource, they should be returned to the pool once they are no longer required.

If an application repeatedly executes the same query, but with different input parameters, then performance can be improved by using a `java.sql.PreparedStatement` instead of `java.sql.Statement`. Turning off auto commit for read only operations can also increase performance.

To avoid having to retrieve and process large amounts of data, sometimes it is beneficial to use database stored procedures for implementing some of the application logic. Alternatively, in some cases, calls to the database can be minimized by using a single statement that returns multiple result sets.

There are different types of JDBC drivers available, some written in pure Java and others that are native. Although use of a native driver can reduce portability, performance can be better with a native driver.

3.6 Event-driven processing

When an application has to react to events or conditions (both internal or external), there are two methods of designing the system. In the first method, known as polling, the system periodically ascertains the status and reacts accordingly. This method, while simple, is also less efficient because you cannot always predict when it might have to be invoked.

The second method, known as event-driven processing, is more efficient but also more complex to implement. In the case of event-driven processing, you require a signaling mechanism to control when a particular thread should run. In Java programs, you can use the wait(), notify(), and notifyAll() functions inherited from the Object class to signal a thread. These functions allow threads to block on an object until such time as the desired condition is met and then to start running again. This design reduces CPU usage because threads do not consume execution time while blocked and can instantly awaken when notify() methods are called. The event-driven method provides better response time compared to that of the polling method.

3.7 Exceptions

The creation of exception objects is an expensive process that degrades performance. When an exception is created, it has to gather a stack trace describing where it was created. Building those stack traces requires taking a snapshot of the runtime stack, and that is the expensive part. The stack is recorded all the way from main or Thread.run (at the bottom of the stack) right up to the top. The stack trace snapshot happens in the Throwable constructor through a native method call to fillInStackTrace(). This method is responsible for walking the stack frame to collect trace information. The catching and throwing of exceptions is not the expensive part.

Good programming practice dictates that exceptions should be used for error conditions only, and not control flow.

3.8 New input/output library

The new input/output (NIO) library was introduced in Java 1.4 as a supplement to the existing java.io package. It introduced an alternative approach of reading and writing data, from the existing streaming model. New capabilities such as file blocking and asynchronous I/O were also introduced. In the following sections, we describe these features in the NIO library in more detail.

3.8.1 Buffered read and writes using channels

Most of the programs work with external data stored either in local files or coming from other computers on the network. Java has a concept of working with streams of data. After a physical data storage is mapped to a logical stream, a Java program reads data from this stream serially — byte after byte, character after character, and so on. Some of the types of streams are byte streams (InputStream, OutputStream) and character streams (Reader and Writer). The same physical file could be read using different types of streams, for example, FileInputStream, or FileReader. The classes that work with streams are located in the java.io package. Java 1.4 introduced the new package java.nio with improved performance. With the NIO library, all I/O is dealt in blocks of data. Each operation produces or consumes a block of data in one step.

All data is handled with buffers. Data is directly read from or written to a buffer. As buffers represent the data container, a Channel is required in which you can read from and write data to. Channels are the stream equivalent in the I/O library. Unlike streams, however, they can be bidirectional, opened for read, for writing, or both. To read a file, you have to get the Channel from the FileInputStream, create a Buffer and read from the Channel into the Buffer. Writing to a file is a similar process to the read. You get the Channel from the FileOutputStream, create a Buffer and populate it with the data we wish to write, and finally write from the Buffer into the Channel. From the foregoing read and write operations, notice that you never directly read and write to the Channel, only via the Buffer.

3.8.2 File locking

File locking is notoriously specific to the operating system and even the file system, and not done by all in a consistent manner. Some implementations provide only exclusive locks, while other implementations provide for shared locks. With NIO, file locks are built right into the FileChannel class. Prior to Java 1.4, apart from native method calls, there was another way to check or set for file locks.

The file locks are regular Java objects of type FileLock that allow different parts of the system to coordinate data access. These locks do not prevent data access. Example 3-29 illustrates the usage of file locks.

Example 3-29 Restricting access using file locking

```
RandomAccessFile raf = new RandomAccessFile("sharedfile.txt","rw");

// Obtain exclusive access to file, to carry out sensitive operations.
FileChannel fc = raf.getChannel();
fc.lock();

// Do file operations...

// Now that all was needed to be done to the file was completed,
// we can release the lock.
lock.release();
```

3.8.3 Asynchronous I/O

With traditional Java I/O model, when an application thread reads or writes data, it blocks until the operation completes. NIO introduced Asynchronous I/O calls, where the application thread does not have to block for read or write operations. This is only applicable for socket channels (SocketChannel, ServerSocketChannel, and DatagramChannel). FileChannel cannot be placed in non-blocking mode.

With Asynchronous I/O, rather than blocking, it registers your interest in particular I/O events such as the arrival of new data, and the system tells you when such an event occurs. This allows you to do a number of inputs or outputs without having to create new threads. You can listen for I/O events on an arbitrary number of channels without having to poll. When a channel is non-blocking, read() or write() calls always return immediately, whether they transferred any data or not. This enables a thread to check if data is available without getting stuck. As shown in Example 3-30, if there is no data to be read from the Channel, the read() call returns zero, and the thread bypasses the processing of the input buffer and continues its processing.

Example 3-30 Using non-blocking I/O to read from a file

```
// Bind to a port expecting to receieve data.
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking (false);
ssc.socket().bind(new InetSocketAddress(portNbr));
```

```
// Create an Input Buffer to store the data read from the channel.
ByteBuffer ib = ByteBuffer.allocate (1024);

// Read data from channel.
if (ssc.read (ib) != 0)
{
  // Process data read from the channel.
  processInputBuffer(ib);
}
```

Polling to determine when input is ready on a non-blocking channel introduces the practice of periodically checking for input, rather than being notified when there is data to be read. If the application has to respond to inputs arriving on a connection, then the conventional blocking model might seem a better option. However, as the number of connections serviced increases, so does the number of threads required, as each is listening on a single connection. A single thread is unable to manage multiple I/O channels. But polling quickly on non-blocking Channels consumes many CPU cycles and many unproductive I/O requests. I/O requests are expensive, as they generate system calls that in turn entail context switches.

The NIO class Selector aids a single thread in managing many I/O channels. It provides an alternation to polling or thread-per-channel/stream. You create a Selector instance, register one or more non-blocking channels with it, indicating for each, what events are of interest. The call on the select() method blocks until at least one of the registered events occurs, followed by the processing of the event. This processing is normally contained within a loop, as depicted in Example 3-31.

Example 3-31 Managing many I/O channels with a single thread

```
// Maintain list so we can close sockets later.
ServerSocketChannel sscList[] = new
ServerSocketChannel[portNbrList.length];

Selector selector = Selector.open();

// Bind to each port expecting to received incoming data.
for (int i=0; i < portNbrList.length ; i++)
{
  ServerSocketChannel ssc = ServerSocketChannel.open();
  ssc.configureBlocking (false);
  ssc.socket().bind(new InetSocketAddress(portNbrList[i]));

  // Configure socket channel for accepting new connections.
```

```
      ssc.register(selector, SelectionKey.OP_ACCEPT);

      sscList[i]=ssc;
    }

    while (true)
    {
      // Wait until an event occurs.
      selector.select();

      // We will no go through the list of raised events and process
      // each one individually.
      Iterator it = selector.selectedKeys().iterator();

      while (it.hasNext())
      {
        SelectionKey key = (SelectionKey) it.next();
        ServerSocketChannel ssc = null;
        SocketChannel sc = null;;

        // If event is an incoming connection waiting to a server socket,
        // accept it and configure the socket channel for reading rather
        // than accepting new connections.
        if (key.isAcceptable())
        {
          ssc = (ServerSocketChannel) key.channel();
          sc = ssc.accept();

          sc.configureBlocking (false);
          sc.register (selector, SelectionKey.OP_READ);
        }

       // If the event is triggered from incoming data, read the data from
       // the channel and process it accordingly.
       if (key.isReadable())
       {
         sc = (SocketChannel)key.channel();

         // Create an Input Buffer to store the data read from the channel.
         ByteBuffer ib = ByteBuffer.allocate (1024);

         // Read data from channel.
         if (sc.read(ib) != 0)
         {
           // Process data read from the channel.
```

```
            processInputBuffer(ib);
        }

    }

    // Remove key so we don't process again it next time.
    it.remove();
}
```

This example is a bit simplistic, and does not deal with removing closed channels from the Selector. It demonstrates how much simpler and more scalable implementations can be in using Selectors compared to the thread-per-socket model. The key point is that a Selector object does the hard work of checking the state of a potentially large number of channels. You just act on the result of selection, and do not have to check each one yourself.

The Async IO package is designed to provide fast and scalable input/output (IO) for Java applications using sockets and files. It provides an alternative to the original synchronous IO classes available in the java.io and java.net packages, where scalability is limited by the inherent “one thread per IO object” design.

3.9 Java 5 features

Java 5 encompasses 15 component Java Specification Requests (JSRs) and nearly 100 other significant updates. This section covers only major changes that ease development and improve performance and scalability. A comprehensive list of changes is given in the Sun documentation site at the following URL:

http://java.sun.com/j2se/1.5.0/docs/index.html

3.9.1 Ease of development

The Java 5 virtual machine specification adds several features and functions to benefit application developers to make development quicker, easier, and less error prone. Features such as enhanced for loops, enumerated types, static imports, C style formatted input/output, variable arguments, concurrency utilities, generics, auto-boxing of primitives, and annotations are covered in the following sections.

Generics

Generic types in Java provide the ability during compile time to detect inappropriate downcasting of objects. A common practice in Java applications is to downcast expressions to datatypes of a more specific type. The most common reason to downcast in the Java language is that classes are often used in specialized ways that restrict the potential runtime types of arguments returned by method calls. For example, suppose we are adding and retrieving elements to an ArrayList. The element types stored in an ArrayList vary for each program. As such, the interface to add or retrieve an element is of type Object. This provides the maximum flexibility, however, when retrieving elements from the ArrayList, it requires to downcast the return type of Object to a more appropriate type. Prior to generics, the way we access collections such as ArrayLists is shown in Example 3-32.

Example 3-32 Accessing collections prior to Java 5

```
ArrayList list = new ArrayList();
list.add(0, new Integer(1));
Integer listEntry = (Integer) list.get(0);
```

Every downcast in a program can potentially raise ClassCastException, however, applying generic types help to reduces this substantially. Using generified Collections, Example 3-33 shows how we now can access elements in the list.

Example 3-33 Accessing generified collections in Java 5

```
ArrayList<Integer> list =  new ArrayList<Integer>();
list.add(0, new Integer(1));
Integer total = list.get(0);
```

Generics in the Java language are implemented almost entirely in the compiler, which performs type checking and type inference, and then generates ordinary, non-generic bytecodes.

Further details can be obtained from the DeveloperWorks article:

http://www.ibm.com/developerworks/java/library/j-genjava.html?n-j-3151

Enhanced loops

Traversing all entries with a Collection usually involves the usage of an Iterator. Iterators provide sequential access through a Collection. Example 3-34 illustrates the usage of an Iterator to get all of the elements in an ArrayList.

Example 3-34 Accessing a Collection prior to Java 5

```
ArrayList list = new ArrayList();

// populate list with Integer objects ...
```

```
for (Iterator i = list.iterator(); i.hasNext();)
{
    Integer listEntry=(Integer)i.next();

    // do something with listEntry ...
}
```

With the addition of generics, this is made a little bit easier, as shown in Example 3-35.

Example 3-35 Using generics with Collection Iterators

```
for (Iterator<Interger> i = list.iterator(); i.hasNext();)
{
    Integer listEntry=i.next();

    // do something with listEntry ...
}
```

The enhanced loop can replace the iterator when traversing through a Collection, as shown in Example 3-36.

Example 3-36 Enhanced loop for traversing Collections

```
for (Integer listEntry : list)
{
    // do something with listEntry ...
}
```

The compiler generates the looping code necessary, and with generic types, no additional casting is required. The same can also be applied to array types. In Example 3-37 we access an array of Integers using the traditional means.

Example 3-37 Traversing and array prior to Java 5

```
int list = new int[20];

// populate list ...

for (int i = 0; i < list.length; i++)
{
    int listEntry = array[i]);

    // do something with listEntry ...
}
```

Using enhanced loops simplifies traversing the array as shown in Example 3-38.

Example 3-38 Enhanced loop for traversing arrays

```
for (int listEntry : list)
{
    // do something with listEntry ...
}
```

Enumerated types

Enumerated types in previously releases were represented as static final constants, as shown in Example 3-39.

Example 3-39 Enumerations as simple static final types

```
public class Status
{
    public static final int SUCCESS = 0;
    public static final int FAILURE = 1;
    public static final int RETRY = 2;
}
```

There are problems with implementing enumerates like this, such as these:

- They are not type safe, so you can pass in any other int value where Status is required.
- As int types, when you print out a Status type, only the int value is displayed. This does not inform you of what it actually represents.

A workaround to these problems is to expand on the constant class Status, as shown in Example 3-40.

Example 3-40 Enumerations as custom static final types

```
public class Status
{
    private String descr;

    public static final Status SUCCESS = new Status("SUCCESS");
    public static final Status FAILURE = new Status("FAILURE");
    public static final status RETRY   = new Status("RETRY");

    private Status(String statusDescr)
    {
        description = statusDescr;
    }
```

```
    public toString()
    {
        return super.toString() + "[" + descr + "]";
    }
}
```

With the introduction of enumerated types, this now becomes more simplified, resembling their C and C++ counterparts. In Example 3-41, the Status type is now defined using enumerated types.

Example 3-41 Enumerations as defined in Java 5

```
public enum Status { SUCCESS, FAILURE, RETRY };
```

With enumerated types, we have new implementations of Set and Maps called EnumSet and EnumMap.

EnumSet

EnumSet is a high-performance Set for enums. It is similar to BitSet in that it implements a vector of bits, each entry as a single long line. Example 3-42 shows EnumSet operations that provide additional value.

Example 3-42 EnumSet valued added operations

```
import java.util.EnumSet;

enum MyEnumSet
{
    E0,
    E1
    E2,
    E3,
    E4,
    E5,
    E6,
    E7,
    E8,
    E9
};

public class EnumSetExample
{

    // test method
```

```
    public static void main ( String[] args )
    {
        EnumSet<MyEnumSet> setA = EnumSet.<MyEnumSet> of (E1,E4,E5);

        // set A = E1, E4, E5

        EnumSet<MyEnumSet> setB = EnumSet.complementOf(setA);

        // set B = E2, E3, E6, E7, E8, E9

        EnumSet<MyEnumSet> setC = EnumSet.range(MyEnum.E6, MyEnum.E9);

        // set C = E6, E7, E8, E9

    }
}
```

EnumMap

Similarly, EnumMap is a high-performance Map for enum keys. All enum keys must be of the same enum type. As it is optimized to only contain enum keys, performance is gained, as it does not have to call hashCode() to find its location within the map. Internally the map is implemented as an array and the keys maintained in the order which the enum constants are declared. You should always use EnumMap to map an enum to a value.

Static imports

Many classes use constants defined within other classes. When accessing a constant in another class, the class names have to be prefixed before the constant, as shown in Example 3-43.

Example 3-43 Prefixing constants with class name

```
import java.lang.Math;

public class SimpleMathTest
{
  public static void main(String args[])
  {
      double newValue = Math.PI + 1;
  }
}
```

To reference the PI constant in the Math class, we have to call Math.PI. If the constant referenced was contained within a class that is not a part of the Java

SDK or J2EE libraries, you could potentially extend that class and call the constant directly without the class name as the prefix. Java SDK and J2EE libraries tend to have the final modifier to prevent classes from extending them.

With static import, you no longer have to prefix constants with the class name or extend the class with the constant. All that has to be done is to declare a static import of the class with the constant and then use the constant name freely with your code, as shown in Example 3-44.

Example 3-44 Using import static statements for constants

```
import static java.lang.Math.PI;

public class SimpleMathTest
{
  public static void main(String args[])
  {
      double newValue = PI + 1;
  }
}
```

You can also use wildcards with the import static statement

```
import static java.lang.Math.*;
```

However, if constants from another class are extensively used, it is best to include the constant to make the code easier to read.

Variable arguments

In previous releases, arrays or Lists were used by methods that accepted an arbitrary number of arguments. With the introduction of variable arguments (also commonly referred to as varags), methods can have an unspecified number of arguments without explicitly using arrays or Lists, similar to C and C++. This can be achieved by using an elipse (...) within the method signature, as shown in Example 3-45.

Example 3-45 Method with variable argument lists

```
public void foo(int lvalue, String... svalues)
{
    for (String str : svalues)
    {
        // do something with str
    }
}
```

Parameters that a method requires to be explicitly passed in, must precede the variable arguments. Even though not specified in the method signature, the multiple arguments passed are treated as a array of the specified type by the method. Example 3-46 shows that you can directly assign the variable argument to an array of that type within the method.

Example 3-46 Varags hidden array implementation

```
public void foobar(String... svalues)
{
    String sarray = svalues;
    // do something sarray
}
```

Thus you can pass in either multiple arguments, comma separated, or an array of that type, as shown in Example 3-47.

Example 3-47 Invocation of methods with varargs

```
foo(1, “Hello”);
foo(2, “Hello”, “World”);
String sarray = new String[4];
// populate sarray with String values
foo(10, sarray);
```

This is upward compatible with preexisting APIs such as the String and System classes. In Example 3-48, we show how varags simplifies the printing of strings using the System class.

Example 3-48 Comparing System.out.prin method invocation without and with varags

```
System.out.println(“Your password for “ + username + “ will expire in “
+ daysexpr + “ days.”);
System.out.printf(“Your password for %s will expire in %d days %n.”,
username, daysexpr);
```

Using the printf() method makes the string to be printed more readable. For those familiar to C and C++, you can use ‘\n’ for the newline character, however, we recommend that you use the Java ‘%n’ for cross-platform support.

Concurrency utilities

Included in Java 5 are new packages that are to be used as building blocks in the development of concurrent applications. They aim to simplify development by providing a high-performance, scalable, and thread-safe framework of classes that would commonly be used in developing a concurrent software solution.

Having common utilities across applications makes for easier maintenance, faster development, and encourages commonality between applications. The utilities provided in the java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks packages cover thread pools, concurrent collections, semaphores, latches, and barriers.

Primitive auto-boxing and auto-unboxing

A common problem with using Collections is the inability to handle primitives. Collections can only store object references, so if we have to store a primitive such as int, we first must convert it into an equivalent object representation such as Integer and then store the Integer. This is known as *boxing* a primitive with an appropriate wrapper class. On retrieval of the object from the Collection, if necessary to convert it back to a primitive value, we have to call the intValue method on the Integer object. This process of extracting the object's primitive value is known as *unboxing*. This is illustrated in Example 3-49 using generics.

Example 3-49 Boxing and unboxing primitives for Collection storage

```
int ioriginal = 15;
ArrayList<Integer> list = new ArrayList<Integer>(10);
list.add(new Integer(ioriginal));
Integer iobj = list.get(0);
int iprim = iobj.intValue();
```

The Java 5 feature of auto-boxing alleviates the conversion of primitives to objects for storage in Collections and likewise auto-unboxing with the conversion from objects back to primitives. In Example 3-50 the primitive is passed in and retrieved from the Collection.

Example 3-50 Auto-boxing and auto-unboxing primitives for Collection storage

```
int ioriginal = 15;
ArrayList<Integer> list = new ArrayList<Integer>(10);
list.add(ioriginal);
int iprim = list.get(0);
```

Meta data

Meta data was introduced to reduce effort in development and deployment by providing a common infrastructure for development and runtime tools. Additional data know as annotations can be added to your code with the intent to create documentation, track down dependencies in code, and perform rudimentary compile-time checking. By allowing information to be maintained in the source file, meta data also eliminates the necessity for "side files" and the maintenance of them when changes occur in the source files. Annotations can be applied to package declarations, type declarations, constructors, methods, fields, parameters, and variables. Annotations take the form of an “at” sign (@), followed by the annotation name. You can supply data to an annotation when it is required — in name=value pairs.

Prior to release Java 5, Javadoc was the closest to a meta data facility. With Javadoc, you mark up your code with a special set of tags and then execute the **javadoc** command to turn the tags into a formatted HTML page that documents the classes the tags are attached to. You can use standard annotations offered by Java 5, such as Deprecate, SuppressWarnings and Override, and also create your own annotation types. To illustrate a simple usage of annotations, Example 3-51 shows how the Overide annotation is used to indicate to the compiler that the method marked should override a superclass method.

Example 3-51 Using the Override annotation

```
public class OverrideExample
{
    @Override
    public String toString()
    {
        String overRideStr= super.toString() + "[OverrideExample]";
        return overRideStr;
    }
}
```

When methods marked with the Override annotation do not override superclass methods as shown in Example 3-52, there is no toStrin() method in the superclass Object, so you get a compilation error for that method.

Example 3-52 Applying the Override annotation to a method that does not override

```
public class OverrideExample
{
    @Override
    public String toStrin()
    {
        // do stuff here
    }
}
```

For a more comprehensive explanation of annotations, refer to the developer works article shown here:

http://www.ibm.com/developerworks/java/library/j-annotate1/

3.9.2 Garbage collection

Considerable effort has been made in Java 5 to improve performance and scalability. Applications that use threading heavily and run on multiprocessor hardware with large amounts of memory might have to customize the behavior of the default collector or have to use an alternative collector. In the release of Java 5, the default settings for the garbage collector, heap size, and runtime compiler have be optimized for a number of platforms. Existing applications that have tuned these settings should re-evaluate their JVM command line arguments to use where possible the new default values. Java 5 also has simplified the tuning of the garbage collector with the introduction of two settings:

- Maximum pause time goal: The maximum pause time in milliseconds or less that the garbage collector can suspend the application to reclaim unused memory
- Application throughput goal: A ratio of the time spent by the garbage collector compared to the time spend by the application for processing

These goals are an indication to the JVM of the desirable values. The JVM adjusts the collection parameters accordingly to ensure that they can be either lower for pause time or greater for throughput. For instance, if the maximum pause time was 60 milliseconds, it ensures that any application pause caused by the garbage collector would be less than 60 milliseconds. If this was not being achieved, the collector would automatically configure the JVM parameters to do so. However, the heap must be sized to at least accommodate all of the objects that are in use by the application, otherwise the goals might not be met.

In previous releases, the configuration of generation sizes within the heap, the survivor spaces sizes, and when objects are to be promoted from the young generation to the tenured generation, had to be configured explicitly.

For further details, refer to the Sun reference documentation on Ergonomics in the 5.0 Java™ Virtual Machine; see the following URL.

http://java.sun.com/docs/hotspot/gc5.0/ergo5.html

3.10 General coding best practices

The best practices given here are mostly intended for coding design. Chapter 4, “Presentation and control layer” on page 107 contains more detail about Java coding best practices.

3.10.1 Do not put business logic in your client

Be careful not to put business logic in your Java client. As business logic tends to have more changes, it becomes difficult to deploy these changes to clients. Use patterns such as Business Delegate to separate view from business calls. Also use value objects to communicate the view with control that is a server-side component. Remember — put as little code as possible in the client.

3.10.2 Always clean up after yourself

If you obtain an object from a pool, always make sure you return it back to the pool.

One of the most common errors we see with Java EE applications, whether running in development, test, or production, are memory leaks. Nine times out of ten, it is because a developer forgot to close a connection (JDBC most of the time) or return an object back into the pool. Make sure that any objects that should be explicitly closed or returned to the pool are so done. Do not be one of the culprits responsible for the offending code.

3.10.3 Plan for version updates

Change is inevitable. Plan for new releases and fix updates so that your customers can stay current.

WebSphere Application Server continues to evolve, and so it should be no surprise that IBM regularly produces fixes for WebSphere Application Server, and that IBM periodically releases new major versions. You have to plan for this. There are two kinds of development organizations that this impacts: in-house developers and third party application vendors. The basic issues are the same, but each is impacted differently.

First, consider fixes. IBM regularly releases recommended updates that fix known bugs in our products. While it is likely impossible to always be running at the latest levels, it is prudent to not fall too far behind. How “far behind” is it okay to be? There is no right answer to this, but you should plan on supporting fix levels within a few months of their release.

Yes, this means upgrades in production a few times a year. In-house developers can feel free to skip certain fix levels and support one fix level at a time to reduce testing costs. Application vendors are not so lucky. If this is your case, then you should support multiple fix levels at the same time so that your customers can run your software in conjunction with other software. If you support only one fix level, it might quite literally be impossible to find fix levels compatible across multiple products. Really, the best approach for vendors is to go with the model of supporting "upwardly compatible fixes." This is the approach IBM uses with regard to support products of other vendors with which we integrate (such as Oracle®, Solaris™, and so on). Refer to our support policy for more information.

Second, consider major version upgrades. Periodically, IBM releases new major releases of our products with major functional upgrades. We continue to support older major releases, but not forever. This means that you must plan for forced moves from one major release to another. This is simply unavoidable and must be considered in your cost model. If you are a vendor, this means you have to upgrade your product to support new versions of WebSphere Application Server from time to time, or your customers could be stranded on unsupported IBM products — which is something we have seen happen more than once! If you are purchasing a product from a vendor, we encourage you to ensure through due diligence that your vendor is committed to supporting new versions of IBM products. Being stranded on unsupported software is a very dangerous situation.

3.10.4 Follow rigorous procedures for development and testing

This includes adopting and following a software development methodology.

Large scale system development is difficult and it should be taken seriously. Yet, too many times we find teams that are lax in their policies, or that casually follow development methods which might not apply for the type of development that they are doing, or that they do not understand well. Perhaps the worst extreme of this is trying on the "Development method of the month" where a team swings from RUP to XP to some other agile method within the life cycle of a single project.

In short, almost any method can work for most teams, provided that the methods are well-understood by the team members, followed rigorously, and adjusted carefully to deal with the specific natures of the technology and team that is using that method.

4

Presentation and control layer

This chapter describes programming considerations that apply to the application presentation and control layer. It focuses on some basic technology best practices. Also covered are application development frameworks such as Struts and JavaServer Faces (JSF). The intent is not only to use the newer technologies when developing new applications, but also to promote good programming practices when maintaining and modifying existing projects and applications for better performance in WebSphere Application Server.

For the newer application development technologies, we focus on basic knowledge and coding practices instead of delving into more advanced features. The chapter is organized into the following major sections:

4.1 Presentation layer

The presentation layer is responsible for rendering the *view*, meaning the graphical representation of the user interface. In the J2EE point of view, client-side access to an application in most cases occurs through a browser. It is also possible to access J2EE applications on the server through a “fat” client, for example, a Java client-side application, a CORBA client-side application, a WebSphere MQ client, or a Web Services client.

In this chapter, we focus on two access types: Web client and Java client-side access. These two types are the most used today. We discuss several different approaches for generating the dialog that describes the user interface, focusing on performance aspects and coding tips. In addition, we cover basic concepts to more easily understand the application coding considerations we are describing. When required, we explain when to use or not use some application development features to gain a performance benefit.

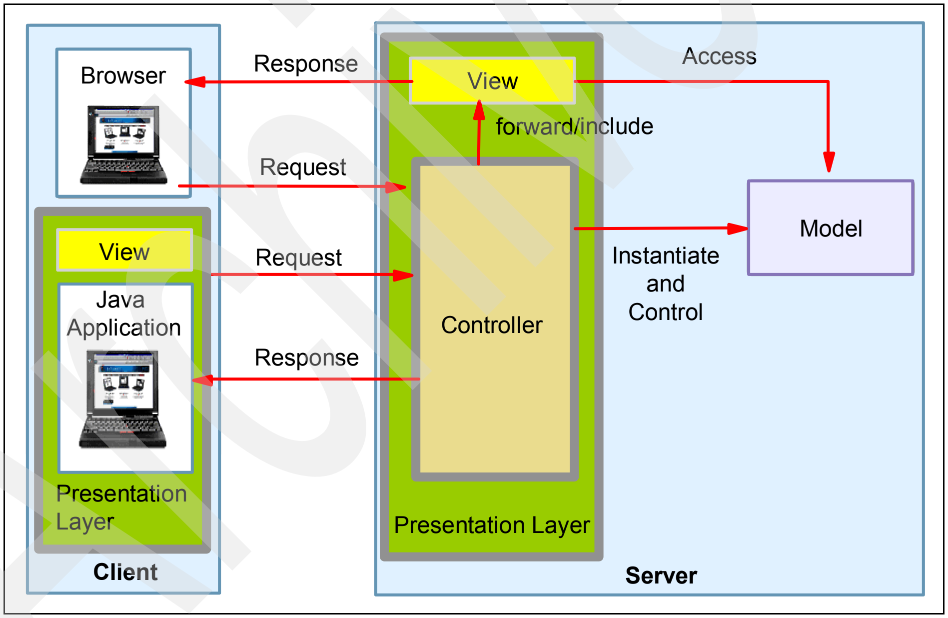


Figure 4-1 Presentation layer

4.2 JavaServer Pages

JavaServer Pages (JSP) technology generates HTML or Extensible Markup Language (XML) output of a Web server dynamically as a response to a Web client request. JSP technology is one of the most powerful, easy-to-use, and fundamental tools in a Web-site developer's toolbox. It combines HTML and XML with Java servlets (server application extensions) and JavaBeans technologies to create a highly productive environment for developing and deploying reliable, interactive, high-performance platform-independent Web sites.

JSPs are translated into Java servlets by a JSP compiler in a Web container. These servlets are then executed in the Web container. JSPs should be used for presentation, while servlets should be used for the control function. The various components involved for presentation are shown in Figure 4-2.

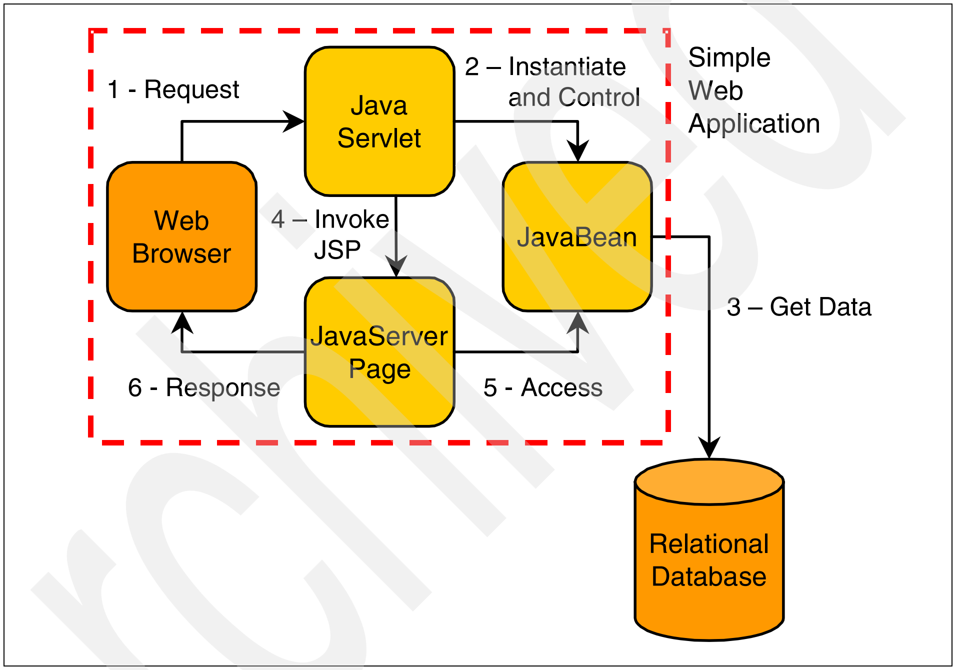


Figure 4-2 Components in a simple Web application

It is a common mistake to mix the purpose of each component. To make your code reusable and easy to maintain, use the Model View Controller (MVC) pattern to avoid accessing the database layer directly from a JSP without a JavaBean component that represents the Model layer of the application. See 2.2.2, “Model View Controller” on page 28 for more information on the Model View Controller pattern. The JavaBeans components used to access databases can be reused in another application. Consider a case when you require a new application that accesses the same database shown in Figure 4-2 for maintenance information purposes. JavaBeans allow the model to be reusable for this new application.

JSP Java code can be placed in Tag Libraries to separate page design, such as HTML design tags, from Java code. Refer to 4.2.9, “Tag libraries” on page 125 for more information.

4.2.1 Use JSPs as your first choice of presentation technology

The most basic reason to select JSPs as your presentation technology is because it is the best supported and best understood Java EE view technology available. Also, JSPs usually offer the best performance of all the technologies discussed in this chapter. Therefore, JSP based frameworks should be considered first when building high-volume Web sites. Simple Web applications built on JSPs and servlets are faster than more complex frameworks such as Struts or JavaServer Faces, which we cover later. However, Struts and JavaServer Faces have other significant advantages; for example, they simplify good application design and are easier to maintain.

Given the introduction of custom tag libraries, the JSP Standard Tag Library (JSTL), and the JSP 2.0 features, it is becoming increasingly easy to build JSPs that do not require any Java code, and that cleanly separate application code and presentation view. It allows separating development between page designer and Java developer. There is significant support (including debugging support) for JSPs built into development environments, for example, IBM Rational Application Developer, and many developers find developing with JSPs easier than developing with XSL — mainly due to how JSP is procedurally based, as opposed to rules-based.

4.2.2 JSP processor phases

Before we get into performance tips for each core syntax JSP element, it is important to understand how a JSP processor works at a high level. Refer to Figure 4-3.

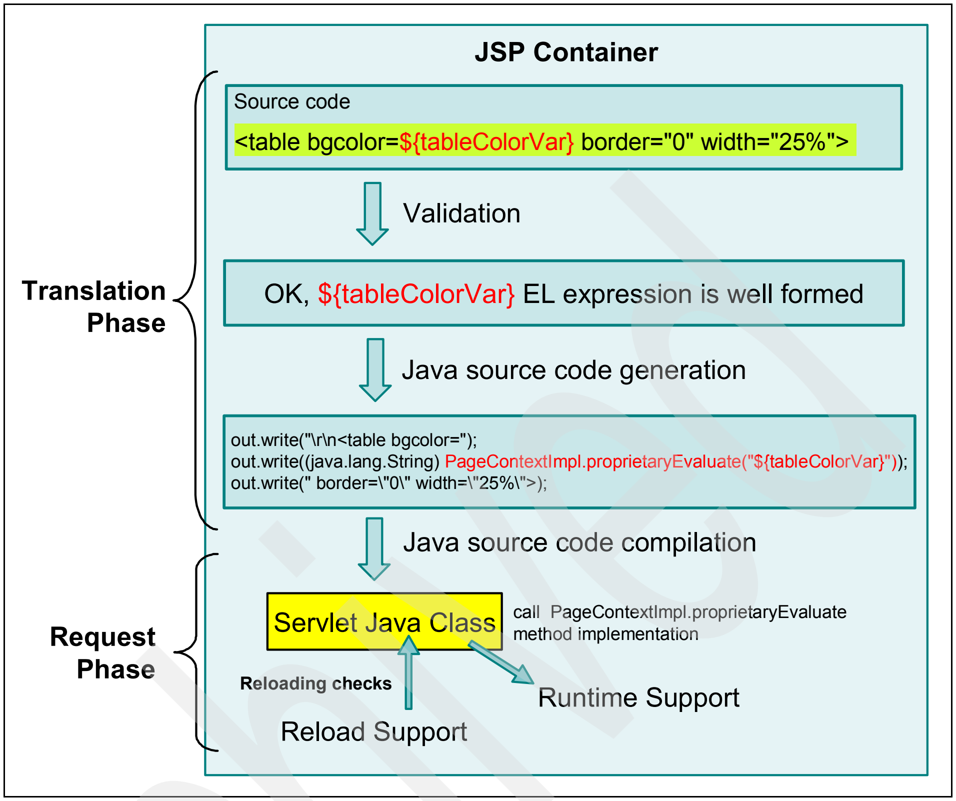


Figure 4-3 JSP processor phases overview

Translation Phase

During the translation phase, the JSP page (also called translation unit) has to be converted from text to an executable form so that it can run in an application server. The executable form of a JSP page is a Java class that implements javax.servlet.Servlet Interface. This means that a JSP is turned into an HttpServlet. For example, WebSphere Application Server's JSP container performs the translation phase in three steps:

1. Validation:

   The JSP container ensures that the syntax of JSP page is correct, and generates informative error messages when it encounters errors. To validate that a JSP page's syntax is correct, the JSP container reads the page character by character, parsing the page's contents. The JSP container looks for character sequences that it recognizes; sequences that indicate there is some syntax that it has to process. On this phase, for example, directives, custom tags, jsp:useBeans are validated.

However, the JSP processor does not recognize text sequences such as HTML syntax and simply puts them into the JSP page implementation class exactly as they appear in the JSP page. Also, the JSP container does not validate what is inside script elements (Declarations, Scriptlets and Expressions). A JSP container does not know how to parse Java code. It just passes the contents into the JSP page implementation class.

2. Java source code generation:

   If the JSP page is valid, the JSP container converts the text in the JSP page into Java source code for what becomes the JSP page implementation class. In this step the JSP processor has rules to convert Expression® Language (EL) expressions, custom tags and useBean actions in Java source code, for example.

3. Java source code compilation:

   The JSP container compiles the .java file it has generated. Java compilation creates a binary class, the JSP page implementation class, which resides on disk as a .class file. If there are Java compilation errors, then error messages are generated that identify the error, the JSP line number, and the equivalent .java source file line number where the error took place.

Request Phase

When a JSP is requested in a running application server, its page implementation servlet class is invoked. On each request, the servlet's service method executes on a thread of its own; the thread on which the request is executing. This way, a single JSP servlet can handle multiple concurrent requests.

From the WebSphere JSP container, we have two phases:

- Runtime support:

  Provides classes and methods used by the servlet. This is called runtime support.

- Reloading:

  Checks to see if the JSP source file, from which the servlet was created, has been modified and has to be re-translated. There are parameters in WebSphere configuration that can be configured. See Chapter 7, “Environmental performance considerations” on page 509 for additional details.

4.2.3 JSP basic syntax

The main syntax elements are described in Table 4-1.

Table 4-1 JSP syntax elements

Type of Element	Element content
Template Content (see 4.2.4, “Template content” on page 113)	Everything in your JSP source file that is not a JSP element. Includes all static content such as HTML.
Directives (see 4.2.5, “Directives” on page 114)	Instructions you place in your JSP to tell the JSP container how to build your page, such as whether to include another file.
Scripting elements (see 4.2.6, “Scripting elements” on page 115)	Declarations (<%!...%>) Scriptlets (<%...%>) Expressions (<%=...%>) Used to embed Java code into your JSPs.
Actions (see 4.2.8, “Actions” on page 117)	Actions provide high-level functionality, in the form of custom XML-style tags, to a JSP without exposing the scripting language. Standard actions include those to create, modify, and otherwise use JavaBeans within your JSP.
Expression Language (EL) (see 4.2.12, “Expression Language” on page 158)	Expression language is used for run-time assignment of values to action element attributes. It was first introduced as part of JSP Standard Tag Library (JSTL) 1.0 specification, but is now part of the JSP 2.0 specification. Example: <table bgcolor=${tableColorVar} border=0 width=”25%”> where ${tableColorVar} is Expression Language.

There is much documentation that provides guidance and help in building JSPs. It is not within the scope of this book teach you how to create a JSP. Our scope is focused on discussing best practices. In 4.10, “References” on page 249, you can find material to assist you in the initial steps to learn JSP coding.

Next we discuss each syntax element.

4.2.4 Template content

Template content is all the static content (HTML and comments) on your page. JSP does no recognize template content. The best practice here is to separate template content from Java code to make it easier to understand and debug.

We talk more about separating HTML from Java code in 4.2.9, “Tag libraries” on page 125.

4.2.5 Directives

Directives provide additional information to the JSP container and describe attributes for your page. We describe three directives normally used in JSPs below. In addition to these, JSP 2.0 introduced three additional directives which are available only to tag files. They are discussed in the JSP Standard Tag Library section (see “Tag libraries” on page 125).

Table 4-2 JSP directives

Directive	Purpose
page	Controls properties of the JSP
include	Includes the contents of a file into JSP at translation time
taglib	Makes a custom tag library available within the including page

The general syntax for a directive is:

```
<%@ directive [attr="value"]%>
```

or

```
<jsp:directive directive [attr="value"] />
```

Where *directive* is *page* or *include* or *taglib* and attr is a directive attribute.

The page directive has many attributes that are covered in more detail in other documentation. For performance issues, we review the two page directive attributes shown next:

- isThreadSafe:

  Indicates whether the resulting servlet is threadsafe. The JSP 2.0 specification advises against using isThreadSafe because the generated servlet might implement the SingleThreadModel interface, which has been deprecated in the Servlet 2.4 specification.

  The best practice is not to do anything. This maintains the default value of “false”. We talk about SingleThreadModel Interface in “Do not use SingleThreadModel” on page 193.

- session:

  Indicates whether the page requires participation in an HttpSession. By default, the value is “true”. You have the option of specifying the value as “false” (in which case, any reference to HttpSession in the page is not resolved).

  The best practice is to not create HttpSessions in JSPs. By default, a JSP session object is created implicitly if one does not exist. However, if the session is not required, creation can be avoided with the following directive:

  ```
  <%@ page session=”false” %>
  ```

4.2.6 Scripting elements

The Scripting elements are used to manipulate objects and perform processing in the page content. The best practice, however, is to not use scripting elements as explained in “Scripting elements: Best practices” on page 117.

Next, we discuss the three types: declarations, scriptlets, and expressions.

Declarations

The following list describes declarations scripting elements:

- Syntax:

  ```
  <%! java declarations %>
  or
  <jsp:declaration> java declarations </jsp:declaration>
  ```

- Purpose:

  You can declare static or dynamic class members or functions.

- Example:

  ```
  <jsp:declaration>
  public static final String LOG_NAME=”mylog.log”;
  public static final int DEFAULT_MESSAGE_ERROR_CODE= 500;
  </jsp:declaration>
  ```

- Recommendation:

  Be careful of use because the variables are class variables, not instance variables. This means that they do not remain in only one request, and they are not thread safe.

  For clarity, never user the <%! syntax for declarations. The XML syntax is self-descriptive and much clearer to the reader than remembering which JSP element uses an exclamation point.

Scriptlets

The following list describes scriptlets and scripting elements:

- Syntax:

```
<% java-statements %>
or
<jsp:scriptlet>
java-statements
</jsp:scriptlet>
```

- Purpose:

  Supports small sets of Java statements that you put inside page to logic flow, such as looping and branching.

- Example:

```
<jsp:scriptlet>
for(int i=0;i<users.size();++i){
out.println(“User Name: “+users(i).getName());
}
</jsp:scriptlet>
```

- Recommendation:

  Using the XML syntax when writing scriptlets is preferable.

  Variables declared here are local variables. This approach is preferable for thread safe code. Each request creates a new set of variables.

  Scriptlets allow you to include Java code within a JSP. This is not generally in keeping with good JSP design. Use tag libraries instead for better reuse and design.

Expressions

The following list describes expressions scripting elements:

- Syntax:

```
<%= a-java-expression %>
or
<jsp:expression> a-java-expression </jsp:expression>
```

- Purpose:

  Used when you have to output a value.

- Example:

```
The local server time is <%= new java.util.Date() %>
```

- Recommendation:

  Do not use a semicolon at the end of the expression command.

4.2.7 Scripting elements: Best practices

Many developers and support staff no longer choose to use scripting elements. There are some motivations for this. For example, use of declarations creates class variables. If you create a variable that contains user information in the declarations section, and then use it in other parts of the JSP code in your runtime environment, then in two or more executions, the correct value for each user gets lost. The correct way is declare variables in a scriptlet session, but it is very easy to make mistakes. Also, it is a bad programming design to mix HTML and comments with Java code. If you look at long line codes of a JSP with this mix, it is difficult to isolate distinct presentation code, difficult to maintain the code, and difficult to debug for errors. It is best to avoid scripting elements.

However, there are cases when you have to initialize variables before your JSP is ready to receive requests. Also, you might be required to deliver resources when the JSP is not receiving requests. The jspInit method, if declared, is called to prepare the page before the first request is delivered. Similarly, a JSP container can reclaim resources used by a JSP page when a request is not being serviced by the JSP page by invoking its jspDestroy method, if declared. For this situation, use declaration syntax as shown in Example 4-1.

Example 4-1 jspInit() and jspDestroy usage

```
<jsp:declaration>
public void jspInit()
{
  // do some initialization work
}
public void jspDestroy()
{
  // do some finalization work
}
</jsp:declaration>
<HTML>
  <BODY>
      <!--Another JSP use? put here, but be carefull with scripting
elements or do not use-->
  </BODY>
</HTML>
```

4.2.8 Actions

Actions provide a higher level of functionality than the declarations, expressions, and scriptlets you have seen thus far. Unlike the scripting elements, actions are independent of any scripting language. In many respects, JSP actions are like built-in custom tags. In fact, only XML syntax is defined for actions; there is no equivalent <% syntax. There are five categories of standard actions in Table 4-3.

Table 4-3 Actions categories

Category	Purpose
JavaBeans components (<jsp:useBean>) (“JavaBeans components” on page 118)	Provides a well-designed integration between JavaBeans and HTML forms.
Including and Forwarding (<jsp:include> and <jsp:forward>) (“Best practices to use composed Web components” on page 152)	Used for composing your JSP with another JSPs in request time.
<jsp:plugin>	The sole purpose is generate the appropriate <OBJECT> or <EMBED> tag to load the Java Plug-in software when browser support for Java is out-of-date or missed.
<jsp:invoke> (“Tag libraries” on page 125)	New with JSP 2.0. Used with tag files.
<jsp:doBody> (“Tag libraries” on page 125)	New with JSP 2.0. Used with tag files.

We do not cover the jsp:plugin action category in this book. However, you can find more details the following URL:

https://www6.software.ibm.com/developerworks/education/j-introjsp/index.html

JavaBeans components

Before we discuss how JavaBeans are used in a JSP, we have to understand what a JavaBean is. The main goal is to understand the basic syntax and best practices.

The JavaBeans component model is a framework and a specification that allows developers to write reusable and portable components that can be used in different frameworks. Also, it can be manipulated visually from a builder tool and can provide some features such as introspection (a bean worker can be analyzed by a builder tool), customization (you can customize behavior and appearance of a bean), events support, properties edition support for programmatic use and persistence support (a bean can be customized from a application builder and your state can be saved and reloaded later).

The class construction does not have to inherit any class or interface. However, to make them from visual containers, JavaBeans have to inherit java.awt.Component.

JSP provides a well-designed integration between JavaBeans and HTML forms. The <jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty> action tags work together to achieve this integration. Table 4-4 briefly explains these components.

Table 4-4 Action tags for javabeans

Action	Purpose
<jsp:useBean>	Prepares a JavaBean to be used in a JSP
<jsp:setProperty>	Sets a property for the JavaBean
<jsp:getProperty>	Gets a JavaBean property value in String Java type

<jsp:useBean>

The <jsp:useBean> tag tells the JSP that you want a bean of a given name (which might be a request-time expression) and scope. You also provide creation information if necessary. The JSP checks to see if a bean of that name and scope already exists. If not, the bean is instantiated and associated with a declared variable. See the syntax in Example 4-2.

Example 4-2 <jsp:useBean> syntax

```
<jsp:useBean id=" name " scope="Bean-Scope" class="Bean-Class"
type="Varible-Class" beanName="Bean-Serializated-Name or ClassName"/>
or
<jsp:useBean id=" name " scope="Bean-Scope" class="Bean-Class"
type="Varible-Class" beanName="Bean-Serializated-Name or ClassName"/>
creation-body
</jsp:useBean>
```

If the bean has to be created and you use the second form of jsp:useBean shown in Example 4-2, the statements that make up the creation-body are also executed.

Table 4-5 gives a brief description of the elements.

Table 4-5 jsp:useBean elements

Attribute name	Description	Required	Example
id	It is in short the variable name to be used in the page coding	yes	`id=”employee”`

Attribute name	Description	Required	Example
scope	The bean is viable during scope definition, which can be page, request, session, or application. The default is page.	No	`scope=”session”`
class	The class of Bean. It is used for creation purposes when the JSP engine does not find it defined in an existing JavaBean scope. The requirement is that it must be a non-abstract and no parameter constructor.	No	`class=”packagename.EmployeeAddress” where packagename is used if class belongs a package`
type	This defines the variable type of id parameter.	No	`type=”pakagename.AddressBean” where packagename is used if class belongs a package`
beanName	The serialized bean name is loaded from a serialized file or serialized class name if we have to create a new instance.	No and if used, *type* attribute is required	`beanName=”data.employee.john”` `beanName=”AddressBean”`

- scope attribute:

  Table 4-5 gives a brief description of the scope types.

Table 4-6 Scope types

Scope	Explanation
page	The bean is good only within the defining JSP and is recreated for each new request.
request	The bean is good throughout that request and is available to included or forwarded pages.
session	The bean is associated with the particular session responsible for its creation and is good for the lifetime of the session.
application	The bean is common to all sessions and is good until the Web application terminates.

Page scope (Example 4-3) depends on your application design. If your application does not have to pass JavaBean objects between JSPs, use *page* scope if possible because it guarantees that used variables are thread safe. For more information, see 4.5.6, “Implement thread safe servlets” on page 192. Also, the object life cycle is related to page execution, and this approach avoids letting objects remain in memory after execution. You get more benefits from this approach in high volume access. Otherwise, a good practice is to use *request* or *session* scope, depending on design. You can get more details on scope in 4.2.10, “Implicit objects” on page 148.

Example 4-3 Page scope usage

```
<jsp:useBean id="trainBean" scope="page"/>
or
<jsp:useBean id="trainBean"/><!--Default Behavior-->
```

- class attribute:

  The functionality here is that if useBean does not find a bean with a scope, a new instance of the object is created. It requires that the Bean class have a public constructor with no parameters (Example 4-4).

Example 4-4 class attribute usage

```
<jsp:useBean id=”employee” class=”EmployeeBean”/>
```

The translation in servlet code is similar to Example 4-5.

Example 4-5 class usage translation

```
EmployeeBean employee=
(EmployeeBean)pageContext.getAttribute(“employee”);
if (employee == null){
employee = new EmployeeBean();
pageContext.setAttribute(“employee”, employee);
```

- type attribute:

  This configuration tells the JSP engine to for search an instance of this type or subtype. If an instance is not found, a **java.lang.InstantiationException** is thrown, and no new instance of Bean is created (Example 4-6).

Example 4-6 type usage

```
<jsp:useBean id=”employee” type=”PersonBean”/>
```

- class and type attributes:

  If useBean does not find a bean in a scope, a JavaBean is created using new instance syntax. This requires that the Bean class has a public constructor with no parameters. The variable type used on the page is from the type parameter information (Example 4-7).

  Example 4-7 class and type usage - 1

```
<jsp:useBean id=”employee” class=”EmployeeBean” type=”PersonBean”/>
```

  It is similar to the following coding (Example 4-8).

  Example 4-8 class and type usage - 2

```
PersonBean employee=
(PersonBean)pageContext.getAttribute(“employee”);
if (employee == null){
employee = new EmployeeBean();
pageContext.setAttribute(“employee”, employee);
}
```

- beanName and type:

  If useBean does not find a bean in a scope, it is created as a JavaBean using **Beans.instantiate()** syntax. The variable type used on the page is from the type parameter (Example 4-9).

  Example 4-9 beanName and type usage - 1

```
<jsp:useBean id=”employee” beanName=”somepackage.EmployeeBean”
type=”PersonBean” scope=”session”/>
<!--somepackage.EmployeeBean.ser is a serialized version of
somepackage.EmployeeBean Object. Remember to use wihout .ser
extension-->
```

  It is similar to the following coding (Example 4-10).

  Example 4-10 beanName and type usage - 2

```
PersonBean employee=(PersonBean)session.getAttribute(“employee”);
if(employee==null){
ClassLoader classloader = this.getClass().getClassLoader();
employee =
java.beans.Beans.instantiate(classloader,”somepackage.EmployeeBean”)
;
session.setAttribute(“employee”,employee);
}
```

The best practice here is not to use <jsp:UseBean> with beanName. It is an expensive operation because the JVM checks the file system for a serialized bean. Hence, you can use another option without performance problems.

For more details about Beans.instantiate functionality, see the following URLs:

http://java.sun.com/products/javabeans/docs/spec.html

http://jcp.org/aboutJava/communityprocess/final/jsr152/

<jsp:setProperty>

The <jsp:setProperty> tag usage is straightforward. The purpose is to assign new values to javaBean properties. See the syntax in Example 4-11.

Example 4-11 jsp:setProperty syntax

```
<jsp:setProperty name="Bean-Name" property="Property-Name"
value="Value" param="HttpRequest-Parameter-Value"/>
```

The best practice here concerns the *property* attribute use. See Table 4-7 for a review of each attribute.

Table 4-7 jsp:setProperty attributes

Attribute	Required	Purpose
name	Yes	Used to identify the name of bean in JSP page.
property	Yes	Used to identify the property name to be set. The property name=”LastName” is equivalent to setLastName on the JavaBean method, for example. If request parameters match, the JavaBean attributes does not require value and param attributes.
value	No. if present *param* cannot be present	Used to set the new value to the property. I can be used with request time attribute expression using syntax <%=...%>. for example: value=<%=lastNameVar%>
param	No. if present *value* cannot be present.	Used to get request parameter attributes. For example, the param=”LastName” is equivalent to request.getParameter(“LastName”).

The requirement to use the <jsp:setProperty> action is that it has to be related to a previous <jsp:useBean> action and the *name* attribute value for <jsp:setProperty> must be equal to *id* attribute on a previous <jsp:useBean> action.

For best practice purposes, if you use “*” for the *property* value, the JSP processor automatically assigns the request value parameter XXX inside method setXXX inside JavaBean. See Example 4-12.

Example 4-12 Parameter with “”value usage*

```
<jsp:useBean id=”employee” class=”EmployeeBean” scope=”session”/>
<jsp:setProperty name=”employee” property=”*”/>
<!--The setProperty action is equivalent to!-->
<%
employee.setFirstName(request.getParameter(“firstName”));
employee.setLastName(request.getParameter(“LastName”));
employee.setAddress1(request.getParameter(“address1”));
employee.setAddress2(request.getParameter(“address2”));
employee.setCity(request.getParameter(“city”));
employee.setZip(request.getParameter(“zip”));
%>
<!----Java Bean code snippet------>
<!--Look equivalence to JavaBean class attributes
public class EmployeeBean {
String firstName;
String lastName;
String address1;
String address2;
String city;
String zip;-->
```

If you look at the previous example, you can see the equivalence between each request.getParameter() syntax with each class attribute inside javaBean code. Keep in mind, for example, that a request with firstName value is related with a setFirstName method and getFirstName method inside the javaBean.

Another tip is about automatic conversion when you use another property inside a bean that is not String type. Consider that EmployeeBean has an attribute age with integer type and an attribute regular with boolean type. See Example 4-13 for clarity.

Example 4-13 setProperty for different types

```
<jsp:useBean id="employee" class="somepackage.EmployeeBean"
scope="session"/>

<!-- Conversion is automatically done to correct type by JSP Processor
-->
<jsp:setProperty name="employee" property="regular" value="true"/>
<jsp:setProperty name="employee" property="age" value="34"/>
```

```
<!-- The same efect you can gain with * param value -->
<!-- For a request =>
http://localhost:9080/JSP/Employee.jsp?regular=true&age=34 -->
<jsp:setProperty name="employee" property="*"/>

<!-- But if you use expression manual conversion is needed. If not
example will not compile -->
<%String age="34";%>
<jsp:setProperty name="employee" property="age"
value="<%=Integer.parseInt(age)%>"/>

<!-- See the results -->
<P>Regular : <jsp:getProperty name="employee" property="regular"/></P>
<P>Age : <jsp:getProperty name="employee" property="age"/></P>

<!--The jsp:getProperty is the same than
out.println(employee.getAge()); for example-->
```

Also, as you can see in the previous example, the jsp:getProperty acts as an out.prinln of JavaBean property value. For more details about useBean features, see the following URL:

http://www.ibm.com/developerworks/edu/j-dw-jsp-i.html

4.2.9 Tag libraries

Scriptlets are good for quick coding. However, in the long run, a less cluttered solution for JSP pages is a better answer. Scriptlets introduce more long-term complexity to your pages than they offer in terms of short-term benefit. They interweave all sorts of HTML with Java code, which makes debugging and authoring tricky. They are not reusable, which often leads developers to cut-and-paste between JSP pages, which in turn leads to multiple versions of the same piece of code. And they make error reporting difficult, since JSP pages have no clean-cut way to output the script errors.

Tag libraries are a standard way of packaging tag extensions for applications using JSPs.

Tag extensions address the problem that arises when a developer wants to use non-trivial processing logic within a JSP. Java code can be embedded directly in the JSP using the standard tags described above. This mixture of HTML and Java makes it difficult to separate development responsibilities (the HTML/JSP designer has to maintain the Java code) and makes it hard to use appropriate tools for the tasks in hand (a page design tool does not provide the same level of support for Java development as a Java development tool). This is essentially the reverse of the problem described when discussing servlets above. To address this problem, developers have documented the View Helper design pattern, as described in *Core J2EE Patterns: Best Practices and Design Strategies* by Crupi, et al. The pattern catalog contained in this book is also available at the URL:

http://java.sun.com/blueprints/corej2eepatterns/Patterns/

Tag extensions are the standard way of implementing View Helpers for JSPs.

Using tag extensions, a Java developer can create a class that implements some view-related logic. This class can be associated with a particular JSP tag using a tag library descriptor (TLD). The TLD can be included in a Web application, and the tag extensions defined within it can then be used in JSPs. The JSP designer can use these tags in exactly the same way as other (standard) JSP tags. The JSP specification includes classes that can be used as a basis for tag extensions and (new in JSP v2.0) a simplified mechanism for defining tag extensions that does not require detailed knowledge of Java.

Using tag libraries instead of Java code in your JSP pages offers the following benefits:

- They help separate presentation from implementation.
- They are easy to maintain and reuse, offering better usability by design.
- They simplify complex actions.
- They provide Java-coded functions without the task of coding in Java.
- They can dynamically generate page content and implement a controlled flow.

JSP Standard Tag Libraries or custom tag libraries can be used in your JSP pages. First, we discuss the JSTL to get an understanding of how to use a tag library. Following that, we discuss how to implement a custom tag library (see “Implementing JSP custom tags” on page 147).

Overview of tag library elements

To understand the tag library mechanism, we must understand the core elements:

1. Tag handlers:

   This is a Java class that implements tag library functionality. The class has to implement one of the following tag interfaces: Tag, IterationTag, BodyTag, and SimpleTag from the javax.servlet.jsp.tagext package. The last tag interface was introduced in JSP 2.0. While classic tags handlers support doStartTag() and doEndTag methods, the SimpleTag interface provides a simple doTag() method which is called once only for any tag invocation. This single method controls all tag logic. It was created to make tag development easier.

However, we do not have to implement these classes directly. The API provides three adapter classes. TagSupport, BodyTagSupport, and SimpleTagSupport implement the IterationTag, BodyTag, and SimpleTag interfaces respectively and provide default implementation of all methods. The programmer only has to override those methods to customize to his requirements. See Example 4-14.

Example 4-14 Tag handler implementation example

```
package somepackage;

import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloTag extends TagSupport {

    // The "person" to say hello to
    private String name;

    // Accept the attribute data
    public void setName(String name) {
      this.name = name;
    }

    public int doEndTag() {
      try{
      StringBuffer message = new StringBuffer("Hello, ");
      message.append(name).append("!");
      pageContext.getOut().println(message.toString());
    } catch (IOException ignored) { }
      return EVAL_PAGE;
    }
}
```

2. Tag Library Descriptor:

   This component describes how the JSP engine finds the tag handler and how it can be used. This archive has a .tld extension. See Example 4-15.

Example 4-15 Tag library descriptor file example

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<taglib>
   <tlibversion>1.1</tlibversion>
   <info>A simple tag library</info>
```

```
  <tag>
    <name>displayName</name>
    <tagclass>somepackage.HelloTag</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
       <name>name</name>
       <required>true</required>
    </attribute>
    <info>Display Name</info>
  </tag>
</taglib>
```

3. Taglib map inside web.xml:

   The XML tag has a function to describe where a JSP Engine finds the tag library descriptor. The basic element (<taglib-uri> value) is an alias and the second element is a tag library location (<taglib-location>). It belongs to the <jsp-config> enclosing tag. See Example 4-16.

   Example 4-16 Code snippet of xml configuration inside web.xml

```
....
<jsp-config>
   <taglib>
    <taglib-uri>helloURI</taglib-uri>
    <taglib-location>/WEB-INF/HelloTagLib.tld</taglib-location>
   </taglib>
</jsp-config>
```

4. The directive on the JSP page to call the tag library:

   This is a directive we use to refer to the tag library. See Example 4-17.

   Example 4-17 Code snippet in JSP to call a tag library

```
<%@ taglib prefix="hello" uri="helloURI"%>
<P><hello:displayName
name="John"/></P>
```

Using a tag library

Now that we have had a basic overview of elements, next we show how tag library execution works when you are using a tag library. In this description, we begin after the tag handler and tag library descriptor discussed previously:

- Putting the information about the tag library inside a JSP page:

  <%@ taglib prefix="hello" uri="HelloTagLib.tld"%>

This is the simplest way to point to tag library information. Note that, in this example, we put the direct tag library name on the JSP page. This forces the JSP engine to search the tag library directly in the same directory where the JSP page exists. The is not secure because if, for example, a user calls http://localhost/JSP/HelloTagLib.tld, it is possible to see all tag library content.

A good place to put your tag library file is inside the WEB-INF root dir. Remember that WEB-INF is where you put your class files, jar files, and web.xml also.

This directory is not a part of the document application root. This means that you cannot view it using a browser.

Using /WEB-INF/HelloTagLib.tld as a value for the URI parameter tag library directive is not recommended because if you have to use a new version of the tag library, for example, HelloTagLib_2.tld, you have to change the code.

JSP supports a solution using a mapping between an alias and a real tag library location. This is called a taglib map, as discussed previously.

- Tag library map details:

  We can use any alias to make a mapping. For example, if we use the tag uri="http://www.itso.ibm.com/JSP/HelloTagLib" it does not means that the JSP engine loads the tag library information from that URI. It means that the JSP engine tries to find an alias inside the web.xml configuration with a key called http://www.itso.ibm.com/JSP/HelloTagLib in the <taglib-uri> xml tag. See Example 4-18.

Example 4-18 Alias from tag lib map example

```
<jsp-config>
   <taglib>
<taglib-uri>http://www.itso.ibm.com/JSP/HelloTagLib</taglib-uri>
    <taglib-location>/WEB-INF/HelloTagLib.tld</taglib-location>
   </taglib>
</jsp-config>
```

The recommended method to define a tag library location when you receive a tag library is that the tag library class and the TLD comes in a jar file using the structure shown in Example 4-19.

Example 4-19 Jar that contains a taglibrary

```
somepackage/HelloTag.class
somepackage/Utility.class
META-INF/taglib.tld
```

It is mandatory from the JSP specification that the tag library name is taglib.tld and it has to be placed in META-INF directory inside the jar. The mapping best practice is similar to what is shown in Example 4-20.

Example 4-20 Best practice taglib mapping

```
<jsp-config>
   <taglib>
<taglib-uri>http://www.itso.ibm.com/JSP/HelloTagLib</taglib-uri>
    <taglib-location>taglibs-hello.jar</taglib-location>
   </taglib>
</jsp-config>
```

The taglibs-hello.jar directory is in <doc-root>/WEB-INF/lib, where other project jars and third party jar files are stored.

If your TLD is not in a jar file, you can put it in <doc-root> or <doc-root>/WEB-INF. See Example 4-20 and Example 4-20.

Example 4-21 Taglib location from <doc-root> directory with / in the beginning

```
<taglib-uri>SomeAlias</taglib-uri>
<taglib-location>/somedir/HelloTagLib.tld</taglib-location>
this tag will points to <doc-root>/somedir/HelloTagLib.tld
```

Example 4-22 Taglib location from <doc-root> directory without / in the beginning

```
<taglib-uri>SomeAlias</taglib-uri>
<taglib-location>somedir/HelloTagLib.tld</taglib-location>
this tag will points to /WEB-INF/somedir/HelloTagLib.tld
```

If a JSP cannot find the tag library, it searches in the same directory that the JSP page is stored in. However, if <taglib-location> is an absolute URL path, an exception is thrown.

Using JSP Standard Tag Library (JSTL)

The JSP Standard Tag Library is a collection of custom tag libraries that implement general purpose functionality common to Web applications, including iteration and conditionalization, data management formatting, manipulation of XML, and database access. By providing standard implementations for typical presentation layer tasks such as data formatting and iterative or conditional content, the JSTL allows JSP authors to focus on application-specific development requirements, rather than “reinventing the wheel” for these generic operations.

Incorporating JSTL core tags

In this example, wel use the JSTL library installed in RAD.

Follow these steps:

1. We create our test project in RAD. For these steps, create a new project as shown in Figure 4-4. In our example, it is JSTL Project.

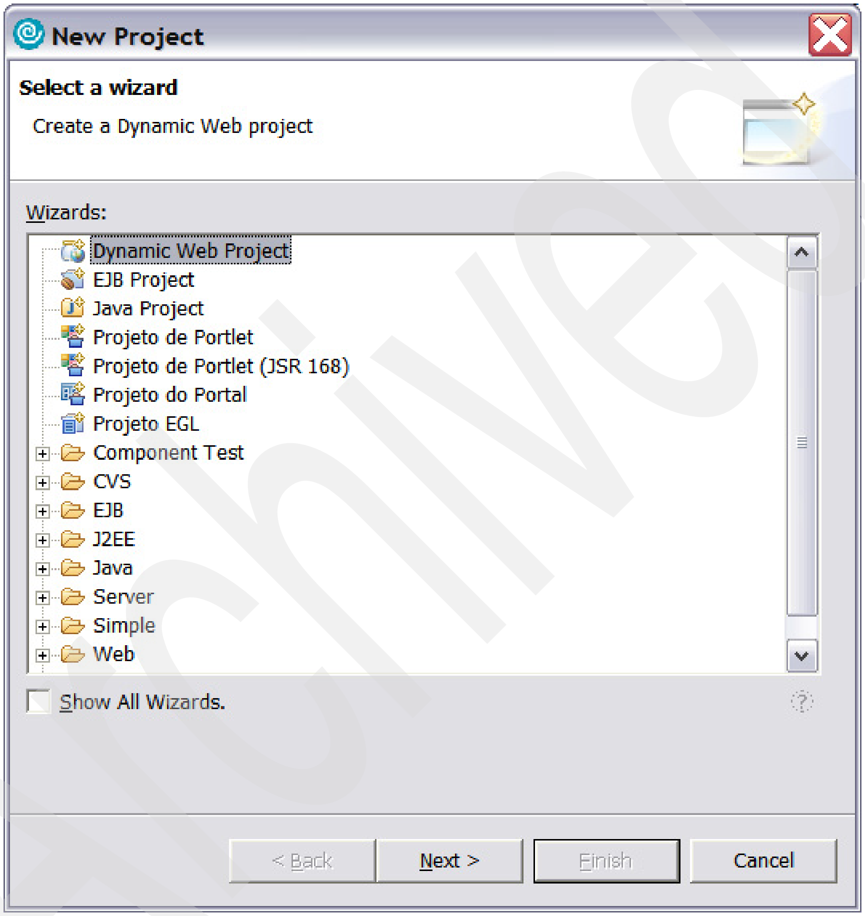


Figure 4-4 Creating a Web project

2. Select **Dynamic Web Project** and click **Next** (see Figure 4-5).

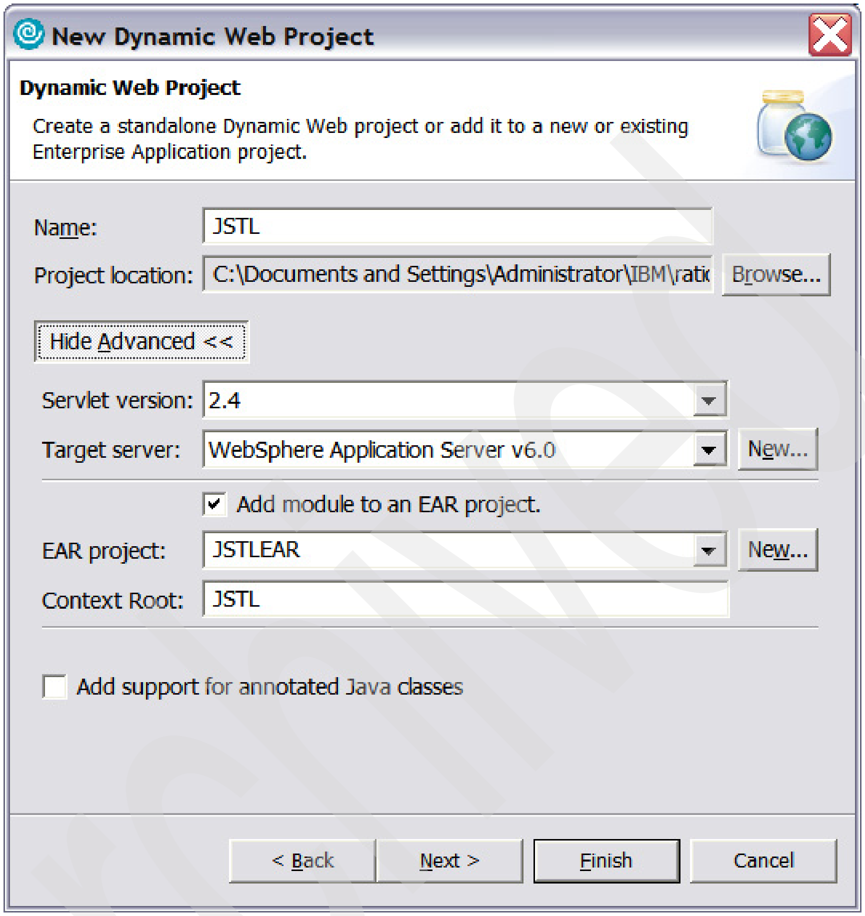


Figure 4-5 Defining a name for a Web project

3. We named our project **JSTL**. Click **Finish** as shown in Figure 4-5.

4. We now create a new JSP file as shown in Figure 4-6.

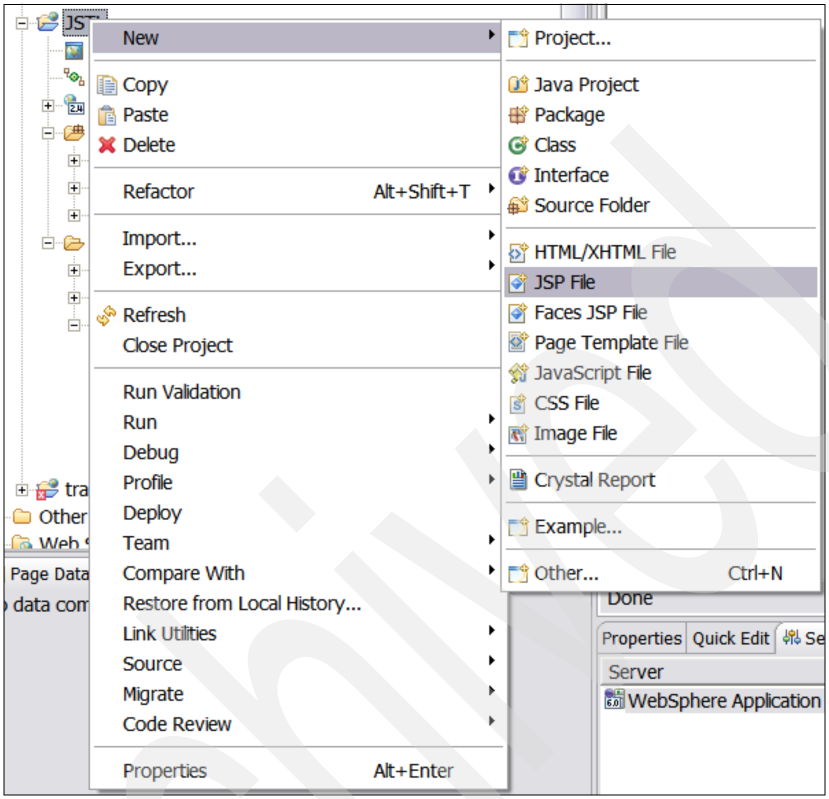


Figure 4-6 Creating a JSP file

5. In the JSP creation screen shown in Figure 4-7, enter `JstlCoreExample` in the JSP page, and check the **Configure advanced options** check box. Click **Next**.

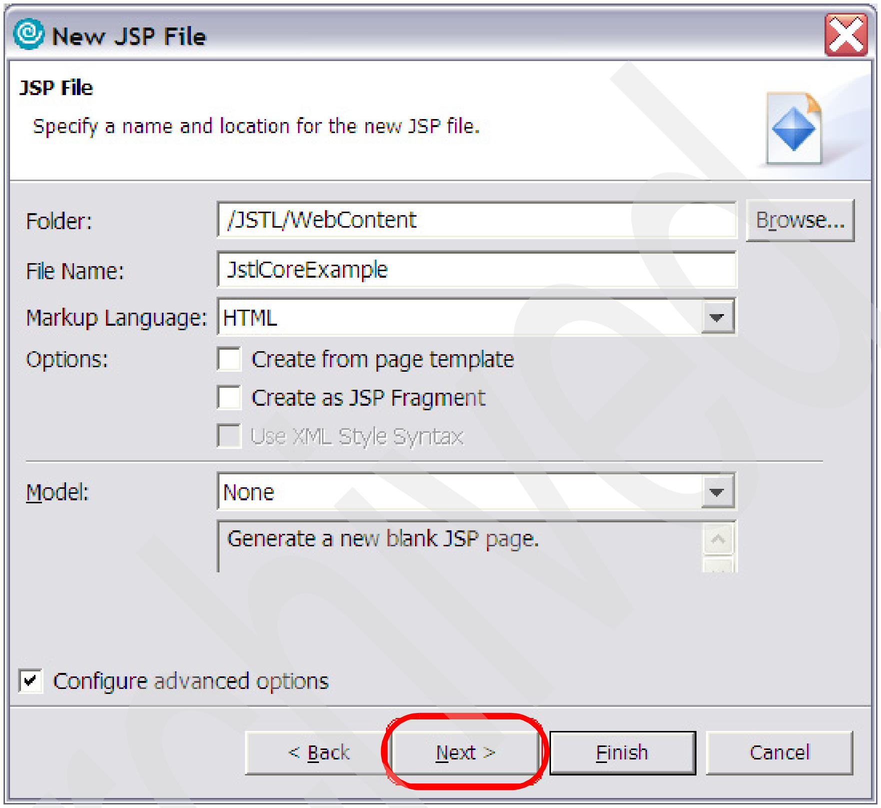


Figure 4-7 Specifying a name and setting advanced options

6. The screen shown in Figure 4-8 is displayed to add tag libraries. Click **Add**.

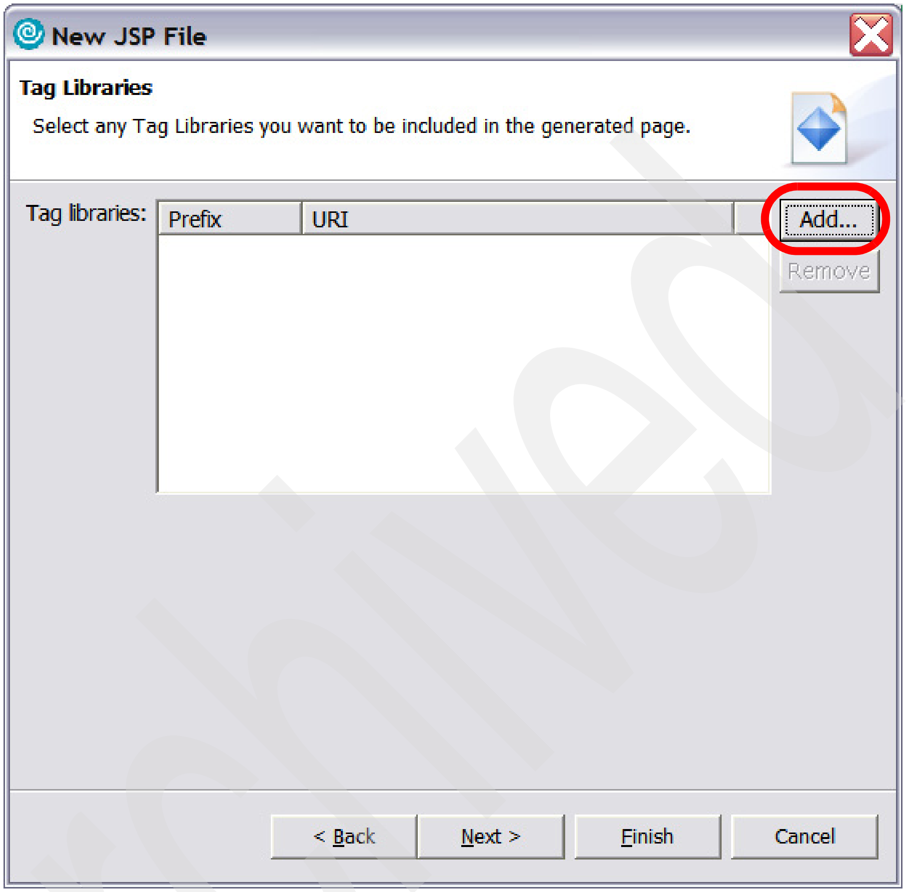


Figure 4-8 Using the Add button to configure a tag library

7. In the next screen, you select the option, http://java.sun.com/jsp/jstl/core/ (Figure 4-9). You can see that the tag library belongs to a standard library from WebSphere. In this case, no copy of a jar is required in the WEB-INF/lib application directory because it is a standard library visible to all Web applications in the WebSphere environment.

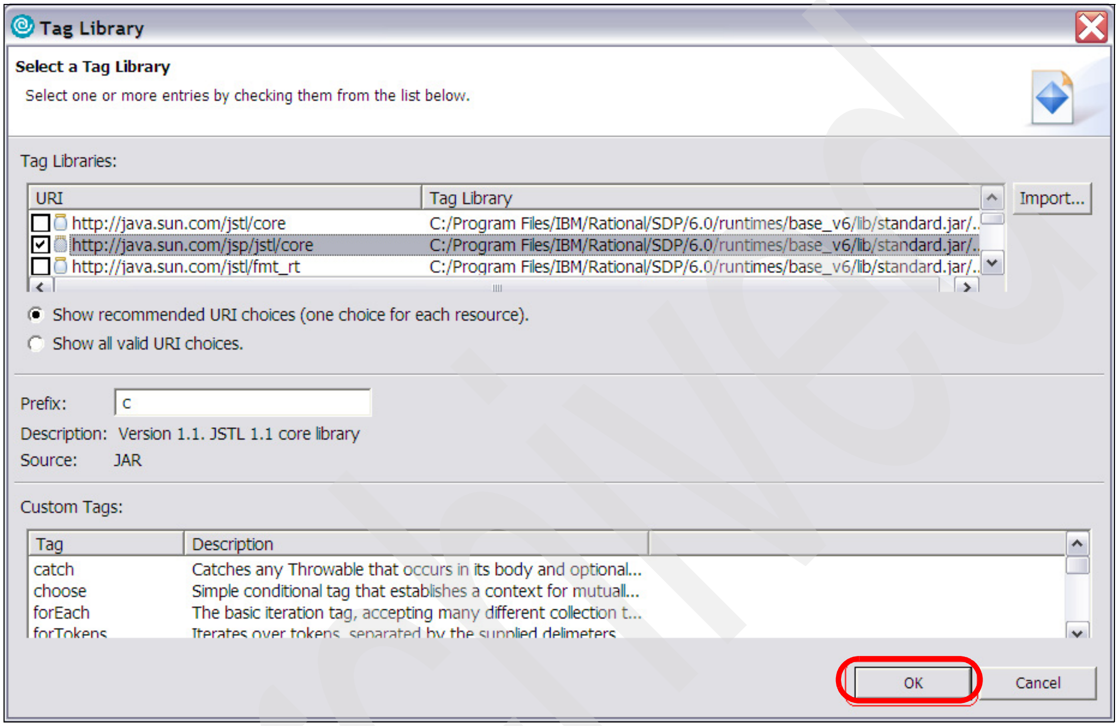


Figure 4-9 Selecting a standard tag library to use

8. After selecting the core tag library, click **OK**, and on the next screen, click **Finish**.

An important thing to notice here is that no configuration inside web.xml is required because the standard tag is from the WebSphere Application Server environment.

The result is that inside the JSP page, you can see that the taglib directive is created automatically for you.

Example 4-23 Taglib directive created from wizard in JSP page

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

You can see the entire example in Example 4-24.

Example 4-24 Taglib directive creation example

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<TITLE>JstlCoreExample.jsp</TITLE>
</HEAD>
<BODY>
<jsp:useBean id="user" scope="session" type="somepackage.PersonBean"
class="somepackage.EmployeeBean">
<jsp:setProperty name="user" property="firstName" param="name"/>
</jsp:useBean>
Hello <c:out value="<%=user.getFirstName()%>" default=="Guest"/>!
</BODY>
</HTML>
```

In our example we mix useBean usage and the JSTL tag **c:out**. The interesting point here is that the c:out tag uses a default value if the user.getFirstName() method returns a null value. Executing our JSP without parameters in the URL returns the results shown in Figure 4-10.

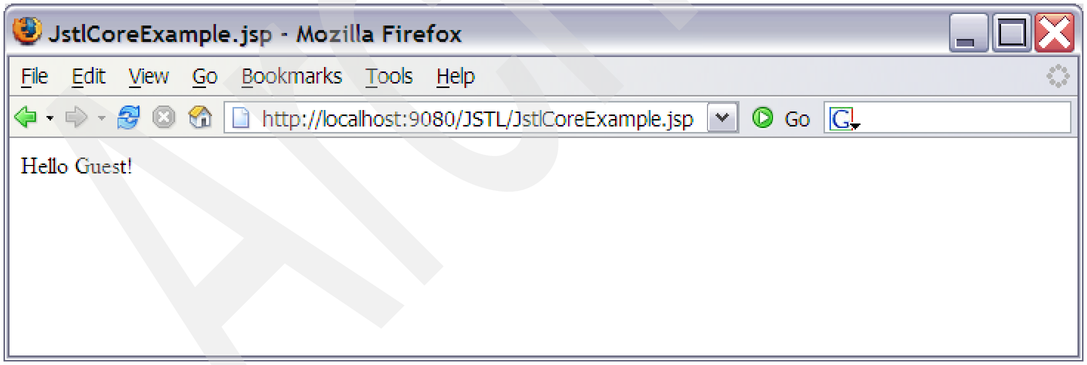


Figure 4-10 Results from our JSP core tag library usage example

Incorporating JSTL third party tags

We use the JSTL library from the Apache Project and RAD in our example. RAD has a core tag library to do internationalization actions, but for our example, suppose that you require a formatCurrency function that does not exist in the default installed JSTLs in RAD.

Follow these steps:

1. Download the required tag library. For our example, we download a tag library from the URL:

   `http://jakarta.apache.org/taglibs/doc/i18n-doc/intro.html`

2. After the download, unzip into a directory.

3. For the following steps, we use the existing project created in the previous example “Incorporating JSTL core tags” on page 131, the JSTL project.

   We now create a new JSP file as shown in Figure 4-11.

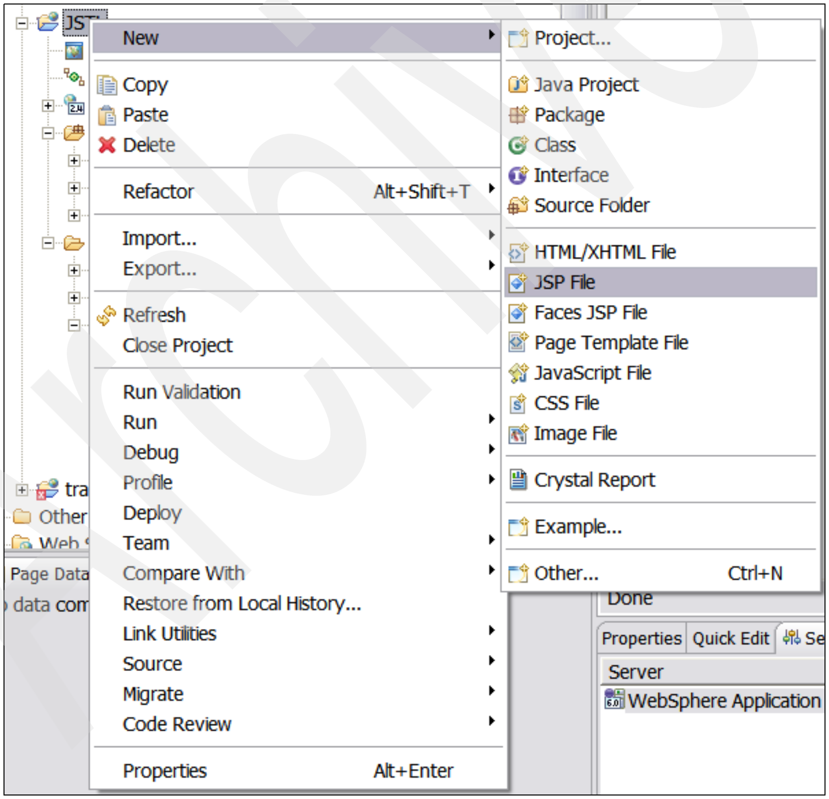


Figure 4-11 Creating a JSP file

4. In the JSP creation screen in Figure 4-12, enter `Internationalization` for the file name and check the **Configure advanced options** check box. Click **Next**.

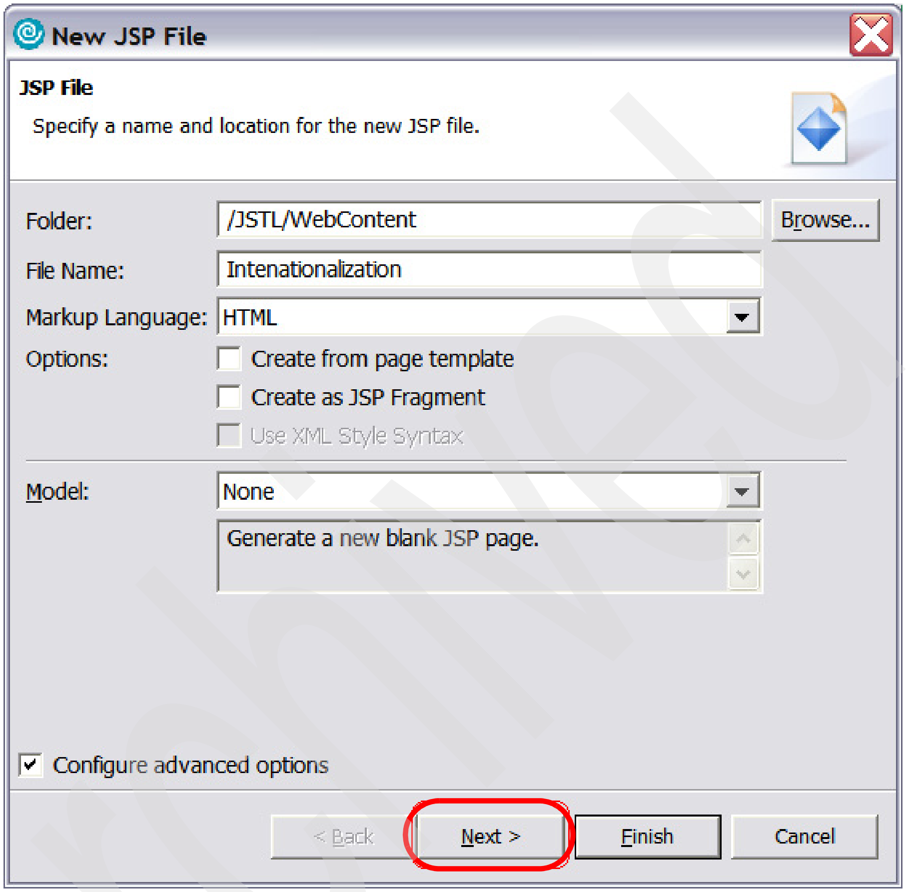


Figure 4-12 Defining the JSP file name and clicking the Next button

5. The screen shown in Figure 4-13 is displayed to add tag libraries. Click **Add**.

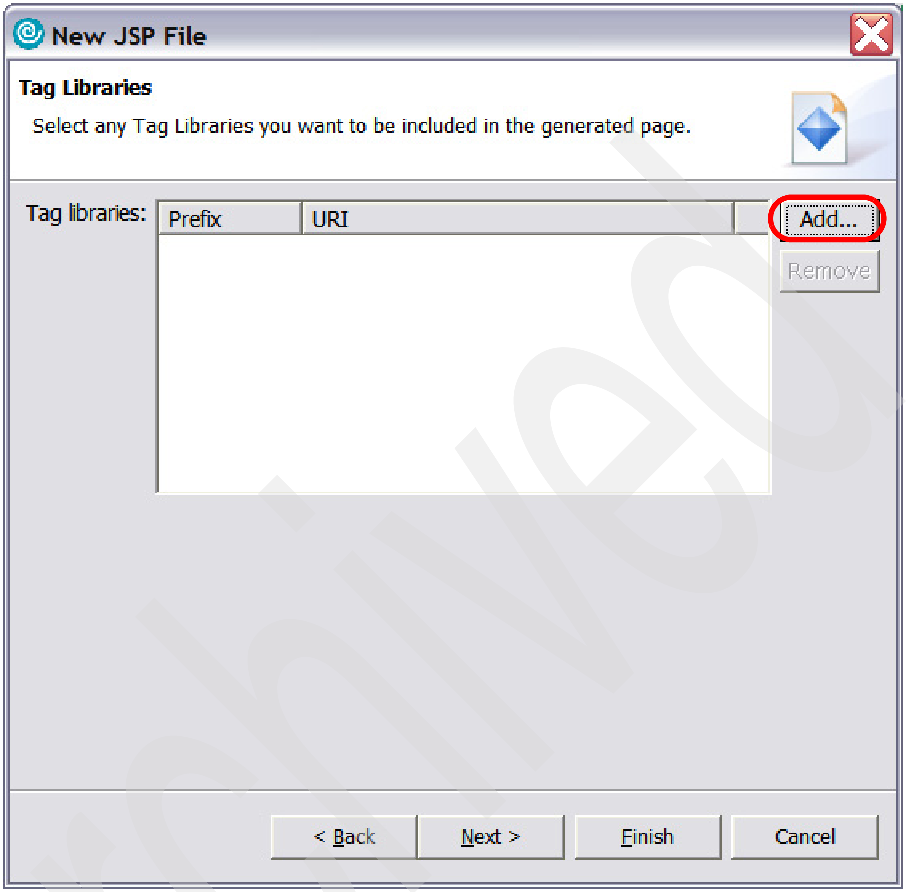


Figure 4-13 Using the Add button to configure a tag library

6. On the next screen (Figure 4-14), click **Import** and search for a jar file in the unzipped directory named `taglibs-i18n.jar`.

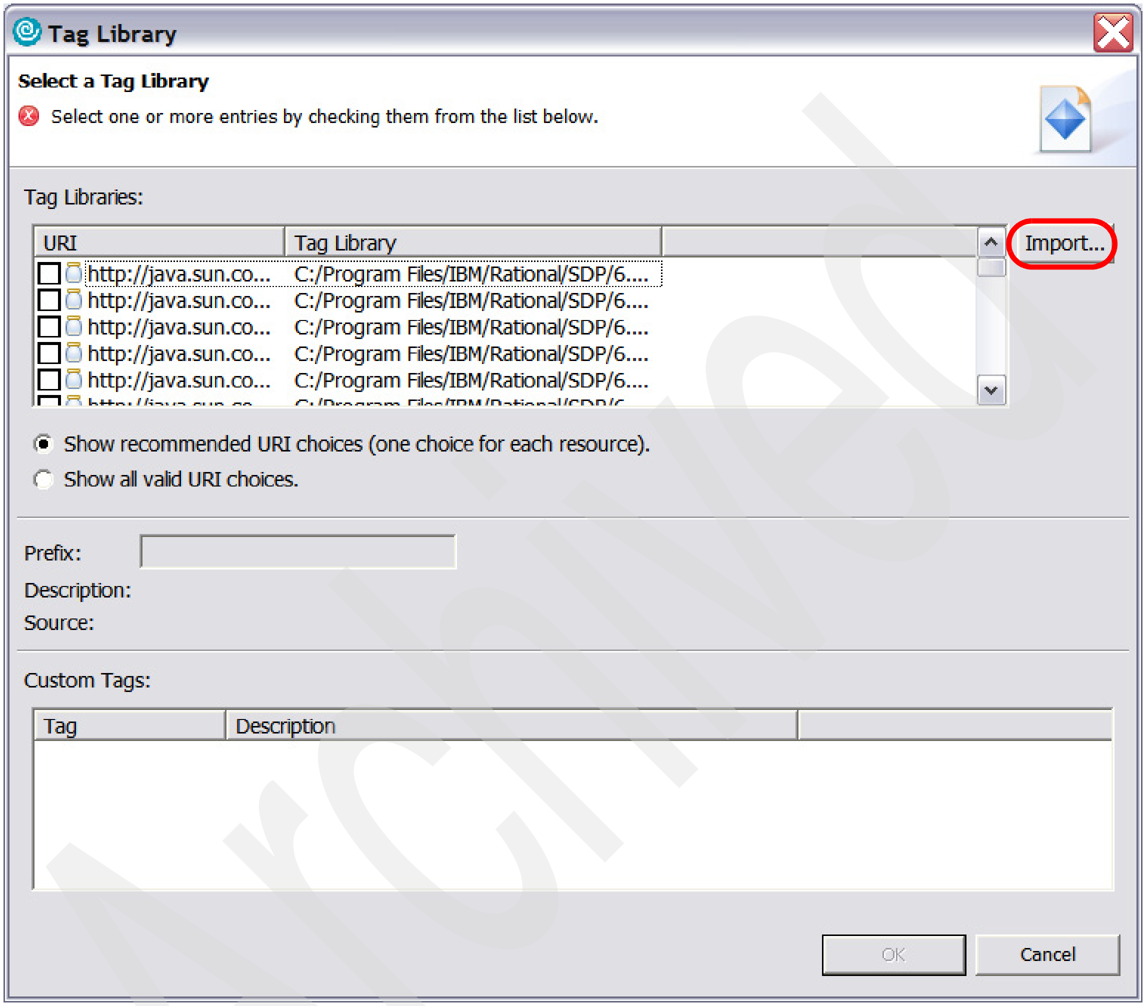


Figure 4-14 Importing a tag library

7. In our example, we selected the directory shown in Figure 4-15.

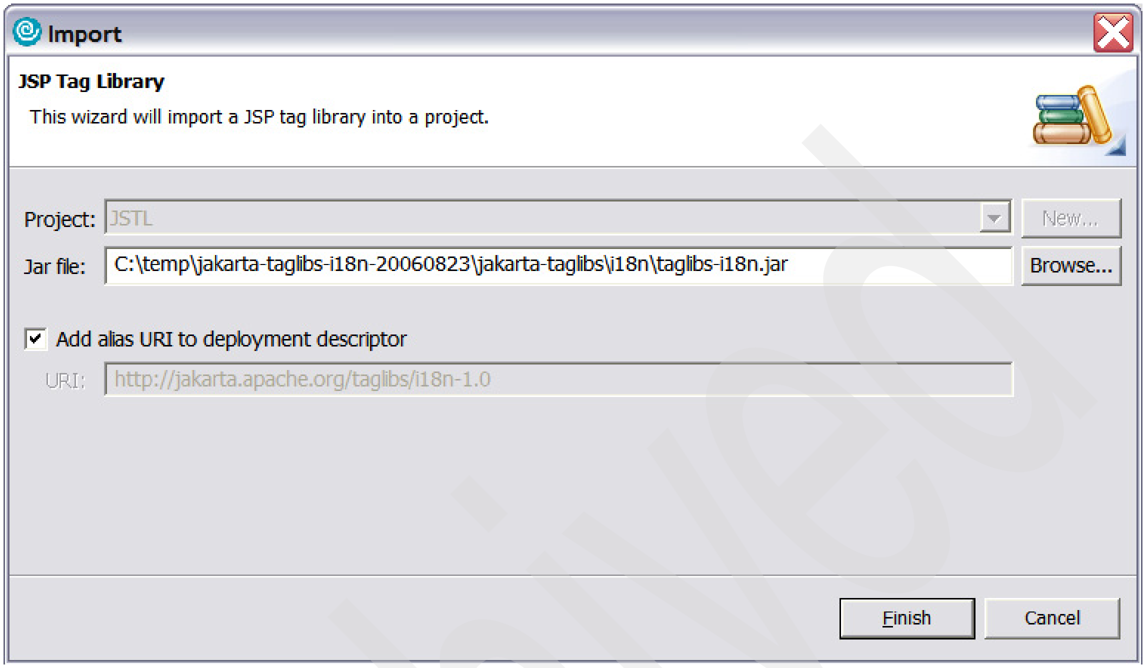


Figure 4-15 Finding the tag library jar

8. Click **Finish** to complete the import.

   The Tag Library window is displayed, showing our tag library in the list (Figure 4-16). Note that Rational Application Developer has opened the Tag Library Descriptor and detected the Custom Tags available.

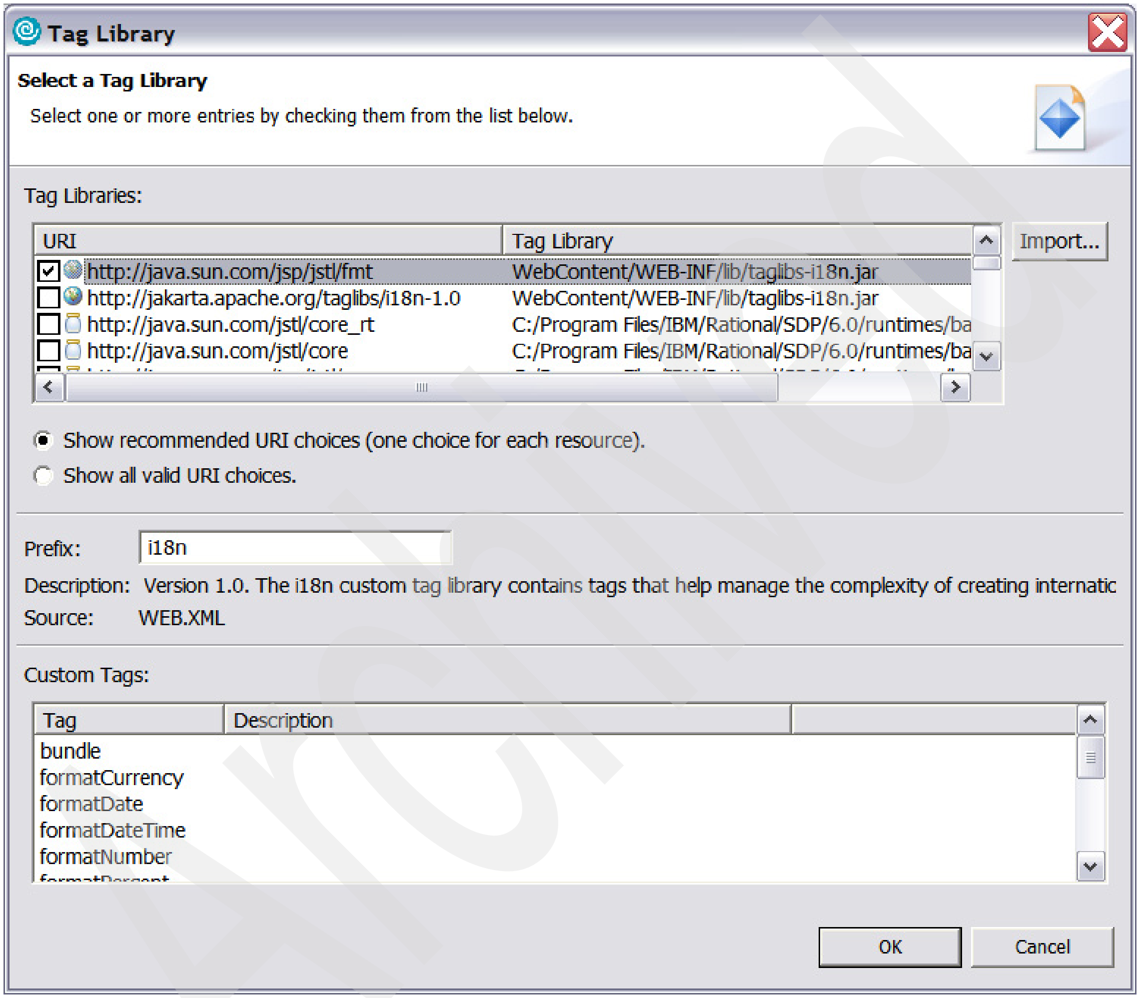


Figure 4-16 Viewing the imported tag library

9. Click **OK**. On the **New JSP** screen, click **Finish**. In the structure of the project, notice that the same directory structure recommended in the beginning of this section is used: the tag jar file is inside the `<doc-root>WEB-INF/lib` directory. See Figure 4-17.

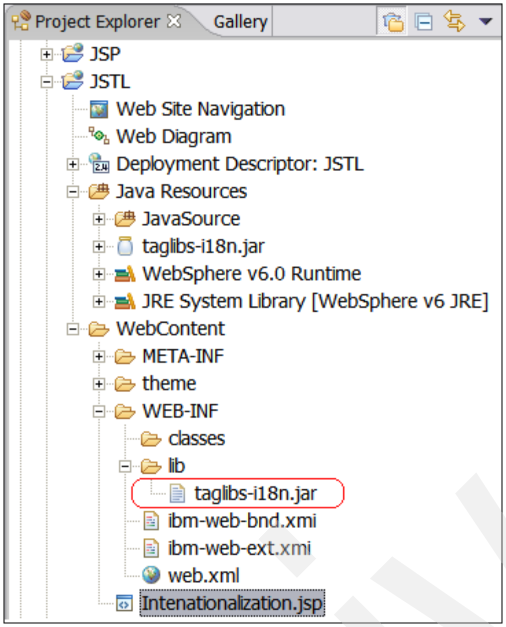


Figure 4-17 Project structure

In short, the RAD wizard for import tag libraries puts the jar in the correct structure for deployment, and puts in the jar in the classpath to compile. In addition, RAD creates the syntax in the JSP page for you:

```
<%@ taglib uri="http://jakarta.apache.org/taglibs/i18n-1.0"
prefix="i18n"%>
```

If you do not use RAD, you can do these steps manually. This means that you must put the jar in `<doc-root>/WEB-INF/lib` and put it in the classpath to compile. Also, we can refer to a tag with this syntax above. Note that if you unzip the jar, the TLD file is inside the jar in `META-INF/taglib.tld` as we described in the beginning of this session. The next step here is to do a taglib map in our web.xml config file. For this, add a <taglib> element to your Web application deployment descriptor in `/WEB-INF/web.xml` as shown in Example 4-25.

Example 4-25 taglib map

```
<jsp-config>
<taglib>
<taglib-uri>http://java.sun.com/jsp/jstl/fmt</taglib-uri>
<taglib-location>/WEB-INF/lib/taglibs-i18n.jar</taglib-location>
</taglib>
</jsp-config>
```

We can now use the taglib in our internationalization JSP example. See the code in Example 4-26.

Example 4-26 Internationalization example

```
<html>
<head>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@ page import="java.util.Date" %>
<%@ page import="java.util.Locale" %>
<title>Examples of using fmt Tag Library Tag</title>
</head>
<body>

<%
    Date d = new Date();
    Number n = new Double( 1234567.89 );
%>
<h2>Default Locale (<%= request.getLocale() %>)</h2>
<%
    Locale locale = request.getLocale();
%>
<fmt:locale locale="<%=locale%>">
<h3>The time is now:-</h3>
<ul><li><fmt:formatTime/></li></ul>
<h3>The date is now:-</h3>
<ul><li><fmt:formatDate pattern="yyyy MMMMM ddd hh:mm:ss"/></li></ul>
<h3>The currency is:-</h3>
<ul><li><fmt:formatCurrency value="<%= n %>"/></li></ul>
</fmt:locale>

<%
    String country = request.getParameter( "country" );
    if ( country == null ) {
        country = "US";
    }
    String language = request.getParameter( "language" );
    if ( language == null ) {
        language = "en";
    }
    locale = new Locale(language,country);
%>

<h2>Country: <%= country %> and Language: <%= language %></h2>
<fmt:locale locale="<%=locale%>">
```

```
<h3>The time is now:-</h3>
<ul><li><fmt:formatTime/></li></ul>
<h3>The date is now:-</h3>
<ul><li><fmt:formatDate pattern="yyyy MMMMM ddd hh:mm:ss"/></li></ul>
<h3>The currency is:-</h3>
<ul><li><fmt:formatCurrency value="<%= n %>"/></li></ul>
</fmt:locale>
</body>
</html>
```

If you execute our JSP above, the result is similar to what is shown in Figure 4-18.

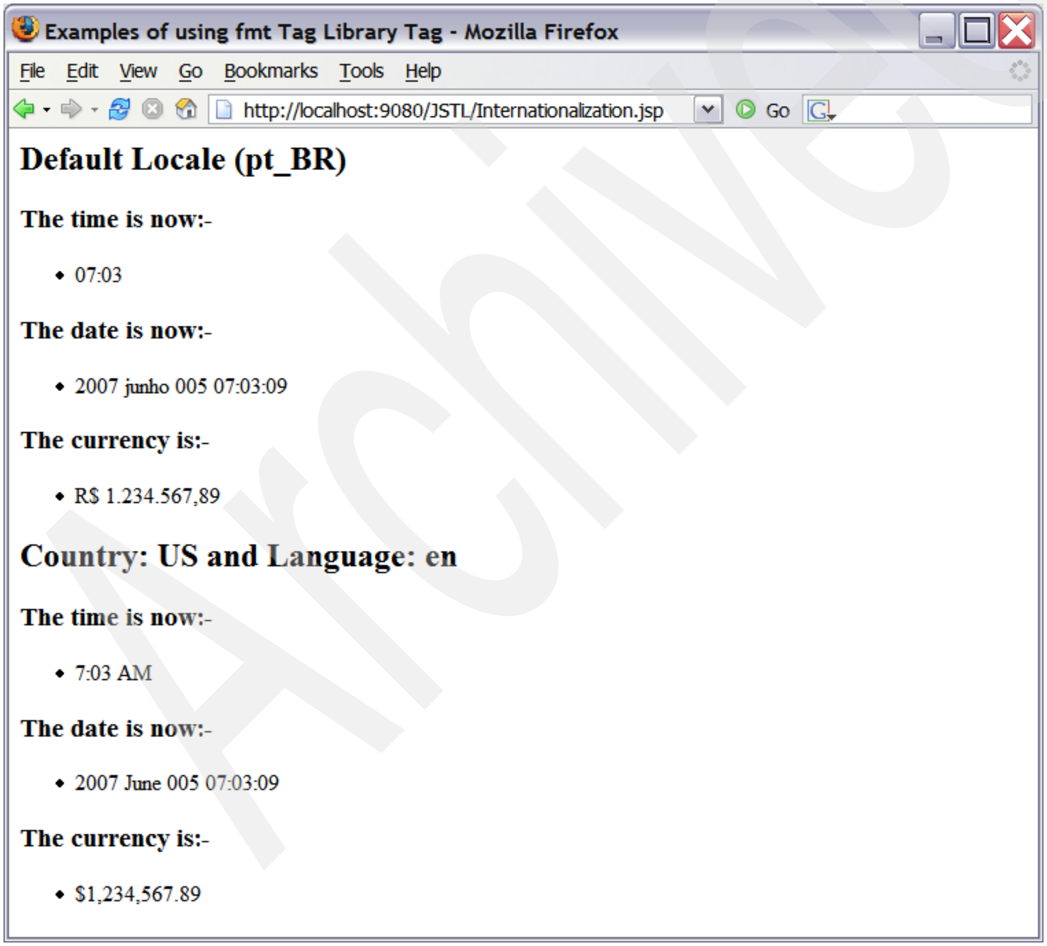


Figure 4-18 Results from JSP that uses JSTL

Implementing JSP custom tags

A custom tag library is a library that has been put together for a specific use or purpose. You create a custom tag if there is no solution available such as in JSTL. Developers working together in a team might create very specific custom tag libraries for individual projects, as well as a more general one for ongoing use.

To create a tag library, follow these steps:

1. Write the tag handler class.
2. Create the tag library descriptor (TLD).
3. Make the TLD file and handler classes accessible (packaging).
4. Reference the tag library.
5. Use the tag in a JSP page.

The last three items were covered in 4.2.9, "Tag libraries" on page 125.

You can see in 4.10, "References" on page 249 how to create you own tag library and TLD file. But more important here for best practices is that you must follow the MVC pattern when you develop a tag. The new tag support for JSP 2.0 SimpleTag makes the development very easy because you can put all tags in one method **public void doTag()**. While it is easy to develop, it can easily get out of hand if you develop a method doTag with 200 lines, for example. Remember that when you develop Java code, it is important that methods do not have to be so extensive. As recommended in the MVC pattern, it is a good practice not to access the database layer directly. Use JavaBeans to make a bridge to database access. It is also important for reusability. See Figure 4-19.

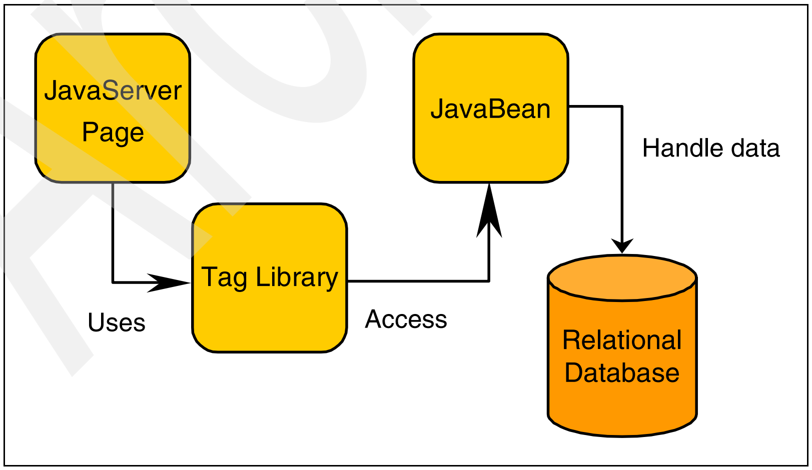


Figure 4-19 Best custom tag Library usage

4.2.10 Implicit objects

The concept behind the JSP architecture is to provide a Web component that allows developers to focus on the presentation of Web content without getting drawn into the details of parsing, programming, and data manipulation. JSP applications are essentially special Web components that a J2EE Web container converts into servlets prior to handling user requests. Within each JSP application is the complete set of implicit objects.

Implicit objects let developers access container-provided services and resources. These objects are defined as implicit because you do not have to explicitly declare them. They are defined in every JSP page and used behind the scenes by the container whether you declare them or not — although you cannot redeclare them. Because implicit objects are declared automatically, we only have to use the reference variable associated with a given object to begin calling methods on it. See the implicit objects in Table 4-8.

Table 4-8 Implicit object types

Object	Description
out	Provides access to the servlet's output stream.
request	Provides access to HTTP request data, as well as providing a context for associating request-specific data.
response	Enables direct access to the HTTPServletResponse object and is rarely used by JSP authors.
session	Is perhaps the most commonly used of the state management contexts. The concept of a "session" is that of a single user interacting with a Web application over several requests.
pageContext	Is the context for the JSP page itself. It provides a single API to manage the various scoped attributes. This API is used extensively when implementing JSP custom tag handlers.
page	Is the instance of the JSP page's servlet processing the current request.
application	Is the broadest context state available. It allows the JSP page's servlet and any Web components contained in the same application to share information.
config	Allows initialization data to be passed to a JSP page's servlet
exception	Houses exception data to be accessed only by designated JSP "error pages."

Although some implicit objects address a single function, several of them provide multiple categories of functionality:

- State management: application, session, request, pageContext
- Flow control: application, config, pageContext, request, session
- Logging and exceptions: application, config, exception, pageContext, request, session
- Input/output control: request, response, out
- Initialization parameters: config

Next we review the implicit objects that are more related to coding for performance: State management, loggin,g and exceptions. We talk more about flow control in 4.2.11, "Best practices to use composed Web components". For more information on input/output control and initialization parameters, see the Implicit JSP Objects article at the following URL:

http://www.ibm.com/developerworks/java/library/j-pj2ee7.html

State management (application, session, request, pageContext)

Four of the implicit objects defined for JSP pages can be used to associate stateful data within a particular context, or scope. Those four scopes are application, session, request, and page. Table 4-9 identifies the four objects and the stateful context each scope defines, and also gives a brief description of what each context details.

Table 4-9 Scopes and implicit objects

Scope	Implicit Object	Description
Application	javax.servlet.Servlet Context	Represents the entire runtime Web module (application). Data that is scoped to an application is shared among all the Web components within a single application module. This is the closest that J2EE offers to "global" data.
Session	javax.servlet.http.Http Session	Represents the current HTTP session. Next to page scope, session scope is the most commonly used context. This is the most commonly used context for providing a persistent, stateful user experience spanning multiple requests.

Scope	Implicit Object	Description
Request	javax.servlet.http.HttpServletRequest	Represents the current HTTP request. This context can span multiple Web components (servlets and JSP pages) that are still a part of the same atomic request. Request-specific data (request methods, URI, HTTP parameters, and so on) provided by the client is automatically stored in a request context. Data can also be programmatically scoped to a request by one servlet or JSP page to be retrieved by another within the same request scope.
Page	javax.servlet.jsp.PageContext	Represents the current JSP page's context. Because the context for a JSP page includes the current request, session, and application, a pageContext instance provides access to all the namespaces associated with a JSP page. It is the default scope for all objects, including JavaBeans components. Objects with page scope are typically bound to a local variable to be accessed within scriptlets, expressions, JavaBeans tags, and custom tags.

From a best-practices standpoint, page scope should be used whenever possible. It is simple, and it is also the default scope for JSP data. Request scope is excellent for sharing data between components at run time in order to process a particular request. Session scope is designed to provide a persistent, stateful experience for a unique user, spanning multiple requests. Application scope should be used only when you have to share data between components and across user sessions.

Logging and exceptions: application, config, exception, pageContext, request, session

If you have to store information related to your Web application in a log, there is one built-in method available to you. The ServletContext interface declares two methods for passing data to a log. One accepts a simple text message: log(java.lang.String), and the other accepts exception information and a text message: log(java.lang.Throwable, java.lang.String). However, there are many ways to obtain the ServletContext interface instance. You can use application, config, exception, pageContext, request, and session.

The implementation of the log method depends on the Web container provider. In WebSphere, both methods go to SystemOut.log that belongs to the application server. The best practice here is to be careful to log methods. Put in key points of code. To trace your code, use IDE tools such as Rational Application Developer (RAD). Avoid using logs for tracing purposes.

Another tip is the use of exceptions. Although sending messages to a log file can be quite useful, there are often times when you would also like to display a helpful user error message in the event of an unrecoverable exception. To do this, you can declare that your JSP page uses a separate page to handle error messages. This is accomplished by including the page directive anywhere in your JSP page. See Example 4-27.

Example 4-27 Allow JSP to handle exceptions in a page

```
<%@ page errorPage="ErrorMessage.jsp"%>
```

Whenever an exception is thrown while processing your JSP page, the exception object is immediately thrown to the designated error page by means of the implicitly declared exception variable.

In order for a JSP page to function as an error page, it must include a directive that declares the page to be a special page designed to handle errors, as shown in Example 4-28.

Example 4-28 Page to handle exceptions

```
<%@ page isErrorPage="true"%>
```

In order for the ErrorMessage.jsp page to be able to act as an error page, this directive must appear somewhere in the page. The error page can then display a friendly error message to the user, and then perhaps log the relevant exception information for the system administrator to review at a later time.

4.2.11 Best practices to use composed Web components

In the following sections we provide some recommendations to keep in mind when using Web components.

Static inclusion

In time-to-marketplace, we have to develop applications quickly to meet business demand. To reach these goals, use composed JSPs to optimize caching and code reuse.

In the JSP world, we can reuse the content or output of another JSP. If we are working with Static inclusion, this means that a JSP includes another JSP content at the time when the first JSP file is translated. On the other hand, Dynamic inclusion is an output of another JSP included inside an output of a requested JSP page.

The include directive makes it very easy to incorporate uniform header and footer files and navigation components into your site.

- Syntax:

  ```
  <%@ include file=”relativeURL” %>
  ```

  or

  ```
  <jsp:directive.include file=”relativeURL”/> <!--XML format-->
  ```

  Note: This relative URL cannot be a result of an expression such as:

  ```
  String relativeURL = “CelsiusCalc.jsp”;
  <%@ include file=”<%=relativeURL%>” %>
  ```

  Also you cannot pass request parameters to included jsp. We are in Translation Time not Execute time.

  ```
  <%@ include file=”other.jsp?param=model” %>
  ```

- How static inclusion works:

  In the translation time of the JSP processor, it sees the static include sentences and inserts the corresponding JSP inside the calling JSP. It appears to be a unique JSP program after the translation phase and ready for request time. See Figure 4-20 and Example 4-29.

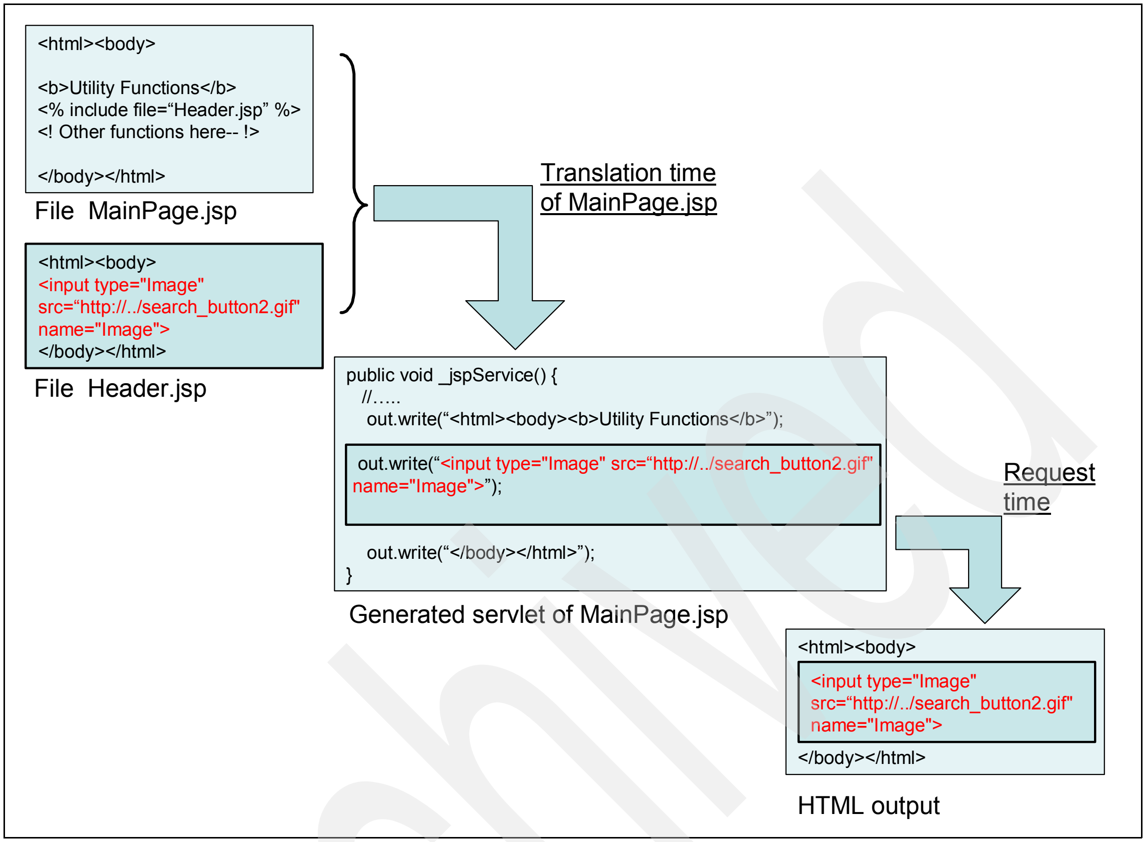


Figure 4-20 Static Inclusion functionality

Example 4-29 Static inclusion code sample

```
<![CDATA[
<%@ page language="java" contentType="text/html" %>
<html>
<head>
    <title>newInstance.com</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
    <link href="/styles/default.css" rel="stylesheet" type="text/css"
/>
</head>

<body>
```

```
<%@ include file="header.jsp" %>
<%@ include file="navigation.jsp" %>
<%@ include file="bookshelf.jsp" %>
<%@ include file="/mt-blogs/index.jsp" %>
<%@ include file="footer.jsp" %>

</body>
</html>
]]>
```

This example shows that you can make a pattern for your pages using static inclusion.

Dynamic inclusion

Dynamic content tends to change frequently and must always be up to date. In this case, there are drawbacks to using the include directive. One of the downsides of the include directive is that it causes your Web browser to cache complete pages. This makes a lot of sense if we are talking about static pages, but if included content is volatile, such as a JSP that loads dynamic data or HTML such as a time stamp, you would require the last version of this file when the Web page is loaded. The include directive does not have that functionality.

You can work around this problem by disabling the browser cache in test and development environments. However, in production where performance is a key factor, disabling caching is not a viable solution. In the production environment, the best solution is to use jsp:include tag for working with dynamic content.

- Syntax:

  ```
  <jsp:include page="filename"/>
  ```

- How dynamic inclusion works:

  The <jsp:include> action transfers the control of the request to the included JSP temporarily. When the included JSP finishes its processing, control returns to the including page. This means that the included component is interpreted and the resulting response is included. If the page is HTML, you get the HTML essentially unchanged. But if it is a Perl script, a Java servlet, or a CGI program, you get the interpreted result from that program. Because interpretation happens at every page request, the results are never cached as they were with the include directive. See Figure 4-21, Example 4-30, and Example 4-31.

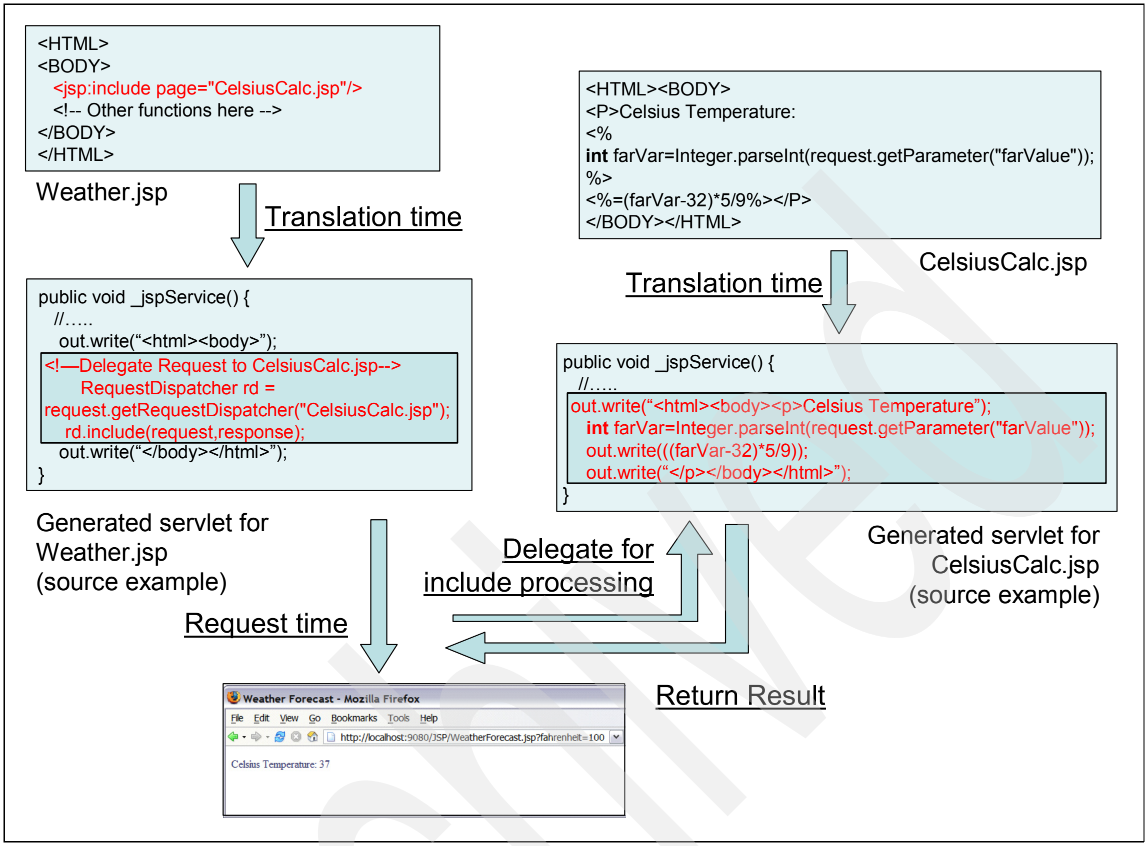


Figure 4-21 Dynamic inclusion functionality

Example 4-30 Dynamic inclusion code sample (Weather.jsp)

```
<HTML>
<TITLE>Weather Forecast</TITLE>
</HEAD>
<BODY>
<P>
   <jsp:include page="CelsiusCalc.jsp"/>
   <!-- Another Functions here -->
</BODY>
</HTML>
```

Example 4-31 Dynamic inclusion code sample (CelsiusCalc.jsp)

```
<HTML>
<TITLE>Celsius</TITLE>
```

```
</HEAD>
<BODY>
<P>Celsius Temperatute:
<%
if (request.getParameter("farValue")!=null){
int farVar=Integer.parseInt(request.getParameter("fahrenheit"));
out.println((farVar-32)*5/9);
}else{
out.println("param farVar not found, usage
http://...?fahrenheit=value");
}%>
</P>
</BODY>
</HTML>
```

The important thing here is that only body content of CelsiusCalc.jsp is returned to the main Page. The HTML Title tag, for example, remains the same from the Weather.jsp.

Dynamic caching

JSPs that are composed of several other JSPs, on one hand, frequently use the `<jsp:include>` tag and therefore offend the best practice listed previously, but on the other hand, the different components can easily be cached and re-used. So the use of caching can reduce the performance disadvantages of compositional JSPs, while facilitating the development of complex pages.

WebSphere Application Server provides a functionality called *Dynamic caching service* to cache JSPs, thereby making it possible to have a master JSP that includes multiple JSP components, each of which can be cached using different cache criteria. For example, think of a complex portal page, which contains a window to view stock quotes, another to view weather information, and so on. The stock quote window can be cached for five minutes, the weather report window for ten minutes, and so on.

If you are not planning to use Dynamic Caching, the best practice is to minimize the use of the `<jsp:include>` because each included JSP is a separate servlet.

Forwarding

The `<jsp:forward>` action delegates processing to another JSP page. The main difference from the `<jsp:include>` action is that control does not return to the including page. See Figure 4-22.

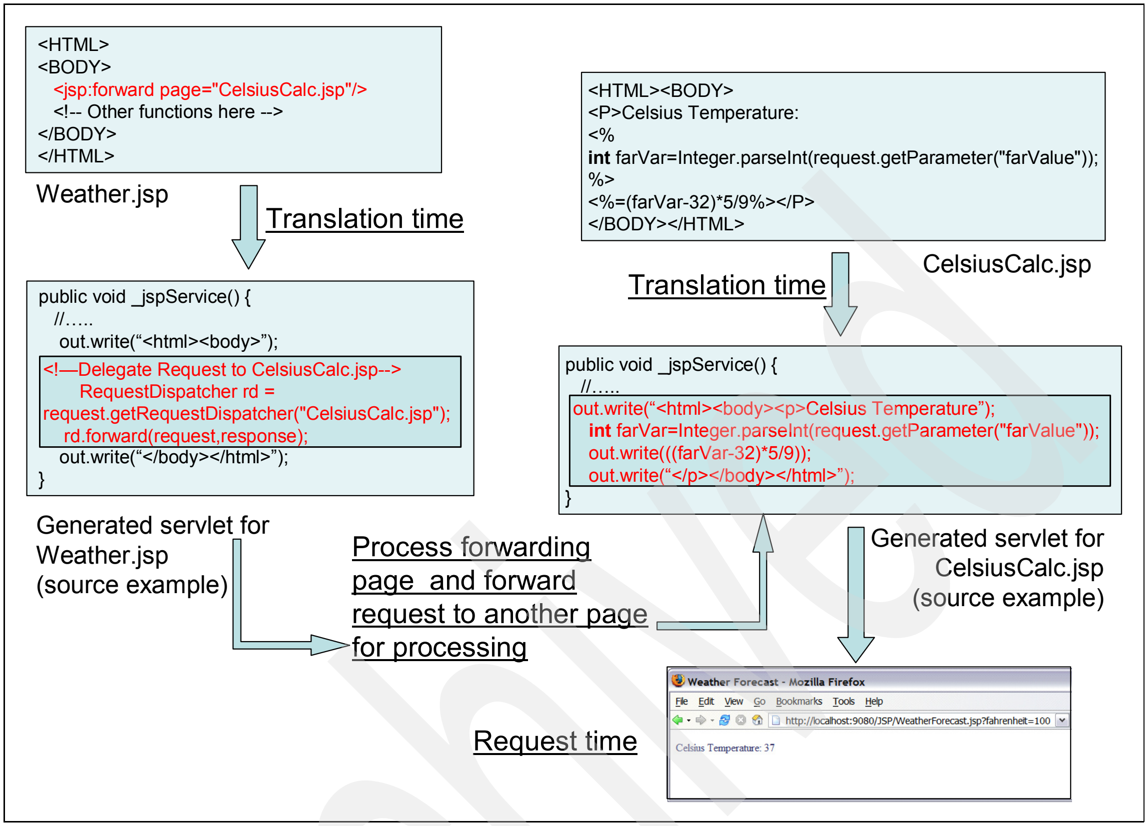


Figure 4-22 Forwarding functionality

As the page control is delegated to another page without return, the HTML layout (the page title, for example) is done by another page (CelsiusCalc in Figure 4-22).

Using JavaBeans is the best way to pass data between JSPs

If you have a simple page with a few JSPs in your project, JavaBeans might not be required. Even in this case, using JavaBeans is a good way to separate data from presentation in your Java coding. Also when it comes to programming a real Web site or Web application interface, you generally require a communication mechanism to pass data between the parent page and included files.

4.2.12 Expression Language

Expression Language (EL) is used for run-time assignment of values to action element attributes. It was first introduced as part of the JSP Standard Tag Library (JSTL) 1.0 specification, but is now part of the JSP 2.0 specification. As part of JSTL, you could only use expression language with JSTL actions. Now, as an integral part of the JSP 2.0 specification, you can use EL with template text, as well as with standard and custom actions. Expression language is inspired by both ECMAScript and XPath expression languages, and uses the features of both languages, as well as introduces some new ones. For example, expression language performs data-type conversions automatically.

The main advantage of using EL in JSPs is to enforce writing scriptless JSPs. You can do this through the configuration element `scripting-invalid`. Setting the value of this element “true” allows the use of Expression Language, but prohibits the user from using Java scriptlets, Java expressions, or Java declaration elements within JSPs.

As we discussed previously, the JSP Standard Tag Library (JSTL) is a collection of JSP 1.2 custom tag libraries that implement basic functionality common to a wide range of server-side Java applications. By providing standard implementations for typical presentation-layer tasks such as data formatting and iterative or conditional content, JSTL allows JSP authors to focus on application-specific development requirements, rather than “reinventing the wheel” for these generic operations.

Of course, you could implement such tasks using the JSP scripting elements: scriptlets, expressions, and declarations. Conditional content, for example, can be implemented using three scriptlets, highlighted in Listing 1. Because they rely on embedding program source code (typically Java code) within the page, though, scripting elements tend to complicate the software maintenance task significantly for JSP pages that use them. The scriptlet example in Listing 1, for instance, is critically dependent upon proper matching of braces. Nesting additional scriptlets within the conditionalized content can wreak havoc if a syntax error is inadvertently introduced, and it can be quite a challenge to make sense of the resulting error message when the page is compiled by the JSP container.

Example 4-32 Implementing conditional content through scriptlets

```
<% if (user.getRole() == "member")) { %>
    <p>Welcome, member!</p>
<% } else { %>
    <p>Welcome, guest!</p>
<% } %>
```

Fixing such problems typically requires a fair bit of programming experience. Whereas the markup in a JSP page might typically be developed and maintained by a designer well-versed in page layout and graphic design, the scripting elements in that same page require the intervention of a programmer when problems arise. This shared responsibility for the code within a single file makes developing, debugging, and enhancing such JSP pages a cumbersome task. By packaging common functionality into a standardized set of custom tag libraries, JSTL allows JSP authors to reduce or eliminate the requirement for scripting elements and avoid the associated maintenance costs.

For this reason, we now discuss EL concepts in more depth.

Implicit objects and EL basics

EL expressions support several implicit objects. The following table describes these objects.

Table 4-10 Implicit objects in Expression Language

Implicit object	Description
pageScope	A map of all page-scope variables and their values
requestScope	A map of all request-scoped variables and their values
sessionScope	A map of all session-scoped variables and their values
applicationScope	A map of all application-scoped variables and their values
pageContext	An object of pageContext class
param	A map of all request parameter values wherein each parameter is mapped to a single String value
paramValues	A map of all request parameter values wherein each parameter is mapped to a single String array
header	A map of all request header values wwherein each parameter is mapped to a single String value
headerValues	A map of all request header values wherein each parameter is mapped to a single String array
cookie	A map of all request cookie values wherein each cookie is mapped to a single javax.servlet.http.Cookie value
initParam	A map of all application initialization parameter values wherein each parameter is mapped to a single String value

Examples of implicit objects

Let us consider an example detailing the use of implicit objects. The table below describes how you can access and resolve information such as request attributes, session attributes, and request parameters (first column of the table) using EL (second column of the table).

Table 4-11 Examples of implicit objects

Source JSP	Expression Language sentence
request.getAttribute("name");	${requestScope["name"]}
session.getAttribute("name");	${sessionScope["name"]}
request.getParameter("name");	${param.customerName}
request.getHeader("User-Agent");	${header["user-agent"]}
request.getHeader("Host");	${header["host"]}

In the foregoing examples, User-Agent returns Browser information.

Syntax of Expression Language

The syntax of Expression Language is quite simple:

- You can use a [ ] operator to access properties of JavaBeans objects, lists, or arrays of objects.
- You can also use a . operator to access the properties of a JavaBean object.
- You can use arithmetic operators for computations.
- You can use standard Java relational operators for relational comparisons.
- Logical operators are also available for your use.
- Literals of boolean, integer, floating point, string, and null are available.
- You can also use conditional operators for conditional processing.

Next, let us discuss the above syntactic rules in detail.

Arithmetic operators

Expression Language can use the following five arithmetic operators to act on integer and floating point values:

- + operator for addition
- - operator for subtraction
- * operator for multiplication
- / operator for division
- % operator for remainder

Table 4-12 describes the arithmetic operators through some examples.

Table 4-12 Arithmetic operations examples

Expression for arithmetic operators	Result
${2.7 + 5.6}	8.3
$(-2 -7}	-9
${10%4}	2
${9/2}	4.5

Relational operators

Relational operators operate on two operands and always return a boolean result. You can use the following relational operators with EL:

- == operator for evaluating “equal condition”
- ! = operator for evaluating “are not equal condition”
- < operator for evaluating “less than” condition
- > operator for evaluating “greater than” condition
- <= operator for evaluating “less than equal to” condition
- >= operator for evaluating “greater than equal to” condition

Table 4-13 describes the relational operators through some examples.

Table 4-13 Relational operations

Expression using relational operator	Result
${10 == 2*5}	true
${10 > 4}	true
${10 < 10/2}	false
${10 <= 20/2}	true
${10 != 2*5}	false

Logical operators

Logical operators operate on two expressions and always return a boolean value. Expression Language can use the following logical operators:

- The && operator returns true if both expressions evaluate to true.
- The || operator returns true if one of the expressions evaluates to true.

- The ! operator returns the inverse of the evaluation result.

Table 4-14 describes the logical operators through some examples.

Table 4-14 Logical operations

Expression using logical operator	Result
${10 > 4 && 4 < 16}	true
${10 > 4 && 16 < 5}	false
${10 > 4 || 16 < 5}	true
${ ! (10 > 4) }	false

Conditional operator

This operator is used for conditional processing. Depending on the boolean result of the condition, one of the two possible results is returned.

The examples in Table 4-13 show conditional operator usage.

Table 4-15 Conditional operator usage examples

Expression using conditional operator	Result
${(5<6) ? 5 : 6}	5
${(5<6) ? 5 : 6}	6

Accessors ([] and . operators)

We use [] and . operators to look for a named property in a bean or in a collection. Let's consider an example where customer is the name of a bean with property SSN. To access the property SSN, you can use the following expression shown in Example 4-33.

Example 4-33 Accessor example

```
${customer['SSN']}
or
${customer["SSN"]}
```

The value within the brackets must be a string literal for the property's name, or a variable that holds the property's name. You can even use a complete EL expression that resolves to a property.

Here are a few general rules that exist while evaluating expr-a[expr-b]:

- Evaluate expr-a into value-a.
- If value-a is null, return null.
- Evaluate expr-b into value-b.
- If value-b is null, return null.
- If value-a is a map, list, or array, then evaluate whether value-b resolves to a property for it.

Using the . operator, the alternative syntax could be as shown in Example 4-34.

Example 4-34 . operator usage examples

```
${customer.SSN}
${customer.address.zip}
```

Notice that this last example uses a zip property from an address object that is an attribute of the customer object.

Combining core JSTL elements and EL sentences

To illustrate the interaction of JSTL tags with the Expression Language, we look at several of the tags from the JSTL core library. As is true with any JSP custom tag library, a taglib directive must be included in any page that you want to be able to use this library's tags. The directive for this specific library appears in Example 4-35.

Example 4-35 JSTL core directive in JSP page

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

The first JSTL custom tag we consider is the <c:set> action. As indicated, scoped variables play a key role in JSTL, and the <c:set> action provides a tag-based mechanism for creating and setting scoped variables. The syntax for this action is shown in Example 4-36, where the `var` attribute specifies the name of the scoped variable, the `scope` attribute indicates which scope the variable resides in, and the `value` attribute specifies the value to be bound to the variable. If the specified variable already exists, it is simply assigned the indicated value. If not, a new scoped variable is created and initialized to that value.

Example 4-36 <c:set> action syntax

```
<c:set var="name" scope="scope" value="expression"/>
```

The `scope` attribute is optional and defaults to page.

Two examples of the <c:set> are presented in Example 4-37. In the first example, a session-scoped variable is set to a String value. In the second, an expression is used to set a numeric value: a page-scoped variable named square is assigned the result of multiplying the value of a request parameter named x by itself.

Example 4-37 <c:set> action examples

```
<c:set var="timezone" scope="session" value="CST"/>
<c:set var="square" value="${param['x'] * param['x']}"/>
```

Rather than using an attribute, you can also specify the value for the scoped variable as the body content of the <c:set> action. Using this approach, you could rewrite the first example in Example 4-37 as shown in Example 4-38. Furthermore, as we see momentarily, it is acceptable for the body content of the <c:set> tag to employ custom tags itself. All content generated within the body of <c:set> is assigned to the specified variable as a String value.

Example 4-38 Putting a value inside <c:set> body content

```
<c:set var="timezone" scope="session">CST</c:set>
```

The JSTL core library includes a second tag for managing scoped variables, <c:remove>. As its name suggests, the <c:remove> action is used to delete a scoped variable, and takes two attributes. The var attribute names the variable to be removed, and the optional scope attribute indicates the scope from which it should be removed and defaults to page, as shown in Example 4-39.

Example 4-39 <c:remove action example>

```
<c:remove var="timezone" scope="session"/>
```

While the <c:set> action allows the result of an expression to be assigned to a scoped variable, a developer often wants to simply display the value of an expression, rather than store it. This is the role of JSTL's <c:out> custom tag, the syntax of which appears in Example 4-40. This tag evaluates the expression specified by its value attribute, then prints the result. If the optional default attribute is specified, the <c:out> action instead prints its value if the value attribute's expression evaluates either to null or an empty String.

Example 4-40 <c:out> syntax

```
<c:out value="expression" default="expression" escapeXml="boolean"/>
```

The escapeXml attribute is also optional. It controls whether or not characters such as “<“, “>”, and “&”, which have special meanings in both HTML and XML, should be escaped when output by the <c:out> tag. If escapeXml is set to true, then these characters are automatically translated into the corresponding XML entities (<, >, and &, respectively, for the characters mentioned here).

For instance, suppose there is a session-scoped variable named user that is an instance of a class that defines two properties for users, username and company. This object is automatically assigned to the session whenever a user accesses the site, but the two properties are not set until the user actually logs in. Given this scenario, consider the JSP fragment shown in Example 4-41. Once the user has logged in, this fragment displays the word “Hello,” followed by his or her username and an exclamation point. Before the user has logged in, however, the content generated by this fragment is instead the phrase, “Hello Guest!” In this case, because the username property has yet to be initialized, the <c:out> tag instead prints out the value of its default attribute (that is, the character string, “Guest”).

Example 4-41 <c:out> action with default content

```
Hello <c:out value="${user.username}" default=="Guest"/>!
```

Next, consider Example 4-42, which uses the <c:out> tag's escapeXml attribute. If the company property has in this case been set to the Java String value “Flynn & Sons”, then the content generated by this action, in fact, is Flynn & Sons. If this action is part of a JSP page generating HTML or XML content, then the ampersand in the middle of this string of characters might end up being interpreted as an HTML or XML control character and interrupt the rendering or parsing of this content. If the value of the escapeXml attribute is instead set to true, however, the generated content instead is Flynn & Sons. A browser or parser encountering this content should have no problems with its interpretation. Given that HTML and XML are the most common content types in JSP applications, it should come as little surprise that the default value for the escapeXml attribute is true.

Example 4-42 <c:out> action with escaping disable

```
<c:out value="${user.company}" escapeXml=="false"/>
```

In addition to simplifying the display of dynamic data, the ability of <c:out> to specify a default value is also useful when setting variable values through <c:set>. As highlighted in Example 4-38 on page 164, the value to be assigned to a scoped variable can be specified as the body content of the <c:set> tag, as well as through its value attribute. By nesting a <c:out> action in the body content of a <c:set> tag, the variable assignment can leverage its default value capability.

This approach is illustrated in Example 4-43. The behavior of the outer <c:set> tag is straightforward enough: it sets the value of the session-scope timezone variable based on its body content. In this case, however, that body content is generated through a <c:out> acon. The value attribute of this nested action is the expression ${cookie['tzPref'].value}, which attempts to return the value of a cookie named tzPref by means of the cookie implicit object. (The cookie implicit object maps cookie names to corresponding Cookie instances, which means you must use the dot operator to retrieve the actual data stored in the cookie through the object's value property.)

Example 4-43 Combining <c:set> and <c:out> to provide default variable values

```
<c:set var="timezone" scope=="session">
   <c:out value="${cookie['tzPref'].value}" default=="CST"/>
</c:set>
```

Consider the case, however, in which this is the user's first experience with the Web application using this code. As a result, there is no cookie named tzPref provided in the request. This means that the lookup using the implicit object returns null, in which case the expression as a whole returns null. Since the result of evaluating its value attribute is null, the <c:out> tag instead outputs the result of evaluating its default attribute. Here, this is the character string CST. The net effect, then, is that the timezone scoped variable is set to the time zone stored in the user's tzPref cookie or, if none is present, use a default time zone of CST.

Differences between scriptlets and ELs in one example

To observe the improvements that can be achieved by using EL instead of scriptlets, consider an example of using jsp:useBean with scriptlets and the updated version changing scriptlets to EL and core JSTL tag libraries (see Example 4-44 and Example 4-45).

Example 4-44 jsp:useBean with scriptlets example

```
<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<TITLE>WelcomeScriptlet.jsp</TITLE>
</HEAD>
<BODY>
<jsp:useBean id="user" scope="session" type="somepackage.PersonBean"
class="somepackage.EmployeeBean"></jsp:useBean>

<%if(user.getFirstName()==""){%>
```

```
  <P>Welcome, guest !</P>
 <%}else{%>
  <P>Welcome <%user.getFirstName();%></P>
 <%}%>
</BODY>
</HTML>
```

Example 4-45 Example with EL and JSTL instead of scriptlets

```
<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<TITLE>WelcomeEL.jsp</TITLE>
</HEAD>
<BODY>
<jsp:useBean id="user" scope="session" type="somepackage.PersonBean"
class="somepackage.EmployeeBean"></jsp:useBean>

<c:set var="user" scope="session" value="${sessionScope['user']}"/>
<c:choose>
<c:when test="${user.firstName == ''}">
    <p>Welcome, guest!</p>
  </c:when>
  <c:otherwise>
    <p>Welcome,${user.firstName}!</p>
  </c:otherwise>
</c:choose>
</BODY>
</HTML>
```

As can be seen, the conditional if (<c:when>) is much simpler and easier to mix HTML tags with EL instead of HTML tags with scriptlets. Now consider a JSP with thousands of lines of code. With scriptlets it is more difficult to maintain and understand both from a page design and from a Java programmer point of view.

4.2.13 Use composed JSPs to optimize caching and code re-use

On the one hand, JSPs that are composed of several other JSPs frequently use the `<jsp:include>` tag and therefore offend the best practice listed below; but on the other hand, their different components can easily be cached and re-used. So the use of caching can reduce the performance disadvantages of compositional JSPs, while facilitating the development of complex pages.

WebSphere Application Server provides a functionality called *Dynamic caching service* to cache JSPs, thereby making it possible to have a master JSP that includes multiple JSP components, each of which can be cached using different cache criteria. For example, think of a complex portal page, which contains a window to view stock quotes, another to view weatherinformation, and so on. The stock quote window can be cached for five minutes, the weather report window for ten minutes, and so on.

4.2.14 Best practices summary for JSPs

Here we provide a list of best practices for working with JSPs:

- Prefer JSPs as your first choice of presentation technology.

  See 4.2.1, "Use JSPs as your first choice of presentation technology" on page 110
- Avoid using scripting elements.

  See 4.2.7, "Scripting elements: Best practices" on page 117
- Use existing Tag Libraries.

  See 4.2.9, "Tag libraries" on page 125
- Create custom Tag Libraries.

  See "Implementing JSP custom tags" on page 147
- Use implicit objects in JSP pages.

  See 4.2.10, "Implicit objects" on page 148
- Use composed JSPs.

  See 4.2.11, "Best practices to use composed Web components" on page 152
- Use EL (Expression Language) instead of scripting elements.

  See 4.2.12, "Expression Language" on page 158
- Combine expression languages with tag libraries.

  See "Combining core JSTL elements and EL sentences" on page 163

4.3 XML/XSLT processing

With the help of the Extensible Stylesheet Language Transformation (XSLT), which is a part of the XSL standard, you can easily transform any XML-based document into another. In the presentation layer, this technology is usually used to generate an XML-based document that is recognized by a browser, like HTML, XHTML or other XML-based documents that can be understood by other client devices such as handheld devices. To do this, XSLT first parses the source XML document to determine which parts have to be transformed. The parts that should be transferred, and the manner of transformation, are pre-defined in one or more templates. So if a matching part is found, XSLT transforms this part of the source document into the resulting XML document. All other parts that do not match any of the templates remain unmodified and are simply copied into the result. See Figure 4-23.

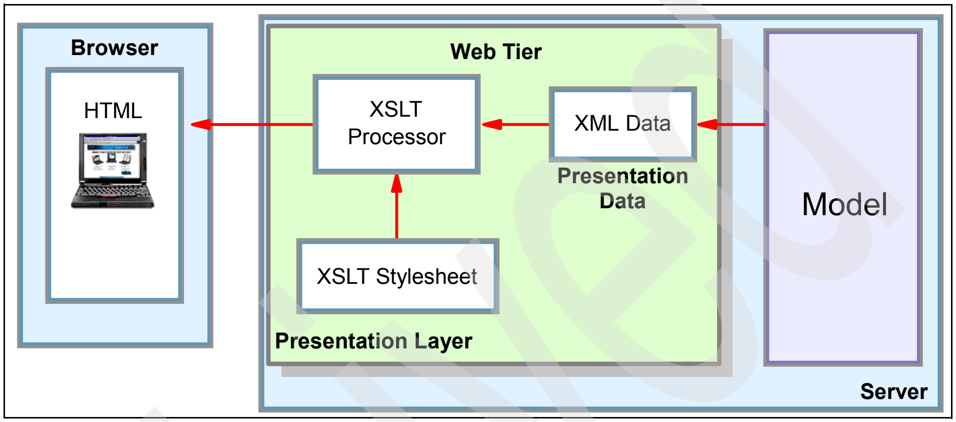


Figure 4-23 Server-sided XSLT processing

In general, the XSLT transformation can take place either on the application server or in a browser/client that supports XSLT. We only cover the server-side XSLT processing, because XSLT processing on the client is not possible in most cases due to security reasons, network traffic, or limited browser support. A valid compromise could be to implement a servlet that checks the client for an XML-enabled browser and returns the XML directly to the client to get client-side XSLT. Server-side XSLT is only done when the client is not XML-enabled.

4.3.1 Server-sided XSLT processing

The use of XML/XSLT processing on the application server to generate the view of your application implies a lot of processing overhead and, from a performance point of view, it is therefore only recommended in cases where you really have multiple presentation output types that must be supported. Performance tests done at IBM comparing the relative speed of XSL and JSP show that in most cases, a JSP is several times faster at producing the same HTML output as an equivalent XSL transform, even when compiled XSL is used.

While this is often not an issue, in performance-critical situations, it can create problems. This does not mean that you should never use XSL, but it might not be the recommended way to generate the view for high volume Web sites. However, there are certain cases, especially in the pervasive computing sector, where XSLT might be the best/easiest solution for rendering the views. Here the power and the abilities of using XSL to support multiple mobile devices countervail the processing overhead by far.

But this kind of requirement is most often the exception rather than the rule. If you are using XSLT just for producing the HTML rendering for each page, then this is overkill and causes more problems for your developers than it can solve.

If you decide to use XSLT processing on the server, you should use the following best practices to minimize the performance impact:

- Use Extensible Stylesheet Language Transformation compiled (XSLTC), the compiled version of XSLT, whenever possible

  XSLTC directly compiles the stylesheet into a Java class and is therefore the much faster alternative. It is about three times faster than the XSLT interpreter.

- Keep XSL stylesheets as simple as possible; use XML Data Transfer Object

  It is a common best practice to generate a value object to transfer the data between the business logic layer and the presentation layer. The same should be done using XSLT. The business logic should return a Data Transfer Object (or SDO) that contains the XML representation of the data required in the view. There is no requirement to have any complicated logic in the XSL stylesheet to generate or collect the XML data that is required for the view. The generation of an XML Data Transfer Object really simplifies the stylesheet processing and greatly improves performance.

4.4 Control layer

Whatever the presentation technology, requests for domain state and behavior are be done through a Controller object defined for the particular presentation requirements. See Figure 4-24.

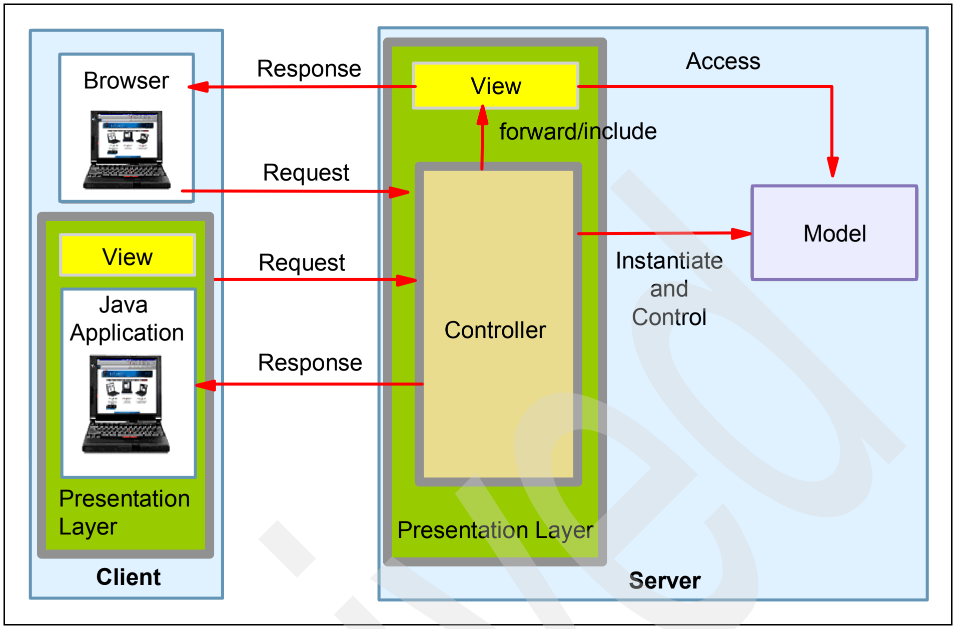


Figure 4-24 Showing controller component

For Web-based applications, the best practice is that Java Servlets must be used to implement this work. If you are using a framework, like Struts or JavaServer Faces (JSF), controller objects based on Java servlets are already defined to handle the incoming client requests. These frameworks provide a clean separation between the presentation and the control objects, so if you decide not to use one, you must take care of this yourself.

For server-based applications that support “fat clients,” meaning clients that have to be installed on every client machine, the implementation of the controller object depends on the communication protocol that is used. If you are using HTTP or HTTPS, for example, SOAP/HTTP or SOAP/HTTPS, to communicate with the application server, Java servlets is the best practice choice to control incoming requests. If RMI/IIOP or JMS is your preferred protocol to send requests to the application server, Enterprise JavaBeans, more precisely EJB session beans, are the best fit to handle incoming RMI/IIOP traffic and EJB message-driven beans for JMS.

We discuss both Struts (see 4.6, “Struts” on page 199) and JavaServer Faces (see 4.7, “JavaServer Faces” on page 220). The heart of the Struts framework is ActionServlet (org.apache.struts.action.ActionServlet). The heart of the JSF framework is FacesServlet (javax.faces.webapp.FacesServlet). In JSF we note more statements that JSP could be changed by another presentation technology.

4.4.1 General best practices for the control layer

Irrespective of the technology that is used to control the incoming requests, observe the following guidelines.

Keep controller objects as simple as possible

The controller should neither contain any business logic, nor generate any kind of presentation. Its purpose is just mediation between the presentation and the business logic layer and conversion from one interface to another. It can be thought of as a pipe between presentation and logic layer. When an entry arrives, the controller knows to what component it requires to be redirected. if you are using Struts for example, the framework has already defined in the architecture as ActionServlet. A worst case example is when there are thousands of code lines in one servlet for a doGet or doPost method containing controller, business logic and data access layer logic. With this approach it is very difficult to make your code reusable and also is very complicated to find which layer an error is in. When you mix up layers, you mix up exceptions.

Do not put your control layer in the client

If you are using a Java client to access the server remotely, be careful to not code the controller function in the client. The reason is simple: If you have to add some corrections or additional function in your controller, you have to update the logic in each client. The Java client should contain only the presentation code and the components that must call the controller. In those components, and the controller itself, use value objects to make a loose coupling between layers.

4.5 Servlets

As a server-side entity in a J2EE architecture, servlets are an effective player in Web programs. In this section we focus on important performance tips to help you create well constructed code in base statements.

4.5.1 General best practices for servlets

Observe the following guidelines when using servlets.

Abstract parent class for all servlets

Consider creating an abstract parent class, using the template inheritance pattern, for all your controller servlets (and Struts action handlers) to realize a common behavior. This abstract class is a good place to put standard code that you want executed in *all* servlets, such as tracing, logging or additional security.

Avoid the “killer” servlet

Use multiple servlets instead of one massive one. This avoids undue routing logic and also allows you to configure J2EE security for individual servlets. This guideline also simplifies the maintenance and team development process.

4.5.2 HttpSession best practices

Because the protocols used for communication between the client and the application server are usually based upon a request/response model and therefore stateless, the application developer has to take care of the state himself. For example, if a request is submitted from the browser to the application server, the server just receives the request, process it, and send a response back. After this transaction is complete, there is no way for the protocol to hold state information about the transaction itself. Therefore, this state information has to be stored either on the client or on the server.

In a Web application, state information relating to each client is typically stored in an HTTP session, which is identified by some unique identifier that is associated with an HTTP cookie. In an environment with a single application server, session information can be stored in-memory by IBM WebSphere Application Server V6.

However, it is more common to use a clustered environment with multiple application servers to provide scalability and improve fault tolerance. In this scenario, session information has to be made available for multiple or even all cluster members. In WebSphere Application Server V4.x and earlier, this was achieved using a session persistence database that was available to all clones in a server group. In addition to this, a new mechanism for memory-to-memory replication was introduced in IBM WebSphere Application Server V5.0.

In general, HTTP sessions might better fit your requirements if you are building systems that only require a Web front end. There is an alternative to HttpSession: stateful session beans.

Stateful session beans

A stateful session bean is used to capture state information that must be shared across multiple consecutive client requests that are part of a logical sequence of operations. The client must obtain an EJB object reference to a stateful session bean to ensure that it is always accessing the same instance of the bean.

WebSphere Application Server currently supports the clustering of stateful session bean home objects among multiple application servers. However, it does not support the clustering of a specific instance of a stateful session bean. Each instance of a particular stateful session bean can exist in just one application server and can be accessed only by directing requests to that particular application server. State information for a stateful session bean cannot be maintained across multiple application server cluster members. Thus, stateful session bean instances cannot participate in WebSphere workload management.

One significant improvement introduced in WebSphere Application Server V6 is the failover support for stateful session beans, which means that the state information maintained by a stateful session bean can survive various types of failures now. This is achieved by utilizing the functions of the Data Replication Service (DRS) and server workload management (WLM).

However, HTTPSession is the preferred way to store session information.

The following guidelines are important to ensure performance and scalability while using HTTP sessions.

Keep HTTP sessions small

HTTP sessions should only be used to store information about application state; it is not a data cache! You should always try to minimize the amount of data stored in the session. Since the session must be shared, it must be serialized, which also involves serializing all objects that are reachable from the session. It means that all objects have to implement java.io.Serializable. Serialization in Java is an expensive operation. If persistent sessions are used, the serialized session data must be stored in the database (usually as a BLOB), which introduces further overhead.

WebSphere Application Server has HttpSession configuration options that can optimize the performance impact of using persistent HttpSessions. The HttpSession configuration options are discussed in Chapter 7, “Environmental performance considerations” on page 509. Also consider alternatives to storing the entire servlet state data object graph in the HttpSession.

Figure 4-25 compares the relative performance of a sample application with a single object of different sizes. As the size of the objects stored in the HttpSession increases, throughput (requests by second) decreases, in large part due to the serialization cost.

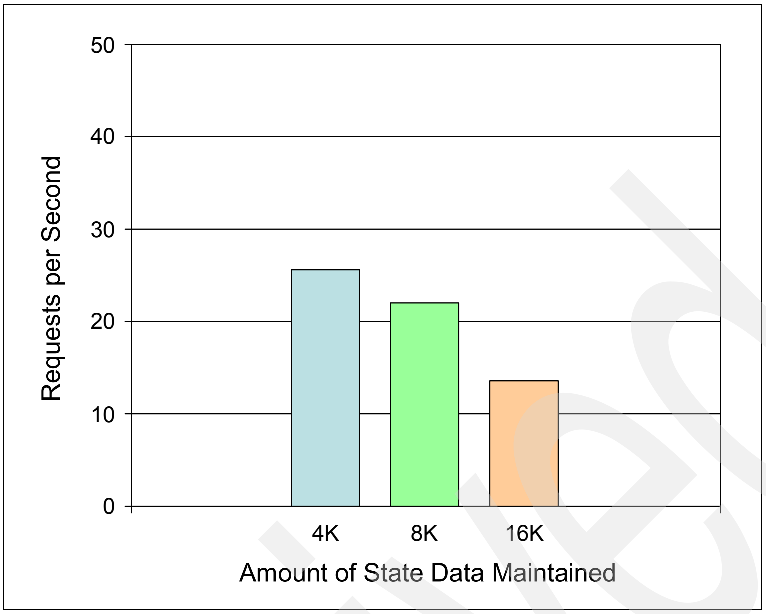


Figure 4-25 Throughput relation with Http Session data size

For example, assume that a given application stores 1 MB of information for each user session object. If 100 users arrive over the course of 30 minutes, and we assume that the session timeout remains at 30 minutes, the application server instance must allocate 100 MB just to accommodate the newly arrived users in the session cache:

1 MB for each user session * 100 users = 100 MB

Note that this number does not include previously allocated sessions that have not timed out yet. The memory required by the session cache could be considerably higher than 100 MB.

In short, you should keep those HTTP sessions small. If you do not, your application's performance can suffer. A good rule of thumb is something under 2K-4K. This is not a hard rule. 8K is still okay, but obviously slower than 2K. Just watch this value and prevent the HttpSession from becoming a dumping ground for data that “might” be used. When using HttpSessions, store only as much state as you require for the current business transaction and no more.

Improving HttpSession performance with smart serialization

There are a lot of alternatives possible to try to reduce the data to be placed in HttpSession. Some approaches are too complex and the programmer forgets that Http session servers to keep the state between requests are not a persistent mechanism itself.

Application servers go to great lengths to ensure that session management is efficient, but there are limits to what they can do without application domain knowledge. Recall that there are three basic approaches to keeping session state valid across multiple clustered application servers:

- Storing session objects in a shared database:

  When configuring persistent sessions in a shared database in WebSphere Application Server, use a dedicated data source. To avoid contention for JDBC connections, do not reuse an application data source or the WebSphere Application Server repository for persistent session data.

- Using memory to memory replication state between clusters:

  For WebSphere 6.1, this approach is done using the data replication service available in distributed server environments.

- Using session affinity:

  The Servlet 2.4 specification requires that an HTTP session be:

  – Accessible only to the Web application that created the session. The session ID, but not the session data, can be shared across Web applications.

  – Handled by a single JVM for that application at any one time. In a clustered environment, any HTTP requests associated with an HTTP session must be routed to the same Web application in the same JVM. This ensures that all of the HTTP requests are processed with a consistent view of the user's HTTP session. The exception to this rule is when the cluster member fails or has to be shut down.

    WebSphere is able to assure that session affinity is maintained in the following way: Each server ID is appended to the session ID. When an HTTP session is created, its ID is passed back to the browser as part of a cookie or URL encoding. Then, when the browser makes further requests, the cookie or URL encoding is sent back to the Web server. The Web server plug-in examines the HTTP session ID in the cookie or URL encoding, extracts the unique ID of the cluster member handling the session, and forwards the request.

All objects placed in HttpSession have to implement the java.io.Serializable interface. You can use a technique to choose which objects really have to be placed in HttpSession. This technique is called smart serialization. The attributes that do not have to be placed in HttpSession are put in a transient qualifier. See Example 4-46 for more explanation.

Example 4-46 Code sample using smart serialization

```
public class EmployeeBean implements java.io.Serializable
{
```

```
            private String employeeId;
            private String employeeName;
            private transient AddressBean employeeAddress;

            /**
             Return the cached employee address. If it hasn't been
             fetched, retrieve it from the DAO layer.
            */

             public AddressBean getEmployeeAddress() {
                 if (employeeAddress == null) {
                   EmployeeDAO dao = new EmployeeDAO();
                   employeeAddress =
                      dao.fetchAddressFor(getEmployeeId());
                   }
                 return employeeAddress;
             }

             public String getEmployeeId() {
                 return employeeId;
             }

             public String getEmployeeName() {
               return employeeName;
             }

             public void setEmployeeAddress(Address employeeAddress) {
               this.employeeAddress = employeeAddress;
             }

             public void setEmployeeId(String employeeId) {
               this.employeeId = employeeId;
             }

             public void setEmployeeName(String employeeName) {
               this.employeeName = employeeName;
             }

      }
```

What happens is that in normal circumstances, each of the transient fields (for example, Address) are retrieved once and then cached in memory in the Employee object contained in the HttpSession. That means the Web application code has fast access to them. However, in the event of a failure or termination of that application server, when the application server “fails over” to another JVM, the Employee object read back in contains (initially) those fields that were non-transient, which are the fields necessary to reconstruct the other transient fields. This technique solves several problems elegantly:

- All application code that uses the HttpSession is completely unaware of what is managed persistently and what is transient. No more abstraction concerns.
- The HttpSession can contain as many objects as you like as long as most of the data is transient. That is, the amount of data persisted is small and this results in good performance. Of course, sessions cannot be so large that you run out of memory.
- Since failover is rare, the data in transient fields are rarely reloaded. In the event of a failover, there is a slightly increased response time, but all succeeding accesses are fast because the values are fetched directly out of memory.
- As a side benefit, by using lazy instantiation techniques, user data that is not read by the application is never fetched from the back end.

Notice that the code above is a value object (VO). Because you have attributes and accessors (sets and gets for access attributes), and the transient address object when requested (lazy instantiation), the code asks the Data Access Object (DAO) pattern to fetch data from the database, and you can see the Model View Controller pattern here. Inside this code there is no direct access to the database, which is a also a best practice.

Improving HttpSession performance with externalization

The operations of the storing/recovering state, which are based on object serialization, can be improved by using Java externalization. Externalization can be up to 40% faster than serialization. Nevertheless, in order to reduce the amount of data stored in the session, avoid storing large, complex object graphs in it. Sometimes, it can be beneficial to store objects in the session, although they can be recreated or retrieved to avoid the overhead of doing so. In these cases, consideration should be given to making these attributes transient. If this is done, you have to ensure that the application code handles the transient attributes having null values. Alternatively, the readObject() method of the object could be overwritten to recreate the transient data when the object is deserialized. Again, Java externalization can be used to eliminate introspection and improve performance.

To understand how to use this approach, consider a more complex object graph with an EmployeeBean that extends a PersonBean. The PersonBean has an attribute called AddressBean and uses a PersonDAO class to fetch data from the database. See Figure 4-26.

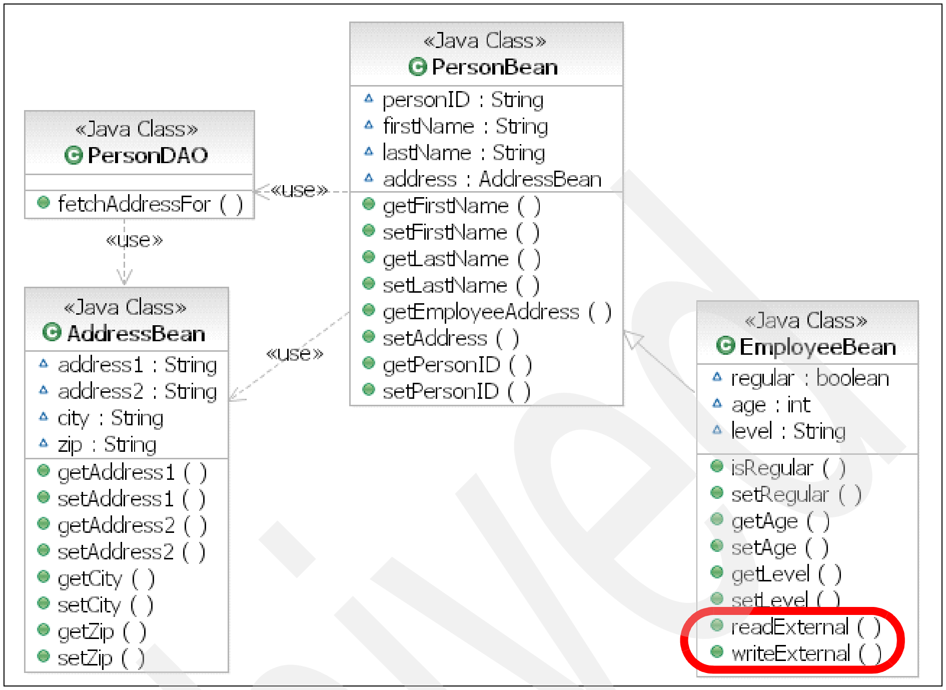


Figure 4-26 Class diagram for data to use externalization approach

EmployeeBean is the class that we have to place in HttpSession. The class we used implements java.io.Serializable in EmployeeBean. For this reason you can see methods readExternal() and writeExternal() in this class. Now our job is to choose what data we have to keep in session. Remember that we can choose data from the PersonBean superclass. For the PersonBean class, you can see the access to database data controlled by a Data Access Object (DAO) pattern. For more details on DAO, refer to “Data Access Object” on page 43.

See Example 4-47 showing how we did an EmployeeBean class implementation.

Example 4-47 EmployeeBean class implementation

```
package somepackage;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class EmployeeBean extends PersonBean implements Externalizable{
```

```
boolean regular;
int age;
String level;

public boolean isRegular() {
    return regular;
}
public void setRegular(boolean regular) {
    this.regular = regular;
}
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
public String getLevel() {
    return level;
}
public void setLevel(String level) {
    this.level = level;
}
public void readExternal(ObjectInput in) throws IOException,
        ClassNotFoundException {
    //read super class fields
    personID = (String) in.readObject();
    firstName = (String) in.readObject();
    lastName = (String) in.readObject();
    /*
      if superclass implements Serializable
      only that you need is putting super sentence
      Exmple: readExternal(out);
    */
    // now we take care of this subclass's fields
    age = in.readInt();
    regular = in.readBoolean();
    level = (String)in.readObject();
}
public void writeExternal(ObjectOutput out) throws IOException {
    //write super class fields
    out.writeObject(personID);
    out.writeObject(firstName);
    out.writeObject(lastName);
    /*
      if superclass implements Serializable
```

```
          only that you need is putting super sentence
          Exmple: super.writeExternal(out);
        */
        // now we take care of this subclass's fields
        out.writeInt(age);
        out.writeBoolean(regular);
        out.writeObject(level);
    }
}
```

Note that to persist, we are using out.writeXXX, and for read data, in.readXXX. Concerning transient data, remember that if you have to recreate transient data, you can use lazy instantiation. This means that data is returned when called. HttpSession usage in the servlet code does not change the externalization approach. See the code snippet in Example 4-48.

Example 4-48 Session usage with externalization

```
HttpSession https = request.getSession();
if (https.isNew())
{
     //This a only an example
     //Because you need to catch the employee information
     //from somewhere
     EmployeeBean employee = new EmployeeBean();
     employee.setFirstName("john");
     employee.setLastName("smith");
     employee.setPersonID("1234");
     employee.setAge(34);
     employee.setLevel("Senior");
     https.setAttribute("employeeData",employee);
}
```

Enable security integration for securing HTTP sessions

HTTP sessions are identified by session IDs. A session ID is a pseudo-random number generated at runtime. Session hijacking is a known attack on HTTP sessions and can be prevented if all the requests going over the network are enforced to be over a secure connection (meaning, HTTPS). But not every configuration in a customer environment enforces this constraint because of the performance impact of SSL connections. Due to this relaxed mode, an HTTP session is vulnerable to hijacking, and because of this vulnerability, WebSphere Application Server has the option to tightly integrate HTTP sessions and WebSphere Application Server security.

You should enable security integration in the session management component of WebSphere Application Server so that the sessions are protected in a manner that only users who created the sessions are allowed to access them.

See Chapter 10. Session Management in *WebSphere Application Server V6.1: System Management and Configuration*, SG24-7304 for more information.

Always invalidate unused HTTP sessions

The session object can be garbage collected after it has been invalidated. This can be done programmatically or after a predefined time-out period during which the session was not accessed. To allow the memory used by the session to be reclaimed as early as possible, it is best to explicitly invalidate the session when finished with it rather than waiting for the time-out. This might require the introduction of logout functionality into the application, and training for the users to make use of this functionality rather than simply closing the browser.

HttpSession objects live inside the WebSphere servlet engine until:

- The application explicitly and programmatically releases them using the API, javax.servlet.http.HttpSession.invalidate ().
- WebSphere Application Server destroys the allocated HttpSession when it expires (by default, after 1800 seconds or 30 minutes). You can set up this feature in web.xml for each Web application putting timeout in minutes (Example 4-49).

  Example 4-49 Session timeout application configuration in web.xml file

  ```
  <web-app>
    <!--Others parameters -->
    <session-config>
      <session-timeout>5</session-timeout>
    </session-config>
  </web-app>
  ```

- WebSphere Application Server can only maintain a certain number of HttpSessions in memory.

  However, be careful about where you put this procedure. It is more common to put it in a Logout Servlet. In statistical terms, a percentage of users use the logout button of a Web program and another portion closes the browser. You can alleviate database unused sessions by following the procedure shown in Example 4-50.

Example 4-50 Programatic HttpSession invalidation

```
   package somepackage;
   import java.util.*;
```

```
   import javax.servlet.*;
   import javax.servlet.http.*;

   public class ApplicaitonLogOutServlet extends HttpServlet
   {
       public void doGet(HttpServletRequest request,
   HttpServletResponse response) throws     ServletException,
   IOException
   {
       HttpSession mySession = request.getSession(false);
       if (mySession != null)
       {
         // Invalidate the Session Here !!!!!
         mysession.invalidate();
         // Invalidate the Session Here !!!!!
   }
   //------------------------------------------------------------------
   //
   // Some other Application Logoff Processing and Output Reply Back
   // to Browser
   //
   //------------------------------------------------------------------
         }
}
```

- Use manual update and either the sync() method or time-based write if necessary.

  This approach is used in applications that read session data, and update infrequently. With END_OF_SERVICE as write frequency, when an application uses sessions and anytime data is read from or written to that session, the LastAccess time field updates. If database sessions are used, a new write to the database is produced. This activity is a performance hit that you can avoid using the Manual Update option and having the record written back to the database only when data values update, not on every read or write of the record.

  To use manual update, turn it on in the session management service. Additionally, instead of the generic HttpSession, the application code must use the `com.ibm.websphere.servlet.session.IBMSession` class. Within the IBMSession object, there is a sync method. This method tells the WebSphere Application Server to write the data in the session object to the database. This activity helps the developer to improve overall performance by having the session information persist only when necessary.

An alternative to using the manual updates is to utilize the timed updates to persist data at different time intervals. This action provides similar results as the manual update scheme.

Do not cache references to HTTP sessions

References to the session should always be obtained from the current servlet context as required; they should not be cached by the application. This ensures that the session objects can be reclaimed when the session is invalidated. Never use static or instance variables to refer a session. Get a session from a servlet method. Also avoid putting static objects in a session. See Example 4-51.

Example 4-51 BadServlet example

```
package somepackage;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.SingleThreadModel;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class BadServletExample extends HttpServlet implements
SingleThreadModel {

   static HttpSession wrongHttpSessionVariable;
   static Object wrongObjectToPutInSession;

   public BadServletExample() {
      super();
   }

   protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
   {
      wrongHttpSessionVariable = request.getSession(true);

wrongHttpSessionVariable.setAttribute("badObject",wrongObjectToPutInSes
sion);
   }
}
```

Take care when using HTML frames

Special care must be taken when using HTML frames when each frame is displaying a JSP belonging to a different Web application on the same server. In this case, a session should only be created and accessed by one of the pages. Otherwise, although a session is created for each page, the same cookie is used to identify the session. This means that the cookie for each newly created session overwrites the previous cookie, and only one of the sessions is accessible. The remaining sessions are created but are inaccessible and thus consume memory until the time-out interval is reached. If the Web application was split into multiple applications in order to improve scalability, consider combining all of the Web applications into a single one, and using clustering to achieve the required scalability.

When using multi-framed pages, follow these guidelines:

- Create a session in only one frame or before accessing any frame sets. For example, assuming there is no session already associated with the browser and a user accesses a multi-framed JSP file, the browser issues concurrent requests for the JSP files. Because the requests are not part of any session, the JSP files end up creating multiple sessions and all of the cookies are sent back to the browser. The browser honors only the last cookie that arrives. Therefore, only the client can retrieve the session associated with the last cookie. We recommend creating a session before accessing multi-framed pages that utilize JSP files.
- By default, JSP files get an HTTPSession using request.getSession(true) method. So by default, JSP files create a new session if none exists for the client. Each JSP page in the browser is requesting a new session, but only one session is used per browser instance. A developer can use <% @ page session="false" %> to turn off the automatic session creation from the JSP files that do not access the session. For more information on this JSP setup, see 4.2.5, “Directives” on page 114. If the page requires access to the session information, the developer can use the following coding to get the already existing session that was created by the original session creating the JSP file:

  ```
  <%HttpSession session =
  javax.servlet.http.HttpServletRequest.getSession(false); %>
  ```

  This action helps to prevent breaking session affinity on the initial loading of the frame pages.
- Update session data using only one frame. When using framesets, requests come into the HTTP server concurrently. We recommend modifying session data within only one frame so that session changes are not overwritten by session changes in the concurrent frameset.

Avoid using multi-framed JSP files where the frames point to different Web applications. This action results in losing the session created by another Web application because the JSESSIONID cookie from the first Web application gets overwritten by the JSESSIONID created by the second Web application.

Do not use hidden form fields or cookies for session purposes

In some cases, there are programs that use hidden form fields or cookies to store data. Note that there is a 4 KB limit on the total size of all cookies for a particular site and it can be difficult if you have to use cookies for other features inside your application. Also, be aware that the use of hidden fields increases the page size and the data can be seen by the user when viewing the HTML source. This is reason enough to not use this approach.

Other alternatives

Defer persistence of session data to the business logic layer.

Data can also be persisted into a database by the business logic. By using native data types instead of serialized BLOBs, it is often possible to achieve better performance. It is also possible to read and write only the data that has changed, rather than the entire data set as is normally the case with BLOBs. The application must remove data when it is no longer required (after a time-out period). This can be implemented by placing an object that implements the `HttpSessionBindingListener` interface into the session, and placing the clean up code in the `valueUnBound()` method.

For example, you can do a wrapper database class using Data Access Object pattern to a database. A good point that can be explored is that a key of database can be retrieved using HTTPSession getId() method. With this, the key value from a client request does not have to be propagated between requests. Because you do not have automatic cleanup of this data in the database (you are not using data in an HTTPSession), a cron job running to clear old session data with a time criterion is an important choice.To do this, be careful with transaction controls to access data to avoid locks in database rows when the cron deletes the data. Also keep in mind that for each access to data, you have to do a update the "last access" time field. This approach causes a lot of updates in the database. Refer to "Data Access Object" on page 43.

See the following URLs for examples of improving HttpSession performance with a sample DB solution.

http://www-128.ibm.com/developerworks/websphere/library/bestpractices/store_objects_in_httpsession.html

http://www.ibm.com/developerworks/websphere/library/bestpractices/store_objects_in_httpsession.html

4.5.3 Use webcontainer approaches to create/deliver resources

The following sections provide alternatives to how a webcontainer can be used to create and deliver resources.

Use the contextInitialized() method

The `contextInitialized()` method can be used to initialize resources that are to be used for all servlets in an application. This method is called by a Web container before a Web application can be ready for requests. To use this feature, make sure your class implements javax.servlet.ServletContextListener interface.

Use the servlet.init() method

The `javax.servlet.Servlet.init()` method can be used to perform expensive operations that must be performed once only, rather than using the `doGet()` or `doPost()` methods of the servlet. By definition, the `init()` method is thread-safe. The results of operations in the `HttpServlet.init()` method can be cached safely in servlet instance variables, which become read-only in the servlet service method. A typical use for this would be to cache any JNDI lookups, for example, to EJBs or to data sources.

Use the HttpServlet destroy() method

As you used the `init()` method for expensive operations and to cache some data as JNDI lookups, you should use the `destroy()` method to release these resources to avoid memory leaks. This method gives you an opportunity to clean up any resources that are being held (for example, memory, file handles, threads) and make sure that any persistent state is synchronized with the servlet's current state in memory. But remember that the destroy method is not called in the end of a servlet execution or when the servlet instance is destroyed by the container. Example 4-52 shows using the init(), destroy(), and contextInitialize() methods.

Example 4-52 initialization resources best practice code

```
package somepackage;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.Servlet;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;

public class Initialization extends HttpServlet implements Servlet ,
ServletContextListener {
   public void init(){
       //Put your initialization code here to initialize resouce that
will be seen by
       //this servlet.
       //you can use instance variables
   }

   protected void doGet(HttpServletRequest arg0, HttpServletResponse
arg1) throws ServletException, IOException {
       //Put your coding here
   }

   public void destroy(){
       //Put your deliver resouces procedures here.
       //But only when servlet is destroyed this method will be called
   }

   public void contextInitialized(ServletContextEvent arg0) {
       //Here you can put a resource that will be seen by all Servlets
       Map countryList = new HashMap();
       //Initialize countryList
       ServletContext appContext = arg0.getServletContext();
       appContext.setAttribute("countryList",countryList);
   }
   public void contextDestroyed(ServletContextEvent arg0) {
      //Here you can release a web application resouce
   }
}
```

4.5.4 Compose your servlets

You can compose servlets with other servlets or other JSPs as we have seen in 4.2.11, "Best practices to use composed Web components" on page 152. There are two methods:

- Forward: This transfers an execution of a request to another servlet or JSP and does not return to the forwarding servlet.
- Include: The including servlet gains execution control after the included servlet or JSP finishes execution.

In Example 4-53, you can see that a servlet called Forward delegates processing to another JSP.

Example 4-53 ForwardServlet example

```
package somepackage;
import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ForwardServlet extends HttpServlet implements Servlet {

      public ForwardServlet() {
         super();
      }

      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
         try{
                request.setAttribute("message",new
PresentationMessageBean(0));
            }
            catch(Exception e){
                request.setAttribute("message",new
PresentationMessageBean(1035,e));
            }
            RequestDispatcher rd =
getServletContext().getRequestDispatcher("jspResponse.jsp");
            rd.forward(request,response);
         }
}
```

In Figure 4-27, the jspResponse is in the same Web application doc-root.

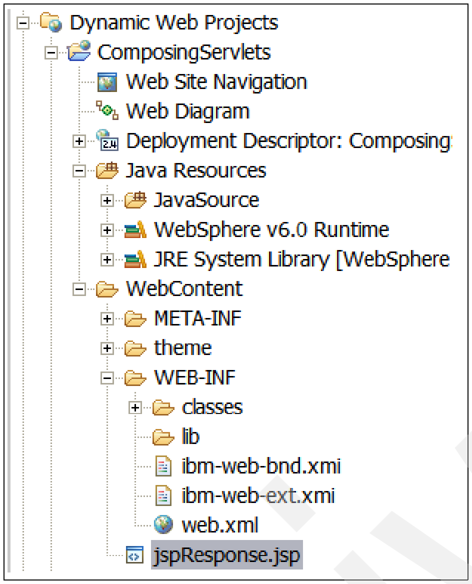


Figure 4-27 Location of jspResponse

You can also call another servlet or JSP in another Web application. The unique requisite is that the Web application that has to be found must be available in the same Java Virtual Machine that the calling Web application is in.

For this example, we use the ServletContext class to get another Web context, and after that, to get RequestDispatcher. See Example 4-54.

Example 4-54 include another Web resource from another Web application

```
protected void frameworkCall (ServletContext context,
HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
   PrintWriter writer = response.getWriter();

   writer.write("<HTML>");
   writer.write("<HEAD>");
   writer.write("</HEAD>");
   writer.write("<BODY>");

   //Include intestmentSummary.jsp resource placed
   //in another web application with context root /investiments
   RequestDispatcher rd = getRequestDispatcher ( context,
"/investments", "/investmentSummary.jsp");
```

```
   rd.include(request, response);

   //Include the accountSummary.jsp placed in anono aplicativo da Web
   //in another web application with context root /banking.
   rd = getRequestDispatcher ( context, "/banking",
"/accountSummary.jsp");
   rd.include(request, response);

   writer.write("</BODY>");
   writer.write("</HTML>");

}
private RequestDispatcher getRequestDispatcher (ServletContext context,
String contextName, String resource) {
   return
context.getContext(contexName).getRequestDispatcher(resource);
}
```

In this example you pass the object ServletContext that can be reached using the .getServletContext() where this is the including servlet.

Remote Request Dispatcher

Remote Request Dispatcher (RRD) is a WebSphere Application Server extension that can be connected to a Web container, which allows a Servlet or a JSP to include another resource in another JVM (Java Virtual Machine). See the following URL:

http://wm-live.world.mii-streaming.net/live/cbcnm/pull/1

Search for Remote Request Dispatcher.

4.5.5 Avoid presentation layer servlets

We recommend that you avoid presentation layer coding in servlets. Use JSPs to present data. Servlet presentation code creates a binding between the page designer and Java programmer that makes it difficult to maintain and divide work in different teams. Also, problem determination is more difficult. Continuing from Example 4-53 on page 189, we can see that for a call from ForwardServlet:

```
RequestDispatcher rd =
getServletContext().getRequestDispatcher("jspResponse.jsp");
rd.forward(request,response);
```

We call the jspResponse.jsp that contains the presentation logic. See Example 4-55.

Example 4-55 jspReponse.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
       pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<jsp:useBean id="message" scope="request"
type="somepackage.PresentationMessageBean">
</jsp:useBean>
<c:set var="messageTitle" scope="request"
value="<%=message.getPresentationMessageTitle()%>"/>
<c:set var="message" scope="request"
value="<%=message.getPresentationMessage()%>"/>

<TITLE><c:out value="${messageTitle}" default="Default Title"/></TITLE>
</HEAD>
<BODY>
<c:out value="${message}" default="Processing OK"/>
</BODY>
</HTML>
```

4.5.6 Implement thread safe servlets

First we give you an overview of how a Web container works: A Web container receives many requests from browsers in a production enviroment. When a container receives a request for a servlet, a thread called the dispatcher thread picks up another thread from a pool to assist the request. The threads inside the pool, called worker threads, allow multiple requests to be served concurrently.

The goal is to receive a request from the dispatcher thread and call a servlet instance for processing the service method of the desired servlet that was called from the URL. If a dispatcher thread receives another request and the first worker thread is busy with a servlet instance, it takes another worker thread to process request, no matter if a request is to the same servlet that was called in the first request. Hence, if a request is to the same servlet, the service() method of the requested servlet is called two times: one for each worker thread — but the servlet instance is the same. See Figure 4-28.

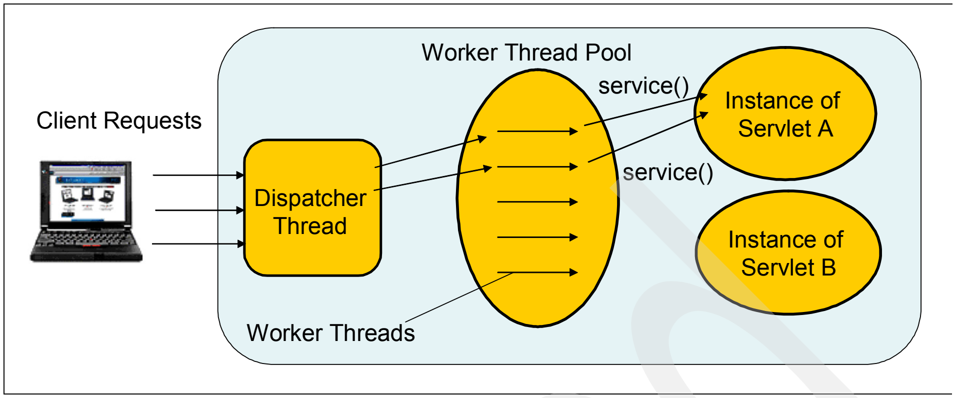


Figure 4-28 Servlet Container Worker Threads functionality

For example, if you use an inside service() method or the called methods such as doGet and doPost, a Thread.sleep method, it does not affect the response performance for a unique servlet. This is because if you execute each service() method in each thread, then each thread handles your Thread.sleep() command separately.

Do not use SingleThreadModel

The normal functionality of a container improves performance to do tasks in parallel. When you use the SingleThreadModel interface in a servlet, the servlet container handles servlets either by synchronizing access to a single instance of the servlet, or by maintaining a pool of servlet instances and dispatching each new request to a free servlet, depending of container implementation. If synchronizing the servlet service() method is used, this approach reduces the throughput, and consequently increases the response times experienced by users. If the pool of servlet instances is used, the same problem is experienced because the number of Objects can increase hugely.

The WebSphere servlet engine handles the servlet's reentrancy problem by creating separate servlet instances for each user. Because this causes a great amount of system overhead, you should avoid SingleThreadModel. In addition, with the Servlet 2.4 specification that is included in J2EE 1.4 and by consequence in WebSphere Application Server 6.1, SingleThreadModel interface is deprecated. If a servlet has shared variables that have to be protected, it is preferable to do so using synchronization of the relevant accesses, as we see in “Avoiding or minimizing synchronization in servlets” on page 196. The coding is shown in Example 4-56.

Example 4-56 SingleThreadModel bad usage example

```
public class BadServletExample extends HttpServlet implements
SingleThreadModel
```

See Figure 4-29 to understand SingleThreadModel functionality.

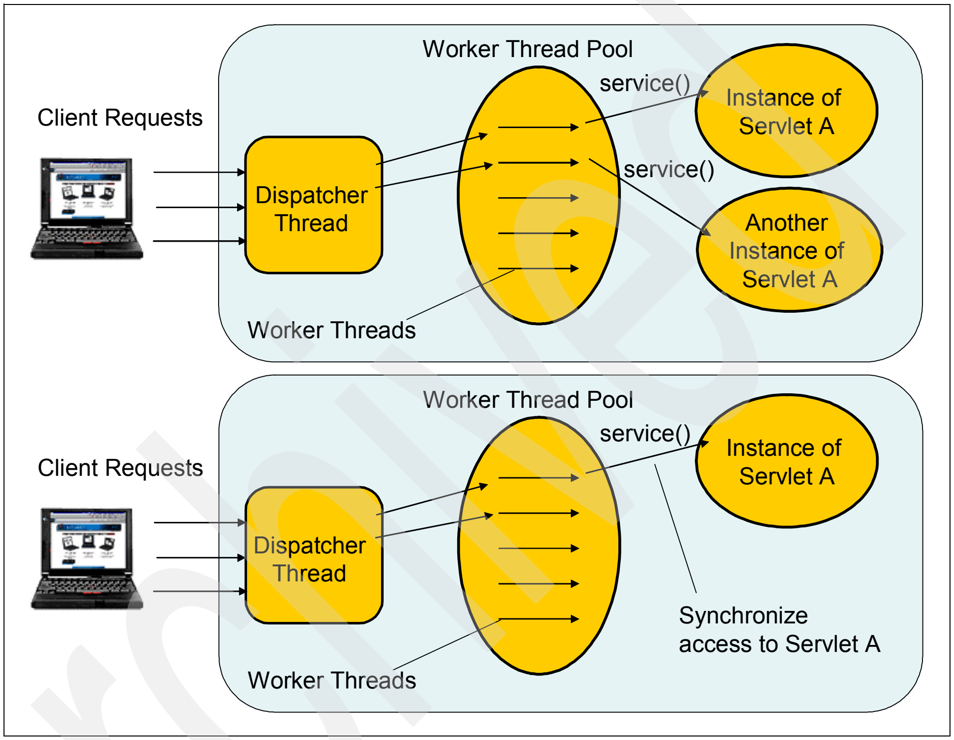


Figure 4-29 Two types of single thread model functionality inside container

Best practices for variables usage in servlets

The correct usage of servlet scope variables in your code can make a huge difference in both performance and runtime errors. A thread safe variable means that if the object variable is accessed from multiple threads, the state is consistent after these calls, independent of processing order. However, a thread unsafe variable can be inconsistent depending on the thread schedule sequence of processing. In any Java class, the servlet has three variable types:

- Class or Static variables:

  Class variables can exist in any instances of a class. They are not thread safe, so do not use them for volatile data such as user information. In short, use static variables only for constants and read-only data.

- Instance variables:

  Instance variables can be used for all service() servlet methods executed from threads. They are not thread safe, so do not use them for volatile data such as user information. A common usage is for operations that consume processing and remain in all servlet life cycles as a connection created by a DAO (Data Access Object) mechanism. Put this processing in a servlet.init() method for one servlet or contextInitialized() method that is executed when a class implements a ServletContextListener interface. See section 4.5.3, “Use webcontainer approaches to create/deliver resources” on page 187.

- Local variables:

  Local variables are unique variables that are thread safe because they exist in a method execution. This means that each service() entry point method and subsequent calls to doGet or doPost, if used, has your set of variables per each WorkerThread that is calling. Also, they are eligible to be garbage collected at the end of execution. The recommendation is to use local variables when possible.

If you have to keep local variables live after execution of a servlet, you can use request, session, or context scope. See Table 4-16.

Table 4-16 Scope variables usage

Scope	Usage
request	This is used to keep data live when you use composed Web components, for example, to a dispatch or forward commands.
session	This is used to keep data live between different requests, for example, in a shopping application.
context	This is used to keep data live between all servlets in a Web application, for example, for an application’s configuration data.

You can see an example of variables best practice in Example 4-57.

Example 4-57 Variables best practice example

```
package somepackage;

import java.io.*;
import javax.servlet.http.*;

public class ThreadSafeServlet extends HttpServlet
{
   //Class Variables usage
   static final int STORE_CODE = 10;
```

```
   static final String DEFAULT_GENERAL_MESSAGE = "General failure, try
Again !";

   //Instance Variables usage
   public DAOAccessor dao;

   public void init(){
        dao = new DAOAccessor();
        dao.initializeConnections();
   }

   public void doGet(HttpServletRequest request, HttpServletResponse
response){
        //Local variables usage
        String accountNumber = "";
        try{
            accountNumber = request.getParameter("account");
        }
        catch(Exception e){}
   }
}
```

Avoiding or minimizing synchronization in servlets

No synchronization in servlets presents the best option, because if large sections of code are synchronized, an application effectively becomes single threaded, and throughput decreases dramatically. However, if the application design cannot avoid synchronization, then use a "Lock Object" and lock the smallest possible code path. Do not synchronize the servlet service method or the doGet and doPost methods. These methods are the major code paths. Synchronizing these methods or any of the servlet methods locks the entire servlet instance. Example 4-58 shows an example using a "Lock Object" to protect the servlet instance variable numberOfRows.

Example 4-58 MinimizeSynchronizationServlet example

```
public class MinimizeSynchronizationServlet extends HttpServlet
{
private int numberOfUsers = 0;

private Object lockObject = new Object();

  public void doGet(HttpServletRequest request, HttpServletResponse
response)
           throws ServletException, IOException
```

```
  {
           //Some code here...

           synchronize(lockObject){
             numberOfUsers = getAllUsersConnected() + 1;
           }
           try {
             //Another code here
           }
           catch (Exception es){
             // Error handling code here
           }
   }
}
```

Be careful when you use more than one mutex variable (lock Object). The code in Example 4-59 could be right, but is not.

Example 4-59 Bad usage of mutex objects in synchronized blocks

```
public class BadMutexUsage extends javax.servlet.http.HttpServlet
 implements javax.servlet.Servlet {

 public String mutex = "";
 public String mutex2 = "";

 protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
  synchronized (mutex) {
    //Your code here
  }
  synchronized (mutex2) {
    //Your code here
  }
 }
}
```

The mutex and mutex2 Strings are the same object because they were initialized with the same constant "". The object is returned from a pool of unique strings. Be wary of this example. Use Objects to do synchronization instead. It is simpler and you do not have to use a specialized class such as String (shown in Example 4-59) to do this approach.

Another interesting point is that if even you are using primitive variable types in two separate command lines, there is no guarantee that they are thread safe. See Example 4-60.

Example 4-60 ThreadNotSafeServlet example

```
public class ThreadNotSafeServlet extends HttpServlet
{

 int a = 0;
 int b = 0;

 public void doGet(HttpServletRequest request, HttpServletResponse
response){

   //Local variables usage
   try{
         a++;
         b = a + 1;
     }
     catch(Exception e){}
   }
}
```

This servlet could cause problems. If a worker thread T1 executes the line a++ and at a later time another worker thread T2 executes the same line a++, the results are unpredictable. Then the best practice is shown in Example 4-61.

Example 4-61 ThreadSafeServlet using correct locking for primitives

```
//Instance Servlet variable
private Object lockObject = new Object();

public void doGet(HttpServletRequest request, HttpServletResponse
response)                 throws ServletException, IOException
{
           //Some code here...
           synchronize(lockObject){
               a++;
               b = a + 1;
           }
}
```

For more details on synchronization techniques, see on Chapter 3, “General coding considerations” on page 57.

4.5.7 Optimizing service

When you code your `service()` method, or doGet or doPost, and there is a requirement to write to an output stream, avoid the use of PrintWriter to send binary data, because there is overhead in PrintWriter usage. PrintWriter is for use with character output streams and it encodes data to bytes. For binary data, use ServletOutputStream.

Also, do partial flushes with the flush method of ServletOutputStream and PrintWriter class when there is a lot of data to be sent to the client. Otherwise, the client might think that the servlet is not responding and might perform many unnecessary requests.

4.5.8 Use a framework that implements a controller

If you are planing to develop a controller that implements an MVC pattern, consider using a framework that does it for you. Do not reinvent the wheel! Struts offers a solid and tested framework.

4.6 Struts

Struts is an open source framework you can use to build Web applications and is based on the popular Model View Controller (MVC2) design paradigm. The Struts framework control layer uses technologies such as servlets, JavaBeans, and XML. The view layer is implemented using a JSP approach as a foundation and enhancing it with a powerful custom tag library that speeds up Web application development and implements the concept and the advantages of a Model View Controller based architecture. Its tag libraries provide features such as parsing and validation of user input, error handling, and internationalization support. The Struts architecture encourages the implementation of the concepts of the Model View Controller architecture pattern. By using Struts, you can get a clean separation between the presentation and business logic layers of your application.

4.6.1 Model View Controller model 2 pattern with Struts

In 2.2.2, "Model View Controller" on page 28, we described general concepts and architecture of the MVC pattern. But remember that the difference between Model View Controller (MVC) model 1 and model 2 is that there is always a controller that dispatches requests from clients and select views. So all calls from one JSP to another are not done directly. Calls pass to a controller that decides the JSP for the next view, for example.

Figure 4-30 depicts Struts components in relation to the MVC pattern:

- Model: Struts does not provide model classes. The business logic must be provided by the Web application developer as JavaBeans or EJBs.
- View: Struts provides action forms to create form beans that are used to pass data between the controller and view. In addition, Struts provides custom JSP tag libraries that assist developers in creating interactive form-based applications using JSPs. Application resource files hold text constants and error message, translated for each language, that are used in JSPs.
- Controller: Struts provides an action servlet (controller servlet) that populates action forms from JSP input fields and then calls an action class where the developer provides the logic to interface with the model.

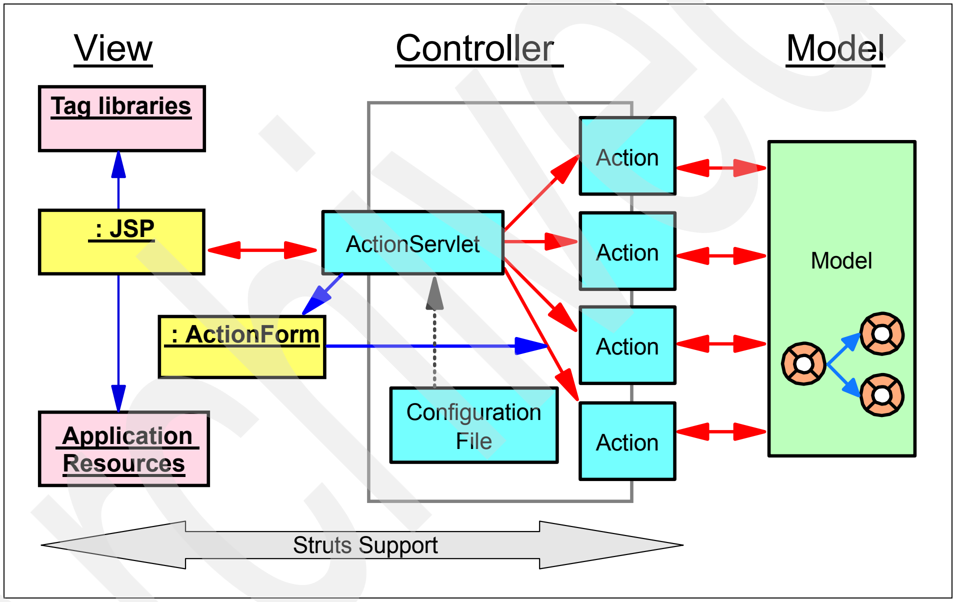


Figure 4-30 Struts components in the MVC architecture

A typical Struts Web application is composed of the following components:

- A single servlet (extending org.apache.struts.action.ActionServlet) implements the primary function of mapping a request URI to an action class. Before calling the action class, it populates the form bean associated to the action with the fields from the input JSP. If specified, the action servlet also requests the form bean to validate the data. It then calls the action class to carry out the requested function. If form bean validation fails, control is returned to the input JSP so the user can correct the data. The action servlet is configured by an XML configuration file that specifies the environment and the relationship between the participating components.
- Multiple JSPs that provide the end-user view. Struts includes an extensive tag library to make JSP coding easier. The JSPs display the information prepared by the actions and request new information from the user.
- Multiple action classes (extending any one of the Struts action classes such as org.apache.struts.action.Action) that interface with the model. When an action has performed its processing, it returns an action forward object, which determines the view that should be called to display the response. The action class prepares the information required to display the response, usually as a form bean, and makes it available to the JSP. Usually the same form bean that was used to pass information to the action is used also for the response, but it is also common to have special view beans tailored for displaying the data. An action forward has properties for its name, address (URL), and a flag specifying if a forward or redirect call should be made. The address to an action forward is usually hard coded in the action servlet configuration file, but can also be generated dynamically by the action itself.
- Multiple action forms (extending any one of the Struts Action Form classes like org.apache.struts.action.ActionForm) to help facilitate transfer form data from JSPs. The action forms are generic Javabeans with getters and setters for the input fields available on the JSPs. Usually there is one form bean per Web page, but you can also use more coarse-grained form beans holding the properties available on multiple Web pages (this fits very well for wizard-style Web pages). If data validation is requested (a configurable option), the form bean is not passed to the action until it has successfully validated the data. Therefore the form beans can act as a sort of firewall between the JSPs and the actions, only letting valid data into the system.
- One application resource file per language supported by the application holds text constants and error messages and makes internationalization easy.

Figure 4-31 shows the basic flow of information for an interaction in a Struts Web application.

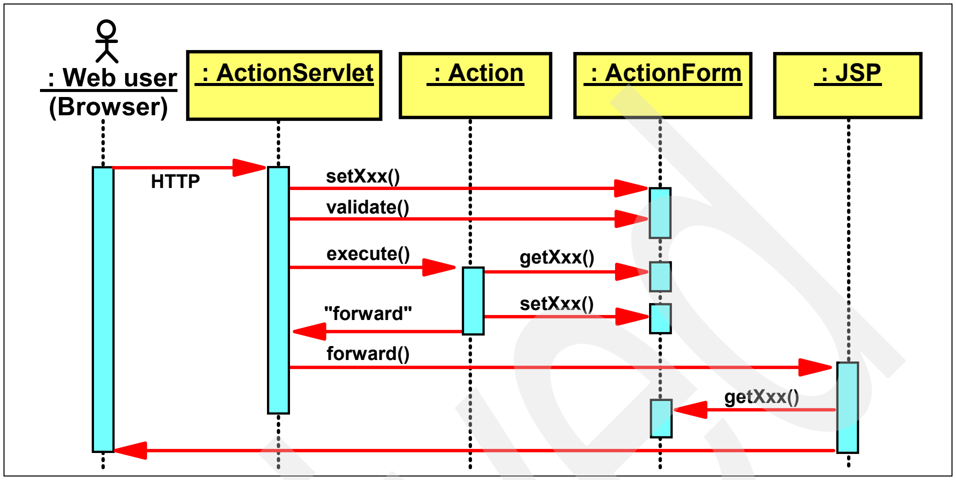


Figure 4-31 Struts request sequence

A request from a Web browser reaches the Struts ActionServlet. If the action that handles the request has a form bean associated with it, Struts creates the form bean and populates it with the data from the input form. It then calls the validate method of the form bean. If validation fails, the user is returned to the input page to correct the input. If validation succeeds, Struts calls the action's execute method. The action retrieves the data from the form bean and performs the appropriate logic.

Actions often call session EJBs to perform the business logic. When done, the action either creates a new form bean (or other appropriate view bean) or reuses the existing one, populates it with new data, and stores it in the request (or session) scope. It then returns a forward object to the Struts action servlet, which forwards to the appropriate output JSP. The JSP uses the data in the form bean to render the result.

4.6.2 General performance considerations

In general, the Struts Framework has proven its performance and scalability. It is widely used and it also enforces the Model View Controller architecture, which is a big advantage compared to JSP-servlet based applications. Struts makes heavy use of JSP tag libraries, which results in a small performance overhead. This overhead can be easily reduced by using the dynamic caching framework of IBM WebSphere Application Server V6, which now includes support for the Struts framework, including tiles.

4.6.3 Overview of Struts components

First, we explain the Struts components in the context of best practices and the role each one plays in your Web application development.

Action

Every Action of your application extends Struts' org.apache.struts.action.Action. These Action classes provide an interface to the application's Model layer, acting as a wrapper around the business logic. The goal of an Action class is to process a request, via its execute method, and return an ActionForward object that identifies where control should be forwarded (such as a JSP, Tile definition, Velocity template, or another Action) to provide the appropriate response. For it, each Action class must provide its case-specific implementation to the perform() method. The perform() method always returns a value of type ActionForward.

To be more specific, the Action class defines two methods that could be executed depending on your servlet environment (Example 4-62).

Example 4-62 Two method types for action

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
ServletRequest request, ServletResponse response) throws Exception;

public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws
Exception;
```

ActionForm

Every ActionForm of your application extends Struts' org.apache.struts.action.ActionForm. ActionForms are simple JavaBeans that encapsulate and validate request parameters. To validate your request data, your ActionForm's validate() method must give a case-specific implementation. ActionForms serve as a carrier of request data to the Action class. A JSP object combines with a respective ActionForm to form your application's View layer, where almost every form field of the JSP object maps to an attribute of the corresponding ActionForm (Example 4-63).

Example 4-63 ActionForm example

```
package somepackage.forms;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.validator.ValidatorForm;
public class LogonForm extends ValidatorForm
```

```
{
    private String ssn = null;
    public String getSsn() {
        return ssn;
    }
    public void setSsn(String s) {
        this.ssn = s;
    }
    public void reset(ActionMapping mapping, HttpServletRequest
request) {
     // Reset values are provided as samples only. Change as
appropriate.
        ssn = null;
    }
    public ActionErrors validate(ActionMapping mapping,
            HttpServletRequest request) {
      ActionErrors errors = super.validate(mapping,request);
       if ((field == null) || (field.length() == 0)) {
        errors.add("field", new
org.apache.struts.action.ActionError("error.field.required"));
       }
       return errors;
    }
}
```

JSP custom tag libraries

The JSP custom tag libraries are a collection of actions presented as tags. This allows you to separate presentation from other application tiers. The libraries are easy to use and you can read them in XML-like fashion. You can easily maintain the JSP components by minimizing the use of Java scriptlets in them. The JSP tags that Struts provides include HTML, logic, and bean tags (Example 4-64).

Example 4-64 Struts jsp custom tag libraries

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
...
<html:form action="/logon">
  <TABLE border="0">
    <TBODY>
      <TR>
        <TH>Please enter your Customer Number(ssn): </TH>
        <TD><html:text property="ssn" /></TD>
      </TR>
```

```
      <TR>
        <TD><html:submit property="submit" value="Submit" /></TD>
        <TD><html:reset /></TD>
      </TR>
    </TBODY>
  </TABLE>
</html:form>
```

ActionErrors

You use ActionErrors to support exception handling. An ActionError traps and propagates an application exception to the View layer. Each one is a collection of ActionError instances. ActionErrors encapsulate error messages, while the </html:errors> in the Presentation layer renders all error messages in the ActionError collection.

4.6.4 Reuse data across multiple ActionForms

We continue by showing you ways to get the most out of the framework. First, Struts recommends that you associate every JSP object with an ActionForm, which encapsulates data represented in the screen. You access the form data in the JSP object using accessory methods found in ActionForm. Example 4-65 shows the conventional use of ActionForm tag in the View layer.

Example 4-65 ActionForm usage in JSP

```
<html:form action="/bp1">
<html:text  property="attrib1" />
</html:form >
```

The ActionForm called “BP1AForm” includes the attribute attrib1, as well as its getter and setter methods. In the configuration file struts-config.xml, the action “/bp1” maps to BP1AForm using the name attribute. This facilitates data display in the JSP (Example 4-66).

Example 4-66 struts-config.xml snippet, initial configuration

```
<action path="/bp1" type="somepackage.BP1AForm" name="bp1AForm"
scope="request">
```

To implement this best practice with reuse of data in ActionForms, Struts recommends that you do two things (Example 4-67):

- Create a JavaBean (BP1BForm) with attributes that form an attribute subset in BP1AForm, along with the attributes' getter and setter methods.

- Replace the attributes in BP1AForm with the bean BP1BForm by associating the bean with bp1AForm. Now you can access this attribute subset in BP1AForm through BP1BForm.

Example 4-67 Accessing form attributes in JSP

```
<html:form action="/bp1">
<bean:define name="bp1AForm" property="bp1BForm" id="bp1B"
                type="somepackage.BP1BForm" />
<html:text  name="bp1B" property="subsetAtt1" />
</html:form >
```

Points to remember

The main advantage of this practice is that you can use it when you have multiple ActionForms to access a set of attributes. When following this best practice, you might want to keep the following considerations in mind:

- Struts implements the <bean:define/> tag.
- When the code <%@ taglib uri="struts-bean.tld" prefix="bean" %> points to struts-bean.tld, the <bean:define/> tag starts to work in the JSP components.
- BP1AForm's validation framework, which extends ActionForm, must validate BP1BForm's data.

When creating Action classes in your application, instead of directly extending org.apache.struts.action.Action, create an Action class (IntermediateAction) by extending org.apache.struts.action.Action to handle common things in your application. All other Action classes extend this IntermediateAction class.

4.6.5 Design guidelines for Actions

When you code an Action class, remember the following design guidelines:

- Write code for a multi-threaded environment: Our controller servlet creates only one instance of your Action class, and uses this one instance to service all requests. Thus, you have to write thread-safe Action classes. Follow the same guidelines you would use to write thread-safe Servlets. Here are two general guidelines to help you write scalable, thread-safe Action classes:
  - Only Use Local Variables: The most important principle for thread-safe coding is to use only local variables, not instance variables, in your Action class. Local variables are created on a stack assigned (by your JVM) to each request thread, so you do not have to worry about sharing them. An Action can be factored into several local methods, so long as all variables required are passed as method parameters. This assures thread safety, as the JVM handles such variables internally using the call stack which is associated with a single Thread.

- Conserve Resources: As a general rule, allocating scarce resources and keeping them across requests from the same user (in the user's session) can cause scalability problems. For example, if your application uses JDBC and you allocate a separate JDBC connection for every user, you are probably going to run into some scalability issues when your site suddenly shows up on Slashdot. You should strive to use pools and release resources (such as database connections) prior to forwarding control to the appropriate View component — even if a bean method you have called throws an exception.

- Don't just throw it, catch it! Have you ever used a commercial Web site only to have a stack trace or exception thrown in your face after you have already typed in your credit card number and clicked the purchase button? Let us just say it does not inspire confidence. Now is your chance to deal with these application errors — in the Action class. If your application specific code throws exceptions, you should catch these exceptions in your Action class, log them in your application's log (servlet.log("Error message", exception)) and return the appropriate ActionForward.

It is wise to avoid creating lengthy and complex Action classes. If you start to embed too much logic in the Action class itself, you might begin to find the Action class hard to understand, maintain, and impossible to reuse. Rather than creating overly complex Action classes, it is generally a good practice to move most of the persistence, and "business logic" to a separate application layer.

When an Action class becomes lengthy and procedural, it might be a good time to refactor your application architecture and move some of this logic to another conceptual layer; otherwise, you could be left with an inflexible application which can only be accessed in a Web application environment. The framework should be viewed as simply the foundation for implementing MVC in your applications. Struts provides a useful control layer, but it is not a fully featured platform for building complete MVC applications.

4.6.6 Use the Action class to handle requests

Typically, when using the Struts framework, for every action that the JSP component requests your application to execute, the application must extend Struts' org.apache.struts.action.Action to create an Action class. This individual Action class interfaces with the application's Model layer while processing the request.

To implement this practice, Struts recommends that you follow these steps:

1. Create an Action class, say BP2Action, by extending org.apache.struts.action.Action.
2. Create all other Action classes in your Web application by extending BP2Action.
3. In BP2Action, create a method performTask(), as in public abstract ActionForward performTask(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException.
4. In BP2Action add one or more generic methods to the application, for example, serverSideValidate(). You can decide on the method's access modifier by considering the following factors:
   - If all Action classes must implement this method, make it abstract.
   - If some Action classes must provide a case-specific implementation, declare the method protected and give it a default implementation.
5. In BP2Action, declare method perform() as final. Invoke the above generic method, which must always be called before processing the request. Now call the method performTask() created in step 3.
6. In every Action class extending BP2Action, add method performTask() with a case-specific implementation.

Advantages

This practice has two main advantages. First, it helps you avoid redundant code in every Action class of your Web application. Second, it gives the application more control over generic tasks by centralizing the behavior in one Action class.

4.6.7 Use Struts validation framework

The Struts validation framework provides automatic validation of forms using configuration files. To explain this best practice, we are using Rational Application Developer.

Consider the following logon.jsp example (Example 4-68).

Example 4-68 Logon.jsp example

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<html:html>
<HEAD>
<TITLE>logon.jsp</TITLE>
```

```
</HEAD>

<BODY>

<P><B>Simple Validation Example</B></P>
<HR>

<center><html:errors/></center>

<html:form action="/logon">
  <TABLE border="0">
    <TBODY>
      <TR>
        <TH>Please enter your Customer Number(ssn): </TH>
        <TD><html:text property="ssn" /></TD>
      </TR>
      <TR>
        <TD><html:submit property="submit" value="Submit" /></TD>
        <TD><html:reset /></TD>
      </TR>
    </TBODY>
  </TABLE>
</html:form>
</BODY>
</html:html>
```

This logon.jsp calls a Struts framework that has an action called LogonAction with a LoginForm associated. See the struts-config.xml snippet (Example 4-69).

Example 4-69 Snippet of struts-config.xml

```
<action-mappings>
  <action path="/logon" type="somepackage.LogonAction" name="logonForm" 
scope="request">
   <forward name="success" contextRelative="false" 
path="/welcomeCustomer.jsp">
   </forward>
   <forward name="failure" contextRelative="false" path="/logon.jsp">
   </forward>
  </action>
  <action path="/logoff" type="somepackage.LogonAction">
   <forward name="success" contextRelative="false" path="/logon.jsp">
   </forward>
  </action>
 </action-mappings>
```

Remember that, in the beginning of Struts life cycle, if the action that is to handle the request has a form bean associated with it, Struts creates the form bean and populates it with the data from the input form. It then calls the validate method of the form bean. Instead of using programmatic validation, you can use the files, validation.xml and validator-rules.xml, which are the two configuration files used by the Struts validation framework to validate forms.

To do this, the LogonForm program does not have your own validation. It is commented using /* comments. See Example 4-70.

Example 4-70 LogonForm

```
package somepackage.forms;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionMapping;
import org.apache.struts.validator.ValidatorForm;

/**
 * Form bean for a Struts application.
 * Users may access 1 field on this form:
 * <ul>
 * <li>ssn - [your comment here]
 * </ul>
 * @version 1.0
 * @author
 */
public class LogonForm extends ValidatorForm

{

    private String ssn = null;

    /**
     * Get ssn
     * @return String
     */
    public String getSsn() {
        return ssn;
    }

    /**
     * Set ssn
     * @param <code>String</code>
     */
```

```
    public void setSsn(String s) {
        this.ssn = s;
    }

    public void reset(ActionMapping mapping, HttpServletRequest
request) {

        // Reset values are provided as samples only. Change as
appropriate.

        ssn = null;

    }

/*
    public ActionErrors validate(ActionMapping mapping,
            HttpServletRequest request) {

        ActionErrors errors = super.validate(mapping,request);

        //ActionErrors errors = new ActionErrors();
        // Validate the fields in your form, adding
        // adding each error to this.errors as found, e.g.

        // if ((field == null) || (field.length() == 0)) {
        //   errors.add("field", new
org.apache.struts.action.ActionError("error.field.required"));
        // }
        return errors;

    }
*/
}
```

To validate the logonForm using the Struts validation framework, do these steps:

1. Import the EAR project called Control/Struts.ear after unzipping our add material.
2. In the RAD Web perspective, add the Struts validator plug-in and required property to the plug-in indicating the location of the validation configuration files. Expand **Dynamic Web Projects** → **Struts** → **WebContent** → **WEB-INF**.
3. Double-click the struts-config.xml file to open in the Struts Configuration Editor.

4. Click the **Plug-ins** tab in the Struts Configuration Editor.
5. Click **Add...** in the Plug-ins field and select the ValidatorPlugIn in the Class Selection Wizard.
6. Click **OK** to close the Class Selection Wizard. The Struts Validator Plug-in has now been added.
7. Add the required parameter by clicking **Add** in the Plug-in Mapping Extension field.
8. In the Property and Value fields, enter in pathnames and /WEB-INF/validator-rules.xml,/WEB-INF/validation.xml, respectively, as seen in Figure 4-32.

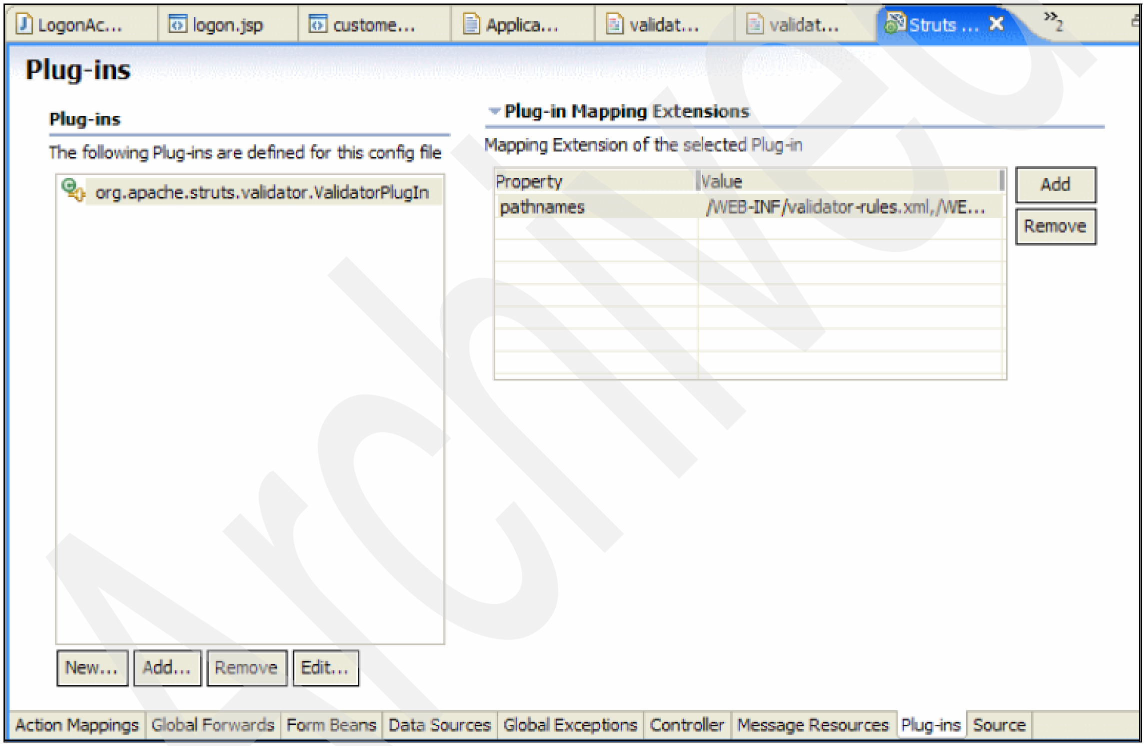


Figure 4-32 Struts configurator editor - Adding validator plugin

9. . Save the configuration file and close the Struts configuration file.

The validation.xml file contains all the Struts form beans and the fields within the form bean that are validated, and the rule to be applied to validate the bean. The validation.xml for the logonForm is shown in Example 4-71.

Example 4-71 validation.xml snippet - LogonForm

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE form-validation PUBLIC "-//Apache Software Foundation//DTD
Commons Validator Rules Configuration 1.0//EN"
       "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <global>

  </global>
  <formset>
    <form name="logonForm">
      <field property="ssn" depends="required">
        <arg0 key="form.ssn" />
      </field>
    </form>
  </formset>
</form-validation
```

The validator-rules.xml file contains the rule configurations for all the rules defined in the validation.xml file. In our previous example, the rule that is defined is required for the field ssn, as shown in Example 4-71. The snippet for the required rule is shown in Example 4-72.

Example 4-72 Validation-rules.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE form-validation PUBLIC "-//Apache Software Foundation//DTD
Commons Validator Rules Configuration 1.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <global>
    <validator name="required"
      classname="org.apache.struts.validator.FieldChecks"
      method="validateRequired"
      methodParams="java.lang.Object,
org.apache.commons.validator.ValidatorAction,
org.apache.commons.validator.Field,
org.apache.struts.action.ActionErrors,
javax.servlet.http.HttpServletRequest"
      msg="errors.required" />
  </global>
</form-validation>
```

4.6.8 Do not make direct JDBC calls from Actions

Some developers make JDBC calls from Actions, but such practices are not recommended. The Struts best practice is for Actions to delegate business and system logic calls to another component, such as using a business facade. The Struts Action passes appropriate values to one or methods on the facade. The outcome is used to determine an appropriate response. Often, the outcome of an Action is described as either “success” or “failure”.

4.6.9 Use ActionForm to work on session data

In a Struts-based Web application, each ActionForm extends org.apache.struts.action.ActionForm. These ActionForms encapsulate page data and provide a validation framework to validate request parameters.

Most Web applications maintain data in session to make them available throughout the application. This best practice addresses this Web application feature. It allows methods toSession() and fromSession() to move session data to and from the form data. Thus, it addresses session data maintenance in a Web application.

To adhere to this practice, follow these steps:

1. Create an abstract class named BP3Form by extending org.apache.struts.action.ActionForm.
2. In BP3Form, add methods with access modifiers as in public abstract void toSession(SessionData sessionData) and void fromSession(SessionData sessionData).
3. In every ActionForm, extend BP3Form and implement the abstract methods in which the form data is transported to and from the session.
4. The corresponding Action class can determine the order in which these methods are called. For example, you could invoke method toSession() on the ActionForm just before actionForward is determined.

When to use it: This practice is most useful when session data is maintained as a single object and/or every page manipulates or uses session data.

4.6.10 Handle exceptions effectively

Conventionally, when an application exception occurs in an Action class, the exception is first logged. Then the class creates an ActionError and stores it in the appropriate scope. This Action class then forwards control to the appropriate ActionForward. Example 4-73 shows how the Action class handles exceptions.

Example 4-73 Exception handling in an Action class

```
try {
//Code in Action class
}
catch (ApplicationException e) {
   //log exception
   ActionErrors actionErrors = new ActionErrors();
   ActionError actionError = new ActionError(e.getErrorCode());
   actionErrors.add(ActionErrors.GLOBAL_ERROR, actionError);
   saveErrors(request, actionErrors);
}
```

While conventional exception handling procedures save exception information in every Action class, the best practice is to avoid redundant code while handling exceptions. To use this practice, Struts recommends following these steps:

1. Create an Action class, such as BP4Action, by extending org.apache.struts.action.Action.
2. Create all other Action classes in your Web application by extending BP4Action.
3. In BP4Action, declare variable ActionErrors actionErrors = new ActionErrors();.
4. In BP4Action, create a method performTask() as in public abstract ActionForward performTask(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response, ActionErrors actionErrors) throws IOException, ServletException.
5. In BP4Action, declare method perform() as final. Then invoke generic methods, which must always be called before processing the request. Now you can call the method performTask() created in the previous step.
6. While implementing method performTask() in every Action class (by extending BP4Action), handle application exceptions as shown in Example 4-74.

Example 4-74 Using Action errors effectively

```
try{
//Code in Action class
}
catch(ApplicationException appException) {
   //Log exception
   //Add error to actionErrors
   actionErrors.add(ActionErrors.GLOBAL_ERROR,
      new ActionError(appException.getErrorCode()));
}
```

In BP4Action, after invoking the method performTask(), save the ActionErrors using saveErrors(request, errors).

Advantages

This main advantage of this practice is that it avoids code redundancy in every Action class that handles ActionErrors.

4.6.11 Choose JSP instead of XSLT for rendering the view in Struts

If you decide to use the Struts framework, you should try to use JavaServer Pages instead of Extensible Stylesheet Language Transformation (XSLT) to generate the user interface, whenever possible, for performance reasons. For more information about XSLT, see 4.3, "XML/XSLT processing" on page 168.

4.6.12 Using Tiles

This section provides a short overview about Tiles in preparation for when we discuss caching Web components in 4.8, "Caching Web components" on page 240 to get more performance.

Tiles overview

Tiles framework builds on the jsp:include feature of JavaServer Pages (JSP) architecture, and comes bundled with the Struts Web application framework. This framework helps to reduce the duplication between JSP files, as well as making layouts flexible and easy to maintain. The Tiles structure provides a full-featured, robust framework for assembling presentation pages from component parts.

Creating a template with Tiles

Consider a set of pages with the same layout in common in terms of header, navigation, and footer layout. Duplicating an HTML code for these components to this page can be a nightmare and it is not a good practice. The JSP include tags (jsp:include) or Tiles inserts (<tiles:insert>) could do the service easily.

Now consider if the position of the HTML body changes — for example, on one page it is on the left, and on another page it is on the right. The new challenge is to find a convenient way to switch the Web UI from one layout to another. The Tiles template feature meets this requirement.

A Tiles template is a generic layout that does not contain any actual content. It contains attributes that are placeholders to enable things such as page URI and string value to be inserted later. Content pages like News.jsp and Solutions.jsp can reference the template and insert the page URI through the attributes.

To create a template page, you use Tiles insert tags. The insert tag in a Tiles template has only one attribute named attribute, which becomes the placeholder for the JSP page URI insertion. The template code to create the header placeholder is <tiles:insert attribute="header" />.

The template defines a common layout of Web UI components that concrete pages can reference. The JSP code in Example 4-75 is a sample template using Tiles.

Example 4-75 Layout.jsp is a Tiles Template (TemplatingTilesWeb)

```
<TITLE>IBM Solutions</TITLE>
</HEAD>
<BODY>
<TABLE border="0">
 <TBODY>
 <TR>
 <TD colspan="2"><tiles:insert attribute="header"/></TD>
 </TR>
 <TR>
 <TD width="20%" valign="top"><tiles:insert
attribute="navigation"/></TD>
 <TD width="80%" valign="top"><tiles:insert attribute="body"/></TD>
 </TR>
 <TR>
 <TD colspan="2"><tiles:insert attribute="footer"/></TD>
 </TR>
 </TBODY>
 </TABLE>
</BODY>
</HTML
```

After a template is defined, it cannot be used on its own. An actual JSP page has to reference the template and provide page URI for the attributes. The line <tiles:insert page="/Layout.jsp"flush="true"> is used to reference the template.

The concrete JSP pages delegate the layout composition to the template Layout.jsp. They have to specify the page implementation names for the attribute using the Tiles put tag. The code <tiles:put name="header" value="/Header.jsp"/> inserts the page /Header.jsp into the attribute named header.

Example 4-76 and Example 4-77 show the JSP code for Solutions.jsp and News.jsp, which use the Tiles template feature.

Example 4-76 Solutions.jsp with the Tiles template feature (TemplatingTilesWeb)

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert page="/Layout.jsp" flush="true">
<tiles:put name="header" value="/Header.jsp"/>
<tiles:put name="navigation" value="/Navigation.jsp"/>
<tiles:put name="body" value="/Fragment-Solutions.jsp"/>
<tiles:put name="footer" value="/Footer.jsp"/></tiles:insert>
```

Example 4-77 News.jsp with the Tiles template feature (TemplatingTilesWeb)

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert page="/Layout.jsp" flush="true">
<tiles:put name="header" value="/Header.jsp"/>
<tiles:put name="navigation" value="/Navigation.jsp"/>
<tiles:put name="body" value="/Fragment-News.jsp"/>
<tiles:put name="footer" value="/Footer.jsp"/>
</tiles:insert>
```

To understand the Tiles facility in more detail, see Figure 4-33.

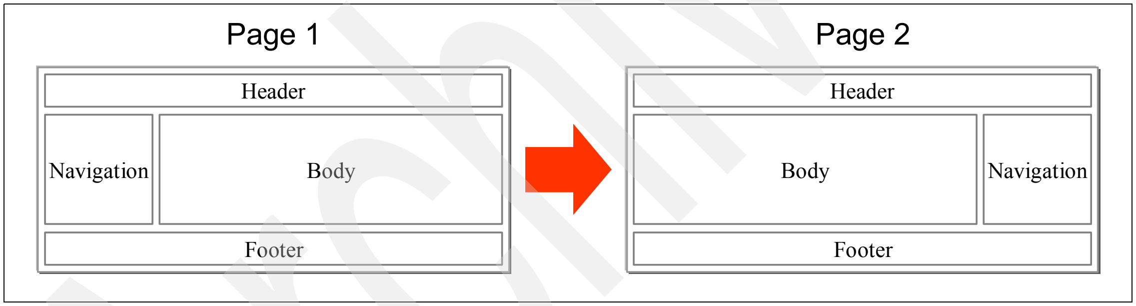


Figure 4-33 Changing page layout with tiles

If you want to change the layout from Page 1 to Page 2, all you have to do is to modify the template Layout.jsp as shown in Example 4-78, without touching concrete pages such as Solutions.jsp and News.jsp. Compared to modifying multiple pages when a template is not used, using the Tiles template feature can significantly reduce maintenance costs.

Example 4-78 Change the template to create a new layout

```
<HTML>
<HEAD>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<TITLE>IBM Solutions</TITLE>
</HEAD>
```

```
<BODY>
 <TABLE border="0">
 <TBODY>
 <TR>
 <TD colspan="2"><tiles:insert attribute="header"/></TD>
 </TR>
 <TR>
 <TD width="80%" valign="top"><tiles:insert attribute="body"/></TD>
 <TD width="20%" valign="top"><tiles:insert
attribute="navigation"/></TD>
 </TR>
 <TR>
 <TD colspan="2"><tiles:insert attribute="footer"/></TD>
 </TR>
 </TBODY>
 </TABLE>
</BODY>
</HTML>
```

For more information about Tiles integration with Struts, see:

http://www.ibm.com/developerworks/websphere/library/techarticles/0311_fung_yu/fung_yu1.html

http://www.ibm.com/developerworks/java/library/j-strutstiles.html

4.6.13 Do not use form beans to transfer data to business logic layer

Although it can generate a small performance overhead to copy the data into a custom created Data Transfer Object, you should not use the view-helper class (form bean) to pass the data to the business logic. The use of form beans for this purpose creates a dependency on the Struts framework that you do not want to force into the Business Process layer. Therefore, try not to use form beans for data transfer, in order to provide a clean separation between the layers and to avoid data conversion at the business logic layer. You might also want to use reflection to easily copy the data between the two objects.

4.6.14 Use servlet/controller best practices to implement action handlers

Actions are multi-threaded like servlets. They communicate with the model, invoke business logic, and return the model objects to the view. Finally, they perform tasks very much like controller servlets. Therefore, you have to follow all of the best practices associated with servlets (see 4.5, “Servlets” on page 172).

4.7 JavaServer Faces

For years, developers have been using servlets and JavaServer Pages (JSP) technology to build Web-based user interfaces. As applications become more and more complex, we face many challenges. If we mix logic and presentation code, we find it extremely difficult to develop large applications consisting of hundreds or even thousands of Web pages. The construction of custom components is another big challenge: A simple table viewer requires a significant amount of time to develop and test. In addition, there is no easy way to port those HTML-based user interfaces onto other platforms, like handheld devices. Struts, a popular Web framework, and some other proprietary technologies solve some but not all of the problems.

JavaServer Faces (JSF) technology is a server-side user interface component framework for Java technology-based Web applications, which is now generally available in WebSphere Application Server V6. One of the greatest advantages of JavaServer Faces technology, compared to standard JavaServer Pages, is that it offers a very clean separation between behavior and presentation. Designed to ease the burden of developing and maintaining applications that run on Java application servers and render their UIs back to a target client, JSF leverages existing, standard UI and Web-tier concepts without limiting developers to a particular markup language, protocol, or client device.

In Figure 4-34 we can see a UI that has been created with JSF, which runs on the server side and renders back to the target client.

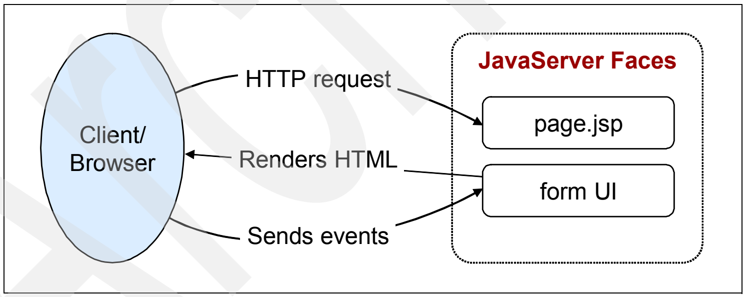


Figure 4-34 JSF simple functionality

This means that the look (rendering) of a *User Interface (UI)* component and its feel (behavior) are now split up, which makes it possible to generate different layouts (while implementing multiple renderers) for one and the same UI component. You can therefore easily add support for multiple different clients to a JavaServer Faces application. Another important goal of JavaServer Faces technology is to deliver — compared to JavaServer Pages, for example — a richer and more responsive UI experience.

JavaServer Faces components automatically provide their own event handling, support client-sided and server-sided validation, and maintain their UI state on the server. Page navigation, data conversion, and internationalization support are also included in the JavaServer Faces framework. This is done by adding a backing bean to each page of the application. A backing bean is a Java bean that defines all properties and methods from the associated UI components used on this page.

4.7.1 JavaServer Faces features and benefits

In general, the JSF Framework is very powerful and easy to use. One reason is that the JSF's reference implementation includes an extensive JSP tag library; there is also very good tooling support from several different products, such as IBM Rational Application Developer V7.0. Even though the reference implementation demonstrates JSF with JSP, it is important to realize that JSF is not exclusively tied to the JavaServer Pages technology.

The following is a list of the key features and benefits of using JSF for Web application design and development:

- Standards-based Web application framework:

  JSF is a standards-based Web application framework. JavaServer Faces technology is the result of the Java Community process JSR-127 and evolved from Struts. JSF addresses more of the Model View Controller pattern than Struts, in that it more strongly addresses the view or presentation layer though UI components, and addresses the model through managed beans. Although JSF is an emerging technology and likely to become a dominant standard, Struts is still widely used. JSF is targeted at Web developers with little knowledge of Java and eliminates much of the hand coding involved in integrating Web applications with back-end systems.

- Event driven architecture:

  JSF provides server-side rich UI components that respond to client events.

- User interface development:

  UI components are de-coupled from its rendering. This allows for other technologies such as WML to be used (for example, mobile devices). JSF allows direct binding of user interface (UI) components to model data. Developers can use extensive libraries of prebuilt UI components that provide both basic and advanced Web functionality.

- Session and object management:

  JSF manages designated model data objects by handling their initialization, persistence over the request cycle, and cleanup.

- Validation and error feedback:

  JSF allows direct binding of reusable validators to UI components. The framework also provides a queue mechanism to simplify error and message feedback to the application user. These messages can be associated with specific UI components.

- Internationalization:

  JSF provides tools for internationalizing Web applications, including supporting number, currency, time, and date formatting, and externalization of UI strings.

Despite all the advantages of the new JavaServer Faces technology, there are also a few drawbacks:

- JavaServer Faces is a relatively new technology, so it has not proven its performance and stability in very large projects.
- First performance tests indicate that although the Faces Framework is more powerful, it is just a little bit slower than JavaServer Pages or Struts, because of its higher level of abstraction and the more complicated life cycle. When a client makes a request for a page containing JSF UI components, the JavaServer Faces implementation must perform several tasks, such as validating the data input of all components in the view and converting input data to types specified on the server side.

  The performance and scalability of the JavaServer Faces Framework really depends on the implementation of the UI components you are using. If, for example, one UI component stores the state of thousands of table rows in its backing bean and therefore takes one minute to render the view, you cannot blame the JavaServer Faces architecture for this. So if you add custom UI components to your application, always check if they are optimized for performance and scale well.

- You should control the amount of data that is stored in your backing beans.

  JavaServer Faces implementations often provide very powerful UI components that make it easy for you to display data from your back end on the Web. Make sure that the amount of data that should be rendered does not get too much. For example, do not misuse a UI component to fetch a whole database table and store it in the backing bean, just to display some entries. This might work well for a small number of users, but it neither scales well nor satisfies performance expectations for a larger number of users.

4.7.2 JavaServer Faces architecture

The JSF application architecture can be easily extended in a variety of ways to suit the requirements of your particular application. You can develop custom components, renderers, validators, and other JSF objects and register them with the JSF runtime.

First of all, we group the JSP architecture in MVC components.

Model View Controller architecture

Applications built with JavaServer Faces are intended to follow the Model View Controller (MVC) architectural pattern. According to the MVC pattern, a software component should separate its business logic along the following lines:

- Model: Encapsulates the state and behavior of the application
- View: Renders the model
- Controller: Processes user events and drives model and view updates

As shown in Figure 4-35, we can group the important components of JSF as they fall into these categories:

- Model:
  - Managed beans make up the model of a JSF application. These Java beans typically interface with reusable business logic components or external systems, such as a mainframe or database. Also JSF moves the data between managed beans and user interface components. JavaBeans are defined in the configuration file to hold the data from JSF components.
- View:
  - JSPs make up the view of a JSF Web application. These JSPs are created by combining model data with predefined and custom-made UI components.
  - Tag libraries: The JSF components are implemented in tag libraries.
  - Validators: Java classes to validate the content of JSF components, for example, to validate user input.
  - Events: Java code executed in the server for events (for example, a push button). Event handling is used to pass managed beans to business logic.
  - UI components represented as stateful objects on the server
- Controller:
  - The FacesServlet, one servlet which drives navigation and object management, makes up most of a JSF application's controller. Event listeners also contribute to the controller logic.

– Configuration file: An XML file (faces-config.xml) that contains the configuration.

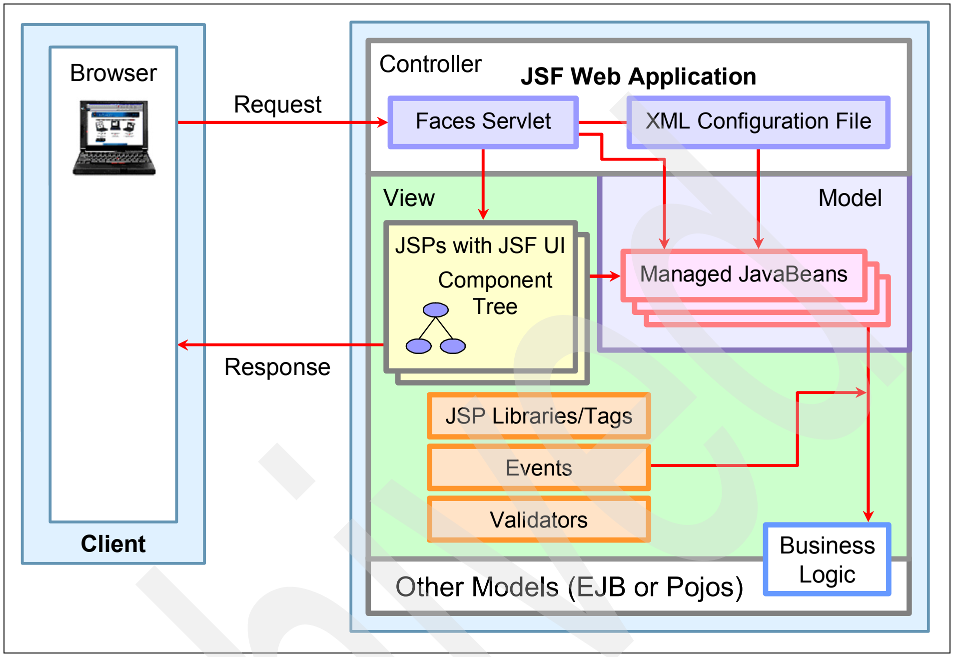


Figure 4-35 JavaServer Faces architecture

JavaServer Faces is layered directly on top of Servlet API, and although JSF uses JSP custom tag libraries for presentation, JSF can use another presentation technology instead creating your own custom components directly from the component classes, and generating output for various client devices. This is the reason that we put in the control chapter. JSF is not a view in MVC patter, is a complete framework.

A typical JavaServer Faces application that is using JSP pages for presentation that render to HTML must include a custom tag library that defines the tags representing UI components and have another custom tag library for representing other core actions, such as validators and event handlers. JavaServer Faces implementation provides both of these tag libraries.

4.7.3 FrameWork responsibilities

With a clear separation between page design and coding, we can split some responsibilities. JSF allows for more secure development and design phases. This is very important, mainly in big projects. Here is an overview of each responsibility:

- Page authors: They are graphics designers who have a lot of experience in HTML; also, they are the primary users of JSP tags.
- Application developers: This includes program event handlers, validators, converters, as well as extra helper classes.
- Component writers: These are experienced programmers who prefer to create a custom UI with a programming language. These people can create their own components directly from the UI component classes, or they can extend the standard components provided by JavaServer Faces technology.
- Application architects: They design the application, defining the scalability, configuration beans, page navigation, and register objects within application.
- Tools vendors: They build tools to allow easy JSF programming with wizards and visual facilities. An example is Rational Application Developer V6 and V7, which offer a lot of tools.

Putting responsibilities in a scenario

Now we put the responsibilities described above in a scenario for better understanding.

An application developer has written some Java classes which provide an object-oriented interface for a set of tables in a database. The developer packages the code into a JAR file and hands it off to a page author.

A page author creates a new JSF application and imports the supplied JAR file from the application developer. He creates a new JSF page for interfacing with the database. He adds a class from the JAR to the page as a managed bean, then drags it on to the page. A set of fields are created with which to update the record. He drags a command button onto the page from the Web design palette, and associates it with a supplied update method in the supplied business logic. He adds navigation rules to display other pages depending on the success of the update operation. Finally, he tests the JSF page by running it on the application server.

The end user accesses the Web application for the database application, and can now update the database directly through a familiar Web interface. The application developer has no Web skills, the page author has no Java skills, and the end user does not even know he is updating a database.

4.7.4 JavaServer Faces life cycle

JSF is typically used within Web applications and thus handles HTTP requests and produces HTTP responses. The JSF run time manages a sequence of events known as the JSF life cycle. The main actor here is the FaceServlet.

The FacesServlet (javax.faces.webapp.FacesServlet), is the heart of a JSF application. It is responsible for processing and using the metadata in the faces-config.xml to handle the model life cycle and navigation. It receives UI events and maps them to the appropriate business logic. The FacesServlet processes external input, also known as the request, in a set of phases, which eventually result in the production of new output, or a response.

This life cycle has seven phases, as shown in Figure 4-36. The solid lines show normal flow of control, whereas the dashed lines show alternate flows that are mostly used to handle errors.

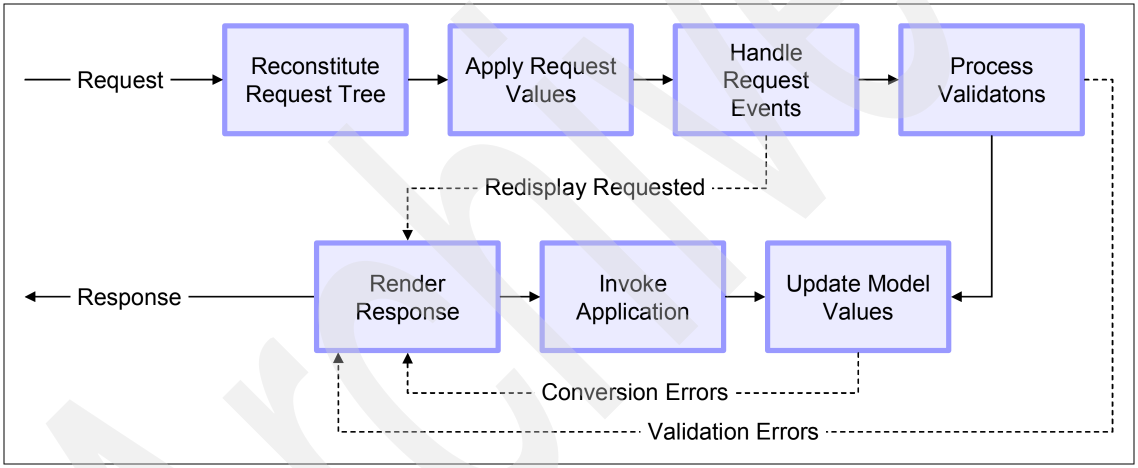


Figure 4-36 JSF life cycle

Associating with a simple example

Suppose that an end user is currently viewing a Web page, updateCustomer.jsp, which is used to update a customer record. The page contains name and address fields, which are bound to a customer model object, and a Submit button, which is bound to an update method. There is a navigation rule defined which forwards to the page customerList.jsp on a successful update. The user changes the customer's last name from Smith to Smythe, and clicks **Submit**. Using this example, we split these actions into the phases described next.

Reconstitute request tree

The first thing that happens when a request comes in (as a result of a link or button being clicked) is the reconstruction of the request tree. A JSF page is a tree of components, and the JSF run time instantiates this tree for use by the other phases of the life cycle, including the wiring of event handlers (registered as listeners), validators, and navigators. If the tree has already been constructed (perhaps because this is a second request within the same page), the JSF run time restores the application state information from the server.

Reconstitute request tree simple example

JSF looks up the stored view information for updateCustomer.jsp, and successfully finds it.

Apply request values

When the components tree is restored, values from the request (typically from an HTML form) are applied to the components in the tree. For example, input fields have values sent from the client, and these values have to be applied to the component tree. Each component is responsible for extracting its values from the request using various decode methods.

Apply request values simple example

The component bound to the customer object's last name property pulls the value Smythe from the request and stores it in itself. Remember that Smith was changed to Smythe.

Handle request events

Events might be outstanding in the Handle Request Events phase, in which case the JSF run time dispatches events to the listeners. A typical example of such event handling occurs when an event handler has registered interest in value changes of a value (typically through an input field).

Handle request events simple example

Suppose that a changed field was a date field that automatically puts "/" between day and month for example. depends of the data that you type the field appearance is changing. this case can be considered a handle request event that only this field has to be changed.

Process validations phase

Next comes the Process Validations phase. The JSF run time processes all validations registered on the components of the tree. If values cause a validation rule to fire, an error message is queued, and the run time progresses to the Render Response phase, causing the page to be rendered again with error messages next to the erroneous components.

Process validations simple example

The validator associated with the last name component determines that the new value is all letters, and hence, is valid.

Update model values

Now that the JSF run time knows that the data are valid, it can update the values in the model components. The run time updates the bean properties that the UI component's attributes point at. If conversion errors occur, the run time redirects to the Render Response phase, and the error messages are displayed on the page.

Update model values simple example

The last name component updates the last name field in the customer JavaBean object.

Invoke application example

The run time then handles any application-level events, and all listeners are activated based on all queued events.

Invoke application simple example

JSF invokes the update method (application-level events), which writes the customer data from the customer object back into the database. JSF sees that the method returned success and determines that the next page is the customerList.jsp.

Render response

Finally, the run time invokes the component's encoding functions and renders the component tree to produce the final HTML that is sent to the client who made the original request.

Render response simple example

JSF forwards to customerList.jsp, which is rendered to the client. While rendering, JSF stores the component structure from the page as the saved view.

4.7.5 JavaServer Faces page components

Each JSF page is a tree constructed of components; this tree is often called the JSF view. Each component in turn has a list of child components, a set of attributes, one or more validators, and one or more event handlers. A JSF component's attributes store component-specific information such as the URL for an image component and the text for a label component.

Validators are used by the Process Validations phase, and the event handlers are used by the Handle Request Events phase. JavaServer Faces components can either render themselves or delegate to an external renderer.

JSF component event handling can be managed directly by the component or by delegating to event handlers that are registered with the component. This added layer of flexibility allows you to build flexible UIs as well as build custom components and custom event-handling code.

If you use Rational Application Developer (RAD), for example, to design your page with JSF components, remember that there is a large “infrastructure” on which you can rely and which prevents a lot of unnecessary coding. This infrastructure comes in the form of libraries of components and includes:

- A tag library for rendering UI components on pages
- A tag library for event handling, validation, and conversions
- UI components that run on the server, manage application state, and generate the HTML to be displayed
- Backing Beans, which work with the UI components and maintain properties that the UI components use as well as process events on behalf of the UI components

The tag libraries used when developing JSF applications represent UI components and core actions such as validation and event handling. The component tag library eliminates the requirement to hard-code UI components, and the core tag library makes it easy to register events, validators, and other actions with the components. In RAD, some of these tag libraries became part of your project when you chose to create a Faces JSP Page; others became part of the project when you added the HTML JSF library to the JSF page. After these libraries are available to you, you add components to your page directly from the palette.

4.7.6 Use JSF from an application example

Initially, the best practice here is developing using JSF, which has several advantages, as we discussed in prior topics. We describe the facility to develop with JSF. Of course, if you have an abstraction layer above simple components, such as JSP and servlets, there are some drawbacks as described in early JSF sections. With a base theory and practice described in sections such as JSP and Servlets, the developer has a solid understanding. As you might have realized by now, JSF already has a collection of development and design best practices.

For more information to help initial developers understand how to do their first example using Rational Application Developer, there is a good DeveloperWorks tutorial in:

`http://www.ibm.com/developerworks/edu/i-dw-r-webuis-i.html`

4.7.7 Best practices to use JSF and/or Struts

In this section we discuss the relationship of the Struts framework to the JavaServer Faces framework. We compare and contrast the two frameworks, then discuss how you might use them in conjunction. Also, we show a practical example of use. For more details about JSF architecture, see 4.7.2, “JavaServer Faces architecture” on page 223. For more details on Struts, see 4.6.1, “Model View Controller model 2 pattern with Struts” on page 199.

Comparing JavaServer Faces and Struts

Struts and JavaServer Faces contain many of the same features, and both have a very similar overall structure. However, depending on the requirements of the application, it is useful to examine the various discrepancies between the two frameworks.

See Table 4-17 to understand the differences and similarities.

Table 4-17 JavaServer Faces and Struts feature comparison

Features	JavaServer Faces	Struts
Components	► Rich data bound UI components with events provided ► Custom components. JSF makes it relatively easy to combine complex GUIs into a single manageable component	► Struts-specific tag library ► Only very basic, form bean bound components provided
Device independence	► Reader kits that provide device independence	► None
Event handling and validation	► Validation framework ► Many predefined validators	► Validation driven by an XML descriptor (validation.xml)

Features	JavaServer Faces	Struts
Scripting	► Script can be atached to events ► All components accessible from scripts	► Scripts written in Java action classes ► Form data, but not components, accessible
Page flow	► Simple navigation file (faces-config.xml). It is simpler. ► XML-based	► Sophisticated, flexible framework ► XML-based
Session and object management	► Automatic	► Manual
HTML and HTTP limited	► No	► Yes
Simple Controller and bean definitions	► Does not require your controller and beans class to extend any particular parent class	► Struts requires that you extend a particular class (Action) and method (execute) for controller and bean class
Current tool support	► Less supported by IDEs	► Supported by many widely IDEs.
Tiles like support	► JSF does not have but you can extract Tiles from struts and use with JSF	► Comes with a powerful page layout facility
Post and Get support	► JSF does not support GET, so you cannot bookmar results pages	► Yes support boths
Client-side validation built-in	► Not supported. AJAX can help with this feature	► Support using form-field validation

Apart from the features of the frameworks, it is important to consider the strength of their relative tools, maturity, and future directions. Refer to Table 4-18.

Table 4-18 JavaServer Faces and Struts considerations

	JavaServer Faces	**Struts**
Tooling	Extensive Support for JSF in RAD 6 and improvements in RAD 7	Extensive, mature IDE support.
Maturity	Not mature as Struts but can be considered well mature.	Quite mature, stable, not subject to significant changes.
Recent and Future Plans	In continuous development. There are a lot of complementary frameworks to extend this functionality as AJAX for example. The tools are in the continuous improvement.	Struts 2 was recently launched as a effort to improve productivity in Struts, making Web application development faster, easier, and more productive than ever before.

Choosing Struts and/or JavaServer Faces

JavaServer Faces originated as a follow-on to the Struts framework, largely driven by the original creator of Struts, Craig McClanahan. However, JSF is not intended to be a complete replacement for Struts. In fact, when deciding which framework to use when developing your applications, the choice between JSF and Struts is not mutually exclusive.

The "Struts Faces" component contains an add-on library that supports the use of JavaServer Faces user interface technology in a Struts-based Web application, in place of the Struts custom tag libraries.

The Struts Faces Integration Library should work with any implementation of JavaServer Faces, version 1.0 or later. Note that the Struts Faces component is intended as a "bridge" between Struts 1 and JavaServer Faces. The typical use case is a team that would like to experiment with JSF components but does not want to completely rework an existing application.

For more information, see the following URLs:

- *WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects*, SG24-6361:

  `http://www.redbooks.ibm.com/abstracts/sg246361.html?Open`

- Struts Faces:

  `http://struts.apache.org/1.x/struts-faces/index.html`

- Struts 2:

  http://struts.apache.org/2.x/
- Apache Shale:

  http://shale.apache.org/

4.7.8 JSF and AJAX integration

JSF introduced a strong standard into the world of Java-based Web application frameworks. Meanwhile, advanced user interface (UI) features to the browser client in a standards-based way are introduced by AJAX. Richness of fat client interfaces that a lot of developers were awaiting from AJAX make this possible.

Asynchronous JavaScript and XML (AJAX) is arguably the most popular technology on the Web today, because it is the cornerstone of what is often called Web 2.0. The good news is that JSF and AJAX are highly complimentary. They provide developers with a complete client-side and server-side technology set for building highly interactive and standards-based Web applications when combined.

However, creating an AJAX application is not an easy task, especially when you have to integrate it with another framework, such as JavaServer Framework. IBM Rational Application Developer V7 provides AJAX functionality for the JSF components, which makes the task much easier.

AJAX advantages and disadvantages

Although not really new, AJAX technology has become very popular in the last year or two. Many major Web sites use it to improve their users' experiences. In fact, improving the user experience is what AJAX is all about.

In the usual Web applications developed over the past decade, the interactions between the user and the browser, as well as between the browser and the server, are well-defined and visible: The user sees a page in the browser, takes an action (picks something from a contextual drop-down menu, or selects a few check boxes), and then instructs the browser to communicate with the server by clicking a link or a Submit button. The browser sends a request to the server and passes the user's input in that request. The server processes the request and sends back a response, which is either a new page or the same page, but updated.

Such Web applications are now commonly referred to as Web 1.0. They have two distinct weak points from the user experience point of view:

- The interaction between the browser and the server is initiated by a limited number of controls on the page — usually only by links and buttons. Rarely would a server be notified immediately after the user selects a check box on the page or marks a selection in a combo box.
- The interaction between the browser and the server results in updating the entire browser window. This is often so slow that the user has to wait a significant amount for time for the page to update. Worse, when the same page is reloaded or refreshed, it usually flickers in the browser window.

The new generation of Web applications, commonly called Web 2.0, works around these weak points by using AJAX technology. In AJAX, interactions between the browser and the server occur in the background, unnoticed by the user. They are also more targeted than the usual browser-server interactions, in that only a subset of the page can be sent to the server, and the server can return only a subset of the page to be updated. As a result of this approach, communication between the browser and server can be initiated by almost any event, such as a selection change in a combo box or a check box or hovering mouse pointer. This results in significant benefits:

- Communication is faster, because less data is transmitted.
- The user stays on the same page, because less navigation from page to page is required.
- The reloaded page does not flicker, because only small regions of the page get updated with each AJAX request.

The ideas behind AJAX are quite simple: Listen for an event in the browser, send a background request to the server, and update part of the page when server responds. But the implementation can be very complicated. It requires in-depth knowledge of JavaScript, client-server communication protocols, and server-side code. The differences between versions of major browsers make it even trickier to develop and debug. However, IBM Rational Application Developer Version 7 provides everything you require to develop AJAX-enabled Web applications without having to implement all of the low-level details.

Rational Application Developer V7 provides:

- Extensions to JSF that allow AJAX requests to be processed within the JavaServer Framework
- A JavaScript library that can initiate AJAX requests in all recent versions of the major browsers and process server responses by updating only parts of the page.

How to use AJAX with JavaServer Faces components

Adding AJAX to a JavaServer Faces page is a four-step process:

1. Identify the area of the page to be updated by the AJAX request. In Rational Application Developer V7, you can use AJAX with the content of almost any panel component. The panels range from simple containers, such as <h:panelGroup> and <h:panelGrid>, to feature-rich panels, such as menus (<hx:panelMenu>) and dialogs (<hx:panelDialog>).
2. Select the type of AJAX request to use. There are three different kinds of AJAX requests supported in the Rational Application Developer V7 JSF library:
   - GET request for the same page (<hx:ajaxRefreshRequest>)
   - POST request for the same page (<hx:ajaxSubmitRequest>)
   - GET request for another page (<hx:ajaxExternalRequest>)
3. Configure parameters to pass to the server with the AJAX request:
   - For GET requests, you can pass values of various input fields on the page.
   - For POST request, the entire form is submitted.
4. Identify the event that initiates the AJAX request. This can be any client-side JavaScript event, such as onclick for a button, onblur for an input field, or onchange for a check box.

Let us walk through all of these steps, using a simple example of a “Hello, world” type of application. You build a page with two fields: input and output. After the user tabs out of the input field, you use AJAX to send the value that the user entered to the server and to update the output field with a greeting.

Set up your Web project

To begin, create a Web project (see Figure 4-37):

1. Select **File** → **New** → **Project** → **Dynamic Web Project** from the menu.
2. In the New Project wizard:
   a. Enter a project name (for example, HelloWorld).
   b. Select the Faces Project configuration.
   c. Select Add project to an EAR.
3. Click **Finish**.

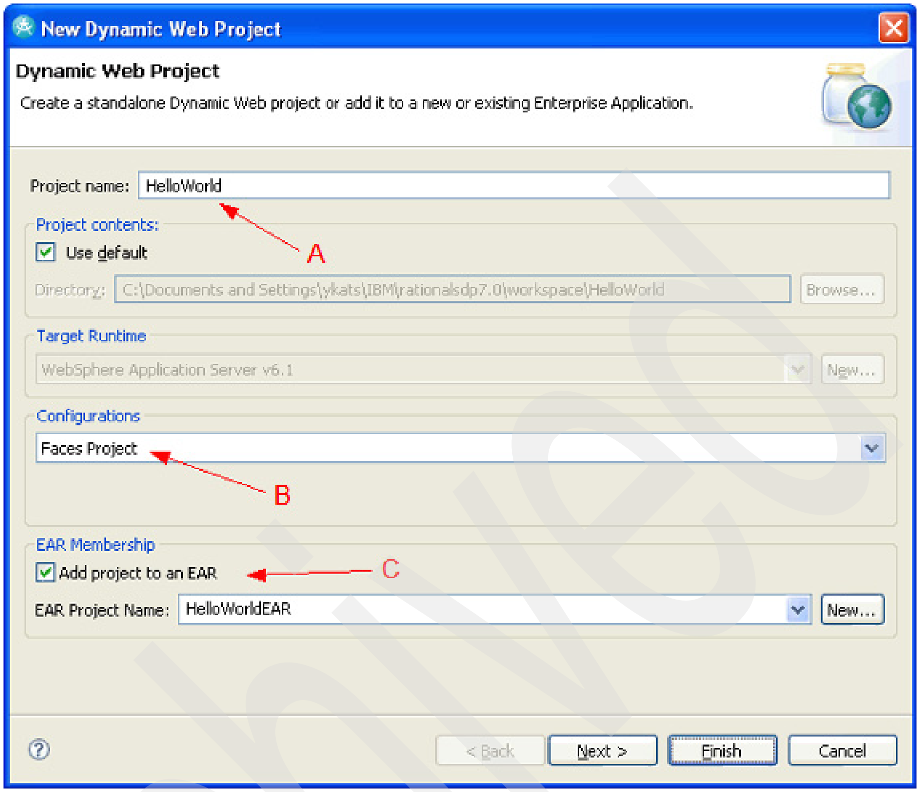


Figure 4-37 Creating a Web Project HelloWord

To create a Web page (see Figure 4-38):

1. Right-click the project name in the Project Explorer.
2. Select **New** → **Web Page** from the context menu.
3. In the New Web Page wizard, enter a page name (for example, hello).
4. Click **Finish**.

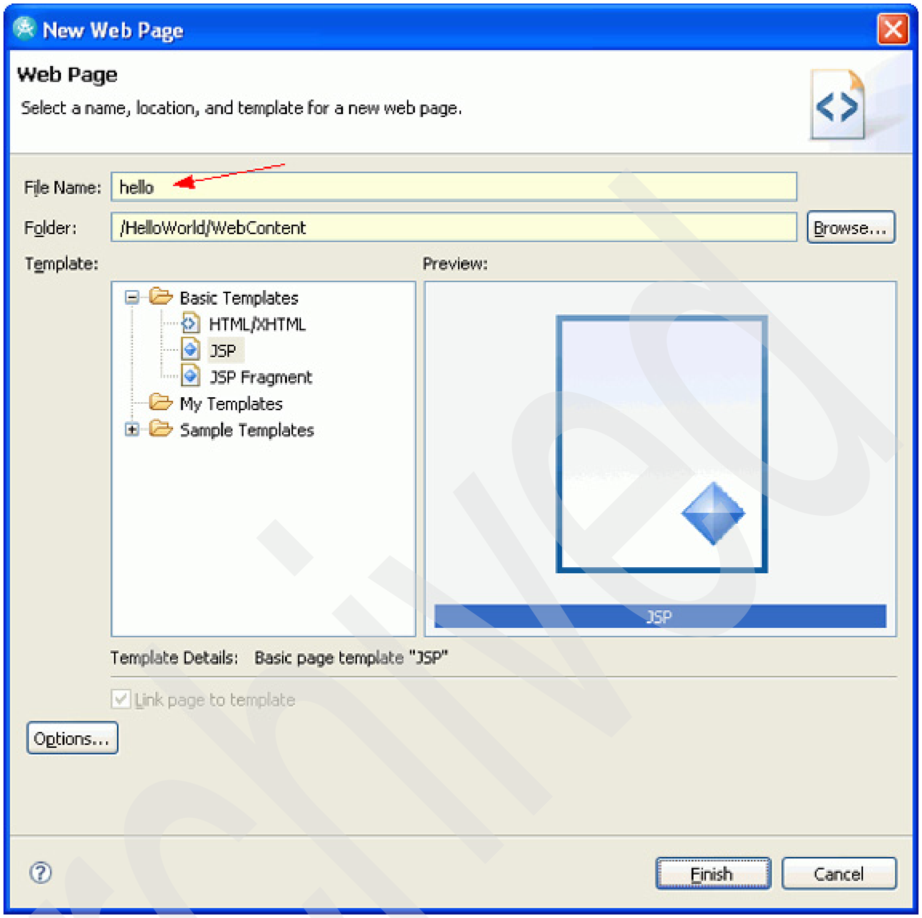


Figure 4-38 Creating a hello Web page with new Web Page wizard

Add components to the page

Now that you have a page to work with, you can add the components. You use an inputText component for the text field where the user types a name and an outputText component to show the greeting. Because you are going to update the outputText with AJAX, you have to put it inside of a panel component. You use a panelGroup component for this page.

To add the components:

1. Drag an Input component from the Enhanced Faces Components drawer of the palette onto the page.

2. Drag a Panel Group box component from the palette onto the page below the Input component. When you are prompted for the group box type, select **Group**.
3. Drag an Output component from the palette onto the Panel Group box.

Add AJAX support to the panel

To make content of the panel updatable via AJAX (in this case, an Output field), you have to mark the panel as “AJAXable” and configure the parameters that you want the user's request to pass to the server (see Figure 4-39):

1. Select the outputText component and switch to the Properties view.
2. In the Properties view, select the h:panelGroup tag, which is directly above the h:outputText tag in the left-side tag navigator.
3. Select the AJAX page for the h:panelGroup tag.
4. Click the **Allow AJAX updates** check box.
5. Select **Refresh** as the AJAX request type.

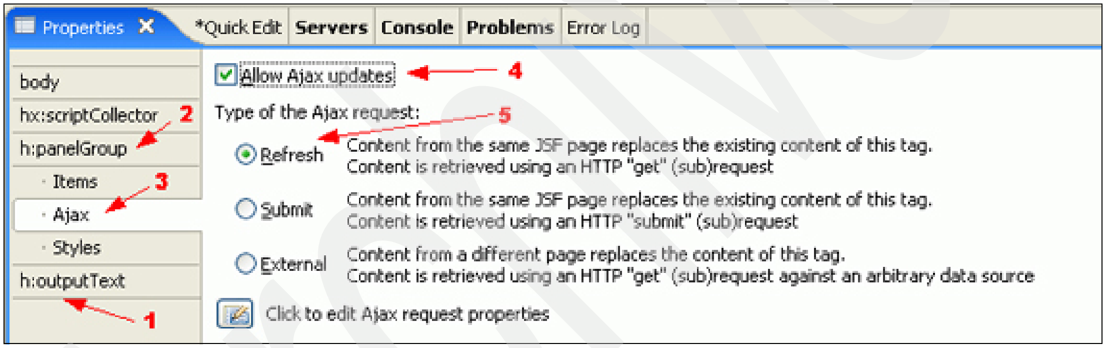


Figure 4-39 panelGroup properties

This example uses a refresh request to show how parameters can be passed with an AJAX request. Alternatively, a submit request would submit the entire form. In that case, because the form on the sample page contains just one input field, you would not have to configure parameters for the AJAX request at all.

To configure parameters for the AJAX request, select **Click to edit AJAX request properties** on the AJAX properties page (Figure 4-39).

On the Properties page for the hx:ajaxRefreshRequest tag:

1. Click **Add Parameter** for the parameters to send from the browser.
2. Select the name of the Input component (in this case, text1) from the combo box (see Figure 4-40).

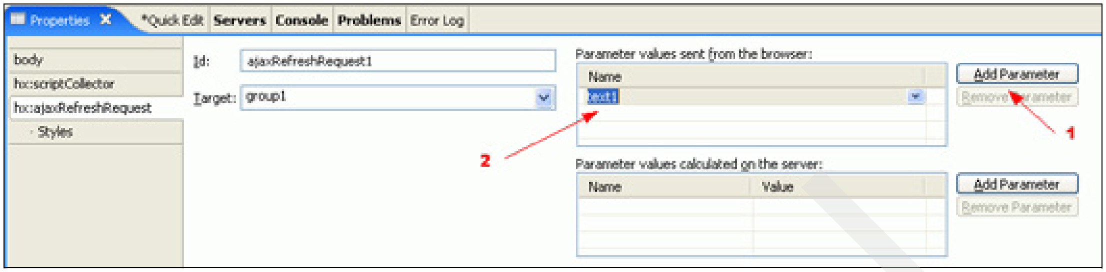


Figure 4-40   ajaxRefreshRequest properties

You have configured the panelGroup tag to be updated by an AJAX request and to use the value of the Input field as a parameter for the request. The only thing left to do is to make the outputText component use this parameter to display a greeting (see Figure 4-41):

1. Select the outputText component.
2. Enter Hello, #{param.text1} into the Value field.

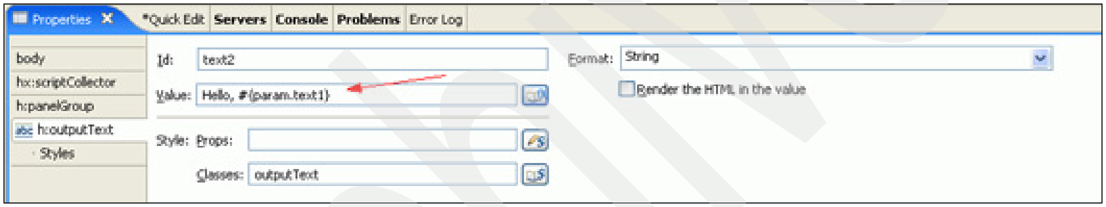


Figure 4-41   outputText Properties

Initiate the AJAX request

If you look back to the four steps required to use AJAX, you see that you have already completed the first three steps. Now you just have to identify the event to trigger the AJAX request. To update the greeting as soon as the user tabs out of the input field, you use the onblur event on the inputText component (see Figure 4-42):

1. Select the inputText component.
2. Switch to Quick Edit view.
3. In the Quick Edit view:
   a. Select the onblur event in the list of events on the left side.
   b. Click the **Use predefined behavior** check box.
   c. Select the **Invoke AJAX behavior on the specified tag** action.
   d. Select the name of the panelGroup (in this case, **group1**) as the target.

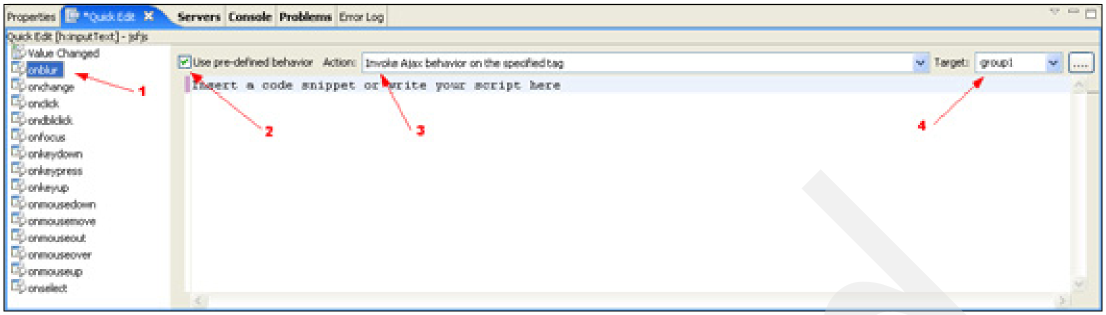


Figure 4-42 Quick edit view

Now you can save the page and run it on a server. When the browser window opens, you see an input field and the “Hello” text beneath it. As soon as the user types anything in the field and then tabs out, the greeting is updated with the text that the user typed in the input field. See Figure 4-43.

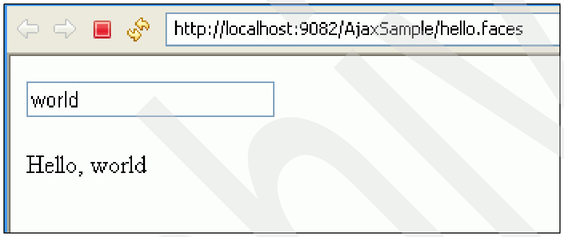


Figure 4-43 Running the Web page on server

As you can see, you were able to build a simple yet functional AJAX page with standard JSF components and absolutely no JavaScript code.

This tutorial is based on the article at the following URL:

http://www.ibm.com/developerworks/rational/library/06/1205_kats_rad2/

4.8 Caching Web components

Server-side caching techniques have long been used to improve Internet performance of Web applications. In general, caching improves response time and reduces system load. Since the introduction of dynamic caching, WebSphere Application Server is also able to cache dynamic content that changes from time to time. Although dynamic caching can be added later to nearly any Web-based application, because the programming model is unaffected, it requires a proactive and effective invalidation mechanism to ensure the freshness of the content.

To maximize the performance improvement, it might also make sense to decompose the pages into several small cacheable units (see 4.2.13, "Use composed JSPs to optimize caching and code re-use" on page 167). Therefore, you should plan to integrate the IBM dynamic caching technology as soon as possible in your development life cycle. Although JSP servlet caching is an IBM extension, and is not part of the current J2EE specification, the application changes for invalidating cache entries are marginal. Therefore, the application is still portable and compliant with the J2EE specification, but still benefits from the performance optimizations provided by the dynamic cache. The cache policy itself is specified declaratively and configuration is through XML deployment descriptors.

IBM WebSphere Application Server V6 now provides support for the following presentation technologies:

- JavaServer Pages/servlets
- Struts and Tiles

For more information on dynamic caching, see Chapter of *WebSphere Application Server V6 Scalability and Performance Handbook*, SG24-6392. You can see significant improvements in the performance of a Web application, and WebSphere Application Server offers a built-in dynamic caching service for caching such content.

4.8.1 Configuring cache policy for your servlet and JSP

Dynamic caching in WebSphere Application Server supports the caching of Java servlets and JSP files. The servlet and JSP cache can be enabled or disabled using the Web container settings in the WebSphere Application Server administrative console. See 4.6.14, "Use servlet/controller best practices to implement action handlers" on page 219. In addition, the dynamic cache also requires a cache policy for each cacheable servlet and JSP file. This policy defines a set of rules for the cache to decide when and how to cache an object. These rules are stored in the cachespec.xml file. This file is localized inside the Web module WEB-INF or enterprise bean META-INF directory.(See the WebSphere Application Server Information Center for more information.)

Because all JSP files are compiled into servlets by WebSphere Application Server, JSP files and servlets are identical from the viewpoint of the dynamic cache. Then the same set of rules applies for servlets and JSP files. For example, here is a sample cache policy for the servlet com.ibm.sample.TimeStampServlet.class, which has a servlet-mapping of /TimeStamp. See Example 4-79.

Example 4-79 Caching policy configuration example

```
<cache-entry>
 <class>servlet</class>
  <name>/TimeStamp</name>
 <cache-id>
  <component id="location" type="parameter">
   <required>true</required>
  </component>
  <timeout>180</timeout>
 </cache-id>
</cache-entry
```

In this example, the servlet response is cached based on the request parameter location. In addition to the request parameter, the servlet and JSP responses can also be cached based on the request attribute, servlet path, pathinfo, HTTP session, request header, request locale, and cookie.

The <name> element defines either a fully qualified class name or the URI of a servlet and JSP file. In WebSphere Application Server releases prior to V6.0, only one cache policy (that is, <cache-entry>... </cache-entry> in the cachespec.xml file) is permitted for each servlet and JSP file. A one-to-one mapping existed between the cache policy and the servlet class. In WebSphere Application Server V6, however, multiple cache policies are supported for a single servlet. For example, if the previously mentioned servlet has mappings /TimeStamp, /TimeStamp1 and /TimeStamp2 defined in the web.xml file, you can have three cache policies in addition to the one mentioned above for the same servlet, as shown in Example 4-80.

Example 4-80 Caching policies for the same servlet

```
<cache-entry>
  <class>servlet</class>
   <name>/TimeStamp1</name>
  <cache-id>
    <component id="time" type="cookie">
     <required>true</required>
    </component>
  </cache-id>
</cache-entry>

<cache-entry>
  <class>servlet</class>
   <name>/TimeStamp2</name>
  <cache-id>
    <component id="attr" type="attribute">
```

```
      <required>true</required>
    </component>
  </cache-id>
</cache-entry>
<cache-entry>
  <class>servlet</class>
   <name>com.ibm.sample.TimeStampServlet.class</name>
  <cache-id>
    <component id="" type="pathinfo">
      <required>true</required>
    </component>
  </cache-id>
</cache-entry>
```

The dynamic cache looks through the cache policies in the order in which these policies are defined in the cachespec.xml file until it finds a rule that matches the current request, then builds a cache ID using that rule.

4.8.2 Configuring cache policy for your Struts and Tiles

Struts and Tiles caching is an extension of servlet and JSP caching. Enabling servlet caching using the Web container setting in the administrative console automatically enables Struts and Tiles cache. In addition to enabling the servlet cache, a cache policy is also required to cache a Struts or Tiles response.

Struts configuration

The Struts framework provides the controller component in the MVC model 2 application. A servlet called org.apache.struts.action.ActionServlet.class is the controller. A servlet mapping of *.do is added for this Struts ActionServlet servlet in the web.xml file of the application so that every request for a Web address that ends with .do is processed by this servlet. The ActionServlet servlet uses the information in the struts-config.xml file to decide which Struts action class is called to actually run the request for the specified resource.

As mentioned earlier, only one cache policy is supported per servlet in releases prior to WebSphere Application Server V6, but in the case of Struts, every request URI ending in .do maps to the same ActionServlet.class. To cache Struts responses, the cache policy has to be written for the ActionServlet servlet based on its servlet path.

For example, consider two Struts actions: /HelloParam.do and /HelloAttr.do. To cache their responses based on the request parameter ID, and the request attribute arg respectively, the cache policy looks as shown in Example 4-81.

Example 4-81 Caching policy example prior WebSphere 6.x

```
<cache-entry>
 <class>servlet</class>
 <name>org.apache.struts.action.ActionServlet.class</name>
 <cache-id>
  <component id="" type="servletpath">
      <value>/HelloParam.do</value>
  </component>
  <component id="id" type="parameter">
      <required>true</required>
  </component>
 </cache-id>
 <cache-id>
  <component id="" type="servletpath">
      <value>/HelloAttr.do</value>
  </component>
  <component id="arg" type="attribute">
      <required>true</required>
  </component>
 </cache-id>
</cache-entry>
```

However, in WebSphere Application Server V6.x, with the support for mapping multiple cache policies for a single servlet, the previous cache policy can be rewritten as shown in Example 4-82.

Example 4-82 Caching policy for WebSphere Application Server 6.x

```
<cache-entry>
 <class>servlet</class>
 <name>/HelloParam.do</name>
 <cache-id>
    <component id="id" type="parameter">
      <required>true</required>
    </component>
 </cache-id>
</cache-entry>
<cache-entry>
 <class>servlet</class>
 <name>/HelloAttr.do</name>
 <cache-id>
    <component id="arg" type="attribute">
      <required>true</required>
    </component>
 </cache-id>
```

```
</cache-entry>
```

Tiles configuration

Because the Tiles framework is built on the jsp:include tag, everything that applies to JSP caching also applies to Tiles. Similar to the jsp:include case, the fragments included using the tiles:insert tag are cached correctly only if the flush attribute is set to true. The only extra feature is caching based on the tiles attribute. For example, consider a page layout defined using the Tiles templating feature, as shown in the layout.jsp file in Example 4-83.

Example 4-83 layout.jsp

```
<html>
<body>
 <%String categoryId = request.getParameter("categoryId")+"test";%>
 <tiles:insert attribute="header">
  <tiles:put name="categoryId" value="<%= categoryId %>" />
 </tiles:insert>
 <TD width="70%" valign="top">
  <tiles:insert attribute="body"/>
 </TD>
 <TR>
  <TD colspan="2"><tiles:insert attribute="footer"/></TD>
 </TR>
</body>
</html>
```

In the foregoing code, a tile attribute categoryId is defined using the nested tiles:put tag. This tile attribute is passed on to header.jsp, for example, when this defined layout is used in a JSP file. Keep in mind that a first JSP insert layout.jsp passing an initialization parameter as shown in Example 4-84.

Example 4-84 A JSP using layout.jsp and passing parameters to others JSPs

```
<html>
<body>
<tiles:insert page="/layout.jsp?categoryId=1002" flush="true">
 <tiles:put name="header" value="/header.jsp" />
 <tiles:put name="body" value="/body.jsp" />
 <tiles:put name="footer" value="/footer.jsp" />
</tiles:insert>
</body>
</html>
```

The header.jsp can retrieve the value of categoryId using <tiles:useAttribute> tag that came from a first page that uses layout.jsp. See Example 4-85.

Example 4-85 Header.jsp example

```
<HTML>
<HEAD>
<TITLE>header.jsp</TITLE>
</HEAD>
<tiles:useAttribute id="categoryId" name="categoryId" />
<%= categoryId%>
<table id="header">
<tr><td><%= System.currentTimeMillis() %></td></tr>
</table>
</HTML>
```

To cache the header.jsp file based on the value of the categoryId attribute, the cache policy shown in Example 4-86 can be used.

Example 4-86 Cache file example

```
<cache-entry>
 <class>servlet</class>
 <name>/header.jsp</name>
 <cache-id>
          <component id="categoryId" type="tiles_attribute">
      <required>true</required>
           </component>
 </cache-id>
</cache-entry>
```

The Struts and Tiles frameworks have been around for quite some time and have become very popular with developers for developing J2EE Web applications. Other than the differences discussed here, Struts and Tiles caching is very similar to servlet and JSP caching. Hence, enabling caching for a Struts and Tiles application is fairly simple if you are already familiar with servlet and JSP caching.

4.9 Java client programming

In some intranet applications, Java client programming is used. Next, we briefly explain what technologies are used for graphical user interfaces.

4.9.1 Abstract Window Toolkit

The Abstract Window Toolkit (AWT) is the original GUI toolkit for Java. It has been enhanced since it was originally introduced, but the basic structure remains the same. The AWT includes the following features:

- A wide range of user interface components, represented by Java classes such as java.awt.Frame, Button, Label, Menu, and TextArea
- An event-handling model to deal with events such as button clicks, menu choices, and mouse operations
- Classes to deal with graphics and image processing
- Layout manager classes to help with positioning components in a GUI
- Support for drag-and-drop functionality in GUI applications

The AWT is implemented natively for each platform's JVM. AWT interfaces typically perform relatively quickly and have the same look-and-feel as the operating system, but the range of GUI components that can be used is limited to the lowest common denominator of operating system components and the look-and-feel cannot be changed.

More information on the AWT can be found at:

`http://java.sun.com/j2se/1.4.2/docs/guide/awt/`

4.9.2 Swing

Swing is a newer GUI component framework for Java. It provides Java implementations of the components in the AWT and adds a number of more sophisticated GUI components, such as tree views and list boxes. For the basic components, Swing implementations have the same name as the AWT component with a J prefix and a different package structure, for example, java.awt.Button becomes javax.swing.JButton in Swing. Swing GUIs do not normally perform as quickly as AWT GUIs, but have a richer set of controls and have a pleadable look-and-feel.

More information on Swing can be found at:

`http://java.sun.com/j2se/1.4.2/docs/guide/swing/`

4.9.3 Standard Widget Toolkit

The Standard Widget Toolkit (SWT) is the GUI toolkit provided as part of the Eclipse Project and used to build the Eclipse GUI itself. The SWT is written entirely in Java and uses the Java Native Interface (JNI™) to pass the calls through to the operating system where possible. This is done to avoid the lowest common denominator problem. The SWT uses native calls where they are available and builds the component in Java where they are not. In many respects, the SWT provides the best of both worlds (AWT and Swing):

- It has a rich, portable component model, like Swing.
- It has the same look-and-feel as the native operating system, like the AWT.
- GUIs built using the SWT perform well, like the AWT, since most of the components simply pass through to operative system components.

A disadvantage of the SWT is that, unlike the AWT and Swing, it is not a standard part of J2SE™ V1.4. Consequently, any application that uses the SWT has to be installed along with the SWT class libraries. However, the SWT, like the rest of the components that make up Eclipse, is open source and freely distributable under the terms of the Common Public License.

More information on the SWT can be found at:

http://www.eclipse.org/swt/

4.9.4 Java components providing a GUI

There are two types of Java components that might provide a GUI:

- Stand-alone Java applications: Launched in their own process (JVM). This category would include J2EE Application Clients, which we come to later.
- Java applets: Normally run in a JVM provided by a Web browser or a Web browser plug-in.

An applet normally runs in a JVM with a very strict security model, by default. The applet is not allowed to access the file system of the machine on which it is running and can only make network connections back to the machine from which it was originally loaded. Consequently, applets are not normally suitable for applications that require access to databases, since this would require the database to reside on the same machine as the Web server. If the security restrictions are relaxed, as might be possible if the applet was being used only on a company intranet, this problem is not encountered.

An applet is downloaded on demand from the Web site that is hosting it. This gives an advantage in that the latest version is automatically downloaded each time it is requested, so distributing new versions is trivial. On the other hand, it also introduces disadvantages in that the applet can often be downloaded several times even if it has not changed, pointlessly using bandwidth, and the developer has little control over the environment in which the applet is run.

4.10 References

In the following sections we list documentation and URLs that you can study for additional information.

4.10.1 JSP best practices

For more information, see the following references:

- Introduction to JavaServer Pages:

  `http://www.ibm.com/developerworks/edu/j-dw-jsp-i.html`

  `https://www6.software.ibm.com/developerworks/education/j-introjsp/index.html`

- Best Practices column in developer works:

  `http://www.ibm.com/developerworks/java/library/j-jspcol.html`

- Implicit JSP Objects — article:

  `http://www.ibm.com/developerworks/java/library/j-pj2ee7.html`

- JSTL (JSP Standard Tag Library) usage:

  `http://www.ibm.com/developerworks/java/library/j-jsp05273.html`

- EL (Expression Language) usage:

  `http://www.ibm.com/developerworks/java/library/j-jstl0211.html`

- How to create a custom tag Library:

  `http://www.ibm.com/developerworks/java/library/j-pj2ee8/`

- Developking Web applications using JSPs and servlets:

  Chapter 11, *Rational Application Developer V6 Programming Guide*, SG24-6449

4.10.2 JSF best practices

For more information, see the following references:

- *Integrating Struts, Tiles, and JavaServer Faces* document, at:

  `http://www.ibm.com/developerworks/java/library/j-integrate/`

- *WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects,* SG24-6361.:

  `http://www.redbooks.ibm.com/abstracts/sg246361.html?Open`

4.10.3 Servlets best practices

For more information, see the following references:

- Best practices to use HTTPsessions:

  http://www-128.ibm.com/developerworks/websphere/library/bestpractices/store_objects_in_httpsession.html

  http://www-128.ibm.com/developerworks/websphere/library/bestpractices/httpsession_performance_serialization.html

  http://www.ibm.com/developerworks/websphere/library/bestpractices/releasing_and_invalidating_httpsessions.html

- WebSphere Application Server Infocenter:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

- Thread safe servlets — article:

  http://www.javaworld.com/javaworld/jw-07-2004/jw-0712-threadsafe.html

- Avoiding or minimizing servlet synchronization — article:

  http://www.ibm.com/developerworks/websphere/library/bestpractices/avoiding_or_minimizing_synchronization_in_servlets.html

4.10.4 Struts best practices

For more information about the Struts Framework, go to:

- "Struts, an open-source MVC implementation":

  http://www.ibm.com/developerworks/ibm/library/j-struts/

- Struts home page:

  http://struts.apache.org/

- Struts with Tiles:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0311_fung_yu/fung_yu1.html

- Building Controller components:

  http://struts.apache.org/1.3.8/userGuide/building_controller.html

5

Business logic layer

This chapter introduces concepts and best practices that apply to the business logic layer. Enterprise Java Beans (EJBs) plays an important part of this layer. In this chapter we discuss best practices for new code, focusing on EJB 3.0. However, we also describe best practices for existing code using EJBs in previous versions such as EJB 2.0/2.1. In addition, you can develop code without using EJBs, and we explain the best practices for these cases.

The chapter is organized into the following major sections:

5.1 Introduction

When developing at the business logic layer, the first thing that you must focus on is the core business of the application. You can complete a business logic implementation using Plain Old Java Objects (POJO), but it is more likely that you will use EJBs.

If you require a business class that must have persistence, remote access, clustering, pooling, transaction control, or cache, use EJB for your project. Otherwise, if you require only transaction management, for example, then you can use the Java Transaction API (JTA). If you want your application to have only distributed computing, then the RMI/RMI-IIOP or Web service approaches are satisfactory.

For simple components to make an application well constructed and reusable, refer to Chapter 2, “Application planning and design” on page 23 for planning and design, and Chapter 3, “General coding considerations” on page 57 for general coding best practices.

In this chapter, we explain best practices to follow to create effective EJBs for your system.

> **Note:** For best practice recommendations of previous versions of EJB (EJB 2.x), see the following links:
>
> `http://www.ibm.com/developerworks/java/library/j-ejbcol.html`
>
> `http://www.ibm.com/developerworks/websphere/techjournal/0501_col_hambrick/0501_col_hambrick.html`
>
> In addition, if you require a basic understanding and best practices for building EJB 2.x applications, see Chapter 16 “Develop Web applications using EJBs” in *Rational Application Developer V7 Programming Guide* , SG24-7501:
>
> `http://www.redbooks.ibm.com/redpieces/abstracts/sg247501.html?Open`

5.2 EJB 3.0 for business logic

In this section, we focus in Enterprise Java Beans 3.0 best practices for business logic using:

- Session beans (stateless/stateful)
- Message-driven beans (MDB)

Entity beans are explained in "Java Persistence API: Entity Beans 3.0" on page 380.

Note: The EJB 3.0 implementation is available in:

- IBM WebSphere Application Server Community Edition V2.0:

  http://www.ibm.com/developerworks/downloads/ws/wasce/?S_TACT=105AGX10&S_CMP=WASCE

- IBM WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0:

  https://www14.software.ibm.com/iwm/web/cc/earlyprograms/websphere/was61ejb3/download.shtml

5.2.1 History of Enterprise Java Beans

The IBM original EJB specification in 1997 is an important development in the world of Java technology. EJBs and the J2EE application servers that contained them were rapidly adopted by the enterprise development world. However, the criticisms of EJBs came just as quickly as the adoption of J2EE. Chief among these criticisms was that EJBs were difficult to understand and tedious to develop.

The EJB specification was designed to solve complex problems, such as distributed computing, transaction management, and data persistence. Complex problems often lead to complex solutions. The original EJB specifications reflect the complex problems they were designed to solve. As a result, developers experienced a lot of pain when working with EJBs. When you understood EJBs, you could use sophisticated tools to make your development easier. There was still a significant learning curve for EJB newcomers.

The EJB 1.0 and 1.1 specifications were developed and released by Sun Microsystems. All subsequent new specifications have been created using Java Specification Requests (JSRs) and approved using the Java Community Process (JCP). The involvement of the community is the key to the evolution of Java technology. The pain felt by EJB developers found a voice in the JCP. The result was the EJB 3.0 specification, finalized in May 2006.

EJBs were no longer just about solving complex problems, they were about solving these problems through easy and straightforward development. The focus of EJB 3.0 is on making initial development easier and applications more maintainable. Here you find out how much simpler a 3.0 EJB is compared to a 2.1 EJB.

5.2.2 Enterprise Java Beans: Reasons to use

In this section we describe some reasons to use session beans. We then extend to other types such as message-driven beans and even entity beans, which are discussed in the next chapter.

Services available from application server

The first reason to use session beans could be the same reason we can find in early versions of EJBs. Session beans allow the developer team to focus on business code, instead of creating services such as transaction demarcation, security, pooling, resource manager, concurrency, thread safe, and so on, which are provided by the EJB container in J2EE (Java 2 Enterprise Edition) application server implementations.

Reusability and ease of use

In early versions of EJBs, there were complications in the component reusability for other systems that did not require these services. We had to follow specific rules of code such as implementing EJBHome interfaces (local and/or remote), EJBObject interfaces (local and/or remote), and a bean. This meant having three to five classes. Also, we were obligated to put in callback methods such as ejbCreate(), ejbRemove() and so on, even if we did not implement any code. In other words, for example, if we would use EJBs in another system for which there is no use of container services, it would be difficult to reuse the same components.

With the POJO approach, EJB 3.0, you can develop an EJB in the same way that you would develop a normal Java class, and the reusability becomes easy. More good news is that you can replace deployment descriptors, which were difficult to maintain in early versions, with annotations beside the code, thus facilitating the service configuration as well.

> **Note:** A special point here is that annotations and deployment descriptors are not mutually exclusive. You can mix them, for example, when an exception case occurs for which you have to put in a special container behavior. In this situation, deployment configuration replaces annotation configuration.

An open Java standard and several support vendors

EJB 3.0 was developed from the Java Community Process with no exclusive group of people. This is an open standard and does not have any dependency of a proprietary solution. There are several players that implement EJB 3.0, such as IBM, Oracle, and BEA. Also, there are open source groups such as Apache (Geronimo) and JBoss.

Note: IBM WebSphere Application Server Community Edition V2.0 (powered by Apache Geronimo) is a free application server implementing Java 5. available at the following URL:

http://www.ibm.com/developerworks/downloads/ws/wasce/?S_TACT=105AGX10&S_CMP=WASCE

EJBs as important players to SOA

A main concept of Service Oriented Architecture (SOA) is the concept of service. A service can be defined as a "*discoverable resource that executes a repeatable task, and is described by an externalized service specification*".

Based on this definition, the best EJB components are service oriented, with functions that are:

- Coarse grained: The functions are designed to minimize the number of calls between layers.
- Stateless: All data associated with a given function is passed in and out on a single call, enabling any client request to be handled by any running instance in any order.
- Mediatable: The data passed in and out is serializable such that it can be easily transformed.
- Adaptable: The linkage between client and service implementation is logical (loosely coupled) rather than physical (tightly coupled). Loose coupling enables a different implementation to be substituted without changing the application, and enables the client and server to be distributed or co-located as required.
- POJO oriented: In early EJB versions, POJO delegate classes were often used to simplify invoking services in a common way, and Helper classes were used to simplify building services. Now with EJB 3.0, this has become easier because of POJO oriented building.
- Exposed as Web services: EJB components, as loosely coupled services, are easily exposed as Web services, facilitating the integration of other services to challenge business requirements to build the Business Application Services layer in SOA. These integrated services are joined with emerging choreography, orchestration, and collaboration technologies such as Web Services Business Process Execution Language (WS-BPEL), Electronic Business XML Business Process Specification Schema (EbXML BPSS) and Web Services Choreography Description Language (WS-CDL).

5.2.3 EJB server and EJB container: Overview

In this section we review the components of the EJB infrastructure.

EJB server

An EJB server is the part of an application server that hosts EJB containers. It is also called an Enterprise Java Server (EJS). WebSphere Application Server is an EJS.

The EJB server provides the implementation for the common services available to all EJBs. The EJB server's responsibility is to hide the complexities of these services from the component requiring them. The EJB specification outlines eight services that must be provided by an EJB server:

- Naming
- Transaction
- Security
- Persistence
- Concurrency
- Life cycle
- Messaging
- Timer

EJB container

The Enterprise Java Beans container functions as a runtime environment for enterprise beans by managing and applying the primary services that are required for bean management at runtime. In addition to being an intermediary to the services provided by the EJB server, the EJB container also provides for EJB instance life cycle management and EJB instance identification. EJB containers create bean instances, manage pools of instances, and destroy them.

Containers are transparent to the client in that there is no client API to manipulate the container, and there is no way for a client to tell in which container an enterprise bean is deployed.

One of the container's primary responsibilities is to provide the means for remote clients to access components that live within them. Remote accessibility enables remote invocation of a native component by converting it into a network component. EJB containers use the Java RMI interfaces to specify remote accessibility to clients of the EJBs or Web Services end points.

The responsibilities that an EJB container must satisfy can be defined in terms of the primary services. Specific EJB container responsibilities are as follows: Note the similarity to the list in "EJB server" on page 256. This is due to the unspecified division of responsibilities between the EJB server and container.

Naming

The container is responsible for registering the unique lookup name in the JNDI namespace when the server starts up, and binding the appropriate object type into the JNDI namespace.

Transaction

The EJB container can handle the demarcation of transactions automatically, depending on the EJB type and the transaction type attribute, both described in the EJB module's deployment descriptor. When the container demarcates the transactions, applications can be written without explicit transaction demarcation code (for example, begin, commit, rollback).

Security

The container provides security realms for enterprise beans. It is responsible for enforcing the security policies defined at the deployment time whenever there is a method call, through access control lists (ACL). An ACL is a list of users, the groups they belong to, and their rights, and it ensures that users access only those resources and perform those tasks for which they have been given permission.

Persistence

The container is also responsible for managing the persistence of a certain type of bean (discussed later in this chapter) by synchronizing the state of the bean's instance in memory with the respective record in the data source.

Concurrency

The container is responsible for managing the concurrent access to components, according to the rules of each bean type.

Life cycle

The container controls the life cycle of the deployed components. As EJB clients start sending requests to the container, the container dynamically instantiates, destroys, and reuses the beans as appropriate. The container can provide some resource utilization optimizations, and employ techniques for bean instance pooling.

Messaging

The container must provide for the reliable routing of asynchronous messages from messaging clients (JMS or otherwise) to message-driven beans (MDBs). These messages can follow either the peer-to-peer (queue-based) or publish/subscribe (topic-based) communication patterns.

Timer

Enterprise applications can model business processes that are dependent on temporal events. To implement this characteristic, the container must provide a reliable and transactional EJB Timer Service that allows callbacks to be scheduled for time-based events. Timer notifications can be scheduled to occur at a specific time, after a specific elapsed duration, or at specific recurring intervals.

5.2.4 Session beans

Session beans play an important role when you write business logic. In EJB 2.1 and prior versions, session beans were often used. With EJB 3.0, this process becomes simpler and closer to normal Java Class (like POJO). Before we go more deeply into the behavior of session beans for EJB 3.0 as a best practice, we must understand what is meant by a session, as well as the basic concepts: stateless and stateful. After that, we explain motivations for their use.

What is a session?

A session is a connection between a client and a server, and this connection exists for a finite period of time. A session can be *stateless*, meaning that no state of requests is maintained between invocations. An HTTP request and a session can be *stateful* when a state is maintained between invocations — for example, a Telnet session to a server in which the user can type several commands, where each command is an invocation and the state is maintained, for example, the UNIX ID that is doing the commands.

Motivations to use session beans

In 5.2.2, “Enterprise Java Beans: Reasons to use” on page 254 we explained some generic reasons to use EJBs. Next, we explain specific motivations to use session beans:

- Thread safe and concurrency: If client requests are directly handled by your business logic and there are remote demands of clients at same time to use the service. In this case, thread safe and concurrency plays an important point, and by using session beans, the container provides this service to you.
- Expose your service remotely and with Web services: The use of EJBs as Web services is discussed in “EJBs as important players to SOA” on page 255. Also, if you want to expose your service remotely, the best choice includes the Remote Method Invocation (RMI) and the use of session beans.
- Security and transaction: If you require a service with security features and you can use EJB container services that contains J2EE built-in security rules ready to use. Moreover, EJB is integrated with Java Authentication and Authorization Service (JAAS).

Also, if you require a service that controls transaction integrity of other sub-tasks in the principles of atomicy, consistency, isolation, durability (ACID), the container EJB can do this for you.

- Interceptors: A nice addition to the EJB specification is the use of interceptors. One thing missing from EJB components has been the ability to perform Aspect Oriented Development (AOP) for things such as pre/post-processing and cross cutting concerns, much like servlet filters do for servlets. You can now develop an interceptor class and apply it to a bean. This feature is applied when the code is not related to business logic; for example, you can use interceptors for login and auditing. We explain this later as a best practice.
- Timer services: This feature has been offered since the EJB 2.1 specification. If you require a schedule service, session beans are a good choice. This requires that you implement an interface, as we explain later.

Session beans: Types

The difference between session beans and other types is that they are short lived objects whose lifetime exists only, for example, while processing a client request (*stateless* mode), in a request/response mode, or while a client is using the session (*stateful* mode), such as in an Internet purchase, while putting several products in a basket before sending the order. This means that in a *stateful* session, there is a conversational state with the client, and in a *stateless* session, there is no conversational state. Session beans do not represent data in a database (entity beans do) and do not manage asynchronous processing directly from a request (message-driven beans do).

Session beans: Format

Because EJB 2.x programs require that you extend specific classes, provide several interfaces, and write deployment descriptors, they are viewed as "overloaded" Java objects that are no longer plain; rather, you require a J2EE container to run and test them. EJB 3.0 changes that:

- EJB components no longer require home interfaces. In addition, EJB components are no longer required to provide different interfaces or extend any EJB-specific classes.
- J2SE 5.0 annotations are now a major facilitator for implementing EJB 3.0 components. By specifying special annotations, developers can create POJO classes that are EJB components, as an alternative to XML.
- EJB 3.0 introduces the notion of a business interface, rather than separate remote and local interfaces.

The basic format of a session bean is shown in Figure 5-1.

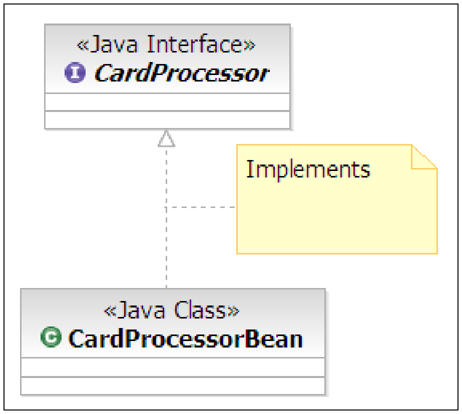


Figure 5-1 Session bean format example

Looking at this example, we have two different parts in a session bean format:

- **One or more business interfaces:** Here, the client invokes the bean in a loosely coupled way. We put the declaration of bean methods there. The interesting point here is about the polymorphic way that a single EJB can implement different interfaces for different clients. For example, a single CardProcessorBean is available for customers using the CardProcessor interface and the CardProcessorAdmin interface (see Figure 5-2).

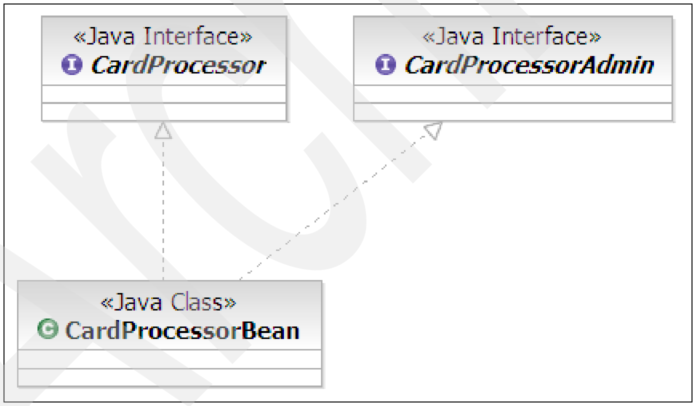


Figure 5-2 A session bean implementing multiple interfaces

Going back to early versions of EJB, we could have four interfaces, two for EJBHome (EJBHome and EJBLocalHome) used to control the bean life cycle, and two for EJBObject (EJBObject and EJBLocalObject). EJB 3.0 has greatly simplified the construction.

Now that we have one interface, how do we specify whether the methods are remote or local? We use annotations that were provided with J2SE 5.0.

In Example 5-1, we used the @Remote annotation for customer access in the CardProcessor interface.

Example 5-1 Using annotations in business interface CardProcessor

```
package com.ibm.itso.sg247497.ejb3;

@Remote
public interface CardProcessor {
  public double balance(String card) throws Exception;
  public boolean check(String card) throws Exception;
  public void debit(String card,double amount) throws Exception;
  public void credit(String card,double amount) throws Exception;
}
```

In Example 5-2, we used the @Local annotation for interface CardProcessorAdmin.

Example 5-2 Using annotations in business interface CardProcessorAdmin

```
package com.ibm.itso.sg247497.ejb3;

@Local
public interface CardProcessorAdmin {
  public void block(String card) throws Exception;
  public void invalidate(String card) throws Exception;
}
```

- **The EJB bean class:** This provides the concrete implementation of the business interface. The bean class is a normal Java class and the implementation is done using the implements clause. See Example 5-3.

Example 5-3 Bean class implementation example

```
package com.ibm.itso.sg247497.ejb3;

@Stateless
public class CardProcessorBean implements CardProcessor,
CardProcessorAdmin {

  //CardProcessor implemented methods
  public double balance(String card) throws Exception {
   //Put your Business Logic Here
   return 0;
```

```
    }

    public boolean check(String card) throws Exception {
      //Put your Business Logic Here
      return true;
    }

    public void debit(double amount) throws Exception {
      //Put your Business Logic Here
    }

    //CardProcessorAdmin implemented methods
    public void block(String card) throws Exception {
      //Put your Business Logic Here
    }

    public void invalidate(String card) throws Exception {
      //Put your Business Logic Here
    }
  }
```

Looking at the previous example, we can see the @Stateless annotation that indicates a stateless session bean. If you have to use a stateful session bean, you must use @Stateful. This informs the EJB container how to manage the bean. We see the particulars and best practices in the next sections.

Programming rules to write a session bean

Next we discuss the rules to follow when building a session bean.

Business interface

A session bean requires at least a business interface.

> **Note:** You can annotate either the business interface or the bean class with @Local or @Remote. Enabling the annotation on the bean class is useful when you choose to let the container generate the business interface for you.

No final or abstract class

A session bean class cannot be implemented as final or abstract because the container must manipulate it. However, you can subclass another session bean or another POJO class.

Default no arg constructor

A default no arg constructor is required in a session bean class. If there are no constructors defined in the class, the compiler puts one in by default. If you defined other constructors, you must define a no arg constructor.

Business methods definition

Business methods can be defined either in a superclass or a bean class. However, it should be public and not final or static.

Callback methods and life cycle (overview)

Life cycle callbacks are bean methods and are not exposed in a bean interface that the container invokes during a life cycle transition or event. You can define callback methods either in superclass or bean class. Prior to EJB 3.0, you had to implement callback methods, such as ejbCreate(), on the bean class; bean classes had to implement all the methods, whether they used them or not. In most cases, these method implementations were empty. Callbacks are now handled through annotations, using either callback methods or callback listener classes.

There are two events in a bean life cycle that are common for all session beans: creation and destruction. These methods are defined with the @PostConstruct and @PreDestroy annotations. Two more events for stateful session beans, called passivation and activation, are represented with the @PrePassivate and @PostActivate annotations.

To write a method with these annotations, you can make it public, private, protected, or default (package).

In Example 5-4. we write code to respond to a callback using callback methods.

Example 5-4 Using callback methods

```
@stateless
public class CardProcessorBean implements CardProcessor {

  //CardProcessor implemented methods
  public double balance(String card) throws Exception {
   //Put your Business Logic Here
   return 0;
  }

//Other business methods

  @PostConstruct public void initializeResouces(){
    connection = datasource.getConnection();
```

```
  }

  @PreDestroy public void deliverResources(){
    connection.close();
    connection = null;
  }
}
```

The previous code enables you to implement code after a bean instance is created and before destroying the instance by container. If you wanted to use a callback listener, you could create a callback listener class. See Example 5-5.

Example 5-5 Using callback listener

```
public class CardProcessorCallbackListener
{
  @PrePassivate public deliverResources(Object obj)
  {
    CardProcessor card = (CardProcessor) obj;
   //perform logic
  }
}
```

A callback class that is not part of the bean class has to take in a java.lang.Object parameter. The container then passes the bean instance. A bean class adds the callback listener class by using a special callback annotation at the bean class level. See Example 5-6.

Example 5-6 Referencing callback listener in bean class

```
@CallbackListener CardProcessorCallbackListener
@stateless
public class CardProcessorBean implements CardProcessor {

  //CardProcessor implemented methods
  public double balance(String card) throws Exception {
   //Put your Business Logic Here
   return 0;
  }

  //Other business methods
}
```

Avoid using the word "ejb" in business methods declarations

Avoid using business methods that start with "ejb" because it can cause problems with infrastructure processing.

Use java.io.Serializable

If you intend to use remote invocations, make sure that the parameters and return type implement java.io.Serializable.

Stateless session beans

Stateless session beans do not maintain the conversational state with the client. This means there is no guarantee that the same session bean responds to the same client between requests. Stateless beans are pooled; in other words, it seems that there are a number of instances in the pool to service clients. See Figure 5-4.

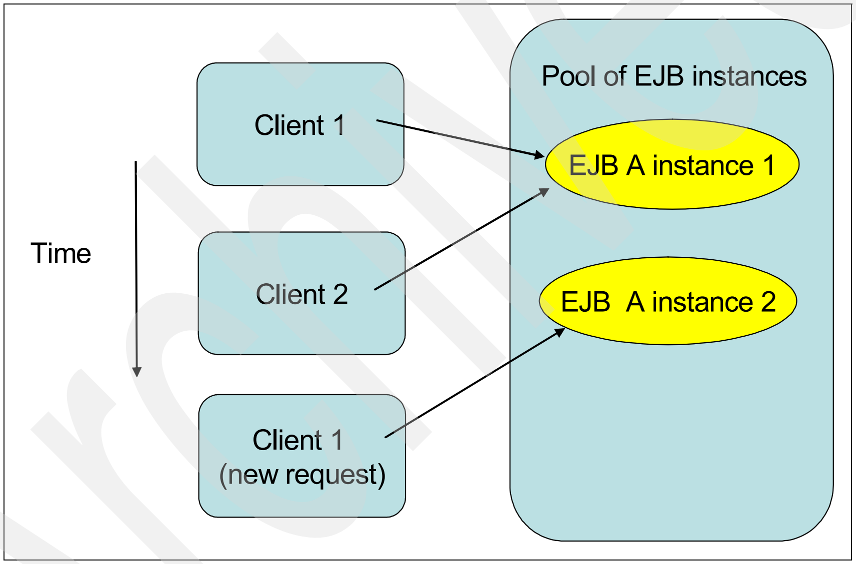


Figure 5-3 Execution example of some requests to ejb stateless pool

In this example, the first request from client 1 to EJB A used EJB A instance 1. The next request from client 1 used EJB A instance 2. A request from client 2 uses EJB A instance 1. In other words, there is no guarantee that EJB A instance 1 serves all client 1 requests.

Stateless life cycle (overview)

To understand the life cycle of a stateless session bean, see Figure 5-4.

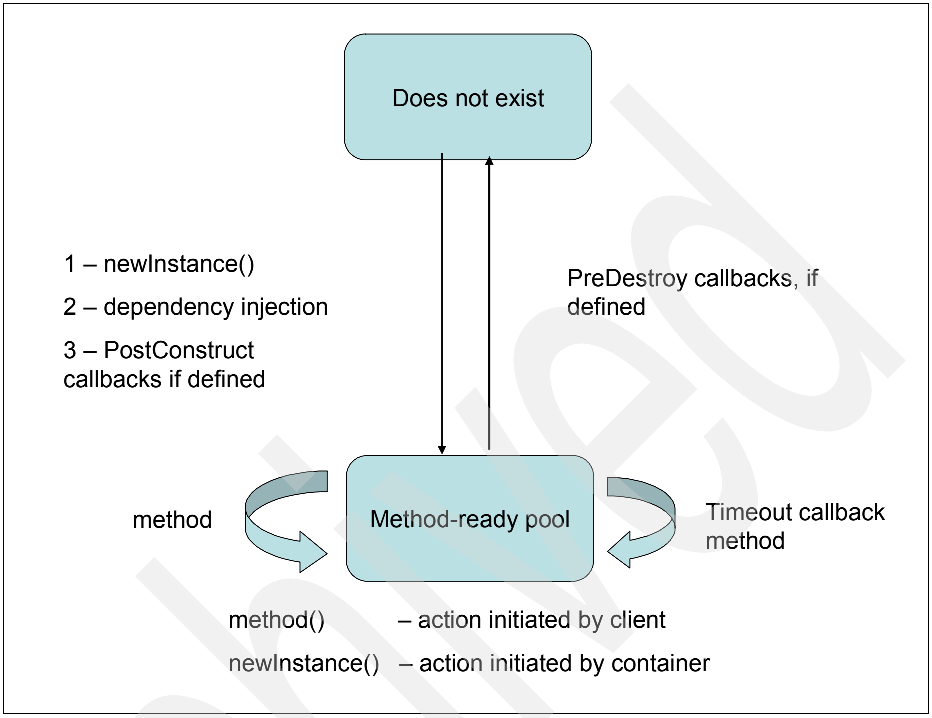


Figure 5-4 Stateless session bean life cycle

The first step in the life cycle is when the newInstance method is invoked on the session bean class to instantiate a bean class. The container injects the bean's SessionContext and performs any other applicable Dependency Injection that is specified by metadata annotations on the bean class or by the deployment descriptor. The container then calls the PostConstruct life cycle callback interceptor methods for the bean, if defined. The container can perform the instance creation at any time with no direct relationship to a business invocation from client.

In the method-ready pool state, the session bean instance is now ready to receive a business method as a delegated call from any client or a call from the container to the timeout callback method.

Usually, when the container wants to reduce the number of instances in the method-ready pool and no longer requires the instance, the container invokes the PreDestroy life cycle callback interceptor methods for it, if defined. This event finalizes the life of the stateless session bean instance.

Annotations used in stateless session beans

In Table 5-1 we explain the annotations used in stateless session beans as a summary of annotation concepts.

Table 5-1 Annotations used in stateless session beans

Annotation	Purpose	Usage example

@Stateless	Marks a bean class (POJO class) as a stateless session bean.	`@Stateless`
		`public class CardProcessorBean`
		`implements CardProcessor`

@Local	Used for local interfaces designated for clients that access the session bean instance from the same JVM.	`@Local`
		`public interface`
		`CardProcessorAdmin`

@Remote	Used for remote interfaces designated for clients that access the session bean instance from another JVM.	`@Remote` `public interface CardProcessor`
@WebService	Exposes a stateless session bean as a SOAP based Web service to be invoked by a .NET application, for example.	`@WebService` `public interface CardProcessor`
@PostConstruct	Used as a container callback method after bean is constructed.	`@PostConstruct public void` `initializeResouces()`
@PreDestroy	Used as a container callback method before bean is destroyed.	`@PreDestroy public void` `deliverResources()`

@Resource	Used to inject resources such as data sources. The parameter is a JNDI and tells the container to find a resource in a described JNDI location and assign to parameter variable. This step is done after bean creation.	`@Resource(name="jdbc/UserCardDS"`
		`)`
		`public void`
		`setDataSource(DataSource`
		`datasource){`
		`   this.datasource = datasource;`
		`}`

For more information on callback methods, see “Callback methods and life cycle (overview)” on page 263.

Note: When you use a remote interface, you can extend java.rmi.Remote. However, this is not mandatory, and the EJB container inserts it when performing byte-code enhancements.

Stateful session beans

Stateful beans are used when you must record and remember the conversational state between client requests. In other words, the next request has to know the previous request state. There is no huge difference between stateless and stateful beans. The most significant changes refer to the way that the container manages resources. See Figure 5-5.

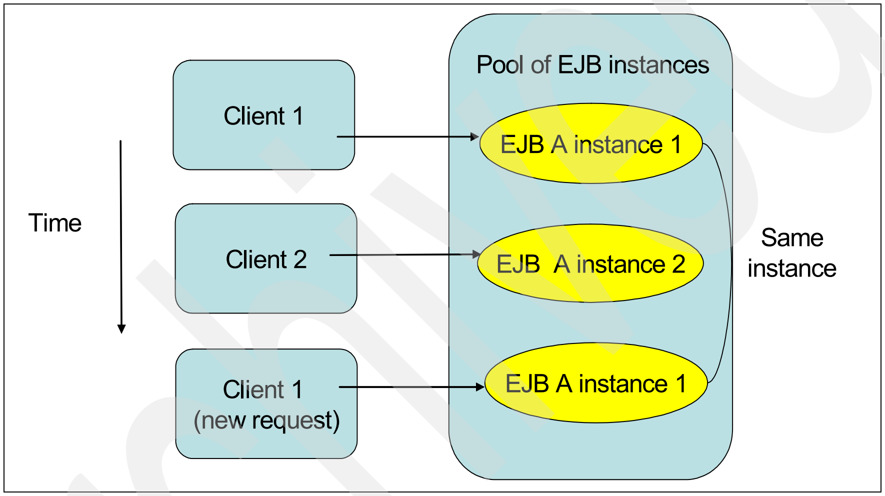


Figure 5-5 Execution example of some requests to EJB stateful pool

In this case, EJB A instance 1 is used for all client 1 requests. EJB A instance 1 is held by client 1 until the session is disconnected, making this instance unavailable to other clients. If a large number of concurrency clients call a stateful EJB, this could cause a drawback in terms of memory usage. To alleviate this problem, a technique called passivation and activation is used.

Passivation means that the container serializes and saves the bean instance into a persistence storage such as a file or database when the container decides to save resources. For example, when the client is idle and the session timeout has not been reached, or a high number of concurrency clients are reached.

In activation, the container does the opposite process, retrieving a requested bean instance from a persistence storage, deserializing, and moving back to memory.

When you code a stateful session bean, all variables for which you must keep state should be put in as Java primitive types or implementing java.io.Serializable. Variables that you do not need to keep state for can be put in a transient attribute.

Stateful bean life cycle (overview)

In Figure 5-6, notice that the stateless session bean life cycle is more complex and very different from stateless session beans. This situation is caused by passivation and activation control, which are necessary.

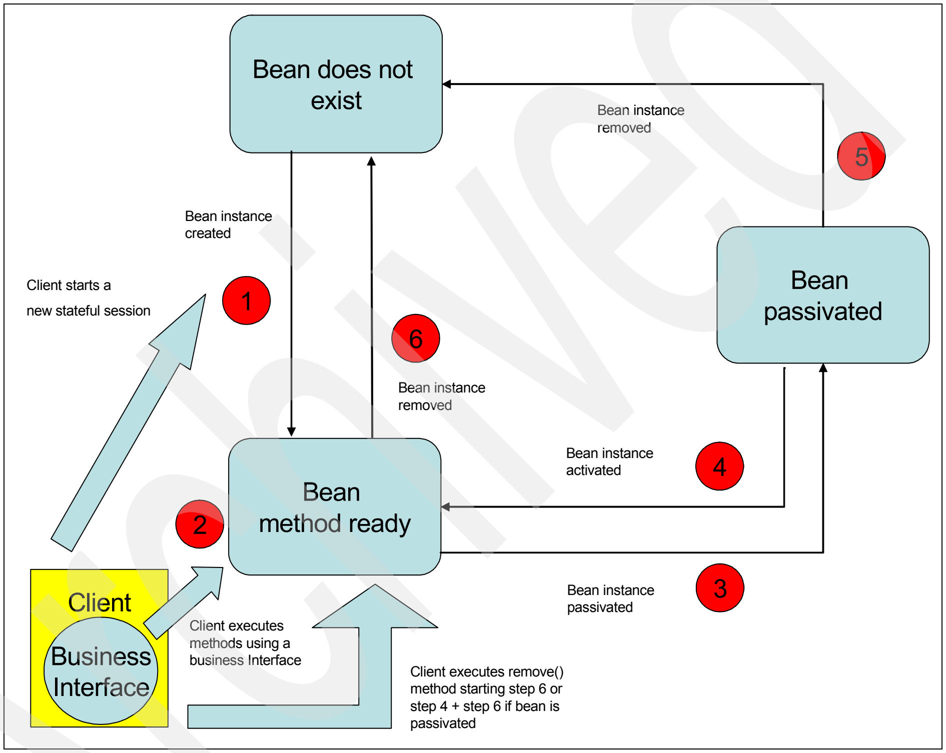


Figure 5-6 Stateful bean life cycle

Looking at Figure 5-6, note that stateful beans do not have a pool state. See the following numbered steps, corresponding to the numbers in the figure:

1. A new bean instance is created using a default constructor when a new client session is started. Resources described in @Resource annotations are injected and the bean is left in memory.
2. In method ready state, the client executes methods on a stateful bean using a business interface. The container waits for the next request, counting idle time between requests.

3. If an idle time limit is reached, the container passivates the bean instance. This means that the container moves this bean from memory; it is serialized and persisted in a temporary storage managed by the container.
4. If a client invokes the passivated bean again, the container loads the passivated bean from a temporary storage.
5. If a client does not invoke a passivated bean within a period of time, the bean is destroyed.
6. If a client executes a method to remove, this action starts at step 6. If the bean is passivated, step 4 is executed first before removal.

Annotations used in stateful session beans

Stateful beans use the same annotations used by stateless beans except for @WebService, which is a SOAP-based Web service with stateless behavior. See Table 5-1 on page 267. Beyond the annotations used by stateless beans, stateful beans use the coding shown in Table 5-2.

Table 5-2 Annotations used only for stateful beans

Annotation	Purpose	Usage example

@PrePassivate	Programs some logic before the bean state is safe into persistence storage. For example, java.sql.Connection cannot be serializable; then it is a good practice to deliver the connection to the data source pool.	`@PrePassivate`
		`@Remove()`
		`public void`
		`deliverResources(){`
		`    if (con!=null)`
		`      con.close();`
		`    con = null;`
		`}`

@PostActivate	Programs some logic after the bean is retrieved from persistence storage to memory. It can be used to get a new database connection to use.	`@PostActivate`
		`  public void getResources(){`
		`    //get Datasource code`
		`    con =`
		`datasource.getConnection();`
		`}`

@Remove	Indicates that a method that is using this annotation is be executed by the container before destroying the bean instance.	`@PrePassivate`
		`@Remove()`
		`public void`
		`deliverResources(){`
		`    if (con!=null)`
		`      con.close();`
		`    con = null;`
		`}`

Note: You can see that the examples of @PrePassivate and @Remove are the same. We can refer more than one annotation to a unique method. The container in this example executes the deliverResouces() method two times before the bean is passivated to persistence storage because the java.sql.Connection is not serialized, and before destroying the bean instance, in order not to hold resources that are never used.

Some performance considerations for stateful session beans

When we code a stateful bean, we must take care of the following issues according to best practices:

- Memory, disk, and network considerations: Because stateful session beans maintain state between invocations, you must be aware of the amount of data that is being passivated. This concern is the same as when you use an HttpSession in a servlet. Since the container stores session data information in memory, putting a large amount of data in a stateful session can cause performance drawbacks and disk space problems for passivation. Consider using simpler types of data, for example, AccountID instead of the whole Account object. Network overhead may also be a consideration if stateful beans are clustered, because the state is replicated between different instances of the EJB container.
- EJB container tuning for passivation rules: Consider tuning the EJB container for passivation rules to get more performance. For example, if you want to avoid the passivation and activation process, you have to make the maximum number of active instances and the maximum number of clients equal. See Chapter 7, "Environmental performance considerations" on page 509 for more information about WebSphere Application Server 6.1 best practices for configuration.
- Alternatives to using stateful beans: If you are using a Web application, for example, servlets, JSPs, and associated framework such as Struts, JSF, and others, consider using the session data managed by the Web container, which performs better than stateful session beans. However, if you are developing a J2SE client for which you must keep state between invocations with an application server, stateful session beans could be a good choice.

  For more information on HttpSession best practices, see 4.5.2, "HttpSession best practices" on page 173 for best practices for the presentation and control layer.

  Another option could be to write a stateless session bean with logic to persist data in the database and load data from it, keeping the state between invocations. But this approach is not recommended for two main reasons: proprietary code to maintain, and no guarantees for scalability and cluster effectiveness issues when these services are offered by the EJB container.

Programming rules to write client code

The general rules for writing a session bean are as follows:

1. The client obtains a reference to the beans using Dependency Injection (DI) or using Java Naming and Directory Interface (JNDI) lookups.
2. All session calls are done using the business interface to accomplish the business requirements.
3. In the case of a stateful bean, as a best practice, the last invocation is the remove() method to deliver resources more quickly.

The difference between DI and JNDI calls

DI is a new feature in EJB 3.0 that leaves the dependencies between components loosely coupled. The idea is that one component calls another component or resource using only interfaces, and we can glue components and resources together using configuration instead of code. One desirable result of this is that the application can have its implementation changed by using a simple application reconfiguration.

> **Note:** To learn more about the Dependency Injection pattern, see this URL:
>
> http://www.martinfowler.com/articles/injection.html

Using JNDI means that our code does a manual lookup and the bean retrieves resources and components that it requires explicitly. The result is that dependencies of other components and resources are hard-coded inside the bean. In other words, you can think of DI as an abstraction of JNDI lookups.

Using Dependency Injection

Example 5-7 shows a servlet that invokes an EJB for processing some tasks. For this goal we use @EJB annotation.

Example 5-7 Servlet invoking an EJB

```
public class ServletCard extends HttpServlet{

       @EJB
       private CardProcessor cardProcessor;

       public void service(HttpServletRequest request,
              HttpServletResponse response) throws ServletException,
IOException{
           String cardId = request.getParameter("cardId");

           if (cardProcessor.check(cardId)){
```

```
                double ammount = cardProcessor.balance(String card);
            }
            //Other code parts
        }
}
```

As we can note, regardless of EJB annotation, the cardProcessor variable is used as a simple local Plain Old Java Object. Another interesting point is that behind the scenes, when the servlet is loaded, the servlet container looks up an instance of CardProcessor EJB, CardProcessorBean in the JNDI tree, and sets the cardProcessor variable with the retrieved EJB reference. If the EJB is remote, the container looks up the EJB too. In Example 5-8, an EJB is calling another EJB using DI.

Example 5-8   EJB calling another using Dependency Injection

```
@stateless
public class CardProcessorBean implements CardProcessor {
  @EJB
  Account account;

  public void payCard(String accountID,String card,double amount){
        if (account.balance(accountID) < amount){
            //Process InsufficientFundsException
        }
        else {
            account.withdraw(amount);
            credit(String card,double amount);
            //Other code Parts
        }
  }
  //Another code parts
}
```

Also, we can call an EJB from a client in a JVM other than the application server as a J2EE client container application. See Example 5-9.

Example 5-9   J2EE client container example

```
//Client as a J2EE client container

package com.ibm.itso.sg247497.ejb3.client;

import javax.ejb.EJB;
```

```
import com.ibm.itso.sg247497.ejb3.CardProcessor;

public class RemoteEJBCard {

   @EJB(beanName = "CardProcessorBean")
   private static CardProcessor cardProcessor;

   public static void main(String[] args) {

      try {
         char option = ' ';
         while (option != '0') {
            System.out.println("\n\nEnter a number for the operation:"
                  + "\n\t0 - Exit" + "\n\t1 - get card balance");
            option = (char) System.in.read();
            switch (option) {
            case '0':
               break;
            case '1':
               System.out.println("\n\nEnter the card number:");
               byte[] inputData=null;
               System.in.read(inputData);
               String card = new String(inputData);
               System.out.println("\nThe balance for card " + card
                     + " is : " + cardProcessor.balance(card));
               break;
            default:
               System.out.println("\nInvalid option - " + option);
               break;
            }
            System.in.skip(4);
         }
      } catch (Exception e) {
         System.out.println("Unexpected exception:");
         e.printStackTrace();
      }
   }
}
```

Using Java Naming and Directory Interface

Next, we describe various circumstances for using this interface:

- Outside the container, with a helper class:

  When you call an EJB using another JVM from outside the container, it is similar to previous versions of EJB except that is not necessary for the home interface to instantiate a bean. See Example 5-10.

  Example 5-10 EJB Client code

```
public void payCard(String account,String card,double amount){
        InitialContext context = new InitialContext();
        Account account =
(Account)context.lookup("java:comp/env/ejb/Account");
        if (account.balance(accountID) < amount){
            //Process InsufficientFundsException
        }
        else {
            account.withdraw(amount);
            //Other code Parts
        }
}
```

- Inside the container, with an EJB accessing another EJB:

  Consider the case where a servlet is accessing an EJB or a resource, for example, a JDBC resource, an EJBContext, or a JMS resource. In this case, we use two annotation types: @EJB and @Resource.

  To get access to another EJB, the first step is to inform the container that you require access to another EJB using the @EJB annotation defining a dependency. See Example 5-11.

  Example 5-11 Referencing another EJB for call with @EJB annotation

```
@EJB(name="ejb/Account" beanInterface=Account.class)
@stateless
public class CardProcessorBean implements CardProcessor
```

  Note that you must put, in the name parameter, the JNDI complete path to reach this other EJB and the business interface class.

  The next step is getting access to container services using the same class used in previous versions of EJB, that is a SessionContext. However, we use annotation @Resource, which informs a container to instantiate the SessionContext for you. This technique is called injection (Example 5-12).

Example 5-12 SessionContext instantiation with @Resource annotation

```
@EJB(name="ejb/Account" beanInterface=Account.class)
@stateless
public class CardProcessorBean implements CardProcessor {
  @Resource
  SessionContext context;
  //Another code parts
}
```

Now we invoke the target EJB inside a method (Example 5-13).

Example 5-13 Getting another EJB using SessionContext lookup method

```
@stateless
public class CardProcessorBean implements CardProcessor {
  @Resource
  SessionContext context;

  public void payCard(String accountID,String card,double amount){
        Account account = (Account)context.lookup("ejb/Account");
        if (account.balance(accountID) < amount){
            //Process InsufficientFundsException
        }
  }
  //Another code parts
}
```

Note: In regard to @EJB usage, consider the following situation:

```
@EJB CardProcessor cardProcessor1;
@EJB CardProcessor cardProcessor2;
...
if (cardProcessor1.equals(cardProcessor1)) { // this test must
return true
...
}
...
if (cardProcessor1.equals(cardProcessor2)) { // this test must also
return true
...
}
```

if you are using the same business interface from the same session bean type, the equals() method always returns true.

Using Web services

A Web service client accesses a stateless session bean through the Web service client view, which is described by the Web Services Description Language (WSDL) document for the Web service that the bean implements. @WebService annotations are used in method declarations that are exposed as Web services and are only available for stateless session beans. For more information, see “Annotations used in stateless session beans” on page 267.

The Web service client view of an enterprise bean is location independent and it can be remote. Web service clients might be Java clients and/or clients not written in the Java programming language. A Web service client that is a Java client accesses the Web service by means of JAX-WS, which is a new programming model for Web services from J2EE 5, or JAX-RPC client APIs. Access through Web service clients occurs through SOAP 1.1, SOAP 1.2, or plain XML over HTTP or HTTPS.

> **Note:** For more information about JAX-WS and JAX-RPC difference and other specifications, see the following articles:
>
> http://www.ibm.com/developerworks/webservices/library/ws-tip-jaxwsrpc.html
>
> http://www.ibm.com/developerworks/websphere/library/techarticles/0707_barcia/0707_barcia.html
>
> Also see the Web services feature pack for WebSphere 6.1:
>
> http://www-1.ibm.com/support/docview.wss?rs=180&uid=swg21264563

To conclude this topic, we discuss a complete example of calling a stateless session bean using Web services. First we start with a stateless session bean configuration. We can use our previous example with CardProcessorBean (Example 5-14).

Example 5-14 Annotations used example in stateless beans for Web Services

```
@Stateless
@WebService(serviceName="CardServices", portName="CardServicePort")
public class CardProcessorBean implements CardProcessor {

  @WebMethod
  public double balance(String card) throws Exception {
   //Put your Business Logic Here
   return 0;
  }
}
```

As you can see, we did not do any coding for Web services features. We put in @WebService annotation with two members: serviceName and servicePort that tell the container about the name and port name of the Web service, and these configurations appear in the <service> tag of a WSDL config file (Example 5-15).

Example 5-15 WSDL service tag definition of a session bean

```
<service name="CardServices">
  <port name="CardServicePort" binding="tns:CardServicePortBinding">
     <soap:address location="http://localhost:80/CardService"/>
  </port>
</service>
```

Another annotation, @WebMethod, as shown in Example 5-14 on page 277, exposes a method in the WSDL file. But this annotation is optional; if omitted, all methods are exposed in the WSDL file.

> **Note:** To learn about some initial Web services concepts, refer to 6.12, "Web Services" on page 471.
>
> Because a WSDL file is an XML file, you can find some background information about XML at the following link:
>
> http://www.ibm.com/developerworks/edu/x-dw-xmlintro-i.html
>
> To learn more about the WSDL structure, see Section 2, "Service Definition" at the following link:
>
> http://www.w3.org/TR/wsdl

After configuring the bean with these annotations, we must redeploy the application. The EJB container knows how to receive incoming SOAP message requests to the bean and properly dispatch the bean doing a mapping between XML data as a SOAP request to Java. When the response is completed from the EJB method, the container knows how to translate the Java return data to the SOAP response message. These mapping rules between XML data and Java are a feature of the Java Architecture for XML binding (JAXB).

To learn more about JAXB, go to following link:

http://www.ibm.com/developerworks/xml/library/x-pracdb2.html

Example 5-16 shows client code that accesses the session bean Web service.

In this sample, we used JAX-WS 2.0, which is the successor to JAX-RPC 1.1.

Example 5-16 Web service client code

```
//Web Service Client (StandAlone) using JAX-WS

package com.ibm.itso.sg247497.ejb3;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class WSClientCard{

  static String host = "localhost";
  static String portType = "CardProcessor";
  static String serviceName = "CardServices"
  static String serviceEndPointAddress =
                "http://"+ host+":8080/"+ serviceName;
  static String nameSpace = "http://itso.sg247497.ejb3";

  public static void main(String[] args) throws Exception{

    System.out.println("\n\nEnter the card number:");
    byte[] inputData;
    System.in.read(inputData);
    String card = new String(inputData);

    URL wsdlLocation = new URL(serviceEndPointAddress + "/" + portType
+ "?WSDL");
    QName serviceQName = new QName(nameSpace, serviceName);

    //Dynamic service mode usage
    Service service = Service.create(wsdlLocation, serviceQName);
    CardProcessor cardPort = service.getPort(CardProcessorBean.class);
    System.out.println(" Dynamic service test
:"+cardPort.balance(card));

   //Static service mode usage
   CardServices card = new CardServices();
   CardProcessor cardPort = card.getCardServicePort();
   System.out.println(" Static service test :"+cardPort.balance(card));
}
```

Session beans: Best practices

In the following topics, we discuss best practices for session beans.

Use stateless session beans instead of stateful session beans

In most cases, stateless session beans are the best choice instead of stateful beans because of the resources required, such as disk, memory, and network bandwidth, to implement a stateful approach. Implementations that avoid storing the client user state scale and perform the best. You should design implementations to avoid storing state. Using stateless session beans instead of stateful beans makes your system more resilient to failover.

If state storage is required, use the HttpSession to store user-specific state. Ensure that the size of the state data and the time that the state is stored are kept to the smallest possible values. J2EE application servers providing for stateful session bean failover can work around some issues, but stateful solutions are not as scalable as stateless ones. The use of stateful session beans pushes state to your application server, which is undesirable. It increases system complexity and complicates failure scenarios. One of the key principles of robust distributed systems is stateless behavior whenever possible.

We recommend that you choose a stateless session bean approach for most applications. Any user-specific state necessary for processing should either be passed in as an argument to the EJB methods (and stored outside the EJB through a mechanism like the HttpSession) or be retrieved as part of the EJB transaction from a persistent back-end store (for instance, through the use of entity beans). Where appropriate, this information can be cached in memory, but beware of the potential challenges that surround keeping the cache consistent in a distributed environment. Caching works best for read-only data.

Prefer local interfaces rather than remote interfaces

If you want to make sure that your client invokes an EJB from the same JVM, use local interfaces. Remote interfaces involve network resources such as Remote Method Invocation (RMI) or SOAP access and use more resources than local interfaces.

Remote object invocations are expensive compared to local calls, requiring request and return object serialization and de-serialization. Use as many local object accesses as possible, and where feasible, deploy all application components in the same tier. For example, having Web and EJB components in the same tier ensures local access to EJBs. You should use the EJB “local” APIs to avoid request and response object pass-by value semantics (requiring serialization and de-serialization).

Avoid accessing EJB entity beans from the presentation layer. Instead, use a session facade to contain complex interactions and reduce the number of distributed business objects accessed by the presentation layer. When a client application accesses an entity bean directly, each getter method is a remote call.

A session facade can access the entity bean locally, collect the data in a structure, and return it by value. When remote objects are required, design course grain interfaces to reduce the number of invocations required and make sure that only the minimal required data is sent and retrieved.

Be careful when using Dependency Injection (DI)

When you use Dependency Injection, be careful to not use a stateful session bean referring to a stateless session bean. This is a concern because when you use an injection, the variables used are instance variables and have the class scope. Then, in this case, a global variable of a stateless bean is in the stateful bean class. When the stateless bean comes back to the pool, the state of the stateful bean turns inconsistent.

Use session facade for better results

In previous versions of EJB, we could access entity beans from remote clients directly. The best practice was to create a stateless session facade to access entity beans. In other words, you would only access entity beans from stateless session beans. The session facade brings the following benefits:

- Dependencies between client and server are minimized, increasing our ability to reuse entity beans in multiple applications, and masking changes in the entity bean from the client. It also better defines the programming logic, allowing session beans to control the logic.
- It acts as a transaction facade, enforcing the control of transaction activities to be executed on the server's behalf. If a client controls the entity bean without a session facade, this could increase the transaction time, thus decreasing concurrency for other clients accessing this entity.
- It minimizes data traffic when the entity bean is used. Without session facade, a higher data traffic from updates may result.

With EJB 3.0, no entity beans can be accessed remotely, making this pattern easy. It means that a stand alone client is forced to access entity beans through session beans. However, if you are using a local call to an entity bean from a servlet or JSP, the session facade pattern brings the benefits related above, improving the application design.

It is not a good practice to use only one session bean for all of your business logic. Modeling of use cases for each independent business case is necessary in order to plan for reuse and to program the session beans according to this approach. Also, when you use an entity bean, do not put domain data logic in your session bean, only delegate the CRUD operations to an entity bean. The exception is when a session bean controls a workflow of entities that are not related directly. In this case, a domain logic operation is required.

Note: You can find more details in 6.5, “Entity Beans 2.x” on page 347 or 6.6, “Java Persistence API: Entity Beans 3.0” on page 380.

Tuning stateless session beans

Stateless tuning involves the following considerations:

- Pool size tuning: The pool size should be tuned according to the EJB container vendor configuration. Other application components such as HTTP connections, servlet connections, and database connections, which are used in your application must also be analyzed. For more details about pool size tuning in WebSphere Application Server 6.1, see “Set EJB container pool size” on page 530.
- Resource caching with effectiveness: We must plan the components that fit best for caching. For example, a database connection is not a good component to cache because you decrease reusability to other beans when you reach the maximum session beans connected. In this case, the database connection number can be lower than the session bean pool size to better reuse this resource, and you can get a connection when it is required in a business method scope. Another resource might be a data source that is gotten from a JNDI operation; that is an expensive operation and can be cached. This approach is nice because a data source is a ConnectionFactory, not the connection.

Tuning stateful session beans

Stateful tuning involves the following considerations:

- Cache configuration: When you do not properly configure the cache for your stateful session beans, this hugely affects the performance of your application. Caching is reached when the number of clients exceed the number of maximum stateful instances allowed. During caching, the state of the bean is kept in storage, and the bean instance is available to another client request. Stateful session beans are held in the EJB cache until they are removed by the application or their session timeout value is reached.

  The EJB cache size and the cache cleanup intervals can be tuned to provide optimal performance. Determining the appropriate values for these parameters will require that you analyze your environment.

  For more information on EJB cache tuning, see:

  – EJB Container Tuning:

    `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.ejbfep.multiplatform.doc/info/ae/ae/rprf_ejbcontainer.html`

– Tuning the EJB cache using the trace service:

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.ejbfep.multiplatform.doc/info/ae/ae/tejb_tunecash.html`

– Chapter 7, “Environmental performance considerations” on page 509

- Serialization and deserialization control: The serialization and deserialization process consumes CPU cycles and I/O resources. This step occurs in the passivation and activation processes of stateful session beans respectively. It is best practice to build your session bean with minimum data to passivate. In other words, put in simple attributes to passivate. See “Some performance considerations for stateful session beans” on page 271 for more discussion about activation and passivation.

5.2.5 Message-driven beans

Messaging systems play an increasingly important role in enterprise computing. Java Message Service (JMS) is the Java API that enables loosely coupled Java clients to make asynchronous interactions with messaging systems such as IBM WebSphere MQ. JMS is a low-level API that enables applications to connect to messaging systems. To simplify coding for message consumption, a specific kind of enterprise bean, called the message-driven bean (MDB), was introduced based on the JMS API.

The message-driven bean

Message-driven beans (MDBs) act as JMS message listeners for asynchronous messages. When a message arrives at the destination or endpoint that is serviced by a message-driven bean, the container detects and invokes the message-driven bean.

JMS is an abstraction of the interfaces and classes that enable Java messaging clients to communicate with messaging systems. Just as JDBC provides an abstract implementation to access relational databases, JMS provides an abstract implementation to access messaging systems. With the help of JMS, messaging clients are portable across messaging products such as WebSphere MQ and SonicMQ. To better understand the JMS concepts, refer to 6.11, “Java Message Service” on page 456.

MDB motivations

Message-driven beans execute in the same container as session and entity beans, and thus benefit from the infrastructure that the container provides.

Based on the JMS API, MDBs support both publish/subscribe and point to point models. They act as a subscriber of a topic or a receiver of a queue, which is a message listener in the asynchronous communication model.

Multithreading message consumers

MDBs can process multiple messages concurrently. The container provides multithread services for message-driven beans, keeping a pool of MDB instances available for incoming messages. When a message arrives, an instance from the pool is assigned for processing.

Simplified coding

With MDBs, aspects of processing messages, such as looking up connection factories or destinations, creating connections, opening sessions, creating consumers, and so on, are done for you with the default configuration in EJB 3.0.

Characteristics of MDBs

Compared to session and entity beans, MDBs have the following characteristics:

- Like stateless session beans, MDBs do not maintain conversational state between requests. They might have instance variables throughout the life cycle, but due to the pooling of bean instances by the EJB container, the bean instances that consume the messages can be different between requests. That is why the conversational state might not be stored properly.
- Like a stateless session bean, the EJB container maintains many bean instances of the same type in a pool. This enables concurrent message consumption and processing when several messages are delivered at the same time, which means that MDBs can deliver better performance and scalability.
- Unlike session and entity beans, MDBs do not have remote or home interfaces. An MDB is a listener and it is not a remote process call component.
- Unlike session and entity beans, MDBs do not expose any business methods that can be invoked by clients, such as a servlet, EJB, or Java application.
- Like session and entity beans, MDBs must use the JMS API to send messages when they act as a message producer. However, sending messages from MDBs is not recommended — MDBs should delegate this task to the business logic layer.

Programming rules for writing an MDB

MDBs, like other EJB types, are POJO classes and follow simple rules of construction and annotations. We explain this in the following sections.

Must implement javax.jms.MessageListener

When you build an MDB, it is mandatory that it implements javax.jms.MessageListener using the implements keyword, with annotations, or is denoted in the deployment descriptor as an MDB.

Example 5-17 shows the implementation using the implements keyword.

Example 5-17 MDB class declaration using implements keyword

```
public class CreateUsersCardProcessor implements
javax.jms.MessageListener
```

Also see the annotation shown in Example 5-18.

Example 5-18 MDB class declaration using annotations

```
@MessageDriven(
      name="CreateUsersCardJMSProcessor",
      messageListenerInterface="javax.jms.MessageListener")
public class CreateUsersCardProcessor{
```

MDB class should be concrete

This means that a message-driven bean class cannot be abstract or final. It must be a POJO class and also cannot be a subclass of another MDB.

MDB must be public with no arg constructor

The MDB class must have a no argument constructor. If you do not declare another constructor, the compiler constructs a no argument constructor for you. If you declared another constructor, you must create a no argument constructor.

Do not define a finalize method; use PreDestroy callback instead

If you require a cleanup process, you should define a method designated with @PreDestroy annotation.

Define methods specified in message listener interface

You must implement methods defined in the message listener interface, but they must be public and not static or final.

Do not throw javax.rmi.RemoteException or any runtime exception

Do not use javax.rmi.RemoteException or any runtime exception. However, the MDB bean instance is finalized if a RuntimeException is thrown. Remember that RuntimeExceptions do not have to be declared and are thrown normally by Java functionality.

Life cycle of MDBs

The EJB container is responsible for controlling the life cycle of an MDB, as explained in Figure 5-7.

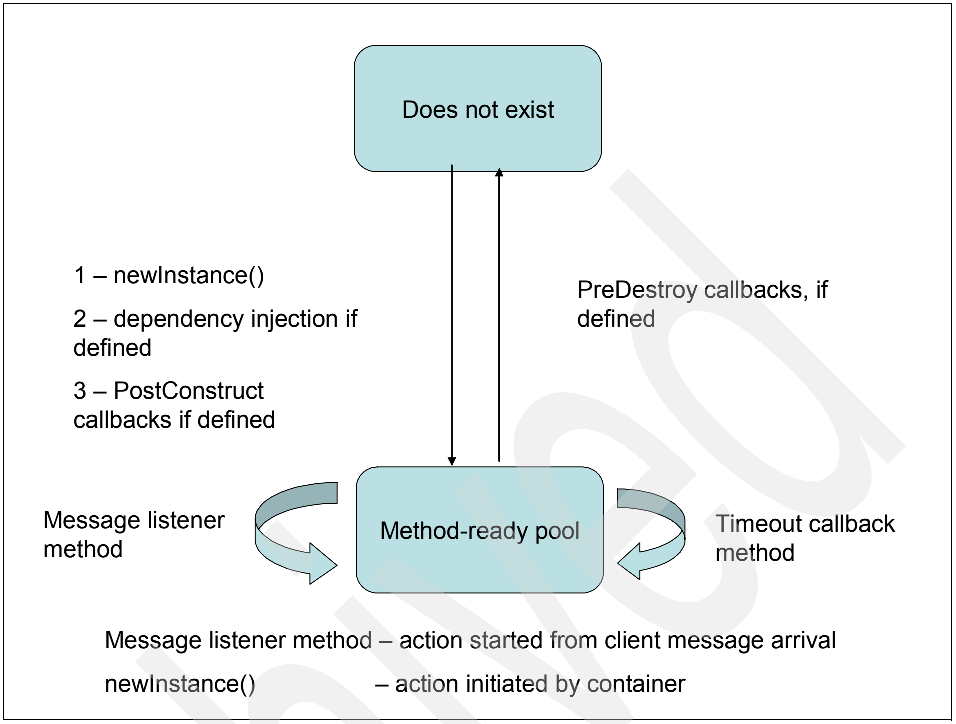


Figure 5-7 Message driven bean life cycle

The container can perform the instance creation at any time with no direct relationship to a business invocation from client.

Here we expand upon the numbered steps in the previous diagram:

1. First, the newInstance method is invoked on the MDB class and a new instance is created.
2. Next, the EJB container injects the MessageDrivenContext and any other Dependency Injections specified by annotations or by the deployment descriptor.
3. The container calls the PostConstruct callback method, if defined. The message-driven bean instance is now ready to accept messages that arrive on its associated destination or endpoint.
4. When a message arrives, the container pulls an idle bean out of the pool to process the message. If additional messages arrive and there are no MDB instances available to process the message, the container will increase the pool size (assuming the maximum pool size has not been reached).

5. The bean's onMessage method is executed. When execution is complete, the EJB container puts the idle bean back into the method-ready pool to wait for another incoming message. Another processing activity that can occur at this state is a call from the container to the timeout callback.
6. When the container no longer requires the instance (which usually happens when the container wants to reduce the number of instances in the method-ready pool), the container invokes the PreDestroy life cycle callback methods for it, if any are defined. This ends the life of the message-driven bean instance.

Annotations used in message-driven beans

There are few annotations defined specifically for message-driven beans; only @MessageDriven and @ActivationConfigProperty. In the following topics, we explain the annotations used in message-driven beans.

The @MessageDriven annotation

The purpose of the @MessageDriven annotation is set up the configuration of message-driven beans. The coding in Example 5-19 shows the usage of this annotation.

Example 5-19 @MessageDriven annotation usage example

```
@MessageDriven(
      name="CreateUsersCardJMSProcessor",
      messageListenerInterface="javax.jms.MessageListener"
      activationConfig= {
          @ActivationConfigProperty(
               propertyName="destinationType",
               propertyValue="javax.jms.Queue"),
          @ActivationConfigProperty(
               propertyName="destinationName"
               propertyValue="jms/CreateUsersRequestQueue")
      }
)
```

In this example, notice that all three parameter types are all optional.

- The name parameter provides a label for the MDB. If not specified, the class name is used.
- The messageListenerInterface is discussed in "Must implement javax.jms.MessageListener" on page 284. If omitted, the interface must be denoted in the deployment descriptor.
- The activationConfig parameter is used to configure listener specific properties.

If all parameters are omitted, the annotation for the message-driven bean will look like Example 5-20.

Example 5-20 @MessageDriven annotation with default values

```
@MessageDriven
public class CreateUsersCardProcessor
```

Using @ActivationConfigProperty

The activationConfig parameter describes the message listener configuration. It has @ActivationConfigProperty annotations that provide name/value pairs for each configuration property defined.

An MDB is by definition a JMS consumer. The EJB container takes care of subscribing the bean to the desired topic or connecting it to the desired queue based on its deployment descriptor or annotations that are described in this section.

The following information is required for the container to deploy an MDB. It is the application assembler's responsibility to make sure that the settings are correct according to the business requirements. The @ActivationConfigProperty is used to define this information.

- Destination Type:

  A queue or a topic. An MDB either connects to a queue or subscribes to a topic. The container also has to know how to look up the instance of the destination through the use of a JNDI name. See Example 5-21.

  Example 5-21 @ActivationConfigProperty for destination

  ```
  @ActivationConfigProperty(
              propertyName="destinationType",
                 propertyValue="javax.jms.Queue")
  ```

- Destination Name:

  Defines the JNDI name of the destination the MDB listens to for messages. See Example 5-22.

  Example 5-22 @ActivationConfigProperty for destination name

  ```
  @ActivationConfigProperty(
     propertyName="destinationName"
     propertyValue="jms/CreateUsersRequestQueue")
  ```

- ConnectionFactory:

  For MDBs that connect to a queue, the container has to know the JNDI name of the QueueConnectionFactory instance that creates connections to the queue. For MDBs that subscribe to a topic, the container has to know the JNDI name for the TopicConnectionFactory instance that creates connections to the topic. See Example 5-23.

  Example 5-23 @ActivationConfigProperty for ConnectionFactory

```
@ActivationConfigProperty(
propertyName="connectionFactoryJndiName",
   propertyValue="jms/QueueConnectionFactory")
```

- Durability:

  An MDB that subscribes to a topic can choose to use durable or non-durable subscriptions with the topic. Durability is not applicable to the message receivers of a queue.

  In durable subscription, it is the responsibility of the server to store messages that a subscriber misses while disconnected from the JMS server. The messaging server sends all of the unexpired messages it stores to the subscriber when it reconnects.

  If an MDB subscribes to a topic, it has two options for durability: Durable or Nondurable. If Durable is selected, then when it reconnects, the MDB gets all of the messages that it missed during the disconnected period. See Example 5-24 to specify durability.

  Example 5-24 @ActivationConfigProperty for durability

```
@ActivationConfigProperty(propertyName="SubscriptionDurability",
propertyValue="Durable")
```

- Acknowledge mode:

  A message is not removed from a queue until the consumer agrees with this request. This action is called acknowledge mode. By default, a JMS session is assumed to be AUTO_ACKNOWLEDGE. This means that the session acknowledges receipt automatically after a message has been received or processed successfully. Another configuration supported is DUPS_OK_ACKOWLEDGE, which is similar to AUTO_ACKNOWLEDGE, but the application can handle delivery of duplicate messages. In terms of @ActivationConfigProperty, this is shown in Example 5-25.

  Example 5-25 @ActivationConfigProperty for acknowledge mode

```
@ActivationConfigProperty(
   propertyName="acknowledgeMode"
```

```
propertyValue="DUPS_OK_ACKNOWLEDGE")
```

- Message selector:

  An MDB can select the messages for consumption based on their headers and properties. A message selector is not a part of the Message object, but it is related to the message headers and properties. A message selector lets a JMS consumer declare which messages it wants to receive by using message headers and properties as criteria in conditional expressions.

  An MDB works as a message consumer; therefore it can use the message selector definition. Message selectors are based on a subset of the SQL-92 conditional expression syntax, which is used in the WHERE clauses of SQL statements. For example, consumers with a simple message selector like InventoryID = 12345 receive all messages that have a property named InventoryID with the value 12345. In terms of @ActivationConfigProperty annotation, see Example 5-26.

  Example 5-26 @AtivationConfigProperty for message selector

```
@ActivationConfigProperty(propertyName="messageSelector",
propertyValue="RECIPIENT = 'MDB'")
```

Other annotations used in MDBs

Table 5-3 shows an overview of other annotations used in MDBs. It does not include transaction control, which is explained in 5.2.6, "Best practices for advanced concepts of EJBs" on page 291. MDBs behave similar to stateless session beans in terms of life cycle and the callback annotations are the same (@PostConstruct and @PreDestroy). The @WebService annotation is exclusive for stateless beans. @Local and @Remote are not related to MDBs because they do not have synchronous client access, and of course, the @Stateless annotation is not used because it describes a stateless bean to the container.

Table 5-3 Other annotations used in MDBs

Annotation	Purpose	Example
@PostConstruct	Used as a container callback method after the bean is constructed.	`@PostConstruct public void initializeResouces()`
@PreDestroy	Used as a container callback method before the bean is destroyed.	`@PreDestroy public void deliverResources()`

Annotation	Purpose	Example

@Resource	Used to inject resources such as data sources. The parameter tells the container to find a resource in a described JNDI location and assign it to a parameter variable. This step is done after bean creation.	`@Resource(name="jdbc/UserCardDS")`
		`public void setDataSource(DataSource datasource){`
		`this.datasource = datasource;`
		`}`

Message-driven beans: Best practices

In the following sections we describe some message-driven best practices.

Modeling your message-driven in right way

When you develop your message-driven beans, avoid putting business logic in the onMessage method to ensure separation of business logic from message specific concerns. If, for example, you develop another set of business logic, POJOs could be a better choice, increasing reusabilities to other parts of code that do not use message concerns.

Tuning message-driven beans

We can consider an MDB as a stateless session bean with an onMessage() method that is triggered by the container when a message arrives. The pool settings are the same as the stateless session bean. Parameters that should be tuned include initial pool size, maximum pool size, resize factor, and pool idle time timeout settings.

Tuning Java Message Service

If you can, avoid using DUPS_OK_ACKNOWLEDGE, because when using this approach, you can receive duplicated messages increasing the network bandwidth. For more information on JMS tuning, see 6.13.7, “JMS” on page 508.

5.2.6 Best practices for advanced concepts of EJBs

In the following sections, we discuss EJB best practices for advanced concepts such as transactions, interceptors, and timers.

EJB transactions

Before going into depth about EJB transactions, we discuss some transaction concepts.

Transactions: An overview

So what is a transaction? Why are they so important? Consider this very simplistic case of a banking transaction: You transfer $100 from your checking account to your savings account. On further investigation, this might involve two smaller operations:

- Your bank subtracts $100 from your checking account.
- Your bank adds $100 to your savings account.

If the bank reduced your checking balance by $100 but failed to increase your savings balance by $100, you might be a little upset. We prefer to think of the two operations as one operation. So if $100 is never added to your savings account, $100 should never be subtracted from your checking account!

Similarly, there are business cases in applications that take on this all-or-nothing approach. Some large operations consist of one or more smaller steps. For the operation to complete, all steps within the operations must complete, or none at all. This behavior is known as atomicity.

Atomicity is one of four characteristics (or properties) that transactions must guarantee. The other three properties are:

- *Consistency*
- *Isolation*
- *Durability*

Together these four properties are called the *ACID properties*.

ACID properties

Transactions exhibit these well-known ACID properties:

- Transactions are atomic. All operations are considered a single unit of work. This is an all-or-nothing approach, as discussed previously.
- Transactions are consistent. After a transaction executes, it must leave the system in a consistent (or legal) state. What defines a legal state is up to the system. To follow the earlier example, after any withdrawals, your bank dictates that you leave your checking account with a positive balance.
- Transactions are isolated. Each transaction behaves in isolation of other transactions executing on the same resource. This is achieved through lock synchronization of data.
- Transactions are durable. Updates to resources must survive system failures, such as hardware or network failures. In distributed systems, recovery processes are required when networks fail or databases crash.

Transaction models

There are two popular transaction models: flat transactions and nested transactions. EJBs support the flat transaction model.

Flat transactions are a series of operations performed as a single unit of work. There can be only two results from this unit of work: either success or failure. If all steps in a transaction complete successfully, the transaction is committed and all persistent data changes performed by the operation become permanent. If any steps within the transaction fail, the transaction rolls back and reverses any data effects caused by the steps in the transaction.

Nested transactions allow for transactions (or units of work) to be nested in other transactions. Transactions nested in other transactions allow rollbacks to occur without affecting their parent transactions. Failed nested transactions can continue to retry. If failure continues, the parent transactions might be rolled back.

EJB transactions support

EJBs are a framework for component development. You develop EJBs that run within an EJB container. Among other things, the EJB container provides the benefit of transactions.

Also, the EJB architecture supports distributed transactions. This transaction feature occurs when multiple participants within a single transaction are physically distributed across a network. Distributed transactions allow for different types of resources to participate in the transaction. Here are some examples of distributed transactions:

- A single session bean begins a transaction and updates database A. It invokes a second session bean running on the same application server to update database B. The first session bean commits the transaction. Both database updates occur in the same transaction.
- A single session bean begins a transaction and updates database A. It invokes a second session bean running on a different application server to update database B. The transaction managers for each application server ensure that both databases are updated in the same transaction.
- A single session bean begins a transaction and updates database A, followed by a Java Message Service (JMS) operation. Both units of work are part of the same transaction. If the JMS operation were to fail, the transaction would not update the database.
- A Java client to explicitly demarcate transaction boundaries before updating multiple databases on multiple EJB servers.

Several transaction managers must work together to perform a distributed transaction. Usually a single transaction manager (called the transaction coordinator or distributed transaction manager) is appointed to coordinate the other transaction managers.

Transaction managers, in turn, coordinate with resource managers to perform the necessary commits or rollbacks on their resources (perhaps a database or a messaging server). Most databases have their transaction managers and resource managers tightly coupled together.

Transaction boundaries

When implementing EJB transactions, you are demarcating transaction boundaries: who begins the transaction, who commits or aborts the transaction, and when you should use the transactions. It is up to the EJB container and server provider to provide transaction management and a low-level transaction communication protocol.

There are two demarcation options:

- The declarative option, in which you can delegate transaction implementation to the EJB container. (This option is the focus of the remainder of this chapter.)
- The programmatic option, in which the enterprise beans provide the commit or abort information themselves in their code.

With declarative transaction demarcation, the EJB container applies transaction boundaries on an enterprise bean's methods based on instructions declared by the application developer in the EJB deployment descriptor or annotation for EJB 3.0 applications. This is called a container-managed transaction (CMT). The EJB container is responsible for controlling transaction boundaries.

When implementing programmatic demarcation of transactions, the application developer is responsible for programming transaction logic and boundaries into enterprise bean code. This is called a bean-managed transaction (BMT).

With bean-managed transactions, you programmatically control your transaction boundaries and decide when transactions begin, commit, and roll back. Within bean-managed transactions, you can choose between implementing Java Transaction API (JTA) or Java Database Connectivity (JDBC) transactions. JTA transactions use the javax.transaction.UserTransaction interface to control transactions, while JDBC transactions control the behavior of transactions by performing operations directly through the java.sql.Connection interface.

If you are using session beans or MDBs, you can implement bean-managed or container-managed transactions. Entity beans, however, can only use container-managed transactions.

Table 5-4 summarizes these options by transaction type for each enterprise bean implementation.

Table 5-4 Transaction type for each enterprise bean

Transaction type	Session bean	Entity bean	Message-driven bean
Bean-managed	Yes	No	Yes
Container-managed	Yes	Yes	Yes

Transaction attributes

You specify transaction attributes for the entire enterprise bean, for individual methods on the bean, or for both. The following choices are available:

- Required: The bean must always run within a transaction. If the client has started a transaction, the bean joins the transaction. If the client has not started one, the EJB container starts a new transaction. Use this attribute when you want your bean to always run within a transaction.
- RequiresNew: The bean always starts a new transaction. If the client has started a transaction, this transaction is suspended until the new transaction commits or aborts. After the new transaction completes, the existing transaction resumes. Use this attribute when you want your bean to run as a single unit of work and exhibit all of the ACID properties.
- Supports: If the client has started a transaction, the bean joins the transaction. However, the EJB container does not start a new transaction if one does not exist. Use this attribute for non-mission-critical operations on your enterprise bean.
- Mandatory: The client must start a transaction when the bean is invoked. A new transaction can't be created. If there is no transaction already started when the bean is invoked, an exception is thrown. Use this attribute when a bean is part of a larger system. Usually a third party might be responsible for starting the transaction. This is a safe option to use because it guarantees that the bean becomes part of a transaction.
- NotSupported: Bean cannot be involved in a transaction. If the client has started a transaction, the existing transaction is suspended until the bean's method has completed. After completion, the existing transaction is resumed. If the client has not started the transaction, a new transaction is not created. Use this attribute when you do not require your bean to exhibit any of the ACID properties, such as for non-system-critical operations like reporting.
- Never: If the client has started a transaction, the bean throws an exception. You might never want your bean to participate in a transaction. Use this attribute in those cases.

In terms of bean support, session beans and entity beans support all transaction attributes. Message-driven beans support only the NotSupported and Required attributes. This restriction is related to the fact that no client invokes an MDB directly.

Best practices for using EJB transactions

In the following sections we offer best practices recommendations for EJB transactions.

Use distributed transactions only when necessary

Be aware that distributed transactions are slower than using local transactions. All transaction participants require more system resources. Network chatter between the transaction coordinator and all the transaction participants affects system response time. Distributed transactions just take longer due to the number of transaction managers and resource managers involved.

Use container-managed transactions and the required attribute

If you are unsure of which transaction type to use for your bean, we recommend using container-managed transactions with the required attribute.

For developers, using container-managed transactions is simpler and requires less work. No transactional logic is required in your bean method. You demarcate transaction boundaries at the method level on the enterprise bean. Your bean method must either run within the context of a transaction or not.

Prefer to use container-managed transactions

Learn how two-phase commit transactions work in Java EE and rely on them rather than developing your own transaction management. The container is almost always better at transaction optimization.

Using container-managed transactions provides two key advantages that are nearly impossible to obtain without container support: composable units of work, and robust transactional behavior.

If your application code explicitly begins and ends transactions (perhaps using javax.jts.UserTransaction, or even native resource transactions), future requirements to compose modules, perhaps as part of a refactoring, often require changing the transaction code. For example, if module A begins a database transaction, updates the database, and then commits the transaction, and module B does the same, consider what happens when you try to use both from module C. Now, module C, which is performing what is a single logical action, is actually causing two independent transactions to occur.

If module B were to fail during an operation, module A's work is still committed. This is not the desired behavior. If, instead, module A and module B both used container-managed transactions, module C can also start a container-managed transaction (typically implicitly via the deployment descriptor) and the work in modules A and B are implicitly part of the same unit of work without any requirement for complex rework.

If your application must access multiple resources as part of the same operation, you require two-phase commit transactions. For example, if a message is removed from a JMS queue and then a record is updated in a database based on that message, it is important that either both operations occur — or that neither occurs. If the message was removed from the queue and then the system failed without updating the database, this system is inconsistent. Serious customer and business implications result from inconsistent states.

We occasionally see client applications trying to implement their own solutions. Perhaps the application code might try to “undo” the queue operation if the database update fails. We do not recommend this. The implementation is much more complex than you might think (imagine what happens if the application crashes in the middle of this). Instead, use two-phase commit transactions. If you use container-managed transactions and two-phase commit capable resources (like JMS and most databases) in a single container-managed transaction, WebSphere Application Server takes care of the dirty work. It makes sure that the transaction is entirely done or entirely not done, including failure cases such as a system crash, or database crash.

The implementation maintains transactional state in transaction logs. We cannot emphasize enough the importance of relying on container-managed transactions transactions if the application accesses multiple resources. If the resources you are using cannot provide for two-phase commit, then of course you have no choice but to use a more complex approach — but you should do everything possible within your power to avoid this situation.

Use bean-managed transactions only to meet specific requirements

In container-managed transactions, the demarcation boundaries are at the bean-method level. In some cases this might not be granular enough. If you require stricter control of your transaction boundaries or expect to have long-running processes within your enterprise beans, use bean-managed transactions. It is best to have your transactions to run for as short a time as possible.

By using bean-managed transactions, you can limit the duration of the transactions to be short-lived. You can isolate the database operations within the transaction and allow the longer-running processes to run outside the scope of the transaction. This ensures that you do not block any other transactions from accessing the same data.

@TransactionManagement and @TransactionAttribute annotation

The @TransactionManagement annotation specifies if the particular bean uses container manager transactions or bean manager transactions.

For example if you use the @TransactionManagement attribute TransactionManagementType.CONTAINER in the bean declaration, it means that a container-managed transaction is used. On the other hand, if you define the @TransactionManagement attribute TransactionManagementType.BEAN in the bean declaration, it means that a bean-managed transaction is used.

Another important annotation is @transactionAttribute, which tells the container how to manage transactions and can be applied either to individual bean methods or the entire bean. For more information about transaction types, see in "EJB transactions" on page 291.

See Example 5-27, which is a complete coding example using the described annotations. In this example, a container-managed transaction with the required attribute is being used.

Example 5-27 @transactionManagement attribute usage

```
@stateless
@TransactionManagment(TransactionManagmentType.CONTAINER)
public class CardProcessorBean implements CardProcessor {
  @Resource
  SessionContext context;

  @EJB
  Account account;

  @TransactionAttribute(TransactionAttribute.REQUIRED)
  public void payCard(String accountID,String card,double amount){
      try{
          if (account.balance(accountID) < amount){
              //Process InsufficientFundsException
           }
           else{
              account.debit(amount);
           }
        }
        catch(InsufficientFundsException ife){
           context.setRollbackOnly();
        }
```

```
        catch(AccountException ae){
           context.setRollbackOnly();
        }
   }
   //Another code parts
}
```

Rolling back a transaction in container-managed transactions

If an appropriate business condition occurs, a container-managed transaction method can ask the container to roll back the transaction. However, the rollback is not done until the end of the transaction; only a flag to roll back is set for the container. After the end of the transaction, the rollback is done. We took a small code snippet from Example 5-27 on page 298 to explain this situation. See Example 5-28.

Example 5-28 Rollback a transaction

```
catch(AccountException ae){
   context.setRollbackOnly();
}
```

Interceptors

A nice addition to the EJB specification is the use of interceptors. One thing missing from EJB components has been the ability to perform Aspect Oriented Development (AOP) for things such as pre/post-processing and cross cutting concerns, much as servlet filters do for servlets. You can now develop an interceptor class and apply it to a bean.

The code in Example 5-29 is an example of an interceptor that audits invocations of the CardProcessorBean class.

Example 5-29 Interceptor class for auditing

```
public class CardProcessorRequestAudit {
   @AroundInvoke
   public Object auditCardOperation(InvocationContext inv) throws
   Exception {
      try {
         Object result = inv.proceed();
         Auditor.audit(inv.getMethod().getName(),
inv.getParameters[0]);
         return result;
      } catch (Exception ex) {
         Auditor.auditFailure(ex);
         throw ex;
```

```
        }
    }
}
```

The interceptor in the previous example intercepts the call to the target EJB method, then calls the proceed() method on the InvocationContext. This enables the call to flow through to the actual EJB method that was invoked. After the target EJB method returns, it uses the metadata in the InvocationTarget to get the method name and parameters of the target EJB component that was called. The interceptor can then be applied to the bean class as shown in Example 5-30.

Example 5-30 Interceptor annotation in bean class

```
@stateless
@Interceptors({CardProcessorRequestAudit})
public class CardProcessorBean implements CardProcessor {

  //CardProcessor implemented methods
  public double balance(String card) throws Exception {
   //Put your Business Logic Here
   return 0;
  }

  //Other business methods

}
```

Additionally, you can develop interceptor methods that are implemented inside the bean class and also specify multiple interceptors, in which case the order in which they are called is specified by the order in which they are defined in the bean class. Interceptors can also be applied using XML away from the bean, which is preferred in AOP, since you want to transparently apply cross cutting concerns to beans.

Timers

Since the EJB2.1 specification, a new support for scheduling using the EJB container was introduced. This is the EJB Timer Service, which you can code using the Timer Service API.

> **Note:** In this section we focus on developing a time service using the EJB 3.0 specification. To develop a time service for the EJB 2.1 specification, see the following URL:
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/tejb_timerserviceejb.html

Timer services are used for building J2EE applications that depend on time based services. Time based services are mostly used in scheduling workflow type applications that define a configurable sequence of activities or tasks that take place at a particular point of time. Before EJB 2.1, one had to manually code for building and deploying time-based workflow systems. But, with the invent of EJB 3.0, with annotations and Dependency Injections, this has become easier.

Timers: Services provided by the container

Time services provided by the container. Developers can use this service to register enterprise beans to receive time-based notifications. We can use timers only in stateless session beans and MDBs because the bean types are stateless and asynchronous. However, timers can survive a container crash or restart. Timers are transactional, it means that when a failure happens in a timeout method, a rollback occurs. See Figure 5-8 for an explanation of EJB functionality.

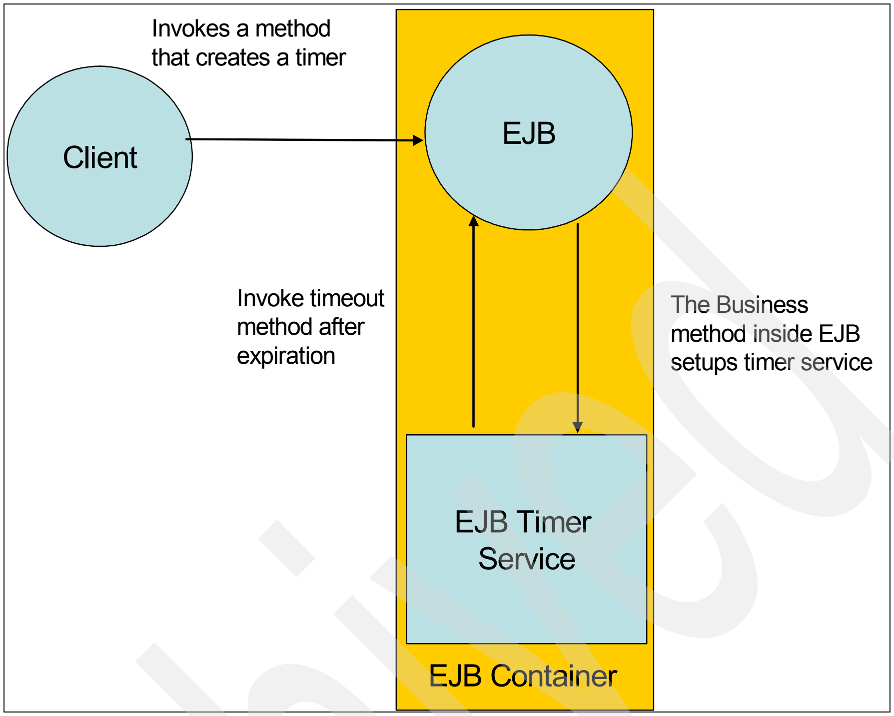


Figure 5-8 EJB timer service functionality

As the first step, the client invokes a business method that creates a time service and registers a callback in the EJB timer service. The EJB container invokes the timeout method in the bean instance when timer expires.

Timers: Object types

The EJB container supports two types of Timer objects:

- Single action timer:

  A single action timer (or a single interval timer) is one that expires only once.

  EJB supports two ways for building a single interval timer:

  – The first way is to create the timer object in such a way that it expires at a particular period of time, specified as a Date.

  – The other way is make the timer to expire after certain period of time (such as after 10 hours or 1 day) which is usually specified in milliseconds. After the timer expires, the enterprise bean receives a notification and the container calls the ejbTimeout() method or another method that has @Timeout annotation.

- Interval timer:

  The interval timer (or multiple action timer) has multiple expirations at regular intervals of time.

  EJB supports two ways to build these timers:

  – The first approach is to create a timer object to have an initial expiration at some period of time specified and to have the subsequent expirations occurring at a specified interval.

  – The second approach is to construct a timer whose initial expiration happens after an elapsed duration of time (in milliseconds) and to have the subsequent expirations happen after a specific interval. For every expiration of the Timer object, the container continues to call the ejbTimeout() method (or the method that is annotated with @Timeout annotation) until the bean explicitly calls the cancel() method of a javax.ejb.Timer instance.

Using the timer service

In Example 5-31, we provide a code sample showing how to use a time service.

Example 5-31 Using a time service example

```
@stateless
public class CardProcessorBean implements CardProcessor {
  @Resource TimerService timerService;

  public boolean check(String card){
        //Code for check card
        timerService.createTimer(15*60*100, 15*60*1000, card);
  }
  @Timeout
  public void monitorCardChecks(Timer timer){
        String card = (String)timer.getInfo();
        //  code to monitor card activity
  }
}
```

Looking at the foregoing code, the line with the @Resource annotation tells the container to inject a time service instance. The check() method registers the timer and specifies the time interval. At the expiration of each interval, the monitorCardCheck() method is called by EJB time service. This is denoted by the @TimeOut annotation.

The cancel method in the Timer object is used to cancel the service. Upon invoking this method, the enterprise bean registered for notifications no longer receives callbacks. See Example 5-32.

Example 5-32 Timer interface using example

```
@Timeout
public void monitorCardChecks(Timer timer){
     String card = (String)timer.getInfo();
     //  code to monitor card activity
     if (cancel) // for some reason cancel the timer
       timer.cancel();
}
```

When to use EJB timers

It is a good practice to use EJB timers when you require a scheduling service as part of your a business logic and EJB timers are provided as a built-in container service. The timers should be persistent, capable of surviving an application server crash.

5.3 General best practices considerations for business logic

Following are best practice recommendations when planning and creating business logic components.

5.3.1 Choose a Web application framework that can work with or without using EJBs

Use common, proven frameworks such as Apache Struts, JavaServer Faces, and Eclipse RCP. Use proven patterns and do not re-invent the wheel.

Developing a framework for user-interface development significantly improved developer productivity over building UI applications directly to the base servlet and JSP specifications. As a result, many companies developed their own UI frameworks that simplified the task of interface development.

As open-source frameworks like Apache Struts began to develop, it seemed that the switchover to these new frameworks would be automatic and quick. It seemed that the benefits of having an open-source community supporting the framework would be readily apparent to developers, and that they would gain universal acceptance very rapidly — not only for new development, but in retrofitted applications as well.

What has proven surprising is that this has turned out not to be the case. There are still many companies maintaining or even developing new user-interface frameworks that are functionally equivalent to Struts or JSF. There are many reasons why this could be true: Organizational inertia, “not invented here” syndrome, lack of perceived benefit in changing working code, or possibly even a slight sense of hubris in thinking that one could do things “better” than the open-source developers did in a particular framework.

However, the time is long past when any of these reasons are worth using as an excuse not to adopt a standard framework. Struts and JSF are not only well accepted in the Java community, but fully supported within the WebSphere runtimes and Rational tool suites as well. Likewise, in the rich client arena, the Eclipse Rich Client Platform (RCP) has also gained wide acceptance for building standalone rich clients. While not a part of the Java EE standard, these frameworks are now a part of the Java EE community, and should be accepted as such.

Using EJBs, Apache Struts and JavaServer Faces, which are part of J2EE since V1.4, are the best choices to work with, because there is a good integration with EJB technology and there is extensible tool support in the market. Rational Application Developer has a complete tool support with wizards to make the development easy.

5.3.2 Apply automated unit tests in the business logic layer

We recommend strongly that you unit test not only your business logic but the whole program. You can use proven tools, such as JUNIT, for testing. This tool is well integrated with development tools such as Eclipse and Rational Application Developer with wizards. With JUNIT you can easily create unit tests and you can mix several unit tests in a test suite enabling regression tests. For more information about JUNIT, see the URL:

`http://www.junit.org/`

For EJB tests for versions prior to EJB 3.0, there is an Apache framework called Cactus. For more information, see the following URL:

`http://jakarta.apache.org/cactus/writing/howto_ejb.html`

With EJB 3.0, there are huge improvements in tests because EJBs are considered POJOs. Even for a remote client test we can use the J2EE client container making it easy to run standalone unit tests.

5.3.3 Prefer to develop core business logic in POJOs

Use plain old Java objects (POJOs) or simple Java beans to develop your business logic. By following this recommendation, any technology dependency is only related to Java and not to other frameworks such as JavaServer Faces or Struts. Using this approach, it is easy to change or reuse your business logic for use in other systems. With EJB 3.0, this approach is easier because of the annotations approach taken to set up an EJB and rules that do not affect the behavior of a POJO.

5.3.4 Build a better exception-handling framework

The first thing to think about when designing a solid exception-handling scheme is the abstraction of what we call low-level or system-level exceptions. These are generally core Java exceptions that report errors in network traffic, problems with JNDI or RMI, or other technical problems in an application. RemoteException, EJBException (EJB 2.1 and before), and NamingException are common examples of low-level exceptions in enterprise Java programming.

These exceptions are fairly meaningless, and can be especially confusing when received by a client in the Web tier. Throwing this type of exception to the customer can be a problem in terms of error control and application design. Design your exception-handling scheme so customers see a friendly message, and meaningful information about the error is logged.

5.3.5 Central coding and DAO

Avoid putting your business logic in the GUI since it can make it difficult to maintain and find errors, as well as to isolate the access to the data access layer Data Access Object (DAO). Refer to “Data Access Object” on page 43 for more information.

5.3.6 Embrace the qualities of service provided by the application server environment

Design applications to be clusterable using WebSphere Application Server Network Deployment.

We have already mentioned the importance of leveraging WebSphere Application Server security and transactional support. One more important area that we see ignored far too often is clustering. Applications must be designed and delivered to run in a clustered environment. Most realistic environments require clustering for scalability and reliability. Applications that do not cluster lead quickly to disaster.

Closely related to clustering is supporting WebSphere Application Server Network Deployment. If you are building an application that you intend to sell to others, make sure your application runs on WebSphere Application Server Network Deployment and not just the single server versions.

5.3.7 Plan for using Java EE security from day one

Turn on WebSphere security. Lock down all your EJBs and URLs to at least all authenticated users. Do not even ask — just do it.

It is a continual source of astonishment to us how few customers we work with originally plan to turn on WebSphere Application Server's Java EE security. In our estimate, only around 50% of the customers we see initially plan to use this feature. We have even worked with several major financial institutions (banks, brokerages, and so on) that did not plan on turning security on; luckily this situation was usually addressed in review prior to deployment.

Not leveraging Java EE security is a dangerous game. Assuming your application requires security (almost all do), you are betting that your developers can better build a security infrastructure than the one you bought from the Java EE vendor. That's not a good bet. Securing a distributed application is extraordinarily difficult.

For example, suppose that you have to control access to EJBs using a network-safe encrypted token. In our experience, most home-grown security infrastructures are not secure, with significant weaknesses that leave production systems extremely vulnerable.

Reasons cited for not using Java EE security include: fear of performance degradation, belief that other security products like IBM Tivoli Access Manager and Netegrity SiteMinder handle this already, or ignorance of the features and capabilities of WebSphere Application Server security. Do not fall into these traps. In particular, while products like Tivoli Access Manager provide excellent security features, they alone cannot secure an entire Java EE application. They must work hand in hand with the Java EE application server to secure all aspects of the system.

Another common reason given for not using Java EE security is that the role-based model does not provide sufficiently granular access control to meet complex business rules. Though this is often true, this is no reason to avoid Java EE security. Instead, leverage the Java EE authentication model and Java EE roles in conjunction with your specific extended rules. If a complex business rule is required to make a security decision, write the code to do it, basing the decision upon the readily available and trustable Java EE authentication information (the user's ID and roles).

5.4 References

Here is a list of references that you might find useful when reading this chapter:

- SOA terminology overview, Part 1: Service, architecture, governance, and business terms:

  http://www.ibm.com/developerworks/webservices/library/ws-soa-term1/
- The EJB Advocate: Is it ever best to use EJB components without facades in service oriented architectures?

  http://www.ibm.com/developerworks/websphere/techjournal/0506_ejba/0506_ejba.html
- Get to know Java EE 5:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0707_barcia/0707_barcia.html
- EJB 3.0 specification (Java Community Process):

  http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html
- Integrating Message-Driven Beans into Enterprise Applications with WebSphere Studio: Part 1 -- JMS and Message-Driven Beans:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0304_yu/yu1.html
- IBM WebSphere Developer Technical Journal: The top Java EE best practices:

  http://www.ibm.com/developerworks/websphere/techjournal/0701_botzum/0701_botzum.html
- The Redbooks publication, Rational Application Developer V7, SG24-7501:

  http://www.redbooks.ibm.com/redpieces/abstracts/sg247501.html?Open
- Weighing the options for Apache Geronimo EJB transactions, Part 1: Container-managed transactions:

  http://www.ibm.com/developerworks/opensource/library/os-ag-ejbtrans1

6

Integration layer

This chapter focuses on some technologies involved with the enterprise integration layer. We offer tips that can be used for legacy applications when you do not have the option to make major changes. In addition, we cover some new technology directions and best practices. For some new technologies, we include an overview along with best practices.

The chapter is organized into the following major sections:

6.1 The integration layer: EIS integration

In the J2EE world, the integration layer, also referred to as the Enterprise Information System (EIS tier), handles EIS software and includes enterprise infrastructure systems. Today most companies have some sort of a back-end EIS that processes business transactions and maintains business data in a database. Examples of a corporate EIS are:

- Enterprise Resource Planning (ERP) systems (for example, SAP®)
- A Customer Information Control System (CICS®) and Information Management System (IMS™) system running business transactions
- A corporate database running SQL stored procedures
- A corporate database storing company data
- Another EIS process that has be integrated, for example, using a Web service
- A combination of some or all of these scenarios

6.1.1 Levels of EIS integration

Architects and designers have designed and successfully implemented different back-end integration solutions for many years. Chances are good that there are still production systems running in many companies today that integrate with a back-end system that is running CICS / COBOL, through Systems Network Architecture (SNA), using Advanced Program to Program Communication (APPC). In these environments, it is likely that the company has created a framework for their back-end integration requirements. A common approach used by such frameworks is to create a request record and to parse a response record using connection management.

The way that applications were architected, designed, and constructed in the past (the “client/server era”) is quite different from how we perform those tasks in the era of the Web. Even now, when we talk about on demand computing and creating composite applications using services-based architecture styles, we can expect our architectures, designs, and implementations to be quite different in the future.

There are three different levels of back-end systems integration:

- Applications directly to a back-end system (Level 1)

  Before the J2EE Connector Architecture (JCA or J2C) specification was created, many project teams used the Common Connector Framework (CCF) from IBM to integrate with back-end systems. In some instances, developers created their own legacy frameworks using sockets and proprietary protocols between their applications and a corporate back-end system.

  This level of integration is typically where the application layer integrates directly with a back-end system

- Applications to an EIS integration with a J2C component and service (Level 2)

  This level of back-end integration is where an application uses a J2C component that has been exposed as a service that is using Web Services technology.

- Applications to a EIS system process (Level 3)

  This level of back-end integration is where an application invokes a business process, or rather a system process, to handle its back-end integration.

  The key difference between level 3 and the previously discussed levels is that the EIS service or a J2C service are not called directly or explicitly by any application. Using this level of EIS integration, any application (or any enterprise business process) integrates with a back-end system using a system process that is itself exposed as a service. This system process, referred to as the EIS system process in this discussion, contains all of the back-end system integration rules (including logging, exception handling, routing, transformation, configuration management, versioning, and so on) that usually are implemented and maintained by an application.

  As with integration level 2, the EIS system process is not part of the application. It is a business process that is exposed with Web-services technologies using a service-based architecture style.

  The EIS system process can be implemented using Business Process Execution Language (BPEL) and Business Process Choreographer.

Figure 6-1 illustrates the integration levels.

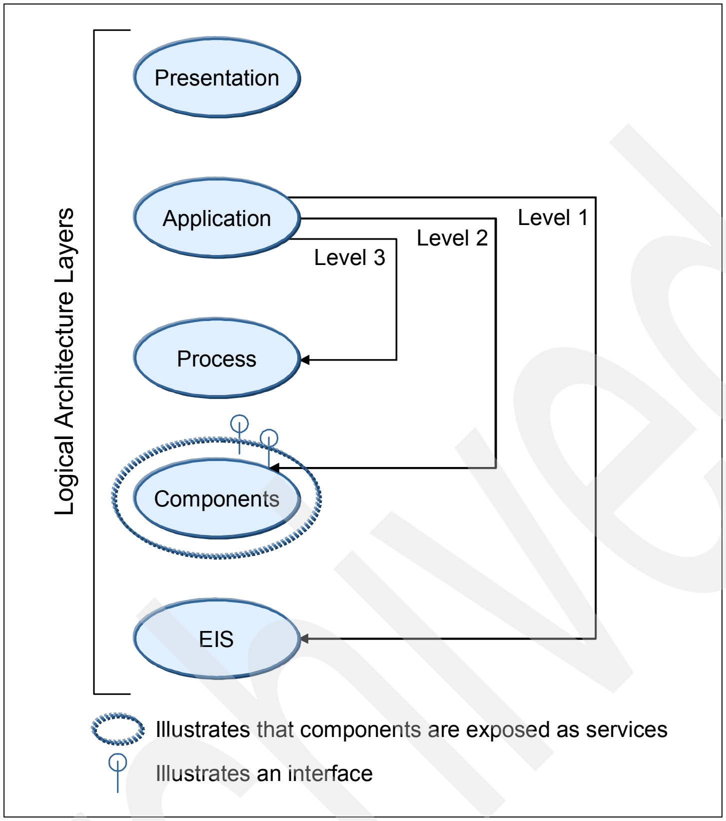


Figure 6-1 EIS integration levels

The scope of this chapter is to explain best practices to access the back-end system directly (Level 1 of integration) and best practices to use Web Services that are part of Level 2 and Level 3 integration. We do not cover BPEL and Business Process Choreographer.

Level 1 of integration

The three types of integration best practices that we discuss for Level 1 of integration are:

- EIS integration with data access layer
- EIS integration with J2EE Connector Architecture (JCA or J2C)
- EIS integration with Java Message Service (JMS)

EIS integration with data access layer

The purpose of the data access layer (see 6.2, “Data access layer” on page 314) is to provide a flexible and portable data programming model that separates the data access, which usually depends on some kind of Enterprise Information System (EIS), from the data itself. In addition, the data access layer decouples the application code from data access code, to enable business logic reuse and simplify application maintenance. The subtopics described are:

- Java Object Serialization (see 6.3, “Java object serialization”)
- Java and SQLJ (see 6.4, “JDBC and SQLJ”)
- Entity Beans (see 6.5, “Entity Beans 2.x”)
- Java Persistence API (see 6.6, “Java Persistence API: Entity Beans 3.0”)
- iBATIS Data Mapper framework (see 6.7, “iBATIS”)
- Java Data Objects (see 6.8, “Java Data Objects”)
- Service Data Objects (see 6.9, “Service Data Objects”)

EIS integration with J2C

The purpose of J2C integration or JCA integration (these two mnemonics refer to J2EE Connector Architecture) in this chapter is to discuss performance tips to access Customer Information Control System (CICS) and IBM Information Management System (IMS) back-end systems. The subtopic described is:

- Java 2 Connector Architecture (see 6.10, “Java 2 Connector Architecture”)

EIS integration with JMS

This section discusses best practices integration with EIS using the Java Message Service (JMS) that is a MOM (Message Oriented Middleware) software. The back-end that represents a message service provider can be a WebSphereMQ for example. The subtopic described is:

- Java Message Service (see 6.11, “Java Message Service”)

Level 2 and Level 3 of integration

Level 2 of integration, which concerns process integration, and level 3 of integration, which concerns business integration, are covered in:

- Web Services (see 6.12, “Web Services”)

6.2 Data access layer

In this section we make a comparison between the technologies described in the following sections related to the data access layer. This comparison describes the features available (Yes) and not available (No) to several technologies.

Table 6-1 Comparison between persistent data storage/retrieval options

Supports:	Serialization	JDBC	EJB 2.x	JDO	JPA
Java objects	Yes	No	Yes	Yes	Yes
Advanced OO concepts	Yes	No	No	Yes	Yes
Transactional integrity	No	Yes	Yes	Yes	Yes
Concurrency	No	Yes	Yes	Yes	Yes
Large data sets	No	Yes	Yes	Yes	Yes
Existing schema	No	Yes	Yes	Yes	Yes
Relational and non-relational databases	No	No	Yes	Yes	No
Queries	No	Yes	Yes	Yes	Yes
Strict standards / portability	Yes	No	Yes	Yes	Yes
Simplicity	Yes	Yes	No	Yes	Yes

If you analyze this table, the serialization approach is simple to develop but does not have the major important features for persistence. Java Database Connectivity (JDBC) supports more options than serialization, and simplicity too. However, JDBC does not have Java Objects support or advanced OO and portability concepts important to high productivity in Java development. EJB 2.x addresses some of the lack of productivity in JDBC.

On the other hand, this is not simple to develop and demands experienced developers to implement advanced topics such as entity relations. The artifacts to do some more advanced functions are very complex. Java Data Objects (JDO) addresses all these features, however, it is not a J2EE standard and does not have many tools available for development support. The future of JDO is not clear. Java Persistence API (JPA) is in fact a standard implemented as part of EJB 3.0 specification. However, it does not implement object oriented databases. The main players in application servers are heading toward this standard.

In the following topics, keep in mind that we have best practices for several technologies compared here. This is very important because there is much legacy code installed in both large and small businesses. These businesses do not have time or money to modify or rewrite all code to new technology adoption. These best practices can help you to improve your systems to make more use of several resources available in WebSphere Application Server environment.

6.3 Java object serialization

Object serialization is the simplest of Java persistence strategies. Object serialization is a process of flattening object graphs to a linear sequence of bytes. Objects graphs are relations realized as a result of the inheritance, association, and aggregation of objects. An object's non-transient instance properties are written to persistent storage in the form of bytes. The values of instance properties are the values in the memory at the time serialization is performed. For a Java object to be serializable, it must at minimum implement the java.io.Serializable interface, which has the structure shown in Example 6-1.

Example 6-1 Java serialization

```
package java.io;
public interface Serializable
{}
```

As you can see, the java.io.Serializable interface does not declare any methods. It is a marker, or tagged, interface. It tells the Java Runtime Environment that the implementing class can be serialized. Example 6-2 shows a sample class that implements this interface.

Example 6-2 Serialization usage example

```
import java.io.Serializable;

public class MySerializableObject extends MySuperClass implements
Serializable
{
 private String property1 = null;
 private String property2 = null;

 public String getProperty1()
 {
    return property1;
 }

 public void setProperty1(String val)
 {
```

```
    property1 = val;
 }
 public String getProperty2()
 {
    return property2;
 }

 public void setProperty2(String val)
 {
    property2 = val;
 }

 private void writeObject(ObjectOutputStream out)
      throws IOException
 {
    out.writeObject (getProperty1 ());
    out.writeObject (getProperty2 ());
 }

 private void readObject (ObjectInputStream in)
      throws IOException, ClassNotFoundException
 {
    setProperty1 ((String) in.readObject ());
    setProperty2 ((String) in.readObject ());
 }
}
```

You do not have to implement the writeObject(...) and readObject(...) methods yourself to perform serialization; the Java Runtime Environment has the default implementation of these methods available. However, you can override these methods and provide your own implementation of how the object state is to be stored.

There are some points about serialization to keep in mind. First, the entire object graph (that is, all parent and referenced classes) is serialized during serialization. Second, all instance variables of a Serializable class should themselves be Serializable unless they have been specifically declared transient, or the writeObject(...) and readObject(...) methods have been overridden to serialize only serializable instance variables. If this latter rule is violated, an exception is thrown at runtime.

Normally, a serialized stream contains only one serialized instance of any given object and makes back references to it from other objects that share references to it. It is often desirable to serialize an object independent of any references that other objects might maintain to it. The unshared read and write methods allow objects to be serialized as new and unique objects, achieving an effect similar to object cloning but with less overhead.

6.3.1 Serialization drawbacks

Serialization involves externalizing object graphs from memory to persistent storage (like a hard disk). This involves a lot of I/O overhead. Generally, serialization is not the optimal choice for applications that:

- Manage hundreds of thousands of megabytes of persistent data
- Update serializable objects frequently

Serialization is a bad choice for storing enterprise data because:

- Serialized streams of bytes are readable only by the Java language. This is a major drawback because enterprise systems are generally heterogeneous, with many applications cooperating with each other and working on the same data.
- Object retrieval involves lot of I/O overhead.
- There is no query language for retrieving data from a serialized object graph.
- Serialization has no built-in security mechanism.
- Serialization does not offer any transaction control mechanisms per se, so it cannot be used within applications that require concurrent access without making use of additional APIs.

6.4 JDBC and SQLJ

The objective of this section is to discuss best practices when the Java Database Connectivity (JDBC) approach is used. We also talk briefly about Structured Query Language for Java (SQLJ). More about SQLJ can be found in Appendix A, “Additional best practices for SQLJ” on page 583. Of course, we have other alternatives to use, such as Entity Beans, Java Data Objects (JDO), SDO, and so on. But the main goal here, when you have existing code, is that you can tune to gain performance. This can be of benefit if you have many legacy applications implemented and you do not have time or budget to modify or rewrite your system to use new technologies.

If you use a framework to persist data for your application, in reality the framework takes care of JDBC connections for you. However, do not expect every framework to implement the best practices for you.

6.4.1 Driver types: Overview

JDBC defines how a Java application communicates with databases (). In JDBC we have a driver, which is at the middle layer (it implements the JDBC API interfaces that maps Java to database specific language), and a database, which is at the bottom (to store physical data). The driver available depends on Database Management System (DBMS) vendors. Refer to Figure 6-2.

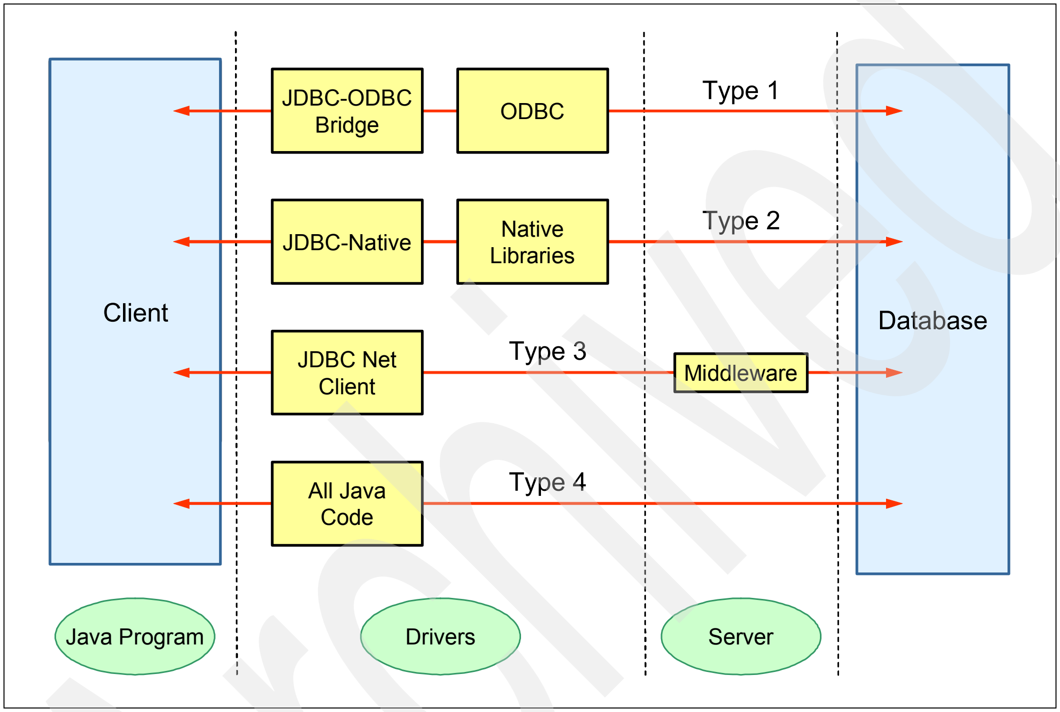


Figure 6-2 JDBC driver types

In this figure, there are four types of drivers, briefly described in Table 6-2.

Table 6-2 JDBC types short description

JDBC type	Description
Type 1	JDBC convert calls in Open Database Connectivity (ODBC) using JDBC-ODBC bridge. This is used when the database does not have a JDBC driver.
Type 2	This driver uses a database client and uses native drives to connect to the database. For this you require a client database installed in the machine that you have the client code. In DB2 the DB2JDBC Type 2 driver is quite popular and is often referred to as the app driver.

JDBC type	Description
Type 3	The JDBC Type 3 driver is a pure Java implementation that must talk to middleware for communication with the server. In DB2 this approach is done by a DB2 JDBC Applet Server and this driver was designed to enable Java applets for access to DB2 data sources. In this scenario, the application talks to another machine where a DB2 client has been installed. The JDBC Type 3 driver is often referred to as the net driver, appropriately named after its package name (COM.ibm.db2.jdbc.net.)
Type 4	This driver type is 100% Java and does not require middleware or native drivers. This is the predominant type used today.

6.4.2 Data type mapping

For optimal performance, we recommend properly mapping the Java data types used in the application for your database type. In this topic we relate Java data types to the DB2 column data types. The main reason for this is to avoid unnecessary conversions and thus reduce performance. Table 6-3 shows a summary of recommended mappings of Java to DB2 data types. Primitive Java types should be used for NOT NULL columns and corresponding wrapper types for nullable columns.

Table 6-3 Mapping data types

Java data type	DB2 data type	Comment
boolean Boolean	SMALLINT	No direct mapping in DB2; SMALLINT is the best match. Zero indicates false, and any non-zero value indicates true.
byte Byte	SMALLINT	No direct mapping in DB2; SMALLINT is the best match.
short Short	SMALLINT	
int Integer	INTEGER	
long Long	DECIMAL(19.0)	No 64-bit integer in type in DB2; DECIMAL with precision 19 can hold all long values.
java.math.BigInteger	DECIMAL(19.0)	
float Float	REAL	

Java data type	DB2 data type	Comment
double	DOUBLE or FLOAT	FLOAT is a synonym for DOUBLE in DB2.
java.math.BigDecimal	DECIMAL(p,s)	p = precision, s = scale.
java.lang.String	CHAR(n)	Fixed-width column of length n.
	VARCHAR(n)	Variable-width column of maximum length n.
	GRAPHIC(n)	Fixed-width column of length n.
	VARGRAPHIC(n)	Variable-width column of maximum length n.
byte[]	CHAR(n) FOR BIT DATA	
	VARCHAR(n) FOR BIT DATA	
java.sql.Date	DATE	
java.sql.Time	TIME	
java.sql.Timestamp	TIMESTAMP	
java.sql.Blob	BLOB(n)	
java.sql.Clob	CLOB(n)	
	DBCLOB(n)	

6.4.3 Using static SQL

Static SQL is generally faster than dynamic SQL and should be used wherever it is possible. When using static SQL, parsing and access path calculation are done at compile time and not at runtime. Java's implementation of static SQL is called SQLJ. Figure 6-3 shows the differences between dynamic and static SQL.

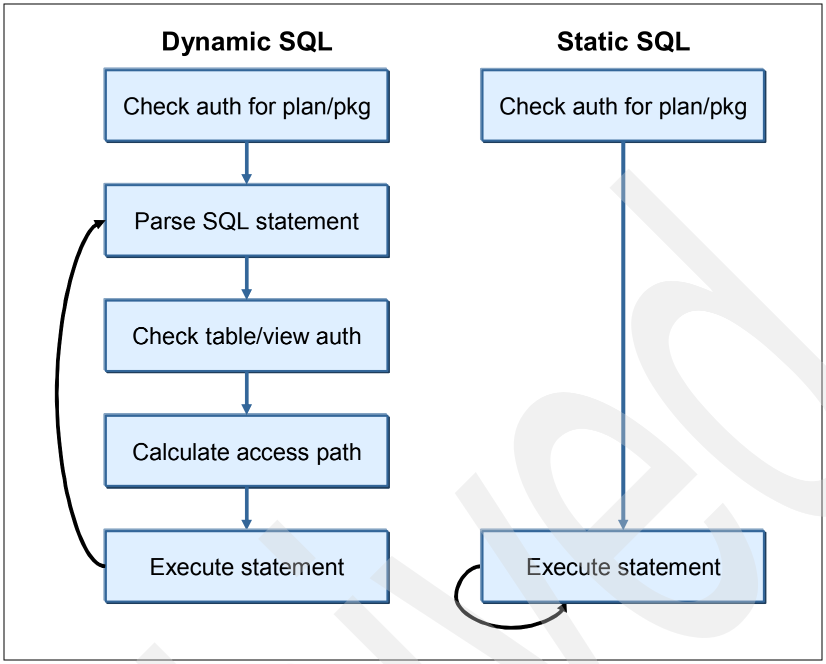


Figure 6-3 Dynamic versus static SQL

In some cases, it is not possible to use static SQL, for example, if a GUI application allows a large number of options on your SQL queries. In this case, the dynamic statement caching should be turned on in the DB2 database. You can find more information about this in Chapter 10 of *Performance Monitoring and Best Practices for WebSphere on z/OS*, SG24-7269.

Example 6-3 and Example 6-4 show the equivalent code necessary in JDBC and SQLJ respectively to select the address for a given name from table EMP.

Example 6-3 JDBC

```
java.sql.PreparedStatement ps = con.prepareStatement("SELECT ADDRESS
        FROM EMP WHERE NAME=?");
ps.setString(1, name);
java.sql.ResultSet rs = ps.executeQuery();
rs.next();
addr = rs.getString(1);
rs.close();
```

Example 6-4 SQLJ

```
#sql [con] { SELECT ADDRESS INTO :addr FROM EMP WHERE NAME=:name };
```

The chart in Figure 6-4 shows the results of a simple SQL performance comparison between JDBC and SQLJ. Using JDBC, the SQL statement was preloaded into the dynamic statement cache and always found there during the measurement. The measured SQL statements are:

- Open - 4 Fetch - Close, selecting four rows containing columns of different data types
- Four inserts, inserting four rows containing columns of different data types
- Four singleton selects, selecting four rows containing columns of different data types

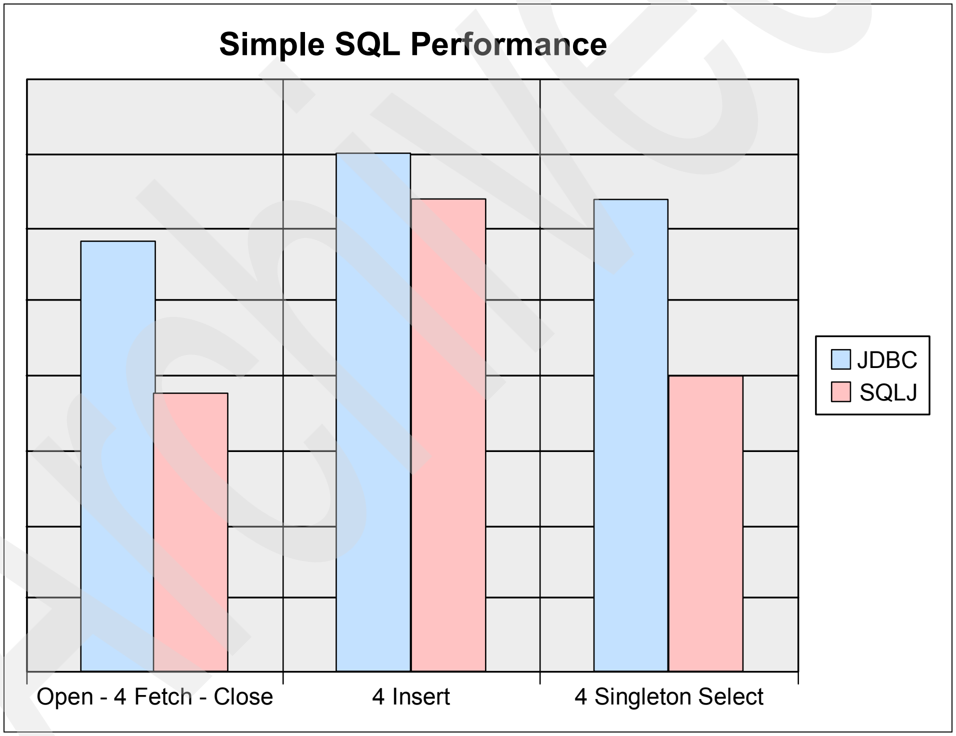


Figure 6-4 Comparison JDBC and SQLJ

SQLJ applications are portable even to DBMSs that do not support SQLJ. When the preparations, such as precompile and BIND, for an SQLJ program have not been done, the SQLJ runtime automatically emits JDBC calls. SQLJ is key to high performance, but does not lock a customer into a particular DBMS. The only performance advantage of using JDBC over SQLJ is where the SQL statements are very complex and literals are provided.

For more information about SQLJ, see Appendix A., “Additional best practices for SQLJ” on page 583.

6.4.4 Use a data source to get connections

To get a connection, you have two approaches using java.sql.DriverManager or using javax.sql.Datasource (only available with JDBC 2.0 specification). See the Example 6-5 and Example 6-6.

Example 6-5 DriverManager code snippet using java.sql.DriverManager

```
//This part is recommeded to put in some initialization class method
//or servlet.init() method for example if you use Servlets
try {
  // Load the DB2(R) Universal JDBC Driver with DriverManager
  Class.forName("com.ibm.db2.jcc.DB2Driver");
} catch (ClassNotFoundException e) {
     e.printStackTrace();
}
//In another code part
try {
    user = getUser();
    password = getPassword();
    java.sql.Connection con =
        java.sql.DriverManager.
        getConnection("jdbc:db2:Sample",user,password);
} catch (SQLException e) {
    e.printStackTrace();
}
```

Example 6-6 Code snippet using javax.sql.Datasource

```
//This part is recommeded to put in some initialization class method
//or servlet.init() method for example if you use Servlets
java.util.Hashtable env = new java.util.Hashtable();
env.put(Contex.INITIAL_CONTEXT_FACTORY,
          "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
ctx = new InitialContext(env);
java.sql.DataSource ds =
(javax.sql.DataSource)ctx.lookup("jdbc/SAMPLE");
//In another code part
try {
```

```
    user = getUser();
    password = getPassword();
    java.sql.Connection con =
        ds.getConnection("jdbc:db2:Sample",user,password);
} catch (SQLException e) {
    e.printStackTrace();
}
```

We recommend using DataSource. The advantage of getting connections from this approach is that DataSource provides a connection pool that offers some guarantee for an application to reuse connections. The DataSource object is retrieved from Java Naming and Directory Interface (JNDI). The implementation of DataSource is done by vendor, for example you can find this feature in WebSphere. The vendor simply creates a DataSource implementation class and binds it to the JNDI tree. Example 6-7 shows how a vendor creates an implementation class and binds it to the JNDI tree.

Example 6-7 Bind example in JNDI tree

```
DataSourceImpl dsi = new DataSourceImpl();
dsi.setServerName("db2");
dsi.setDatabaseName("Demo");
Context ctx = new InitialContext();
ctx.bind("jdbc/demoDB", dsi);
```

This code registers the DataSourceImpl object to the JNDI tree, then the programmer can get the DataSource reference from JNDI tree without knowledge of the underlying technology.

The best practice here is avoiding the overhead of acquiring a javax.sql.DataSource for each SQL access. This is an expensive operation that severely impacts the performance and scalability of the application. Instead, servlets should acquire the javax.sql.DataSource in the Servlet.init() method (or some other thread-safe method) and maintain it in a common location for reuse. See Example 6-8 about DataSource usage in a Servlet.

Example 6-8 DataSource usage in a Servlet

```
public class DataSourceServletExample extends HttpServlet
{
  // Caching the DataSource - It is obtained in the Servlet.init()
method
  private javax.sql.DataSource ds = null;

  // This Happens Once and is Reused
  public void init(ServletConfig config) throws ServletException
```

```
  {
    super.init(config);
    Context ctx = null;
    try
    {
      java.util.Hashtable env = new java.util.Hashtable();
      env.put(Contex.INITIAL_CONTEXT_FACTORY,
          "com.ibm.ejs.ns.jndi.CNInitialContextFactory");

      ctx = new InitialContext(env);
      ds = (javax.sql.DataSource)ctx.lookup("jdbc/SAMPLE");
      ctx.close();
    }
    catch(Exception es)
    {
       es.printStackTrace();
    }
    finally{
    try{
      if (ctx!=null)
      ctx.close()
    }
    catch(Exception e){//Make your handlings}
   }
  }
}
```

When you use DataSource Object, you can control the maxconnections, minconnections, connectiontimeout and Unused Timeout in a central point. It helps technical support for capacity planning connections and if necessary ask Database administrations to increase the maxconnections in the database server. For example, a data source A with 30 maxconnections for a DB2 database instance, and another data source B with 20 maxconnections for the same DB2 database instance, represent for the DBA, 50 connections for that database instance. See the parameters for DataSource in WebSphere Application Server in Table 6-4 for a short description.

Table 6-4 DataSource configuration parameters

Parameter Name	Description
Maximum Connections	Specifies the maximum number of physical connections that you can create in this pool.
Minimum Connections	Specifies the minimum number of physical connections to maintain.

Parameter Name	Description
Connection Timeout	Specifies the maximum time that a application can wait in a queue to receive a connection from Datasource.
Reap Time	Specifies the interval, in seconds, between runs of the pool maintenance thread.
Unused Timeout	Specifies the interval in seconds after which an unused or idle connection is discarded.
Aged Timeout	Specifies the interval in seconds before a physical connection is discarded.
Purge Policy	Specifies how to purge connections when a stale connection or fatal connection error is detected

For more details about DataSource configuration in WebSphere, see 7.2.9, “Tuning data sources and associated connection pools” on page 554.

For example, if you are using a configuration with 3 minconnections and 10 maxconnections, the data source starts with 3 connections and only gets a new connection if more than 3 clients are requesting a connection at same time. If the number of clients reaches more than 10, the remaining clients are put in a queue and a connection timeout counter is started. If the client thread reaches the connection timeout value, an exception is thrown.

We also recommend to use one data source for each application or for each application module. For example an application A, used for administration account purposes in a bank, uses 100ms for Unused Timeout timeout because it uses the connections for many data selects. In another case an application B, used for login account purposes, has to use a Unused Timeout of 10ms because this application uses the connection fast. In short, a group of applications uses a connection for a long time and another uses it for a short time. This means that the best practice here is instead of using only one data source, use two data sources to configure the best tuning parameters for each application.

6.4.5 Control your transactions

What is a transaction? Before we define this term, first we must define the concept of application state. An application's state encompasses all of the in-memory and on-disk data items that affect the application's operation — everything the application “knows.” Application state can be stored in memory, in files, or in a database. In the event of a system failure — for example, if the application, network, or computer system crashes — we want to ensure that when the system is restarted, the application's state can be restored.

We can now define a transaction as a related collection of operations on the application state, which has the properties of atomicity, consistency, isolation, and durability. These properties are collectively referred to as ACID properties.

- *Atomicity* means that either all of the transactions' operations are applied to the application state, or none of them are applied; the transaction is an indivisible unit of work.
- *Consistency* means that the transaction represents a correct transformation of the application state — that any integrity constraints implicit in the application are not violated by the transaction. In practice, the notion of consistency is application-specific. For example, in an accounting application, consistency would include the invariant that the sum of all asset accounts equal the sum of all liability accounts.
- *Isolation* means that the effects of one transaction do not affect other transactions that are executing concurrently; from the perspective of a transaction, it appears that transactions execute sequentially rather than in parallel. In database systems, isolation is generally implemented using a locking mechanism. The isolation requirement is sometimes relaxed for certain transactions to yield better application performance.
- *Durability* means that once a transaction successfully completes, changes to the application state can survive failures.

What do we mean by "survive failures?" What constitutes a survivable failure? This depends on the system, and a well-designed system is able to explicitly identify the faults from which it can recover. The transactional database running on a desktop workstation is robust to system crashes and power failures, but not to my office building burning down. A bank would likely not only have redundant disks, networks, and systems in its data center, but perhaps also have redundant data centers in separate cities connected by redundant communication links to allow for recovery from serious failures such as natural disasters. Data systems for the military might have even more stringent fault-tolerance requirements.

JDBC transactions are controlled using the Connection object. The JDBC Connection interface (java.sql.Connection) provides two transaction modes: auto-commit and manual commit. The java.sql.Connection offers the following methods for controlling transactions (see Example 6-9).

Example 6-9 Connection Interface

```
public interface Connection {
 //Other Methods declarations
 boolean getAutoCommit();
 void setAutoCommit(boolean autocommit);
 void commit();
 void rollback();
 //Other Methods declarations
}
```

If a JDBC transaction uses AutoCommit mode *true*, this means that a transaction starts and commits after each statement's execution on a connection. Therefore, a programmer does not have to write a commit() method explicitly after each statement. This mechanism is good for programmers that want to execute a single statement. However, to commit multiples statements, you must use AutoCommit mode of *false* and issue the commit() method after a set of statements execute. This is called a batch transaction. Use rollback() in a catch block to rollback the transaction whenever an exception occurs in your program. The code sample in Example 6-10 illustrates the batch transaction approach.

Example 6-10 Transaction control JDBC Usage

```
Connection conn =null;
  PreparedStatement pstmt1,pstmt2;

  try {
    InitialContext ctx = new InitialContext();

    ds = (DataSource)ctx.lookup("jdbc/BankDS");
    conn = ds.getConnection();
    conn.setAutoCommit(false);

    pstmt1 = conn.prepareStatement("UPDATE ACCOUNT SET BALANCE = ?
WHERE CUSTOMER_ID = ?");
    pstmt1.setString(1, "500.00");
    pstmt1.setString(1, "0964564");
    pstmt1.executeUpdate();

    pstmt2 = conn.prepareStatement("INSERT INTO
TRANSRECORD(DETAILS,TYPE,AMOUNT,TRANSRECORD_ID,ACCOUNT_ID)
VALUES(?,?,?,?,?)");
    pstmt2.setString(1, "withdraw");
    pstmt2.setString(1, "D");
    pstmt2.setString(1, "200.00");
```

```
    pstmt2.setString(1, "960402382");
    pstmt2.executeUpdate();

    conn.commit();
  } catch (NamingException e) {
    e.printStackTrace();
  } catch (SQLException e) {
    try {
    conn.rollback();
    } catch (SQLException e1) {
    e1.printStackTrace();
    }
    e.printStackTrace();
  } }
  finally{
   //Finally procedures to close PrepareStatements and Connections
}
```

With JDBC transaction demarcation, you can combine multiple SQL statements into a single transaction, and this batch transaction gives good performance by reducing commit calls after each statement's execution.

However, one of the drawbacks of JDBC transactions is that the transaction's scope is limited to a single database connection. A JDBC transaction cannot span multiple databases.

6.4.6 Using JTA

Next, we see how transaction demarcation is done using JTA. Because JTA is not as widely known as JDBC, we start with an overview.

JTA overview

The Java Transaction API (JTA) and its sibling, the Java Transaction Service (JTS), provide distributed transaction services for the J2EE platform. A distributed transaction involves a transaction manager and one or more resource managers. A resource manager is any kind of persistent datastore. The transaction manager is responsible for coordinating communication between all transaction participants. The relationship between the transaction manager and resource managers is shown in Figure 6-5.

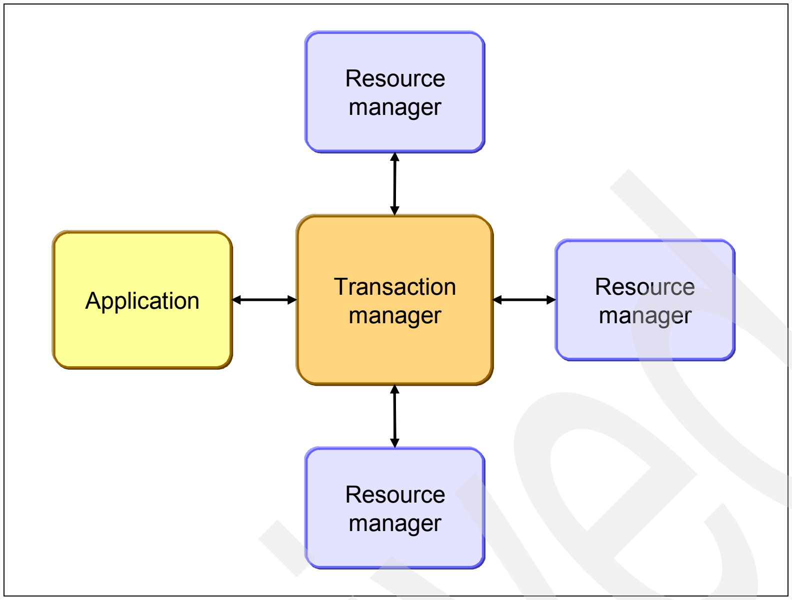


Figure 6-5 Transaction and Resources Managers

JTA transactions are more powerful than JDBC transactions. Whereas a JDBC transaction is limited to a single database connection, a JTA transaction can have multiple participants. Any one of the following Java platform components can participate in a JTA transaction:

- JDBC connections
- JDO PersistenceManager objects
- JMS queues
- JMS topics
- Enterprise JavaBeans
- A resource adapter that complies with the J2EE Connector Architecture specification

Transaction demarcation with JTA

To demarcate a transaction with JTA, the application invokes methods on the javax.transaction.UserTransaction interface. Example 6-11 shows a typical JNDI lookup for the UserTransaction object.

Example 6-11 User Transaction instantiation code snippet

```
import javax.transaction.*;
import javax.naming.*;
// ...
InitialContext ctx = new InitialContext();
```

```
Object txObj =
  ctx.lookup("java:comp/UserTransaction");
UserTransaction utx = (UserTransaction) txObj;
```

The foregoing code sample only explains the UserTransaction object instantiation. However, keep in mind that as a DataSource, you should cache the object reference because a JNDI tree search is an expensive operation. After the application has a reference to the UserTransaction object, it can start the transaction as shown in Example 6-12.

Example 6-12 Starting a transaction with JTA

```
InitialContext ctx = new InitialContext(); utx.begin();
if(ds == null)
    ds = (XADataSource)ctx.lookup("jdbc/BankDS");
   conn = ds.getXAConnection().getConnection();
   pstmt1 = conn.prepareStatement("UPDATE ACCOUNT SET BALANCE = ? WHERE
CUSTOMER_ID = ?");
   pstmt1.setString(1, "500.00");
   pstmt1.setString(2, "0964564");
   pstmt1.executeUpdate();

   if(ds2 == null)
    ds2 = (XADataSource)ctx.lookup("jdbc/CrediCardDS");
   conn2 = ds2.getXAConnection().getConnection();
   pstmt2 = conn2.prepareStatement("INSERT INTO
TRANSRECORD(DETAILS,TYPE,AMOUNT,TRANSRECORD_ID,ACCOUNT_ID)
VALUES(?,?,?,?,?)");
   pstmt2.setString(1, "monthly debit");
   pstmt2.setString(2, "D");
   pstmt2.setString(3, "200.00");
   pstmt2.setString(4, "960402382");
   pstmt2.executeUpdate();
   utx.commit();
} catch (Exception e) {
    try {
         if (utx != null)
           utx.rollback();
        } catch (Exception e1) {
             e1.printStackTrace();
         }
     e.printStackTrace();
  }
  finally{//Close Statements and Connections Procedures
```

When the application invokes a commit(), the transaction manager uses a two-phase commit protocol to end the transaction.

For transaction control, the javax.transaction.UserTransaction interface provides the following methods:

- public void begin()
- public void commit()
- public void rollback()
- public int getStatus()
- public void setRollbackOnly()
- public void setTransactionTimeout(int)

To start a transaction, the application calls begin(). To end a transaction, the application calls either commit() or rollback().

For more information on JTA, see the following URL:

`http://java.sun.com/products/jta/`

Using JTA and JDBC

Developers often use JDBC for low-level data operations in Data Access Object (DAO) classes. If you plan to demarcate transactions with JTA, you require a JDBC driver that implements the javax.sql.XADataSource, javax.sql.XAConnection, and javax.sql.XAResource interfaces. A driver that implements these interfaces is able to participate in JTA transactions. An XADataSource object is a factory for XAConnection objects. XAConnections are JDBC connections that participate in JTA transactions.

You are required to set up the XADataSource using your application server's administrative tools. Consult the application server documentation and the JDBC driver documentation for specific instructions.

J2EE applications look up the data source using JNDI. When the application has a reference to the data source object, to obtain a connection to the database, it calls javax.sql.DataSource.getConnection().

XA connections are different from non-XA connections. Remember that XA connections are participating in a JTA transaction, meaning that XA connections do not support JDBC's auto-commit feature. Also, the application must not invoke java.sql.Connection.commit() or java.sql.Connection.rollback() on an XA connection. Instead, the application should use UserTransaction.begin(), UserTransaction.commit(), and UserTransaction.rollback().

Choosing the best approach

We have discussed how to demarcate transactions with both JDBC and JTA. Each approach has its advantages and you have to decide which one is most appropriate for your application.

Implement your DAO classes with JDBC when:

- Transaction demarcation code must be embedded inside the DAO class.
- The caller has no way to demarcate the transaction.
- Transaction scope is limited to a single JDBC Connection.

JDBC transactions are not always suitable for complex enterprise applications. Demarcate your transactions with JTA when:

- Your transactions span multiple DAOs or multiple databases
- Transaction demarcation code must separated from the DAO
- The caller is responsible for demarcating the transaction
- The DAO participates in a global transaction

The JDBC approach is attractive due to its simplicity; the JTA approach offers greater flexibility. The implementation you choose depends on the specific requirements of your application.

Choose the best isolation level

Isolation level represent how a database maintains the level of data integrity and concurrency. A higher isolation level leads to increased data integrity, while a lower isolation level leads to more concurrency and better performance. A lower isolation level also decreases row locking, so the probability of deadlock is reduced.

JDBC supports five isolation types:

- TRANSACTION_NONE
- TRANSACTION_READ_UNCOMMITED
- TRANSACTION_READ_COMMITED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE

Before we talk about JDBC isolation types in more depth, we review dirty reads, phantom reads, and non-repeatable reads, which can occur due to concurrent transactions.

- Dirty read problem:

  The following steps illustrate the dirty read problem:

  a. Database row has PRODUCT = X and PRICE = 20.
  b. Connection1 starts Transaction (T1).
  c. Connection2 starts Transaction2 (T2).
  d. T1 updates PRICE =40 for PRODUCT = X.
  e. Database has now PRICE = 40 for PRODUCT = X.
  f. T2 reads PRICE = 20 for PRODUCT = X.
  g. T2 commits the transaction.
  h. T1 rollbacks the transaction because of some problem.

  The problem is that T2 should be PRICE=40 for PRODUCT=X. However, it incorrectly gets PRICE=20 because of the uncommitted read. It is very dangerous in critical transactions if you read inconsistent data. If you are sure that your data is not going to be accessed concurrently, then you can allow this situation by setting TRANSACTION_READ_UNCOMMITED or TRANSACTION_NONE. This improves performance. Otherwise, to avoid this problem, you have to use TRANSACTION_READ_COMMITED.

- Unrepeatable read problem:

  The following steps illustrate the unrepeatable read problem:

  a. Database row has PRODUCT = X and PRICE = 20.
  b. Connection1 starts Transaction1 (T1).
  c. Connection2 starts Transaction2 (T2).
  d. T1 reads PRICE =20 for PRODUCT = X.
  e. T2 updates PRICE = 40 for PRODUCT = X.
  f. T2 commits the transaction.
  g. Database row has PRODUCT = X and PRICE = 40.
  h. T1 reads PRICE = 40 for PRODUCT = X.
  i. T1 commits the transaction.

  Here the problem is that Transaction1 reads 10 the first time and reads 20 the second time. However, the correct value would always be 10 during the T1 transaction. You can control this problem by setting isolation level as TRANSACTION_REPEATABLE_READ.

- Phantom read problem:

  The following steps illustrate the phantom read problem:

  a. Database has a row PRODUCT = X and SELLER_ID = 123.
  b. Connection1 starts Transaction1 (T1).
  c. Connection2 starts Transaction2 (T2).
  d. T1 selects a row with a condition SELECT PRODUCT WHERE SELLER_ID = 123.
  e. T2 inserts a row with a condition INSERT PRODUCT= Z WHERE

     SELLER_ID= 123.
  f. T2 commits the transaction.
  g. Database has 2 rows with that condition.
  h. T1 select again with a condition SELECT PRODUCT WHERE SELLER_ID=123 and gets 2 rows instead of 1 row.
  i. T1 commits the transaction.

  The problem here is that T1 must get 1 row instead of 2 rows from the SELECT operation during T1 life time. Setting isolation level as TRANSACTION_SERIALIZABLE solves this problem.

See Table 6-5 for an overview of isolation types.

Table 6-5 Isolation types

Isolation Types	Short Explanation	Performance
TRANSACTION_NONE	No isolation level is configured.	FASTEST
TRANSACTION_READ_UNCOMMITTED	Isolation level permits dirty reads, nonrepeatable reads, and phantom reads.	FASTEST
TRANSACTION_READ_COMMITTED	Isolation level prohibits dirty reads only.	FAST
TRANSACTION_REPEATABLE_READ	Isolation level is the second highest isolation level. It prohibits dirty reads and nonrepeatable reads, but allows phantom reads.	MEDIUM
TRANSACTION_SERIALIZABLE	Isolation level is the highest, most restrictive, isolation level. It prohibits dirty reads, nonrepeatable reads, and phantom reads.	SLOW

As you have seen, the isolation levels have a huge impact on application performance. In Example 6-13. you can see the JDBC usage for isolation level for using the java.sql.Connection interface.

Example 6-13 java.sql.Connection methods and constants for isolation level

```
public interface Connection {
public static final int  TRANSACTION_NONE = 0
public static final int  TRANSACTION_READ_COMMITTED = 2
public static final int  TRANSACTION_READ_UNCOMMITTED = 1
public static final int  TRANSACTION_REPEATABLE_READ = 4
public static final int  TRANSACTION_SERIALIZABLE = 8
int getTransactionIsolation();
void setTransactionIsolation(int isolationlevelconstant);
}
```

You can get the existing isolation level with getTransactionIsolation() method and set the isolation level with setTransactionIsolation(int isolationlevelconstant) by passing the foregoing constants to this method.

Choosing the right isolation level for your program

Choosing the right isolation level for your program depends on your application requirements. If you have to write a program that searches product information in a database, you can choose TRANSACTION_READ_UNCOMMITED because you do not have to worry about database updates or insertion. Another search gets the new values. This approach improves performance significantly.

If you write a critical program that requires full integrity, such as financial analysis, you can choose TRANSACTION_SERIALIZABLE for maximum safety. But keep in mind that there is a price, a trade-off between the safety and performance: more safety but less performance.

If you do not have to deal with concurrent transactions in your application, then the best choice is TRANSACTION_NONE to improve performance.

Two other isolation levels are more specific. TRANSACTION_READ_COMMITED isolation is the better choice if your application requires only committed records. However, if your application has to read a row exclusively until you finish a unit of work (transaction), then TRANSACTION_REPEATABLE_READ is the best choice.

The last key point here is to be careful when selecting isolation support because some database providers do not support all these values.

6.4.7 Releasing resources best practices

Remember that when you use the connection pool, closing a connection means that it returns back to the connection pool rather than closing the direct connection to the database. If a resource is not closed, for example, java.sql.ResultSets or java.sql.Statements, the number of database cursors increases hugely in your application. Also if a connection is not sending data to a data source, the maxconnection can be easily reached, causing access problems in your application. The question is, what safe approach to follow to close unusable resources?

Use the *finally* syntax to close resources such as ResultSets, Statements, and Database Connections, in this order. Using this approach to release resources acquired in a method is reliable but can easily get unwieldy when multiple resources are involved. Consider a method that uses a JDBC Connection to execute a query and iterate the ResultSet. It acquires a Connection, uses it to create a Statement, and executes the Statement to yield a ResultSet. But the intermediate JDBC objects Statement and ResultSet have close() methods of their own, and they should be released when you are done with them. However, the “obvious” way to clean up, shown in Example 6-14, does not work.

Example 6-14 Bad practice to release multiple resources

```
public void enumerateFoo() throws SQLException {
  Statement statement = null;
  ResultSet resultSet = null;
  Connection connection = getConnection();
  try {
    statement = connection.createStatement();
    resultSet = statement.executeQuery("SELECT * FROM Foo");
    // Use resultSet
  }
  finally {
    if (resultSet != null)
     resultSet.close();
    if (statement != null)
     statement.close();
    connection.close();
  }
}
```

The reason this “solution” doesn't work is that the close() methods of ResultSet and Statement can themselves throw an SQLException, which could cause the later close() statements in the final block not to execute. That leaves you with several choices, all of which are annoying: wrap each close() with a try..catch block, nest the try...finally blocks as shown in Example 6-15, or write some sort of mini-framework for managing the resource acquisition and release.

Example 6-15 Best Practice example to deliver resources

```
public void enumerateBar() throws SQLException {
  Statement statement = null;
  ResultSet resultSet = null;
  Connection connection = getConnection();
  try {
    statement = connection.createStatement();
    resultSet = statement.executeQuery("SELECT * FROM Bar");
    // Use resultSet
  }
  finally {
    try {
      if (resultSet != null)
        resultSet.close();
    }
    finally {
      try {
        if (statement != null)
          statement.close();
      }
      finally {
        connection.close();
      }
    }
  }
}

private Connection getConnection() {
  //Get a Connection
  return connection;
}
```

Keep in mind that when you deliver resources, everything can throw an exception. You then have to look at the method specification to see which exceptions it can throw.

6.4.8 Optimization with Statements

A JDBC Statement object is used to send your SQL statements to the Database Management System (DBMS), and should not be confused with an SQL statement. A JDBC Statement object is associated with an open connection, and not any single SQL statement. You can think of a JDBC Statement object as a channel sitting on a connection, passing one or more of your SQL statements (which you ask it to execute) to the DBMS. They also provide some methods to fine tune performance. Programmers can overlook these fine tuning methods, resulting in poor performance. The following tips can help you improve performance by using Statement interfaces:

- Choose the right Statement interface.
- Do batch update.
- Do batch retrieval using Statement.
- Use Statement with multiple results.

Choose the right Statement interface

Three types of Statement interfaces in JDBC can represent the SQL query and execute that query: Statement, PreparedStatement, and CallableStatement.

Statement is used for static SQL statements with no input and output parameters, PreparedStatement is used for dynamic SQL statements with input parameters, and CallableStatement is used for dynamic SQL statements with both input and output parameters. However, PreparedStatement and CallableStatement can be used for static SQL statements as well. CallableStatement is mainly meant for stored procedures. Here are some tips:

- Use PreparedStatement over Statement if you are writing an SQL statement to be used more than once — it performs better as a PreparedStatement than as a Statement object. Every time you process a statement, you go through a two step process: the statement is prepared (parsing), and the statement is processed (compiled). When you use a prepared statement, the statement is prepared only at the time that it is constructed, not each time it is processed. In other words, it is pre-parsed and pre-compiled by the database once for the first time and then onwards it reuses the parsed and compiled statement. Though it is recognized that a PreparedStatement performs faster than a Statement, this advantage is often neglected by programmers.
- Use CallableStatement if you require SQL procedures with high performance for extensive operations. CallableStatement is used to call stored procedures. CallableStatement gives better performance when compared to PreparedStatement and Statement when there is a requirement for a single request to process multiple complex statements. All stored procedure parsing and compiling is done in the database and this approach improves performance. However, there is a price for it: you lose portability.

Do batch update

After JDBC 2.0, you can use a powerful API for batch processing. Batch processing allows you to accumulate a group of SQL statements and send them for processing in one go. A typical batch processing scenario might involve a banking application that updates a number of accounts every quarter. Batch processing is a powerful feature in that it reduces the number of round trips from Java code to a database.

The Statement interface provides an addBatch(String) method for adding an SQL statement to a batch. When you have added all your SQL statements to the batch, you can execute them in one go using the executeBatch() method.

The executeBatch() method then executes the SQL statements and returns an array of int values. This array contains the number of rows affected by each statement. Putting a SELECT statement or other ResultSet-returning SQL in a batch results in an SQLException.

A simple example of batch processing with the java.sql.Statement is shown in Example 6-16.

Example 6-16 batch code sample

```
Statement stmt = conn.createStatement();
stmt.addBatch("DELETE FROM Users");
stmt.addBatch("INSERT INTO Users VALUES('rod', 37, 'circle')");
stmt.addBatch("INSERT INTO Users VALUES('jane', 33, 'triangle')");
stmt.addBatch("INSERT INTO Users VALUES('freddy', 29, 'square')");
int[] counts = stmt.executeBatch();
```

Batch processing is a nice way to handle SQL code when you do not know how many times a particular statement is going to run. For instance, if we tried to insert 100 records into a database without batching, the performance might be affected. If we wrote a script to add 10,000 records, things could get nasty. Adding batch processing helps to improve the performance, and in the latter case even improves the readability of the code.

Java object serialization does not support batch processing. Typically, serialization involves working on the extent (the association graph) of an object, so batch processing does not make sense in such a case. Batch processing thus affords you a flexibility in terms of the timing and grouping of data updates that is not necessarily available with serialization.

Do batch retrieval using Statement

When an SQL query is executed, the number of rows of data that a driver physically copies from the database to the client is called the fetch size. If you have to performance-tune a query, you can adjust the fetch size to fit your requirements.

With getFechSize() you get the default number of rows that is provided by the driver. With setFetchSize() you can increase the number of rows to be fetched at a time from database, improving performance. In this case, the next time data has to be fetched from the database, the driver copies over as many rows as are specified by the current fetch size. See the code snippet in Example 6-17.

Example 6-17 Code snippet fetch size features usage

```
// Get the fetch size of a statement
 PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM ACCOUNT
WHERE WHERE CUSTOMER_ID = ?");
 pstmt.setString(2, "0964564");
 int fetchSize = pstmt.getFetchSize();

 // Set the fetch size on the statement
 pstmt.setFetchSize(100);

 // Create a result set
 ResultSet resultSet = pstmt.executeQuery();
```

Use Statement with multiple results

A limitation of the JDBC 2 specification is that statements that return multiple results must have only one ResultSet open at any given time. As a part of the changes in JDBC 3.0, the specification allows the Statement interface to support multiple open ResultSets. It is important to note, however, that the execute() method still closes any ResultSets that were opened from a previous call to execute(). So, to support multiple open results, the Statement interface adds an overloaded version of the method getMoreResults(). The new form of the method takes an integer flag that specifies the behavior of previously opened ResultSets when the getResultSet() method is called. The interface defines the flags as shown in Table 6-6.

Table 6-6 Statement options for previous ResultSets behavior

Option	Description
CLOSE_ALL_RESULTS	All previously opened ResultSet objects should be closed when calling getMoreResults().
CLOSE_CURRENT_RESULT	The current ResultSet object should be closed when calling getMoreResults().
KEEP_CURRENT_RESULT	The current ResultSet object should not be closed when calling getMoreResults().

A code sample using multiple ResultSets is shown in Example 6-18.

Example 6-18 Multiple ResultSets usage example

```
String procCall = "";
// Set the value of procCall to call a stored procedure.
// ...

CallableStatement cstmt = connection.prepareCall(procCall);
boolean retval = cstmt.execute();
if (retval == false) {
  // The statement returned an update count, so handle it.
  // ...
} else { // ResultSet
  ResultSet rs1 = cstmt.getResultSet();
  // ...

  retval = cstmt.getMoreResults(Statement.KEEP_CURRENT_RESULT);
  if (retval == true) {
    ResultSet rs2 = cstmt.getResultSet();

    // Both ResultSets are open and ready for use.
    rs2.next();
    rs1.next();
    // ...
  }
}
```

6.4.9 Optimization with ResultSet

The ResultSet interface represents data that contains the results of executing an SQL Query, and it provides a number of methods and constants to work with that data. It also provides methods to fine tune retrieval of data to improve performance. The following fine tuning tips can help you improve performance by using the ResultSet interface:

- Do batch retrieval using ResultSet.
- Use proper get methods.
- Use getXXX(int) instead of getXXX(String).
- Avoid getObject calls for Java primitive types.

Do batch retrieval using ResultSet

ResultSet interface also provides a batch retrieval facility like Statement as mentioned in "Do batch retrieval using Statement" on page 341. It overrides the Statement behavior. Initially you can get the default value of fetch size using ResultSet.getFetchSize() and sets the size as required. See Example 6-19.

Example 6-19 setFetchSize usage

```
ResultSet.setFetchSize(50);
```

This feature improves performance significantly when you retrieve a large number of rows, such as search functionality.

Use proper get methods

The JDBC API defines that each getxxx() method returns a matching Java object (for example, getString() returns a String object). The processing cost of each getxxx() method is mainly determined by the cost of the object's constructor call. Returning values of Java native data types, like an integer, is much cheaper than returning complex objects, like a Timestamp object.

Based on this information, the database can be designed for high performance. You should avoid retrieving "expensive" data types if you do not require them. Just performing a "SELECT *" from a table, and later sorting out in the application which fields you want to use, is too much luxury for a Java application. You should only select the fields you really use in your query. Also, never make it a default in your application to include TIMESTAMP and DATESTAMP in your queries; again, only select them if you really require them. A glance at Figure 6-6 shows the relative cost of all getxxx() methods. Retrieving a Date column is about 21 times more expensive than retrieving a short column.

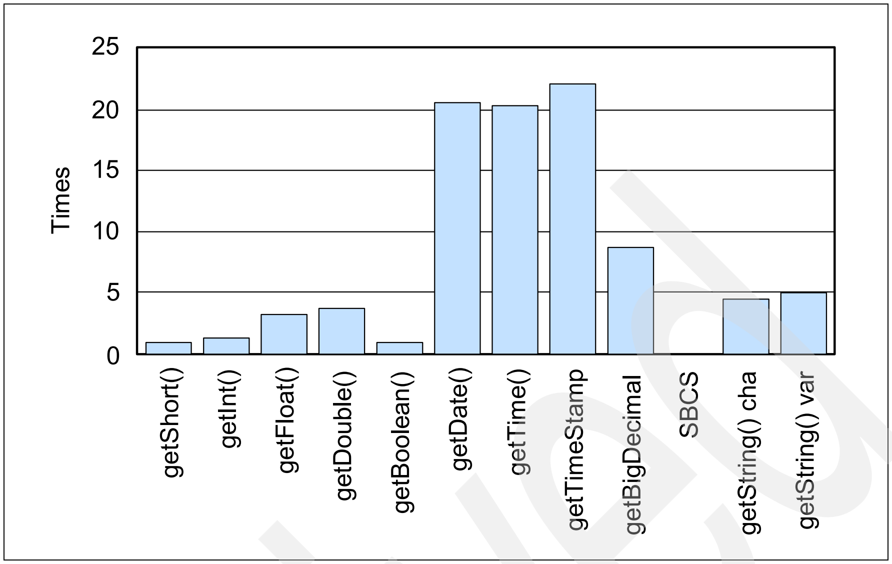


Figure 6-6 Relative cost of getXXX() processing

The JDBC API allows you to use different getxxx() methods to retrieve a database column, and using a non-matching getxxx() method is syntactically correct, but it causes a performance overhead per column. The overhead heavily depends on the DB2 data type, for example.

Figure 6-7 shows the overhead retrieving a column of a certain data type using the getString() method instead of the matching getxxx() method in percents. The overhead you get when a getString() method is used to receive a SMALLINT instead of the matching getShort() is about 380%.

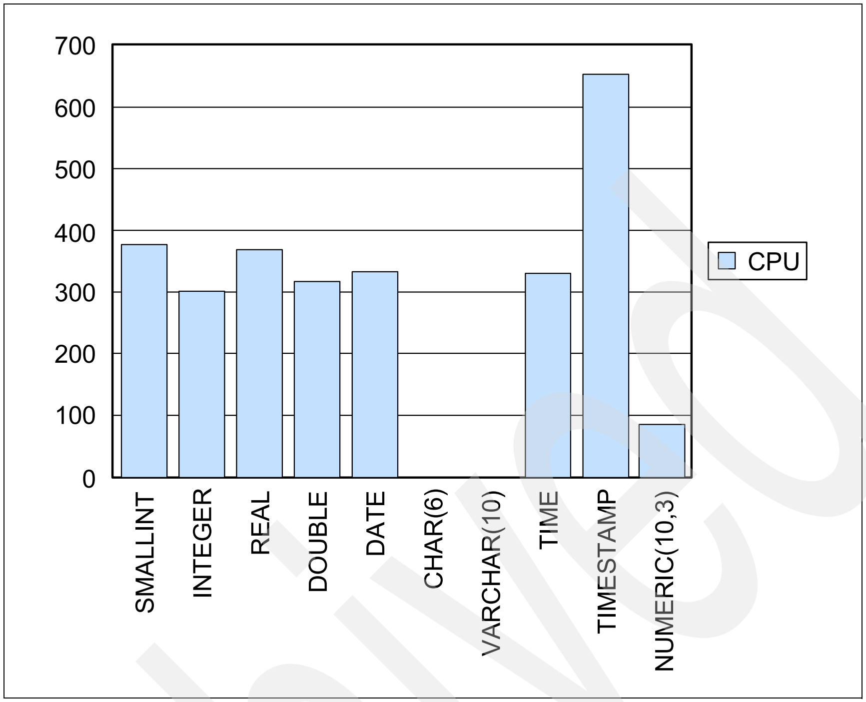


Figure 6-7 Comparing getString() when used to get all types

The best practice here is to use getXXX() according to the requested type.

Use getXXX(int) instead of getXXX(String)

Use the ResultSet getXXX methods that take numeric values instead of the versions that take column names. While the freedom to use your column names instead of numeric constants seems like an advantage, the database itself only knows how to deal with column indexes. Therefore, each getXXX method with a column name you call must be resolved by the JDBC driver before it can be passed to the database. Because getXXX methods are typically called inside loops that could be processed millions of times, this little bit of overhead can rapidly accumulate.

Avoid getObject calls for Java primitive types

When getting values from the database of primitive types (ints, longs, floats, and so on), it is far faster to use the get method specific to the primitive type (getInt, getLong, getFloat) than to use getObject. The getObject call does the work of the get for the primitive type, and then creates an object to return to you. This is typically done in loops, potentially creating millions of objects with short lifespans.

Using getObject for primitive commands has the added drawback of frequently activating the garbage collector, further degrading performance.

6.4.10 Optimization with SQL Query

This is an area where programmers generally make an incorrect choice. We provide the following tips to help you get better performance.

Avoid SELECT * SQL queries

SELECT * FROM... is a common way to state a query in SQL. Often, however, you do not have to query all the fields. For each column that is to be returned, the JDBC driver must do the additional work of binding and returning the row. Even if your application never uses a particular column, the JDBC driver has to be made aware of it and has to reserve space for its use. If your tables have few columns that are not used, this is not a significant overhead. For a large number of unused columns, however, the overhead can be significant. A better solution is to list the columns that your application is interested in individually, as shown in Example 6-20.

Example 6-20 Select statement best practice

```
SELECT COL1, COL2, COL3 FROM...
```

Cache the read-only data or updated occasionally

There are tables on most database schemas that are read-only or have updates occasionally. For read-only tables, writing operations occurred when there is a maintenance on the system, in the system runtime execution. There are no writing operations in these tables.

If your application reads data for these tables, the best practice is to read the table information only once and cache. This solution improves performance significantly.

For more information, go to this URL:

http://www.javaworld.com/javaworld/jw-07-2001/jw-0720-cache.html

For read-only data, this approach works fine. If you have some data that occasionally is updated in runtime execution, you can add a schedule do a refresh between specific configured times.

6.4.11 Fetch small amounts of data iteratively

If your application has to fetch large amounts of data, and if you use a single search, the response time can be very slow or not viable. The solution for this problem is to use a paging approach.

In a paging approach, you return a specific data amount of data to the client in a page. For example, instead of returning 1000 results directly to the client, return 10 iterations of 100 results and add a Next button for the next 100 and cache the search data at the server-side.

However, you can use stored procedures to return data iteratively instead of through server-side caching. In this case, the server-side application uses stored procedures to return small amounts of data iteratively.

This second approach gives better performance because is not necessary to keep data for paging in-memory but the first result can be used if the data volume is not very large.

6.5 Entity Beans 2.x

The goal of this section is to explore some best practices to get improved performance from existing code using Entity Beans. For new code development, consider the following sections:

- JPA as described in 6.6, “Java Persistence API: Entity Beans 3.0” on page 380
- iBATIS as described in 6.7, “iBATIS” on page 422

Entity Beans are server-side components that represent business objects stored in a persistent storage mechanism. This means, they provide an object view of transactional data in an underlying datastore that can be accessed from multiple, either local or remote, clients.

There are two different types of Entity Beans available:

- Container Managed Persistence (CMP) Entity Beans:

  A Container Managed Persistence CMP) bean is an Entity Bean for which the container handles the interactions between the enterprise bean and the data source. The container is responsible for synchronization of instance fields with the persistent store. When you develop a Container Managed Persistence bean, the application is insulated from the details of the persistence mechanism.

- Bean Managed Persistence (BMP) Entity Beans:

  A Bean Managed Persistence (BMP) Entity Bean is simply an entity EJB where the developer manually implements the service methods to manage persistence, most notably `ejbLoad()` to load the persistent state from the backing store and `ejbStore()` to store it.

6.5.1 Container Managed versus Bean Managed Persistence

The decision to use Container Managed Persistence or Bean Managed Persistence Entity Beans is usually easy. Whenever possible, you should prefer CMP to BMP, for the following reasons:

- Portability and flexibility:

  BMP Entity Beans contain hard-coded SQL statements that require not only a specific database layout, but can also be dependent on the database vendor. These hand-optimized SQL statements are difficult to port to another relational database or database layout. CMP Entity Beans, on the other hand, use an abstract persistence schema to specify the CMP and CMR fields in the deployment descriptor. These fields are mapped to the relational database fields during deployment. The deployment tool then generates the database vendor-specific classes. These steps ensure a high degree of flexibility and portability regardless of the used relational database schema or vendor.

- Faster development:

  If CMP is used, nearly all bean code can be automatically generated using actual development tools, such as IBM Rational Application Developer V6. Therefore, developers can concentrate on writing the business logic and assign the persistence and relationship management logic to the deployment tool and the EJB container. If BMP is used, the developer is responsible for loading and persisting the bean data itself.

- Performance:

  To write high-performance and scalable BMP Entity Beans, bean developers — who are usually responsible for writing the business logic — must be highly skilled in database programming as well. Unfortunately this is usually not the case — this is the domain of database administrators, not of bean developers. Another problem is that a higher level of optimization in the bean class automatically increases the difficulty to port the bean to another database. With CMP Entity Beans, the deployment tool can generate highly optimized code for every specific data source. Usually the performance of CMP Entity Beans is by far better than corresponding BMP Entity Beans. For further performance increase, IBM WebSphere Application Server V6 can also generate deployed code that uses SQLJ stored procedures to access IBM DB2 UDB.

- Automatic relationship management:

  Bean developers can define relationships to other CMP beans in the CMP Entity Beans deployment descriptor. This includes cardinality, navigation, and cascade delete which are automatically generated and maintained by the EJB container. When BMP Entity Beans are used, relationship management, including integrity checks and navigation, has to be implemented by the bean developer.

Although CMP Entity Beans have overwhelming advantages over BMP Entity Beans, there are also some drawbacks:

- Persistence limited to relational databases:

  Even though there are some approaches to extend the persistence mechanism to other than relational data sources, they are not yet available and therefore the usage of CMP Entity Beans is currently limited to relational databases. With BMP Entity Beans, access to non-relational data sources, for example, using a JCA adapter, is possible.

- Reduced control over database access:

  With BMP, the developer has complete control over the data access logic and the persistence management of the Entity Bean. Therefore, the developer has the ability to call vendor-specific functions or perform complex database joins to realize huge persistence schemas. In some rare cases, this high level of data access control might be required.

6.5.2 Considerations on the techniques used

The techniques used in the following sections refer to tuning your Entity Bean for better performance in three ways:

- In design and code: Coding and design tips to gain performance are given.
- In deployment descriptor: The developer or application assembly can set up this configuration using for example Rational Application Developer or WebSphere Application Server toolkit. In this case the setup is related to a specific Entity Bean in an EAR (Enterprise Archive). Note: A deployment descriptor contains specific parameters to an EJB or groups of EJBs that belong to the same EAR.
- In an Application Server: In this case the tuning is more generic and relates to an EJB Container that can run different EJB projects and is inside an Application Server.

In the following sections, for examples of techniques to gain performance in Entity Beans, we use Application Server Toolkit 6.1 because there are a lot of cases where we have to change the deployment descriptor. This approach changes the container service functionality of the Enterprise Bean. But if you have Rational Application Developer, you can accomplish the same results.

6.5.3 General tips for EJB Entity Beans performance

While Entity Beans can reduce the amount of coding work required to access persisted data, care must be taken with the use of Entity Beans to avoid performance problems.

Access Entity Beans from session beans

Avoid accessing EJB Entity Beans from client or servlet code. Instead, wrap and access EJB Entity Beans in EJB session beans. This satisfies two performance concerns:

- When the client application accesses the Entity Bean directly, each getter method is a remote call. A wrapping session bean can access the Entity Bean locally and collect the data in a structure, which it returns by value.
- An Entity Bean synchronizes its state with its underlying data store at the completion of each transaction. When the client application accesses the Entity Bean directly, each getter method becomes a complete transaction. A store and a load follow each method. When the session bean wraps the Entity Bean to provide an outer transaction context, the Entity Bean synchronizes its state when the outer session bean reaches a transaction boundary.

Avoid extremely fine-grained EJB models

Although local interfaces, introduced in EJB 2.0, make more fine-grained EJB models possible, take care that you do not carry the granularity to excess. Think of using dependent value classes as an alternative. They can increase performance, because no separate call and no EJB relationship is required.

Reduce method calls with coarse granularity

The best practice here is to use a Value Object to do remote calls instead of individual calls on each data object increasing the remote traffic, as shown in Example 6-21.

Example 6-21 Using remote calls for each data access

```
remoteObject.getFirstName();
remoteObject.getLastName();
remoteObject.getCity();
remoteObject.getState();
remoteObject.getZipCode();
remoteObject.getCrediCardNumber();
```

In Example 6-21, there are six network calls from the client to the remote object. You can minimize these network calls by reducing the number of method calls using a Value Object to get all this information required in one remote call. See Example 6-22.

Example 6-22 Using a Value Object to reduce remote calls

```
// create an object and fill that object locally
CustomerInfo customer = new CustomerInfo();
customer.setFirstName("firstname");
customer.setLastName("lastname");
customer.setCity("Austin");
customer.setState("TX");
customer.setZipCode("78749");
customer.setCrediCardNumber("287459754974");
remoteObject.getPersonInfo(person); // send object through network
```

In this example, there is only one network call instead of six network calls.

Use the coarse grained approach to minimize the number of network calls and improving performance for Entity Beans.

Control serialization in remote EJBs

If you use an Object to pass data from a client to an EJB, be careful when you use Value Objects. All attributes of Value Objects have to be Serializable because they are sent over the network. In addition, if you do not have to use some attribute to send to an EJB, make this attribute transient.

Do not use entity EJBs to read large amounts of data

Entity EJBs are best used for manipulating small amounts of data. Returning large result sets from (default) EJB finders can be inefficient and it is often better to use JDBC directly from a session bean for this.

Beware of significant use of EJB inheritance

In cases where Entity Beans significantly use ejb-inheritance, care must be taken to ensure that performance is adequate. If performance problems are encountered, they can potentially be addressed by reducing the use of inheritance in the model or by use of Bean Managed Persistence (BMP) in preference to Container Managed Persistence (CMP). Another strategy is to avoid turning each entity (table) into a single EJB— in some cases two or more related entities can be represented by a single EJB.

Use local interfaces

If the EJB client is located in the same Java Virtual Machine as the EJB, you can take advantage of the local interface. No network tasks are necessary and the bean parameters are passed by reference. This increases performance. So if you intend to deploy your EJB clients in the same JVM where the EJB itself is deployed, you should use the local interface. Usually local interfaces are used to access entity EJBs from the Facade.

Use caching options from EJB Container

To improve performance, some caching techniques can be used. Selection of the appropriate option requires an understanding of how the Entity Beans are to be used, as there is a trade-off between minimizing database reads and supporting Workload Management (WLM). For more details, see 6.5.7, “Understanding caching options to improve performance” on page 358.

Optimize EJB transaction and isolation level settings

In EJBs we have declarative transaction management. So the developer does not have to take care of all the resources required in the transaction, the container does. Nevertheless the definition of the transaction attributes in the deployment descriptor is an important task that can change application behavior and performance dramatically.

WebSphere Application Server V6 includes a feature called application profiling that allows you to optimize transaction and isolation level settings. This feature allows you to dynamically adjust the access intent settings to the actual runtime requirements of the work currently performed. Application profiling enables you to configure multiple access intent policies on the same Entity Bean. It reflects the fact that different units of work have different use patterns for enlisted entities and can require different kinds of support from the server runtime environment.

For more information about access intent settings, see 6.5.12, “Understanding EJB access intents for best practices” on page 369.

Getting remote nested or root cause exception in EJB Clients

If you are using a remote EJB client, a good practice is get the nestable or root cause of an exception. Using this approach, you can correct logic in your client. The following URI shows how to set up this feature in WebSphere Application Server 6.1.

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/tejb_rollback.html

6.5.4 Developing a read-only Entity Bean

Following is a usage scenario and example for writing an EJB application that uses a read-only Entity Bean.

Usage scenario

A customer has a database of catalog pricing and shipping rate information that is updated daily no later than 10:00 PM local time (22:00 in 24-hour format). They want to write an EJB application that has read-only access to this data. That is, this application never updates the pricing database. Updating is done through some other application.

Example 6-23 Customer Entity Bean local interface

```
public interface ItemCatalogData extends EJBLocalObject {

   public int getItemPrice();

   public int getShippingCost(int destinationCode);
}
```

Applying this technique

The code in the stateless SessionBean method (assume that it is a TxRequired) invokes this EntityBean to figure out the total cost including shipping. This would look like the code shown in Example 6-24.

Example 6-24 SessionBean code to access read-only EntityBean

```
// .....
// Some transactional steps occur prior to this point, such as removing
// the item from
// inventory, etc.
// Now obtain the price of this item and start to calculate the total
cost to the purchaser

ItemCatalogData theItemData =
     (ItemCatalogData)
ItemCatalogDataHome.findByPrimaryKey(theCatalogNumber);

int totalcost = theItemData.getItemPrice();

// ... some other processing, etc. in the interim
// ...
// ...
// Add the shipping costs
```

```
totalcost = totalcost +
theItemData.getShippingCost(theDestinationPostalCode);
```

At application assembly time, the customer sets the EJB caching parameters for this bean as follows:

- ActivateAt = ONCE
- LoadAt = DAILY
- ReloadInterval = 2200

On the first call to the getItemPrice() method after 22:00 each night, the EJB container reloads the pricing information from the database. If the clock strikes 22:00 between the call to getItemPrice() and getShippingCost(), the getShippingCost() method still returns the value it had prior to any changes to the database that might have occurred at 22:00, since the first method invocation in this transaction occurred prior to 22:00. Thus, the item price and shipping cost used remain in sync with each other.

6.5.5 EJB QL enhancements from EJB 2.1 specification

The new specification extends the existing query to make it consistent SQL-like. Support for the following aggregate function has been added:

- AVG, MIN, MAX, SUM, COUNT

Support ordering at the database level, ORDER BY clause, has been added to the query language. The new EJB 2.1 specification supports an additional numeric function, MOD. e EJB QL examples with new features are shown in Example 6-25, Example 6-26, and Example 6-27.

Example 6-25 Return the maximum salary among the employees of the company

```
SELECT MAX(e.salary) FROM EMPLOYEE AS e
```

Example 6-26 Return all employees with even numbered ids

```
SELECT OBJECT(e) from EMPLOYEE AS e WHERE MOD(e.id,2)=0
```

Example 6-27 Return employee records sorted by asc order of employee's last name

```
SELECT OBJECT(e) FROM EMPLOYEE AS e ORDER BY e.lastName
```

6.5.6 Tuning the Entity Bean container pools

In this section we discuss tuning the EJB container for Entity Bean behavior at a high level. In an EJB container, you have two types of pools that belong to the life cycle of a Entity Bean:

- Instance Pool: When the bean is not associated to data in a database.
- Instance Cache Pool: When the bean is in a ready state to be used. It means that the bean has data associated with a database.

See Figure 6-8 to help visualize these pools in an Entity Bean life cycle.

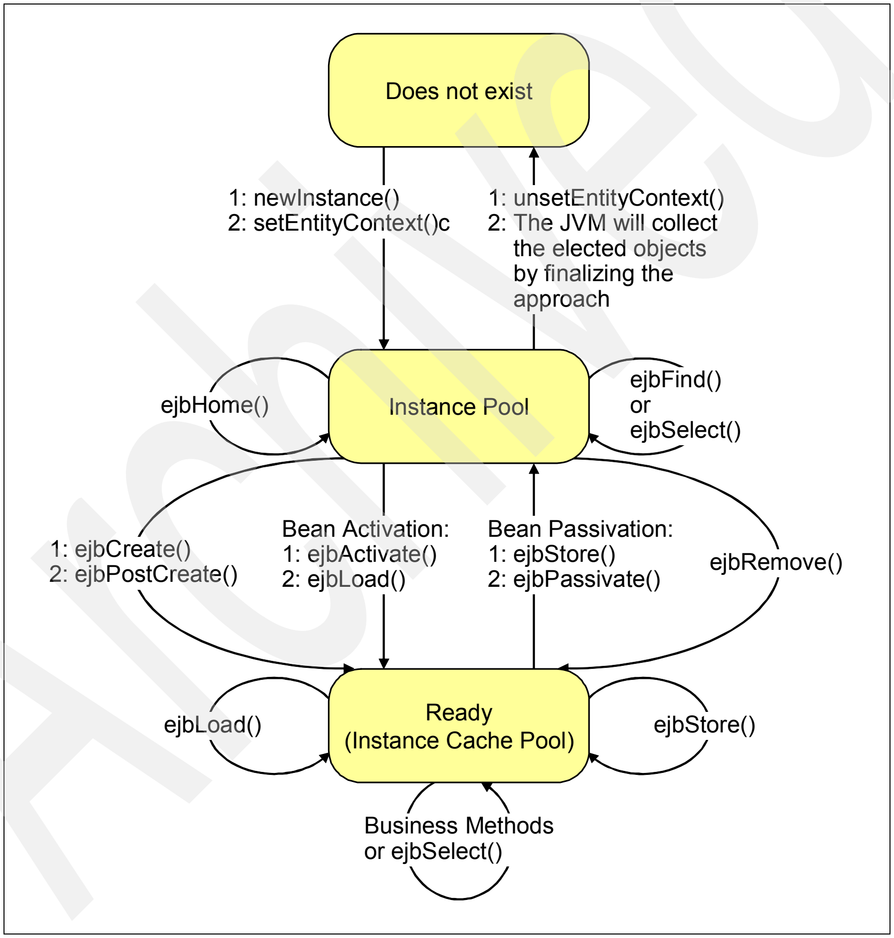


Figure 6-8 Entity Beans life cycle

Notice in Figure 6-8 that the first state of an Entity Bean is *does not exist*. If you set up your minimum pool size to 25, when the application server starts up, it executes 25 beans using class.newInstance() and setEntityContext() method of each bean. If you set the maximum pool size to 50 and this number is reached, the next clients must wait for an Entity Bean be delivered to this pool from an Instance cache pool when a bean is released. As the number of clients decreases, the EJB container can deliver resources until minimum pool size is reached using unsetEntityContext() and object's finalize().

The next step is when a client calls a create() method. The EJB container counter calls the corresponding ejbCreate() method on one of the beans in the instance pool and creates a row in the database and populates the values to the variables and puts it in the instance cache after returning primary key. At this stage, an EJBObject is assigned to the client that communicates to the bean in the instance cache pool. Next, the container calls the ejbPostCreate() method. At this stage, the bean is moved from the pool to the cache and is ready to serve the client's business methods.

Next, when the client calls the business method, the container calls ejbLoad() that updates the bean's state, executes the business method, and calls the ejbStore() method to store the data in the database. Note that depending on configuration of your bean, these steps are not executed by the container if a Bean has exclusive access to database (a commit option type A).

If the concurrent active clients are more than the cache size, then the container passivates a bean and calls ejbStore(), ejbPassivate() methods and puts it back in the instance pool. If the idle client calls again some time later, the container calls ejbLoad() to get the latest data, and calls ejbActivate() method and puts it in the instance cache.

Next, if the client calls the remove() method, the container calls the ejbRemove() method that removes the data from the database and puts the bean back in the instance pool from instance cache pool.

Using the foregoing summary, we can understand that:

- The client is responsible to control the life cycle of a bean — this means the creation of data in database and removing data from database using ejbCreate() and ejbRemove() methods.
- The container controls the life cycle in the pool and cache and also activation and passivation processes in the cache.
- Both client and container are responsible to control ejbLoad() and ejbStore() methods, depending upon client's method calls and container activation and passivation process.

- The overall life cycle depends upon the client's concurrent operations, instance pool size, and instance cache size.

Next, we discuss tuning pools of Entity Beans.

Tuning instance pool size

As shown in the Entity Bean life cycle in Figure 6-8 on page 355, we understand that we can control creation and destruction of beans by describing pool size (min and max). This configuration is vendor specific. If this size is less (if your default size is less or you configure a smaller size) then the container has to put the clients in the queue when the number of concurrent clients accessing (create/finder/home methods) are more than the max pool size. Also, instance cache depends upon the instance pool because the instance cache has to get beans from the instance pool. So if the pool size is small, it degrades the performance and clients take more time to execute.

For best performance here, you must set the maximum beans in the pool equal to the number of maximum concurrent client accesses (create/finder/home methods), so that it reduces creation and destruction of beans.
To configure instance pool size, see 7.2.4, "Tuning the EJB container" on page 528.

Tuning instance cache size

As shown in the Entity Bean life cycle Figure 6-8 on page 355, we understand that we can control activation and passivation indirectly by describing the instance cache size in the EJB container setup of an application server. See 7.2.4, "Tuning the EJB container" on page 528 for a discussion on tuning the instance cache size.

Activation and passivation are expensive. It depends on your commit option schema. In the worst case, for every activation, the container calls ejbLoad() to get the latest data from the database and calls the ejbActivate() method. For passivation, in the worst case, the container calls ejbStore() to store data in the database and calls the ejbPassivate() method. The ejbLoad() and ejbStore() methods communicate with the database to synchronize the latest data. If the concurrent active clients (when the client calls business methods) are more than the instance cache size, then activation and passivation occur often, thus affecting performance. In order to increase performance, configure optimal cache sizes. Cache size must be equal to concurrent active clients accessing the bean.

The important thing here is that the instance cache size and pool size for Entity Beans are larger than session beans. The beans in the pool and cache should accommodate the Entity Beans requirements, like finder methods that return a large number of records and populate the data in the beans. So be careful when you configure Entity Bean pool size and cache size. If you are not sure about what the exact parameters are, use the default pool size and cache size.

Use setEntityContext() method as cache

The setEntityContext() method is called only once in a bean's life time. Because Entity Beans in the pool are reused by a number of other clients, you can cache any bean specific resources like Entity home references and DataSource references in this method. You have to declare those resources as instance variables and acquire them in this method. These resources are specific to a bean but not available globally. For global reuse, it is better to use the technique related to Cache EJBHome object references.

Also, you can use this approach to acquire other resources. Remember that you should not acquire physical resources like database connections in these methods. If there are more concurrent clients and the pool size is greater, it is better to acquire such resources in each method and release them in that method only. Use setEntityContext() method to cache bean specific resources that are required by other clients as well.

6.5.7 Understanding caching options to improve performance

The EJB container has three types of caching that can be performed for Entity Beans between transactions. These three commit options affect performance and behavior of a Entity Bean life cycle. According to the option, you can increase or decrease your Entity Bean performance.

Commit option A

The bean stays ready and attached to the EJBObject. It means that the entity instance is still associated to the client. Also the Container keeps the Entity locked and nobody out of Entity Bean can change the Bean state. When a business method is invoked the Container does not do ejbAcivate() and ejbLoad() in the activation process because the data is synchronized all the time while bean is active.

Commit option B

In this case the bean stays attached to the EJBObject and loaded with data but the bean is marked as invalid. It means that the Container knows that the bean can be stale between method invocations, for example, to be available to another client. Then the Container does an ejbLoad() for each business method invoked from the client but does not execute ejbActivate() in the activation process.

Commit option C

After a method is called, the bean is passivated at the end of the transaction and goes back to the instance pool. The next time that a client calls a business method, a bean comes out of the pool, is activated and loaded, and the business method is passed to the bean.

Applying this technique

To apply this technique, we use Rational Application Developer V7. From this starting point, perform the following steps.

1. Open your EJB Project in a J2EE perspective.
2. Open deployment descriptor of project (ejb-jar.xml).
3. Go to the Bean view of the deployment descriptor.
4. Inside the Bean view, look at the Bean Cache section as shown in Figure 6-9.

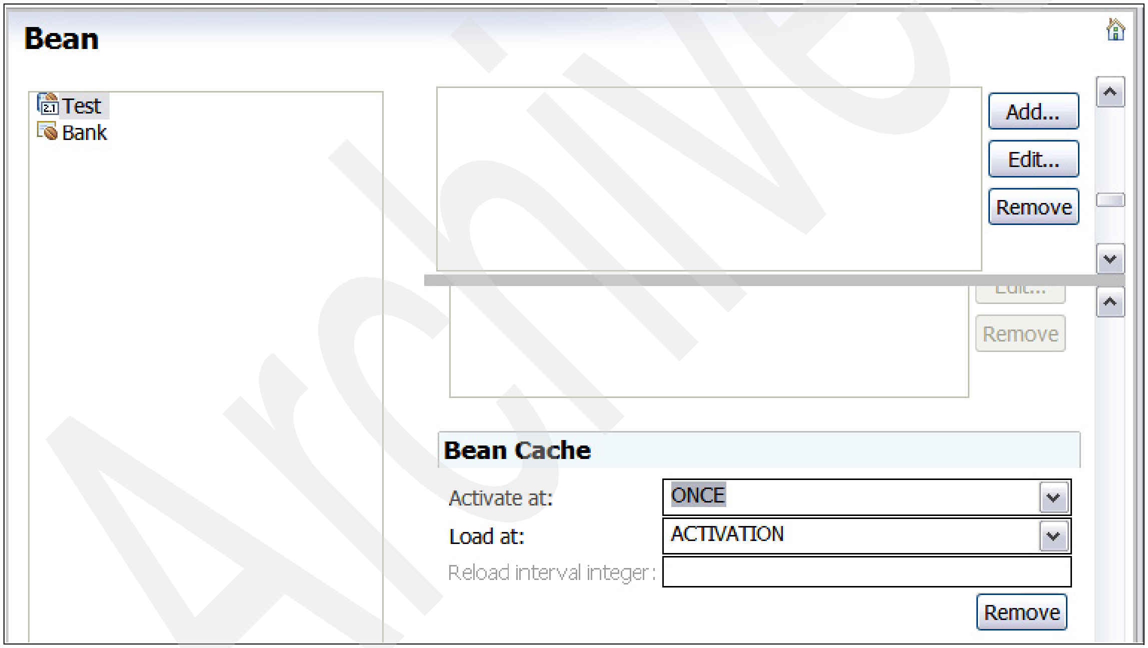


Figure 6-9 Setting up Entity Bean using WebSphere extensions

Now we show you the relation between theory and practice. If you want to use:

- Commit option A: You set **Activate at: ONCE**. That means that ejbActivate() occurs once between transactions and **Load at: ACTIVATION** for means that the ejbLoad() is called once when bean is activated.

- Commit option B: You set **Activate at: ONCE**. That means that ejbActivate() occurs once between transactions and **Load at: TRANSACTION** for means that the ejbLoad() is called between transactions.
- Commit option C: You set **Activate at: TRANSACTION** and **Load at: TRANSACTION**. This configuration means that whole activation process is used for each transaction unit.

You can see other options in the **Load at:** parameter, such as:

- INTERVAL: Performs an ejbLoad from a specific interval defined in **Reload Interval Integer** field.
- DAILY: Creates ejbCache on a daily basis.
- WEEKLY: Creates ejbCache on a weekly basis.

These three last parameters should be used if you know the update frequency.

6.5.8 Defining data cache settings for a bean for performance

Lifetime in cache usage settings are WebSphere Application Server extensions to the Enterprise JavaBeans 2.x specification. For more information, see the WebSphere Application Server documentation. Lifetime in cache settings can provide a way for you to improve performance for beans that are only occasionally updated.

Applying this technique

To define data cache settings (this option is only available to 2.x CMP Entity Beans):

1. Using Rational Application Developer, switch to the J2EE perspective.
2. In the Project Explorer view, select the desired EJB module.
3. Right-click on the Deployment Descriptor Editor and select Open With from the pop-up menu.
4. On the Beans page of the editor, go to the Data Cache section.
5. In the **Lifetime in cache** field, specify the lifetime, in seconds, of cached data for an instance of this bean type. This value indicates how long the cached data is to exist beyond the end of the transaction in which the data was retrieved.
6. In the **Lifetime in cache usage** field, select one of the following values to indicate how the lifetime-in-cache setting is to be used by the caching mechanism:
   - OFF: When this value is used, the value of Lifetime in cache is ignored.

- ELAPSED_TIME: When this value is used, the value of Lifetime in cache is added to the time at which the transaction in which the bean instance was retrieved is completed. The resulting value becomes the time at which the cached data expires.
- CLOCK_TIME: When this value is used, the value of Lifetime in cache represents a particular time of day.
- WEEK_TIME: Usage of this value is the same as for CLOCK_TIME except that the value of Lifetime in cache can represent more than 24 hours, but not more than 7 days.

The use of a value other than OFF requires that finders on the enterprise bean have an access type of Read (wsPessimissticRead or wsOptimisticRead) because EJB applications are not permitted to update such CMP beans.

See Figure 6-10 for this configuration.

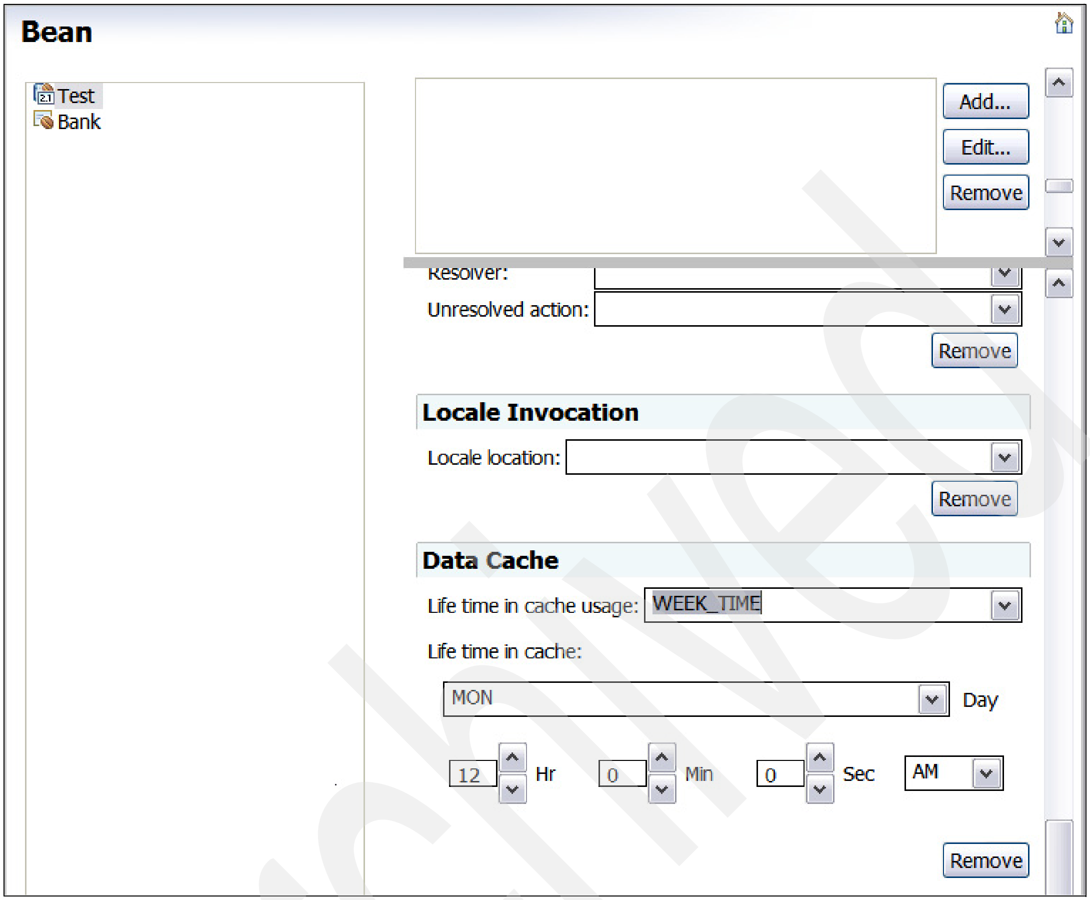


Figure 6-10 Data Cache configuration in EJB deployment descriptor

For additional information on data cache settings, see the WebSphere Application Server documentation.

6.5.9 Improving passivation behavior

In the section 6.5.6, “Tuning the Entity Bean container pools” on page 355 we gave an overview about the Entity Bean life cycle. There we showed that the passivation process of an Entity Bean concerns two methods: ejbStore(), which refreshes the data in the database; and ejbPassivate(), which sends the bean from the ready pool to the instance pool. WebSphere Application Server has some configuration options to improve performance in this process.

Applying this technique

To apply this technique we use Rational Application Developer version 7. From this starting point, perform the following steps.

1. Open your EJB Project in a J2EE perspective.
2. Open deployment descriptor of project (ejb-jar.xml).
3. Go to Bean view of deployment descriptor.
4. Inside Bean view, look at the WebSphere extension section. See Figure 6-11.

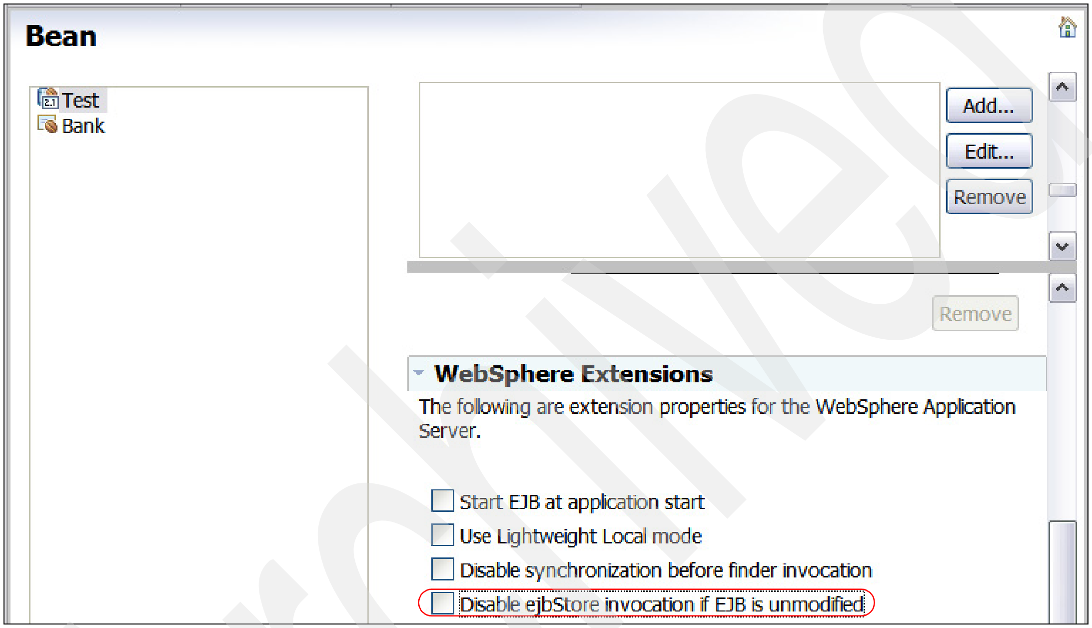


Figure 6-11 Setting up Entity Bean using WebSphere extensions

The parameter highlighted in Figure 6-11 is a WebSphere extension. It detects that if the data is not modified, the container does not execute ejbStore() that belongs to the passivation process.

6.5.10 Applying lightweight local model to an Entity Bean

WebSphere Application Server provides a special operational mode called lightweight local mode, which can improve the performance of Entity Bean methods. You can decide which Entity Beans in your application to run in this mode.

In lightweight local mode, the container streamlines the processing that it performs before and after every method on the local home interface and local business interface of the bean. This streamlining can result in improved performance when Entity Bean operations are called locally from within an application. Because some processing is skipped when running in lightweight local mode, this mode can be used in certain scenarios only.

Lightweight local mode is patterned somewhat after the Plain Old Java Object (POJO) entity model introduced in the Enterprise JavaBeans (EJB) 3.0 specification. Using lightweight local mode, you can obtain some of the performance advantages of the POJO entity model without having to convert your existing EJB 2.x application code to the new POJO model. You can apply lightweight local mode to both Container Managed Persistence (CMP) and Bean Managed Persistence (BMP) entity types that meet the specific criteria.

When to use the lightweight local mode

Lightweight local mode is designed for Entity Beans that are created, found, and called using the Session Facade pattern. Under this pattern, Entity Bean local home and local business methods are called from within methods of a stateless session bean or stateful session bean. The session bean methods, which can be called remotely or locally, provide security control and transaction demarcation for the Entity Beans that are accessed by the session bean.

You can apply lightweight local mode only to an Entity Bean that meets the following criteria:

- The bean implements an EJB local interface.
- No security authorization is defined on the Entity Bean local home or local business interface methods.
- No run-as security attribute is defined on the local home or local business methods.
- The classes for the calling bean and the called Entity Bean are loaded by the same Java class loader.
- The Entity Bean methods do not call the WebSphere Application Server-specific Internationalization Service or Work Area Service.

The first criterion prevents CMP 1.x beans from supporting lightweight local mode, because the 1.x beans cannot have local interfaces.

In addition, lightweight local mode provides its fullest performance benefits only to Entity Bean methods that do not have to start a global transaction. This condition is true if you ensure that your Entity Bean also meets the following criteria:

- A global transaction is already in effect when the Entity Bean home or business method is called. Typically, this transaction is started by the calling session bean.
- The local business interface methods and the local home methods of the Entity Bean use the following transaction attributes only: REQUIRED, SUPPORTS, or MANDATORY.

If an Entity Bean method that is running in lightweight local mode must start a global transaction, the bean still functions normally but only a partial performance benefit is realized.

You can mark an Entity Bean that defines a remote interface or a TimedObject interface, in addition to the local interface, for lightweight local mode. However, the performance benefit is apparent only when the bean is called through its local interface.

Applying this technique

You can decide which Entity Beans in your application to run in this mode.

You can apply lightweight local mode to specific EntityBean types within your application in two ways. You can use application server tooling, or the Marker interface technique.

Using Application Server Tooling procedure

1. Start the Application Server Toolkit. See “Starting WebSphere Application Server Toolkit” on page 642.
2. Select the EJB deployment descriptor of the Entity Bean that you want to work with.
3. In the property pane, select the WebSphere Extension tab.
4. Check the box labeled Use Lightweight Local mode.
5. Select OK.
6. Save your changes.

Marker interface technique

Use the marker interface technique when a group of beans within the application is related through a common inheritance hierarchy, and all the beans in the hierarchy are to be marked. For an application with a large number of beans in a hierarchy, this technique is the most efficient.

To use a marker interface, code your bean implementation class to implement the com.ibm.websphere.ejbcontainer.LightweightLocal interface. The bean implementation class does not have to directly implement the interface; any parent class or interface can also implement it. For details, see the com.ibm.websphere.ejbcontainer package in the API documentation section of the information center.

6.5.11 Using partial column updates for CMP beans

Previously, the WebSphere Application Server implementation of the Container Managed Persistence (CMP) bean method ejbStore always stored all of the persistent attributes of the CMP bean to the database, even if only a subset of persistent attribute fields were changed. This unnecessary performance degradation is eliminated in this release of the product.

For Enterprise JavaBeans (EJB) 2.x CMP Entity Beans, you can use the partial update feature to specify how you want to update the persistent attributes of the CMP bean to the database. This feature is provided as a bean level persistence option, called PartialOperation, in the access intent policy configured for the bean. PartialOperation has two possible values:

- NONE

  Partial update is turned off. All of the persistent attributes of the CMP bean are stored to the database. This is the default value.

- UPDATE_ONLY

  This specifies that updates to the database occur only for the persistent attributes of the CMP bean that are changed.

Effects on performance

Performing partial updates increases performance in several ways:

- By reducing query execution time, since only a subset of the columns are in the query. Improvement is higher for tables with many columns and indexes. When the table has many indexes only the indexes affected by the updated columns have to be updated by the back-end database.
- By reducing network I/O because there is less data to be transmitted.
- By saving any processing time for non-trivially mapped columns (if a column uses converters/composers/transformations), by partially injecting the input record.
- By eliminating unnecessary firing of update triggers. If a CMP bean field is not changed, any trigger depending only on the corresponding column is not fired.

Although partial update improves performance in general, it can adversely affect performance too:

- If you enable partial update for a bean for which your application modifies several different combinations of columns during the same time span, then the prepared statement cache maximum for the connection is reached very quickly. As a result, statement handles are evicted from the cache based on least recent usage. This results in statements being prepared again and again, decreasing performance for all CMP functions (not just limited to ejbStore()).
- Partial update query templates cached in the function set increase memory use. The increase is linear relative to the number of fields in the CMP bean for which the partial update access intent option is turned on.
- The PartialOperation persistent option, when used in combination with the Batch Update persistent option, affects the performance of the batch update because now each partial query is different. There is an execution time cost incurred for generating a partial update query string dynamically. Since query fragments are stored for each column, the execution cost to assemble the query fragments is linear, based on the number of CMP bean fields dirtied.
- There are condition checks for each CMP field (for example, to inspect the dirty flags, to execute the preparedStatement setXXX() calls, and so on).

Considerations for using partial update

The performance gains you hope to achieve should be weighed against the possible instances where degradation can occur. You can use the following guidelines to help you make the decision:

- Partial update might not benefit an application that only involves a small table (few columns) with simple data types and no update triggers. The cost to assemble the partial query dynamically would probably outweigh the performance gain.
- Partial update is a benefit if there is a complex data type that is not updated often. An example of a complex data type might be an employee bean with a “photo” CMP attribute mapped to a BLOB OR VARGRAPHIC or similar complex back-end type that is typically stored in a different location in the database manager implementation.
- Partial Update might benefit if there are several VARCHAR type columns and only a very few of them are updated typically.
- It is better not to use the partial operation if the application can randomly be updating different combinations of columns and the number of assignable columns (non-key) is higher than five. This generates many different partial queries and fills up the prepared statement cache quickly. But if the bean does not have too many columns (four or less) and it has complex data types, then you might consider turning partial update on, with the option of

increasing the statement cache size to allow for the increased number of queries.

- Partial Update is beneficial when there are update triggers required on a subset of columns.
- Partial Update is beneficial when the table has many columns and indexes and only a few indexes are touched by a typical update.

Restrictions

By default, batch update of update queries is disabled for all CMP beans for which partial update is enabled. In other words, partial update takes precedence over batch update. Batch update of delete and insert queries is not affected.

Batch update performance is affected when both batch update and partial update persistence options are used on the same bean, because each partial query is different. To group the similar partial update queries into a batch update, you can use the JVM property -Dcom.ibm.ws.pm.grouppartialupdate=true.

Grouping of partial updates only helps when there are several partial queries with the same shape in a transaction. Otherwise, grouping partial updates has the opposite affect on performance. Because this setting is not on a bean level basis, you should be careful when turning it on. Because this affects all beans that have both partial update and batch update on, you must make sure that batch update of partial queries does indeed increase performance when viewed across all the beans for which both updates are on.

So you should determine which situation gives the best performance for your application: batch update only or partial update only or both (with grouppartialupdate flag set to true).

To set the JVM property:

1. Open the server.xml file.
2. Change the value of -Dcom.ibm.ws.pm.grouppartialupdate=true to -Dcom.ibm.ws.pm.grouppartialupdate=false.

Applying this technique

Follow these steps to apply the technique:

1. Start the Application Server Toolkit. See “Starting WebSphere Application Server Toolkit” on page 642.
2. Open the J2EE perspective to work with J2EE projects. Click **Window** → **Open Perspective** → **Other** → **J2EE**.
3. Open the Project Explorer view. Click **Window** → **Show View** → **Project Explorer**.

4. Open the EJB Deployment Descriptor. Click **EJB Projects** → **project** → **ejbModule** → **META-INF** → **ejb-jar.xml**. The Deployment Descriptor editor opens.
5. In the Deployment Descriptor editor, select the **Access** tab. The access page opens.
6. In the Default Access Intent for Entities 2.x (Bean Level) section of the access page, select the bean for which you want to set partial operation. If an access intent has already been configured for this bean, click the **Edit** button to edit the access intent policy. Otherwise, click the **Add** button to add an access intent policy to the bean. This opens the Add access intent window.
7. Select the Persistence Option check box if it is not already checked.
8. Select the Partial Operation check box. Use the drop-down list next to the Partial Operation check box to select your preference:
   - NONE

     Partial update is turned off. All of the persistent attributes of the CMP bean are stored to the database. This is the default value.
   - UPDATE_ONLY

     Specifies that updates to the database occur only for the persistent attributes of the CMP bean that are changed.
9. Select **Finish**.

6.5.12 Understanding EJB access intents for best practices

A Java 2 Enterprise Edition (J2EE) application server has a highly multi-threaded execution profile that allows multiple transactions to be active simultaneously. The persistent application data associated with these transactions is held in a Relational Database Management System (RDBMS). The application data held in the database is loaded into container managed Enterprise JavaBeans (EJBs) when required by the application within the scope of a transaction.

It is important that this data is correctly protected from the competing requirements of those transactions that require read access, and those that require update access. The EJB container, persistence manager, and the relational resource adapter and database work together to control the concurrency. Application data integrity is essential, and is ensured by the application server engine. WebSphere Application Server takes care of these low-level details so the application developers do not have to.

To support different database access methods, WebSphere Application Server defines an extension to the EJB deployment descriptor that enables developers to customize the concurrency control settings specific to their applications' transactional requirements. Different policies are implemented as declarative annotations that give WebSphere hints on how to access the data. These policies are referred to as Access Intents. WebSphere enables these to be configured manually and can provide performance benefits. Contention on transactional resources, such as a relational database in this case, can become a bottleneck. Tuning the way the application server accesses the persistent data could be used to improve throughput and response times, as well as lead to lower resource contention.

Attributes of an access intent policy

Access intents are a combination of database-specific isolation levels, locking policies and, in some cases, read-ahead hints. Each access intent policy defined is an aggregate of a set of properties used to control different aspects of the persistence and concurrency handling involved in dealing with EJB entities.

Database isolation level

A transaction processing system must support complete isolation of a transaction from other concurrently running transactions. Complete isolation, however, might have an impact on performance and throughput. If each transaction executes in a serial fashion, our multi-threaded application server would not perform very well.

The ANSI SQL-92 standard defines four levels of database isolation that we can use to gain some form of parallelism in our transactions. They are identified with respect to what particular data access characteristics they allow (for example, dirty reads, non-repeatable reads or phantom reads). The isolation level is set on the database connection and, as we see later, the access intent policy in use determines how the application server sets this parameter. Different database vendors can use different techniques to ensure that the requirements of isolation level are met.

The four different JDBC isolation levels are listed in the following sections, with a description of how DB2 handles each one. Other database vendors might implement this differently, but the general idea is similar.

TRANSACTION_SERIALIZABLE

This is the strongest isolation level. DB2's internal isolation level for this is referred to as Repeatable Read. All the rows in the database table that are affected by the current transaction are locked; no other transaction can insert, delete or update a row in the selection set. Locking is also performed so that rows cannot be inserted in a way that would change the selection set used in other transaction. For example, SELECT * FROM orders WHERE Total > 2000 would not only lock the rows that match the predicate condition, but also all rows in the table. This prevents the presence of phantom rows. Uncommitted changes from other transactions cannot be seen.

TRANSACTION_REPEATABLE_READ

Phantom rows can appear with this isolation level. This is because the statement SELECT * FROM orders WHERE Total > 2000 only locks the rows that meet the condition. If the same statement is reissued within the transaction, the result could be different. It is ensured that the rows updated or read during the transaction cannot be changed by anyone else until the work commits. Repeatable Read maps to DB2's isolation level Read Stability.

TRANSACTION_READ_COMMITTED

DB2 maps this to its Cursor Stability isolation level. If you issue the statement SELECT * FROM orders WHERE Total > 2000 rows are only locked as the resultset is traversed. Obviously, phantom rows can occur, but more importantly, non-repeatable reads can occur. This happens when a transaction reads the same data twice, while another transaction modifies the data between the two reads, resulting in a different result for each read.

TRANSACTION_READ_UNCOMMITED

Transactions are not isolated from each other, and can access uncommitted changes from each other. Currently, none of the access intents in WebSphere Application Server use this isolation level.

For more information about JDBC isolation levels, look at "Choose the best isolation level" on page 333.

Choosing the correct isolation level can be complex. However, each access intent policy in WebSphere Application Server chooses an appropriate isolation level for supported databases, relieving the developer from this level of detail.

Locking strategy

In conjunction with the isolation level, the application server can use two different locking strategies to further control the concurrency of transactions.

Pessimistic locking

A pessimistic locking strategy is one where a lock is obtained early in the transaction and kept until either the transaction is committed or rolled back. Other transactions wanting access to this data must wait until the lock is released. See Figure 6-12 for a better understand of pessimistic locking.

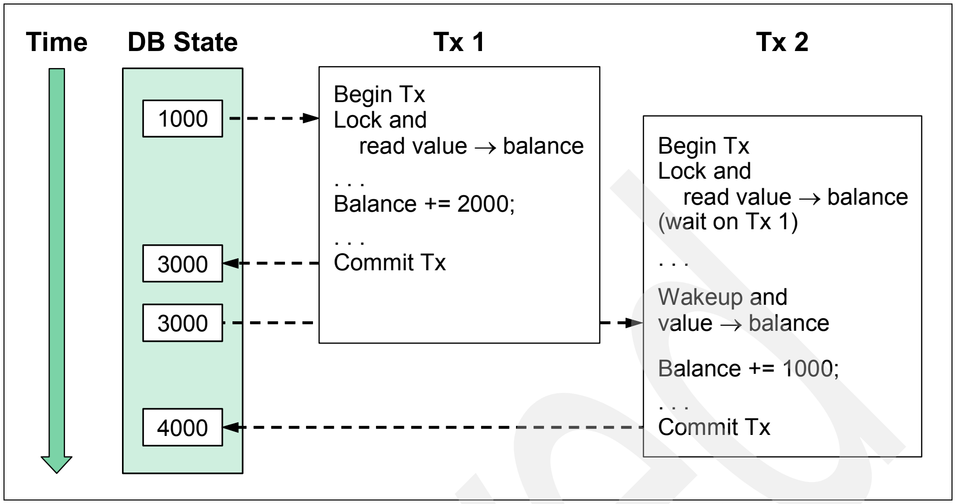


Figure 6-12 Pessimistic locking scenario

Figure 6-12 shows the idea behind pessimistic locking. Transaction 1 (Tx1) reads the value 1000 from the database into the balance variable. When Transaction 2 (Tx2) wants access to the row that contains the balance for this account, Tx2 is blocked until Tx1 has updated the balance to 3000 and committed its changes. Tx2 can then proceed with reading the value 3000 and updating it to 4000.

Optimistic locking

The main problem with a pessimistic approach is that transactions have to wait for each other. A way to avoid this is to be more optimistic and believe that it is unlikely that another transaction would want to update the same entity at the same time. With this kind of thinking, locking the data in the beginning of the transaction can be avoided. Instead, it is only locked at the end of the transaction when it is updated. This method requires a way to ensure that the data has not been altered between the time it was read it and when it was updated. This is known as a WriteWriteConflict. See Figure 6-13.

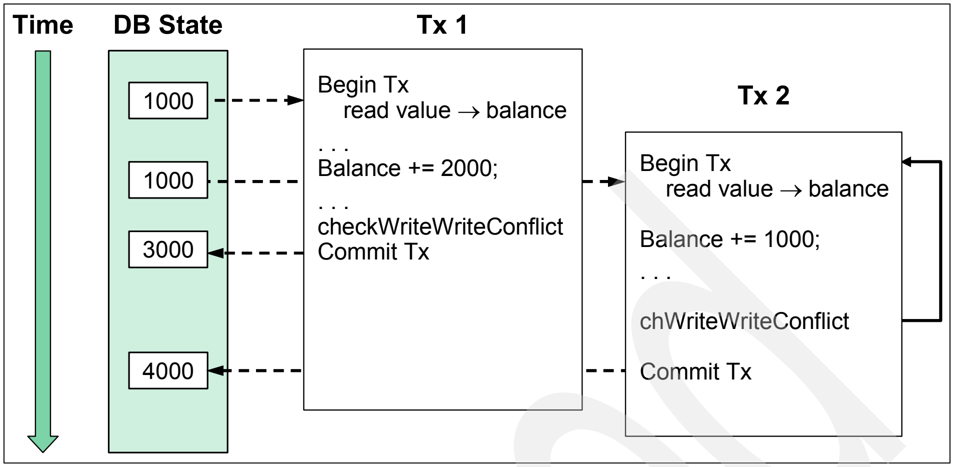


Figure 6-13 Optimistic locking scenario

Figure 6-13 shows the idea behind optimistic locking. Similar to Figure 6-12 on page 372, Transaction 1 (Tx1) reads the balance value as 1000 without locking the row in the database. Transaction 2 (Tx2) is not prevented from doing the same thing, and reads the balance initially as 1000. Before committing the transaction, it must check for conflict and finds that the value is now 3000, which is different from the initial value read. It must now abort the transaction and start the transaction again. The task of checking for a conflict can be done in several ways. An extra column can be used in the database to hold a version or a timestamp. The update is then made with a conditional expression to only update if this field is the same as when the data was read. One or more fields could be used for the same purpose. It would have to be a field that is updated in each transaction.

The use of optimistic locking offers more concurrent access to the data. The drawback is the requirement for collision handling, which is very expensive if collisions are frequent. For this reason it is only appropriate to apply optimistic locking when collisions are rare. If the application access pattern is predominately update access, then a pessimistic approach might be advantageous in that it avoids optimistic concurrency update failures.

Access Intent policies

We now look at the different Access Intents that exist in WebSphere Application Server.

The main identifier for a policy is its locking strategy and type. The type of an access intent can be either read or update. If you assign a read policy to an Entity Bean, an exception is thrown if you attempt to update it. That is, you cannot call any setter methods. It is also interesting to consider what isolation level is to be used, especially if you have other systems accessing the same database. It is important to realize that different database vendors can have their own meanings of the database isolation level. Most implementations are like that shown in Table 6-7 (except for Oracle, which is mapped to different isolation levels). See your database provider's documentation to determine the differences. Table 6-7 shows the seven different policies currently defined in WebSphere Application Server. This data is based on using DB2 as the RDBMS.

Table 6-7 Access Intent policies in WebSphere

Policy	Type	Locking strategy	Isolation level
wsPessimisticUpdate	Update	Pessimistic	Repeatable Read
wsPessimisticUpdateWeakestLockAtLoad (Default policy)	Update	Pessimistic	Repeatable Read
wsPessimisticUpdateNoCollision	Update	Pessimistic	Read Committed
wsPessimisticUpdateExclusive	Update	Pessimistic	Serializable
wsPessimisticRead	Update	Pessimistic	Repeatable Read
wsOptimisticUpdate	Update	Optimistic	Read Committed
wsOptimisticRead	Read	Optimistic	Read Committed

- For other databases scenarios, see the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rdat_isolevtab.html

Pessimistic policies

As described previously, the intent of the pessimistic approach is to ensure that the data loaded into the Entity Bean is locked during the entire duration of a transaction. The normal behavior of a database access is to get a read lock on the queried data, which gets transformed into an update lock if data is updated.

We want to make sure that all the rows are locked when the data is fetched from the database. This can be done by using a specific SQL query referred to as SELECT..FOR UPDATE. The update clause ensures that an update lock is held on the selected rows. A subsequent transaction issuing the same statement on any of the rows is blocked until the first transaction commits its changes. There are five different pessimistic policies defined in WebSphere Application Server. They all have lock hints described by their name, wsPessimistic#hint#, where #hint# further describes our intention with the Entity Bean.

- wsPessimisticUpdate

  No specific hint is specified. The isolation level is be set to Repeatable Read and a "SELECT..FOR UPDATE" SQL statement is used to get an update lock on the row in the database. If you are looking to create a true pessimistic scenario, then this is the policy to use. A drawback is that you cannot have complex queries defined in your finder methods. This is because a SELECT..FOR UPDATE does not support grouping and ordering, so you cannot create a finder that would get a set of rows from the database and order them by a specific column.

- wsPessimisticUpdateWeakestLockAtLoad

  This is the default policy. If no access intent is specified in the deployment descriptor, WebSphere Application Server uses this hint. Compared to wsPessimisticUpdate, it does not use a SELECT..FOR UPDATE clause to obtain an update lock. The container loads the data with the weakest lock available for the targeted database. This means that the rows are locked as shared. When an attempt to update is made, the lock is promoted to an update lock.

- wsPessimisticUpdateNoCollision

  Uses Read Committed, which is a lower isolation level. It does not use a FOR UPDATE clause when loading the data, so the rows do not get locked. Using this policy, one must ensure that there are no concurrent updates to the same row or range of rows. That is, if concurrent transactions update an Entity Bean with same primary key, updates could get lost. Avoid using this policy.

- wsPessimisticUpdateExclusive

  This ensures that the transaction has exclusive access to the data being updated. Database isolation is set to Serializable and a SELECT..FOR UPDATE statement is used when loading the data. Both readers and writers must wait until the exclusive transaction has committed its changes. This policy should be used with caution since it can create serious bottlenecks in the application. It has the same limitation as wsPessimisticUpdate in that complex queries are not supported.

- wsPessimisticRead

  If you know that an Entity Bean is only to be read and not updated, then this policy simulates a read-only lock. If an attempt to update the data in the Entity Bean is made, then an exception is thrown. Be careful and ensure that no other external applications are accessing the same table(s) with different intentions.

- Optimistic policies

  By applying an optimistic policy, data is not locked when the Entity Bean is loaded. Instead, an overqualified update statement UPDATE .. WHERE is used when the Entity Bean is stored back into the database. We see later how to configure this in WebSphere Studio. Situations where an Entity Bean is not updated very often benefits from an optimistic policy. Also if there are a lot more reads than updates on the Entity Bean, an optimistic locking strategy might improve performance.

- wsOptimisticUpdate

  This is the only policy we can use if we choose to implement an optimistic locking strategy to update Entity Beans. It issues a normal select statement to load the data into the Entity Bean, perform its updates, and then issue an overqualified update statement when committing the data.

- wsOptimisticRead

  This only differs from wsPessimisticRead in that the isolation level is set to Read Committed.

- Read-ahead hint

  An extra feature that can be applied to an access intent policy is a read-ahead hint. This is only applicable when there is a relation defined between entities. The idea is to minimize database calls by caching the related beans. For example, if bean A has a relation to B and C, we specify a read-ahead hint on these relations. Then, when A is loaded, one entity of C is loaded since it is a 1:1 relationship, and a matching number of Bs is loaded since it is a 1-M relationship. Read-ahead hints apply only to the findByPrimaryKey method on CMP 2.x Entity Beans. The application server generates a more complex SQL query that brings in the data from multiple tables. When we later request to access these beans, no database access is necessary. See Figure 6-14.

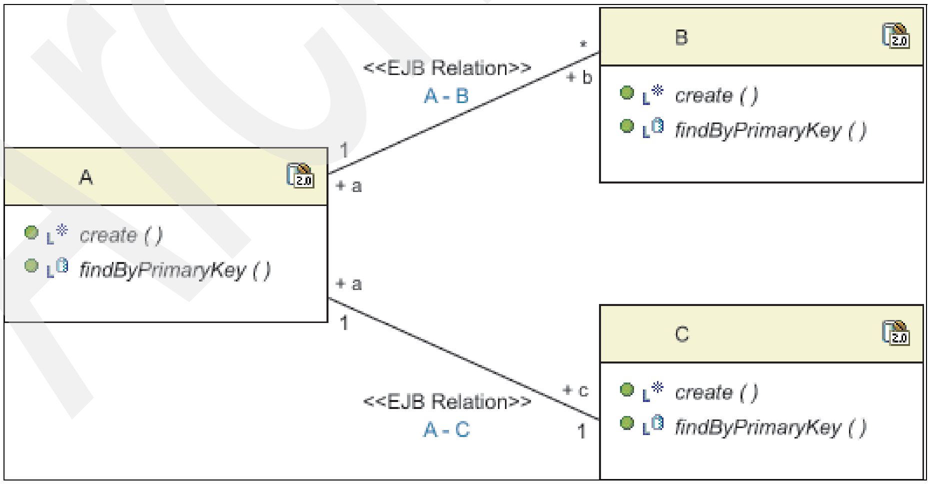


Figure 6-14 EntityBean with container manager relation

Read-ahead hints can be defined on more than one level. If D relates to E and E to F, a hint can be created so that when D is loaded the corresponding E and F entities are also loaded. One should not use read-ahead hints unless the corresponding Container Managed Relations (CMR) bean is touched in the transaction. The hint does imply an extra cost on the database due to the join statement between tables, which should be avoided if not required.

Best practices to use access intent policies

Now we go to best practices for access intent usage.

Deprecated support for methods

Support for applying access intent policies at the method level is deprecated from WebSphere Application Server Version 6.0. In this practice of configuring access intent, you apply a policy to methods within the scope of an EJB module so that the policy becomes the default access intent for all requests upon those methods.

Access intent design considerations

Best practice: Refrain from over-tuning an application. You can introduce errors by incorrectly using the access intent service. For example, misuse of the wsPessimisticUpdate-NoCollision policy can result in lost updates; inappropriately setting the collection increment value can introduce performance issues; and problem determination is more difficult when an application is confusingly configured with multiple access intent policies.

> **Note:** Clarity and simplicity should be your guiding principles when using the access intent service. This is even more important when applying access intent polices within the scope of application profiles.

Even though access intent policies can be configured on any method of an Entity Bean, some attributes of a policy can only be leveraged by the runtime environment under certain conditions. For example, concurrency and access intent are only used for CMP Entity Beans when the ejbLoad() method is driven to open a connection to, and read data from, a given resource; that data is cached and used to drive the proper queries during invocation of the ejbStore() method. Read-ahead hints are only used during the execution of a finder for a bean. Finally, the collection increment and resource manager prefetch increment are only used on multi-object finders.

Configuring policies on methods that do not use those policies is not an error (only certain attributes of any policy are used, even when the policy is appropriately applied to a method). However, configuring policies unnecessarily throughout an application obscures the design of the application and complicates the maintenance of the application.

Access intent with BMP Entity Beans

Access intent's declarative functionality provides great power to you as a CMP Entity Bean developer. You can provide hints on how WebSphere Application Server is to manage the details of persistence without having to explicitly manage any of the persistence logic from within the application. There are situations, however, in which you might have to develop BMP Entity Beans. Because the only meaningful difference between BMP and CMP components is who provides the persistence logic, BMP Entity Beans should be able to leverage access intent hints just as WebSphere Application Server does on behalf of CMP Entity Beans. BMP Entity Beans that use the access intent service participate in application profiling; that is, the value of the access intent attributes can differ from request to request, allowing the BMP Entity Bean to seamlessly modify its persistence strategy.

You can apply access intent policies to BMP Entity Bean methods as well as CMP Entity Bean methods. Because access intent hints are not contractual in nature, there is no obligation for a BMP Entity Bean to exploit them. BMP Entity Beans are expected to use only those access intent attributes that are important to that particular bean.

The current access intent policy is bound into the java:comp namespace for a particular BMP Entity Bean. That policy is current only for the duration of the method call during which the access intent policy was retrieved. In a typical scenario, you would cache the access type during invocation of the ejbLoad() method so that appropriate actions can be taken during invocation of the ejbStore() method.

General access intent best practices

When applying access intent policies to Enterprise JavaBeans (EJB) methods, consider the following issues.

- Start by configuring the default access intent policy for an entity. After your application is built and running, you can more finely tune certain access paths in your application using application profiling or method-level access intent.
- Do not mix access types. Avoid using both pessimistic and optimistic policies in the same transaction. For most databases, pessimistic and optimistic policies use different isolation levels. This can result in multiple database connections, which prevents you from taking advantage of the performance benefits possible through connection sharing.
- Take care when applying wsPessimisticUpdate-NoCollision. This policy does not ensure data integrity. No database locks are held, so concurrent transactions can overwrite each other's updates. Use this policy only if you can be that only one transaction is attempting to update persistent store at any given time.

6.5.13 Enterprise JavaBeans Data Mediator Service

The Enterprise JavaBeans (EJB) Data Mediator Service (DMS) is the Service Data Objects (SDO) Java interface that, given a request in the form of EJB queries, returns data as a DataGraph containing DataObjects of various types.

This differs from a normal EJB finder or ejbSelect method, which also takes an EJB query but returns a collection of EJB objects (all of the same type) or a collection of container managed persistence (CMP) values.

The EJB DMS enables you to specify an EJB query that returns a data graph (the DataGraph) of data objects (DataObjects). The query can be expressed as a compound EJB query contained in a string array of EJB query statements. One advantage of using a DataGraph is that much of the code written in an EJB facade session bean that deals with creating, populating, and updating copy helper objects can be replaced with a DataGraph and a DMS.

Important: The EJB Data Mediator Service has support for EJB2.x container managed persistence (CMP) Entity Beans only.

For more information about this feature, see the following links:

- `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/cejb_ejbmed.html`
- `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/rejb_ejbmedqarg.html`

The foregoing information is more relevant if you have a project with SDO and EJB. However, the best practice here is to use another method more suitable for long lived solutions in persistence such as JPA, as described in 6.6, “Java Persistence API: Entity Beans 3.0” on page 380.

6.5.14 EJB session bean: direct access to back-end

Reading large amounts of data with Entity Beans, for example, just to display a large scrollable list of data, implies a big performance overhead, because a lot of transactional aware EJB objects are unnecessarily instantiated from the EJB container. Therefore, in domains where a set of objects exist whose state is frequently read but very rarely updated, the usage of EJB Entity Beans might be overkill. In these cases, a different approach using an EJB session bean that directly accesses the back-end, might be more advisable. Update operations are also possible because EJB session beans support transactions that can be automatically handled by the container.

Here we list advantages and disadvantages, as well as some alternatives:

- Advantages of this solution:
  - High performance for large amounts of data
  - Full transactional support
  - Universal data access to all types of back-end systems
- Disadvantages of this solution:
  - Limited portability: Implementation highly depends on data schema and type of back-end system
  - Not very easy to use: missing tooling support
  - No caching available
- Alternatives:
  - EJB 3.0 with Java Persistence API (JPA)
  - Session Beans stateless with iBATIS if the first alternative is not available.

6.6 Java Persistence API: Entity Beans 3.0

The Java Persistence API was created as an effort among several development communities such as Hibernate, Java Data Objects (JDO), TopLink®, EJB suppliers and individual workers to unify the persistence Java model. The goal of this persistence Java model was to create a standard in terms of mapping the object/relational way, simplify the programming model, improve performance in a runtime environment, improve search capabilities, support pluggable implementations such as JDBC, and create a unique persistence methodology for J2EE and J2SE environments.

As JPA is new for a lot of developers, in the following sections we explain the main concepts of JPA starting from a point that it is a new persistence technology. We also recommend best practices for its use.

Since the EJB 3.0 specification has become available, a clear separation between persistence and programming model has become apparent. The EJB3 expert group produced three specifications:

- EJB Core Contracts and Requirements
- EJB 3.0 Simplified API
- Java Persistence API

The first and second part of the specification take care of the programming model for session beans, message-driven beans, deployment rules, annotations and so on. The third part addresses persistence items such as entities, ORM (Object relational mappings) metadata, query language and persistence manager interfaces that work with both Java 2 Enterprise Edition (J2EE) and Java 2 Standard Edition (J2SE). We can say that the third part is called Java Persistence API (JPA).

You can think of JPA as the successor to well-known transparent persistence technologies such as JDO and Hibernate because you can add a transparent persistence layer to Plain Old Java Object (POJO) also using a specification placed in EJB 3.0. Hibernate, for example, has interfaces to be used as a JPA provider being in compliance with the JPA specification. POJO is nothing more than a normal class such as Java Beans. However, we cannot use just any class for persistence. There are some rules that have to be applied that we cover in the next sections.

First, we focus on the JPA specification in the following key areas:

- A new POJO-based programming (domain model)
- JPA programming model (overview)
- Own object/relational mapping (metadata)

We then cover more items related to persistence that are standardized, such as:

- Object versioning for optimistic concurrency control
- Database key generation
- Lazy vs. eager loading of fields
- Inheritance and polymorphism
- JPQL
- Best practices for scalability and performance

At the end of this section we talk about JPA adoption in your programs.

6.6.1 Domain Model and POJO-based programming

The first step when you develop an enterprise application is creating the domain model. Domain model means you list the entities and the relationships between them. After that you create UML class diagrams to represents your domain model. The natural evolution in your analysis is mapping your actors (Objects) into Java Objects. If you consider that a domain object has attributes (instance variables in Java Object) and behavior (methods in Java Object), the POJO usage is a normal way in your analysis, design and coding.

To insure a close persistence to the domain model, a new POJO-based programming model is introduced in JPA which applies equally well to both the Java EE and Java SE environments. With POJO adoption, you can select the class in your domain model to be entity objects and apply some basic rules to be ready to persistence in JPA.

POJO, as was mentioned previously, is similar to a JavaBean class. In POJO, you declare business methods that define the behavior and properties that define its state. POJO has some restrictions to make available entities for persistence.

Other important considerations in your domain model and in your Java implementation are:

- Relationships:

  In Java code when an object has a reference to another.

- Multiplicity or cardinality:

  Refers to the nature of relationships that can be:

  – One-to-one: Each side of the relationship can have only one object at most. For example, one Account can have only one checking; on the other hand, one checking can be related to only one account. Another example is that a Customer can have only one CustomerInfo, and CustomerInfo can be related to only one Customer.

  – One-to-many: An object instance can be related to several instances to another object. For example, one account can have several transaction records. However, one transaction record can be related to one account only.

  – Many-to-many: If both sides of the relationship have more than one object of another, this relation is many-to-many. For example, an Author can have several books. On the other hand, a book can be written by several authors.

- Optionality:

  Indicates that a relation does not always exist. For example, a Customer object might not always have a BillingAddress associated. In this case, the one-to-one relation is optional.

POJO for JPA building rules

The restrictions to coding POJOs to make them ready for persistence are as follows:

- Default or No-Arg Constructor:

  The JPA specification requires a constructor with no arguments for each persistent class. The constructor can be nonpublic, but at least package visible. It means that public, protected and package is allowed for constructor.

- Final class and methods are not allowed:

  The entity class cannot be final for methods.

- Identity Fields:

  All entity classes must declare one or more fields which together form the persistent identity of an instance. These entity fields can be implemented as protected or private properties. The accessor methods follow the same JavaBeans specification pattern. See Example 6-28.

  Example 6-28 Pattern for identity fields

```
public class User {
 //Another implementation here
 private String username;

 public User(); // No-argument class constructor
 public String getUsername(){
   return username;
 }
 public void setUsername(String username){
   this.username = username;
 }
}
```

  Annotations can now be used to define metadata directly in the application code. Also you can use an XML-based mechanism.

  The entity POJOs no longer have to implement any EJB framework-based interfaces.

- The programming model also defines a detachment model for the POJO (Entity) objects. This should be a benefit to the client/server and Web application environments.
- Besides enhancing the EJB Query Language (also known as JPQL), the programming model also enables the use of native SQL queries.

6.6.2 JPA programming model

We now introduce some rules to identify a POJO class as a persistence class for JPA work. This feature can be described using both Java annotations and separate xml from Java. We explain using Java annotations.

Make a class persistence capable

The first rule is to use the @Entity annotation that allows a POJO to be persistence capable. The entity concept means that an entity Object can be made persistent by a persistence framework. See Example 6-29.

Example 6-29 @Entity annotation usage

```
@Entity
public class Customer implements Serializable {
//Declaration of attributes here and methods
}
```

An entity requisite is to have a non argument constructor with at least a package visible as we saw in "POJO for JPA building rules" on page 382. In the foregoing example, we did not include a constructor because you can implicitly use the public Customer() constructor.

Establishing a primary key

The second rule is about primary key concerns. All entity objects require a primary key declared. The primary key can be mapped to field-based or properties (via setter and getter methods).

The code snippet in Example 6-30 shows a field-based persistence example.

Example 6-30 id field-based example

```
@Entity
public class Customer implements Serializable {
@id private Long customerId;
}
```

In the foregoing example, the persistence provider would infer that the id field should be persisted because of the @id annotation. The annotations can be used in getters methods (properties) instead of a field-base model. See Example 6-31.

Example 6-31 id property-based example

```
@Entity
public class Customer implements Serializable {
private Long customerId;
@id
public Long getCustomerId(){
return customerId;
}
```

A tip here is that setter methods with annotations (@id for example) are ignored by persistence providers.

A primary key can be simple or composed. In the case of composed, you can use another Java class to represent a Composed Primary Key. See Example 6-32.

Example 6-32 Composed primary key

```
package itso.bank.jpa.model;

import java.util.Date;

public class TransrecordPK implements java.io.Serializable{
  Date transrecordId;
  Long accountId;

  public TransrecordPK(){}

  public boolean equals(Object obj){
    if(this == obj)
      return true;
    if(obj == null || obj.getClass() != getClass())
      return false;
    TransrecordPK other = (TransrecordPK) obj;
    return (other.accountId.equals(accountId)
        && other.transrecordId.equals(transrecordId));
  }

  public int hashCode(){
    return super.hashCode();
  }
}
```

To accomplish this feature in a class in which we have to refer to the composed primary key, we can use @IdClass annotation or @EmbeddedId. The @IdClass usage requires that you repeat @id for fields related to Primary Key class in the main entity class. See Example 6-33.

Example 6-33 @idClass usage

```
@Entity
@IdClass(TransrecordPK.class)
public class Transrecord{

    public Transrecord(){}
    @id
    protected Long accountId;
```

```
    @id
    protected Date transrecordId;

    //....
}
```

You can realize a redundancy in repeating the identity fields in the main class. However, you can more easily note what are the id fields. On the other hand, you can avoid this by using the @Embedded annotation. See Example 6-34.

Example 6-34 @Embedded annotation example

```
@Entity
public class Transrecord{

    public Transrecord(){}

    @EmbeddedId
    protected TransrecordPK transrecordPK;

    //....
}
```

Persisting other fields

By default, the persistence provider saves all entities in fields or properties with public or protected setters and getters. The best practice here is that you should use field-based with accessor methods. In large scale systems mainly, accessor methods are more elegant. To avoid persistence, you can use the @transient annotation or transient Java attribute. In Example 6-35 you can see the usage to avoid persistence.

Example 6-35 Transient usage

```
public class Transrecord {
  private Long accountId;
  private double amount;
  private String details;
  private Date transrecordId;
  private char type;
  @Transient
  protected Long activeTransactionCount;
  /** Or use this approach below
   transient protected Long activeTransactionCount;
  *
  */
```

```
  //...
}
```

Relationships between entities

In 6.6.1, “Domain Model and POJO-based programming” on page 381 we talked about relationship, optionality, and multiplicity. Now we translate these concepts into JPA language. In short, an Entity has a relation with another entity when it holds an object reference to this other entity.

@OneToOne

We use @OneToOne annotation to mark uni and bidirectional one-to-one relationships. This might not make sense, but it is very normal in a domain model.

- Unidirectional one-to-one:

  Consider. for example. a relation between Customer Object and CustomerInfo Object. A Customer Object can have at most one instance of CustomerInfo Object and CustomerInfo cannot exists without Customer. See Example 6-36 using annotations.

  Example 6-36 @OneToOne usage

```
@Entity
public class Customer {
  @Id
  protected Long customerId;
  protected String firstname;
  protected String lastname;
  protected String title;
  @OneToOne
  protected CustomerInfo customerInfo;
}

@Entity
public class CustomerInfo {
  @Id
  protected Long customerinfoId;
  protected String street;
  protected String city;
  protected String state;
  protected String zipcode;
  protected String email;
  protected userId;
  protected password;
```

```
  protected Date password_expiration;
}
```

In the foregoing example, Customer Class has only one attribute CustomerInfo. The @OneToOne annotation says that the persistence provider keeps this relation in the database. Furthermore, you can use @OneToOne relation with properties instead of field approach. See Example 6-37.

Example 6-37 @OneToOne relation usage with properties

```
@Entity
public class Customer {
  private Long customerId;
  private String firstname;
  private String lastname;
  private String title;
  private CustomerInfo customerInfo;

  @OneToOne
  public CustomerInfo getCustomerInfo(){
     return this.customerInfo;
  }

  public void setCustomerInfo(CustomerInfo customerInfo){
     this.customerInfo = customerInfo;
  }
}

@Entity
public class CustomerInfo {
  @Id
  protected Long customerinfoId;
  protected String street;
  protected String city;
  //....
}
```

- Bidirectional one-to-one:

  Analyzing the foregoing relationship, suppose that you have to get Customers from a CustomerInfo, for example, to get Customers that have a password soon to expire. In this case, you have to reach Customers from the CustomerInfo Object, then a Bidirectional one-to-one is required. Example 6-38 example explains how it works.

Example 6-38 One-to-One bidirectional relationship

```
@Entity
public class Customer {
  @Id
  protected Long customerId;
  protected String firstname;
  protected String lastname;
  protected String title;
  @OneToOne
  protected CustomerInfo customerInfo;
}

@Entity
public class CustomerInfo {
  @Id
  protected Long customerinfoId;
  protected String street;
  protected String city;
  protected String state;
  protected String zipcode;
  protected String email;
  protected String userId;
  protected String password;
  protected Date password_expiration;
  @OneToOne(mappedBy="customerInfo", optional="false");
  protected Customer customer;
}
```

Note that in the foregoing example the Object customer that shows a relation from CustomerInfo to Customer. Another interesting point are the @OneToOne parameters. The first parameter mappedBy="customerInfo" tells the JPA container that customerInfo object in the Class Customer is the "owning" side of the relationship. The second parameter is optional and says that the CustomerInfo object cannot exist without Customer Object being present.

@OneToMany and @ManyToOne

This relation is the most common in domain objects. In this case an entity has two or more references to another. In Java terms it means that an entity has a Collection type of another Object for example java.util.Collection, java.util.List, java.util.Set. An important thing is that this relation is bidirectional, it means that while one side of relation is one-to-many the other side is many-to-one. See Example 6-39.

Example 6-39 One-to-many bidirectional example

```
@Entity
public class Account {
 @Id
 protected Long accountId;
 protected double balance;

 @OneToMany(mappedBy="account")
 protected Set<Transrecord> transrecord;
 //..
}

@Entity
public class TransRecord{
 @Id
 protected Long transrecordId;
 protected double amount;
 protected String details;
 protected char type;
 //...
 @ManyToOne
 protected Account account;
 //...
}
```

In the @OneToMany annotation the mappedBy parameter has the same effect as viewed in @OneToOne annotation. It means that the side that has the owning of relationship is in TransRecord object that is account object. As observed in line below from @OneToMany annotation that we are using generics. If you do not use generics you have to use a new parameter called targetEntity to indicate to persistence provider which type the Set is related to. Then the foregoing example code snippet appears similar to Example 6-40.

Example 6-40 targetEntity usage

```
@Entity
public class Account {
 @Id
 protected Long accountId;
 protected double balance;

 @OneToMany(targetEntity=Transrecord.class, mappedBy="account")
 protected Set transrecord;
 //..
```

```
}
```

Looking again at Example 6-39 on page 390, we can see that as this relation is bidirectional in class Transrecord, there is an Account object. Another interesting point is that as ManyToOne is always the owner of the relation, there is no necessity to use the mappedBy parameter in annotation @ManyToOne.

@ManyToMany

This type of relation is not so common in enterprise applications. Consider a relation between Authors and Books. An author can have written several books and a Book can been written by several authors. See the code in Example 6-41.

Example 6-41 ManyToMany relationship usage

```
@Entity
public class Book {
 @Id
 protected String isbn;
 protected String title;
 //...
 @ManyToMany
 protected Set<Author> authors;
 //...
}

@Entity
public class Author{
 @Id
 protected Long authorId;
 protected String name;

 //....
 @ManyToMany(mappedBy="authors")
 protected Set<Book> books;
}
```

As we can see, the authors variable is assigned to @ManyToMany annotation and it is the owner of this bidirectional relation. On the other hand, the books Object that belongs to Author and in this case this other @ManyToMany annotation means that this side of the relation is the subordinated side. The mappedBy parameters guarantee this subordination.

6.6.3 Introducing entity manager and persistence life cycle

Entity manager is the most important part of Java Persistence API. All entities in JPA have a life cycle that is managed by an EntityManager Object. Then an EntityManager performs several actions that affect the life cycle of entity instances. Before looking at entity states, we take an overview of the EntityManager Interface.

The EntityManager interface

The EntityManager function can be thought of as a bridge between relational databases and objects. It makes an interpretation from the object/relational mapping specified for an entity and saves the entity in the database. See Figure 6-15.

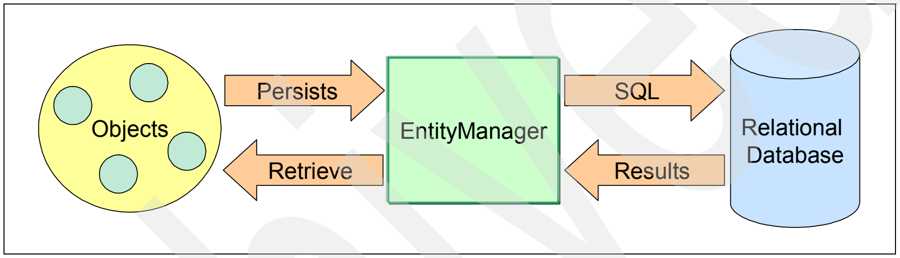


Figure 6-15 The EntityManager functionality

To perform this functionality, EntityManager is a small and easy to use interface. You can see the some of the supported methods in Table 6-8.

Table 6-8 EntityManager methods

Method Signature	Explanation
public void persist(Object entity);	Persists an entity into the database.
public void remove(Object entity);	Removes an entity from the database.
public void flush();	Synchronizes the states of entities managed by EntityManager with database.
public void refresh(Object entity);	Refreshes an entity from the database.
public void close();	Closes an application-managed EntityManager.
public Query createQuery(String jpqlString);	Create a dynamic Query String in JPQL-Like statement.

Method Signature	Explanation

public Query createNativeQuery(String sqlString);
public Query createNativeQuery(String sqlString, Class result);
public Query createNativeQuery(String sqlString, String resultSetMapping);
	Creates a dynamic query using SQL-Like statement.

public Query createNamedQuery(String name)	Creates a named query based in name parameter that belongs to annotation in entity class called @NamedQuery(name=<query_name>,queryString=<query_string>).
public EntityTransaction getTransaction();	Gets a transaction object that can be used to manually start or end a transaction.
public void joinTransaction();	Invokes a EntityManager to join an existing JTA transaction.
public <T> T merge(T entity);	Invoke an EntityManager to merges an entity and returns a merged entity.

We provide some usage examples in Example 6-42, Example 6-43, and Example 6-44.

Example 6-42 Persist Example

```
public Order createNewOrder(Customer customer) {
Order order = new Order(customer);
entityManager.persist(order);
return order;
}
```

Example 6-43 Find and remove example

```
public void removeOrder(Long orderId) {
Order order = entityManager.find(Order.class, orderId);
entityManager.remove(order);
}
```

Example 6-44 change example (merge)

```
public OrderLine updateOrderLine(OrderLine orderLine) {
return entityManager.merge(orderLine);
}
```

Here we list the four states of an entity:

- Transient or New
- Persistent or Managed
- Detached
- Removed

In this section, we discuss these types. Figure 6-16 shows an overview of these four elements and the relations between them.

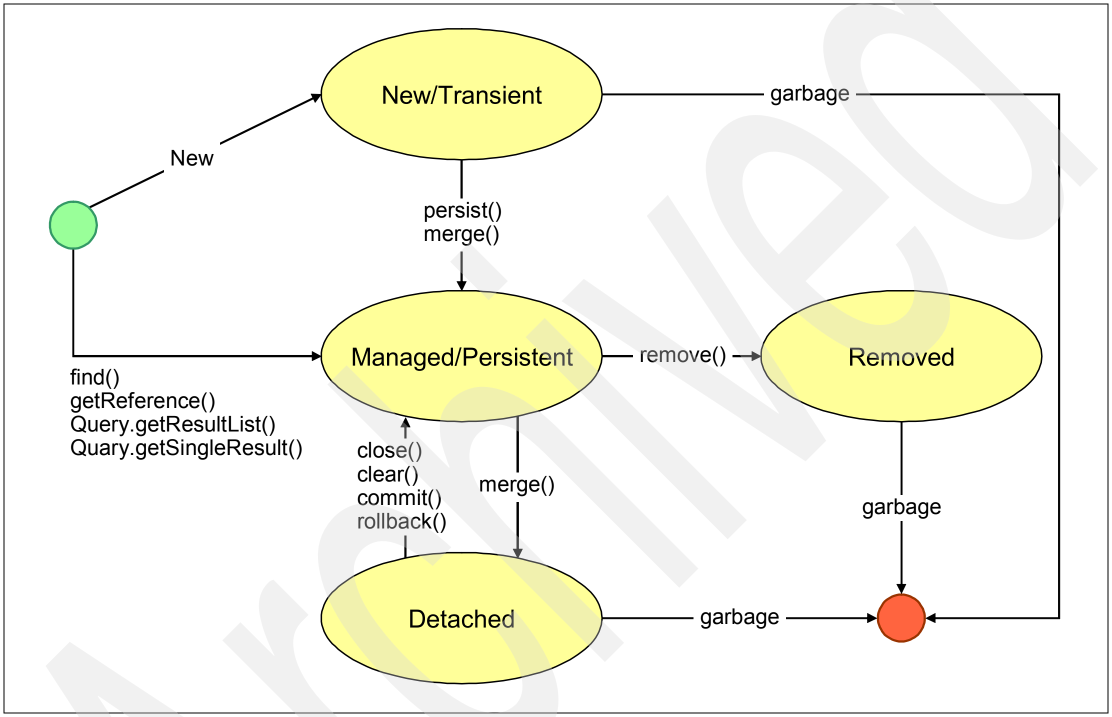


Figure 6-16 Entity states and relations overview

All these methods except Query.getResultList(), Query.getSingleResult(), and garbage action are EntityManager methods. Next we describe these elements.

Transient or new objects

When you use the new operator, the objects instantiated are not persistent immediately. We can infer that its state is transient, which means that they are not associated to any database table row. if an object of this type is not referred by any other object, its state is lost and is available for garbage collection. By default, objects that are referenced only by other transient instances are also transient. To become persistent or managed requires a call to a persistence manager or assigning a reference from an already persistent instance.

Persistent or managed objects

If an entity instance has a primary key value as a database identifier, then it means that an entity instance is a persistent and managed instance. A persistent instance can be created by an application and made persistent by calling one of the persistence manager methods. Also it can be created when a reference was created from another persistent object or by query execution, by an identifier lookup, or from another persistent instance you navigate to its object graph.

Removed objects

You can remove an object executing explicitly by removing the method from the persistence manager object. A removed state means that an object has been scheduled for deletion at the end of the unit of work, and while the end of work is not reached, it is still managed by the persistence manager. Moreover, this removed object should not be used because it is to be deleted when the unit of work finishes.

Detached objects

An instance is considered detached after a unit of work completes and the persistence context is closed. This state indicates that their state is no longer guaranteed to be synchronized with the database state because it is not attached to a persistent context.

6.6.4 Object/relational mapping

O/R mapping is a crucial element of an application designed around JPA. It directly affects the way the entity manager populates the domain objects. As a result, a mapping change can be felt as far as the presentation tier. You can produce especially undesirable effects by changing fetch type or cascade type.

You have two ways to define the mapping: metadata (annotations) and a mapping file. Although the metadata approach is heavily promoted, you should be mindful of the baggage it carries. The approach essentially entangles two logical layers of an application: the domain model and the mapping information. Being separate, these logical layers require separate testing using different techniques. The metadata approach does not hamper the testability of the layers per se. Rather, it causes two layers to appear to be one, which might or might not be a problem depending on a range of factors.

One factor that can affect the choice of a mapping approach is the project team structure. In a small project with a few developers, a small number of tables (fewer than 100 as a rule of thumb) and no dedicated mapping person, taking the annotation approach for defining mappings is probably the best bet because working with annotations is generally faster. For medium-sized and large projects that have a dedicated mapping person or team, a smarter choice is to go with a mapping-files approach. This approach reduces resource contention and adds another degree of freedom to the development process. The metadata-based mapping approach proved to be more time-efficient for the PetStore application.

The goal of an O/R mapping layer is to shield the rest of the application from the effects of changes in the underlying database. When we migrated the PetStore application back-end to PostgreSQL, the mapping layer did not require any changes at all. This can be attributed to the fact that both the original database and PostgreSQL support sequences, so the primary key generation strategy stayed intact. In the general case, you should expect some rework in the area of the mapping related to object ID handling. In short, the JPA specification explicitly defines the object/relational (O/R) mapping instead of relying on vendor-specific O/R mapping of past releases. This should help with vendor plugability and application migration.

Thorough test coverage of the mapping is of paramount importance. You must cover all the relationship mappings to make sure fetching behavior and transitive persistence is tested. You can leverage out-of-container use of JPA to perform the task (more on that in the next section).

Mapping entities bases

In this section we examine some fundamental features of EJB 3.0 JPA O/R mappings such as @Table, @Column, @Enumeration, @Lob, @Temporal and @Embeddable. See Example 6-45.

Example 6-45 Entity mapping

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(name="CUSTOMER_TOKEN",
  pkJoinColumns=@PrimaryKeyJoinColumn(name="CUSTOMER_ID"))
public class Customer implements Serializable{

  //Column Mappings
  @Id
  @Column(name="CUSTOMER_ID", nullable=false)
  protected Long customerId;

  @Column(name="FIRSTNAME", nullable=false)
  protected String firstname;

  @Column(name="LASTNAME", nullable=false)
  protected String lastname;

  //Enumerated field
```

```
  @Enumerated(EnumType.STRING)
  @Column(name="CUSTOMER_TYPE", nullable=false)
  protected CustomerType customerType;

  //Blob Column
  @Column(name="TOKEN" ,table="CUSTOMER_TOKEN")
  @Lob
  @Basic(fetch=FetchType.LAZY) //Lazy loading
  protected byte[] securityToken;

  //Temporal field
  @Column(name="JOIN_DATE", nullable=false)
  @Temporal(TemporalType.DATE)
  protected Date joinDate;

  //Embedded field
  @Embedded
  protected CustomerInfo customerInfo;

  public Customer() {
  }
}

@Embeddable
public class CustomerInfo implements Serializable{
  @Column(name="USERID", nullable=false)
  protected String userId;

  @Column(name="PASSWORD", nullable=false)
  protected String password;

  @Column(name="EMAIL", nullable=false)
  protected String email;

  @Column(name="STREET", nullable=false)
  protected String street;

  @Column(name="CITY", nullable=false)
  protected String city;

  @Column(name="STATE", nullable=false)
  protected String state;

  @Column(name="ZIPCODE", nullable=false)
  protected String zipcode;
```

```
}
```

Table mapping

The table mapping is done using the @Table annotation and the name parameter represents the table name in database.

Column mapping

The column mapping is done using the @Column annotation and name params represents the column name in Table specified in @Table annotation. If you use @SecondTable annotation in a case where you map a table to more than one column, you can use the parameter table. See Example 6-46.

Example 6-46 Column annotation with table usage

```
@Column(name="TOKEN" ,table="CUSTOMER_TOKEN")
```

@Enumerated annotation

The enumeration type was introduced in Java 1.5. Note in Example 6-45 on page 396 that there is a Class called CustomerType. The class is defined as shown in Example 6-47.

Example 6-47 Enumeration class

```
public enum CustomerType {GOLD, STANDARD, MASTER};
```

We decided to save, in a database, the String value such as “GOLD”. Then we set the enumeration Type as String (Example 6-48).

Example 6-48 Enumeration annotation

```
@Enumerated(EnumType.STRING)
```

If we decided to set the array order for example 0,1,2... we could use:

```
@Enumerated(EnumType.ORDINAL)
```

CLOBs and BLOBs mappings

To map a CLOB or BLOB data type, we use @Lob annotation. See the code snippet in Example 6-49, which was extracted from Example 6-45 on page 396.

Example 6-49 @lob annotation usage

```
@Lob
@Basic(fetch=FetchType.LAZY) //Lazy loading
```

The @Basic annotation states that we should load this data from a database when it is first accessed. This approach is very important because CLOBs or BLOBs consume much memory and have to be loaded if necessary. Otherwise we can use fetch=FetchType.EAGER.

Temporal mapping types

In general, databases support temporal data types such as DATE (day, month and year) or TIME (storing just time without DATE presentation). The default value if not specified is TemporalType.TIMESTAMP. However, in Example 6-50 we defined it as date.

Example 6-50 Temporal types usage

```
@Column(name="JOIN_DATE", nullable=false)
@Temporal(TemporalType.DATE)
protected Date joinDate;
```

Entity to multiple table mapping

Sometimes entity data has to be mapped from two different tables. This strategy is done using a code snippet from our source example (Example 6-51).

Example 6-51 An entity for two tables strategy

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(name="CUSTOMER_TOKEN",
  pkJoinColumns=@PrimaryKeyJoinColumn(name="CUSTOMER_ID"))
public class Customer implements Serializable{
```

This kind of relation is created by mapping a foreign key in the secondary table pointing to the primary key in the first table. The parameter pkJoinColumns sentence does this service. In this case, CUSTOME_ID is the primary key for the first column called CUSTOMER and foreign key to CUSTOMER_TOKEN table.

Database key generation

There are three popular ways to manually generate primary key values that are identifiers, sequences, and tables. All the three types are supported by JPA. Otherwise you can leave it to the persistence provider to generate a database key. These types are described in the next sections.

Identity columns as generators

Some databases support identity columns. By changing the customerId field declaration in Example 6-45 on page 396, it looks as shown in Example 6-52.

Example 6-52 Database identity columns

```
@Id
@GeneratedValue(strategy=Generation.IDENTITY)
@Column(name="CUSTOMER_ID", nullable=false)
protected Long customerId;
```

This coding assumes that identity constraints exist in the CUSTOMER_ID column in table CUSTOMER. When we use generator type, the value of the identity field is only available when the data is safely on the database.

Using database sequences as generators

To use sequences as generators, the first step is generated in a database. In DB2, for example, a sequence is created using the command in Example 6-53.

Example 6-53 Sequence command database example

```
CREATE SEQUENCE CUSTOMER_SEQUENCE START WITH 1 INCREMENT BY 10
```

In the next step we create a sequence sentence for JPA (Example 6-54).

Example 6-54 Sequence generator annotation in JPA

```
@SequenceGenerator(name="CUSTOMER_SEQUENCE_GENERATOR"
sequenceName="CUSTOMER_SEQUENCE" initialValue=1, allocationSize=5)
```

In the foregoing example we created a sequence generator called CUSTOMER_SEQUENCE_GENERATOR related to CUSTOMER_SEQUENCE created in our DB2 database. By default, if you do not declare otherwise, the initialValue is 0, and the allocationSize (the increment) is 50.

Finally we define the generation key for our CUSTOMER_ID column as shown in Example 6-55.

Example 6-55 Generated key for sequence generator

```
@Id
@GeneratedValue(strategy=GeneratorType.SEQUENCE,
generator="CUSTOMER_SEQUENCE_GENERATOR")
@Column(name="CUSTOMER_ID")
protected Long customerId;
```

Using sequence tables as generators

You can use sequence tables as generators. In DB2 the command is shown in Example 6-56.

Example 6-56 Create Table sequence command

```
CREATE TABLE SEQUENCE_GENERATOR_TABLE (SEQUENCE_NAME VARCHAR(60) NOT
NULL,SEQUENCE_VALUE INTEGER NOT NULL, PRIMARY KEY (SEQUENCE_NAME))
```

The next step is to insert an initial value manually in the table created. See Example 6-57.

Example 6-57 Insert an initial value to the sequence table

```
db2 => INSERT INTO SEQUENCE_GENERATOR_TABLE (SEQUENCE_NAME,
SEQUENCE_VALUE) VALUES('CUSTOMER_SEQUENCE', 1)
```

These two commands combined have the same effect as using a database sequence. To accomplish this command in JPA, use the @TableGenerator annotation shown in Example 6-58.

Example 6-58 @TableGenerator

```
@TableGenerator(name="CUSTOMER_TABLE_GENERATOR”,
table="SEQUENCE_GENERATOR_TABLE", pkColumnName="SEQUENCE_NAME",
valueColumnName="SEQUENCE_VALUE", pkColumnValue="CUSTOMER_SEQUENCE")
```

You can specify the initialValue and allocationSize in the same form as database sequence generation.

The final step is do the annotation for the id field customerId as shown in Example 6-59.

Example 6-59 Final annotation configuration

```
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
generator="CUSTOMER_TABLE_GENERATOR")
@Column(name="CUSTOMER_ID")
protected Long customerId;
```

Embeddable class mapping

Figure 6-17 shows how @Embeddable/@Embedded annotations mapping to Embeddable class mapping works. It describes a customer table structure.

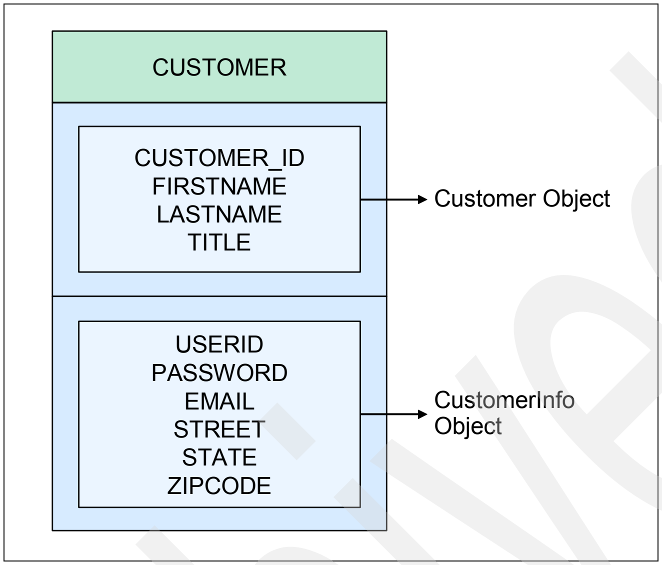


Figure 6-17 Table for Embeddable class mapping

With the foregoing example, we map the customer table into two different objects: Customer Object and CustomerInfo Object. To use this approach, we implement it using @Embeddable/@Embedded annotations. See Example 6-60.

Example 6-60 Embeddable class mapping example

```
@Entity
@Table(name="CUSTOMER")
public class Customer implements Serializable{

  //Column Mappings
  @Id
  @Column(name="CUSTOMER_ID", nullable=false)
  protected Long customerId;

  @Column(name="FIRSTNAME", nullable=false)
  protected String firstname;

  @Column(name="LASTNAME", nullable=false)
  protected String lastname;
```

```
  //Embedded field
  @Embedded
  protected CustomerInfo customerInfo;

  public Customer() {
  }
}

@Embeddable
public class CustomerInfo implements Serializable{
  @Column(name="USERID", nullable=false)
  protected String userId;

  @Column(name="PASSWORD", nullable=false)
  protected String password;

  @Column(name="EMAIL", nullable=false)
  protected String email;

  //Other mappings
}
```

The @Embedded annotation refers to customerInfo attribute inserted into Customer class and @Embeddable refers to CustomerInfo class.

Mapping relations

In 6.6.1, "Domain Model and POJO-based programming" on page 381, we explained about entity relation concepts and in "Relationships between entities" on page 387 the annotations rules for each relationship type. Now we finish using the object relational annotation mapping to make a whole example.

One-to-one

One-to-one relationships are mapped using a parent and foreign key approach. There are two types for this approach, depending on where the foreign key exists: using @JoinColumn and @PrimaryKeyJoinColumn annotations.

- @JoinColumn

  Use this approach when an entity has its primary key but contains the foreign key reference to the table which the referenced entity child is mapped. From Example 6-36 on page 387 we changed the example, doing a resume and adding @JoinColumn annotation. The result is shown in Example 6-61.

Example 6-61 @JoinColum example usage

```
@Entity
@Table(name="CUSTOMER")
public class Customer {
  @Id
  @Column(name="CUSTOMER_ID", nullable=false)
  protected Long customerId;

  @Column(name="FIRSTNAME", nullable=false)
  protected String firstname;

  //Another fields mapping

  @OneToOne
  @JoinColumn(name="CUST_INFO_ID",
     referencedColumnName="CUSTOMER_INFO_ID, updatable=false)
  protected CustomerInfo customerInfo;
}

@Entity
@Table(name="CUSTOMER_INFO")
public class CustomerInfo {
  @Id
  @Column(name="CUSTOMER_INFO_ID")
  protected Long customerinfoId;

  @Column(name="STREET", nullable=false)
  protected String street;

  @Column(name="CITY", nullable=false)
  protected String city;
  //....
}
```

The name parameter of @JoinColumn annotation refers to the foreign key in CUSTOMER table. Then CUST_INFO_ID is a column in CUSTOMER table that refers CUSTOMER_INFO_ID in table CUSTOMER_INFO. If you omit the second parameter referencedColumnName, the JPA processor uses the @Id of CustomerInfo object automatically. It means that it uses column CUSTOMER_INFO_ID automatically if not declared in the referencedColumnName parameter in the @JoinColumn annotation.

- @PrimaryKeyJoinColumn

  In Example 6-62, two entity instances share the same primary key. However, one is a foreign key and the other is a primary key. In Example 6-62, the CUSTOMER_INFO table has a foreign key CUSTOMER_INFO_ID pointing to the CUSTOMER_ID that is the primary key of the CUSTOMER table. Also, CUSTOMER_INFO_ID is the primary key of the CUSTOMER_INFO table.

  Example 6-62 @PrimaryKeyJoinColumn usage

```
@Entity
@Table(name="CUSTOMER")
public class Customer {
  @Id
  @Column(name="CUSTOMER_ID", nullable=false)
  protected Long customerId;

  @Column(name="FIRSTNAME", nullable=false)
  protected String firstname;

  //Another fields mapping

  @OneToOne
  @PrimaryKeyJoinColumn(name="CUSTOMER_ID",
     referencedColumnName="CUSTOMER_INFO_ID, updatable=false)
  protected CustomerInfo customerInfo;
}

@Entity
@Table(name="CUSTOMER_INFO")
public class CustomerInfo {
  @Id
  @Column(name="CUSTOMER_INFO_ID")
  protected Long customerId;

  @Column(name="STREET", nullable=false)
  protected String street;

  @Column(name="CITY", nullable=false)
  protected String city;
  //....
}
```

One-to-many and many-to-one

As we described in "@OneToMany and @ManyToOne" on page 389, this relation is more common in our domain model and by consequence in our systems. This relation is implemented as a primary-key and foreign-key association in the underlying database. See Figure 6-18.

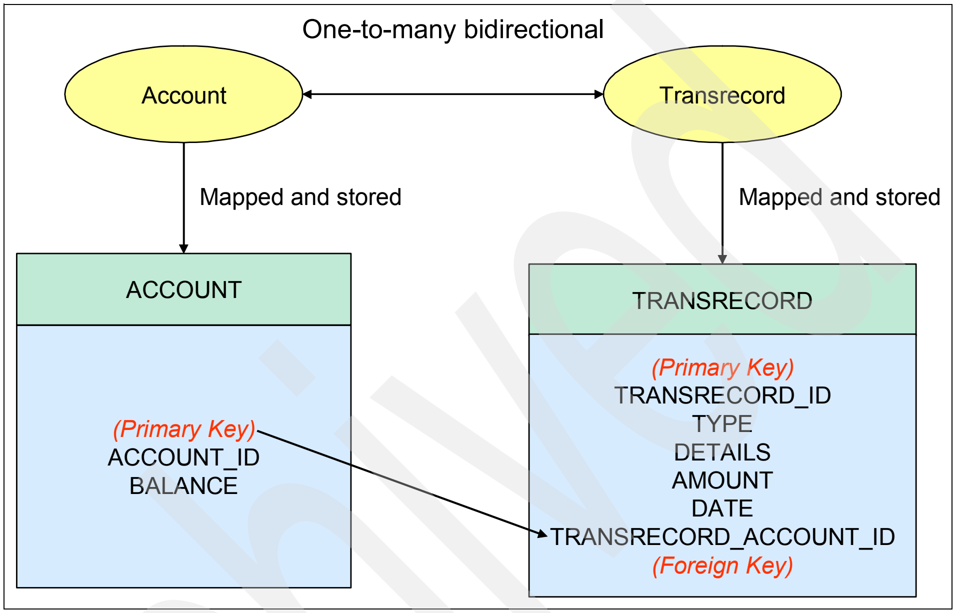


Figure 6-18 One-to-many bidirectional O/R mapping example

Looking from the source in "One-to-many bidirectional example" on page 390, we changed to the example shown in Example 6-63.

Example 6-63 One-to-many bidirectional example using O/R mapping

```
@Entity
public class Account {
 @Id
 @Column(name="ACCOUNT_ID")
 protected Long accountId;

 //...

 @OneToMany(mappedBy="account")
 protected Set<Transrecord> transrecord;
 //...
}
```

```
@Entity
@Table(name="TRANSRECORD")
public class TransRecord{
 @Id
 @Column(name="TRANSRECORD_ID")
 protected Long transrecordId;

 //...
 @ManyToOne
 @JoinColumn(name="TRANSRECORD_ACCOUNT_ID",
     referencedColumnName="ACCOUNT_ID")
 protected Account account;
 //...
}
```

In the foregoing case, several instances of TransRecord refer to one Account. At the same time, several instances of TRANSRECORD records refer to the same record in the ACCOUNT table. The TRANSRECORD table has a foreign key, which is TRANSRECORD_ACCOUNT_ID referring to the ACCOUNT_ID in table ACCOUNT. To define the described relation, @JoinColumn annotation is used. Moreover the mappedBy parameter in @OneToMany annotation defines the owner of relation that is account attribute in TransRecord. In short, in a bidirectional relation of one-to-many, the owner of relation is in the entity side that is, the many side of the relation, and keeps the foreign key.

Note: JPA does not support unidirectional one-to-many relationships. However, these relationships are not so common. When you use such relationships, you must do the navigability yourself. Then, for easier maintenance, we recommend that you change to bidirectional one-to-many, many-to-one in your model.

Many-to-many

Many-to-many relations can be split in two relations of one-to-many stored in an association or join table. In the database world, we map each primary key in both of the relation sides in the foreign keys pair in a join table. See Figure 6-19.

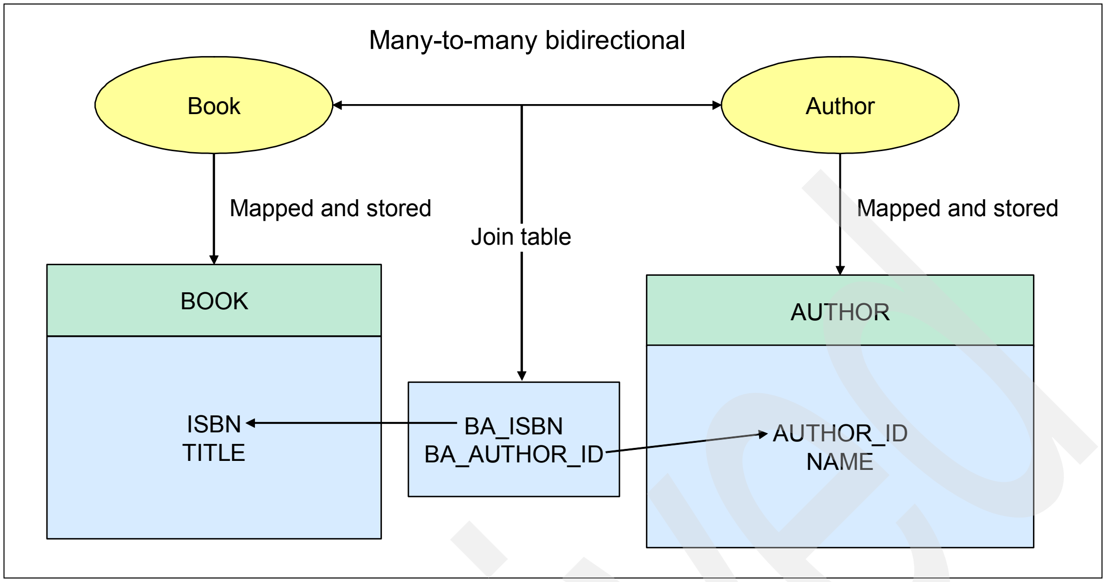


Figure 6-19 Many-to-many relationship database modeling

To represents this relationship in O/R mapping, we use the @JoinTable annotation. See Example 6-64.

Example 6-64 Many-to-many relationship mapping

```
@Entity
public class Book {
 @Id
 protected String isbn;
 protected String title;
 //...
 @ManyToMany
 @JoinTable(name=BOOKS_AUTHORS,
     joinColumns=
        @joinColumn(name="BA_ISBN",
              referenceColumnName="ISBN"),
     inverseJoinColumns=
        @joinColumn(name="BA_AUTHOR_ID",
              referenceColumnName="AUTHOR_ID"))
 protected Set<Author> authors;
 //...
}

@Entity
```

```
public class Author{
 @Id
 protected Long authorId;
 protected String name;

 //....
 @ManyToMany(mappedBy="authors")
 protected Set<Book> books;
}
```

The @JoinTable in Book class specifies the association or join table. In our example this is the BOOKS_AUTHORS table, which contains two columns. The first column is BA_ISBN, which is a foreign key mapped to the ISBN column in table BOOK. The second column is BA_AUTHOR_ID, which is a foreign key mapped to the AUTHOR_ID column in the AUTHOR table. This rule is described in joinColumns and inverseJoinColumns on both sides of relation. joinColumns represents the owning of relation, and inverseJoinColumns represents the subordinate side of the relation. The mappedBy element on the Author.books attribute points to Book.authors. This sentence also indicates that Author.books is a subordinate element and Book.author is the owner of the relation.

Inheritance

To start this approach, consider the domain model example shown in Figure 6-20.

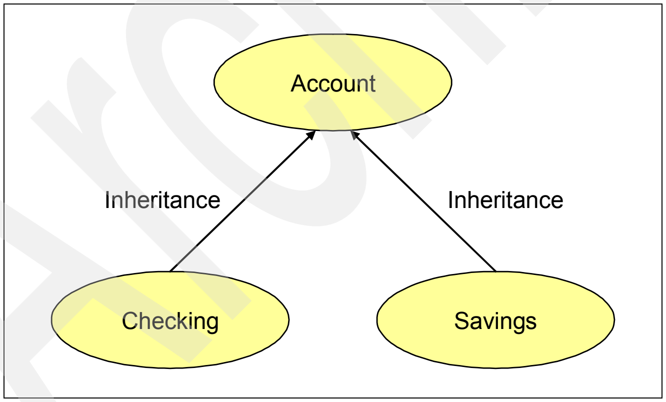


Figure 6-20 Domain model for inheritance example

In this example, we have Checking and Savings inheriting from Account. We can see this model in the following inheritance approaches that are supported: single table, joined tables, and table per class.

- Single table:

  All classes in inheritance schema are mapped to only one table where one field value identify a specific subclass. See Figure 6-21.

ACCOUNT_ID	BALANCE	ACCOUNT_TYPE	OVERDRAFT	MINAMOUNT
1996524342	5,000	C	200	NULL
1244126063	7,000	V	NULL	500

Figure 6-21 Single table approach

In this table, one field called ACCOUNT_TYPE is a discriminator column with value ‘C’ that represents checking type and ‘S’ that represents savings. This is the main point to represent a single table in the inheritance approach. ACCOUNT_ID and BALANCE have values that are common to all accounts. When the persistence provider persists Checking type, it fills all columns except MINAMOUNT, which is left with a NULL value because Checking type does not require this column to be filled. In Example 6-65 we can see this approach implemented in annotations inside the code.

Example 6-65 Code with inheritance mapping using single table

```
@Entity
@Table(name="ACCOUNT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="ACCOUNT_TYPE",
           discriminatorType=DiscriminatorType.STRING,lengh=1)
public abstract class Account{
     @Id
     @Column(name="ACCOUNT_ID", nullable=false)
     protected Long accountId;
     @Column(name="BALANCE", nullable=false)
     protected double balance;
     @Column(name="ACCOUNT_TYPE", nullable=false)
     protected char accountType;
}

@Entity
@DiscriminatorValue(value="C")
public class Checkings extends Account{
    @Column(name="OVERDRAFT")
```

```
    protected double overdraft;
}

@Entity
@DiscriminatorValue(value="S")
public class Savings extends Account{
    @Column(name="MINAMOUNT")
    protected double minamount;
}
```

In this coding example, the @Inheritance annotation with parameter strategy.SINGLE_TABLE represents this inheritance type. @DiscriminatorColumn points to a field that indicates the branch to the two types of inheritance and @DiscriminatorValue the value types. Also note that the common values stay and abstract class and are not null. On the other hand, the optional values are in the subclasses.

- Joined tables:

  The inheritance root is represented by a table and each subclass is represented to a separated table that contains specific fields to subclass and columns that represents its primary keys. The relation with root and descendants in the hierarchy is with a one-to-one relationship. In the joined-tables strategy, the parent of the hierarchy has the common fields and the children's specific fields. See Figure 6-22.

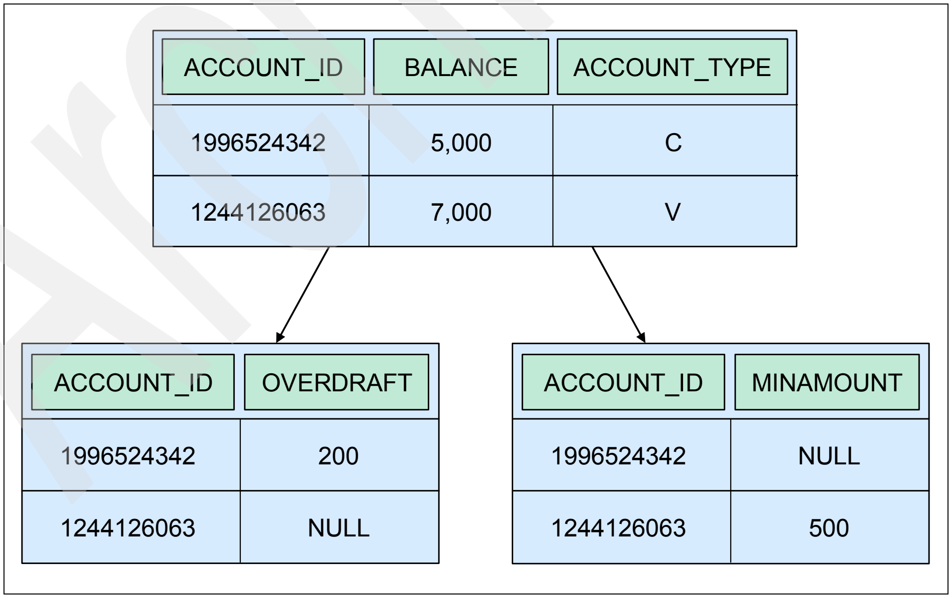


Figure 6-22 Joined-tables strategy

The code below represents the implementation in JPA. The unique difference from one table is that the relation between parent and children tables are implemented using @PrimaryKeyJoinColumn annotations. In this case ACCOUNT_ID represents the foreign key. See the code in Example 6-66.

Example 6-66 Joined-table strategy

```
@Entity
@Table(name="ACCOUNT")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="ACCOUNT_TYPE",
           discriminatorType=DiscriminatorType.STRING,lengh=1)
public abstract class Account{
     @Id
     @Column(name="ACCOUNT_ID", nullable=false)
     protected Long accountId;
     @Column(name="BALANCE", nullable=false)
     protected double balance;
     @Column(name="ACCOUNT_TYPE", nullable=false)
     protected char accountType;
}

@Entity
@DiscriminatorValue(value="C")
@PrimaryKeyJoinColumn(name="CUSTOMER_ID")
public class Checkings extends Account{
    @Column(name="OVERDRAFT")
    protected double overdraft;
}

@Entity
@DiscriminatorValue(value="S")
@PrimaryKeyJoinColumn(name="CUSTOMER_ID")
public class Savings extends Account{
    @Column(name="MINAMOUNT")
    protected double minamount;
}
```

This approach is better for design but worse for performance as compared to the single table approach because of polymorphic queries that require joining of multiples tables.

- Table per class:

  Each class in the inheritance schema is mapped to a separated table where all properties of each class are mapped to database columns including inherited properties. This is simpler than other approaches but worse from a relational and OO point of view. See Figure 6-23.

ACCOUNT_ID	BALANCE
1996524342	5,000
1244126063	7,000

ACCOUNT_ID	BALANCE	OVERDRAFT
1996524342	5,000	200

ACCOUNT_ID	BALANCE	MINAMOUNT
1244126063	7,000	500

Figure 6-23 Tables-per-class approach

Notice that some columns such as BALANCE are repeated using this approach.

This type of inheritance is implemented in the superclass and each subclass as in your own table has the @Table annotation. See Example 6-67.

Example 6-67 Table-per-class approach

```
@Entity
@Table(name="ACCOUNT")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Account{
    @Id
    @Column(name="ACCOUNT_ID", nullable=false)
    protected Long accountId;
    @Column(name="BALANCE", nullable=false)
    protected double balance;
```

```
}

@Entity
@Table(name="CHECKINGS")
public class Checkings extends Account{
    @Id
    @Column(name="ACCOUNT_ID", nullable=false)
    protected Long accountId;
    @Column(name="OVERDRAFT", nullable=false)
    protected double overdraft;
}

@Entity
@Table(name="SAVINGS")
public class Savings extends Account{
    @Id
    @Column(name="ACCOUNT_ID", nullable=false)
    protected Long accountId;
    @Column(name="MINAMOUNT", nullable=false)
    protected double minamount;
}
```

In this coding example, notice that @Id annotation is repeated between subclasses. The problem using this approach is that it does not offer good support of polymorphic queries because each subclass is mapped to its own table.

Now that we know the three types of inheritance strategies, the best approach is single-table, which is simple and has good performance because it avoids joins in multiple tables in polymorphic queries. However, single-table does not manage a large amount of data very well, because of NULL-values columns. In that case, consider the joined-table approach. The worst approach is the table-per-class approach because it uses almost no relational database features. Also, this approach is more difficult to implement reliability in JPA and it was made optional for providers in the EJB 3.0 specification. It is not recommended to use.

Polymorphism

We can consider a polymorphic relation between entities when a relation can refer to instances of a subclass in some side of the relation. For instance, consider an example between Transrecord and Account that is a one-to-many bidirectional relationship. When we retrieve the relation from TransRecord, the retrieved instance can be Savings or Checking instance. This is an advantage of JPA that you make a relation between superclass (Account) and another class (Transrecord) and the relation is polymorphic automatically.

6.6.5 JPQL overview

JPQL is considered an evolution from EJB Query Language (EJBQL), which is the query language from EJB 2.x. The important point here is that JPQL, as is EJBQL, is a superset of SQL query to work with objects. Moreover, each JPQL query is translated to an SQL query by a JPQL query processor that is supplied by a JPA provider and executed in the database.

Improvements from EJBQL

The improvements from EJBQL are shown next.

Group by/ having sentences

Group by/ having sentences from EJBQL are shown in Example 6-68 and Example 6-69 respectively.

Example 6-68 Group by example

```
SELECT t.accountId, COUNT(t.transrecordId) FROM Transrecord t GROUP BY
t.accountId
```

Example 6-69 Having example

```
SELECT t.accountId, COUNT(t.transrecordId) FROM Transrecord t GROUP BY
t.accountId HAVING COUNT(t.transrecordId) > 10
```

Subqueries

A subquery is a query inside a query, and EJBQL did not have this support. The syntax of a subquery is shown in Example 6-70.

Example 6-70 Syntax of subquery

```
[NOT] IN/ [NOT] EXISTS / ALL / ANY / SOME (subquery)
```

The code in Example 6-71 shows how to use this query.

Example 6-71 Subquery example

```
SELECT a FROM Account a WHERE EXISTS (SELECT t FROM Transrecord t WHERE
t.accountId = a.ccountId)
```

SQL additional functions

The following functions were included in JPQL: UPPER, LOWER, TRIM, CURRENT_DATE

Joins

The join operator can be used to create a Cartesian product between two entities. See Example 6-72.

Example 6-72 Join example

```
SELECT a FROM Account a INNER JOIN a.Transrecord t WHERE a.accountId
LIKE ?1
```

Polymorphic queries

This means that if a JPQL query retrieves a parent entity in an entity hierarchy, it is not limited to the entity, but all subclasses.

Dynamic queries

In Example 6-73, we are using a JPQL dynamic query, and the parameter is passed using the setParameter method in the Query object.

Example 6-73 Dynamic query using JPQL

```
public List findWithAccount (Long id) {
 Query query = em.CreateQuery (“SELECT a FROM Account a” +
                "WHERE a.accountId LIKE :accId”);
 query.setParameter(“accId”, id);
 query.setMaxResults(10);

 return query.getResultList();
}
```

The same example using native SQL query is shown in Example 6-74.

Example 6-74 Dynamic query using native SQL

```
public List findWithAccount (Long id) {
 Query query = em.CreateNativeQuery (“SELECT * FROM ACCOUNT
                WHERE ACCOUNT_ID LIKE ”+id.longValue());

 return query.getResultList();
}
```

Comparing the two examples, we can note that the unique difference in dynamic queries using JPQL and native SQL is that the named parameter in SQL query is not required by the JPA specification.

We use native SQL query in the next example, and we define what result you want to return dynamically using @SqlResultSetMapping annotation. See Example 6-75.

Example 6-75 Dynamic using @SqlResultSetMapping

```
@SqlResultSetMapping(name = "AccountResults",
   entities = @EntityResult(entityClass =
com.ibm.itso.jpa.Savings.class))
public List findWithAccount (Long id) {
 Query query = em.createNativeQuery("SELECT * FROM ACCOUNT WHERE
ACCOUNT_ID = "+id.longValue(),"AccountResults");

 return query.getResultList();
}
```

Named queries

The main difference between named queries and dynamic queries is the use of annotations (@NamedQuery or @NamedNativeQuery). The named queries as dynamic queries has the same two types using JPQL(@NamedQuery) or using native SQL(@NamedNativeQuery). See Example 6-76.

Example 6-76 Named query using JPQL

```
@NamedQuery(
name="findWithAccount",
queryString="SELECT a FROM Account a" +
"WHERE a.accountId LIKE :accId"
)

@PersistenceContext public EntityManager em;
public List findWithAccount(Long id){
 Query query =
   em.createNamedQuery("findWithAccount");
 query.setParameter("accId", id);

 return query.getResultList();
}
```

In Example 6-77 you can see the same query using native SQL. Also, notice in this example, the type of Account in the result using @SqlResultSetMapping annotation.

Example 6-77 Named query using native SQL

```
@NamedNativeQuery(
name="findWithAccount",
query="SELECT * FROM ACCOUNT
 WHERE ACCOUNT_ID = ?",
resultSetMapping = "AccountResults")
)
@SqlResultSetMapping(name = "AccountResults",
   entities = @EntityResult(entityClass =
com.ibm.itso.jpa.Savings.class))

@PersistenceContext public EntityManager em;
public List findWithAccount(Long id){
 Query query =
   em.createNamedQuery("findWithAccount");
 query.setParameter(1, id.longValue());

 return query.getResultList();
}
```

Details of JPAQL capabilities can be found at the following link from Apache:

http://openjpa.apache.org/docs/latest/manual/manual.html#jpa_query_api

6.6.6 Best practices for scalability and improved performance

Following are some JPA best practices for scalability and improved performance.

Handling locking issues

This feature is an important issue to consider when planning for concurrent issues in our program. Understanding the approaches can guide you during development of an application. If necessary, refer to "Database isolation level" on page 370 and "Locking strategy" on page 371 for information on transaction types (Dirty read, Nonrepeatable read, Phantom read) and locking strategies (optimistic and pessimistic locking) before proceeding.

Optimistic locking and versioning of entities

In terms of resources usage, but only at the expense of reliability, optimistic transactions consume less resources than pessimistic/datastore transactions. Considering that optimistic transactions do not lock datastore records, two transactions might change the same persistent information at the same time, and the conflict is not detected until the second transaction attempts to flush or commit. Despite their drawbacks, the best choice for most applications are optimistic transactions. They offer better performance, better scalability, and lower risk of hanging due to deadlock.

If you have concurrency access to your entities or there are merging of detached entities (an instance is consider detached after a unit of work completed and the persistence context is closed), then the best choice is to enable optimistic locking for those entities. In JPA you sign this approach using a version attribute in an Entity class. This is mandatory in the specification. There are some requirements to do this, such as, we have to put @javax.persistance.Version on a field or property. Also, the persistence field has to be a numeric field such as Long, long, Integer or int. Moreover, this field has to be mapped to a database column in the primary table to which the entity is mapped. See Example 6-78.

Example 6-78 Optimistic locking configuration in annotations

```
@Entity
@Table(name="ACCOUNT")
public class Account implements Serializable {
 @Id
 @Column(name="ACCOUNT_ID")
 protected Long accountId;

 //...

 @Version
 @Column(name= "OPT_LOCK")
 private Long version;

 //...
}
```

Entity manager and locking modes

There are two types of locking modes, READ (LockModeType.READ) and WRITE (LockModeType.WRITE). This feature is used rarely and the EntityManage lock method does the explicit entity lock approach. However, the read locking can be used, for example, when you do reporting data, and the write locking can be used when no one can UPDATE or DELETE an entity when this lock is on, even if another separated transaction is trying to modify the locked write entity. See Example 6-79 and Example 6-80.

Example 6-79 LockModeType.WRITE

```
Author author = em.find(Author.class,authorId);
em.lock(author, LockModeType.WRITE);
author.addBook(book);
book.addAuthor(author);
```

Example 6-80 LockModeType.READ

```
em.lock(account, LockModeType.READ);
//...
Double amount = account.getMinamount();
System.out.println("Minimum amount report");
System.out.println("Name : "+account.getName()+ " Minumum amount: " +
amount.doubleValue());
```

Some tips to improve performance

The following sections briefly describe tips to gain performance in JPA. However, some tips can be considered generic no matter what persistence technology you are using.

Setting up the connection pool

If you do not set up the connection pool properly, this can have a negative impact on performance. Depending on the access volume, you can have more concurrent users than connections available. You have to measure with your capacity planning team which parameters best fit for your application such as MaxConnections, MinConnections, unused timeout, and so on.

Choosing the right inheritance

As we have seen in "Inheritance" on page 409, there are three strategies to implement inheritance. Depending on your point of view, each has advantages and disadvantages. However, for performance, the best approach is single-table because you avoid joins between tables.

Avoid using transactions for read-only queries

If you use a query in which results are not going to be updated, we recommend that you do not use a transaction because it is unnecessary.

Avoid using flush manually

The persistence provider in general optimizes the flush mode at the end of transaction, and by default the flush mode in EJB 3.0/JPA is AUTO. Avoid, if possible, the use of EntityManager's flush method. Its heavy use can be a performance drawback because using flush several times can lead to multiple SQL statements.

Using the best lock strategy

As EJB 3.0 specification says, pessimistic locking is optional and you should use this approach only if absolutely required because, depending on the database provider, you lock the record or page during the transaction. Using optimistic lock is preferred using the lowest lock (READ lock) if it satisfies your requirements.

Use read-only entities

We recommend, if possible, using read-only entities. In general, read-only entities are loaded by the persistence provider into the persistence unit cache and are not discarded. When this feature is used, the persistence provider does not have to calculate changes set and no clone or merge operations have to occur.

Use indexes in queries

This is a general best practice. This task is for your DBA team. Primary keys always have an indexed scan and no additional indexes are required. However, depending on your query design, you can use use additional indexes to improve performance.

Use named queries

As named queries are prepared once and are reused and cached by the persistence provider, we recommend using named queries instead of dynamic queries.

Avoid full table scan

Full table scans are detected, for example, when you use the “SELECT *...” sentence. Unless your table is very small, it consumes a lot of memory because it loads many rows that might not be required. Moreover, it causes a FULL TABLE SCAN in your database and results in a very slow query. Specify the fields that you actually require, and use SELECT * only as a last resort.

Consider redesigning your schema if possible

Do not hesitate to change your schema if you find some performance drawbacks. This might be possible if this schema is not used by other existing applications. For example, you can merge two small tables into one. On the other hand, you can divide a table into multiple tables. Consider a CUSTOMER table that has fields such as a joining date that is not used frequently and can be marked as lazy loaded. This could be a good candidate to create another table such as CUSTOMER_INFO and remodeling your domain model.

6.6.7 JPA adoption considerations

All this simplification does come with a slight cost. Due to the major shift in the programming model, existing EJB 2.x applications have to be rewritten to take advantage of the new JPA features. Also, the new JPA specification does not include automatic Container Managed Relationship (CMR) maintenance. Relationship maintenance is now application-managed.

There are some products that implement full EJB 3.0. Other products implement only JPA. Keep in mind that the goal of JPA is to be plugabble. This means that it would be plugabble even if an EJB container has a existing JPA installed. Another goal is that JPA can be used outside an EJB container.

6.7 iBATIS

Today we have several object relational mappers (ORMs) that come in different types. In Java, most of the popularity goes to those that implement full domain model mapping, the goal of which is to map a whole layer of objects and behaviors to database tables. Some popular ORMs include:

- Hibernate
- Java Data Objects (JDO)
- Java Persistence API (JPA, EJB Entities 3)
- EJB Entity Beans 2.x

Each of these is classified as a full domain model mapper, where tables are mapped to objects, object state is maintained, objects follow a connected model (as client components interact with the object, underlying database operations are implied) either all or some of the time, and an abstract query language works against the object model. In turn, these frameworks generate Java Database Connectivity (JDBC) or SQL code under the covers.

There are some cases, though, when you might decide to use straight JDBC instead. Some reasons might be:

- Developer's knowledge and comfort of SQL. Object query languages still do not relieve you from having to know SQL, since you often have to know how to fine tune query languages.
- Object relational mappers are heavyweight for certain types of applications. For example, batch applications that must execute many update operations serially are often better off executing SQL statement in sequence without all the extra object hydration.

- Stored procedures are required or already exist. There are many valid reasons for using stored procedures. In some scenarios, they reduce the amount of network IO because SQL statements can be executed in sequence at the database.
- The DBA is king. Many development organizations have strict rules about SQL and who might define it. Sometimes only database administrators are authorized to create and fine tune SQL for performance.
- Former environment. For example, applications are being migrated from a platform where SQL queries are already fined tuned and tested.

The general solution in these cases seems to be: Use JDBC. Developers frequently require guidance when building JDBC code; bad JDBC code often results in having scattered data access code all over the place. To achieve what they have to, developers often end up developing some kind of custom JDBC framework or wrapper. This can occur when, for example:

- Applications are running in a J2EE platform. Java objects still have to be passed as Data Transfer Objects from the business logic layer to the view layer, so there has to be some code that moves Result Set Data to Data Transfer Objects and from Data Transfer Objects to SQL updates, inserts, or deletes.
- Applications still want a layer of abstractions. Just because you use JDBC, it does not absolve you from having to layer code correctly.
- Applications want to externalize SQL from their Java code for fine tuning.
- Redundancy is inevitable. When writing JDBC, developers often find themselves writing the same mundane code over and over again such as acquiring connections, preparing statements, looping through results sets, and various other JDBC specific elements.

Solution

There is an object relational mapper that is currently available. Instead of creating a full domain model, its job is to map Java objects directly to SQL statements. This framework is called iBATIS, and its goal is to implement 80% of the JDBC boilerplate code you would otherwise have to do yourself. In addition, it provides a simple mapping and API layer that lets developers quickly develop data access code.

While the O/R approach offered by technologies such as JPA and Hibernate make available an automated persistence, on the other hand, SQL with JDBC offers a hands-on approach allowing more control and flexibility. For these cases, iBATIS offers a good choice for abstracting low-level jobs while keeping intact the SQL/database centric of persistence.

6.7.1 iBATIS basic concepts

iBATIS is an open source object relational mapper whose job is to map objects to SQL statements. Using a simple concept called SQL maps, the goal is to map Java objects to SQL statements.

Simply put, iBATIS consists of two separate frameworks. You use the Data Mapper framework specifically for O/R mapping, which is the mapping of your Java domain objects to relational tables in a database. The DAO framework gives your application a clean and consistent way to access underlying data.

iBATIS Data Mapper framework (Data Mapper)

The Data Mapper is the framework that executes your SQL and maps the results back to objects, saving you from having to do this manually.

The Data Mapper framework does not require you to make any special version of your Java objects. You do not have to implement any interfaces or generate any code. You do not have to subclass some other base object or perform any strange rituals. And you do not have to learn a secondary query language specific to the framework.

You use a simple and straightforward XML format to define the manner in which iBATIS maps your Java objects to the database. You can define the exact query you require directly in SQL and optionally use any proprietary SQL that is specific to the database engine you're using. This capability lets you map your objects exactly the way you want and perform joins exactly the way you want.

iBATIS Data Access Objects framework (DAO framework)

The DAO framework's main goal is to abstract the how and where of your application's data-access or persistence layer from the application's business logic. The DAO framework lets you define interfaces in your application that are responsible for data-centric operations.

For example, if your application uses straight-up Java Database Connectivity (JDBC) for persistence, the DAO framework's goal is to abstract the use of classes and interfaces, such as Connection, PreparedStatement, and ResultSet, away from your application and move it down into a persistence layer instead.

If your application for some reason uses HTTP GETs and POSTs to get and store data, then the DAO framework's purpose becomes to abstract the use of classes, such as HttpUrlConnection, away from your application's business layer. Your application can then use the DAO interfaces to perform operations on your data, and the implementations of these interfaces are abstracted away from your business logic. The implementations can retrieve data from a database, a Web service, or any other source.

The DAO framework does not depend on the use of the Data Mapper framework. You can use both frameworks in a project should you choose (and they pair quite nicely), or you can use each one independently. This tutorial series shows the advantages of using the frameworks alone and together.

Advantages of iBATIS

iBATIS has some advantages over other O/R mapping tools:

- iBATIS does not use its own proprietary query language; it just uses SQL. Some O/R mapping tools, such as Hibernate, use their own query languages in addition to SQL.
- All the queries and updates you want to perform are written in SQL (and stored in .xml files). Some people might consider this a disadvantage, wanting the database abstracted from them completely to avoid having to write any SQL code. This is one reason a lot of developers like Hibernate. But you might prefer to have fine-grained control over exactly what SQL is being executed when you access your objects, rather than having it unpredictably generated for you in a manner dependent on the underlying O/R mapping framework. You can fine-tune your queries and other statements based on recommendations by a database administrator (DBA) or by access plans or query optimizers provided by the tools supplied with your Relational Database Management System (RDBMS). Another benefit of having direct access over the SQL that is written for this layer is that you can take advantage of any proprietary SQL offered by your database.
- iBATIS is easy to use.
- The project is well documented.
- It has no external dependencies. Some of the other O/R mapping frameworks ship with 15 to 20 .jar files and are dependent on specific versions of these files just to let the framework run. You do not want that kind of a headache when developing applications, so the fact that you can use iBATIS without any external dependencies is a huge plus. (Note that some optional configurations let you enable things like an external connection pool or bytecode enhancement, but none of them are required.)

6.7.2 iBATIS Data Mapper framework

In the following sections we talk about these topics that are core of the iBATIS Data Mapper:

- Mapped Statements
- Parameter Maps and inline parameters
- Result Maps
- TransactionManager
- SQLMap example

Mapped Statements

The Data Mapper's core functionality revolves around the concept of Mapped Statements. A Mapped Statement can have what are called Parameter Maps (basically, data input) and Result maps (data output). So a Mapped Statement is essentially an XML element that contains an SQL statement responsible for performing some action and mapping input/output parameters to Java objects. Example 6-81 shows a simple SQL Mapped Statement

Example 6-81 Simple SQL Mapped Statement

```
<select id="getUsernameList"
        resultClass="string"
        parameterClass="account">
     select USERNAME as value from CUSTOMER_INFO
</select>
```

The Mapped Statement in the foregoing example is responsible for querying for all values of the USERNAME column from the CUSTOMER_INFO table. There are several different types of Mapped Statements. As you can see, this particular mapped statement is a <select>. In addition to <select>, you can take advantage of <statement>, <insert>, <update>, <delete>, and <procedure> Mapped Statement elements when using the iBATIS framework. To see more details of Mapped Statements, go to the following link:

http://iBATIS.apache.org/

Parameter Maps and inline parameters

Parameter Maps in the iBATIS framework provide data-input parameters to a Mapped Statement. Parameter Maps are not often used. Otherwise, inline parameters are more frequently used. Example 6-82 shows how a Parameter Map and a Mapped Statement are used.

Example 6-82 Parameter Map usage

```
<parameterMap id="insert-product-param" class="com.domain.Product">
  <parameter property="id" jdbcType="NUMERIC"
         javaType="int" nullValue="-9999999"/>
  <parameter property="description" jdbcType="VARCHAR"
         nullValue="NO_ENTRY"/>
</parameterMap>

<statement id="insertProduct" parameterMap="insert-product-param">
   insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</statement>
```

We can see that the Mapped Statement, *<statement>* tag, in Example 6-82 references the Parameter Map by name, called *insert-product-param*, and that it contains two placeholder question marks. We recognize these as standard placeholders for JDBC PreparedStatements. It applies the values it retrieves from the Parameter Map, in the order in which they are defined, to these placeholders.

The Parameter Map in Example 6-82 on page 426 defines that the com.domain.Product class's id property — getId() — maps to the first placeholder (question mark) in any Mapped Statement that uses this Parameter Map. It goes on (with the next parameter element) to state that the com.domain.Product class's description property — getDescription() — maps to the second placeholder, question mark, in any Mapped Statement that uses it. Within a parameterMap, the order in which parameter elements appear is the same order in which they are applied to the placeholder question marks within the Mapped Statement that uses the parameterMap.

More commonly, input parameters are mapped with inline parameters. See Example 6-83.

Example 6-83 Inline parameter usage

```
<statement id="insertProduct"
parameterClass="com.ibm.itso.ibatis.Product">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
  values (#id#, #description#);
</statement>
```

This syntax replaces #id# with the value returned by getId() from the com.ibm.itso.ibatis.Product class, while #description# is replaced by the value returned by getDescription() of com.domain.Product. You can take a look at the iBATIS documentation for how to specify null values.

Result Maps

Result Maps are like Parameter Maps but are used for output. Result Maps let you define the manner in which you would like your Mapped Statements, typically queries, mapped back into a Java object. Example 6-84 provides a quick look at an example from the iBATIS documentation.

Example 6-84 Result Maps example

```
<resultMap id="get-product-result" class="com.domain.Product">
 <result property="id" column="PRD_ID"/>
 <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>
<statement id="getProduct" resultMap="get-product-result">
```

```
  select * from PRODUCT
</statement>
```

As we can see that the Mapped Statement with the id of getProduct specifically references the Result Map entitled get-product-result and tells the Mapped Statement to map the PRD_ID database column to the Java id property of the com.ibm.itso.ibatis.Product class, and also states to map the PRD_DESCRIPTION database column to the Java description property of the com.ibm.itso.ibatis.Product class.

It is a good practice always to specify the exact columns that we are selecting rather than use SELECT * FROM, for example.

TransactionManager

The TransactionManager element within the Data Mapper framework lets you configure what you want the transaction services to be like for a given configuration. The currently supported types for this element are:

- JDBC: The JDBC transaction manager controls transactions internally via Initial configuration, semantics, and a simple test the java.sql.Connection interface's commit() and rollback() methods.
- JTA: A global Java Transaction API (JTA) transaction is used and requires a UserTransaction to be available via the Java Naming and Directory Interface (JNDI), or whatever other means.
- EXTERNAL: You manage transactions on your own. This is also a fine option for nontransactional data sources for which you must manage any transactions yourself anyway.

SQLMap complete example

See the complete SQLMap in Example 6-85, which represents a complete configuration.

Example 6-85 SQLMap complete example

```
<sqlMap namespace="Product">
   <typeAlias alias="product"
       type="net.humandoing.invoicing.domain.Product"/>
   <resultMap id="productResult" class="product">
    <result property="productId" column="PRODUCT_ID"/>
    <result property="productName" column="PRODUCT_NAME"/>
    <result property="productDesc" column="PRODUCT_DESC"/>
    <result property="quantity" column="QUANTITY"/>
   </resultMap>
   <update id="updateProduct" parameterClass="product">
```

```
      update PRODUCT set PRODUCT_NAME = #productName#,
      PRODUCT_DESC = #productDesc#,
      QUANTITY = #quantity# where
      PRODUCT_ID = #productId#
   </update>
   <select id="getProduct" resultMap="productResult"
    parameterClass="product">
    select PRODUCT_ID, PRODUCT_NAME, PRODUCT_DESC, QUANTITY
    from PRODUCT where PRODUCT_ID = #productId#
   </select>
   <insert id="insertProduct" parameterClass="product">
    insert into PRODUCT (product_id, product_name, product_desc,
quantity)
    values (#productId#, #productName#, #productDesc#, #quantity#)
   </insert>
</sqlMap>
```

6.7.3 SQL map config file

SqlMapConfig.xml is the deployment descriptor for SQLMaps. See Example 6-86.

Example 6-86 SqlMapConfig file

```
<sqlMapConfig>
    <settings useStatementNamespaces="false" />
    <transactionManager type="JDBC">
        <dataSource type="SIMPLE" >
            <property name="JDBC.Driver"
                value="COM.ibm.db2.jdbc.app.DB2Driver"/>
            <property name="JDBC.ConnectionURL"
                value="jdbc:db2:SAMPLE"/>
            <property name="JDBC.Username"
                value="db2admin"/>
            <property name="JDBC.Password"
                value="admin2db"/>
        </dataSource>
    </transactionManager>
    <sqlMap resource="Customer.xml"/>
</sqlMapConfig>
```

Analyzing the foregoing example, we find it contains the following elements:

- <sqlMapConfig> is the root element of the file. The <settings> element is used for defining application-level settings. The useStatementNamespaces defines whether you want to use the fully qualified name of the prepared statement.
- The <transactionManager> element is used to define what kind of transaction management you want to use in your application. We are using JDBC as the transaction manager. It contains <dataSource> as a child element, which defines the type of Connection management you want to use. Also we are using a data source of type SIMPLE. SQLMaps requires information such as the JDBC driver name, URL, and password in order to create the connection pool, so we are using <property> elements for passing that information.
- The <sqlMap> element is used to declare sqlmap config files. These files, as we discussed in "iBATIS Data Mapper framework" on page 425, list the SQL queries that you wish to execute.

Now we discuss the related elements.

Related elements from SQLMap config

From the example of SQLConfig map file in Example 6-86 on page 429, we have to create a Customer.xml file where we list all Contact-table-related SQL queries that we want to execute. See Example 6-87.

Example 6-87 Customer.xml file

```
<sqlMap namespace="Customer">
    <typeAlias alias="customer"
        type="com.ibm.itso.ibatis.Customer"/>
    <select id="getContact"
        parameterClass="int" resultClass="customer"">
            select CUSTOMERID as customerId,
                   FIRSTNAME as firstName,
                   LASTNAME as lastName,
                   TITLE as title from
                   ITSO.CUSTOMER where CUSTOMERID = #id#
    </select>
</sqlMap>
```

We also have to create a Java bean for Customer as described in Example 6-88.

Example 6-88 Customer.class

```
public class Customer {
  protected Integer customerId;
  protected String firstname;
  protected String lastname;
  protected String title;
  //gets and sets methods
}
```

Now we can consider the usage of these elements in Java as shown in Example 6-89.

Example 6-89 Simple Java code usage of SQLMapConfig and resources related

```
Reader configReader =
  Resources.getResourceAsReader("SqlMapConfig.xml");
SqlMapClient sqlMap =
  SqlMapClientBuilder.buildSqlMapClient(configReader);
Customer customer = (Customer)
  sqlMap.queryForObject("getCustomer",new Integer(1234));
```

To use the SELECT query described in Customer.xml, we use queryForObject. Also we have specified int as parameterClass class passing contactId as an integer, along with the name of the query using getCustomer. SQLMaps then returns an object of the Customer class.

6.7.4 The DAO framework

The DAO framework gives you a consistent way to access your application's underlying data structures. It abstracts how data is accessed and lets your business-logic tier communicate with a consistent and predictable set of APIs instead of worrying about the semantics of accessing data via JDBC or HTTP. In essence, the DAO framework acts as a proxy or a facade between your application's business-logic tier and its data tier.

To use the DAO framework, implement the com.ibatis.dao.client.Dao interface. This is just a marker interface that does not contain any methods. Rather, it acts as an identifier to the DaoManager (part of the DAO framework) as an object that can be used to access data.

> **Note:** However, our recommendation is to be careful about the use of this feature because the DAO framework is deprecated after iBATIS 2.3.0. The iBATIS site says that it is to be available separately.

6.7.5 Transactions in the Data Mapper framework

Within the iBATIS Data Mapper framework, the sql-map-config.xml file that you use to configure the Data Mapper contains a <transactionManager/> XML element. Built into the Data Mapper are three types of transaction managers: JDBC, JTA, and EXTERNAL as we have seen in “TransactionManager” on page 428. The JTA transaction manager provides transactions through the use of the Java Transaction API (JTA). To use JTA transactions, you must tell the <transactionManager /> element where it can find a UserTransaction implementation in the Java Naming and Directory Interface (JNDI).

You use the EXTERNAL transaction manager if you want to manage transactions on your own for example, if you have a nontransactional data source, such as flat files. We cover only the JDBC transaction manager in Example 6-90 using the configuration of the Data Mapper < transactionManager /> element.

Example 6-90 TransactionManager element configuration in DataMapper

```
<transactionManager type='JDBC'>
 <dataSource type='SIMPLE'>
   <property value='${driver}' name='JDBC.Driver'/>
   <property value='${url}' name='JDBC.ConnectionURL'/>
   <property value='${username}' name='JDBC.Username'/>
   <property value='${password}' name='JDBC.Password'/>
   <property value='15' name='Pool.MaximumActiveConnections'/>
   <property value='15' name='Pool.MaximumIdleConnections'/>
   <property value='1000' name='Pool.MaximumWait'/>
 </dataSource>
</transactionManager>
```

The JDBC transaction manager manages transactions by turning off auto commit and using the java.sql.Connection interface's commit() and rollback() methods. The only nested elements within the <transactionManager.../> element belongs to the configuration of the database connectivity properties and the connection pool. This demonstrates how easy it is to set up the Data Mapper to deal with JDBC-style transactions for you. Now we see how a transaction might look in the Java language, as shown in Example 6-91.

Example 6-91 Simple example of Transaction in iBATIS/Java

```
//In some part of your program instantiate a SqlMapClient

 Reader reader =
Resources.getResourceAsReader("properties/sql-map-config.xml");
 SqlMapClient client = SqlMapClientBuilder.buildSqlMapClient (reader);
```

```
//In another method execute some transaction

int amount = 50;
Account acctFrom = (Account)
              sqlMap.queryForObject("getAccount",new Integer(1234));
Account acctTo = (Account)
              sqlMap.queryForObject("getAccount",new Integer(1335));

try {
  sqlMap.startTransaction()
  acctFrom.setAmount(new Integer(acct1.getAmount()-amount));
  acctTo.setAmount(new Integer(acct1.getAmount()+amount));
  sqlMap.update("updateAccount", acctFrom);
  sqlMap.update("updateAccount", acctTo);

  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}
```

You can see in the foregoing coding that sqlMap.startTransaction() does exactly what you expect it to: It starts a transaction. sqlMap.commitTransaction() call commits the transaction. After that, you also want to look at the finally{...} block sqlMap.endTransaction() that cleans up. You must always make the call to endTransaction(). That is why you put it in a finally block. If the transaction was not committed previously, endTransaction() rolls back the transaction. No matter what, it is also responsible for cleaning up the underlying resources, such as ResultSet and PreparedStatement, as well as closing your Connection object or returning it to the pool.

6.7.6 Caching in iBATIS

Caching is critical for large applications to scale under heavy load. Within iBATIS, caching specifically relates to the Data Mapper framework, so you configure it in data map definition XML files. We can then apply a given cache configuration to a given Mapped Statement. This section describes the different types of caching the Data Mapper framework provides.

Cache types

The iBATIS framework offers a myriad of cache types. They include:

- A memory-based cache that is managed by the Java Virtual Machine's garbage collector

- Least Recently Used (LRU)
- First In, First Out (FIFO)
- Various caching options available in the iBATIS site documentation:

  http://ibatis.apache.org/

Under the LRU algorithm, the object that has been used the least is the first object to be expelled from the cache when the cache reaches its maximum capacity. Example 6-92 shows a cache model definition, which is an LRU cache in SQLMap file that is flushed every 24 hours and has a maximum size of 1,000 objects.

Example 6-92 LRU cache model definition in SQLMap file

```
<cacheModel id='product-cache' type='LRU'>
 <flushInterval hours='24'/>
 <flushOnExecute statement='insertProduct'/>
 <flushOnExecute statement='updateProduct'/>
 <flushOnExecute statement='deleteProduct'/>
 <property name='size' value='1000' />
</cacheModel>
```

To change this cache to a FIFO cache, you only have to change the type attribute from LRU to FIFO. In a FIFO cache, the object that has been in the cache for the longest period of time is expelled from the cache when it reaches capacity. The cache model XML snippet in the foregoing example also indicates that you wish to have the cache flushed any time the Mapped Statement with an ID of insertProduct, updateProduct, or deleteProduct has been executed. This helps to minimize or eliminate problems that are due to stale data creeping into the cache.

For more advanced configuration information and discussion of caches, you can check out the iBATIS documentation.

6.7.7 Dynamic SQL in iBATIS

You might run into a situation where a static PreparedStatement does not do what you require, and you have to generate ad hoc SQL dynamically. By taking advantage of dynamic SQL, you can write, say, five to 10 lines of XML rather than 200 lines of difficult-to-maintain if/else Java code.

Dynamic mapped statements

Suppose your Web application has a search form to let you search for accounts and that you can enter multiple search criteria. Perhaps you are on the phone with a customer who has forgotten his or her account ID. This is not a problem, because your search screen lets you type in the customer's account ID, first name, last name, or e-mail address, and all combinations and permutations of those pieces of data, to retrieve the account record. Using iBATIS, you can write a Mapped Statement that dynamically constructs the SQL based on whatever parameters are present (and even the values of those parameters).

For more information, see the iBATIS documentation.

http://ibatis.apache.org/

6.8 Java Data Objects

The Java Data Objects (JDO) API is a standard interface-based Java model abstraction of persistence to directly store Java domain model instances into a data store. JDO is developed as a part of a Java Specification Request. With JDO, developers can easily access persistent data that can be stored in various types of back-ends, such as databases, file systems or other transaction processing systems. Similar to Service Data Objects, Java Data Objects also provide a common API to simplify and unify the data access.

The main difference between SDOs and JDOs, however, is that JDOs only solve the persistence issue, whereas SDOs use a more general approach that also includes data representation and data flow between the J2EE tiers. Compared to EJBs, most of the former advantages of JDOs more or less disappeared with the introduction of local interfaces (EJB 2.0). At the moment it is questionable if the Java Community Process intends any further investigation into JDO 2.0, because of the fact that it apparently overlaps with existing Java technologies and with the EJB 3.0 specification. Nevertheless, currently there are a lot of open source frameworks available that implement JDOs.

6.8.1 Advantages: JDO

Here are some advantages of JDO:

- Universal data access for different kinds of data sources
- Transparent persistence layer, full transaction support, such as CMP EJBs
- Good performance, even for large amounts of data
- Lightweight technology — JDO is based on Java objects

There is no EJB container necessary for the “Entity Beans” themselves, but you might want to implement the business logic using EJB session beans.

6.8.2 Disadvantages: JDO

Here are some disadvantages of JDO:

- No built-in security (compared to EJBs).
- Not part of J2EE specification.
- Unsure future mainly with EJB 3.0 specification
- No built-in support for JDO in WebSphere Application Server.
- Limited tooling support available.

6.8.3 Alternatives: JDO

Here are some alternatives to JDO:

- EJB 3.0 with Java Persistence API (JPA)
- Stateless Session Beans with iBATIS, if the first alternative is not available.
- For more information about JDO, see the following URLs:

  http://www.ibm.com/developerworks/db2/library/techarticle/dm-0506bhogal/

  https://www6.software.ibm.com/dw/education/dm/dm0604balani/

6.8.4 Best practices

With EJB 3.0 specification and JPA adoption, we not recommend use of this technology. Even if you have existing projects with JDO technology, we recommend migrating to technologies aligned with J2EE specification.

6.9 Service Data Objects

Service Data Objects (SDOs) is a data programming architecture — in contrast to all other technologies currently available — that unifies data programming across data source types, helping to simplify and unify data access across different data source types, which is becoming more and more crucial in the IT industry. Monolithic applications are not built as often anymore and information is often stored in various types of sources (database, Web service, Lightweight Directory Access Protocol (LDAP), legacy, and so forth). This complexity requires developers to become skilled in many APIs — Java database connectivity (JDBC), Java APIs for XML-Based Remote Procedure Call (JAX-RPC), or J2EE Connector Architecture (JCA), to name a few.

SDO simplifies and complements the Java 2 Platform, Enterprise Edition (J2EE) development mode, providing one unique API to access heterogeneous data sources including relational databases, XML data sources, Web Services, and Enterprise Information Systems. Now we go on to SDO objectives in more depth.

6.9.1 SDO objectives

The goals of SDO are numerous. Basically, there are five main topics in SDO and related technologies address. We discuss these in the next sections.

Data access simplification

The first goal is to provide uniform data access to a wide variety of Enterprise Information Systems (EIS). This comprises databases, legacy (using JCA), XML, or Web Services sources. By using a unique and simple model, applications get rid of the complexity of several data access APIs and frameworks with SDO.

To realize this simple and unified way to handle data, Service Data Objects add a new abstraction layer that is placed on top of existing data access frameworks like EJBs, JDO, or direct JDBC data access. Therefore, Service Data Objects does not replace existing frameworks, instead, they use them as data mediators under the covers. In fact, Service Data Objects are becoming a standard way to implement the Business Delegate and Data Transfer Object patterns (Figure 6-24).

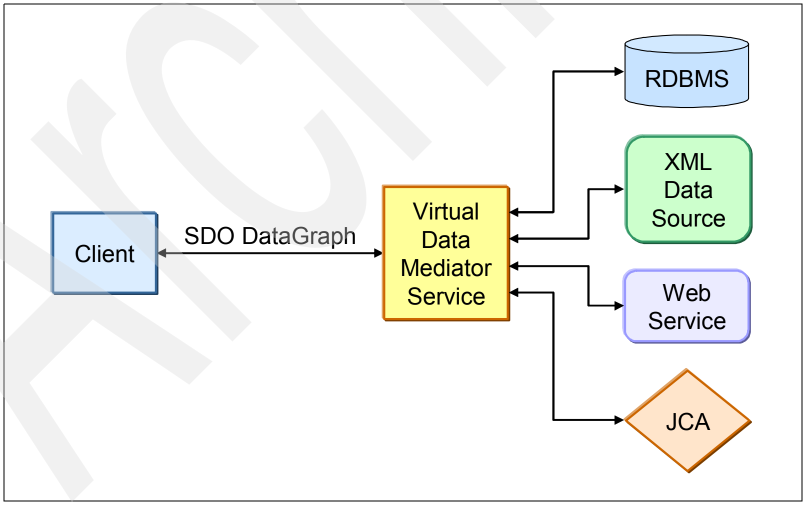


Figure 6-24 Flexibility of Service Data Objects

Data abstraction

Data representation is independent from its source using SDO. This is an implementation of a J2EE pattern called Domain Store. This level of abstraction has several advantages, such as making data manipulation easier and promoting loose coupling between different layers.

Data manipulation

Once the information is retrieved, SDO also wants to offer a uniformed programming language for data manipulation. In short, using the API and its interfaces, an SDO client must be able to read data and perform changes. SDO features both a connected and disconnected model.

Data transport

One part of the SDO concept is based on the Transfer Object and the Transfer Object Assembler patterns. SDO objects are independent from the underlying data sources. They encapsulate information in a Plain Old Java Object (POJO) and they are not related to specific technologies like JPA, JDO, EJB or Servlet.

Consequently, SDO objects are perfect candidates to cross tiers in a J2EE architecture. They can be the by-value Java objects that are created by the integration layer before being sent to the business layer. Additionally, an SDO object can be used to carry information between the presentation layer and the business layer.

Figure 6-25 shows Service Data Objects usage between layers.

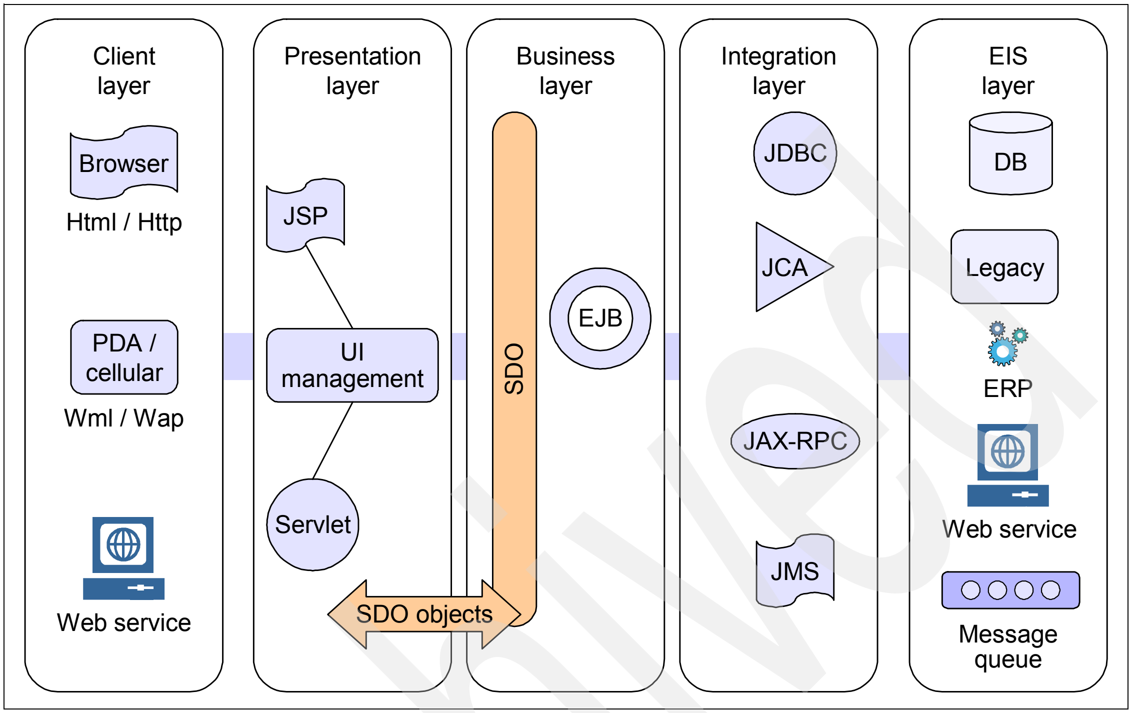


Figure 6-25 SDO usage between J2EE layers

Design patterns adoption

A key objective of SDO is to also encourage the adoption of common J2EE patterns. That is why SDO architecture is based on well-known patterns such as Transfer Object, Data Access Object, Transfer Object Assembler, and Domain Store (see Resources). Using SDO, an application takes advantage of these proven design strategies. It fosters layering and loose coupling.

> **Note:** For more information about design patterns related to SDO, see Chapter 2, “Application planning and design” on page 23.

6.9.2 SDO architecture

The SDO architecture (Figure 6-26) consists of three major components:

- Data object
- Data graph
- Data mediator

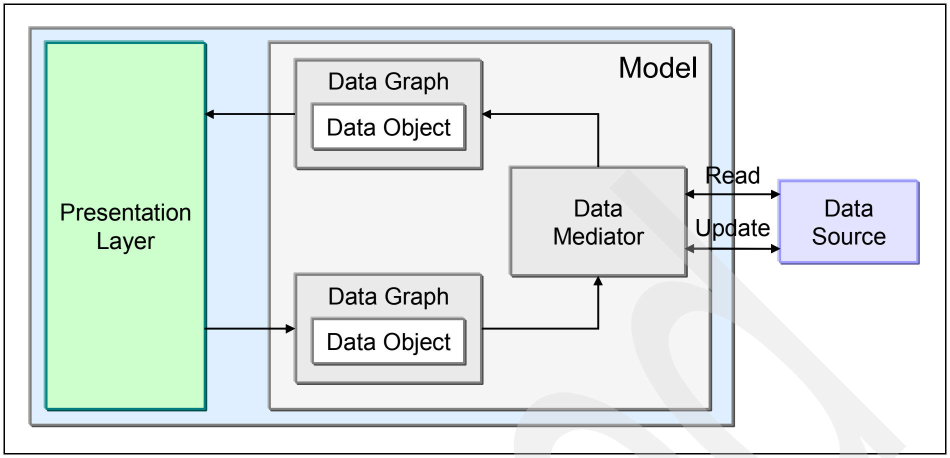


Figure 6-26 SDO architecture

Data object

The data object is designed to be an easy way for a Java programmer to access, traverse, and update structured data. Data objects have a rich variety of strongly and loosely-typed interfaces for querying and updating properties. The implementation of the data object also handles data conversions if required. Data objects store the data using a disconnected, optimistic model, meaning the data is available locally without an active connection to the EIS. Therefore, the data object can be easily used to transfer data between the different application layers. This enables a simple programming model without sacrificing the dynamic model required by tools and frameworks. A data object can also be a composite of other data objects.

Data graph

SDO is based on the concept of disconnected data graphs. A data graph is a collection of tree-structured or graph-structured data objects. Under the disconnected data graphs architecture, a client retrieves a data graph from a data source, mutates the data graph, then applies the data graph changes to the data source. The data graph also contains some metadata about the data object including change summary and metadata information. The metadata API allows applications, tools, and frameworks to introspect the data model for a data graph, enabling applications to handle data from heterogeneous data sources in an uniform way.

Data mediator

The task of connecting applications to data sources is performed by a data mediator. Client applications query a data mediator and get a data graph in response. Client applications send an updated data graph to a data mediator to have the updates applied to the original data source. This architecture allows applications to deal principally with data graphs and data objects, providing a layer of abstraction between the business data and the data source.

> **Important:** Update processing is not dependent on how the DataGraph was originally retrieved. In other words, it is possible to retrieve a DataGraph directly from the data source but to have the deferred updates applied through an EJB or vice versa.
>
> Regardless of which update approach you use, an optimistic concurrency control algorithm is used. Fields designated as consistency fields are read during update to insure that the current value is still equal to the old value of the field in the DataObject.

This means that mediators are components that provide access to a specific data source type. For example, a Siebel® mediator knows how to mediate between changes made to an SDO and the necessary calls to the Siebel API to persist changes to the underlying Siebel records.

Disconnected data architecture

One interesting feature of the SDO model is that it allows a disconnected programming model. As a matter of fact, when an SDO client claims a Data Graph and receives it, it is then disconnected from the DMS. This prevents the DMS from holding locks on the data source. The client can work with the Data Objects without any time constraints, and the changes are applied back to the data source using an optimistic concurrency scenario.

This disconnected model is particularly adapted to n-tier Web-based architectures, because it respects layering techniques, provides ease of use, and a high level of concurrency access.

Enterprise JavaBeans Data Mediator Service

The Enterprise JavaBeans (EJB) Data Mediator Service (DMS) is the Service Data Objects (SDO) Java interface that, given a request in the form of EJB queries, returns data as a DataGraph containing DataObjects of various types. This differs from a normal EJB finder or `ejbSelect` method, which also takes an EJB query but returns a collection of EJB objects (all of the same type) or a collection of container managed persistence (CMP) values.

> **Note:** For more information on this service, go to the following URL.
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/rejb_ejbmedpcon.html

JDBC Data Mediator Service

The Java Database Connectivity (JDBC) DMS is the SDO component that connects to any database that supports JDBC connectivity. It provides the mechanism to move data between a DataGraph and a database. A regular JDBC call returns a result set in a tabular format. This format does not directly correspond to the object-oriented data model of Java, and can complicate navigation and update operations. When a client sends a query for data through the JDBC DMS, the JDBC result set of tabular data is transformed into a DataGraph composed of related DataObjects. This enables clients to navigate through a graph to locate relevant data rather than iterating through rows of a JDBC result set.

> **Note:** For more information on this service, go to the following URL.
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/cdat_jdbcmed.html
>
> For best practices usage of JDBC DMS, see the following links.
>
> http://www.ibm.com/developerworks/webservices/library/ws-bestjdbc/index.html
> http://www.ibm.com/developerworks/webservices/library/ws-bestjdbc2/index.html

6.9.3 Others SDO perspectives

So far, an SDO appears to be much more than an API. It is also a design and programming model. That is why SDO can be involved in multiple enterprise application concepts. Let us take a look at some of them now.

Persistence mechanisms

SDO is not meant to replace existing persistence mechanisms, but instead to leverage their use providing a uniform programming interface. Instead of learning multiple APIs and frameworks, a programmer typically concentrates on one unique programming model (SDO). Behind the scenes, SDO-capable tools and DMS deals with all the specific and cumbersome data source semantics. So without even knowing it, an SDO client, through DMS, could interact with XML, JDBC, Java Data Objects (JDO), Hibernate, Entity Enterprise JavaBeans (EJBs), Web Services, or any other data source.

XML document access

Since that XML represents another form of data access that can be standardized by SDO, the SDO 2.0 specification had several improvements to create and then read an XML document compliant to XML Schema (XSD). To accomplish the same goal without using the SDO 2.0 would require the developer to understand how the XML parser works and tightly integrate the data parsing logic with the application. Later, if the XSD has to change, the application would have to be touched everywhere, which would jeopardize the quality of the code. This is true for solutions such as JAXB or XMLBeans.

The following link includes a good example of using this SDO feature:

http://www.ibm.com/developerworks/webservices/library/ws-sdoxmlschema/

IBM tools

SDO is not just a specification. You can find tools that leverage SDO technology to access a heterogeneous EIS.

WebSphere Application Server Version 6.x, which is J2EE 1.4 compliant, provides a set of programming model extensions to fulfill specific enterprise requirements not yet covered by the specification. In order to foster SOAs, Application Server Version 6.x supports SDO and provides some DMS implementations.

There has been some support for SDO in development world since WebSphere Studio Application Developer 5.1.2. Rational Application Developer V6 and V7 provide support for developing to WebSphere 6.0 and WebSphere 6.1 SDO features respectively.

Open source runtimes and tools

There is an open source project that provides runtime implementations of Service Data Objects, which you can use to build applications and which has some basic tools which assist the use of SDOs. This project is called Tuscany, currently under incubation at Apache. See the Tuscany Web site at Apache:

http://incubator.apache.org/tuscany/

There is also an Eclipse open source project that aims to provide tools to enable developers to build solutions using a service oriented architecture, which uses Service Component Architecture as its core model. This is the Eclipse SOA Tools Platform project, which you can find in the following link:

http://www.eclipse.org/stp/

Service Oriented Architecture

Service Oriented Architecture (SOA) is an industry-standard framework that is interchangeable, adaptive, and flexible. It is all about on demand business; however, SOA is just a concept or blueprint for IT infrastructure. The industry has already adopted Web Services standards to realize SOA applications. A Web service represents a self-contained and self-describing piece of functionality that can be found and accessed by other applications using open standards. The Java community can rely on a set of APIs and technologies to publish, discover, or consume services. It means you can interact with external applications using well-defined protocols like Simple Object Access Protocol (SOAP).

Services Component Architecture: Overview

Services Component Architecture (SCA) fills a major gap in SOA standards; it defines a comprehensive model and architecture for building, composing, and deploying SOA applications on heterogeneous platforms. SCA enables peer-to-peer interactions between services in a distributed SOA architecture. While benefits of SOA have been extensively marketed and recognized as valuable, when it comes to materializing an SOA based IT landscape, the current standards such as Web Services and J2EE are either not sufficient or just too complex.

Web Services define a rich technology stack aimed at wire level interoperability but have very little to offer regarding creation and composition of SOA applications. SCA attempts to fill this gap. It defines a broad suite of specifications that address various different aspects of complex SOA environments including programming of reusable components in various different languages, configuring the components with scenario specific settings, assembling the components into high level composites, and so on.

Web Services standards primarily involve the wire level interoperability of endpoints and does not focus on how the applications behind the endpoint are constructed. One of the major benefits of SOA is reuse. While Web Services enable ubiquitous access of endpoints and promote reuse of endpoints, Web Services do not provide much help when it comes to reuse of implementation artifacts, and so on.

In SOA applications, SCA fills a critical gap and compliments Web Services. Now that we understand what SOA and SCA are, next we explain SDO usage in the SOA world.

SDO usage for SOA

As we explained in the previous section, Web Services does not address all requirements for SOA. SCA helps in some part, however, there is no standard way to carry information within your application. Of course, this can be achieved by developing your homemade Java objects or taking advantage of XML binding frameworks such as JAXB, Castor, XMLBeans, or any other solution from a plethora of technologies available today.

SDO complements the strength that SCA offers for simplifying development of SOAbased solutions. SCA handles the composition of service networks and SDO focuses on simplifying data handling. SDO provides flexible data structures that allow data to be organized as graphs of objects (called data objects) that are composed of properties. SDO delivers unified and consistent access to data from heterogeneous sources. This provides both a simple programming model for the application programmer and lets tools and frameworks work consistently across those heterogeneous data sources. SDO offers a single model for data across the enterprise.

6.9.4 Advantages: SDO

Service Data Objects have the following advantages:

- Uniform access to data across heterogeneous sources:

  As we said, Service Data Objects can access data from a variety of sources, including relational databases, custom data access layers, Web Services, XML data stores, JMS messages, and Enterprise Information Systems.

- Becoming a standard way to implement the Business Delegate/DTO:

  Actually, Service Data Objects are basically Data Transfer Object (DTOs). The Data Mediator Services are part of the specification, but they are not a standard yet. Thus, the Business Delegate with a DTO is the key pattern.

- Support for disconnected programming models:

  Many presentation frameworks, such as Struts or JavaServer Faces, use a disconnected usage pattern of data access. They use some kind of Data Transfer Object, to pass application data between the layers. Service Data Objects perfectly support this model, the disconnected data objects required are automatically generated and an optimistic concurrency model is used.

- Service Data Objects support both static and dynamic data APIs:

  Static data APIs are much easier to use and therefore preferred by application programmers. In some cases however, static Java interfaces for data are not sufficient, for example when it comes to dynamic queries where the shape of the resulting data is not known.

- Good tooling support available for Service Data Objects:

  Although Service Data Objects are very flexible, development tools can easily support them because they provide simple introspection APIs. Also, Service Data Objects can easily be integrated into existing presentation frameworks.

- A good pattern for developing SOA (Service Oriented Architecture):

  Besides WebServices as a key technology for SOA, SDO allows to abstract access to heterogeneous resources being a value technology for SOA demanding. It is true too because as the SDO API concepts are programming language neutral and there is another implementations such as C++ and PHP for example. This feature for SOA to integrate resources is very valuable.

6.9.5 Disadvantages: SDO

But there are also some disadvantages to using Service Data Objects:

- Performance overhead:

  Service Data Objects add another layer on top of existing persistence or data access frameworks, which on the one hand, increases flexibility and simplifies integration of heterogeneous data sources. Even with SDO adopting J2EE best practices patterns in your building, this approach has a cost: It adds some performance overhead.

- Not part of J2EE Specification:

  Service Data Objects are not part of any J2EE Specification. IBM and BEA Systems submitted a Java Specification Request - JSR 235 in December 2003. However, SDO was submitted besides SCA to OASIS (which is concerned with the SOA specification).

6.9.6 Best practices: SDO

If you are developing a traditional (non-SOA) application, you only have relational data, and you are only developing in Java, then EJB 3.0 is a good choice.

If you are developing using a SOA, if you have to access multiple types of data, then SDO is a good choice.

6.9.7 Resource information

- JSR 235: Service Data Objects:

  http://www.jcp.org/en/jsr/detail?id=235

- Introduction to Service Data Objects:

  http://www.ibm.com/developerworks/java/library/j-sdo/

- SDO specifications:

  http://www-128.ibm.com/developerworks/webservices/library/specification/ws-sdo/

- SDO and SCA specifications submitted to OASIS:

  http://www.ibm.com/developerworks/webservices/library/specification/ws-scasdo/

6.10 Java 2 Connector Architecture

This section gives advice on how the J2EE Connector Architecture (JCA or J2C) should be used to access Enterprise Information Systems such as Customer Information Control System (CICS) and Information Management System (IMS).

6.10.1 Re-use of objects

Caching the ConnectionFactory is probably the biggest single performance enhancement that can be made, because it avoids JNDI lookups and reduces the I/O. The ConnectionFactory can be cached on the call into a static variable:

```
(ConnectionFactory)ic.lookup(fqndiName)
```

This gives a significant path length reduction by avoiding the lookup to JNDI each time a connection is required. The main benefit is in the CPU utilization reduction. Caching the initial context javax.naming.Context into a static variable also provides some path length reduction, which improves performance.

Figure 6-27 shows the path length in milliseconds of CPU per CICS transaction when re-using the initial context (IC), the ConnectionFactory (CF), both, and neither, with different communication area (COMMAREA) sizes.

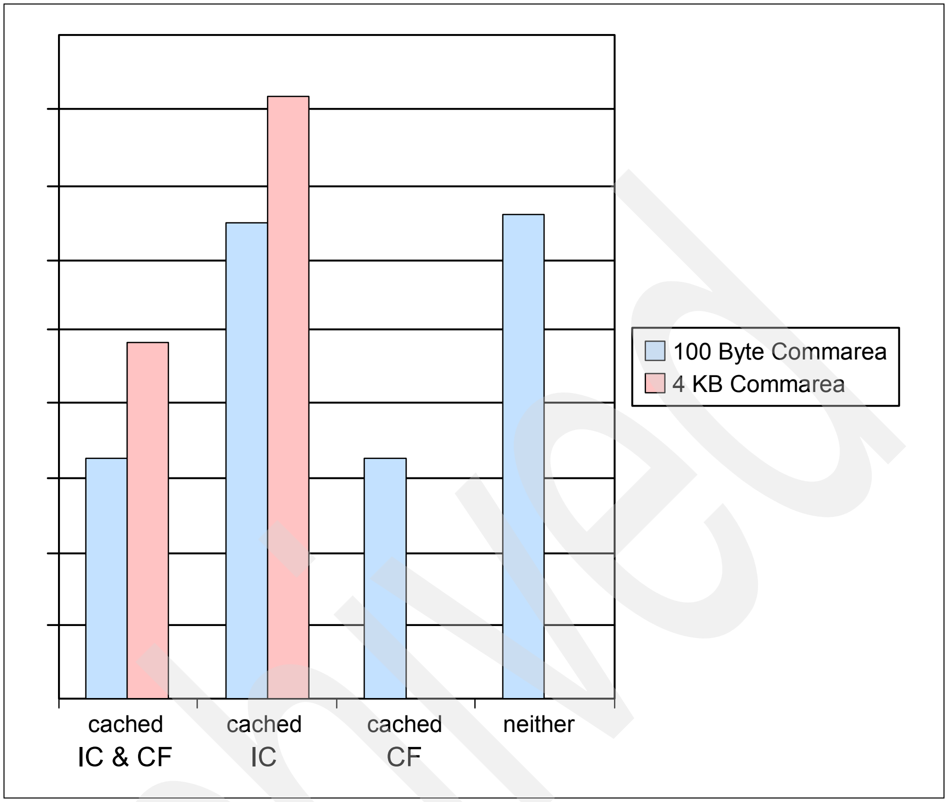


Figure 6-27 Path length using caching of objects

Figure 6-28 shows the throughput as the CICS transaction rate, that is, transactions per second, when re-using the initial context, the ConnectionFactory, both, and neither, with different COMMAREA sizes.

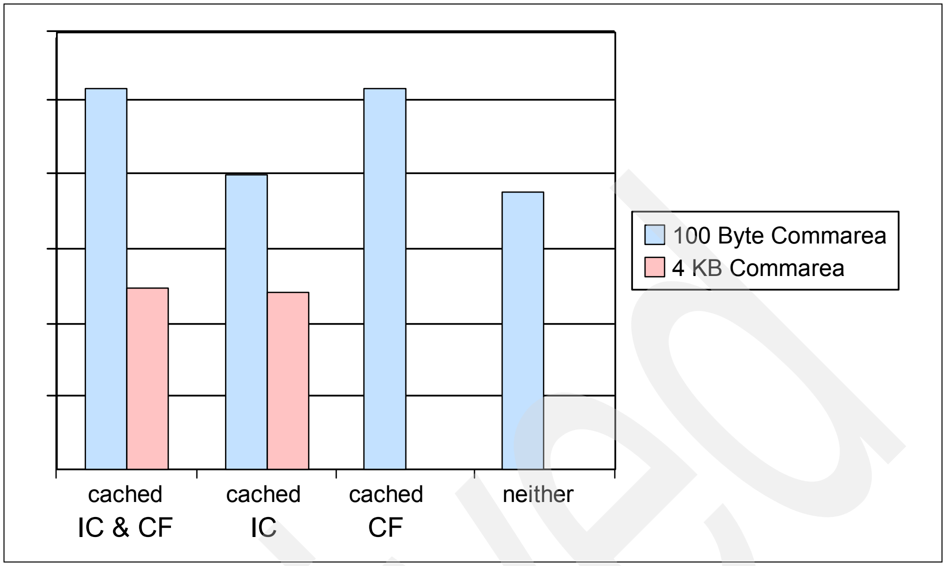


Figure 6-28 Throughput using caching of objects

Both graphs are based on a measurement of a two-phase commit using the local mode of CICS Transaction Gateway.

The JNDI lookup as well as obtaining the connection should not be done within the ejbCreate() method. The usage of unspecified transaction contexts is one of the biggest holes in the EJB specification and assures non-portability.

6.10.2 Managed environment

In a *managed environment*, the application components and the resource adapter are connected with the application server through *contracts*. This means that the application server manages the connection pooling, transactionality, and security for the application. In contrast to a *non-managed environment*, the implementation of the ConnectionManager interface happens within the application server. The access on the resource adapter is regulated by the application server via system contracts.

The use of the managed environment is a key benefit because it is one way to exploit XA and RRS transactions for two-phase commit and JCA connection pooling functions in WebSphere. The exploitation is achieved by defining a ConnectionFactory within the WebSphere Application Server Administrative Console or through WebSphere Admin (wsadmin) scripting. ConnectionFactory can be looked up through a resource reference within the JCA code:

```
cfA = (ConnectionFactory) ic.lookup(cfRefA);
```

6.10.3 Use of transactions

A big advantage of using the managed environment is that all transaction management can be delegated to the Web/EJB containers.

Resource Recovery Services (RRS) transactions provide better performance than XA transactions. This is because the resource adapters using RRS transactions normally use native z/OS® calls and resource-specific interfaces instead of TCP/IP connectivity used with XA resource managers. They avoid the overhead that comes with XA transactions.

Resource Recovery Services is used when connecting locally (not using TCP/IP) between WebSphere Application Server on z/OS and:

- IMS using the IMS Connector for Java and IMS Connect
- IMS using the IMS JDBC connector
- CICS using CICS Transaction Gateway
- DB2 for z/OS using a Type 2 driver
- WebSphere MQ using bindings mode

All RRS compliant resource adapters are required to support the property *RRSTransactional* in their ManagedConnectionFactory and must support a getter method for the property.

Example 6-93 shows an example of using RSS transactions.

Example 6-93 Using RRS transactions

```
java.lang.Boolean.RRSTransactional=true;

java.lang.Boolean getRRSTransactional(){
    // Determine if the adapter can run RRSTransactional based
    // on it's configuration, and set the RRSTransactional property
    // appropriately to true or false.
   return RRSTransactional;
}
```

RRS support is only applicable in a "local" environment, where the back-end must reside on the same system image. CICS and IMS resources adapters can use RRSTransactional support only when these adapters are configured to use local interfaces to their back-end resource manager, which, as stated above, must reside on the same system image as the IBM WebSphere Application Server for z/OS.

These adapters are also capable of being configured to a remote instance of their back-end resource manager. In this case, the adapters respond "false" when the getRRSTransactional() method is invoked and instead of running as

RRSTransactional, they use whichever one of the three types of J2EE Transaction support they have chosen to support. With the CICS TG in local mode, the RRS transactional mode is automatically exploited.

Another point to look at to improve performance is the *commit mode* during a transaction. It should be proved if it is necessary to run your commit modes with SyncLevel Confirm for a one-phase commit or SyncPoint for a two-phase commit. This is not necessary for read-only transactions. Using SyncLevel Confirm or SyncPoint blocks the IMS until your WebSphere Application Server application commits or until WebSphere Application Server has finished waiting for all other units of work involved.

More detailed information about the usage of RRS in WebSphere connectivity can be found in Chapter 8, "WebSphere Application Server for z/OS", of *Systems Programmer's Guide to Resource Recovery Services (RRS)*, SG24-6980.

6.10.4 Connection pooling

Connection pooling is perhaps the key benefit of the JCA managed environment. All actual socket connections can be managed by the WebSphere Application Server Pool Manager and configurable limits can be set and monitored within WebSphere Application Server. Therefore, the application developer only has to obtain a connection handle and the underlying managed connection is handled by the JCA infrastructure. The connection is acquired using the following statement:

```
eciConn = (Connection) cf.getConnection(ecf);
```

The first use of each connection in the pool might take longer, because the physical socket is established. To prevent this, you can write a simple application that primes the pool to establish each connection. With the CICS TG implementation, the getConnection() method does no I/O and this does not occur until the first usage of the connection by an interaction execute. The behavior of a specific implementation apart from CICS TG has to be determined.

For more information, refer to Chapter 10 in *Performance Monitoring and Best Practices for WebSphere on z/OS*, SG24-7497.

6.10.5 Connection usage

When using the J2EE Connector Architecture (JCA) Version 1.0 in WebSphere Application Server Version 5.x, the connection usage should follow the "get-use-close" model. This means that an application always obtains a new connection when it requires one, then uses it and then closes it again when the work is done.

This might sound inefficient, but the connection pooling the application server implements makes the get() operation cheap. Also, different instances or parts of the application can reuse the connection because the application holds on to the connection for only as long as it is required. Therefore, the total resource usage is reduced.

When using the J2EE Connector Architecture (JCA) Version 1.5 in WebSphere Application Server Version 6.x, the connection usage should follow the "cached-handle" model. This means that an application obtains the connection once up-front and caches a reference to it in an instance field. This allows the programmer to delay the close() method on a connection. Instead, the J2C infrastructure disassociates the managed connection from the connection handle when the transaction scope ends, thus letting the JCA infrastructure efficiently handle lazy use of connections. Furthermore, it frees up the developer from the concern of worrying about connection handles.

IBM WebSphere Application Server Version 5.x addressed the drawbacks of the cached-handle connection model with an extension to the JCA 1.0 specification known as "smart connection handles".

With the CICS TG implementation, the advantages of connection pooling are only evident when a remote gateway is used, because the local mode of the CICS TG does not use any I/O for access.

6.10.6 Lazy association

Rather than re-associating the connection handle with the managed connection the next time a method is called, the optimization uses lazy association. If the method does not use the connection, or it only calls simple methods on the connection handle that do not require access to the back-end, a managed connection is not removed from the pool unnecessarily.

Instead, when the connection handle determines that it does have to be reassociated to a managed connection, it can cast the connection manager to a LazyAssociatableConnectionManager and call the associateConnection method.

This method takes the connection handle as the first parameter, followed by the managed ConnectionFactory, and requests information passed on the initial call to allocateConnection. The connection manager then finds another suitable managed connection from the pool and uses the managed connection's associateConnection method to tie it to the connection handle. See Example 6-94.

Example 6-94 Interfaces for dissociation and lazy association

```
public interface DissociatableManagedConnection {

    void dissociateConnections() throws ResourceException;

}

public interface LazyAssociatableConnectionManager {

    void associateConnection(Object connection,
                             ManagedConnectionFactory mcf,
                             ConnectionRequestInfo cxReqInfo)
            throws ResourceException

}
```

This allows the application developer to forget about closing down connections.

6.10.7 Lazy enlistment

Transactions, particularly XA and RRS (global) transactions, are expensive. This makes it more important for a transaction not to do more work than necessary. Interfaces in JCA 1.5 prevent unnecessary enlistment of XAResource objects. You should not enlist in the transaction unless absolutely necessary. The solution therefore is a lazy enlistment. See Example 6-95.

Example 6-95 Interfaces for lazy enlistment

```
public interface LazyEnlistableManagedConnection {

}

public interface LazyEnlistableConnectionManager {

    void lazyEnlist(ManagedConnection mc)
            throws ResourceException;

}
```

The LazyEnlistableManagedConnection interface is a marker interface implemented by the managed connection to indicate to the connection manager that it does not have to eagerly enlist the managed connection in an existing transaction when a new connection is created in a transaction or in a new transaction started when a connection already exists.

If a connection handle is about to perform some work that should be part of any transaction, and its managed connection has not already been enlisted, it should determine whether the connection manager implements the LazyEnlistableConnectionManager interface. If it does, it should call the lazyEnlist method passing the managed connection. This method returns nothing, but if a transaction is associated with the calling thread, the XAResource from the managed connection is enlisted at that point. If the connection is not enlisted, it has to call lazyEnlist again before each subsequent piece of work in order to check that a transaction has not been started since the last time it called the method.

Lazy Connection Enlistment applies when a ConnectionFactory is referenced in a resource reference but is not used in a transaction. This means that no interactions are sent to it and so it is not enlisted.

This kind of enlistment only applies to XA transactions and not to transactions created in a CICS TG local mode.

6.10.8 Best practices for CICS Transaction Gateway

The CICS Transaction Gateway uses the J2EE Connector Architecture to work with CICS, and the information provided in 6.10, "Java 2 Connector Architecture" on page 447 applies here as well. In addition, there are some additional key points that apply to the usage of the CICS Transaction gateway specifically. We go over these in the following sections.

Transmitting data in a COMMAREA

The CICS TG, in combination with CICS Transaction Server, provides built-in data compression for ECI flows. Any trailing nulls in the COMMAREA payload sent between the JCA resource adapter, the Gateway daemon, and CICS are automatically removed when sent over any of the supported network connections. This compression is dynamic and is not visible to either the J2EE client application or the CICS application.

To best exploit the functionality, there are two suggested approaches:

1. If the length of the input and output data structures are known, then we suggest that the J2EE client application builds a record containing just the required input information, and then sets the specified COMMAREA length to the length of the COMMAREA data structure to be used in CICS. This COMAREA structure should be equal to the size of the data to be returned by CICS. In addition, if the J2EE client application has to override how much data is returned, the payload should specify this ahead of time using the setReplyLength method to receive a truncated amount of data.

2. If the length of the input or output data structure cannot be determined, then we suggest that the J2EE client application builds a record containing the required input information, and then sets the specified COMMAREA length to be the maximum possible (that is, 32 KB). The CICS application must then be made capable of handling a full 32 KB of payload and should ensure this data structure is initialized to binary zeroes. The CICS application should then return as much information as necessary, making sure to efficiently utilize the space within the COMMAREA payload, leaving any empty data as trailing binary zeroes. The J2EE client application receives the returned data, but all trailing nulls are removed from the data sent across any network connections.

Data conversion

When writing Java applications to invoke CICS programs, data conversion is a key issue, because CICS evolved in an EBCDIC world, while Java is based on Unicode. Normally, you would convert Java Strings that are stored in Unicode within the JVM to an EBCDIC byte array, which is required by CICS. The alternative is to convert the data to ASCII within the JVM and then convert from ASCII to EBCDIC within CICS. Data conversion from Unicode to ASCII is an efficient operation in Java, as it involves only the removal of the high-order byte, while conversion to EBCDIC requires a table lookup. This means that the high cost of EBCDIC conversion can be transferred to CICS, therefore potentially improving performance within the JVM.

In this case you would use an ASCII code page, such as 8859_1, when creating the byte array:

```
byte abCommarea[] = new byte[27];
abCommarea = "abcd".getBytes("8859_1");
```

After receiving the byte array back from CICS, convert it to a String as follows:

```
String strCommarea = new String(abCommarea,"8859_1");
```

Refer to Chapter 10 in *Performance Monitoring and Best Practices for WebSphere on z/OS*, SG24-7497 for more information on settings and parameters related to the CICS Transaction Gateway.

6.10.9 IMS Connect

If you want to have a closer look at your IMS connections, there is a tool to do performance analysis for IMS Connect called IMS Connect Extensions for z/OS:

http://www.ibm.com/software/data/db2imstools/imstools/imsconnectext.html

The interpretation from data collected by the IMS Connect Extensions for z/OS can be done with the IMS Performance Analyzer for z/OS:

`http://www.ibm.com/software/data/db2imstools/imstools/imspa.html`

Refer to Chapter 10 in *Performance Monitoring and Best Practices for WebSphere on z/OS*, SG24 7497 for more information on settings and parameters related to IMS Connect.

6.11 Java Message Service

In this section we cover introductory concepts and define the scope of JMS. After that we discuss best practices.

6.11.1 Basic concepts

The Java Message Service (JMS) was developed by Sun Microsystems to provide a means for Java programs to access enterprise messaging systems. Before we discuss JMS, let us take a look at enterprise messaging systems.

Enterprise messaging systems, often known as message oriented middleware (MOM), provide a mechanism for integrating applications in a loosely coupled, flexible manner. They provide asynchronous delivery of data between applications on a store and forward basis; that is, the applications do not communicate directly with each other, but instead communicate with the MOM, which acts as an intermediary.

The MOM provides assured delivery of messages (or at least makes its best effort) and relieves application programmers from knowing the details of remote procedure calls (RPC) and networking/communications protocols.

Messaging flexibility

Using MOM's application programming interface (API) makes possible an application A communicating with application B in asynchronous form. See Figure 6-29.

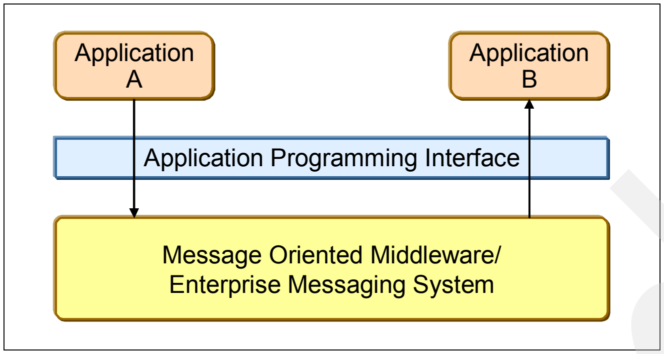


Figure 6-29 MOM functionality (overview)

The MOM routes the message to Application B, which can exist on a completely different computer; the MOM handles the network communications. If the network connection is not available, the MOM stores the message until the connection becomes available, and then forward it to Application B.

Another aspect of flexibility is that Application B might not even be executing when Application A sends its message. The MOM holds the message until Application B begins execution and attempts to retrieve its messages. This also prevents Application A from blocking while it waits for Application B to receive the message.

This asynchronous communication requires applications to be designed somewhat differently from the way most are designed today, but it can be an extremely useful method for time-independent or parallel processing.

Loose coupling

The real power of enterprise messaging systems lies in the loose coupling of the applications. In Figure 6-29, Application A sends its messages indicating a particular destination, for example “order processing.” Today, Application B provides order-processing capabilities. But, in the future, we can replace Application B with a different order-processing program, and Application A is none the wiser. It continues to send its messages to “order processing” and the messages continue to be processed.

Likewise, we could replace Application A, and as long as the replacement continues to send messages for “order processing,” the order-processing program would not have to know there is a new application sending orders.

Point to point / Publish and subscribe

Originally, enterprise messaging systems were developed to implement a point-to-point model (PTP) in which each message produced by an application is received by one other application. In recent years, a new model has emerged, called publish and subscribe (or pub/sub).

Pub/sub replaces the single destination in the PTP model with a content hierarchy, known as topics. Sending applications publish their messages, indicating that the message represents information about a topic in the hierarchy.

Applications wishing to receive those messages subscribe to that topic. Subscribing to a topic in the hierarchy that contains subtopics allows the subscriber to receive all messages published to the topic and its subtopics.

Figure 6-30 illustrates the publish and subscribe model.

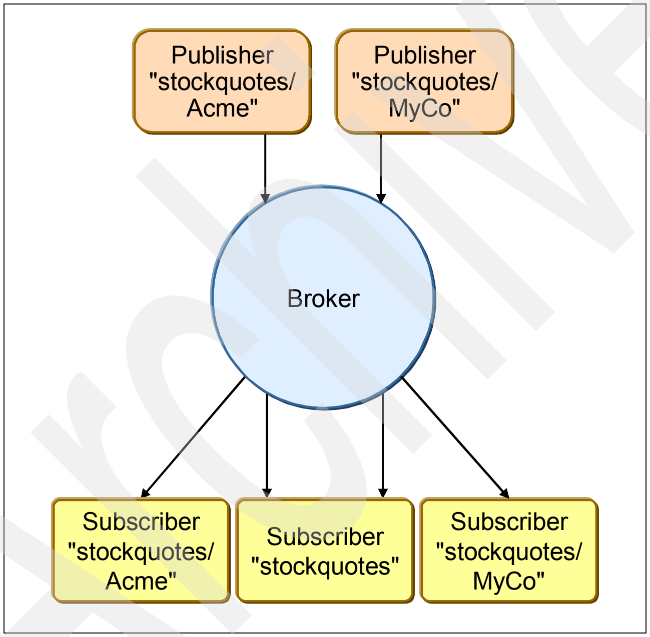


Figure 6-30 Publish-Subscribe model

Multiple applications can both subscribe and publish messages to a topic, and the applications remain anonymous to one another. The MOM acts as a broker, routing the published messages for a topic to all subscribers for that topic.

6.11.2 What the Java Message Service is

Java Message Service (JMS) is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product.

Prior to JMS, each MOM vendor provided application access to its product through a proprietary API, often available in multiple languages, including the Java language. JMS provides a standard, portable way for Java programs to send and receive messages through a MOM product. Programs written with JMS can run on any MOM that implements the JMS standard.

The key to JMS portability is the fact that the JMS API is provided by Sun as a set of interfaces. Products that provide JMS functionality do so by supplying a provider that implements these interfaces.

As a developer, you build a JMS application by defining a set of messages and a set of client applications that exchange those messages.

JMS objectives

To better understand JMS, it helps to know the objectives set by the authors of the JMS specification.

There are many enterprise messaging products on the market today, and several of the companies that produce these products were involved in the development of JMS.

These existing systems vary in capability and functionality. The authors knew that JMS would be too complicated and unwieldy if it incorporated all of the features of all existing systems. Likewise, they believed that they could not limit themselves to only the features that all of the systems had in common.

The authors believed that it was important that JMS include all of the functionality required to implement "sophisticated enterprise applications."

The objectives of JMS, as stated in the specification, are to:

- Define a common set of messaging concepts and facilities.
- Minimize the concepts a programmer must learn to use enterprise messaging.
- Maximize the portability of messaging applications.
- Minimize the work required to implement a provider.

- Provide client interfaces for both point-to-point and pub/sub domains. “Domains” is the JMS term for the messaging models discussed earlier. (Note: A provider does not have implement both domains.)

What JMS does not provide

The following features, common in MOM products, are not addressed by the JMS specification. Although acknowledged by the JMS authors as important for the development of robust messaging applications, these features are considered JMS provider-specific.

JMS providers are free to implement these features in any manner they want, if at all:

- Load balancing and fault tolerance
- Error and advisory system messages and notification
- Administration
- Security
- Wire protocol
- Message type repository

Three sets of interfaces

Basically, in JMS, the three channel interfaces — Destination, Queue, and Topic — have the unfortunate consequence of tripling the number of interfaces in the API. To send or receive messages, a client uses a ConnectionFactory to get a connection, which it uses to create a session and obtain message producers and consumers on the desired queues and topics. The only problem is that each kind of destination has its own factory, connection, session, producer, and consumer interfaces (basically, its own domain of interfaces), as summarized in Table 6-9.

Table 6-9 Relationship of point-to-point and publish/subscribe interfaces

JMS common	Point to point domain	Pub/sub domain
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

Furthermore, JMS provides another set of interfaces for compatibility with XA (distributed) transactions. This means that besides the 18 interfaces shown in the table, JMS has yet another nine interfaces — XA versions of the factory, connection, and session types — three times as many types as are really required.

Of those 27 interfaces, only the six common interfaces (those in the first column) are truly required to send and receive messages. Queue and Topic are sometimes useful for distinguishing the point-to-point and publish/subscribe approaches. The other ten interfaces in the table and at least six of the nine XA interfaces are not really required, at least not for writing client code.

Domain unification

The major change after JMS 1.1 specification from J2EE (Java 2 Enterprise Edition 1.4) that WebSphere Application Server 6.1 implements is the addition of new APIs to support client code that works simultaneously with either the point-to-point or publish/subscribe domains. Specifically, the latest release adds methods to the common interfaces to make the queue and topic extensions polymorphic.

For example, MessageProducer now implements send, so a client can send a message to a destination without knowing whether the destination is a queue or a topic. Likewise, MessageConsumer declares the receive method so that the same client code works whether it's using an implementation of a queue receiver or a topic subscriber.

These new APIs enable the developer to write JMS client code that is much more reusable. It simply accesses destinations and uses them without having to know which destinations are queues and which are topics. Code written to access a queue can also be reused, unchanged, to access a topic in the same way, and vice versa. This was not possible with JMS 1.0.2, for example.

Because the common interfaces are now able to perform almost all of the same tasks as their domain-specific extensions (for example, MessageProducer can do just about everything QueueSender and TopicPublisher can), the domain-specific subinterfaces are really no longer required.

For more information on developing a JMS application step by step in basic concepts, go to the following URLs:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/tmj_devap.html

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/tmj_devcl.html

6.11.3 JMS connection considerations

Before discussing connection considerations, we first explain some basic concepts about connections.

ConnectionFactory

ConnectionFactory is an administered object that is retrieved from JNDI to create a connection to a provider. It contains a createConnection() method, which returns a Connection object.

Connection

Connection encapsulates an active connection to a provider. These are some of its methods:

- createSession(boolean, int);

  Returns a Session object. The boolean parameter indicates whether the Session is transacted or not; the int indicates the acknowledgment mode (see Acknowledgment).

- start();

  Activates the delivery of messages from the provider.

- stop();

  Temporarily stops delivery of messages; delivery can be restarted with start().

- close();

  Closes the connection to the provider and releases all resources held in its behalf.

Start the connection when appropriate

If you start a connection before starting the subscriber/receiver (consumer), then the messages have to wait in the JMS server, or they persist if they are durable messages. This can be unnecessary overhead and to avoid this, ensure that the consumer is ready to accept messages before starting its connection.

Start consumers before producers

If you are starting a system, it can help to start the consumers first. This avoids the possibility of excess buffering/persisting of messages during the interval between a producer starting and the corresponding consumer starting. This is most likely to happen with point-to-point or with durable subscriptions.The solution is to use a container-managed transaction.

Process messages concurrently

JMS provides a facility to process incoming messages concurrently by using a ConnectionConsumer that pools session objects and handles the distribution of work among them. You can create ConnectionConsumer using the methods in Example 6-96 and Example 6-97.

Remember that Queue encapsulates a point-to-point destination. It is an administered object that is retrieved from JNDI.

Example 6-96 ConnectionConsumer for queues

```
public ConnectionConsumer createConnectionConsumer(Queue queue,
String messageSelector, ServerSessionPool sessionPool, int
maxMessages) throws JMSException
```

Remember that Topic encapsulates a pub/sub destination. It is an administered object that is retrieved from JNDI.

Example 6-97 ConnectionConsumer for topics

```
public ConnectionConsumer createConnectionConsumer(Topic topic,
String messageSelector, ServerSessionPool sessionPool, int
maxMessages) throws JMSException
```

In these methods, the main parameters are maxMessages and ServerSessionPool. maxMessages denote the maximum number of messages that can be simultaneously assigned to a server session. ServerSessionPool is an administered object that you configure in a vendor specific manner.

Close the connection when finished

Always call the close() method on JMS connection and session objects when they are no longer required. This approach works with other JMS objects such as MessageProduction/ MessageConsumer, QueueSender/QueueReceiver, TopicPublisher/TopicSubscriber. This topic is a reference to other JMS objects related to this chapter.

This releases the underlying resource handle. It is especially important for the publish-subscribe model, where clients have to deregister from their subscriptions. Closing the objects allows the queue manager to release the corresponding resources in a timely fashion. Failure to do so can affect the capacity of the queue manager for large numbers of applications or threads. It is important to be aware that close of any resources can throw an exception.

See the closing code snippet in Example 6-98.

Example 6-98 Close resources best practices snippet

```
Connection connection = null;
 MessageProducer sender= null;
 Session session = null;

 try {
  //Usage of JMS Resouces

 } catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
 }
 finally{
  //it is important to close all the JMS resources used. To do this,
  //you call the close() method on the various used classes
  //(QueueConnection, QueueSession, QueueSender, QueueReceiver ,
  //MessageProducer ,Message Consumer, TopicPublisher,
  //TopicSubscriber)
  //when the resources are no longer required.

  try{
     try{
        //This example explain MessageProducer
        //but place your queueSender.close(),
        //subcriber.close() and so on, if you are using.
        sender.close();
      }
      finally{
         try{
           session.close();
          }
          finally{
             connection.close();
          }
        }
      }
    } catch (Exception e){
      //Make a handling of exception or throw to the class
      //that called this method
    }
  }
}
```

At the first impression, it can be difficult to understand the nested try...finally blocks, but the objective here is delivery of resources no matter what occurs in previous close execution. In fact, it is more complicated, but more safe.

6.11.4 JMS session considerations

Session is the single-threaded context for sending and receiving messages. Some of its methods are:

- createProducer(Destination)

  Returns a MessageProducer object to send messages to the specified Destination.
- createConsumer(Destination)

  Returns a MessageConsumer object to receive messages from the specified Destination.
- commit()

  Commits all consumed or produced messages for the current transaction.
- rollback()

  Rolls back all consumed or produced messages for the current transaction.
- create<MessageType>Message(...)

  A variety of methods that return a <MessageType>Message — for example, MapMessage, TextMessage, and so on.

A session is used to create multiple producers and consumers. A session can be a QueueSession for a point-to-point or a TopicSession for a publish-subscribemodel.

Choose proper acknowledgement mode

When you create a session object, you can choose any one of the three acknowledgement modes: AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, or DUPS_OK_ACKNOWLEDGE.

- CLIENT_ACKNOWLEDGE mode is not a feasible option (when you have the freedom to choose from the other two options), because the JMS server cannot send subsequent messages until it receives an acknowledgement from the client.
- AUTO_ACKNOWLEDGE mode follows the policy of delivering the message once-and-only once, but this incurs an overhead on the server to maintain this policy.

- DUPS_OK_ACKNOWLEDGE mode has a different policy of sending the message more than once, thereby reducing the overhead on the server (imposed when using the AUTO_ACKNOWLEDGE), but imposes an overhead on the network traffic by sending the message more than once.

But, the AUTO_ACKNOWLEDGE mode cleans up resources early from the persistent storage/memory, which reduces the overhead because of that.

Control transaction

In the code shown in Example 6-99 and Example 6-100, the first parameter indicates the session is a transactional session. The session objects have commit(), rollback(), and isTransacted() methods to deal with transactional messages.

Example 6-99 Transaction messages for topics

```
topicSession =
tConnect.createTopicSession(true,Session.AUTO_ACKNOWLEDGE);
```

Example 6-100 Transaction messages for queues

```
queueSession =
qConnect.createQueueSession(true,Session.AUTO_ACKNOWLEDGE);
```

The problem here is that a transaction starts implicitly when a session is created and ends when a commit() or rollback() method is called. At this stage, after calling a commit() or rollback() method, one more transaction starts implicitly, because there is no explicit method (begin() method) to start a transaction. So, there are a chain of transactions that depend upon commit() or rollback() method calls. Transactional messages are cumulated in the JMS server until the transaction is committed or rolled back. This imposes significant overhead on a JMS server.

Close the session when finished

The reasons for closing sessions are described in “Close the connection when finished” on page 463

6.11.5 JMS destination considerations

The *destination* is a virtual channel between producers and consumers.

Producers send messages to the destination which in turn deliver messages to consumers. To get the destination object, you have to perform a JNDI lookup. The code snippet in Example 6-101 shows a Destination usage.

Example 6-101 Destination usage code snippet

```
//Look up administered objects
//destinationName and factoryName are java.lang.String Objects
InitialContext initContext = new InitialContext();
ConnectionFactory factory =
(ConnectionFactory) initContext.lookup(factoryName);
Destination destination = (Destination)
initContext.lookup(destinationName);
initContext.close();
```

If you require specific access to queues and topics, you can use the commands shown in Example 6-102.

Example 6-102 Lookup of Queues or Topics instead of generic destinations

```
Topic topic = (Topic) initContext.lookup(topicName);

Queue queue = (Queue) initContext.lookup(queueName);
```

For non-durable messages, the time that messages take to deliver to the destination depends upon its number and destination size. If a large number of messages collect in a destination, they take more time to deliver. Set a smaller destination size and smaller number of maximum messages to the destination to improve performance.

Redelivery delay time defines when to redeliver a message if a failure occurs. If this is less, the frequency of redelivery of a message is high, thus increasing network traffic and vice versa. So, high redelivery delay time gives better performance. Redelivery limit defines the number of times a message should be redelivered. Although the probability of guaranteed messaging is less, if the Redelivery limit is less, then the performance is better because the memory overhead for non-durable messages and persistent overhead for durable messages is reduced.

6.11.6 JMS message producer / consumer considerations

The producer sends messages to the destination where a consumer consumes messages from the destination. The message producer/consumer is created by the session object.

You send the messages using the producer. Example 6-103 and Example 6-104 provide two samples of sending a message using a topic and a queue respectively.

Example 6-103 Publishing messages to a topic

```
publisher.publish(Message message);
// or
publisher.publish(Topic topic, Message message, int deliveryMode,
   int priority, long timeToLive);
```

Example 6-104 Sending messages to a queue

```
sender.send(Message message);
// or
sender.send(Queue queue, Message message, int deliveryMode,
   int priority, long timeToLive);
```

The parameters DeliveryMode and TimeToLive are important from a performance perspective. You can provide values for these parameters when you configure ConnectionFactory or destination or when you send a message.

Choose non-persistent messages where appropriate

Delivery mode defines whether the message can be persistent or non-persistent. This parameter ensures that message delivery is guaranteed.

For persistent messages, if you use MessageProducer, the methods is:

`setDeliveryMode(int)`

This sets the delivery mode for subsequent messages sent. Valid values are Deliverymode.PERSISTENT for persistent messages and Deliverymode.NON_PERSISTENT for non-persistent messages.

If you define the delivery mode as persistent, then the message is stored by the JMS server before delivering it to the consumer, as shown in Figure 6-31.

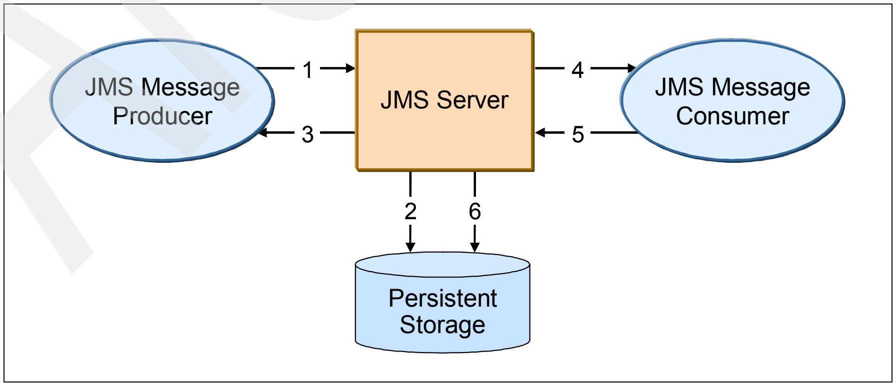


Figure 6-31 Persistent message flow

If you define the delivery mode as non-persistent, then the message is delivered to the consumer without being saved by the JMS server, as shown in Figure 6-32.

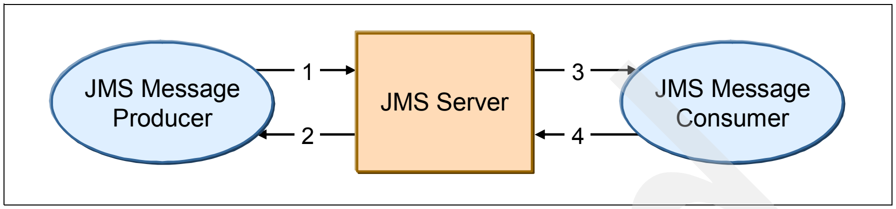


Figure 6-32 Non-persistent message flow

The foregoing figures clearly show the difference between the two delivery modes. When using the persistent delivery mode, each message has to be stored by the JMS server either in the database or the file system before delivery of message to consumer and removed after delivery of message. So as far as possible, restrict the use of persistent delivery mode unless and until absolutely necessary.

The code snippet in Example 6-105 gives a further explanation of this approach.

Example 6-105 SetDeliverMode code snippet usage

```
//Create JMS objects
Connection connection = factory.createConnection();
Session session = connection.createSession(false,
     Session.AUTO_ACKNOWLEDGE);
MessageProducer sender = session.createProducer(destination_queue);
sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

Set Time to live value properly

You can set the age of the message by setting the Time to live parameter after which the message expires. By default, the message never expires, so you should set the optimal message age so as to reduce memory overhead.

If you use MessageProducer, the methods is:

```
setTimeToLive(long)
```

This sets the duration before expiration, in milliseconds, of subsequent messages sent. See the code snippet in Example 6-106 for usage.

Example 6-106 SetTimeToLive code snippet usage

```
//Create JMS objects
Connection connection = factory.createConnection();
Session session = connection.createSession(false,
     Session.AUTO_ACKNOWLEDGE);
MessageProducer sender = session.createProducer(destination_queue);
sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
sender.setTimeToLive(1000L);
```

Receive messages asynchronously

You can receive messages synchronously or asynchronously. To receive asynchronous messages, you have to implement the MessageListener interface and onMessage() method. For receiving synchronous messages, you have to use any of the following methods of MessageConsumer:

- receive()

  Returns the next message that arrives; this method blocks until a message is available.

- receive(long)

  Receives the next message that arrives within long milliseconds; this method returns null if no message arrives within the time limit.

- receiveNoWait

  Receives the next message if one is immediately available; this method returns null if no message is available.

In short, the first method blocks the call until it receives the next message, the second method blocks until a timeout occurs, and the last method never blocks.

Normally, when using asynchronous messaging, the calls are never blocked, so a better option to improve performance is to receive messages asynchronously by implementing MessageListener.

The implementation for WebSphere MQ actually does a receive(5000) in a loop under the covers so it is not true asynchronous and might actually increase the number of MQ operations if a larger number would have been chosen.

Close producer/consumer when finished

When not required anymore the message producers and consumers should be closed. See the topic about JMS connections in “Close the connection when finished” on page 463

6.11.7 JMS message considerations

A Message object contains information that is passed between applications. It contains information as payload, headers, and properties. From the performance perspective, you mainly have to consider the type of message, whether it is a Text, Object, Byte, Stream, or Map message.

The higher level message types such as Map, Stream, and Object require more processing to convert them to and from the internal transmission format. These types should not be used unless they provide exactly the functionality the application requires. ObjectMessage carries a serialized Java object and when you choose ObjectMessage, you have to use "transient" keyword for variables that must *not* be sent over the network to reduce overhead. Also, ByteMessage takes less processing in the JMS client than a TextMessage, because it is the simplest format.

6.11.8 JMS performance testing

If you want to have a closer look at your JMS connections, there is a tool to do performance analysis for JMS called IBM Performance Harness for Java Message Service:

http://alphaworks.ibm.com/tech/perfharness

Also refer to the following Redbooks publications for information about tuning as well as learning more about JMS and WebSphere MQ.

- *WebSphere MQ V6 Fundamentals*, SG24-7128
- *Performance Monitoring and Best Practices for WebSphere on Z/OS*, SG24-7269
- *WebSphere Scalability and Performance Handbook*, SG24-6392

6.12 Web Services

As mentioned at the beginning of this chapter, we have several levels of integration. Web Services are placed in Level 2 and 3 of integration, concerning business integration as a key point for Service Oriented Architecture (SOA).

In this section, we cover some basic concepts that are used in Web Services; starting with what a Web Service is. We present an overview of WSDL, JAX-RPC, and SOAP, as well as technologies that make up part of the Web Services feature pack for WebSphere 6.1 such as JAX-WS and JAXB 2.0 to make a basic alignment. After that, we go on to describe some best practices.

6.12.1 Some concepts

In the following sections we explain some basic concepts around Web Services.

What is a Web service?

It is probably fair to say that the initial hype surrounding Web Services has reached astronomical proportions, so much so that the language used to describe what Web Services are, and how to go about implementing them, is becoming entirely too overloaded and muddled. In fact, one of the most difficult tasks the recently formed W3C Web Services Architecture Working Group has faced so far has been to determine the answer to what seems like an easy question: What is the generic definition of a Web service? Given all the hype, one might think that such a definition would not be hard to come by. The challenge, however, is to overcome the abundance of definitions, most of which are contradictory and only reveal a fragment of what Web Services can be or might become.

When distilling a collection of best practices, one of the most important first steps is to ensure that the language used to describe the various core concepts is clear, concise, and accurate. Unfortunately, you also have to work within the bounds of already existing and accepted vernacular (regardless of how much confusion an existing name or term might cause). The Simple Object Access Protocol (SOAP) itself, for instance, is an improperly named technology in the Web Services world: though the acronym stands for Simple Object Access Protocol, it describes a messaging protocol specification that has little to do with objects and is far from simple to implement.

The working definition of a Web service that the W3C Web Services Architecture group managed to agree on is as follows:

A Web service is a software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts, and supports direct interactions with other software applications using XML-based messages via Internet-based protocols.

For those who think that Web Services are limited to SOAP messages used to invoke the methods of an application over an HTTP connection (as the traditional SOAP stock-quote service does), this definition might be a bit surprising. However, it tells us a couple of interesting things pertinent to our goal of a common vernacular:

- Web Services do not require SOAP
- Web Services do not require HTTP

However, a Web service (as defined by the W3C) does require an XML-based description mechanism of some kind (such as WSDL) that can be used to describe the service's form and function. Remember, the backdrop framework for this industry-wide initiative around Web service technologies is a service-oriented architecture (SOA); for more information, see the Resources section below. As such, a non-XML based description mechanism (such as IDL) would make an SOA implementation more complex and less open.

Note also that the W3C's definition, while mentioning service discovery, does not mention UDDI or any other specific discovery mechanism. This fact becomes important as we walk through the various business scenarios and explore how and why certain service discovery mechanisms are used. Specifically, while early literature on the Web Services architecture asserted that UDDI had a core, essential role to play, real-life implementation of business solutions with Web Services have demonstrated that UDDI's most significant role is actually quite specialized at this time; in fact, many Web Services solutions are built that do not use UDDI in any way. In the vast majority of current business cases, it would be safe to say that these discovery mechanisms are not yet a core component for integrating processes between business partners.

Artifacts used to develop Web Services

With development artifacts you can develop an enterprise bean or a Java bean module into a Web service. This topic describes artifacts used to develop Web Services that are based on the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification.

To create a Web service from an enterprise bean or a Java bean module, the following files are added to the respective Java archive (JAR) file or Web archive (WAR) modules at assembly time:

- Web Services Description Language (WSDL) Extensible Markup Language (XML) file:

  The WSDL XML file describes the Web service that is implemented. We talk more about WSDL in “WSDL” on page 474.

- Service Endpoint Interface:

  A Service Endpoint Interface is the Java interface corresponding to the Web service port type implemented. The Service Endpoint Interface is defined by the Java API for XML Web Services (JAX-WS) or Java API for XML-Based RPC (JAX-RPC) Web Services runtime that you are using:

  – For more about JAX-RPC, see “JAX-RPC” on page 478.
  – For more about JAX-WS, see “JAX-WS” on page 479.

- webservices.xml (JAX-RPC applications only):

  For JAX-RPC applications, the webservices.xml file contains the J2EE deployment descriptor of the Web service specifying how the Web service is implemented. The webservices.xml file is defined in the Web Services for the J2EE specification. For JAX-WS applications, deployment descriptors are not supported and have been replaced by the use of annotations.
- ibm-webservices-bnd.xmi (JAX-RPC applications only):

  This file contains WebSphere product-specific deployment information and is defined in ibm-webservices-bnd.xmi assembly properties.
- Java API for XML-based remote procedure call (JAX-RPC) mapping file:

  The JAX-RPC mapping deployment descriptor specifies how Java elements are mapped to and from WSDL file elements.

The following files are added to an application client, enterprise beans, or Web module to permit J2EE client access to Web Services:

- WSDL file:

  The WSDL file is provided by the Web service implementer.
- Java interfaces for the Web service:

  The Java interfaces are generated from the WSDL file as specified by the JAX-WS or JAX-RPC specification. These bindings are the Service Endpoint Interface based on the WSDL port type, or the service interface, which is based on the WSDL service.
- ibm-webservicesclient-bnd.xmi (JAX-RPC applications only):

  This file contains WebSphere product-specific deployment information, such as security information for JAX-RPC applications. For JAX-WS applications, deployment descriptors are not supported and have been replaced by the use of annotations.
- Other JAX-RPC binding files:

  Additional JAX-RPC binding files that support the client application in mapping SOAP to the Java language are generated from WSDL by the WSDL2Java command tool.

WSDL

Web Services Description Language (WSDL) is an Extensible Markup Language (XML)-based description language. This language was submitted to the World-Wide Web Consortium (W3C) as the industry standard for describing Web Services. The power of WSDL is derived from two main architectural principles: the ability to describe a set of business operations, and the ability to separate the description into two basic units. These units are a description of the operations, and the details of how the operation and the information associated with it are packaged.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definitions of endpoints and messages are separated from their concrete network deployment or data format bindings. This separation supports the reuse of abstract definitions: messages, which are abstract descriptions of exchanged data, and port types, which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports defines a service. Therefore, a WSDL document is composed of several elements.

WSDL architecture

This section explains the architecture of a Web Services Description Language file. The WSDL files are written in Extensible Markup Language. To learn more about XML, see the following URLs:

http://www.ibm.com/developerworks/xml/library/x-contain.html
http://www.ibm.com/developerworks/xml/library/x-toptenjul2007.html
http://www.ibm.com/developerworks/xml/library/xml-schema/

Figure 6-33 shows the structure of the information in a WSDL file:

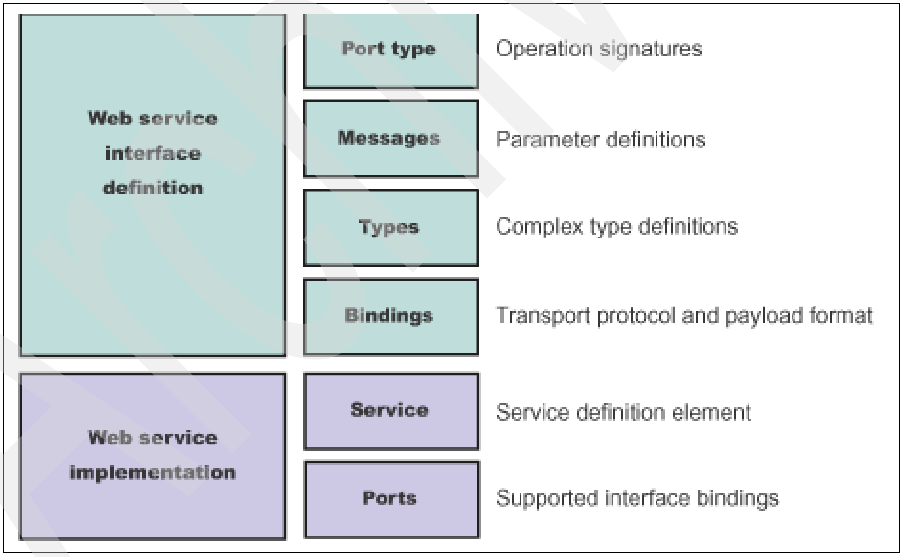


Figure 6-33 WSDL architecture

A WSDL file contains the following main parts:

- Web service interface definition:

  This part contains the elements, as well as the namespaces.

- Web service implementation:

  This part contains the definition of the service and ports.

A WSDL file describes a Web service with the following elements:

- portType:

  This represents a description of the operations and associated messages. The portType element defines abstract operations. See Example 6-107.

  Example 6-107 Port type snippet xml code inside wsdl file

```
<portType name="EightBall">
   <operation name="getAnswer">
      <input message="ebs:IngetAnswerRequest"/>
        <output message="ebs:OutgetAnswerResponse"/>
   </operation>
</portType>
```

- message:

  This represents the description of input and output parameters and return values. See Example 6-108.

  Example 6-108 message type snippet inside wsdl file

```
<message name="IngetAnswerRequest">
    <part name="meth1_inType" type="ebs:questionType"/>
</message>
<message name="OutgetAnswerResponse">
    <part name="meth1_outType" type="ebs:answerType"/>
</message>
```

- types:

  This represents the schema for describing XML types used in the messages. See Example 6-109.

  Example 6-109 xml type snippet inside wsdl file

```
<types>
    <xsd:schema targetNamespace="...">
       <xsd:complexType name="questionType">
         <xsd:element name="question" type="string"/>
       </xsd:complexType>
       <xsd:complexType name="answerType">
         <xsd:element name="answer" type="boolean"/>
       </xsd:complexType>
    </xsd:schema>
</types>
```

- binding:

  The bindings describe the protocol that is used to access a portType, as well as the data formats for the messages that are defined by a particular portType element. See Example 6-110.

  Example 6-110 binding snippet inside wsdl file

```
<binding name="EightBallBinding" type="ebs:EightBall">
     <soap:binding style="rpc"
transport="schemas.xmlsoap.org/soap/http">
        <operation name="ebs:getAnswer">
     <soap:operation soapAction="urn:EightBall"/>
        <input>
          <soap:body namespace="urn:EightBall" ... />
     <!--...
```

  The services and ports define the location of the Web service.

- Service:

  The service contains the Web service name and a list of ports.

- Ports:

  The ports contain the location of the Web service and the binding used for service access. See Example 6-111.

  Example 6-111 Services and port section inside wsdl file

```
<service name="EightBall">
    <port binding="ebs:EightBallBinding" name="EightBallPort">
        <soap:address location="localhost:8080/axis/EightBall"/>
    </port>
</service>
```

Multipart WSDL best practices

WebSphere Application Server supports deployment of Web Services using a multipart Web Services Description Language (WSDL) file. In multipart WSDL files, an implementation WSDL file contains the wsdl:service. This implementation WSDL file imports an interface WSDL file, which contains the other WSDL constructs. This supports multiple Web Services using the same WSDL interface definition.

The <wsdl:import> element indicates a reference to another WSDL file. If the <wsdl:import> element location attribute does not contain a URL, that is, it contains only a file name, and does not begin with http://, https:// or file://, the imported file must be located in the same directory and must not contain a relative path component. For example, if META-INF/wsdl/A_Impl.wsdl is in your module and contains the <wsdl:import="A.wsdl" namespace="..."/> import statement, the A.wsdl file must also be located in the module META-INF/wsdl directory.

We recommend that you place all WSDL files in either the META-INF/wsdl directory, if you are using Enterprise JavaBeans (EJB), or the WEB-INF/wsdl directory, if you are using JavaBeans components, even if relative imports are located within the WSDL files. Otherwise, implications exist with the WSDL publication when you use a path like <location="../interfaces/A_Interface.wsdl"namespace="..."/>. Using a path like this example fails because the presence of the relative path, regardless of whether the file is located at that path or not. If the location is a Web address, it must be readable at both deployment and server startup.

WSDL publication

You can publish the files located in the META-INF/wsdl or the WEB-INF/wsdl directory through either a URL address or file, including WSDL or XML Schema Definition (XSD) files. For example, if the file referenced in the <wsdl-file> element of the webservices.xml deployment descriptor is located in the META-INF/wsdl or the WEB-INF/wsdl directory, it is publishable. If the files imported by the <wsdl-file> are located in the wsdl/ directory or its subdirectory, they are publishable.

If the WSDL file referenced by the <wsdl-file> element is located in a directory other than wsdl, or its subdirectories, the file and its imported files, either WSDL or XSD files, which are in the same directory, are copied to the wsdl directory without modification when the application is installed. These types of files can also be published.

If the <wsdl-file> imports a file located in a different directory (a directory that is not -INF/wsdl or a subdirectory), the file is not copied to the wsdl directory and is not available for publishing.

> **Note:** For JAX-WS Web Services, there is an annotation that can be used to specify the location of the WSDL. This is the @webservice annotation and the attribute is WSDLLocation. The WSDLLocation attribute is optional. If it is not specified, then WSDL is generated and published from the information that is found in the Web service classes.

In the following sections, we discuss the concepts of JAX-WS and JAX-RPC.

JAX-RPC

JAX-RPC stands for Java API for XML-Based RPC, also known as JSR 101. It is a specification that describes Java Application Programming Interfaces (APIs) and conventions for building Web Services and Web service clients that use remote procedure calls (RPC) and XML. It standardizes the Java to WSDL and WSDL to Java mappings, and provides the core APIs for developing Web Services and Web service clients on the Java platform. Often used in a distributed client/server model, an RPC mechanism enables clients to execute procedures on other systems.

The current release of JAX-RPC requires the support of SOAP over HTTP for interoperability purposes. The SOAP specification defines message structure, encoding rules, and conventions for exchanging information in the RPC programming model. These calls and responses are transmitted as SOAP messages over HTTP. In this release, JAX-RPC supports SOAP 1.1 and HTTP 1.1. For more information on SOAP, refer to SOAP.

Although the underlying run-time mechanisms (for example, Java to WSDL serialization, WSDL to Java deserialization, protocol and transport) are very complex, the API hides this complexity from the application developer. On the server side, the developer can provide a Service Endpoint Interface (SEI); alternately an SEI can be created using the Web Services wizards. An SEI is an interface written in the Java programming language that specifies the remote procedures of a Web service. The developer also provides the implementation of a Web service, in the form of one or more Java classes that implement methods of the same signature as those on the SEI. Client programs are also easy to code. A client creates a proxy, a local object representing the service implementation and the SEI, and then simply invokes methods on the proxy.

JAX-RPC is highly interoperable: a JAX-RPC client can access a Web service that is not running on the Java platform and vice versa. This flexibility is possible because JAX-RPC uses SOAP over HTTP, and the Web Service Description Language (WSDL). JAX-RPC was designed to support WS-I.

JAX-WS

Java API for XML-Based Web Services (JAX-WS), which is also known as JSR-224, is the next generation Web Services programming model that extends the foundation provided by the Java API for XML-based RPC (JAX-RPC) programming model. Using JAX-WS, developing Web Services and clients is simplified with greater platform independence for Java applications by the use of dynamic proxies and Java annotations.

The WebSphere Application Server Version 6.1 Feature Pack for Web Services extends the capabilities of this product to introduce support for the JAX-WS 2.0 programming model. JAX-WS 2.0 is a new programming model that simplifies application development through support of a standard, annotation-based model to develop Web Service applications and clients.

The JAX-WS 2.0 programming standard strategically aligns itself with the current industry trend toward a more document-centric messaging model and replaces the remote procedure call programming model as defined by JAX-RPC. Although this product still supports the JAX-RPC programming model and applications, JAX-RPC has limitations and does not support many current document-centric services. JAX-WS is the strategic programming model for developing Web Services and is a required part of the Java EE 5 platform.

Implementing the JAX-WS programming standard provides the enhancements for developing Web Services and clients that are explained in following sections.

Better platform independence for Java applications

Using JAX-WS APIs, developing Web Services and clients is simplified with better platform independence for Java applications. JAX-WS takes advantage of dynamic proxies whereas JAX-RPC uses generated stubs. The dynamic proxy client invokes a Web service based on a Service Endpoint Interface (SEI) which is generated or provided. The dynamic proxy client is similar to the stub client in the JAX-RPC programming model.

Although the JAX-WS dynamic proxy client and the JAX-RPC stub client are both based on the Service Endpoint Interface (SEI) that is generated from a WSDL file, there is a major difference. The dynamic proxy client is dynamically generated at run time using the Java 5 dynamic proxy functionality, while the JAX-RPC-based stub client is a non-portable Java file that is generated by tooling. Unlike the JAX-RPC stub clients, the dynamic proxy client does not require you to regenerate a stub prior to running the client on an application server for a different vendor because the generated interface does not require the specific vendor information. Refer to Chapter 4 of the JAX-WS 2.0 specification for more information on using dynamic proxy clients.

Annotations

JAX-WS introduces support for annotating Java classes with metadata to indicate that the Java class is a Web service. JAX-WS supports the use of annotations based on the Metadata Facility for the Java Programming Language (JSR 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations that are defined by the JAX-WS 2.0 specification. Using annotations in the Java source and in the Java class simplifies development of Web Services by defining some of the additional information that is typically obtained from deployment descriptor files, WSDL files, or mapping metadata from XML and WSDL files into the source artifacts.

We can embed a simple @WebService tag in the Java source to expose the bean as a Web service as described in Example 6-112.

Example 6-112 @WebService annotation usage

```
@WebService

public class QuoteBean implements StockQuote {

       public float getQuote(String sym) { ... }

}
```

The annotation @WebService tells the server runtime environment to expose all public methods on that bean as a Web service. Additional levels of granularity can be controlled by adding additional annotations on individual methods or parameters. Using annotations makes it much easier to expose Java artifacts as Web Services. In addition, as artifacts are created from using some of the top-down mapping tools starting from a WSDL file, annotations are included within the source and Java classes as a way of capturing the metadata along with the source files.

> **Note:** For more information about Java annotations, see the following URL:
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_jaxwsannotations.html

Invoking Web Services asynchronously

With JAX-WS, Web Services can be called both synchronously and asynchronously. JAX-WS adds support for both a polling mechanism and callback mechanism when calling Web Services asynchronously. Using a polling model, a client can issue a request, get a response object back, which is polled to determine whether the server has responded. When the server responds, the actual response is retrieved. Using the polling model, the client can continue to process other work without waiting for a response to return. Using the callback model, the client provides a callback handler to accept and process the inbound response object. Both the polling and callback models enable the client to focus on continuing to process work while providing for a more dynamic and efficient model to invoke Web Services.

A Web service interface has methods for both synchronous and asynchronous requests. Asynchronous requests are identified in bold in Example 6-113.

Example 6-113 WebService interface with sync and async operations

```
@WebService
public interface CreditRatingService {
```

```
      // sync operation
      Score      getCreditScore(Customer customer);
      // async operation with polling
      Response<Score> getCreditScoreAsync(Customer customer);
      // async operation with callback
      Future<?> getCreditScoreAsync(Customer customer,
         AsyncHandler<Score> handler);
}
```

The asynchronous invocation that uses the callback mechanism requires an additional input by the client programmer. The callback handler is an object that contains the application code to be executed when an asynchronous response is received. Example 6-114 is a code example for an asynchronous callback handler.

Example 6-114 Asynchronous callback handler

```
Future<?> invocation = svc.getCreditScoreAsync(customerFred,
        new AsyncHandler<Score>() {
            public void handleResponse (
               Response<Score> response)
             {
               Score score = response.get();
               // do work here...
             }
        });
```

Example 6-115 is a code example for an asynchronous polling client.

Example 6-115 Asynchronous polling client

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerFred);

while (!response.isDone()) {
   // do something while we wait
}
// no cast needed, thanks to generics
Score score = response.get();
```

Data binding with JAXB 2.0

JAX-WS leverages the JAXB 2.0 API and tools as the binding technology for mappings between Java objects and XML documents. JAX-WS tooling relies on JAXB tooling for default data binding for two-way mappings between Java objects and XML documents. JAXB 2.0 data binding replaces the data binding described by the JAX-RPC specification. For more information about JAXB, see “JAXB” on page 484.

Dynamic and static clients

The dynamic client programming API for JAX-WS is called the dispatch client (javax.xml.ws.Dispatch). The dispatch client is an XML messaging oriented client. The data is sent in either PAYLOAD or MESSAGE mode. When using the PAYLOAD mode, the dispatch client is only responsible for providing the contents of the <soap:Body> element and JAX-WS adds the <soap:Envelope> and <soap:Header> elements. When using the MESSAGE mode, the dispatch client is responsible for providing the entire SOAP envelope including the <soap:Envelope>, <soap:Header>, and <soap:Body> elements and JAX-WS does not add anything additional to the message. The dispatch client supports asynchronous invocations using a callback or polling mechanism, </soap:Body></soap:Header></soap:Envelope></soap:Header></soap:Envelop e></soap:Body>

The static client programming model for JAX-WS is the called the proxy client. The proxy client invokes a Web service based on a Service Endpoint interface (SEI) which is generated or provided.

Message Transmission Optimization Mechanism support

Using JAX-WS, you can send binary attachments such as images or files along with Web Services requests. JAX-WS adds support for optimized transmission of binary data as specified by Message Transmission Optimization Mechanism (MTOM).

Multiple payload structures

JAX-WS exposes the following binding technologies to the user: XML Source, SOAP Attachments API for Java (SAAJ) 1.3, and Java Architecture for XML Binding (JAXB) 2.0. XML Source enables a user to pass a javax.xml.transform.Source into the runtime which represents the data in a Source object to be passed to the runtime. SAAJ 1.3 now has the ability to pass an entire SOAP document across the interface rather than just the payload itself. This is done by the client passing the SAAJ SOAPMessage object across the interface. JAX-WS leverages the JAXB 2.0 support as the data binding technology of choice between Java and XML.

SOAP 1.2 support

Support for SOAP 1.2 has been added to JAX-WS 2.0. JAX-WS supports both SOAP 1.1 and SOAP 1.2 so that you can send binary attachments such as images or files along with Web Services requests. JAX-WS adds support for optimized transmission of binary data as specified by MTOM.

JAX-WS and EJB 3.0

JAX-WS also works with EJB 3.0 to simplify the programming model. For example, the code in Example 6-116 shows how easy it is to make an EJB 3.0 POJO into a Web service.

Example 6-116 Stateless bean expose as a web service

```
@WebService public interface StockQuote {
    public float getQuote(String sym);
}

@Stateless public class QuoteBean implements StockQuote {
    public float getQuote(String sym) { ... }
}
```

In “Using Web services” on page 277, you can see more about stateless session beans exposed as Web services.

> **Note:** For more information on JAX-WS, refer to the official JSR-224 specification at the following link:
>
> http://jcp.org/en/jsr/detail?id=224

JAXB

Java Architecture for XML Binding (JAXB) is a Java technology that provides an easy and convenient way to map Java classes and XML schema for simplified development of Web Services. JAXB leverages the flexibility of platform-neutral XML data in Java applications to bind XML schema to Java applications without requiring extensive knowledge of XML programming.

WebSphere Application Server Version 6.1 Feature Pack for Web Services provides JAXB 2.0 standards.

JAXB is an XML to Java binding technology that supports transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB consists of a runtime application programming interface (API) and accompanying tools that simplify access to XML documents. JAXB also helps to build XML documents that both conform and validate to the XML schema.

JAXB provides the xjc schema compiler tool, the schemagen schema generator tool, and a runtime framework. You can use the xjc schema compiler tool to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. You can also start with a set of JavaBeans and use the schemagen schema generator tool to create the XML schema. When the mapping between XML schema and Java classes exists, XML instance documents can be converted to and from Java objects through the use of the JAXB binding runtime API. Data stored in XML documents can be accessed without having to understand the data structure. You can then use the resulting Java classes to assemble a Web Services application.

JAXB annotated classes and artifacts contain all the information required by the JAXB runtime API to process XML instance documents. The JAXB runtime API supports marshaling of JAXB objects to XML and unmarshaling the XML document back to JAXB class instances. Optionally, you can use JAXB to provide XML validation to enforce both incoming and outgoing XML documents to conform to the XML constraints defined within the XML schema.

JAXB is the default data binding technology used by the Java API for XML Web Services (JAX-WS) 2.0 tooling and implementation within this product. You can develop JAXB objects for use within JAX-WS applications.

You can also use JAXB independently of JAX-WS when you want to leverage the XML data binding technology to manipulate XML within your Java applications.

Figure 6-34 illustrates the JAXB architecture.

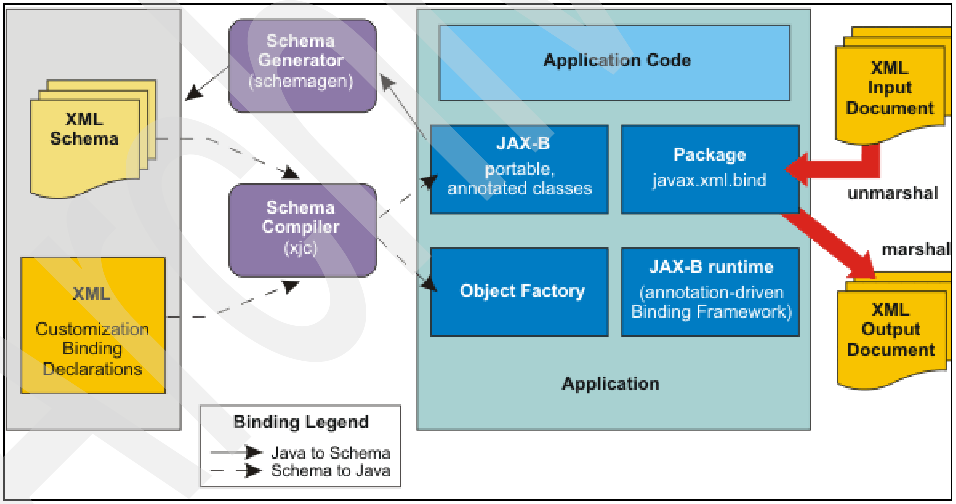


Figure 6-34 JAXB architecture

Figure 6-34 shows that we can generate from a schema the class to be used or from class the schema. When, for example, we generated the class from schema, this class allows the mapping between an XML and objects that is done in unmarshal (XML to Object) and marshal (Object to XML) operations. We explain more about usage of JAXB in the next section.

Using JAXB for XML data binding

You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. An XML schema defines the data elements and structure of an XML document. JAXB technology provides tooling to enable you to convert your XML documents to and from Java objects. Data stored in an XML document is accessible without the requirement to understand the XML data structure.

JAXB annotated classes and artifacts contain all the information that the JAXB runtime API has to process XML instance documents. The JAXB runtime API enables marshaling of JAXB objects to XML files and unmarshaling the XML document back to JAXB class instances. The JAXB binding package, javax.xml.bind, defines the abstract classes and interfaces that are used directly with content classes. In addition the package defines the marshal and unmarshal APIs.

You can optionally use JAXB binding customizations to customize generated JAXB classes by overriding or extending the default JAXB bindings when the default bindings do not meet your business application requirements. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes. JAXB supports binding customizations and overrides to the default binding rules that you can make through various ways. For example, you can the overrides inline as annotations in a source schema, as declarations in an external bindings customization file that is used by the JAXB binding compiler, or as Java annotations within Java class files used by the JAXB schema generator. See the JAXB specification for information regarding binding customization options in the following link:

http://jcp.org/en/jsr/detail?id=222

Using JAXB, you can manipulate data objects in the following ways:

- Use the schema generator schemagen command to generate an XML schema from Java classes. See more information in the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_jaxbjava2schema.html

- Use the schema compiler xjc command to create a set of JAXB-annotated Java classes from an XML schema. See more information in the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_jaxbschema2java.html

- After the mapping between XML schema and Java classes exists, use the JAXB binding runtime to convert XML instance documents to and from Java objects. See more information in the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_jaxbmarshalxml.html

JAXB and Plain Old Java Objects

Beyond other features, JAXB provides a standard mapping between POJOs (Example 6-117) and XML schemas (Example 6-118), using annotations on POJOs.

Example 6-117 POJOs example referring to xml using JAXB

```
@XmlType
public class Trade {
        @XmlElement(name="tickerSymbol")
        public String symbol;
        @XmlAttribute
        int getQuantity() {...}
        void setQuantity() {...}
}
```

Example 6-118 Schema correlation

```
<xs:complexType name="trade">
        <xs:sequence>
        <xs:element
                name="tickerSymbol"
                type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="quantity"
                type="xs:int"/>
</xs:complexType>
```

SOAP

SOAP (formerly known as Simple Object Access Protocol) is a lightweight protocol for the exchange of information in a decentralized, distributed environment. A SOAP message is a transmission of information from a sender to a receiver. SOAP messages can be combined to perform request/response patterns.

SOAP is transport independent but is most commonly carried over HTTP in order to run with the existing Internet infrastructure. SOAP enables the binding and usage of discovered Web Services by defining a message path for routing messages. SOAP is used to query UDDI for Web Services. The workbench supports SOAP 1.1.

SOAP is an XML-based protocol that defines three parts to every message:

- Envelope. The envelope defines a framework for describing what is in a message and how to process it. A SOAP message is an envelope containing zero or more headers and exactly one body. The envelope is the top element of the XML document, providing a container for control information, the address of a message, and the message itself. Headers transport any control information such as quality-of-service attributes. The body contains the message identification and its parameters. Both the headers and the body are child elements of the envelope.
- Encoding rules. The set of encoding rules expresses instances of application-defined data types. Encoding rules define a serialization mechanism that can be used to exchange instances of application-defined data types. SOAP defines a programming language-independent data type scheme based on XSD plus encoding rules for all data types defined according to this model. SOAP encoding is not WS-I compliant and thus the Literal use (which is no encoding) is suggested for interoperable Web Services and required for WS-I compliance.
- Communication styles. Communications can follow a remote procedure call (RPC) or message-oriented (Document) format. These are discussed below.

Binding styles

SOAP supports two different communication styles:

- Remote procedure call (RPC): Invocation of an operation returning a result. Typically used with SOAP encoding, which is not WS-I compliant.
- Document Style: Also known as document-oriented or message-oriented style. This style provides a lower layer of abstraction, and requires more programming work.

Encoding styles

In distributed computing environments, encoding styles define how data values defined in the application can be translated to and from a particular protocol format. The translation process is know as serialization and deserialization.

The SOAP specification defines the SOAP encoding style:

- SOAP encoding: The SOAP encoding style allows you to serialize/deserialize values of data types from the SOAP data model. This encoding style is defined in the SOAP 1.1 standard, and is not WS-I compliant.

WSDL defines the Literal XML encoding style:

- Literal XML: Literal refers to the fact that the document should be read as-is, or unencoded. The document is serialized as XMI, meaning that the message XML complies with the Schema in the WSDL. When using Literal encoding, each message part references a concrete schema definition. Literal encoding is WS-I compliant.

Data model

The purpose of the SOAP data model is to provide a language-independent abstraction for data types used by common programming language types. It consists of:

- Simple XSD types. For example int, string, and date.
- Compound types. There are two kinds of compound types, structs and arrays. Structs are named aggregate types in which each element has a unique name or XML tag. Arrays have elements that are identified by position, not by name.

All elements and identifiers comprising the SOAP data model are defined in the namespace URI. The SOAP standard defines the rules for how data types can be constructed. A project specific XML schema must define the actual data types. The elements of the SOAP specification are defined in http://schemas.xmlsoap.org/soap/envelope/ and http://schemas.xmlsoap.org/soap/encoding/

SOAP implementations

Different implementations of the SOAP protocol are available today. For example, the Apache Foundation provides Apache SOAP, which grew out of an IBM project called SOAP4J, as well as Apache Axis and the IBM WebSphere run-time environment. The provided Web Services tools support Apache SOAP 2.3, Axis 1.0, and IBM WebSphere implementations.

Mappings

A mapping defines an association between a qualified XML element name, a Java class name, and an encoding style. The mapping specifies how, under the given encoding, an incoming XML element with a fully qualified name is converted to a Java class and vice versa.

For more information on Apache SOAP, refer to:

`http://xml.apache.org/soap`

For more information on SOAP, refer to:

`http://www.w3.org/TR/SOAP`

SAAJ

The SOAP with Attachments API for Java (SAAJ) interface is used for SOAP messaging that provides a standard way to send XML documents over the Internet from a Java programming model. SAAJ is used to manipulate the SOAP message to the appropriate context as it traverses through the runtime environment.

The WebSphere Application Server V6.1 Feature Pack for Web Services introduces the Java API for XML Web Services (JAX-WS) programming model, which adds more enhancements to the use of Web Services applications. These enhancements include support for SOAP 1.2 messages, which are supported by SAAJ 1.3.

The Java API for XML-Based RPC (JAX-RPC) programming model supports SAAJ 1.2 to manipulate the XML.

New or updated for this feature pack, the JAX-WS programming model supports SAAJ 1.2 and 1.3.

You can review the differences in the SAAJ 1.2 and SAAJ 1.3 specifications at the following link:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_saajverdiffs.html

How messages are used in Web Services

Web Services use XML technology to exchange messages. These messages conform to XML schema. When developing Web Services applications, there are limited XML APIs to work with, for example, Document Object Model (DOM). It is more efficient to manipulate the Java objects and have the serialization and deserialization completed during run time.

Web Services uses SOAP messages to represent remote procedure calls between the client and the server. Typically, the SOAP message is deserialized into a series of Java value-type business objects that represent the parameters and return values. In addition, the Java programming model provides APIs that support applications and handlers to manipulate the SOAP message directly. Because there are a limited number of XML schema types that are supported by the programming models, the specification provides the SAAJ data model as an extension to manipulate the message.

To manipulate the XML schema types, you have to map the XML schema types to Java types with a custom data binder.

The SAAJ interface

The SAAJ-related classes are located in the javax.xml.soap package. SAAJ builds on the interfaces and abstract classes and many of the classes begin by invoking factory methods to create a factory such as SOAPConnectionFactory and SOAPFactory.

The most commonly used classes are:

- SOAPMessage: Contains the message, both the XML and non-XML parts
- SOAPHeader: Represents the SOAP header XML element
- SOAPBody: Represents the SOAP body XML element
- SOAPElement: Represents the other elements in the SOAP message

Other parts of the SAAJ interface include:

- MessageContext: Contains a SOAP message and related properties
- AttachmentPart: Represents a binary attachment
- SOAPPart: Represents the XML part of the message
- SOAPEnvelope: Represents the SOAP envelope XML element
- SOAPFault: Represents the SOAP fault XML element

The primary interface in the SAAJ model is javax.xml.soap.SOAPElement, also referred to as SOAPElement. Using this model, applications can process an SAAJ model that uses pre-existing DOM code. It is also easier to convert pre-existing DOM objects to SAAJ objects.

Messages created using the SAAJ interface follow SOAP standards. A SOAP message is represented in the SAAJ model as a javax.xml.soap.SOAPMessage object. The XML content of the message is represented by a javax.xml.soap.SOAPPart object. Each SOAP part has a SOAP envelope. This envelope is represented by the SAAJ javax.xml.SOAPEnvelope object. The SOAP specification defines various elements that reside in the SOAP envelope; SAAJ defines objects for the various elements in the SOAP envelope.

The SOAP message can also contain non-XML data that is called attachments. These attachments are represented by SAAJ AttachmentPart objects that are accessible from the SOAPMessage object.

A number of reasons exist as to why handlers and applications use the generic SOAPElement API instead of a tightly bound mapping:

- The Web service might be a conduit to another Web service. In this case, the SOAP message is only forwarded.

- The Web service might manipulate the message using a different data model, for example a Service Data Object (SDO). It is easier to convert the message from a SAAJ DOM to a different data model.
- A handler, for example, a digital signature validation handler, might want to manipulate the message generically.

You might have to go a step further to map your XML schema types, because the SOAPElement interface is not always the best alternative for legacy systems. In this case you might want to use a generic programming model, such as SDO, which is more appropriate for data-centric applications.

The XML schema can be configured to include a custom data binding that pairs the SDO or data object with the Java object. For example, the run time renders an incoming SOAP message into a SOAPElement interface and passes it to the customer data binder for more processing. If the incoming message contains an SDO, the run time recognizes the data object code, queries its type mapping to locate a custom binder, and builds the SOAPElement interface that represents the SDO code. The SOAPElement is passed to the SDOCustomBinder.

For more information about the process of developing applications with SOAPElement, SDO, and custom binders, see the following URL:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_custombinders.html

WSIF

The Web Services Invocation Framework (WSIF) provides a Java API for invoking Web Services, independent of the format of the service or the transport protocol through which it is invoked.

Goals of WSIF

WSIF aims to extend the flexibility provided by SOAP services into a general model for invoking Web Services, irrespective of the underlying binding or access protocols.

SOAP bindings for Web Services are part of the WSDL specification, therefore when most developers think of using a Web service, they immediately think of assembling a SOAP message and sending it across the network to the service endpoint, using a SOAP client API. For example: using Apache SOAP the client creates and populates a Call object that encapsulates the service endpoint, the identification of the SOAP operation to invoke, parameters to send, and so on.

While this process works for SOAP, it is limited in its use as a general model for invoking Web Services for the following reasons:

- Web Services are more than just SOAP services.

  You can deploy as a Web service any application that has a WSDL-based description of its functional aspects and access protocols. If you are using the Java 2 platform, Enterprise Edition (J2EE) environment, then the application is available over multiple transports and protocols.

  For example, you can take a database-stored procedure, expose it as a stateless session bean, then deploy it into a SOAP router as a SOAP service. At each stage, the fundamental service is the same. All that changes is the access mechanism: from Java DataBase Connectivity (JDBC) to Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) and then to SOAP.

  The WSDL specification defines a SOAP binding for Web Services, but you can add binding extensions to the WSDL so that, for example, you can offer an enterprise bean as a Web service using RMI-IIOP as the access protocol. You can even treat a single Java class as a Web service, with in-thread Java method invocations as the access protocol. With this broader definition of a Web service, you require a binding-independent mechanism for service invocation.

- Tying client code to a particular protocol implementation is restricting.

  If your client code is tightly bound to a client library for a particular protocol implementation, it can become hard to maintain.

  For example, if you move from Apache SOAP to Java Message Service (JMS) or enterprise bean, the process can take a lot of time and effort. To avoid these problems, you require a protocol implementation-independent mechanism for service invocation.

- Incorporating new bindings into client code is hard.

  If you want to make an application that uses a custom protocol work as a Web service, you can add extensibility elements to WSDL to define the new bindings. But in practice, achieving this capability is hard.

  For example, you have to design the client APIs to use this protocol. If your application uses just the abstract interface of the Web service, you have to write tools to generate the stubs that enable an abstraction layer. These tasks can take a lot of time and effort. What you require is a service invocation mechanism that allows you to update existing bindings, and to add new bindings.

- Multiple bindings can be used in flexible ways.

  To take advantage of Web Services that offer multiple bindings, you require a service invocation mechanism that can switch between the available service bindings at run time, without having to generate or recompile a stub.

Imagine that you have successfully deployed an application that uses a Web service which offers multiple bindings. For example, suppose that you have a SOAP binding for the service and a local Java binding that lets you treat the local service implementation (a Java class) as a Web service.

The local Java binding for the service can only be used if the client is deployed in the same environment as the service. In this case, it is more efficient to communicate with the service by making direct Java calls than by using the SOAP binding.

If your clients could switch the actual binding used based on run-time information, they could choose the most efficient available binding for each situation.

- A freer Web Services environment enables intermediaries.

  Web Services offer application integrators a loosely-coupled paradigm. In such environments, intermediaries can be very powerful.

  Intermediaries are applications that intercept the messages that flow between a service requester and a target Web service, and perform some mediating task (for example logging, high-availability or transformation) before passing on the message. The Web Services Invocation Framework (WSIF) is designed to make building intermediaries both possible and simple. Using WSIF, intermediaries can add value to the service invocation without requiring transport-specific programming.

WSIF architecture

A diagram depicting the Web Services Invocation Framework architecture, and a description of each of the major components of the architecture.

The Web Services Invocation Framework architecture is shown in Figure 6-35.

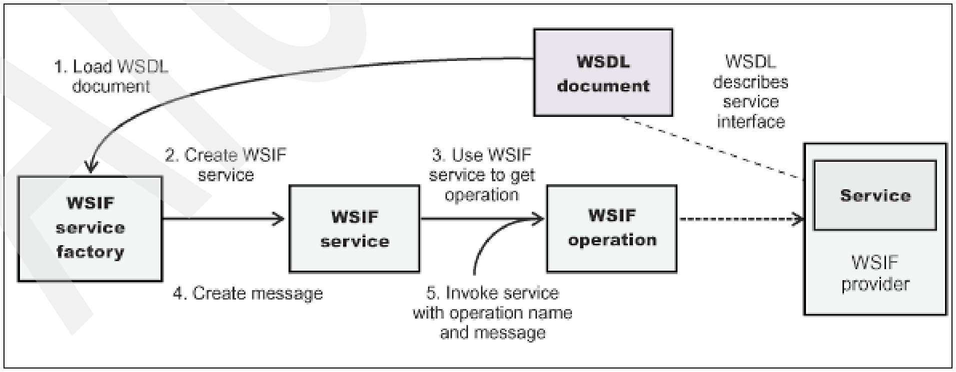


Figure 6-35 WSIF architecture

The components of this architecture include:

- WSDL document:

  The Web service WSDL document contains the location of the Web service. The binding document defines the protocol and format for operations and messages defined by a particular portType.

- WSIF service:

  The WSIFService interface is responsible for generating an instance of the WSIFOperation interface to use for a particular invocation of a service operation. For more information, see Finding a port factory or service at the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwsf_serv.html

- WSIF operation:

  The run-time representation of an operation, called WSIFOperation is responsible for invoking a service based on a particular binding. For more information, see WSIF API reference: Using ports in the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwsf_ports.html

- WSIF provider:

  A WSIF provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol. WSIF includes SOAP providers, JMS providers, Java providers and EJB providers. For more information, see Linking a WSIF service to the underlying implementation of the service in the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twsf_pr.html

WSIF usage scenarios

This topic describes two brief scenarios that illustrate the role WSIF plays in the emerging Web Services environment:

- Scenario: Redevelopment and redeployment:

  When you first implement a Web service, you create a simple prototype. When you want to move a prototype Web service into production, you often have to redevelop and redeploy it.

  The Web Services Invocation Framework (WSIF) uses the same API calls irrespective of the underlying technologies, therefore if you use WSIF:

  – You can reimplement and redeploy your services without changing the client code.

- You can use existing reliable and high-performance infrastructures like Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) and Java Message Service (JMS) without sacrificing the location-independence that the Web service model offers.

- Scenario: Service flow composition:

  A service flow typically invokes a Web service, then passes the response from one Web service to the next Web service, perhaps performing some transformation in the middle.

  There are two key aspects to this flow that WSIF provides:

  - A representation of the service invocation based on the metadata in WSDL.
  - The ability to build invocations based solely on the portType, which can therefore be used in any implementation.

  For example, imagine that you built a meta-service that uses a number of services to build a process. Initially, several of those services are simple Java bean prototypes that are written and exposed through SOAP, but you plan to reimplement some of them as EJB components, and to out-source others.

  If you use SOAP, it ties up multiple threads for every onward invocation, because they pass through the Web server and servlet engine and on to the SOAP router. If you use WSIF to call the beans directly, you get much better performance compared to SOAP and you do not lose access or location transparency. Using WSIF, you can replace the Java bean implementations with EJB implementations without changing the client code. To move some of the Web Services from local implementations to external SOAP services, you just update the WSDL.

We have concluded some basic concepts around Web Services, including some concepts related to new features in WebSphere Application Server 6.1. Now we go on to the best practices.

6.12.2 Web Services architectures and best practices

In this section we consider some best practices in applying Web Services for solving tough architectural problems.

Some Web Services do’s and don'ts

There is a common set of emotions that go along with adopting any new technology. First, when you begin to hear the buzz about a technology, you start to think that it might be useful in solving your particular problems, and feel positively inclined toward it. As you learn more, your excitement grows — perhaps a short proof-of-concept is successful and leads you to jump in with both feet and adopt the new technology for a big new project. Then, the reality of the state of the technology begins to set in, and you start to find the limitations of the new technology. At this point, you might be able to muddle through and make the project successful despite the technology's limitations, or the project might simply crash. The old adage “all panaceas become poison” applies to most new technologies, and it applies no less to Web Services.

In the past two or three years since Web Services have started to be used in practical applications, a number of basic dos and don'ts have emerged about when Web Services are practical, and when they are not. Next, we'll examine some of these basic principles, and discuss some situations where disregarding them have made projects go awry.

Principle: Try not to use XML-based Web Services between the layers of a logical application

Web Services function best where they complement other J2EE technologies, not replace them. For instance, a wonderful way to use Web Services is when connecting from an application client running out on the global internet to business logic written in EJBs inside WebSphere Application Server. Here you get a nice, clean separation of communication between the Controller and Domain layers of your application. This is the same place in which you would use EJBs and so, if you consider Web Services as another object distribution mechanism, then you can see why this would be appropriate. SOAP over HTTP can work in places where RMI over IIOP cannot, and so this allows the XML-based Web Services to complement the existing EJBs.

However, where people often go wrong with this is to assume that if this works between one pair of layers, it would work well between another. For instance, a common anti-pattern that we have seen far too often is a design where a persistence layer is wrapped inside an XML API and then placed in a process separate from the business logic that has to invoke the persistence layer. In versions of this design, we have seen people actually serialize JavaTM objects into XML, send them over the network, deserialize them, perform a database query with the objects thus sent in as an argument, convert the database result set to XML, and then send the result set back across the network, only to be converted into Java objects and finally operated on. There are several major problems with this approach:

1. Persistent objects should ALWAYS remain local to the business object that operates on them. The overhead of serialization and deserialization is something you want to avoid whenever possible.
2. There is not yet a workable, fully-implemented transaction specification for Web Services. In EJBs with RMI-IIOP you have the option (although you are not required to) of including persistence operations in an outer transaction

scope if you use Entity Beans or Session beans with Mapper objects if you so choose. If you introduce a layer of Web Services between the persistent objects and the business objects operating on them, then you lose that ability.

In general, XML Web Services are not appropriate for fine grained interactions, even less so than RMI-IIOP. For instance, do not put it between the view layer and the controller layer of an application. The overhead of the parsing/XML generation and the garbage generation overhead kills the performance of your overall application.

Principle: Be very careful when using Web Services between application servers

In many ways, interoperability between systems is the raison d'etre of Web Services. So, if you are connecting to a system written using Microsoft® .NET, then the use of Web Services is almost a given. While you could use other mechanisms like WebSphere Application Server's COM support, the best solution for interoperability going forward for both the Microsoft and IBM platforms is probably Web Services.

Sometimes it makes sense as when connecting disparate Java application servers from different vendors, but this is a less common occurrence. It is possible (for instance) to connect to EJBs written in WebSphere from a JBoss or WebLogic server by using the WebSphere Thin Application Client. This would be a much better performing solution than one using HTTP and SOAP based Web Services. On the other hand, a more common occurrence is when you want asynchronous invocation of business logic written either in another application server or in some sort of legacy server. In this case, sending XML over JMS makes a lot of sense, and if you wrap your document-oriented XML in a SOAP envelope then it makes even more sense; you can take advantage of the header structure of SOAP and even possibly gain some out-of-the box features like WS-Security support.

Addressing the limitations of Web Services

Web Services have proven to be a useful approach for addressing some of the interesting problems of distributed objects. Since its introduction, SOAP over HTTP has become nearly the de facto standard for application-to-application communication over the Internet. With major Web sites such as UPS, Amazon, and Google supporting the Web Services standards, this technology has become quite entrenched in the corporate I/T world.

However, when we look at using Web Services in an intranet environment, the issues are not quite as clearly defined as they are when discussing systems made available over the global Internet. Web Services provide a number of advantages when using them over the global Internet. For instance:

- Since the most common transport protocol for Web Services is HTTP, which is also the protocol that most of the Internet infrastructure is built around handling, managing, load-balancing, and allowing access to applications through HTTP is often much less troublesome than allowing access through other protocols. For instance, most corporations already employ a “DMZ” firewall policy that allows a set of protected servers to receive incoming traffic on HTTP or HTTPs but over no other protocols. This is rather an ironic situation; most businesses allow HTTP because it is believed to be a “safe” protocol for accessing web content. Now with Web Services, all sorts of business traffic can now flow through the corporate firewall. Simply assuming that because you Web Services traffic flows over HTTP that it is “safe” is inappropriate. Instead, you have to open a dialogue with your security organization on what business functions should be exposed over the internet, and what precautions should be taken to protect them.
- Web Services are quickly becoming ubiquitous. This is due to the curious historical occurrence that for the first time, both Microsoft and the Java industry have backed a single distributed technology. Since SOAP engines and tools that understand the basic protocols are now common, there is no requirement that a Web Services client be written using the same tool as a Web Services server. This enables communication between companies over the global Internet since business partners do not have to assume anything about the way in which either side of the conversation is implemented.

However, when we are considering a system in which the majority of users are working within a corporate intranet, some of the following hurdles to overcome with Web Services and SOAP become more crucial. They are:

- First and foremost, we have found that with the current SOAP engines that there is literally an order-of-magnitude performance difference between Web Services calls and equivalent calls using the remote EJB protocol (RMI-IIOP). While a very large-grained approach with Web Services might be applicable to infrequent communication between business partners, using them in tightly coupled, high-volume internal applications is likely to be inappropriate. For instance, a call-center application where there are dozens or hundreds of requests per minute from each user, is probably not a good candidate for Web Services. In a situation like that the overhead of generating and parsing XML is problematic.
- Even though the industry is making progress with standardized authentication and authorization for Web Services, unfortunately most of the current set of J2EE products do not yet provide full support for this to the extent that they do for the J2EE protocols (like RMI-IIOP).

Looking forward into time, many of these problems could be addressed in a very elegant way by an expansion of the promised multi-protocol support for Web Services. For instance, if a standard RMI-IIOP binding for WSDL was available through JAX-RPC or JAX-WS, then you could simply choose the right port for the job in your WSDL. However, pending standardization of this approach, these problems are still real issues.

So, since no one distribution approach solves all problems, what many organizations have concluded is that they have to support multiple distributed-object protocols and access mechanisms within their enterprise. A single application API might have to be available as an external Web Service using SOAP over HTTP, over RMI-IIOP for internal remote clients, using Local EJB references within an application server, and potentially even using SOAP over MQ for asynchronous interaction. There are two pieces to solving this puzzle; first, how do we provide access to business logic over multiple distribution protocols, and then what client programming model do we use to provide access to the remote business logic.

WSDL and WSIF

You would think that a pattern that (in retrospect) is as obvious as this one would have managed to find its way into one or more open-source projects or commercial products. In fact, that has happened, although it has appeared to have “slipped under the radar screen” of most developers. The key difference here is that WSIF is designed around using WSDL as a “normalized description” of a piece of software using a protocol. If you have a WSDL document with different bindings for a set of technologies (such as SOAP over HTTP or EJBs) then you could use WSIF providers to connect to remote objects implemented using different distribution technologies. More about WSIF see at , “WSIF” on page 492.

6.12.3 Best practices while developing Web Services

In this section we go over best practices for Web Services.

Use simple data types

Even though Web Services were designed with interoperability in mind, it is best to use simple data types where possible. By simple, we mean integers and strings. In addition, compound data types (comparable with structs in C, or records in Pascal) and arrays of simple types are simple.

Anything that does not fall into this pattern should be used carefully. In particular, the Java collection classes and similarly complex data types should be avoided altogether because there might be no proper counterparts at the client side.

Avoid fine-grained Web Services

Web Services use a very simple, yet powerful format for their main protocol: XML. While being able to read and structure XML documents with just any simple text editor eases the use of SOAP, the process of automatically creating and interpreting XML documents is more complex.

Therefore, there is always a point where the complexity of dealing with the

protocol is higher than performing the actual computation. To avoid this problem, design Web Services that perform more complex business logic. This can also mean that your Web service allows for bulk processing instead of multiple invocations with one parameter only.

Avoid Web Services for intra-application communication

This best practice is closely related to the previous practice. Intra-application communication (that is, communication within an application) is generally not exposed to any third-party clients. Therefore, it is not necessary to allow for an interoperable interface in this case. However, try to take into consideration that this might change in the future.

Use short attribute, property, and tag names

This is another practice that is closely related to the previous practices. As each attribute, property, and tag name is transmitted verbatim, the length of a message is directly dependent on the length on the attribute and property names. The general guideline is the shorter the attribute, property, and tag names are, the shorter the transmitted message and the faster the communication and processing.

Avoid deep nesting of XML structures

This is yet another practice that is closely related to the previous practices. Because parsing of deeply nested XML structures increases processing time, deeply nested compound data types should be avoided. This also increases comprehension of the data type itself.

Apply common sense (also known as being defensive)

If a standard or specification is not clear enough, try to implement your Web service such that it can handle any of the interpretations you can think of. An example from a different, although not less instructive, domain is the following excerpt from the TCP/IP specification (RFC 793):

`http://www.ietf.org/rfc/rfc0793.txt?number=793`

Postel's Law: Be conservative in what you do, be liberal in what you accept from others.

Use caching of Web Services as provided by the platform

WebSphere Application Server provides an excellent caching framework that allows for caching of information at various levels. Among these, you can also cache Web service requests, thus save processing time. The cache is easy to set up and can be used on any existent Web service. In addition, caching can be also turned on at the client, thus allowing for even more performance improvements.

Minimize parsing of XML data

If a business function is to be exposed as an XML Web service that leverages SOAP for both internal consumption (EAI) and for external consumption by business partners (B2B), intermediaries such as gateways or service agents should avoid or minimize parsing of the SOAP Body. If a gateway component is used to centralize access of Web Services to the Internet, but no network transport or message manipulation is required (such as SOAP/HTTP to RMI/IIOP), then the gateway should not perform parsing of the SOAP body.

Many system management vendors today provide service agents that front-end the actual Web Services. These components rely on business context information within the SOAP body, such as business partner IDs, transaction correlators, message IDs, and authorization codes in providing their system management capabilities. Using the business context, the service agents provide statistics on business events, enforce business policies, and route requests to meet quality of service commitments. Recently, the Web Services Gateway in WebSphere Application Server V5.1 supports partial parsing of SOAP messages. Likewise, system management vendors have recently started to provide the capability to partially parse SOAP messages to minimize their impact on performance, so it's vital that these capabilities be utilized.

JavaBean versus EJB components

You must now choose whether to use JavaBeans or EJB components for your Web service providers. This decision is no different than when you're architecting your other J2EE applications. If your solution does not require the J2EE runtime support for transaction, security, and management that are enabled through the use of EJB components and the EJB container, then JavaBeans can suffice and provide better performance. If you are using EJB components and deploying them locally within the same JVM as the SOAP engine, then ensure that you deploy them such that they are called using pass by reference. By enabling pass by reference, the parameters of the method are not copied to the stack with every remote call, which can be expensive. Enabling pass by reference can improve performance up to 50%, when the SOAP engine (EJB client) and the Web service provider (EJB Server) are installed in the same application server instance, and remote interfaces are used.

Retrieval of service bindings from a UDDI registry

UDDI registries are used to publish Web Services information for dynamic discovery. Late binding of a service's access point is one use of an UDDI registry that is common today. During the client's service invocation, the UDDI registry is queried to obtain the access point (URL) of the service. This approach is often used to enable the use of backup servers in performing fail-over support for improving service availability. Likewise, it affords the service provider the ability to migrate service requests from one system to another for the purpose of maintenance without disrupting the processing of existing service requests.

However, the UDDI query can add significant path length to a request and, if done for every request, can degrade a client's performance. One approach is to front-end the local service proxy created from the service provider's WSDL with a general proxy that queries the UDDI registry and caches the access points for a given period of time. The client application calls the general proxy that calls the service proxy to invoke the service provider.

If the call to the Service proxy fails due to a non-HTTP Status code of 200, then the general proxy queries the UDDI registry again to see if an alternative address is available for use on a follow-up call to the service proxy. This allows the client application code to be simpler, the UDDI registry queries to be minimized, and the client to bind dynamically to the service in order to ensure higher availability with optimal performance.

Summary

Successfully optimizing performance for Web Services is part experience, part art, and part discipline in being systematic in your approach to measuring criteria, analyzing information, and making sound adjustments. Once you have a solution that is operational, it is an iterative process to fine tune your solution by capturing measurements from simulated loads and making adjusting and measuring again to understand their influence.

6.12.4 Web Services performance best practices

This topic presents best practices for the performance of Web Services applications.

Web Services are developed and deployed based on standards provided by the Web Services for Java 2 Platform, Enterprise Edition (J2EE) specification and the Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) programming models, and is the mechanism used to access a Web service. We'll explain performance considerations for Web Services supported by this specification.

When you develop or deploy a Web service, several artifacts are required, including a Web Services Description Language (WSDL) file. The WSDL file describes the format and syntax of the Web service input and output SOAP messages. When a Web service is implemented in the WebSphere Application Server runtime, the SOAP message is translated based on the J2EE request. The J2EE-based response is then translated back to a SOAP message.

The most critical performance consideration is the translation between the XML-based SOAP message and the Java object. Performance is high for a Web service implementation in WebSphere Application Server, however, application design, deployment and tuning can be improved. See Monitoring the performance of Web Services applications for more information about analyzing and tuning Web Services in the following URL:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_performance.html

If you are using a Web service application that was developed for a WebSphere Application Server version prior to Version 6, you can achieve better performance by running the wsdeploy command. The wsdeploy command regenerates Web Services artifact classes to increase the serialization and deserialization performance.

New or updated for this feature pack The wsdeploy command is supported by JAX-RPC applications that are used with the Feature Pack for Web Services product. The Java API for XML-Based Web Services (JAX-WS) programming model that is introduced in the Feature Pack for Web Services product does not support the **wsdeploy** command. JAX-WS applications do not require and should not run the **wsdeploy** command.

Basic considerations for a high-performance Web Services application

The following are basic considerations you should know when designing a Web Services application:

- Reduce the Web Services requests by using a few highly functional APIs, rather than several simple APIs.
- Design your WSDL file interface to limit the size and complexity of SOAP messages.
- Use the document/literal style argument when you generate the WSDL file.
- Leverage the caching capabilities offered for WebSphere Application Server.
- Test the performance of your Web service.

Additional Web Services performance features that you can leverage

- In-process optimizations for Web Services to optimize the communication path between a Web Services client application and a Web container that are located in the same application server process. For details and enabling this feature, see Web Services client to Web container optimized communication in following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rrun_inbound.html

- Access to Web Services over multiple transport protocols extends existing Java API for XML-based remote procedure call (JAX-RPC) capabilities to support non-SOAP bindings such as RMI/IIOP and JMS. These alternative transports can improve performance and quality of service aspects for Web Services. For more detailed information see RMI-IIOP using JAX-RPC in the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_multijaxrpc.html

- SOAP with Attachments API for Java (SAAJ) Version 1.2 provides a programming model for Web Services relative to JAX-RPC. The SAAJ API provides features to create and process SOAP requests using an XML API. SAAJ supports just-in-time parsing and other internal algorithms. For information about SAAJ or Web Services programming, see SOAP with Attachments API for Java in the following URL:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_saaj.html

  > **Note:** SAAJ 1.3 provides support for Web Services that are developed and implemented based on the Java API for XML Web Services (JAX-WS) programming model.

- The Web Services tooling generates higher performance custom deserializers for all JAX-RPC beans. Redeploying a V5.x application into the V6 runtime can decrease the processing time for large messages.
- Serialization and deserialization runtime is enhanced to cache frequently used serializers and deserializers. This can decrease the processing time for large messages.
- The performance of WS-Security encryption and digital signature validation is improved because of the use of the SAAJ implementation.

IBM provides considerable documentation and best practices for Web Services application design and development that details these items and more.

For a list of key Web sites that discuss performance best practices, see the following URL:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_resourceslearning2.html

6.13 References

The following sections list the references in this chapter divided by sections.

6.13.1 EIS access layer

Managing Information Access to an Enterprise Information System Using J2EE and Services Oriented Architecture, SG24-6371

http://www.redbooks.ibm.com/abstracts/sg246371.html?Open

6.13.2 JDBC and SQLJ

- Advanced DAO programming:

  http://www.ibm.com/developerworks/java/library/j-dao/
- Performance tips for the IBM Developer Kit for Java JDBC driver:

  http://publib.boulder.ibm.com/iseries/v5r1/ic2924/index.htm?info/rzaha/jdbcperf.htm
- Java theory and practice: Good housekeeping practices:

  http://www.ibm.com/developerworks/library/j-jtp03216.html
- What's new in JDBC 3.0:

  http://www.ibm.com/developerworks/java/library/j-jdbcnew/
- *Performance Monitoring and Best Practices for WebSphere on Z/OS*, SG24-7269:

  http://www.redbooks.ibm.com/abstracts/sg247269.html?Open

6.13.3 iBATIS

- Improve persistence with Apache Derby and iBATIS, Part 1: Initial configuration, semantics, and a simple test:

  http://www.ibm.com/developerworks/edu/os-dw-os-ad-ibatis1.html

- Improve persistence with Apache Derby and iBATIS, Part 2: Data definition in Derby:

  http://www.ibm.com/developerworks/edu/os-dw-os-ad-ibatis2.html

- Improve persistence with Apache Derby and iBATIS, Part 3: Transactions, caching, and dynamic SQL:

  http://www.ibm.com/developerworks/edu/os-dw-os-ad-ibatis3.html

6.13.4 JPA

- Get to know Java EE 5:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0707_barcia/0707_barcia.html

- Design enterprise applications with the EJB 3.0 Java Persistence API:

  http://www.ibm.com/developerworks/java/library/j-ejb3jpa.html

- IBM WebSphere software early programs:

  https://www14.software.ibm.com/iwm/web/cc/earlyprograms/websphere/was61ejb3/

- Leveraging OpenJPA with WebSphere Application Server V6.1:

  http://www.ibm.com/developerworks/websphere/techjournal/0612_barcia/0612_barcia.html

6.13.5 Entity Beans

- IBM WebSphere Developer Technical Journal: Understanding WebSphere Application Server EJB access intents:

  http://www.ibm.com/developerworks/websphere/techjournal/0406_persson/0406_persson.html

- Access intent service in WebSphere Info Center:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cejb_axiover.html

- Defining WebSphere extensions and bindings for EJB modules:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.etools.j2ee.ui.ws.ext.doc/topics/cebindejb.html

6.13.6 SDO

- Simplify and unify data with a Service Data Objects architecture:

  http://www.ibm.com/developerworks/webservices/library/ws-sdoarch/

- Service Components Architecture (SCA) and Service Development Objects (SDO) Submitted to OASIS:

  http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-scasdo/SCA_SDO_to_OASIS.pdf

6.13.7 JMS

- Introducing the Java Message Service:

  https://www6.software.ibm.com/developerworks/education/j-jms/index.html

- MS 1.1 simplifies messaging with unified domains:

  http://www.ibm.com/developerworks/java/library/j-jms11/

- SG24-7269 Performance Monitoring and Best Practices for WebSphere on Z/OS:

  http://www.redbooks.ibm.com/abstracts/sg247269.html?Open

6.13.8 Web Services

- Web Services resource for learning in WebSphere Application Server 6.1 infocenter:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_resourceslearning2.html

- IBM WebSphere Developer Technical Journal: Web Services Architectures and Best Practices:

  http://www.ibm.com/developerworks/websphere/techjournal/0310_brown/brown.html

7

Environmental performance considerations

This chapter discusses factors that are critical for high-performance WebSphere Application Server based applications. However, a prerequisite is to ensure that your application has a good design and architecture. The techniques and strategies outlined here can assist you in optimizing the performance of your applications. However, they cannot compensate for a poorly designed or architected application. You should apply the best practices after verifying that the basic design and architecture are appropriate for the application and scalable.

The chapter is organized into the following major sections:

7.1 What is new in V6.1

WebSphere Application Server V6.1 has a new Java Virtual Machine (JVM) designed to improve stability and performance. It provides a Java language compiler and execution environment to support the Java 2 Standard Edition (J2SE) 5 specification. This new JVM is supported on all platforms that ship with an IBM JDK. This new JVM includes a new garbage collection scheme and a new Just-In-Time (JIT) compiler. The IBM Java 5.0 JVM provides major improvements in virtual machine technology to provide significant performance and serviceability enhancements over the earlier IBM Java execution technology.

WebSphere Application Server V6.1 also includes support for running Java Specification Request (JSR) 168 compliant Portlets and JSR 116 Session Initiation Protocol (SIP) Servlets. A Portlet is a Java class and a Web component, similar to a Servlet. It implements javax.portlet.* interfaces and it is packed in a Web ARchive (WAR) File. Portlets accept incoming requests, and return markup fragments. Markup fragments is a part of a document, rather than a complete document (typically HTML, XHTML or WML). SIP is a signaling protocol. The SIP application-layer protocol allows for the creation and management of multimedia communication sessions between devices.

7.2 Application environment tuning

Within the WebSphere Application Server environment, there are many settings that can increase application performance. Many components in WebSphere Application Server have an impact on performance and tuning is highly application-dependent. The purpose of this section is to discuss those tuning parameters. Figure 7-1 shows that WebSphere has several J2EE components such as Servlets, JSPs, and EJBs, and each component has a container.

There are two main aspects to tuning:

- Tuning of code: Code tuning is covered in other chapters of this book.
- Tuning of services: In this chapter we consider the tuning of products that run Java Virtual Machines, J2EE components, or back-end systems, such as databases and Message Queue Servers, for example.

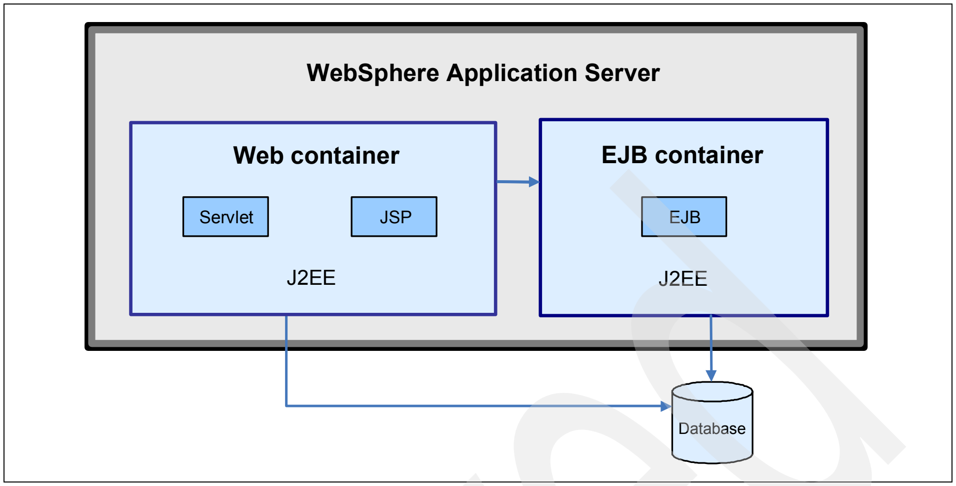


Figure 7-1 Some components and containers to WebSphere Application Server

Inside WebSphere, for example, we have several containers that belong to the J2EE specification. To get better performance, tuning of each container is very important. The access to the back-end system, such as a database, has to be tuned also.

It is important to stress once again that performance tuning is not an exact science. Factors that influence testing vary from application to application, and also from platform to platform. This section is designed to provide a primer for the reader, describing areas that can be tuned to increase performance.

The latest performance tuning information can be found in the WebSphere Application Server V6.1 Info Center.

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

Select the appropriate version of the Info Center you wish to read. Then navigate to the *Tuning performance* section in the contents pane on the left side.

7.2.1 Tuning the Java Virtual Machine

WebSphere Application Server is a Java based server and requires a Java Virtual Machine (JVM) environment to run and support the Java and J2EE applications that run on it. As part of configuring WebSphere Application Server, you can configure the Java runtime environment to tune performance and system resource usage.

A Java runtime environment provides the execution environment for Java based applications and servers such as WebSphere Application Server. Therefore the

Java configuration plays a significant role in determining performance and system resource consumption for WebSphere Application Server and the applications that run on it.

The following tasks provide specific instructions on how to perform the following types of tuning for each JVM. The tasks do not have to be performed in any specific order:

- Java memory or heap tuning
- Garbage collection tuning
- Start up versus runtime performance optimization

Other significant tuning options not described here can be found in the Java technology guides from each vendor.

Configuring the heap size

Java memory or heap tuning controls the amount of memory that is allocated for use by individual application server instances. The following command line parameters are used to adjust the minimum and maximum heap size for each application server instance. The IBM Developer Kit and Runtime Environment, Java2 Technology Edition, Version 5.0 Diagnostics Guide provides additional information on tuning the heap size. It is available on the developerWorks® Web site:

http://www.ibm.com/developerworks/java/jdk/diagnosis/

Heap size setting can set the maximum and initial heap sizes for the JVM.

In general, increasing the size of the Java heap improves throughput until the heap no longer resides in physical memory. After the heap begins swapping to disk, Java performance drastically suffers. Therefore, the maximum heap size must be low enough to contain the heap within physical memory.

The physical memory usage must be shared between the JVM and other applications running on the system, such as the database. For assurance, use a smaller heap, for example 64 MB, on machines with less memory.

Try a maximum heap of 128 MB on a smaller machine, that is, less than 1 GB of physical memory. Use 256 MB for systems with 2 GB memory, and 512 MB for larger systems. The starting point depends on the application.

If performance runs are being conducted and highly repeatable results are required, set the initial and maximum sizes to the same value. This setting eliminates any heap growth during the run. For production systems where the working set size of the Java applications is not well understood, an initial setting of one-fourth the maximum setting is a good starting value. The JVM then tries to adjust the size of the heap to the working set of the Java application.

To use the administrative console to configure the heap size:

1. In the administrative console, click **Servers** → **Application Servers** → ***server***.
2. Under Server Infrastructure, click **Java and Process Management** → **Process Definition** → **Java Virtual Machine**.
3. Specify a new value in either the **Initial heap size** or the **Maximum heap size** field. You can also specify values for both fields if you have to adjust both settings. See Figure 7-2.
4. Click **OK**.
5. **Save** your changes to the master configuration.
6. **Stop** and restart the application server.

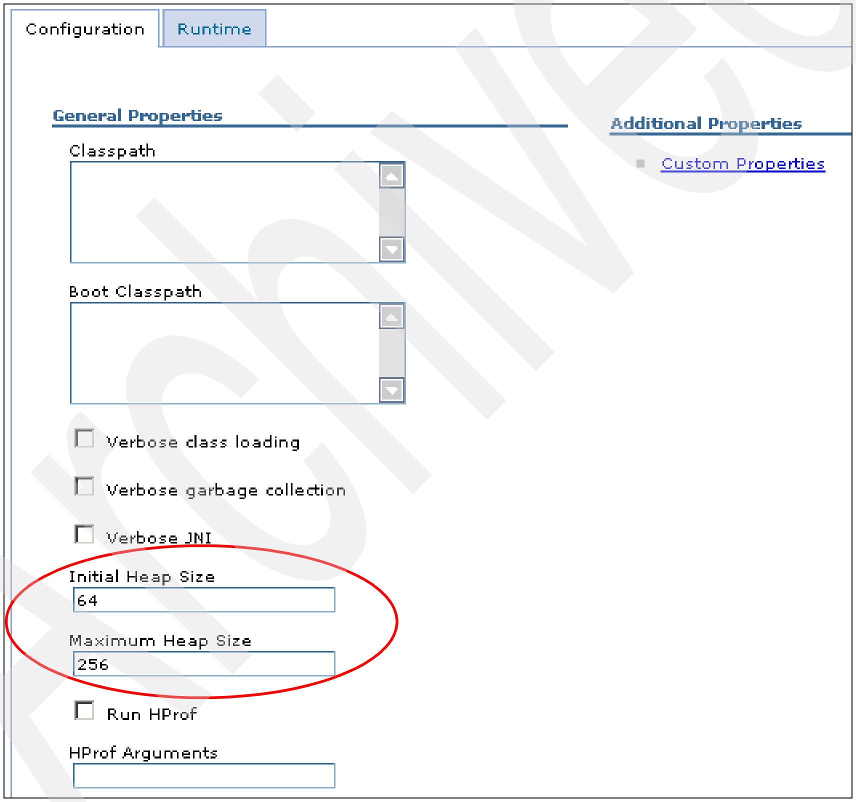


Figure 7-2 JVM Configuration

Tuning garbage collection

You can use JVM settings to configure the type and behavior of garbage collection. When the JVM cannot allocate an object from the current heap because of lack of contiguous space, the garbage collector is invoked to reclaim memory from Java objects that are no longer being used.

To adjust your JVM garbage collection settings:

1. In the administrative console, click **Servers** → **Application Servers** → ***server***.
2. Under Server Infrastructure, click **Java and Process Management** → **Process Definition** → **Java Virtual Machine.**
3. Enter the **–X** option you want to change in the **Generic JVM arguments** field. See Figure 7-3 on page 516.
4. Click **OK**.
5. **Save** your changes to the master configuration.
6. Stop and restart the application server.

The following steps describe the specific **–X** options that JVM garbage collectors support:

1. Use the Java **-X** option to view a list of memory options:

   – **-Xgcpolicy**

     Starting with Java 5.0, the IBM JVM provides four policies for garbage collection. Each policy provides unique benefits:

     - **optthruput**, which is the default, provides high throughput but with longer garbage collection pause times. During a garbage collection, all application threads are stopped for mark, sweep and compaction, when compaction is required. optthruput is sufficient for most applications.
     - **optavgpause**, which reduces garbage collection pause time by performing the mark and sweep phases of garbage collection, does so concurrently with application execution. This concurrent execution cause a small performance impact to overall throughput.
     - **gencon,** which is new in IBM Java 5.0, is a generational garbage collector for the IBM JVM. The generational scheme attempts to achieve high throughput along with reduced garbage collection pause times. To accomplish this goal, the heap is split into new and old segments. Long lived objects are promoted to the old space while short-lived objects are garbage collected quickly in the new space. The gencon policy provides significant benefits for many applications, but is not suited to all applications and is generally more difficult to tune.

- **subpool**, which can increase performance on multiprocessor systems, that commonly use more than 8 processors, is a policy that is only available on IBM System p™ and System z™ processors. The subpool policy is similar to the optthruput policy except that the heap is divided into subpools that provide improved scalability for object allocation.

The recommended value of **gcpolicy** is *optthruput*. The usage is `Xgcpolicy:optthruput`. Setting **gcpolicy** to *optthruput* disables concurrent mark. You should get the best throughput results when you use the optthruput policy unless you are experiencing erratic application response times, which is an indication that you might have pause time problems.

Setting **gcpolicy** to *optavgpause* enables concurrent mark with its default values. This setting alleviates erratic application response times that normal garbage collection causes. However, this option might decrease overall throughput.

– -**Xnoclassgc**

By default, the JVM unloads a class from memory whenever there are no live instances of that class left. Class unloading can degrade performance. The recommended value of this property is *disable*. Turning off class garbage collection eliminates the overhead of loading and unloading the same class multiple times.

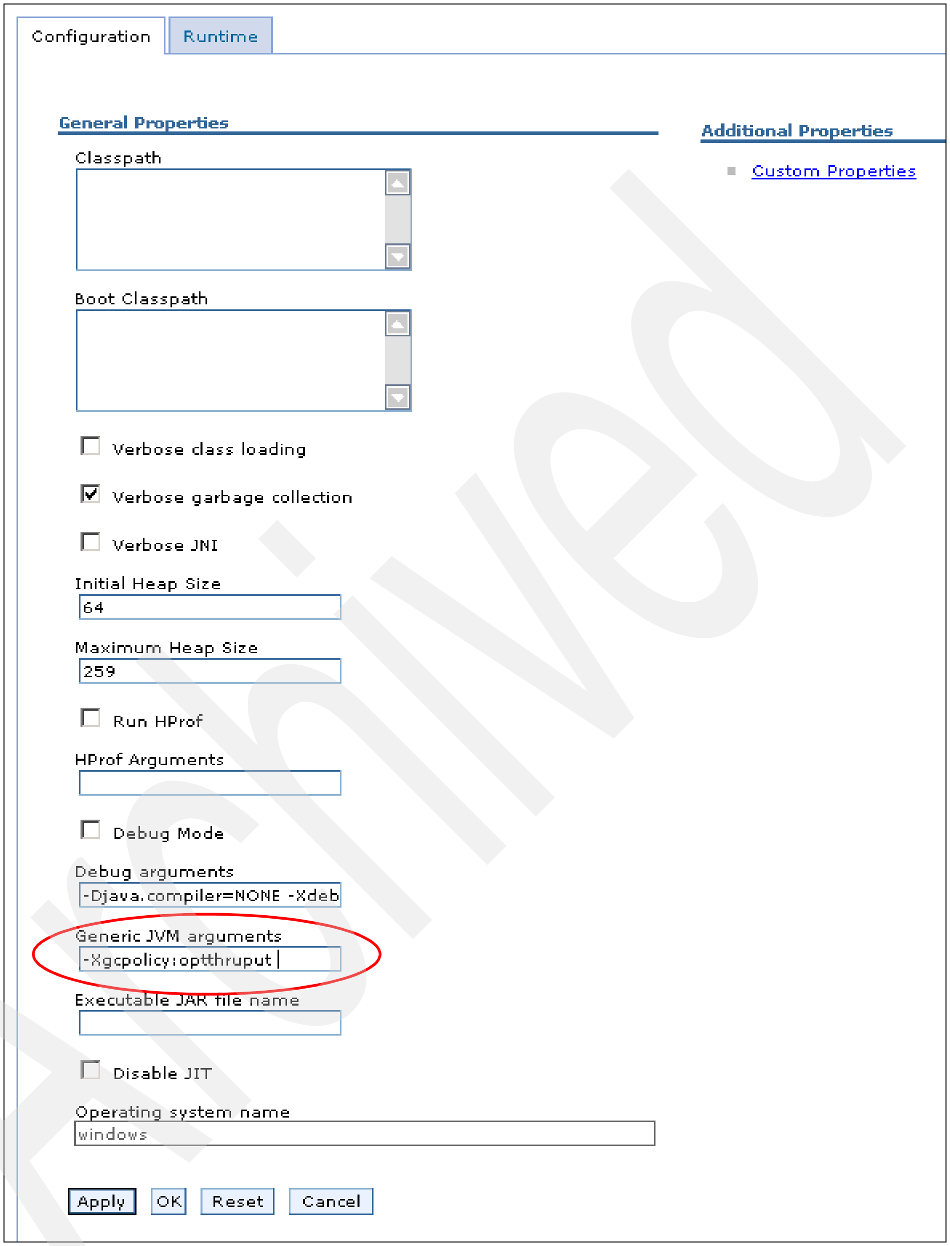


Figure 7-3 Tuning JVM garbage collector

2. Share classes in a cache.

   The share classes option of the IBM Java 2 Runtime Environment (J2RE) Version 1.5.0 lets you share classes in a cache. Sharing classes in a cache can improve startup time and reduce memory footprint. Processes, such as application servers, node agents, and deployment managers, can use the share classes option. By default, the *share classes in a cache* option is enabled and the recommended value is enabled.

   > **Important:** The IBM J2RE 1.5.0 is currently not used on:
   >
   > - Solaris
   > - HP-UX

   If you use this option, you should clear the cache when the process is not in use. To clear the cache, either call the **`<app_server_root>/bin/clearClassCache.bat/sh`** utility or stop the process and then restart the process.

   If you have to disable the share classes option for a process, specify the generic JVM argument **`-Xshareclasses:none`** for that process:

   a. In the administrative console, click **Servers** → **Application Servers** → ***server***.

   b. Under Server Infrastructure, click **Java and Process Management** → **Process Definition** → **Java Virtual Machine**.

   c. Enter **`-Xshareclasses:none`** in the Generic JVM arguments field.

   d. Click **OK**.

   e. **Save** your changes to the master configuration.

   f. Stop and restart the application server.

A complete guide to the IBM Java garbage collector is provided in the IBM Developer Kit and Runtime Environment, Java2 Technology Edition, Version 5.0 Diagnostics Guide. This document is available on the developerWorks Web site:

http://www.ibm.com/developerworks/java/jdk/diagnosis/

7.2.2 Java memory tuning tips

Enterprise applications written in the Java language involve complex object relationships and utilize large numbers of objects. Although the Java language automatically manages memory associated with object life cycles, understanding the application usage patterns for objects is important.

In particular, verify the following conditions:

- The application is not over utilizing objects.
- The application is not leaking objects.
- Java heap parameters are set properly to handle an object usage pattern.

Understanding the effect of garbage collection is necessary to apply these management techniques.

The garbage collection bottleneck

Examining Java garbage collection gives insight to how the application is utilizing memory. Garbage collection is a Java strength. By taking the burden of memory management away from the application writer, Java applications are more robust than applications written in languages that do not provide garbage collection. This robustness applies as long as the application is not abusing objects. Garbage collection normally consumes from 5% to 20% of total execution time of a properly functioning application. If not managed, garbage collection is one of the biggest bottlenecks for an application.

Monitoring garbage collection

You can use garbage collection to evaluate application performance health. By monitoring garbage collection during the execution of a fixed workload, you gain insight as to whether the application is over-utilizing objects. Garbage collection can even detect the presence of memory leaks.

You can monitor garbage collection statistics using object statistics in the Tivoli Performance Viewer, or using the **verbose:gc** JVM configuration setting. The **verbose:gc** format is not standardized between different JVMs or release levels.

For this type of investigation, set the minimum and maximum heap sizes to the same value. Choose a representative, repetitive workload that matches production usage as closely as possible, user errors included.

To ensure meaningful statistics, run the fixed workload until the application state is steady. It usually takes several minutes to reach a steady state.

Detecting memory leaks

Memory leaks in the Java language are a dangerous contributor to garbage collection bottlenecks. Memory leaks are more damaging than memory overuse, because a memory leak ultimately leads to system instability. Over time, garbage collection occurs more frequently until the heap is exhausted and the Java code fails with a fatal out-of-memory exception. Memory leaks occur when an unused object has references that are never freed. Memory leaks most commonly occur in collection classes, such as Hashtable, because the table always has a reference to the object, even after real references are deleted.

High workload often causes applications to crash immediately after deployment in the production environment. This is especially true for leaking applications where the high workload accelerates the magnification of the leakage and a memory allocation failure occurs.

Memory leak testing

The goal of memory leak testing is to magnify numbers. Memory leaks are measured in terms of the amount of bytes or kilobytes that cannot be garbage collected. The delicate task is to differentiate these amounts between expected sizes of useful and unusable memory. This task is achieved more easily if the numbers are magnified, resulting in larger gaps and easier identification of inconsistencies. The following list contains important conclusions about memory leaks:

- Long-running test:

  Memory leak problems can manifest only after a period of time, therefore, memory leaks are found easily during long-running tests. Short running tests can lead to false alarms. It is sometimes difficult to know when a memory leak is occurring in the Java language, especially when memory usage has seemingly increased either abruptly or monotonically in a given period of time. The reason it is hard to detect a memory leak is that these kinds of increases can be valid or might be the intention of the developer. You can learn how to differentiate the delayed use of objects from completely unused objects by running applications for a longer period of time. Long-running application testing gives you higher confidence for whether the delayed use of objects is actually occurring.

- Repetitive test:

  In many cases, memory leak problems occur by successive repetitions of the same test case. The goal of memory leak testing is to establish a big gap between unusable memory and used memory in terms of their relative sizes. By repeating the same scenario over and over again, the gap is multiplied in a very progressive way. This testing helps if the number of leaks caused by the execution of a test case is so minimal that it is hardly noticeable in one run.

  You can use repetitive tests at the system level or module level. The advantage with modular testing is better control. When a module is designed to keep the private module without creating external side effects such as memory usage, testing for memory leaks is easier. First, the memory usage before running the module is recorded. Then, a fixed set of test cases are run repeatedly. At the end of the test run, the current memory usage is recorded and checked for significant changes. Remember, garbage collection must be suggested when recording the actual memory usage by inserting System.gc() in the module where you want garbage collection to occur, or using a profiling tool, to force the event to occur.

- Concurrency test:

  Some memory leak problems can occur only when there are several threads running in the application. Unfortunately, synchronization points are very susceptible to memory leaks because of the added complication in the program logic. Careless programming can lead to kept or unreleased references. The incident of memory leaks is often facilitated or accelerated by increased concurrency in the system. The most common way to increase concurrency is to increase the number of clients in the test driver.

  Consider the following points when choosing which test cases to use for memory leak testing:

  – A good test case exercises areas of the application where objects are created. Most of the time, knowledge of the application is required. A description of the scenario can suggest creation of data spaces, such as adding a new record, creating an HTTP session, performing a transaction and searching a record.
  – Look at areas where collections of objects are used. Typically, memory leaks are composed of objects within the same class. Also, collection classes such as Vector and Hashtable are common places where references to objects are implicitly stored by calling corresponding insertion methods. For example, the get method of a Hashtable object does not remove its reference to the retrieved object.

Using Tivoli Performance Viewer to help find memory leaks

Tivoli Performance Viewer helps to find memory leaks. For the best results, repeat experiments with increasing duration, like 1000, 2000, and 4000 page requests. The Tivoli Performance Viewer graph of used memory should have a sawtooth shape. Each drop on the graph corresponds to a garbage collection. There is a memory leak if one of the following events occurs:

- The amount of memory used immediately after each garbage collection increases significantly. The sawtooth pattern looks more like a staircase.
- The sawtooth pattern has an irregular shape.

Also, look at the difference between the number of objects allocated and the number of objects freed. If the gap between the two increases over time, there is a memory leak.

Heap consumption indicating a possible leak during a heavy workload (the application server is consistently near 100% CPU utilization), yet appearing to recover during a subsequent lighter or near-idle workload, is an indication of heap fragmentation. Heap fragmentation can occur when the JVM can free sufficient objects to satisfy memory allocation requests during garbage collection cycles, but the JVM does not have the time to compact small free memory areas in the heap to larger contiguous spaces.

Another form of heap fragmentation occurs when small objects (less than 512 bytes) are freed. The objects are freed, but the storage is not recovered, resulting in memory fragmentation until a heap compaction has been run.

Heap fragmentation can be reduced by forcing compactions to occur, but there is a performance penalty for doing this. Use the Java `-X` command to see the list of memory options.

Java heap parameters

The Java heap parameters also influence the behavior of garbage collection. Increasing the heap size supports more object creation. Because a large heap takes longer to fill, the application runs longer before a garbage collection occurs. However, a larger heap also takes longer to compact and causes garbage collection to take longer.

For performance analysis, the initial and maximum heap sizes should be equal, as this eliminates heap growing and shrinking delays. Equating initial with maximum heapsize without previous heap size tuning, in most cases, creates an inefficiently used heap: when it is sized too big, the heap is not used by the application entirely and thus memory resources are wasted.

> **Important:** We do *not* recommend that you set the initial and maximum heap sizes equal in a production environment. For details, refer to the following technote:
>
> `http://www.ibm.com/support/docview.wss?rs=180&uid=swg21160795`

When tuning a production system where the working set size of the Java application is not understood, a good starting value for the initial heap size is 25% of the maximum heap size. The JVM then tries to adapt the size of the heap to the working set size of the application.

Figure 7-4 represents three CPU profiles, each running a fixed workload with varying Java heap settings. In the middle profile, the initial and maximum heap sizes are set to 128MB. Four garbage collections occur. The total time in garbage collection is about 15% of the total run. When the heap parameters are doubled to 256MB, as in the top profile, the length of the work time increases between garbage collections. Only three garbage collections occur, but the length of each garbage collection is also increased. In the third profile, the heap size is reduced to 64MB and exhibits the opposite effect.

With a smaller heap size, both the time between garbage collections and the time for each garbage collection are shorter. For all three configurations, the total time in garbage collection is approximately 15%. This example illustrates an important concept about the Java heap and its relationship to object utilization. There is always a cost for garbage collection in Java applications.

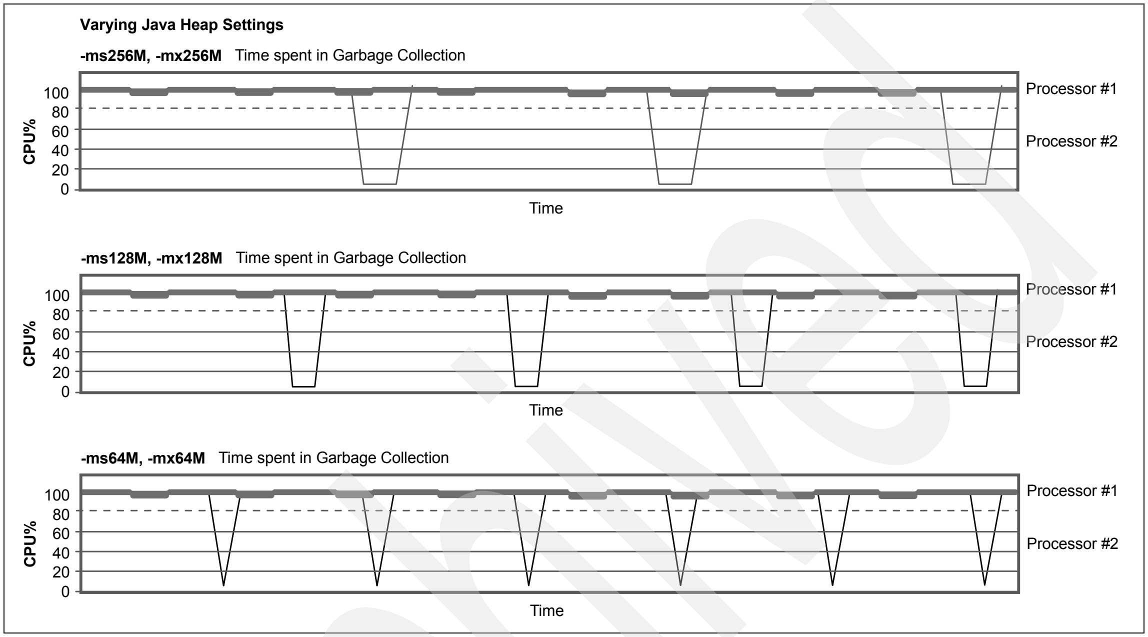


Figure 7-4 Java heap settings

Run a series of test experiments that vary the Java heap settings. For example, run experiments with 128MB, 192MB, 256MB, and 320MB. During each experiment, monitor the total memory usage. If you expand the heap too aggressively, paging can occur. Use the **vmstat** command or the Windows 2000/2003 Performance Monitor to check for paging. If paging occurs, reduce the size of the heap or add more memory to the system.

> **Important:** Make sure that the heap never pages, as that would introduce a enormous performance loss.

When all the runs are finished, compare the following statistics:

- Number of garbage collection calls
- Average duration of a single garbage collection call
- Average time between calls
- Ratio between the average length of a single garbage collection call and the average time between calls

If the application is not over utilizing objects and has no memory leaks, the state of steady memory utilization is reached. Garbage collection also occurs less frequently and for shorter duration.

If the heap free space settles at 85% or more, consider decreasing the maximum heap size values because the application server and the application are under-utilizing the memory allocated for heap.

Configuration update performance in a large cell configuration

In a large cell configuration, you might have to determine which of the two is more important: Configuration update performance or configuration consistency checking. When configuration consistency checking is of concern, a large amount of time might be required to save a configuration change or to deploy a large number of applications.

The following factors influence how much time is required:

- The more application servers or clusters there are defined in cell, the longer it takes to save a configuration change.
- The more applications there are deployed in a cell, the longer it takes to save a configuration change.

If the amount of time required to update a configuration change is unsatisfactory, you can add the `config_consistency_check` custom property to your JVM settings and set the value of this property to false, as follows:

1. In the administrative console, click **System administration** → **Deployment manager**.
2. Under Server Infrastructure, select **Java and Process Management**, and then click **Process Definition**.
3. Under Additional Properties, click **Java Virtual Machine** → **Custom Properties** → **New**.
4. Enter **config_consistency_check** in the Name field and **false** in the Value field. Figure 7-5
5. Click **OK**.
6. **Save** the configuration and restart the deployment manager for the change take effect.

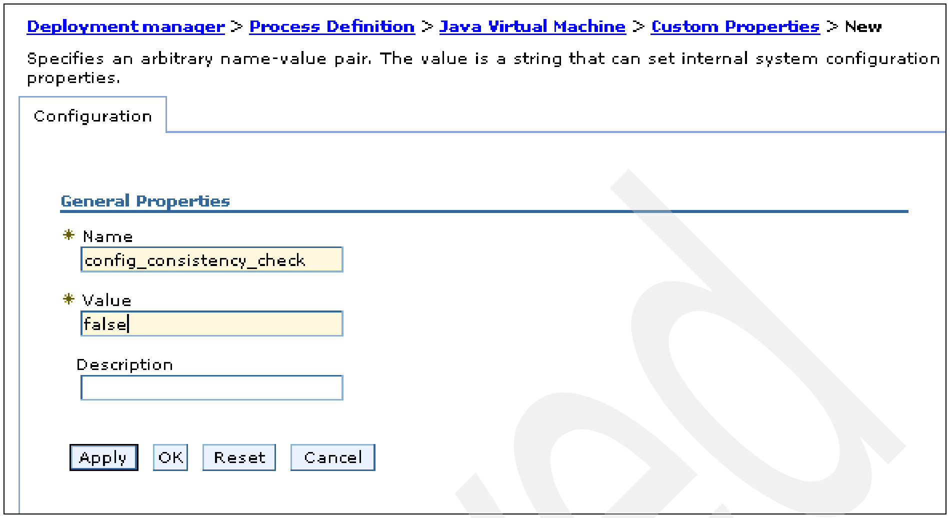


Figure 7-5 Add custom property to your JVM settings for large cell configuration

Additional JVM and garbage collection related resources

- *Java technology, IBM style: Garbage collection policies, Part 1 and 2* from developerWorks:

  http://www.ibm.com/developerworks/java/library/j-ibmjava2/index.html
  http://www.ibm.com/developerworks/java/library/j-ibmjava3/index.html

- *Real-time Java, Part 4: Real-time garbage collection* from developerWorks:

  http://www.ibm.com/developerworks/java/library/j-rtj4/

- *A brief history of garbage collection*, from developerWorks:

  http://www.ibm.com/developerworks/java/library/j-jtp10283/

- *Sensible Sanitation: Understanding the IBM Java Garbage Collection, Parts 1, 2, and 3,* from developerWorks:

  http://www.ibm.com/developerworks/ibm/library/i-garbage1/
  http://www.ibm.com/developerworks/ibm/library/i-garbage2/
  http://www.ibm.com/developerworks/ibm/library/i-garbage3.html

- *Java theory and practice: Garbage collection and performance* from DeveloperWorks

  http://www.ibm.com/developerworks/java/library/j-jtp01274.html

- *Fine-tuning Java garbage collection performance* from developerWorks

  http://www.ibm.com/developerworks/ibm/library/i-gctroub/

- *IBM JVM Diagnostics Guides 5.0* from developerWorks

  http://www.ibm.com/developerworks/java/jdk/diagnosis/

7.2.3 Tuning Web container

The WebSphere Application Server Web container manages all HTTP requests to servlets, JavaServer Pages, and Web Services. Requests flow through a transport chain to the Web container. The transport chain defines the important tuning parameters for performance for the Web container. There is a transport chain for each TCP port that WebSphere Application Server is listening on for HTTP requests. For example, the default HTTP port 9080 is defined in Web container inbound channel chain. Use the following parameters to tune the Web container:

- HTTP requests are processed by a pool of server threads. The minimum and maximum thread pool size for the Web container can be configured for optimal performance. Generally, 5 to 10 threads per server CPU provides the best throughput. The number of threads configured does not represent the number of requests WebSphere can process concurrently. Requests are queued in the transport chain when all threads are busy. To specify the thread pool settings:

  a. Click **Servers** → **Application Servers** → ***server***.

  b. In the Container Settings section expand **Web Container Settings** and click **Web container transport chains**.

  c. Select the normal inbound chain for serving requests. This is usually named **WCInboundDefault,** on port 9080.

  d. Click **TCP Inbound Channel (TCP_2)**.

  e. Under Related Items, click **Thread Pools**.

  f. Select **WebContainer**. See Figure 7-6.

  g. Set the value of **Maximum size** to configure the maximum pool size. The default value is 50. The recommended value depends on your configuration of your other infrastructure such as the Web server, EJB container, and so on.

  h. Click **OK**.

  i. **Save** the configuration and restart the affected application server for the change take effect.

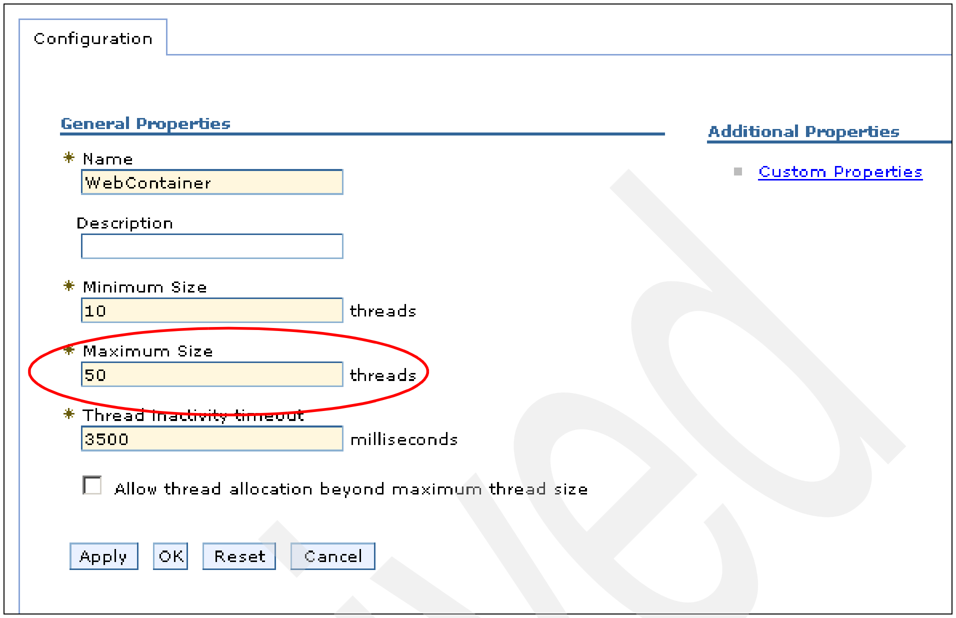


Figure 7-6 Web container thread pool setting

> **Important:** Checking the **Allow thread allocation beyond maximum thread size** box on the Thread Pool Configuration page (see Figure 7-6) allows for an automatic increase of the number of threads beyond the maximum size configured for the thread pool. As a result of this, the system can become overloaded because too many threads are allocated.

- The HTTP 1.1 protocol provides a keep-alive feature to enable the TCP connection between HTTP clients and the server to remain open between requests. By default, WebSphere Application Server closes a given client connection after a number of requests or a timeout period. After a connection is closed, it is recreated if the client issues another request. Early closure of connections can reduce performance. Enter a value for the maximum number of persistent requests to (keep-alive) to specify the number of requests that are allowed on a single HTTP connection. Enter a value for persistent timeouts to specify the amount of time, in seconds, that the HTTP transport channel allows a socket to remain idle between requests. To specify values for Maximum persistent requests and Persistent timeout:

  a. Click **Servers** → **Application Servers** → ***server.***

  b. Then, in the Container Settings section, expand **Web Container Settings** and click **Web container transport chains.**

c. Select the normal inbound chain for serving requests. This is usually named **WCInboundDefault,** on port 9080.

d. Click **HTTP Inbound Channel (HTTP_2)**.

e. Enter Value for **Persistent timeout.**

f. In the Persistent connection setting, select the check box next to **Use persistent (keep-alive) connections** (if not already there).

g. Select the radio button next to **Maximum persistent requests per connection.**

h. Enter value for **Maximum persistent requests per connection**.

i. Click **OK**.

j. **Save** the configuration and restart the affected application server for the change take effect.

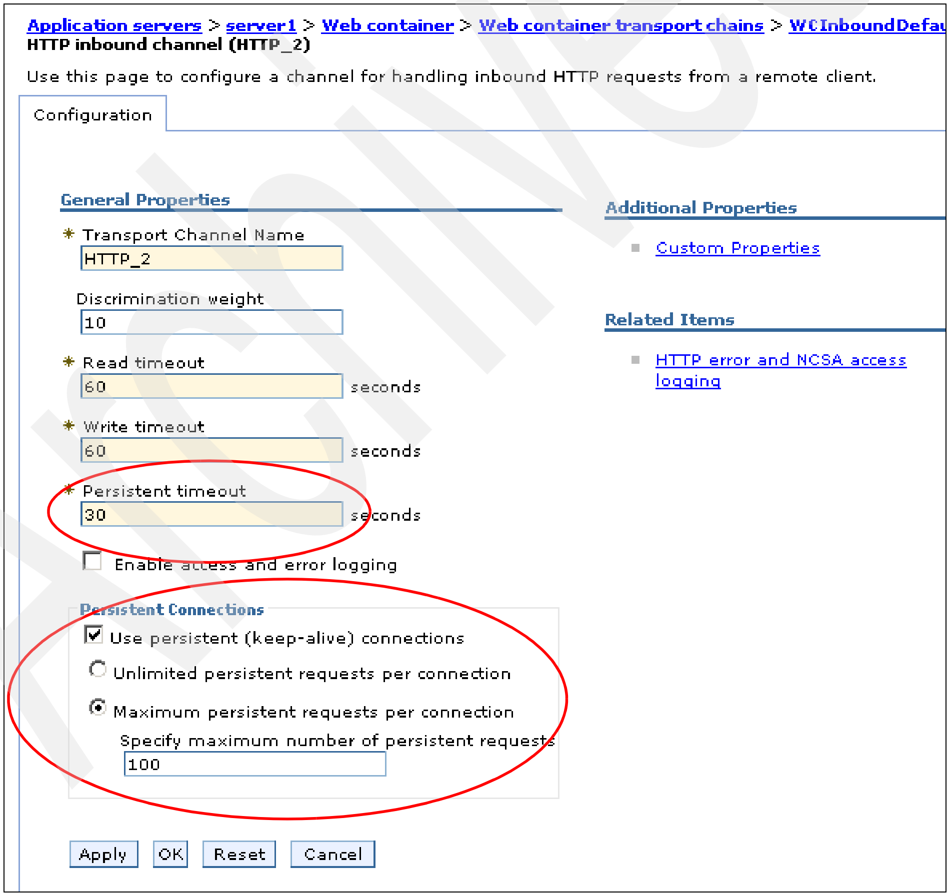


Figure 7-7 HTTP transport channel maximum persistent requests setting

7.2.4 Tuning the EJB container

An Enterprise JavaBeans (EJB) container is automatically created when you create an application server. After the EJB container is deployed, you can use the following parameters to make adjustments that improve performance.

Set the cleanup interval and the cache size

Use this setting to configure and manage the cache for a specific EJB container. To avoid errors from attempting to overload the cache, determine the cache absolute limit. Multiply the number of enterprise beans active in any given transaction by the total number of concurrent transactions expected. Then, add the number of active session bean instances. This value is the limit that the cache can hold.

To view this administrative console page:

1. Click **Servers** → **Application Servers** → ***server***.
2. Then in the Container Settings section expand **EJB Container Settings** and click **EJB cache settings.** See Figure 7-8.

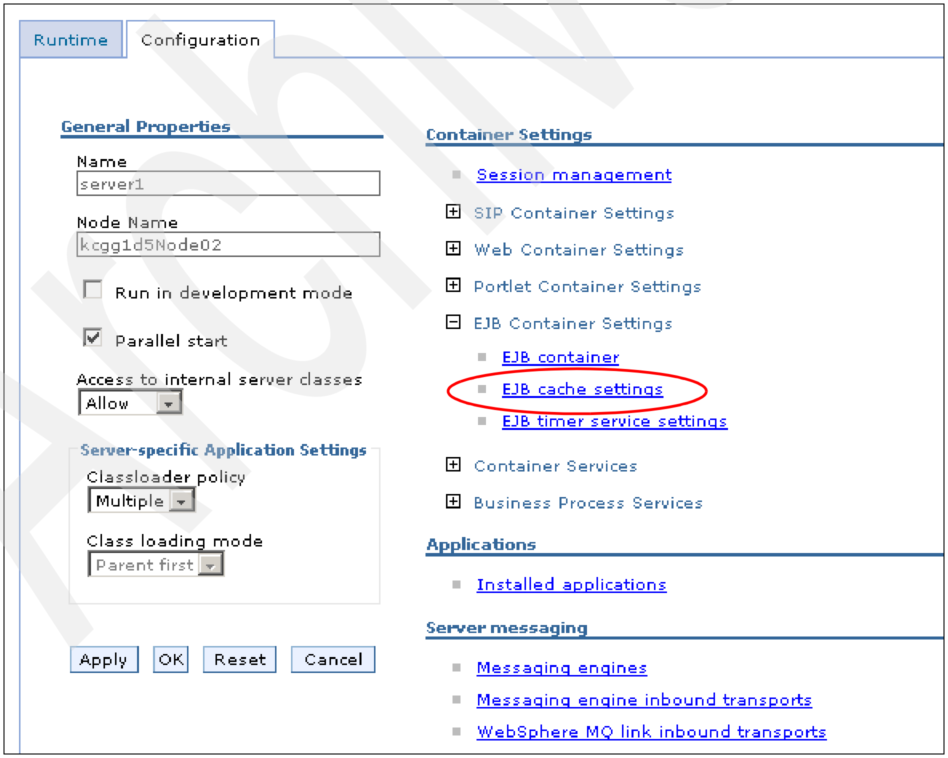


Figure 7-8 EJB cache settings

Cleanup interval

The cleanup interval property specifies the interval at which the container attempts to remove unused items from the cache in order to reduce the total number of items to the value of the cache size. This property unit is milliseconds, the range is from 0 to 2,147,483,647. The default value is 3000.

The cache manager tries to maintain some unallocated entries that can be allocated quickly as required. A background thread attempts to free some entries while maintaining some unallocated entries. If the thread runs while the application server is idle, when the application server has to allocate new cache entries, it does not pay the performance cost of removing entries from the cache. In general, increase this parameter as the cache size increases. See Figure 7-9.

Cache size

The cache size property specifies the number of buckets in the active instance list within the EJB container. This property unit is buckets in the hash table; the range is greater than 0. The container selects the next largest prime number equal to or greater than the specified value. The default value is 2053.

A bucket can contain more than one active enterprise bean instance, but performance is maximized if each bucket in the table has a minimum number of instances assigned to it. When the number of active instances within the container exceeds the number of buckets, that is, the cache size, the container periodically attempts to reduce the number of active instances in the table by passivating some of the active instances. For the best balance of performance and memory, set this value to the maximum number of active instances expected during a typical workload. See Figure 7-9.

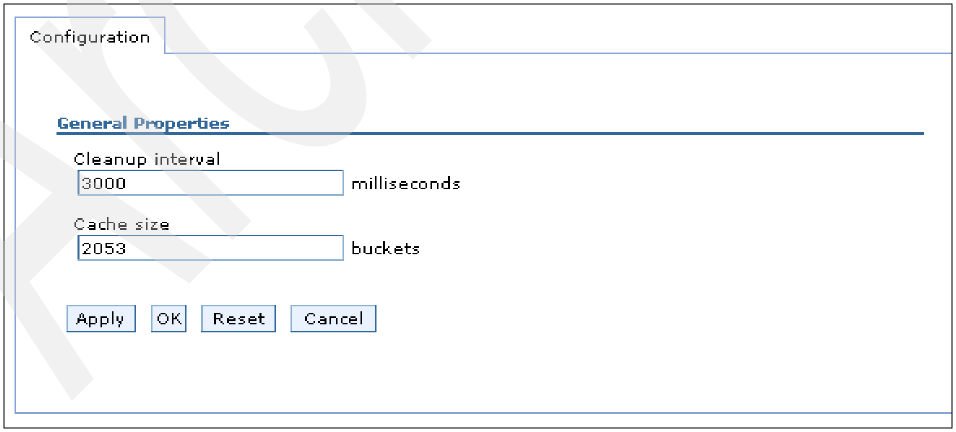


Figure 7-9 Cleanup interval and cache size

Set EJB container pool size

When the application is using the majority of the instances in the EJB Container Pool, the size of those bean pools that are being exhausted should be increased. This can be done by adding the following parameter in the JVM's custom properties tag.

```
-Dcom.ibm.websphere.ejbcontainer.poolSize=<application_name>#<module_na
me>#<bean_name>=<minSize>,<maxSize>
```

Where:

<application_name> is the J2EE application name as defined in the application archive (.ear) file deployment descriptor, for the bean whose pool size is being set.

<module_name> is the .jar file name of the EJB module, for the bean whose pool size is being set.

<bean_name> is the J2EE Enterprise Bean name as defined in the EJB module deployment descriptor, for the bean whose pool size is being set.

<minSize> is the number of bean instances the container maintains in the pool, irrespective of how long the beans have been in the pool (beans greater than this number are cleared from the pool over time to optimize memory usage).

<maxSize> is the number of bean instances in the pool where no more bean instances are placed in the pool after they are used (that is, once the pool is at this size, any additional beans are discarded rather than added into the pool — this ensures that the number of beans in the pool has an upper limit so memory usage does not grow in an unbounded fashion).

To keep the number of instances in the pool at a fixed size, *minSize* and *maxSize* can be set to the same number. Note that there is a separate instance pool for every EJB type running in the application server, and that every pool starts out with no instances in it — that is, the number of instances grows as beans are used and then placed in the pool. When a bean instance is required by the container and no beans are available in the pool, the container creates a new bean instance, uses it, then places that instance in the pool (unless there are already maxSize instances in the pool).

To add the parameter in the administrative console page:

1. Click **Servers** → **Application Servers** → ***server.***
2. In the Server Infrastructure section expand **Java and Process Management** and click **Process Definition.**

3. Under Additional Properties, click **Java Virtual Machine → Custom Properties → New**.
4. Enter **-Dcom.ibm.websphere.ejbcontainer.poolSize** in the Name field and ***<application_name>#<module_name>#<bean_name>=<minSize>,<maxSize>*** the Value field. For example, if the application that you use is Plant By WebSphere, then you can specify parameter below for the Value field: PlantsByWebSphere#PlantsByWebSphereEJB.jar#PlantsByWebSphereEJBObject =125,1327. The <*minSize*> and <*maxSize*> can be adjusted based on demand. See Figure 7-10.

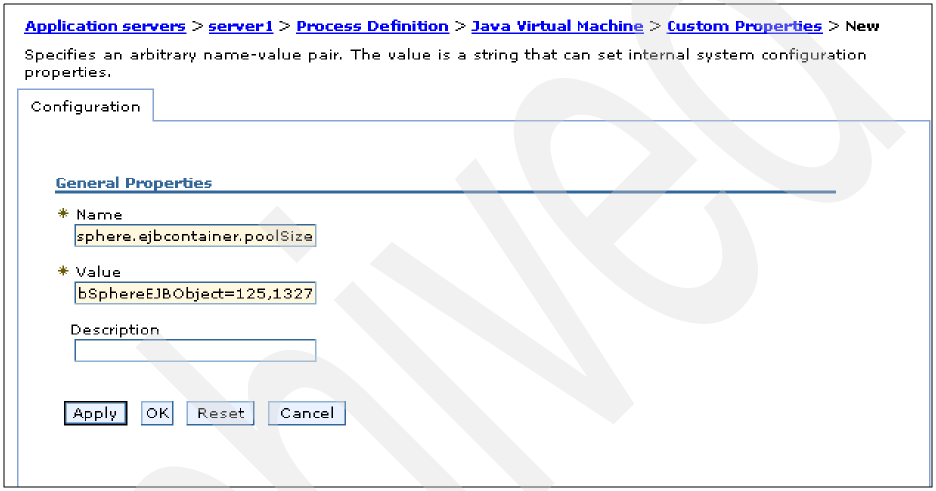


Figure 7-10 Adding custom property for setting the EJB Container Pool size

5. Click **OK**.
6. **Save** the configuration and restart the affected application server for the change take effect

Break CMP enterprise beans into several modules

To increase performance, break Container Managed Persistence (CMP) enterprise beans into several enterprise bean modules during assembly. The load time for hundreds of beans is improved by distributing the beans across several JAR files and packaging them to an EAR file. Load time is faster when the administrative server attempts to start the beans, for example, 8-10 minutes versus more than one hour when one JAR file is used.

You can use an assembly tool such as the Application Server Toolkit (AST) or Rational Application Developer to assemble an EJB module in any of the following ways:

- Import an existing EJB module (EJB JAR file).

- Create a new EJB module.
- Copy code artifacts (such as entity beans) from one EJB module into a new EJB module.

7.2.5 Tuning Object Request Broker

An Object Request Broker (ORB) manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It supports client requests and responses received from servers in a network-distributed environment. Several settings are available for controlling internal Object Request Broker (ORB) processing. You can use these to improve application performance in the case of applications containing enterprise beans.

You can change these settings for the default server or any application server configured in the administrative domain from the Administrative Console or by using a **java** command on a command line.

To use the administrative console to set ORB custom properties, in the administrative console, click **Servers** → **Application Servers** → ***server.*** Then in the Container Settings section expand **Container Services** and click **ORB Service.** See Figure 7-11.

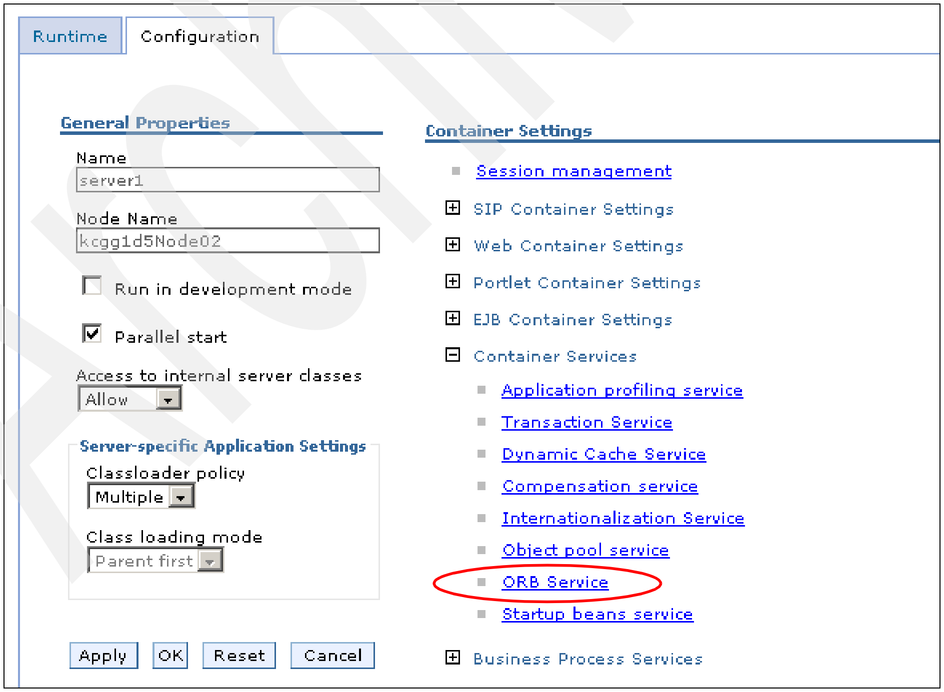


Figure 7-11 ORB Service

You can then change the setting of one of the listed custom properties or click **New** to add a new property to the list. Then click **Apply** to save your change. When you finish making changes, click **OK** and then click **Save** to save your changes.

To use the **`java`** command on a command line, use the **`-D`** option. For example:

```
java -Dcom.ibm.CORBA.propname1=value1
     -Dcom.ibm.CORBA.propname2=value2 ... application name
```

Set pass by reference

For EJB 1.1 beans, the EJB 1.1 specification states that method calls are to be Pass by value. For every remote method call, the parameters are copied onto the stack before the call is made. This can be expensive. The Pass by reference, which passes the original object reference without making a copy of the object, can be specified.

For EJB 2.0 beans, interfaces can be local or remote. For local interfaces, method calls are Pass by reference, by default.

If the EJB client and EJB server are installed in the same WebSphere Application Server instance, and the client and server use remote interfaces, specifying Pass by reference can improve performance up to 50%.

Note that Pass by reference helps performance only when non-primitive object types are being passed as parameters. Therefore, `int` and `float` are always copied, regardless of the call model.

> **Important:** Pass by reference can be dangerous and can lead to unexpected results. If an object reference is modified by the remote method, the change might be seen by the caller.

The use of this option for enterprise beans with remote interfaces violates EJB Specification, Version 2.0 section 5.4. Object references passed to EJB methods or to EJB home methods are not copied and can be subject to corruption.

In Example 7-1, a reference to the same MyPrimaryKey object passes into WebSphere Application Server with a different ID value each time. Running this code with Pass by reference enabled causes a problem within the application server because multiple enterprise beans are referencing the same MyPrimaryKey object. To avoid this problem, set the com.ibm.websphere.ejbcontainer.allowPrimaryKeyMutation system property to true when Pass by reference is enabled. Setting allowPrimaryKeyMutation to true causes the EJB container to make a local copy of the PrimaryKey object. As a result, however, a small portion of the performance advantage of setting Pass by reference is lost.

Example 7-1 Pass by reference problem demonstration

```
Iterator iterator = collection.iterator();
MyPrimaryKey pk = new MyPrimaryKey();
while (iterator.hasNext()) {
   pk.id = (String) iterator.next();
   MyEJB myEJB = myEJBHome.findByPrimaryKey(pk);
}
```

Use the Administrative Console to set allowPrimaryKeyMutation:

1. Click **Servers** → **Application Servers** → ***server***.
2. Then in the Container Settings section expand **Container Services** and click **ORB Service**.
3. Select **Custom Properties**.
4. Create a new property by clicking **New**.
5. Specify **`com.ibm.websphere.ejbcontainer.allowPrimaryKeyMutation`** in the Name field and **`true`** in the Value field. See Figure 7-12.
6. Click **OK** and **save** the changes.

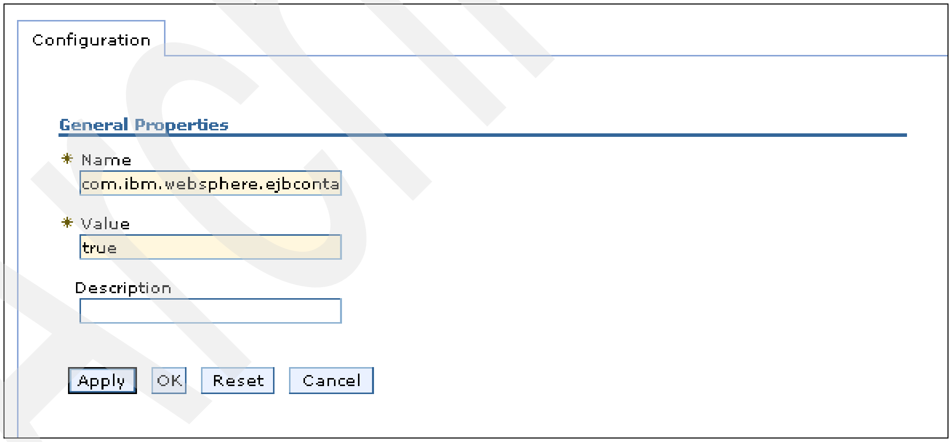


Figure 7-12 Add new custom property in ORB service setting

As a general rule, any application code that passes an object reference as a parameter to an enterprise bean method or to an EJB home method must be scrutinized to determine if passing that object reference results in loss of data integrity or in other problems.

Use the Administrative Console to set this value:

1. Click **Servers** → **Application Servers** → ***server***.
2. Then in the Container Settings section expand **Container Services** and click **ORB Service**.
3. Select the check box **Pass by reference**. See Figure 7-13.
4. Click **OK** and **Apply** to save the changes.
5. Stop and restart the application server.

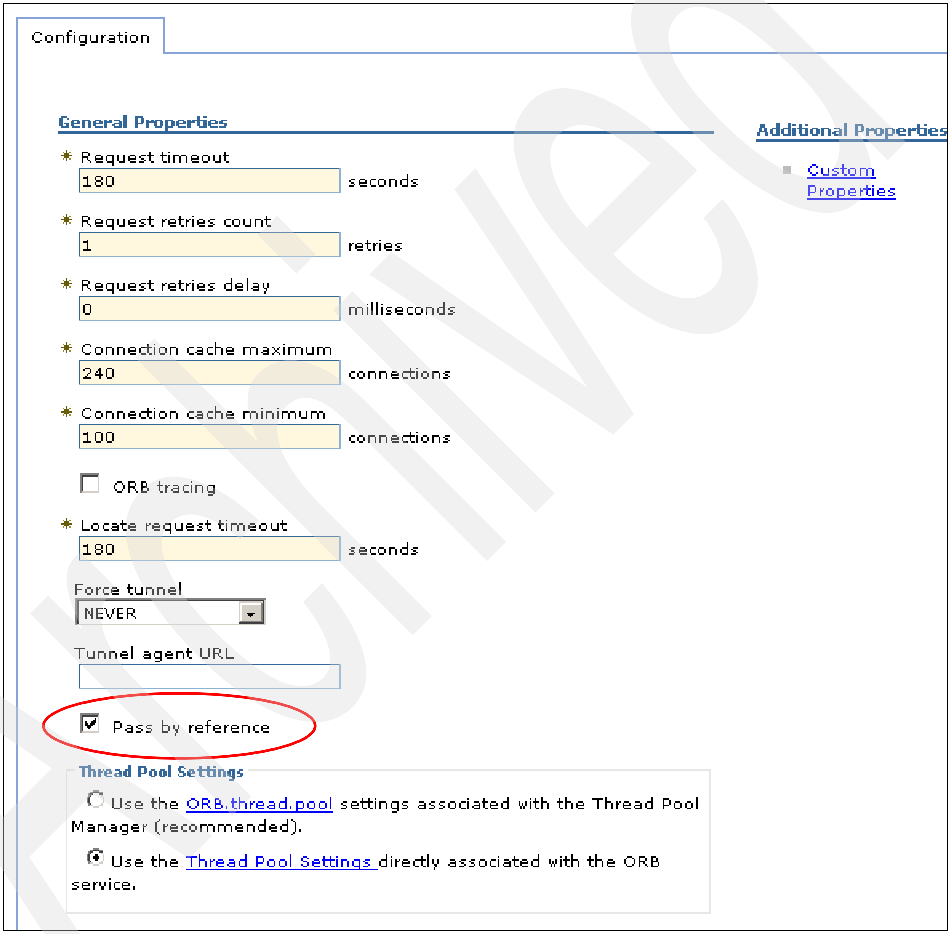


Figure 7-13 Enable pass by reference property

If you use command line scripting, the full name of this system property is `com.ibm.CORBA.iiop.noLocalCopies`.

The default is Pass by value for remote interfaces and Pass by reference for EJB 2.0 local interfaces.

If the application server expects a large workload for enterprise bean requests, the ORB configuration is critical. Take note of the following properties.

Set connection cache

Depending on an application server's workload, and throughput or response-time requirements, you might have to adjust the size of the ORB's connection cache. Each entry in the connection cache is an object that represents a distinct TCP/IP socket endpoint, identified by the hostname or TCP/IP address, and the port number used by the ORB to send a General Inter-ORB Protocol (GIOP) request or a GIOP reply to the remote target endpoint. The purpose of the connection cache is to minimize the time required to establish a connection by reusing ORB connection objects for subsequent requests or replies. (The same TCP/IP socket is used for the request and corresponding reply.)

For each application server, the number of entries in the connection cache relates directly to the number of concurrent ORB connections. These connections consist of both the inbound requests made from remote clients and outbound requests made by the application server. When the server-side ORB receives a connection request, it uses an existing connection from an entry in the cache, or establishes a new connection and adds an entry for that connection to the cache.

The ORB Connection cache maximum and Connection cache minimum properties are used to control the maximum and minimum number of entries in the connection cache at a given time. When the number of entries reaches the value specified for the Connection cache maximum property, and a new connection is required, the ORB creates the requested connection, adds an entry to the cache and searches for and attempts to remove up to five inactive connection entries from the cache. Because the new connection is added before inactive entries are removed, it is possible for the number of cache entries to temporarily exceed the value specified for the Connection cache maximum property.

An ORB connection is considered inactive if the TCP/IP socket stream is not in use and there are no GIOP replies pending for any requests made on that connection. As the application workload diminishes, the ORB closes the connections and removes the entries for these connections from the cache. The ORB continues to remove entries from the cache until the number of remaining entries is at or below the value specified for the Connection cache maximum property. The number of cache entries is never less then the value specified for the Connection cache minimum property, which must be at least five connections less than the value specified for the Connection cache maximum property.

Adjustments to the connection cache in the client-side ORB are usually not necessary because only a small number of connections are made on that side.

Connection cache maximum

This property has two names and corresponds to the size of the ORB connection table. The property sets the standard for the number of simultaneous ORB connections that can be processed.

If there are many simultaneous clients connecting to the server-side ORB, this parameter can be increased to support the heavy load up to 1000 clients. The default value is 240.

Use the Administrative Console to set this value:

1. Click **Servers** → **Application Servers** → ***server***.
2. Then in the Container Settings section expand **Container Services** and click **ORB Service**
3. Update the **Connection cache maximum** field. See Figure 7-14.
4. Click **OK** and **save** the changes.
5. Stop and restart the application server.

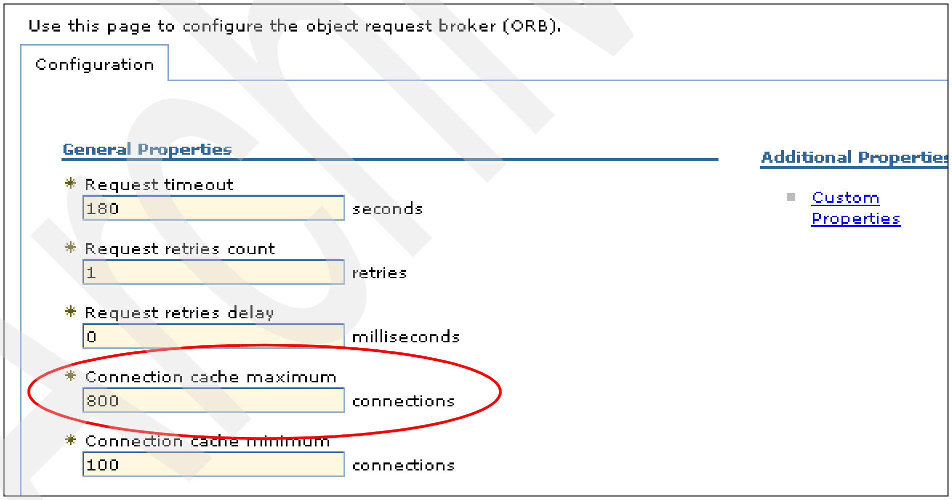


Figure 7-14 Set connection cache maximum property

If you use command line scripting, the full name of this system property is `com.ibm.CORBA.MaxOpenConnections`.

Set server socket queue depth

This property corresponds to the length of the TCP/IP stack listen queue and prevents WebSphere Application Server from rejecting requests when there is no space in the listen queue.

If there are many simultaneous clients connecting to the server-side ORB, this parameter can be increased to support the heavy load up to 1000 clients. The default value is 50.

If you see a “connection refused” message in a trace log, usually either the port on the target machine is not open, or the server is overloaded with queued-up connection requests. Increasing the value specified for this property can help alleviate this problem if there does not appear to be any other problem in the system.

To set the property (in our example we set it to 200), follow these steps:

1. Click **Servers** → **Application Servers** → ***server***.
2. Then in the Container Settings section expand **Container Services** and click **ORB Service**.
3. Click **Custom Properties** → **New**.
4. Specify `com.ibm.CORBA.ServerSocketQueueDepth` in the Name field and **200** in the Value field.
5. Click **OK** and **Apply** to save the changes. See Figure 7-15.
6. **Stop** and restart the application server.

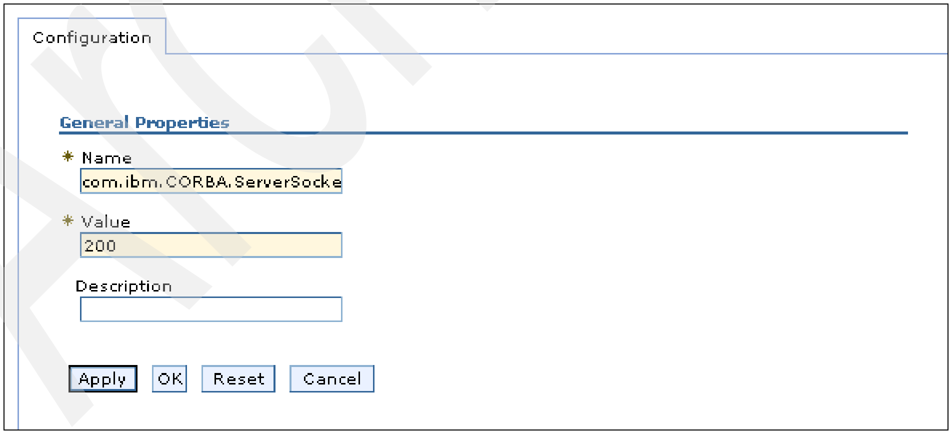


Figure 7-15 Add custom property com.ibm.CORBA.ServersocketQueueDepth

If you use command line scripting, the full name of this system property is `com.ibm.CORBA.ServerSocketQueueDepth`.

Set ORB thread pool size

Method invocations to enterprise beans are only queued for requests coming from remote clients going through the RMI activity service. An example of such a client is an EJB client running in a separate Java Virtual Machine (another address space) from the enterprise bean. In contrast, no queuing occurs if the EJB client (either a servlet or another enterprise bean) is installed in the same JVM that the EJB method runs on and the same thread of execution as the EJB client.

Remote enterprise beans communicate by using the RMI/IIOP protocol. Method invocations initiated over RMI/IIOP are processed by a server-side ORB. The thread pool acts as a queue for incoming requests. However, if a remote method request is issued and there are no more available threads in the thread pool, a new thread is created. After the method request completes, the thread is destroyed. Therefore, when the ORB is used to process remote method requests, the EJB container is an open queue, due to the use of unbounded threads.

Tivoli Performance Viewer can help tune the ORB thread pool size settings. Use a standard workload that represents a typical number of incoming client requests, use a fixed number of iterations, and use a standard set of configuration settings. Watch the `PercentMaxed` counter of the Thread Pools module. If the value of this counter is consistently in the double digits, then the ORB could be a bottleneck and the number of threads in the pool should be increased.

The degree to which the ORB thread pool value has to be increased is a function of the number of simultaneous servlets (that is, clients) calling enterprise beans and the duration of each method call. If the method calls are longer or the applications spend a lot of time in the ORB, consider making the ORB thread pool size equal to the Web container size. If the servlet makes only short-lived or quick calls to the ORB, servlets can potentially reuse the same ORB thread. In this case, the ORB thread pool can be small, perhaps even one-half of the thread pool size setting of the Web container.

The ORB thread pool size is configured from the Administrative Console using these steps:

To change these settings:

1. Click **Servers** → **Application Servers** → ***server.***
2. Then in the Container Settings section expand **Container Services** and click **ORB Service.**
3. Select **Use the thread pool settings directly associated with the ORB service**, and then click **thread pool settings**.

4. Use the **Maximum Size** field to configure the maximum pool size. Note that this only affects the number of threads held in the pool (the actual number of ORB threads can be higher). See Figure 7-16.

5. Save the configuration and restart the affected application server for the change to take effect.

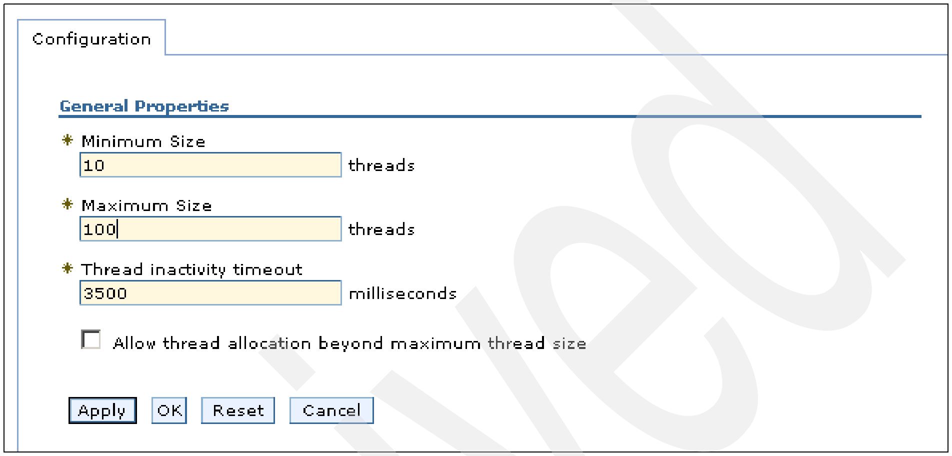


Figure 7-16 ORB thread pool settings

Set fragment size

The ORB separates messages into fragments to send over the ORB connection. You can configure this fragment size through the com.ibm.CORBA.FragmentSize parameter.

To determine and change the size of the messages that transfer over the ORB and the number of required fragments, perform the following steps:

1. In the administrative console, enable ORB tracing in the ORB Properties page.
   a. Click **Servers** → **Application Servers** → ***server.***
   b. Then in the Container Settings section expand **Container Services** and click **ORB Service.**
   c. Select the check box **ORB tracing.** See Figure 7-17.
   d. Click **OK** and **save** the configuration.

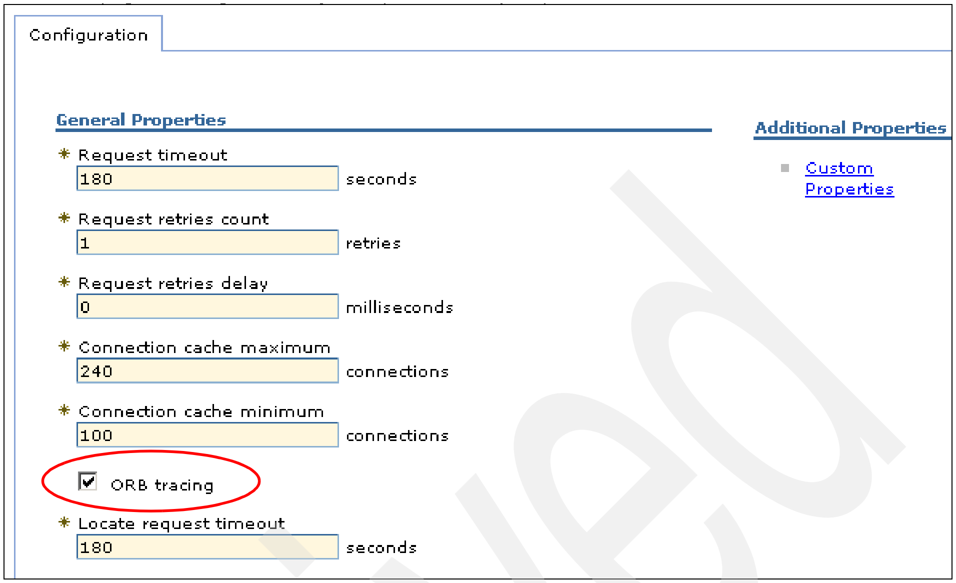


Figure 7-17 Enable ORB tracing

2. Enable ORBRas tracing from the logging and tracing page.

   a. Click **Troubleshooting** → **Logs and Trace** → ***server*** → **Change Log Detail Levels.**

   b. Click **ORBRas** → **Message and Traces Levels** → **Fine.** It is recommended that you select fine instead of finest.

   c. After that you can see `ORBRas=fine` in the text box. See Figure 7-18.

   d. Click **OK** and **save** the configuration.

3. Increase the trace file sizes because tracing can generate a lot of data.

   a. Click **Troubleshooting** → **Logs and Trace** → ***server*** → **Diagnostic Trace.**

   b. In the trace output section, you can edit the **Maximum File Size** property or **Maximum Number of Historical Files** property.

   c. Click **OK** and **save** the configuration. See Figure 7-19.

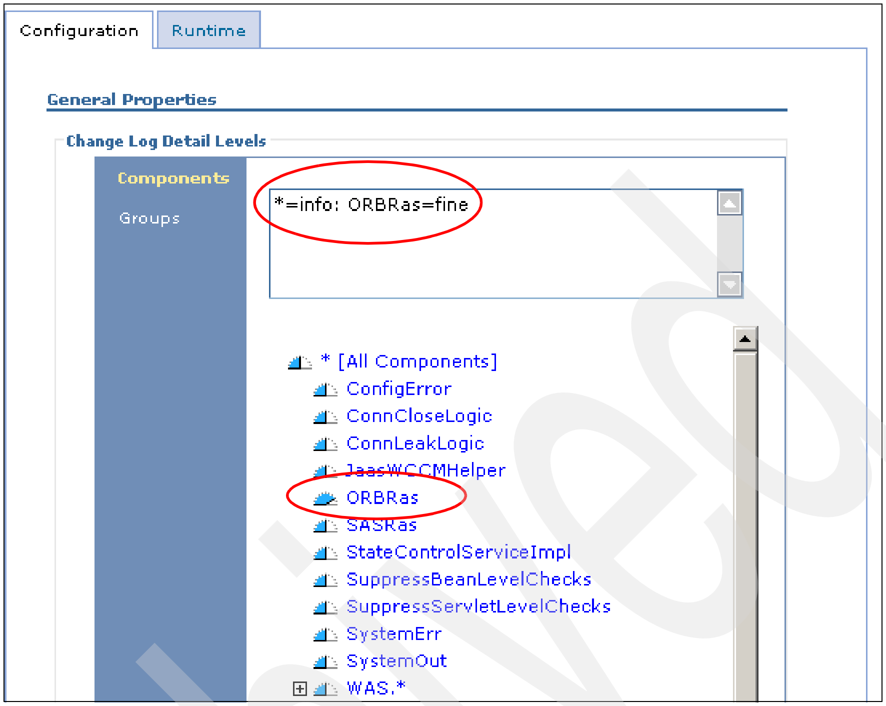


Figure 7-18 Enable ORBRas tracing

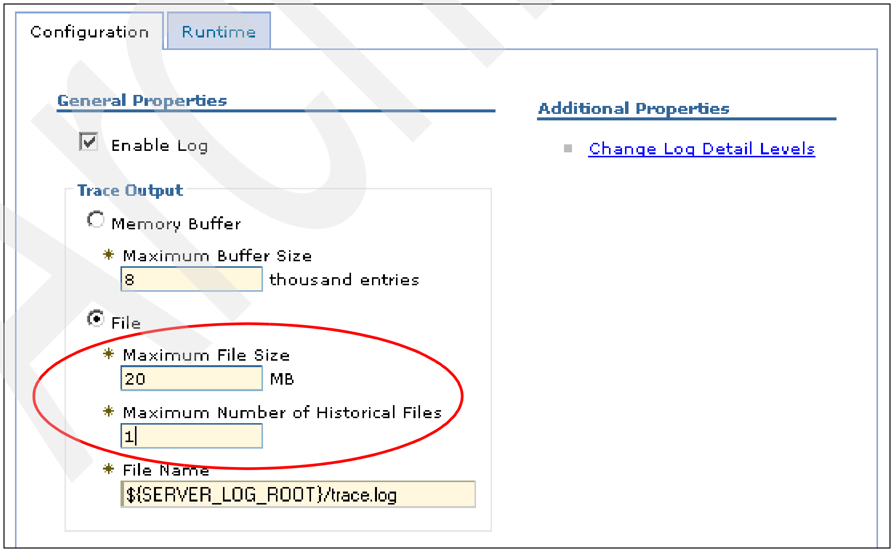


Figure 7-19 Increase the trace file sizes

4. Restart the affected application server and run at least one iteration (preferably several) of the case that you are measuring.
5. Look at the traceable file and do a search for `Fragment to follow: Yes`.

   This message indicates that the ORB transmitted a fragment, but it still has at least one remaining fragment to send before the entire message arrives. A `Fragment to follow: No` value indicates that the particular fragment is the last in the entire message. This fragment can also be the first, if the message fit entirely into one fragment.

   If you go to the spot where `Fragment to follow: Yes` is located, you find a block that looks similar to the following example:

   ```
   Fragment to follow: Yes
   Message size: 4988 (0x137C)
   --
   Request ID: 1411
   ```

   This example indicates that the amount of data in the fragment is 4988 bytes and the Request ID is 1411. If you search for all occurrences of Request ID: 1411, you can see the number of fragments that are used to send that particular message. If you add all the associated message sizes, you have the total size of the message that is being sent through the ORB.

   If you use command line scripting, the full name of this system property is `com.ibm.CORBA.FragmentSize` property.

Remove interceptors

Interceptors are ORB extensions that can set up the context before the ORB runs a request. For example, the context might include transactions or activity sessions to import. If the client creates a transaction, and then flows the transaction context to the server, then the server imports the transaction context onto the server request through the interceptors.

Most clients do not start transactions or activity sessions, so most systems can benefit from removing the interceptors that are not required.

To remove the interceptors, manually edit the **server.xml** file in folder **<AppServer_Root>\profiles\<profile>\config\cells\<cell>\nodes\<node>\servers\<server>** and remove the interceptor lines that are not required from the ORB section. For example, remove line below:

```
<interceptors xmi:id="xxxx" name="com.ibm.xxxxx.xxxxxx"/>
```

Java Native Interface (JNI) reader threads

By default, the ORB uses a Java thread for processing each inbound connection request it receives. As the number of concurrent requests increases, the storage consumed by a large number of reader threads increases and can become a bottleneck in resource-constrained environments. Eventually, the number of Java threads created can cause out-of-memory exceptions if the number of concurrent requests exceeds the system's available resources.

To help address this potential problem, you can configure the ORB to use JNI reader threads where a finite number of reader threads, implemented using native OS threads instead of Java threads, are created during ORB initialization. JNI reader threads rely on the native OS TCP/IP asynchronous mechanism that enables a single native OS thread to handle I/O events from multiple sockets at the same time. The ORB manages the use of the JNI reader threads and assigns one of the available threads to handle the connection request, using a round-robin algorithm. Ordinarily, JNI reader threads should only be configured when using Java threads is too memory-intensive for your application environment.

The number of JNI reader threads you should allocate for an ORB depends on many factors and varies significantly from one environment to another, depending on available system resources and workload requirements. The following potential benefits might be achieved if you use JNI threads:

- Because a fixed number of threads is allocated, memory usage is reduced. This reduction provides significant benefit in environments with unusually large and sustained client-request workloads.
- The time required to dynamically create and destroy Java threads is eliminated because a fixed number of JNI threads is created and allocated during ORB initialization.
- Each JNI thread can handle up to 1024 socket connections and interacts directly with the asynchronous I/O native OS mechanism, which might provide enhanced performance of network I/O processing.

> **Important:** Because JSSE2 does not provide the file descriptor that JNIReader Threads require, you cannot use JNIReader Threads with the default IBMJSSE2 SSL security provider setting. If you attempt to use both of these settings, the server does not start and logs a ClassCast exception on the com.ibm.jsse2.c class.

7.2.6 Tuning XML parser selection

Add XML parser definitions to the **jaxp.properties** file and **xerces.properties** file found in the ***<AppServer_Root>*** **\java\jre\lib** directory to help facilitate server startup. The XMLParserConfiguration value might have to be changed as new versions of Xerces are provided.

In both files, insert the lines shown in Example 7-2.

Example 7-2 XML parser definitions

```
javax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.SAXParserFactoryImpl
javax.xml.parsers.DocumentBuildFactory=org.apache.xerces.jaxp.DocumentBuilder
FactoryImpl
org.apache.xerces.xni.parser.XMLParserConfiguration=org.apache.xerces.parsers.
StandardParserConfiguration
```

7.2.7 Tuning the URL invocation cache

Each JavaServer Page is a unique URL. If you have more than 50 unique URLs that are actively being used, increase the value specified for the invocationCacheSize JVM custom property. This property controls the size of the URL invocation cache.The URL invocation cache holds information for mapping request URLs to servlet resources. A cache of the requested size is created for each worker thread that is available to process a request. The default size of the invocation cache is 50. If more than 50 unique URLs are actively being used (each JavaServer Page is a unique URL), you should increase the size of the invocation cache.

A larger cache uses more of the Java heap, so you might also have to increase the maximum Java heap size. For example, if each cache entry requires 2KB, maximum thread size is set to 25, and the URL invocation cache size is 100; then 5MB of Java heap are required.

The invocation cache is now Web container based instead of thread-based, and shared for all Web container threads.

You can follow this step to change the setting.

1. In the administrative console, click **Servers** → **Application servers** → ***server***.
2. Under Server Infrastructure section, expand **Java and Process Management** and click **Process Definition**
3. Under Additional Properties, **click Java Virtual Machine** → **Custom Properties** → **New**.
4. Specify `invocationCacheSize` in the Name field and the size of the cache in the Value field. The default size for the invocation cache is 500 entries. Since the invocation cache is no longer thread-based, the invocation cache size specified by the user is multiplied by ten to provide similar function from previous releases. For example, if you specify an invocation cache size of 50, the Web container creates a cache size of 500. See Figure 7-20.

5. Click **Apply** and then **Save** to save your changes.
6. **Stop** and restart the affected application server.

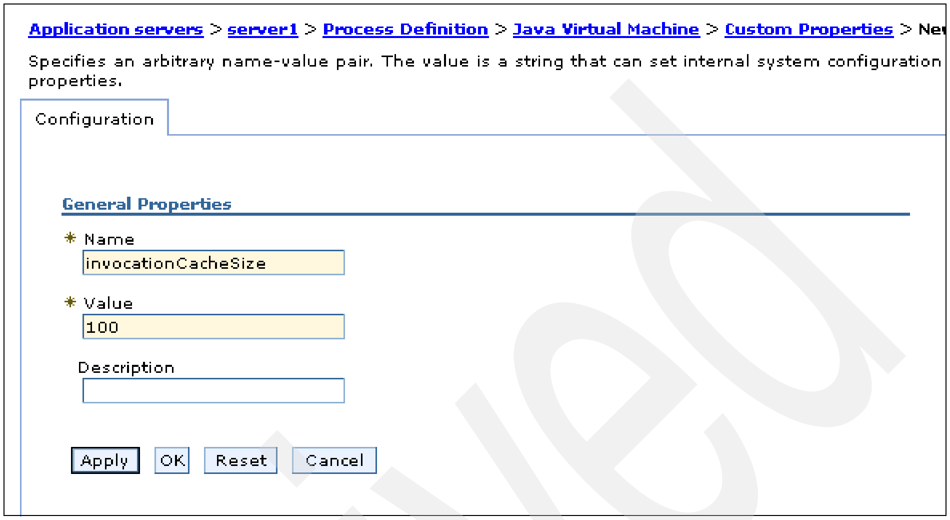


Figure 7-20 Tuning URL Invocation cache size

7.2.8 Tuning transport channel services

The transport channel services manage client connections and I/O processing for HTTP and JMS requests. These I/O services are based on the non-blocking I/O (NIO) features that are available in Java. These services provide a highly scalable foundation to WebSphere Application Server request processing. Java NIO based architecture has limitations in terms of performance, scalability and end user usability. Therefore, integration of true asynchronous I/O is implemented. This implementation provides significant benefits in usability, reduces the complexity of I/O processing and reduces that amount of performance tuning you have to perform.

Key features of the new transport channel services include:

- Scalability, which enables the WebSphere Application Server to handle many concurrent requests
- Asynchronous request processing, which provides a many-to-one mapping of client requests to Web container threads
- Resource sharing and segregation, which enables thread pools to be shared between the Web container and a messaging service
- Improved usability
- Incorporation of autonomic tuning and configuration functions

Changing the default values for settings on one or more of the transport channels associated with a transport chain can improve the performance of that chain.

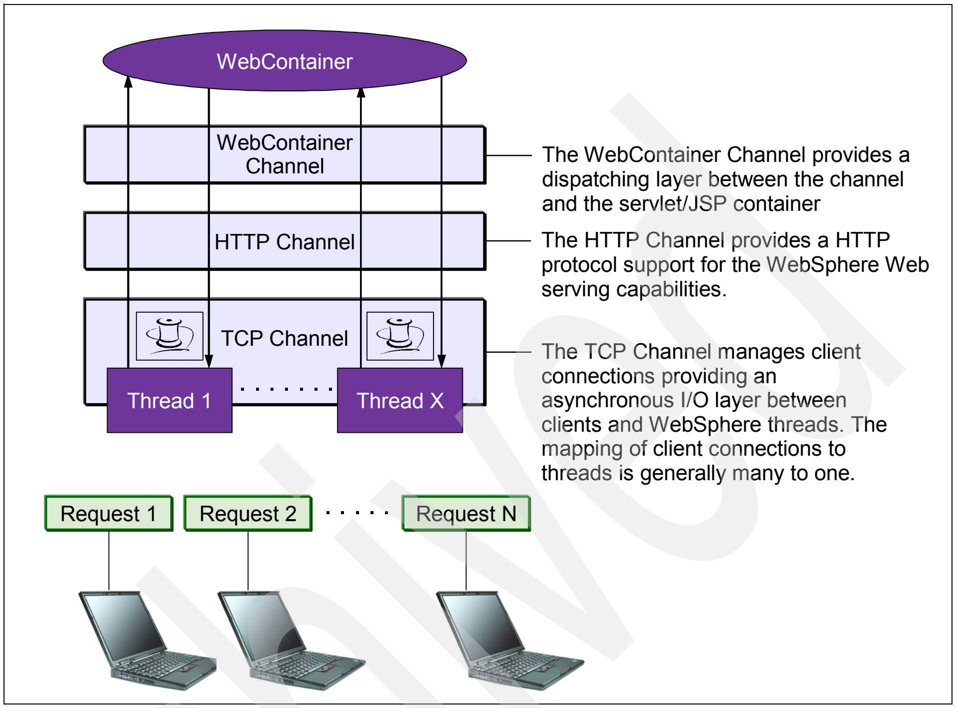


Figure 7-21 Transport Channel Service

Adjust TCP transport channel settings

In the administration console, click **Servers** → **Application servers** → ***server.*** Then in the communication section click **Ports**. Then click **View associated transports** for the appropriate port.

1. Select the transport chain whose properties you are changing.
2. Click on the TCP transport channel defined for that chain.
3. Leave the Maximum open connections parameter set to the default value. This parameter controls the maximum number of connections that are available for a server's use. It should be left at the default value of 20000, which is the maximum number of connections allowed. The transport channel service by default manages high client connection counts and requires no tuning. See Figure 7-22.
4. If client connections are being closed without data being written back to the client, change the value specified for the Inactivity timeout parameter. This parameter controls the maximum number of connections available for a

server's use. Upon receiving a new connection, the TCP transport channel waits for enough data to arrive to dispatch the connection to the protocol specific channels above the TCP transport channel. If not enough data is received during the time period specified for the Inactivity timeout parameter, the TCP transport channel closes the connection.

The default value for this parameter is 60 seconds, which is adequate for most applications. You should increase the value specified for this parameter if your workload involves a lot of connections and all of these connections can not be serviced in 60 seconds. See Figure 7-22.

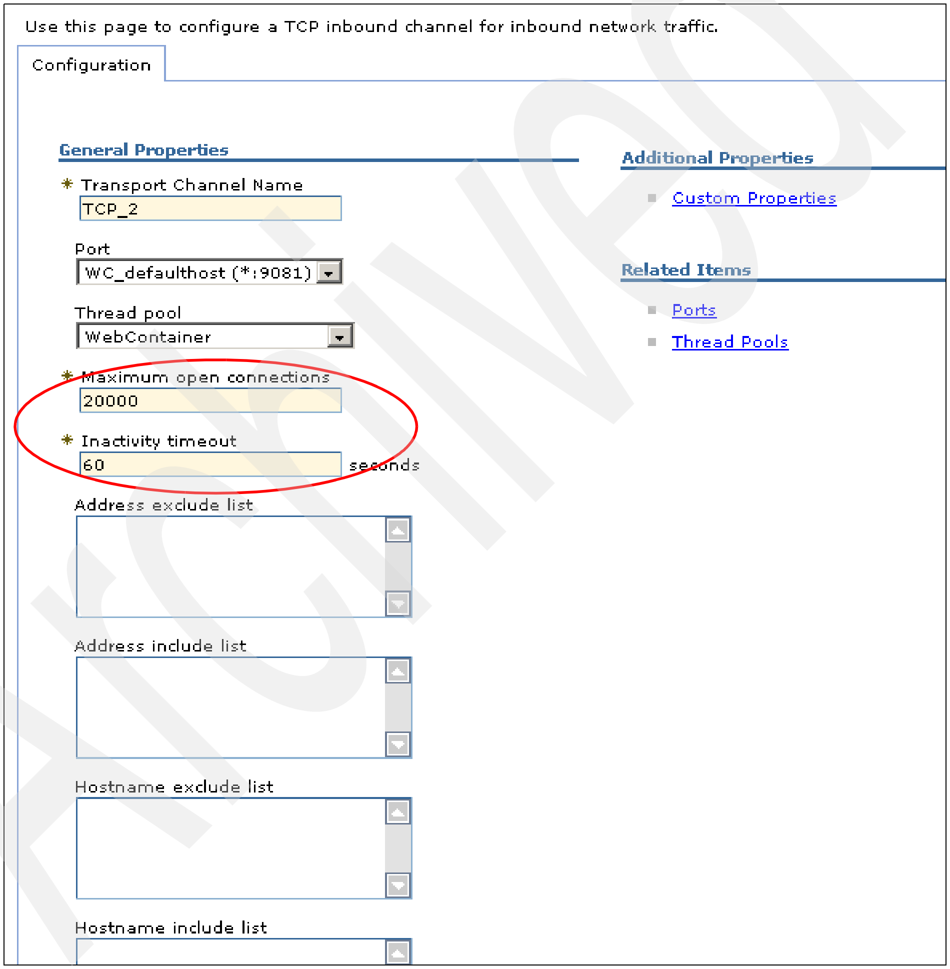


Figure 7-22 TCP transport channel setting

5. Assign a thread pool to a specific HTTP port. Each TCP transport channel is assigned to a particular thread pool. Thread pools can be shared between one or more TCP transport channels as well as with other components. The default settings for a TCP transport channel is to have all HTTP based traffic assigned to the WebContainer thread pool and all other traffic assigned to the Default thread pool. Use the Thread pool pull-down to assign a particular thread pool to each TCP transport channel. The default settings for this parameter has all HTTP based traffic assigned to the WebContainer thread pool and all other traffic is assigned to the Default thread pool. (Thread pool collection describes how to create additional thread pools.)
6. Tune the size of your thread pools. By default, a thread pool can have a minimum of 10 threads and a maximum of 50 maximum threads. To adjust these values, click on **Thread pools** → ***threadpool_name*** and adjust the values specified for the Minimum Size and Maximum Size parameters for that thread pool. See Figure 7-23.

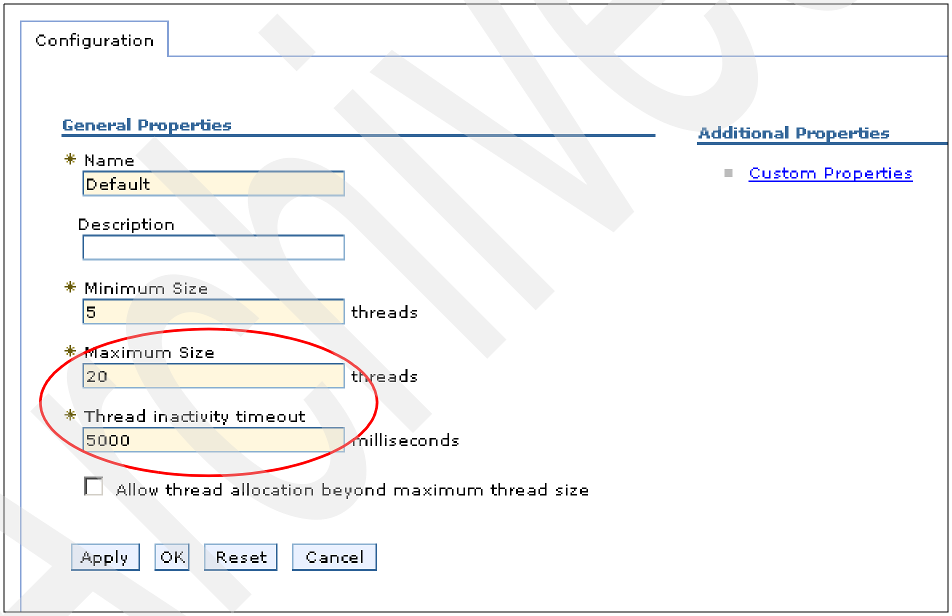


Figure 7-23 Thread pool setting

Typical applications usually do not require more than 10 threads per processor. One exception is if there is some off server condition, such as a very slow backend request, that causes a server thread to wait for the backend request to complete. In such a case, CPU usage is usually low and increasing the workload does not increase CPU throughput. Thread dumps show nearly all threads in a call out to the backend resource. If this condition exists, and the backend is tuned correctly, try increasing the minimum number of threads in the pool until you see improvements in throughput and thread dumps show threads in other areas of the runtime besides the backend call.

The setting for the Grow as needed parameter should not be changed unless your backend is prone to hanging for long periods of time. This condition might indicate that all of your runtime threads are blocked waiting for the backend instead of processing other work that does not involve the hung backend.

Adjust HTTP transport channel settings

In the administration console, click **Servers** → **Application servers** → ***server*** → **Ports**. Then click **View associated transports** for the appropriate port.

1. Select the transport chain whose properties you are changing.
2. Click on the HTTP transport channel defined for that chain.
3. Tune HTTP keep-alive. The Use persistent (keep-alive) connections setting controls whether or not connections are left open between requests. Leaving the connections open can save setup and tear down costs of sockets if your workload has clients that send multiple requests. The default value is true and is the optimal setting in most cases. See Figure 7-24.

   If your clients only send single requests over substantially long periods of time, it is probably better to disable this option and close the connections right away rather than to have the HTTP transport channel setup the timeouts to close the connection at some later time.
4. Change the value specified for the Maximum persistent requests parameter to increase the number of requests that can flow over a connection before it is closed. When the Use persistent connections option is enabled, the Maximum persistent requests parameter controls the number of requests that can flow over a connection before it is closed. The default value is 100. This value should be set to a value such that most, if not all, clients always have an open connection when they make multiple requests during the same session. A proper setting for this parameter helps to eliminate unnecessary setting up and tearing down of sockets. See Figure 7-24.

   For test scenarios in which the client never closes a socket or where sockets are always proxy or Web servers in front of your application server, a value of -1 disables the processing, which limits the number of requests over a single connection. The persistent timeout still shuts down some idle sockets and protect your server from running out of open sockets.
5. Change the value specified for the Persistent timeout parameter to increase the length of time that a connection is held open before being closed due to inactivity. The Persistent timeout parameter controls the length of time that a connection is held open before being closed because there is no activity on that connection. The default value is 30 seconds.

This parameter should be set to a value that keeps enough connections open so that most clients can obtain a connection available when they have to make a request. See Figure 7-24.

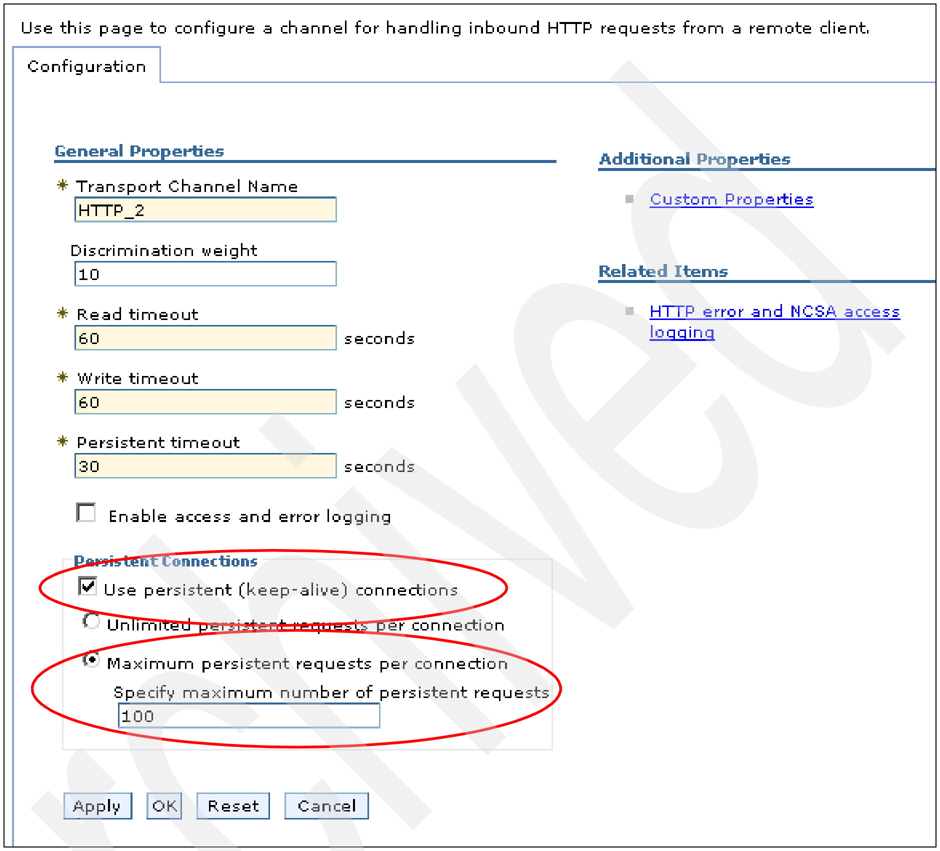


Figure 7-24 HTTP transport channel setting

6. If clients are having trouble completing a request because it takes them more than 60 seconds to send their data, change the value specified for the Read timeout parameter. Some clients pause more than 60 seconds while sending data as part of a request. To ensure they are able to complete their requests, change the value specified for this parameter to a length of time in seconds that is sufficient for the clients to complete the transfer of data. Be careful when changing this value that you still protect the server from clients who send incomplete data and thereby utilize resources (sockets) for an excessive amount of time.

7. If some of your clients require more than 60 seconds to receive data being written to them, change the value specified for the Write timeout parameter. Some clients are slow and require more than 60 seconds to receive data that is sent to them. To ensure they are able to obtain all of their data, change the value specified for this parameter to a length of time in seconds that is sufficient for all of the data to be received. Be careful when changing this value that you still protect the server from malicious clients.

Adjust Web container transport channel settings

In the administration console, click **Servers** → **Application servers** → ***server*** → **Ports**. Then click **View associated transports** for the appropriate port.

1. Select the transport chain whose properties have to be changed.
2. Click on the Web container transport channel defined for that chain.
3. If multiple writes are required to handle responses to the client, change the value specified for the Write buffer size parameter to a value that is more appropriate for your clients. The Write buffer size parameter controls the maximum amount of data per thread that the Web container buffers before sending the request on for processing. See Figure 7-25, the default value is 32768 bytes, which is sufficient for most applications. If the size of a response is greater than the size of the write buffer, the response is chunked and written back in multiple TCP writes.

   If you have to change the value specified for this parameter, make sure the new value enables most requests to be written out in a single write. To determined an appropriate value for this parameter, look at the size of the pages that are returned and add some additional bytes to account for the HTTP headers.

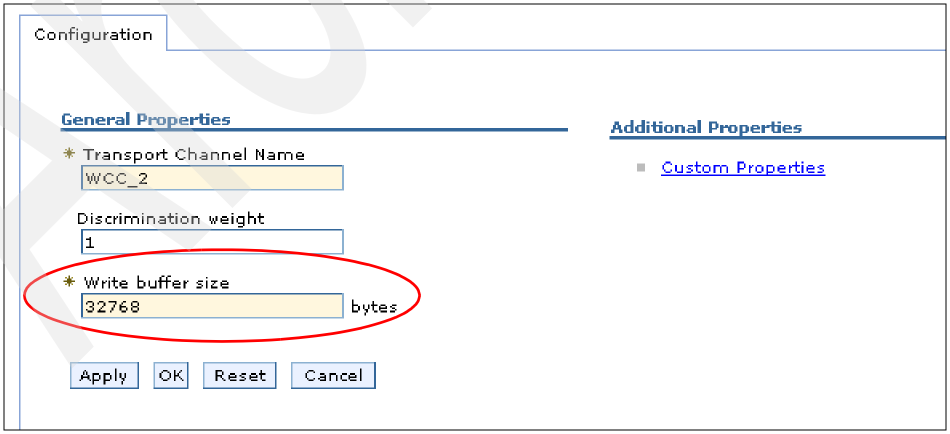


Figure 7-25 Web container transport channel settings

Adjust the settings for the bounded buffer

Even though the default bounded buffer parameters are optimal for most of the environments, you might have to change the default values in certain situations and for some operating systems to enhance performance. Changing the bounded buffer parameters can degrade performance. Therefore, make sure that you tune the other related areas, such as the Web container and ORB thread pools, before deciding to change the bounded buffer parameters.

To change the bounded buffer parameters:

1. In the administrative console, click **Servers** → **Application Servers** → ***server***.
2. Under Server Infrastructure, click **Java and Process Management** → **Process Definition** → **Java Virtual Machine**.
3. Specify one of the following parameters in the Generic JVM arguments field.
4. Click **Apply** or **OK**.
5. Enter one of the following custom properties in the Name field and an appropriate value in the Value field, and then click **Apply** to save the custom property and its setting.

   – `com.ibm.ws.util.BoundedBuffer.spins_take=`*value*

     This parameter specifies the number of times a Web container thread is allowed to attempt to retrieve a request from the buffer before the thread is suspended and enqueued. This parameter enables you to trade off the cost of performing possibly unsuccessful retrieval attempts, with the cost to suspending a thread and activating it again in response to a put operation.

     The default value of this parameter is 4. However, in practice an integer between 2 and 8 have shown the best performance results. To usage is `com.ibm.ws.util.BoundedBuffer.spin_take=5`. It means that five attempts are made before the thread is suspended.

   – `com.ibm.ws.util.BoundedBuffer.yield_take=true` or `false`

     Specifies that a thread yields the CPU to other threads after a set number of attempts to take a request from the buffer. Typically a lower number of attempts is preferable. The default value is false. However, the recommended value is true, but the effect of yields is implementation specific for individual platforms.

– `com.ibm.ws.util.BoundedBuffer.spins_put=`*value*

Specifies the number of attempts an InboundReader thread makes to put a request into the buffer before the thread is suspended and enqueued. This value allows to trade off between the cost of repeated, possibly unsuccessful, attempts to put a request into the buffer with the cost to suspend a thread and reactivate it in response to a take operation.

The default value of this parameter is 4. However, in practice an integer between 2 and 8 have shown the best performance results. To usage is `com.ibm.ws.util.BoundedBuffer.spin_put=5`. It means that five attempts are made before the thread is suspended.

– `com.ibm.ws.util.BoundedBuffer.yield_put=true` or `false`

Specifies that a thread yields the CPU to other threads after a set number of attempts to put a request into the buffer. Typically a lower number of attempts is preferable.

The default value is false. However, the recommended value is true, but the effect of yields is implementation specific for individual platforms.

– `com.ibm.ws.util.BoundedBuffer.wait=`*number of milliseconds*

Specifies the maximum length of time, in milliseconds, that a request might unnecessarily be delayed if the buffer is completely full or if the buffer is empty. The default value is 10000 milliseconds or 10 seconds. A value of 10000 milliseconds usually works well. In rare instances when the buffer becomes either full or empty, a smaller value guarantee a more timely handling of requests, but there is usually a performance impact to using a smaller value.

> **Important:** Click **Apply** and then **Save** to save these changes after you have made any changes.

7.2.9 Tuning data sources and associated connection pools

For better application performance, you can tune some data access resources through the WebSphere Application Server administrative console. Tune these properties of data sources and connection pools to optimize the performance of transactions between your application and datastore.

Data source tuning

To view the administrative console page where you configure the following properties, click **Resources** → **JDBC** → **Data sources** → ***data_source_name.*** Then in the additional properties section click **WebSphere Application Server data source properties**.

Statement cache size

Specifies the number of statements that can be cached per connection. The WebSphere Application Server data source optimizes the processing of prepared statements and callable statements by caching those statements that are not being used in an active connection. Both statement types help reduce overhead for transactions with backend data.

- A prepared statement is a precompiled SQL statement that is stored in a PreparedStatement object. Application Server uses this object to run the SQL statement multiple times, as required by your application run time, with values that are determined by the run time.
- A callable statement is an SQL statement that contains a call to a stored procedure, which is a series of precompiled statements that perform a task and return a result. The statement is stored in the CallableStatement object. Application Server uses this object to run a stored procedure multiple times, as required by your application run time, with values that are determined by the run time. See Figure 7-26.

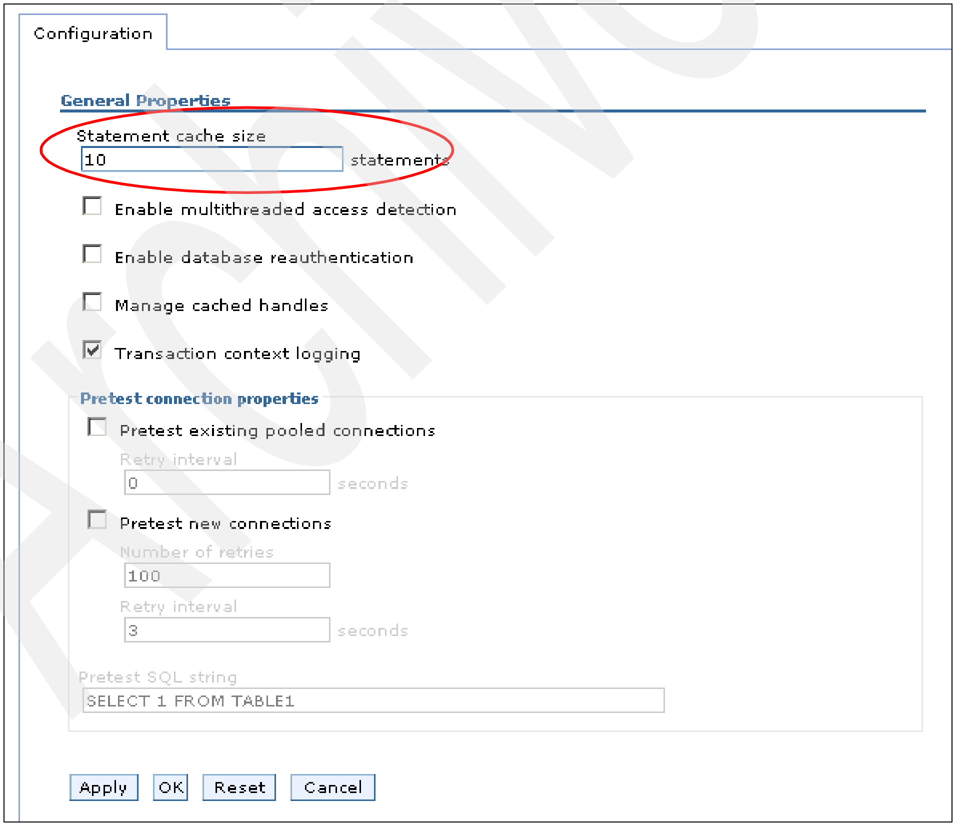


Figure 7-26 WebSphere Application Server data source properties

In general, the more statements your application has, the larger the cache should be. Be aware, however, that specifying a larger statement cache size than required wastes application memory and does not improve performance.

Determine the value for your cache size by adding the number of uniquely prepared statements and callable statements (as determined by the SQL string, concurrency, and the scroll type) for each application that uses this data source on a particular server. This value is the maximum number of possible statements that can be cached on a given connection over the life of the server.

> **Default:** For most databases the default is 10. Zero means there is no cache statement.

Connection pool tuning

To view the administrative console page where you configure the following properties, click **Resources** → **JDBC** → **Data sources** → ***data_source_name.*** Then in the additional properties section click **Connection pool properties**.

Maximum connections

Specifies the maximum number of physical connections that can be created in this pool. These are the physical connections to the backend datastore. When this number is reached, no new physical connections are created; requestors must wait until a physical connection that is currently in use is returned to the pool.

The default value of this properties is 10. For optimal performance, set the value for the connection pool lower than the value for the Web container threadpool size. Lower settings, such as 10 to 30 connections, might perform better than higher settings, such as 100. See Figure 7-27.

Minimum connections

Specifies the minimum number of physical connections to maintain. Until this number is exceeded, the pool maintenance thread does not discard physical connections.

The default value of this properties is 1. If you set this property for a higher number of connections than your application ultimately uses at run time, you do not waste application resources. WebSphere Application Server does not create additional connections to achieve your minimum setting. Of course, if your application requires more connections than the value you set for this property, application performance diminishes as connection requests wait for fulfillment.

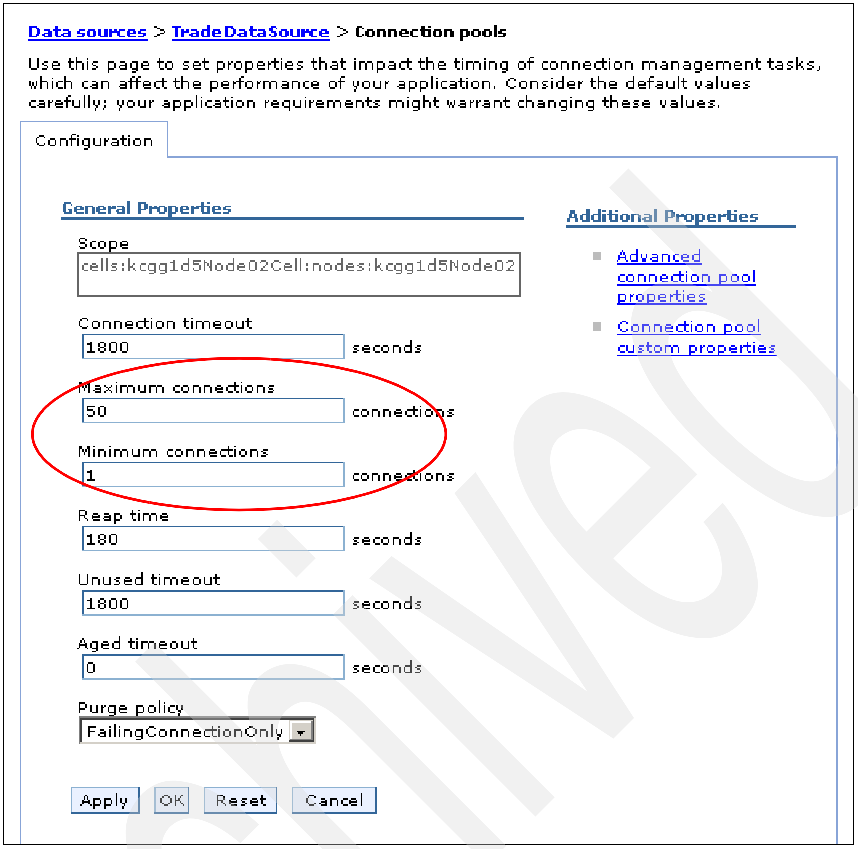


Figure 7-27 Connection pools tuning

7.2.10 Tuning session management

WebSphere Application Server session support has features for tuning session performance and operating characteristics, particularly when sessions are configured in a distributed environment. These options support the administrator flexibility in determining the performance and failover characteristics for their environment.

Performance tuning for session management persistence consists of defining the following characteristics:

- How often session data is written (write frequency settings).
- How much data is written (write contents settings).
- When the invalid sessions are cleaned up (session cleanup settings).

To view the administrative console page where you configure the following properties, click **Servers** → **Application Servers** → ***server*** → **Session Management** → **Distributed environment settings** → **Custom tuning parameters**. See Figure 7-28.

Note: Remember that session management options can also be set at the enterprise application level or at the Web module level.

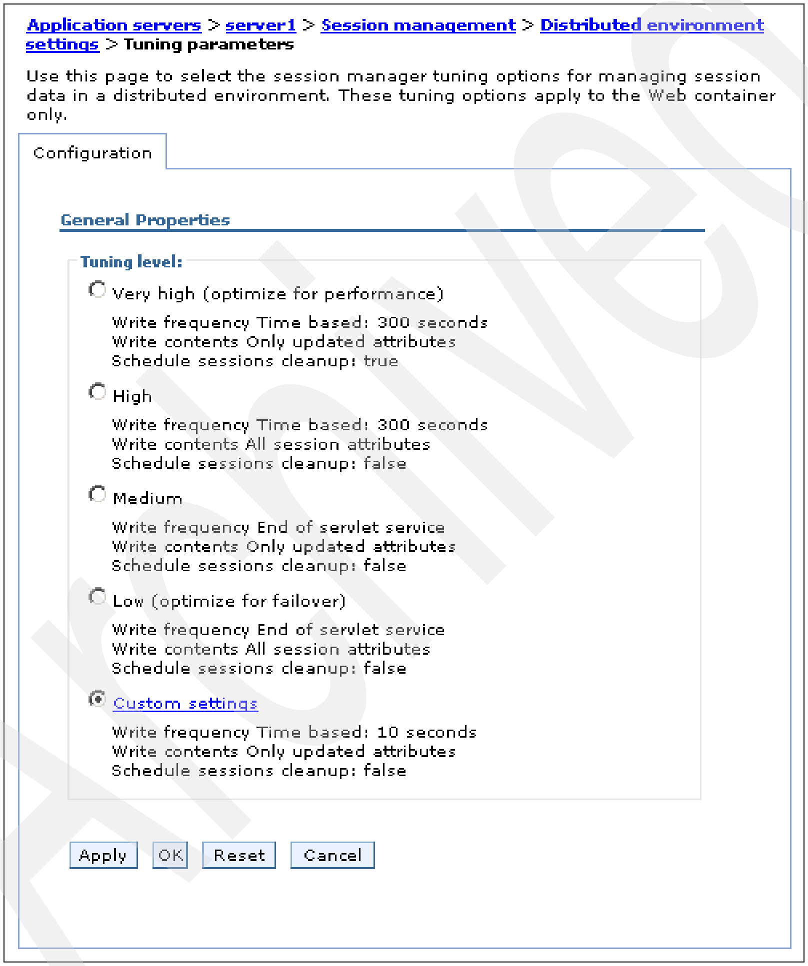


Figure 7-28 Tuning parameters for session management

Several combinations of these settings are predefined and available for selection, or you can customize them by click **Custom settings**. You can set each tuning parameter explicitly. See Figure 7-29.

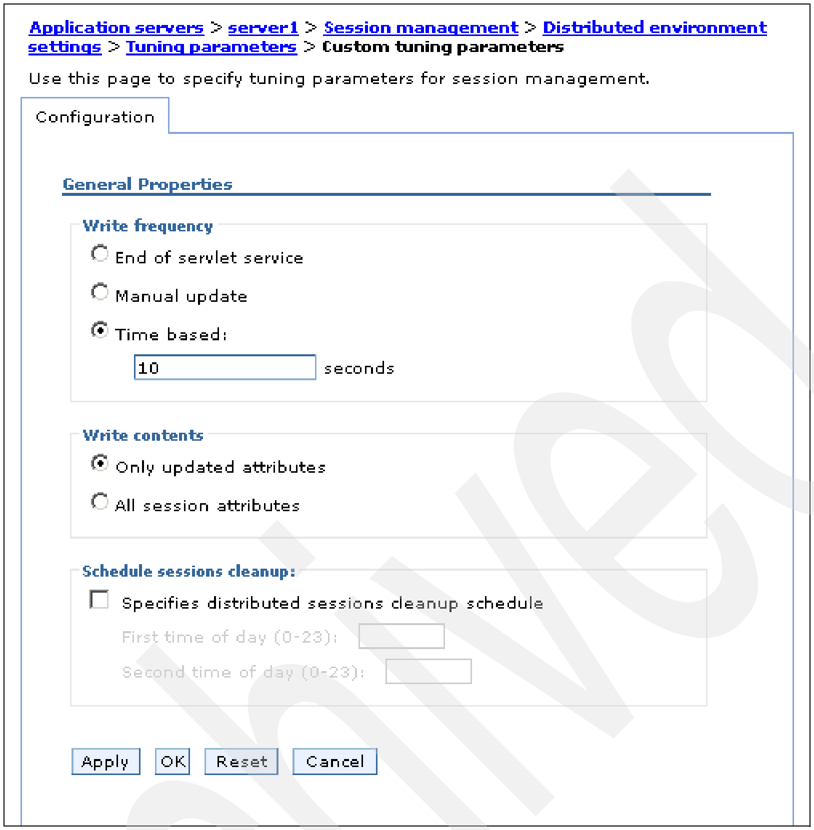


Figure 7-29 Custom tuning parameters

Writing frequency settings

You can select from three different settings that determine how often session data is written to the persistent data store:

- End of servlet service:

  If the session data has changed, it is written to the persistent store after the servlet finishes processing an HTTP request.

- Manual update:

  The session data is written to the persistent store when the `sync()` method is called on the IBMSession object.

- Time-based:

  The session data is written to the persistent store based on the specified write interval value.

Note: The last access time attribute is always updated each time the session is accessed by the servlet or JSP, whether the session is changed or not. This is done to make sure the session does not time out:

- If you choose the end of servlet service option, each servlet or JSP access results in a corresponding persistent store update of the last access time.
- If you select the manual update option, the update of the last access time in persistent store occurs on sync() call or at later time.
- If you use time-based updates, the changes are accumulated and written in a single transaction. This can significantly reduce the amount of I/O to the persistent store.

Consider an example where the Web browser accesses the application once every five seconds:

- In End of servlet service mode, the session would be written out every five seconds.
- In Manual update mode, the session would be written out whenever the servlet issues `IBMSession.sync()`. It is the responsibility of the servlet writer to use the IBMSession interface instead of the HttpSession Interface and the servlets/JSPs must be updated to issue the `sync()`.
- In Time-based mode, the servlet or JSP does not have to use the IBMSession class nor issue `IBMSession.sync()`. If the write interval is set to 120 seconds, then the session data is written out at most every 120 seconds.

End of servlet service

When the write frequency is set to the end of servlet service option, WebSphere writes the session data to the persistent store at the completion of the `HttpServlet.service()` method call. The write content settings determine output.

Manual update

In manual update mode, the session manager only sends changes to the persistent data store if the application explicitly requests a save of the session information.

Note: Manual updates use an IBM extension to HttpSession that is not part of the Servlet 2.4 API.

Manual update mode requires an application developer to use the IBMSession class for managing sessions. When the application invokes the `sync()` method, the session manager writes the modified session data and last access time to the persistent store. The session data written to the persistent store is controlled by the write contents option selected.

If the servlet or JSP terminates without invoking the `sync()` method, the session manager saves the contents of the session object into the session cache (if caching is enabled), but does not update the modified session data in the session database. The session manager only updates the last access time in the persistent store asynchronously, at later time. Example 7-3 shows how the IBMSession class can be used to manually update the persistent store.

Example 7-3 Using IBMSession for manual update of the persistent store

```
public void service (HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
{
   // Use the IBMSession to hold the session information
   // We need the IBMSession object because it has the manual update
   // method sync()
   com.ibm.websphere.servlet.session.IBMSession session =
      (com.ibm.websphere.servlet.session.IBMSession)req.getSession(true);

   Integer value = 1;

   //Update the in-memory session stored in the cache
   session.putValue("MyManualCount.COUNTER", value);

   //The servlet saves the session to the persistent store
   session.sync();
}
```

This interface gives the Web application developer additional control of when and if session objects go to the persistent data store. If the application does not invoke the `sync()` method, and manual update mode is specified, the session updates go only to the local session cache, not the persistent data store. Web developers use this interface to reduce unnecessary writes to the session database, and thereby to improve overall application performance.

All servlets in the Web application server must perform their own session management in manual update mode.

Time-based writes to the session database

Using the time-based write option writes session data to the persistent store at a defined write interval. The reasons for implementing time-based write lies in the changes introduced with the Servlet 2.2 API. The Servlet 2.2 specification introduced two key concepts:

- It limits the scope of a session to a single Web application.
- It both explicitly prohibits concurrent access to an HttpSession from separate Web applications, and allows for concurrent access within a given JVM.

Because of these changes, WebSphere provides the session affinity mechanism that assures an HTTP request is routed to the Web application handling its HttpSession. This assurance still holds in a WLM environment when using persistent HttpSessions. This means that the necessity to immediately write the session data to the persistent store can now be relaxed somewhat in these environments, as well as non-clustered environments, because the persistent store is used now only for failover and session cache full scenarios.

With this in mind, it is now possible to gain potential performance improvements by reducing the frequency of persistent store writes.

> **Note:** Time-based writes requires session affinity for session data integrity.

The following details apply to time-based writes:

- The expiration of the write interval does not necessitate a write to the persistent store unless the session has been touched (`getAttribute/setAttribute/removeAttribute` was called since the last write).
- If a session write interval has expired and the session has only been retrieved (`request.getSession()` was called since the last write), then the last access time is written to the persistent store regardless of the write contents setting.
- If a session write interval has expired and the session properties have been either accessed or modified since the last write, then the session properties are written in addition to the last access time. Which session properties get written is dependent on the write contents settings.
- Time-based write allows the servlet or JSP to issue IBMSession.sync() to force the write of session data to the database.
- If the time between session servlet requests for a particular session is greater than the write interval, then the session effectively gets written after each service method invocation.

- The session cache should be large enough to hold all of the active sessions. Failure to do this results in extra persistent store writes, because the receipt of a new session request can result in writing out the oldest cached session to the persistent store. To put it another way, if the session manager has to remove the least recently used HttpSession from the cache during a full cache scenario, the session manager writes that HttpSession using the Write contents settings upon removal from the cache.
- The session invalidation time must be at least twice the write interval to ensure that a session does not inadvertently get invalidated prior to getting written to the persistent store.
- A newly created session is always written to the persistent store at the end of the service method.

Writing content settings

The options available are:

- Only update attributes are written to the persistent store.
- All session attributes are written to the persistent store.

Session cleanup settings

WebSphere allows the administrator to defer (to off hours) the clearing of invalidated sessions from the persistent store. *Invalidated sessions* are sessions that are no longer in use and timed out. This can be done either once or twice a day. The fields available are:

- First time of day (0-23) is the first hour during which the invalidated persistent sessions are cleared from the persistent store. This value must be a positive integer between 0 and 23.
- Second time of day (0-23) is the second hour during which the invalidated persistent sessions are cleared from the persistent store. This value must be a positive integer between 0 and 23.
- Select **Schedule sessions cleanup** to enable this option.

Also, consider using schedule invalidation for intranet-style applications that have a somewhat fixed number of users wanting the same HTTP session for the whole business day.

Session performance considerations

This section includes guidance for developing and administering scalable, high-performance Web applications using WebSphere Application Server session support.

Session size

Large session objects pose several problems for a Web application. If the site uses session caching, large sessions reduce the memory available in the WebSphere instance for other tasks, such as application execution.

For example, assume a given application stores 1 MB of information for each user session object. If 100 users arrive over the course of 30 minutes, and assume the session timeout remains at 30 minutes, the application server instance must allocate 100 MB just to accommodate the newly arrived users in the session cache:

```
1 MB for each user session * 100 users = 100 MB
```

Note this number does not include previously allocated sessions that have not timed out yet. The memory required by the session cache could be considerably higher than 100 MB.

Web developers and administrators have several options for improving the performance of session management:

- Reduce the size of the session object.
- Reduce the size of the session cache.
- Add additional application servers.
- Invalidate unnecessary sessions.
- Increase the memory available.
- Reduce the session timeout interval.

Reducing session object size

Web developers must consider carefully the information kept by the session object:

- Removing information easily obtained or easily derived helps keep the session object small.
- Rigorous removal of unnecessary, unnecessary, or obsolete data from the session.
- Consider whether it would be better to keep a certain piece of data in an application database rather than in the HTTP session. This gives the developer full control over when the data is fetched or stored and how it is combined with other application data. Web developers can leverage the power of SQL if the data is in an application database.

Reducing object size becomes particularly important when persistent sessions are used. Serializing a large amount of data and writing it to the persistent store requires significant WebSphere performance overhead. Even if the Write contents option is enabled, if the session object contains large Java objects or collections of objects that are updated regularly, there is a significant performance penalty in persisting these objects. This penalty can be reduced by using time-based writes.

> **Notes:** In general, you can obtain the best performance with session objects that are less than 2 KB in size. When the session object exceeds 4-5 KB, you can expect a significant decrease in performance.
>
> Even if session persistence is not an issue, minimizing the session object size helps to protect your Web application from scale-up disasters as user numbers increase. Large session objects require more and more JVM memory, leaving no room to run servlets.

Session cache size

The session manager allows administrators to change the session cache size to alter the cache's memory footprint. By default, the session cache holds 1000 session objects. By lowering the number of session objects in the cache, the administrator reduces the memory required by the cache.

However, if the user's session is not in the cache, WebSphere must retrieve it from either the overflow cache, for local caching, or the session database, for persistent sessions. If the session manager must retrieve persistent sessions frequently, the retrievals can impact overall application performance.

WebSphere maintains overflowed local sessions in memory. Local session management with cache overflow enabled allows an unlimited number of sessions in memory. To limit the cache footprint to the number of entries specified in session manager, use persistent session management, or disable overflow.

> **Note:** When using local session management without specifying the Allow overflow property, a full cache results in the loss of user session objects.

Creating additional application servers

WebSphere also gives the administrator the option of creating additional application servers. Creating additional instances spreads the demand for memory across more JVMs, thus reducing the memory burden on any particular instance. Depending on the memory and CPU capacity of the machines involved, the administrator can add additional instances within the same machine. Alternatively, the administrator can add additional machines to form a hardware cluster, and spread the instances across this cluster.

> **Note:** When configuring a session cluster, session affinity routing provides the most efficient strategy for user distribution within the cluster, even with session persistence enabled. With cluster members, the Web server plug-in provides affinity routing among cluster member instances.

Invalidating unnecessary sessions

If the user no longer requires the session object, for example, when the user has logged out of the site, it should be invalidated. Invalidating a session removes it from the session cache, as well as from the session database.

Increasing available memory

WebSphere allows the administrator to increase an application server's heap size. By default, WebSphere allocates 256 MB as the maximum heap size. Increasing this value allows the instance to obtain more memory from the system, and thus hold a larger session cache.

A practical limit exists, however, for an instance heap size. The machine memory containing the instance has to support the heap size requested. Also, if the heap size grows too large, the length of the garbage collection cycle with the JVM might impact overall application performance. This impact has been reduced with the introduction of multi-threaded garbage collection.

Session timeout interval

By default, each user receives a 30 minute interval between requests before the session manager invalidates the user's session. Not every site requires a session timeout interval this generous. By reducing this interval to match the requirements of the average site user, the session manager purges the session from the cache and the persistent store, if enabled, more quickly.

Avoid setting this parameter too low and frustrating users. The administrator must take into account a reasonable time for an average user to interact with the site when setting the interval. User activities include reading returned data, filling out forms, and so on. Also, the interval must represent any increased response time during peak times on the site, such as heavy trading days on a brokerage site, for example.

Finally, in some cases where the persistent store contains a large number of entries, frequent execution of the timeout scanner reduces overall performance. In cases where the persistent store contains many session entries, avoid setting the session timeout so low it triggers frequent, expensive scans of the persistent store for timed-out sessions. Alternatively, the administrator should consider schedule-based invalidation where scans for invalid object can be deferred to a time that normally has low demand.

7.3 Tuning a Web server

WebSphere Application Server provides plug-ins for several Web server brands and versions. Each Web server operating system combination has specific tuning parameters that affect the application performance.

This section discusses some of the performance tuning settings associated with the Web servers. In addition to the settings mentioned in this section, additional information about Web server tuning can be found in the WebSphere Info Center article called “Tuning Web servers”.

Following is a list of tuning parameters specific to Web servers. The listed parameters might not apply to all of the supported Web servers. Check your Web server documentation before using any of these parameters.

- Tune the IBM HTTP Server 2.0.47.1, Apache 2.0.48, IBM HTTP Server 6.0, and IBM HTTP Server 6.1:

  Monitoring the CPU utilization and checking the IBM HTTP Server error_log and http_plugin.log files can help you diagnose Web server performance problems.

You can also configure the IBM HTTP Server to show a status page:

a. Edit the IBM HTTP Server **httpd.conf** file and remove the comment character (#) from the following lines in this file (Example 7-4).

Example 7-4 Remove comments from httpd.conf

```
#LoadModule status_module, modules/ApacheModuleStatus.dll,
#<Location/server-status>
#SetHandler server-status
#</Location>
```

b. **Save** the changes and **restart** the IBM HTTP Server.

c. In a Web browser, go to: http://*yourhost/*server-status. Alternatively, click **Reload** to update status.

d. Optionally, if the browser supports refresh, go to http://*your_host*/server-status?refresh=5 to refresh every five seconds.

All of these Web servers allocate a thread to handle each client connection. Ensuring that enough threads are available for the maximum number of concurrent client connections helps prevent this tier from being a bottleneck. The settings for these Web servers can be tuned by making changes to the httpd.conf file on the Web server system.

You can check the IBM HTTP Server error_log file to see if there are any warnings about having reached the maximum number of clients (MaxClients). There are several parameters, depending on the specific operating system platform, that determine the maximum number of clients the Web server supports. See the URL:

`http://httpd.apache.org/docs-2.0/mod/mpm_common.html#maxclients`

- Support thousands of concurrent clients:

  It is not unusual for a single IBM HTTP Server system to support thousands of concurrent clients. If your requirements are to support more concurrent clients than the number of threads that are supported by the Web server operating system and hardware, consider using multiple Web servers.

- Respond to a Connection Refused error message:

  Some clients might receive a Connection Refused error message if there is a sudden increase in the number of clients. Increasing the ListenBacklog and StartServer parameters can reduce or eliminate this error.

  – The ListenBacklog parameter indicates to the operating system the maximum allowed number of pending connections. Although the IBM HTTP Server default is 511, the actual value can be much higher or lower depending on the corresponding operating system parameter. To handle large numbers of simultaneous connections, this parameter and the corresponding OS parameter might have to be set to the number (possibly thousands) of expected simultaneous connections.

  – The StartServers parameter indicates the number of IBM HTTP Server processes to initially start. Pre-starting these IBM HTTP Server threads/processes reduces the chance of a user having to wait for a new process to start. You should set this parameter to a value equal to the MinSpareServers parameter so that the minimum number of IBM HTTP Server processes required for this client load is started immediately.

- Prevent the frequent creation and destruction of client threads/processes as the number of users change.

  You can a use the *MinSpareServers* and *MaxSpareServers* to specify the minimum and maximum number of servers (client threads/processes) that can exist in an idle state. To prevent frequent creation and destruction of client threads/processes as the number of users change, set this range large enough to include the maximum number of simultaneous users.

- Change the setting on the Web server's Access logging parameter to reduce the load on the Web server. If you do not have to log every access to the Application Server, change the default value of the Web server's Access logging parameter. This change reduces the load on the Web server.
- Modify the settings of the Load balancing option and Retry interval Web server plug-in properties to improve performance. You can improve the performance of IBM HTTP Server (with the WebSphere Web server plug-in) by modifying the following Web server plug-in configuration properties:
  - Load balancing option, which specifies the load balancing option that the plug-in uses in sending requests to the various application servers associated with that Web server.

    The goal of the default load balance option, Round Robin, is to provide an even distribution of work across cluster members. Round Robin works best with Web servers that have a single process sending requests to the Application Server. If the Web server is using multiple processes to send requests to the Application Server, the Random option can sometimes yield a more even distribution of work across the cluster.

  - Retry interval. which specifies the length of time to wait before trying to connect to a server that has been marked temporarily unavailable.

    The plug-in marks a server temporarily unavailable if the connection to the server fails. Although a default value is 60 seconds, you might have to lower this value in order to increase throughput under heavy load conditions. Lowering the RetryInterval might help when the IBM HTTP Server is configured to have fewer than 10 threads per process.

    How can lowering the RetryInterval affect throughput? If the plug-in attempts to connect to a particular application server while the application server threads are busy handling other connections, which happens under heavy load conditions, the connection might time out, causing the plug-in to mark the server temporarily unavailable. If the same plug-in process has other connections open to the same server and a response is received on one of these connections, the server is marked again. If there are only a few threads per IBM HTTP Server process, there might not be an established connection to this application server. When this situation occurs, the plug-in must wait for the entire retry interval.

> **Note:** Although lowering the RetryInterval can improve performance, if all the application servers are running, a low value can have an adverse affect when one of the application servers is down. In this case, each IBM HTTP Server process attempts to connect and fail more frequently, resulting in increased latency and decreased overall throughput.

Making these changes can help the IBM HTTP Server to support more WebSphere Application Server users. To modify these properties, in the administrative console, click **Servers** → **Web Servers** → ***Web_server_name*** → **Plug-in properties** → **Request routing.**

7.4 DB2 tuning parameters

DB2 has many parameters that you can configure to optimize database performance. This section explain only few parameters in the DB2 that can be tuned for improving application performance in WebSphere Application Server v6.1. For complete DB2 tuning information, refer to *DB2 UDB Administration Guide: Performance*, SC09-4821-01.

http://www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss?CTY=US&FNC=SRX&PBL=SC09-4821-01

7.4.1 DB2 logging

DB2 has corresponding log files for each database that provides services to administrators, including viewing database access and the number of connections. For systems with multiple hard disk drives, you can gain large performance improvements by setting the log files for each database on a different hard drive from the database files.

You can set this parameter at a DB2 command prompt, issue the following command.

```
db2 update db cfg for [database_name] using newlogpath
[fully_qualified_path]
```

The default of this parameter is logs reside on the same disk as the database. However the recommended value is use a separate high-speed drive, preferably performance enhanced through a redundant array of independent disk (RAID) configuration.

7.4.2 DB2 configuration advisor

Located in the DB2 Control Center, this advisor calculates and displays recommended values for the DB2 buffer pool size, the database, and the database manager configuration parameters, with the option of applying these values. See more information about the advisor in the online help facility within the Control Center.

7.4.3 DB2 - MaxAppls and MaxAgents

When configuring the data source settings for the databases, confirm the DB2 MaxAppls setting is greater than the maximum number of connections for the data source. If you are planning to establish clones, set the MaxAppls value as the maximum number of connections multiplied by the number of clones. The same relationship applies to the session manager number of connections. The MaxAppls setting must be equal to or greater than the number of connections. If you are using the same database for session and data sources, set the MaxAppls value as the sum of the number of connection settings for the session manager and the data sources.

For example, MaxAppls = (number of connections set for the data source + number of connections in the session manager) multiplied by the number of clones.

After calculating the MaxAppls settings for the WebSphere Application Server database and each of the application databases, verify that the MaxAgents setting for DB2 is equal to or greater than the sum of all of the MaxAppls values. For example, MaxAgents = sum of MaxAppls for all databases.

7.4.4 DB2 buffpage

This parameter is used to improve database system performance. Buffpage is a database configuration parameter. It defines the amount of memory that is allocated to a new define bufferpool. A buffer pool is a memory storage area where database pages containing table rows or index entries are temporarily read and changed. Data is accessed much faster from memory than from disk.

To view the current value of buffpage for database *<dbname>*, issue the DB2 command **get db cfg for <dbname>** and look for the value **Buffer pool size (page)**. To set buffer pool size to a value of *n*, issue the DB2 command **update db cfg for <dbname> using BUFFPAGE *n*** and set **NPAGES** to -1 as follows.

```
db2   <-- go to DB2 command mode, otherwise the following "select"
          does not work as is
connect to x    <-- (where x is the particular DB2 database name)
select * from syscat.bufferpools
   (and note the name of the default, perhaps: IBMDEFAULTBP)
   (if NPAGES is already -1, there is no need to issue following
   command)
alter bufferpool IBMDEFAULTBP size -1
   (re-issue the above "select" and NPAGES now equals -1)
```

You can collect a snapshot of the database while the application is running and calculate the buffer pool hit ratio as follows:

1. Collect the snapshot:
   a. Issue the **update monitor switches using bufferpool** on command.
   b. Make sure that bufferpool monitoring is on by issuing the **get monitor switches** command.
   c. Clear the monitor counters with the **reset monitor all** command.
2. Run the application.
3. Issue the **get snapshot for all databases** command before all applications disconnect from the database, otherwise statistics are lost.
4. Issue the **update monitor switches using bufferpool off** command.
5. Calculate the hit ratio by looking at the following database snapshot statistics:
   – Buffer pool data logical reads
   – Buffer pool data physical reads
   – Buffer pool index logical reads
   – Buffer pool index physical reads

The default value for buffpage parameter is 250, and the recommended value is depend on the snapshot, you can continue increasing the value until the snapshot shows a satisfactory hit rate.

The buffer pool hit ratio indicates the percentage of time that the database manager did not have to load a page from disk to service a page request. That is, the page is already in the buffer pool. The greater the buffer pool hit ratio, the lower the frequency of disk input and output. Calculate the buffer pool hit ratio as follows:

```
P = buffer pool data physical reads + buffer pool index physical reads
L = buffer pool data logical reads + buffer pool index logical reads
Hit ratio = (1-(P/L)) * 100%
```

7.4.5 DB2 query optimization level

DB2 query optimization level sets the amount of work and resources that DB2 puts into optimizing the access plan. When a database query runs in DB2, various methods are used to calculate the most efficient access plan. The range is from 0 to 9. An optimization level of 9 causes DB2 to devote a lot of time and all of its available statistics to optimizing the access plan.

The optimization level is set on individual databases and can be set with either the command line or with the DB2 Control Center. Static SQL statements use the optimization level that is specified on the **prep** and **bind** commands. If the optimization level is not specified, DB2 uses the default optimization as specified by the dft_queryopt setting. Dynamic SQL statements use the optimization class that is specified by the current query optimization special register, which is set using the SQL Set statement. For example, the following statement sets the optimization class to 1:

```
Set current query optimization = 1
```

If the current query optimization register is not set, dynamic statements are bound using the default query optimization class. The default value of this parameter is 5 and the recommended value is depend on the requirements of the application. High levels should only be used when there are very complicated queries.

7.4.6 DB2 reorgchk

The performance of the SQL statements can be deteriorate after many updates, deletes, or inserts have been made. Use this parameter to obtain the current statistics for data and rebinding.

Use the DB2 **reorgchk update statistics on table all** command to perform the runstats operation on all user and system tables for the database to which you are currently connected. After that you have to rebind packages using the **bind** command. If statistics are available, issue the following command on DB2 CLP:

```
db2 -v "select tbname, nleaf, nlevels, stats_time from
sysibm.sysindexes"
```

If no statistic updates exist, the value of **nleaf** and **nlevels** are -1, and **stats_time** has an empty entry (for example: “-”). If the **runstats** command was previously run, the real-time stamp from completion of the runstats operation also displays under stats_time. If you think the time shown for the previous runstats operation is too old, run the **runstats** command again.

7.4.7 DB2 locktimeout

This parameter specifies the number of seconds that an application waits to obtain a lock. Setting this property helps avoid global deadlocks for applications.

To view the current value of the lock timeout property for database *<dbname>* issue the DB2 **`get db cfg for <dbname>`** command and look for the value, **`Lock timeout (sec)`**. To set lock timeout to a value of *n*, issue the DB2 **`update db cfg for <dbname> using LOCKTIMEOUT n`** command, where ***<dbname>*** is the name of the database and ***n*** is a value between 0 and 30 000 inclusive.

The default value of this parameter is -1, meaning lock timeout detection is turned off. In this situation, an application waits for a lock if one is not available at the time of the request, until either of the following events occurs:

- The lock is granted.
- A deadlock occurs.

The recommended value for this parameter is depend on your database access pattern, if your database access pattern tends toward a majority of writes, set this value so that it gives you early warning when a timeout occurs. A setting of 30 seconds suits this purpose. If your pattern tends toward a majority of reads, either accept the default lock timeout value, or set the property to a value greater than 30 seconds.

7.4.8 DB2 maxlocks

This parameter specifies the percentage of the lock list that is reached when the database manager performs escalation, from row to table, for the locks held by the application. Although the escalation process does not take much time, locking entire tables versus individual rows decreases concurrency, and potentially decreases overall database performance for subsequent attempts to access the affected tables.

To view the current value of the maxlocks property for database ***<dbname>*,** issue the DB2 **`get db cfg for <dbname>`** command and look for the **`Percentage of lock lists per application`** value. To set maxlock to a value of *n*, issue the DB2 **`update db cfg for <dbname> using MAXLOCKS n`** command, where ***<dbname>*** is the name of the database and ***n*** is a value between 1 and 100 inclusive.

Default value of this parameter is referring to the current database information for property default values per operating system. The recommended value is depend on the frequency of lock escalations. If lock escalations are causing performance concerns, you might have to increase the value of this parameter or the locklist parameter, which is described in the following paragraph.

7.4.9 DB2 locklist

This parameter specifies the amount of storage that is allocated to the lock list.

To view the current value of the locklist property for database *<dbname>* issue the DB2 **get db cfg for *<dbname>*** command and look for the **Max storage for lock list** value. To set locklist to a value of *n*, issue the DB2 **update db cfg for *<dbname>* using LOCKLIST *n*** command, where ***<dbname>*** is the name of the database and *n* is a value between 4 and 60 000 inclusive.

Default value of this parameter is referring to the current database information for property default values per operating system. The recommended value is depend on the frequency of lock escalations. If lock escalations are causing performance concerns, you might have to increase the value of this parameter or the maxlocks parameter, which is described in the previous paragraph. You can use the database system monitor to determine if lock escalations are occurring.

7.5 Workload Management

Workload Management is the concept of sharing requests across multiple instances of a resource. Workload Management techniques are implemented expressly for providing scalability and availability within a system. These techniques allow the system to serve more concurrent requests. Workload Management allows for better use of resources by distributing load more evenly. Components that are overworked, and therefore, perhaps a potential bottleneck, can be routed around with Workload Management algorithms. Workload Management techniques also provide higher resiliency by routing requests around failed components to duplicate copies of that resource.

In WebSphere Application Server, Workload Management is achieved by sharing requests across one or more application servers, each running a copy of the Web application. In more complex topologies, Workload Management is embedded in load balancing technologies that can be used in front of Web servers.

Workload Management (WLM) is a WebSphere facility to provide load balancing and affinity between nodes in a WebSphere clustered environment. WLM can be an important facet of performance. WebSphere uses WLM to send requests to alternate members of the cluster if the current member is too busy to process the request in a timely fashion. WebSphere routes concurrent requests from a user to the same application server to maintain session state.

Workload Management optimizes the distribution of client processing tasks. Incoming work requests are distributed to the application servers, enterprise beans, servlets, and other objects that can most effectively process the requests. Workload Management also provides failover when servers are not available, improving application availability.

Workload Management provides the following benefits to WebSphere Application server applications:

- It balances client workloads, allowing processing tasks to be distributed according to the capacities of the different machines in the system.
- It provides failover capability by redirecting client requests if one or more servers is unable to process them. This improves the availability of applications and administrative services.
- It enables systems to be scaled up to serve a higher client load than provided by the basic configuration. With clustering, additional instances of servers, servlets, and other objects can easily be added to the configuration.
- It enables servers to be transparently maintained and upgraded while applications remain available for users.
- It centralizes the administration of servers and other objects.

This section only discusses the concept of clustering application servers for balancing workloads as product functionality to improve performance. This section does not explain how to set up and configure cluster, however, in this section we discuss tuning of the Workload Management.

7.5.1 Clustering application servers

Clustering application servers that host Web containers automatically enables plug-in Workload Management for the application servers and the servlets they host. Routing of servlet requests occurs between the Web server plug-in and the clustered application servers using HTTP or HTTPS, as shown in Figure 7-30.

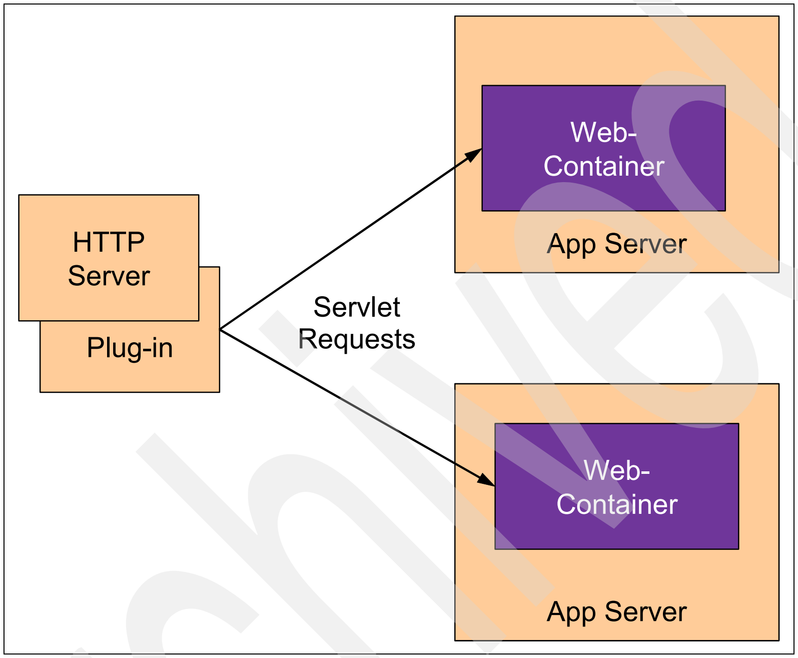


Figure 7-30 Plug-in (Web container) Workload Management

This routing is based on weights associated with the cluster members. If all cluster members have identical weights, the plug-in sends equal requests to all members of the cluster, assuming no strong affinity configurations. If the weights are scaled in the range from 0 to 20, the plug-in routes requests to those cluster members with the higher weight value more often. No requests are sent to cluster members with a weight of 0 unless no other servers are available. Weights can be changed dynamically during runtime by the administrator.

A guideline formula for determining routing preference is:

```
% routed to Server1 = weight1 / (weight1+weight2+...+weightn)
```

Where there are n cluster members in the cluster.

The Web server plug-in temporarily routes around unavailable cluster members.

Workload Management for EJB containers can be performed by configuring the Web container and EJB containers on separate application servers. Multiple application servers with the EJB containers can be clustered, enabling the distribution of EJB requests between the EJB containers, as shown in Figure 7-31.

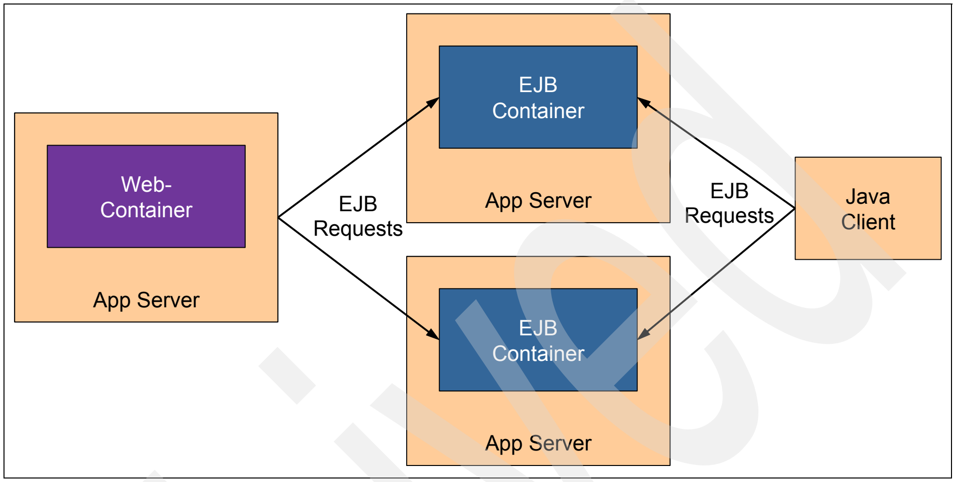


Figure 7-31 EJB Workload Management

In this configuration, EJB client requests are routed to available EJB containers in a round-robin fashion based on assigned server weights. The EJB clients can be servlets operating within a Web container, stand-alone Java programs using RMI/IIOP, or other EJBs.

The server-weighted, round-robin routing policy ensures a distribution based on the set of server weights that have been assigned to the members of a cluster. For example, if all servers in the cluster have the same weight, the expected distribution for the cluster is that all servers receive the same number of requests. If the weights for the servers are not equal, the distribution mechanism sends more requests to the higher weight value servers than the lower weight value servers. The policy ensures the desired distribution based on the weights assigned to the cluster members.

You can also choose to have requests sent to the node on which the client resides as the preferred routing. In this case, only cluster members on that node are chosen (using the round-robin weight method). Cluster members on remote nodes are chosen only if a local server is not available.

When planning for clustering, determine the number of application servers and their physical location. Determine the server weights to assign for application servers based on considerations such as system stability and speed. When creating the cluster, consider using the *prefer local* setting to ensure that when a client (for example, a servlet) calls an EJB, WLM attempts to select the EJB on the same system as the client, eliminating network communication.

7.5.2 Tuning a Workload Management configuration

You can set values for several Workload Management client properties to tune the behavior of the Workload Management runtime. To change the property values, you can use the Java Virtual Machine page of the administrative console or use the wsadmin tool. In cases such as where a servlet is a client to an enterprise bean, use the administrative console page for the application server where the servlet is running to configure the properties. The steps below describe how to change the values using the console.

1. In the administrative console, click **Servers** → **Application Servers** → *server_name* → **Java and Process Management** → **Process Definition**.
2. On the Process Definition page, click **Java Virtual Machine**.
3. specify one or more of the following command-line arguments in the Generic JVM arguments field:
   - **`-Dcom.ibm.CORBA.RequestTimeout=`*`timeout_interval`***

     If your application is experiencing problems with timeouts, this argument changes the value for the com.ibm.CORBA.RequestTimeout property, which specifies the timeout period for responding to requests sent from the client. This argument uses the -D option. timeout_interval is the timeout period in seconds. If your network experiences extreme latency, specify a large value to prevent timeouts. If you specify a value that is too small, an application server that participates in Workload Management can time out before it receives a response.

     Important: Be careful when specifying this property; it has no recommended value. Set it only if your application is experiencing problems with timeouts.

   - **`-Dcom.ibm.websphere.wlm.unusable.interval=`*`interval`***

     If the Workload Management state of the client is refreshing too soon or too late, this argument changes the value for the com.ibm.websphere.wlm.unusable.interval property, which specifies the time interval that the Workload Management client runtime waits after it marks a server as unavailable before it attempts to contact the server again. This argument uses the -D option. interval is the time in seconds between attempts. The default value is 300 seconds. If the property is set to a large value, the server is marked as unavailable for a long period of time. This prevents the Workload Management refresh protocol from refreshing the Workload Management state of the client until after the time period has ended.

7.5.3 Tuning Web server plug-in for balancing workloads

During normal operation, the backlog of connections pending to an application server is bound to grow. Therefore, balancing workloads among application servers in a network fronted by a Web server plug-in helps improve request response time. You can limit the number of connections that can be handled by an applications server. To do this:

1. Go to the **Servers** → **Application Servers** → ***server_name***.
2. Under Additional Properties, click **Web Server Plug-in properties**.
3. Select **Set limit for the Maximum number of connections that can be handled by the Application Server** field.
4. Specify in the Connections field the **maximum number of connections** you want to allow.
5. Then click **Apply** and **Save**.

When this maximum number of connections is reached, the plug-in, when establishing connections, automatically skips that application server, and tries the next available application server. If no application servers are available, an HTTP 503 response code is returned to the client. This code indicates that the server is currently unable to handle the request because it is experiencing a temporary overloading or because maintenance is being performed.

The capacity of the application servers in the network determines the value you specify for the maximum number of connections. The ideal scenario is for all of the application servers in the network to be optimally utilized. For example, if you have the following environment:

- There are 10 application servers in a cluster.
- All of these application servers host the same applications (that is, Application_1 and Application_2).
- This cluster of application servers is fronted by five IBM HTTP Servers.
- The IBM HTTP Servers get requests through a load balancer.
- Application_1 takes approximately 60 seconds to respond to a request
- Application_2 takes approximately 1 second to respond to a request.

Depending on the request arrival pattern, all requests to Application_1 might be forwarded to two of the application servers, say Appsvr_1 and Appsvr_2. If the arrival rate is faster than the processing rate, the number of pending requests to Appsvr_1 and Appsvr_2 can grow.

Eventually, Appsvr_1 and Appsvr_2 are busy and are not able to respond to future requests. It usually takes a long time to recover from this overloaded situation.

If you want to maintain 2500 connections, and optimally utilize the Application Servers in this example, set the number of maximum connections allowed to 50. (This value is arrived at by dividing the number of connections by the result of multiplying the number of Application Servers by the number of Web servers; in this example, 2500/(10x5)=50.)

Limiting the number of connections that can be established with an application server works best for Web servers that follow the threading model instead of the process model, and only one process is started.

The IBM HTTP Server V6.1 follows the threading model. To prevent the IBM HTTP Server from starting more than one process, change the following properties in the Web server configuration file (httpd.conf) to the indicated values:

```
ServerLimit            1
ThreadLimit            4000
StartServers           1
MaxClients             1024
MinSpareThreads        1
MaxSpareThreads        1024
ThreadsPerChild        1024
MaxRequestsPerChild    0
```

7.5.4 Improving performance in a high stress environment

If you use the default settings for a Microsoft Windows operating system, you might encounter Web server plug-in performance problems if you are running in a high stress environment. To avoid these problems, consider tuning the TCP/IP setting for this operating system. Two of the keys setting to tune are **TcpTimedWaitDelay** and **MaxUserPort**.

To tune the TcpTimedWaitDelay setting, change the value of the tcp_time_wait_interval parameter from the default value of 240 seconds, to 30 seconds:

1. Locate in the Windows Registry:

   ```
   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\tcpip\Para
   meters\TcpTimedWaitDelay
   ```

   If this entry does not exist in your Windows Registry, create it by editing this entry as a new DWORD item.

2. Specify, in seconds, a value between 30 and 300 inclusive for this entry. (It is recommended that you specify a value of 30.)

To tune the MaxUserPort setting:

1. Locate in the Windows Registry:

   ```
   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\tcpip\Para
   meters\MaxUserPort
   ```

   If this entry does not exist in your Windows Registry, create it by editing this entry as a new DWORD item.

2. Set the maximum number of ports to a value between 5000 and 65534 ports, inclusive. (It is recommended that you specify a value of 65534,)

See the Microsoft site for further information about these settings.

   http://www.microsoft.com

A

Additional best practices for SQLJ

This appendix provides information and best practices when using Structured Query Language for Java (SQLJ).

See 6.4, “JDBC and SQLJ” on page 317 for additional discussions.

Additional best practices for SQLJ

The following sections discuss the possibilities to improve the performance of a J2EE application accessing a DB2 database using Structured Query Language for Java (SQLJ).

Use positioned iterators, not named iterators

Iterators are the SQLJ equivalent to JDBC *result sets*. There are two ways to define iterators:

- By column name = named iterator
- By position in the select statement = positioned iterator

Named iterators are implemented as a wrapper around positioned iterators, with an associated hash table that maps column names to column numbers. Named iterators are more convenient to use, but more expensive than positioned iterators.

Example A-1 and Example A-2 show an example of the usage of a named iterator and a positioned iterator, respectively.

For best performance, the use of positioned iterators is recommended, as they do not have as much overhead.

Example: A-1 Named iterator

```
// Named Iterator
#sql iterator TestCase001A (short Fkeycr, Time Ftime, BigDecimal Fnum);
....
short wfkeycr;
Time wftime;
BigDecimal wfnum;
...
#sql [myconn] cursor002 = {SELECT FKEY, FTIME, FNUM FROM WRKTB01};
while (cursor002.next()) {
  wfkeycr = cursor002.Fkeycr();
  wftime = cursor002.Ftime();
  wfnum = cursor002.Fnum();
}
```

Example: A-2 Positioned iterator

```
// Positioned Iterator
#sql iterator TestCase001(short, Time, BigDecimal);
....
short wfkeycr;
Time wftime;
BigDecimal wfnum;
...
#sql [myconn] cursor001 = {SELECT FKEY, FTIME, FNUM FROM WRKTB01};
#sql {FETCH :cursor001 INTO :wfkeycr, :wftime, :wfnum};
```

Always customize with online checking enabled

We strongly recommend customizing the SQLJ profile using the online checker. The online checker is called by:

```
db2profc ... -online=<db2_location_name>
```

The online checker accesses the DB2 catalog to check JDBC/SQLJ-supported compatibility and convertibility processing and to determine the length of string columns. Java String objects do not have a concept of length, and can only be obtained from the catalog. In order to have the predicates considered for index access, the information in the *Database Request Module (DBRM)* must match definition in the DB2 catalog by data type and length. Online checking adds that information to the DBRM.

Not only are CHARACTER columns affected, but so are numeric columns. The optimizer will choose a non-matching index scan when the use of a host variable of type LONG to match a column of type INTEGER.

The *serialized profile* should be recustomized after each run of the SQLJ translator and available via the CLASSPATH at runtime.

If the SQLJ serialized profile is not customized, the Java application will execute dynamically using JDBC.

Explicit connection context objects

The default *connection context* object for a program is stored in a static variable of the default connection context class. In a multi-context environment (like WebSphere Application Server), the use of a default connection context is not thread-safe and must not be used.

An additional risk is the usage of the default connection context, which implicates a throughput bottleneck. Closing the context releases the resources maintained by the connection context (like statement handles) and closes the underlying database connection. When the constant KEEP_CONNECTION is passed as an argument, the context will be closed, but the database connection will be retained. This avoids new effort to get the connection when it is needed again.

Example A-3 gives the explicit connection context.

Example: A-3 Explicit connection context

```
// Connection context declaration
#sql context ctx;
...
//get context
myconn=new ctx(Conn1);
...
//use context in SQL
#sql [myconn] {set transaction isolation level read committed};
...
#sql [myconn] cursor001 = {SELECT FKEY,FSMALLINT,FINT FROM WRKTB01
WHERE FKEY >= :wfkey};
...
//close context but keep database connection
myconn.close(ConnectionContext.KEEP_CONNECTION);
```

Figure A-1 shows a simple performance measurement. With an explicit context, the number of transactions per second is nearly two times higher than with a default context.

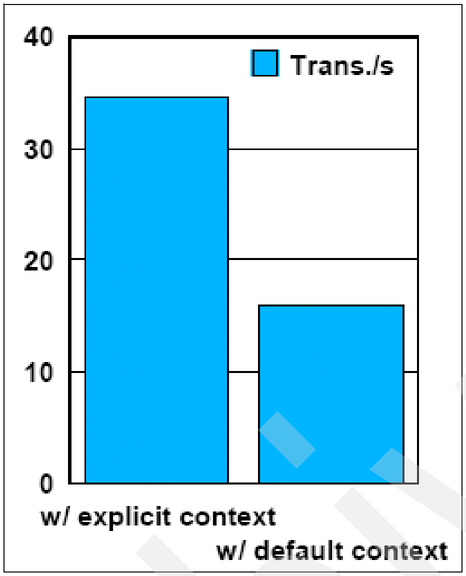


Figure A-1 Explicit context compared to default context

Check explain tables

Whenever DB2 data is read or written, an SQL statement is executed. Once DB2 accepts a statement, an execution plan is created. The execution plan defines how the DBMS finds and writes data. For example, the DBMS has to decide whether an index is used or not and which index is used. To discover this information, the execution plan can be read out. Therefore the SQL statement EXPLAIN PLAN is used.

```
EXPLAIN PLAN [SET STATEMENT_ID=<id>;INTO <table>] FOR <SQL-Statment>
```

When EXPLAIN PLAN is executed, DB2 writes the results into the PLAN_TABLE.

It is a good idea to always bind with EXPLAIN(YES) and to check the PLAN_TABLE for potential performance problems, for example, table space scan, merge scan join, and non-matching index scan.

See Chapter 26, “Using EXPLAIN to improve SQL performance”, in the *Application Programming and SQL Guide*, SC26-9933, for information about how to set up and interpret a PLAN_TABLE.

Alternatively or additionally, you can use DB2 Visual Explain Version 8, which is a free feature of DB2 for z/OS. It lets you graphically analyze the access paths that DB2 chooses, which eliminates the need to manually interpret the plan_table output. You can download Visual Explain at:

http://www.ibm.com/software/data/db2/zos/osc/ve/

Rebind packages regularly

We do not recommend to rebind static plans and packages after each and every REORG. Instead, statistic data, RUNSTATS, should be collected with a REORG. The recommendation is to rebind all plans/packages at least once within the life of any given release and when the RUNSTATS have changed significantly, that is, over 10%.

B

EJB 3 Feature Pack for WebSphere V6.1

In this appendix we introduce a sample application using JPA, which is part of the EJB 3.0 specification. It is included in the EJB 3.0 Feature Pack for WebSphere Application Server V6.1.

This appendix is organized into the following major sections:

Installation of prerequisites

The prerequisite components to install the EJB 3.0 Feature Pack for WebSphere Application Server 6.1 are as follows:

- IBM WebSphere Update Installer V6.1, if not installed in your environment
- WebSphere Application Server V6.1.0.7 or above
- EJB 3.0 Feature Pack beta code

Note: You can find more information about these prerequisites at the following URL:

https://www14.software.ibm.com/iwm/web/cc/earlyprograms/websphere/was61ejb3/download.shtml

Update Installer V6.1 and WebSphere FP installation: Overview

This section provides an overview of the Update Installer V6.1 and WebSphere Application Server Feature Pack installation.

Follow these steps:

1. Execute the `<UPDATE INSTALL ROOT>\UpdateInstaller\install.exe`. This is located in the Update Installer V6.1 code downloaded from the URL listed above. After following the installation steps, the final panel is displayed as shown in Figure B-1.

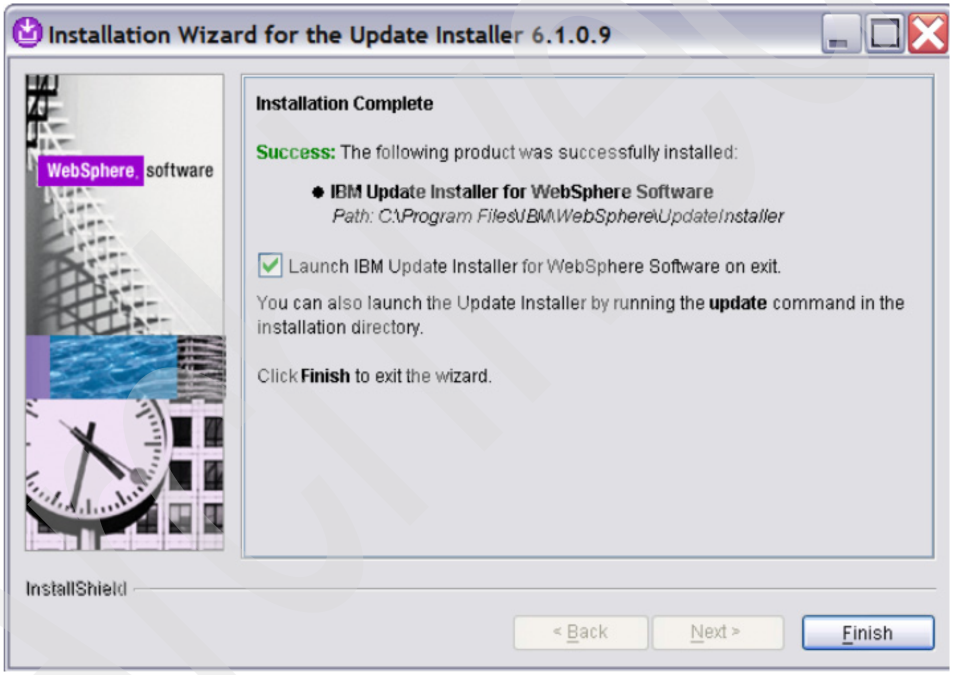


Figure B-1 Success panel of update installer

Click **Finish**. If you checked the option to **Launch IBM Update Installer for WebSphere Software**, you are taken to the next step automatically.

2. Execute the upgrade to V6.1.0.7 or the latest version available. In our example we are upgrading to WebSphere V6.1.0.9. The requirement here is that you have WebSphere Application Server V6.1 already installed. If not, download and install WebSphere Application Server V6.1 first. In Figure B-2, click **Next** to begin start of installation of WebSphere Application Server V6.1.0.9.

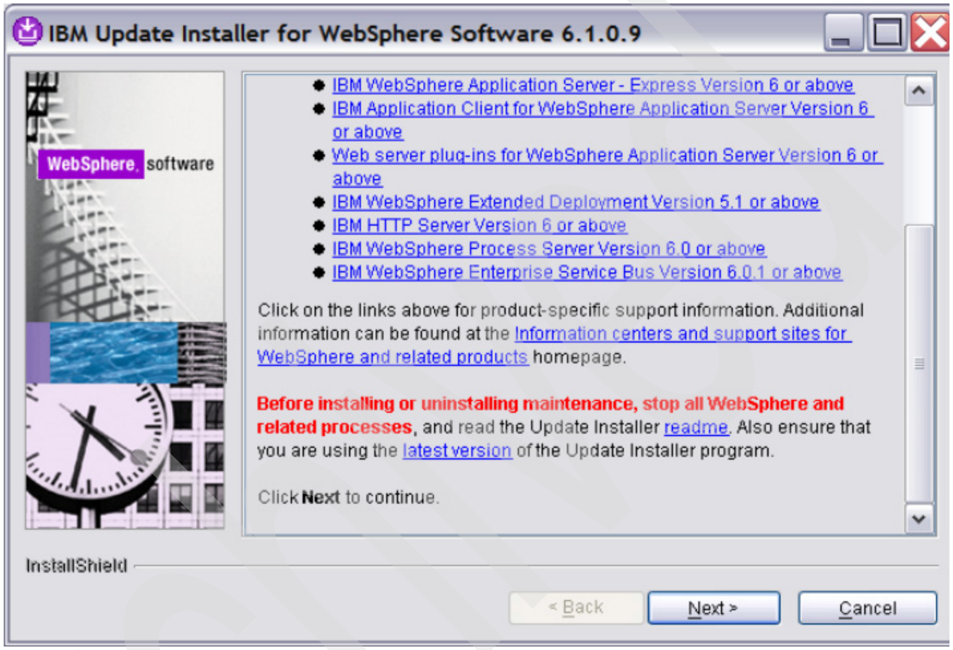


Figure B-2 Update installer for WebSphere software for V6.1.0.9

We are using WebSphere V6.1.0.9, however, the latest version of the EJB Feature Pack includes WebSphere V6.1.0.7 in the home installation directory at the same directory level as the EJB3 directory. See Figure B-3 for more information.

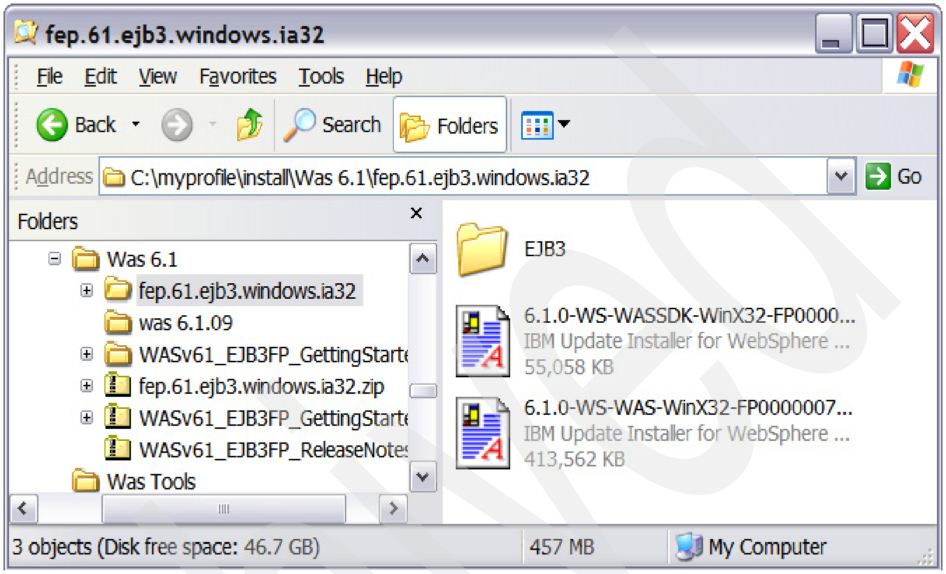


Figure B-3 Location of fix pack V6.1.0.7 inside EJB 3.0 Feature Pack install

3. The panel shown in Figure B-4 is displayed after **Next** has been clicked as described in list item 2 on page 592. Choose the fixpack location.

Figure B-4 Maintenance package selection

4. Select the fixpack location as shown in Figure B-5 and click **Next**.

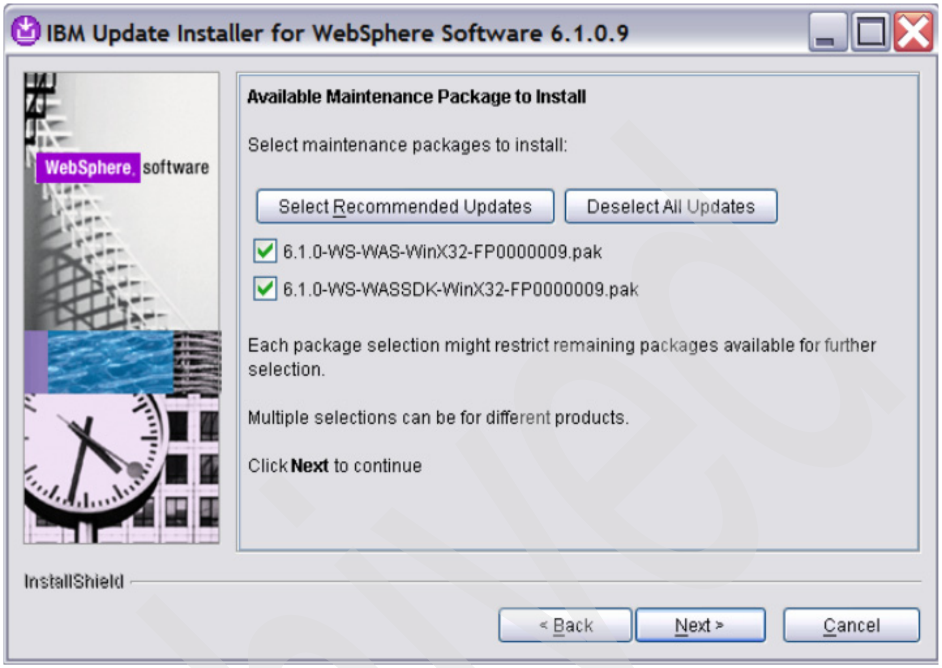


Figure B-5 Selected packages to install for WebSphere upgrade

5. Keep all selected options as shown in Figure B-5 and click **Next**.

6. You see the installation summary as shown in Figure B-6.

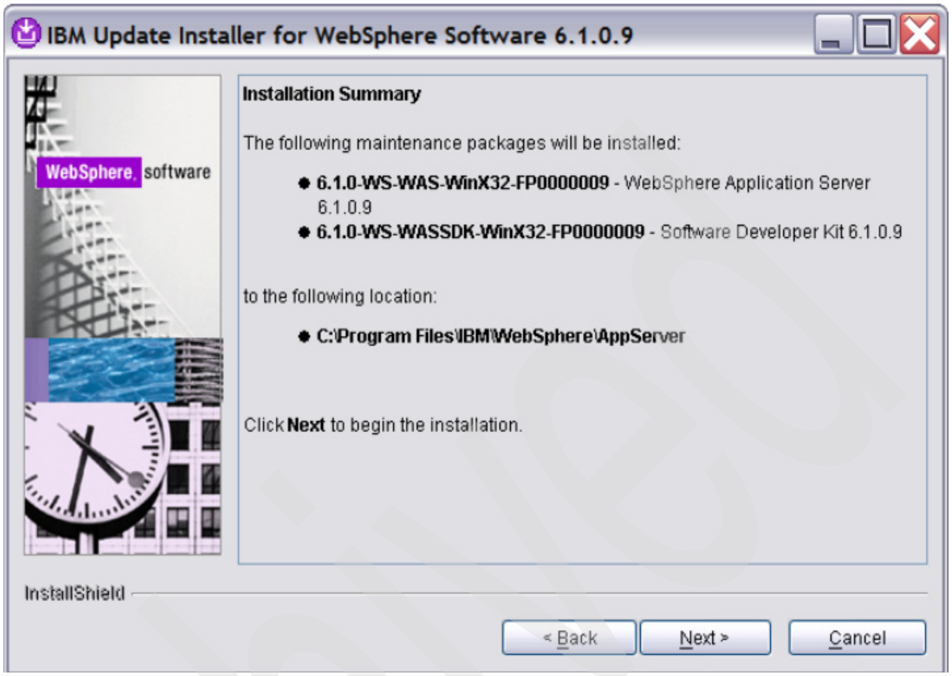


Figure B-6 Installation summary choices

7. Click **Next** in Figure B-6.

8. The installation results are shown in Figure B-7.

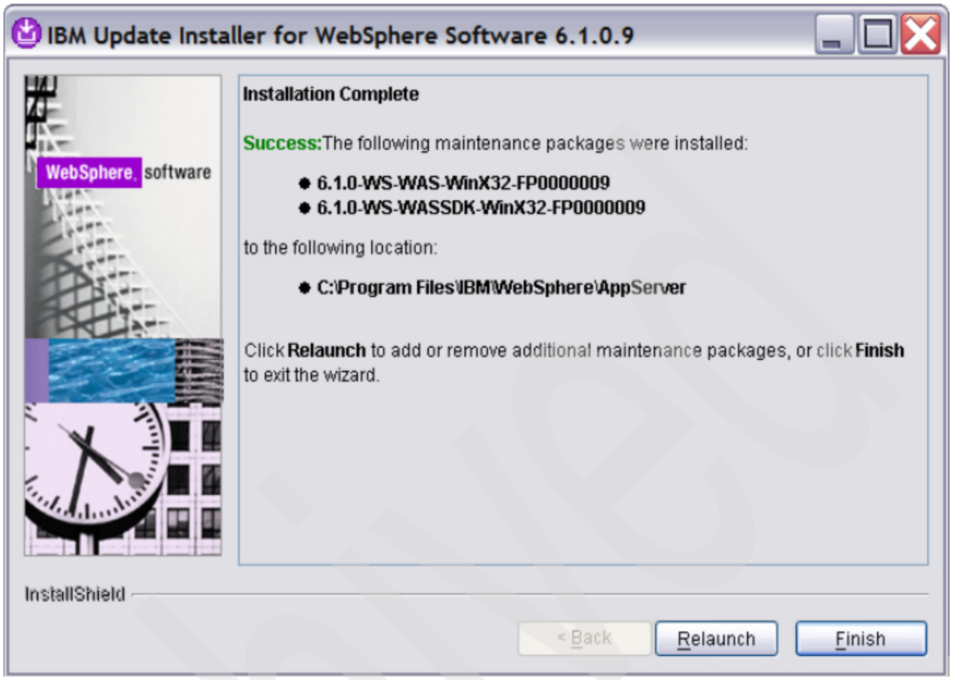


Figure B-7 Installation completion

EJB 3.0 Feature Pack installation

The general flow to install the EJB 3.0 Feature Pack is as follows:

1. Download the Feature Pack install image from:

   https://www14.software.ibm.com/iwm/web/cc/earlyprograms/websphere/was61ejb3/

   It is named fep.61.ejb3.<os>.<platform>.[zip,tar,tar.gz] where *<os>* is the operating system and <platform> is the hardware platform.
2. Make a new directory to extract the install image into and change to that directory.
3. Expand the image into the new directory.
4. Change directory to the EJB3 sub-directory and execute install.exe.
5. Following the steps, you select the WebSphere Application Server home directory as shown in Figure B-8.

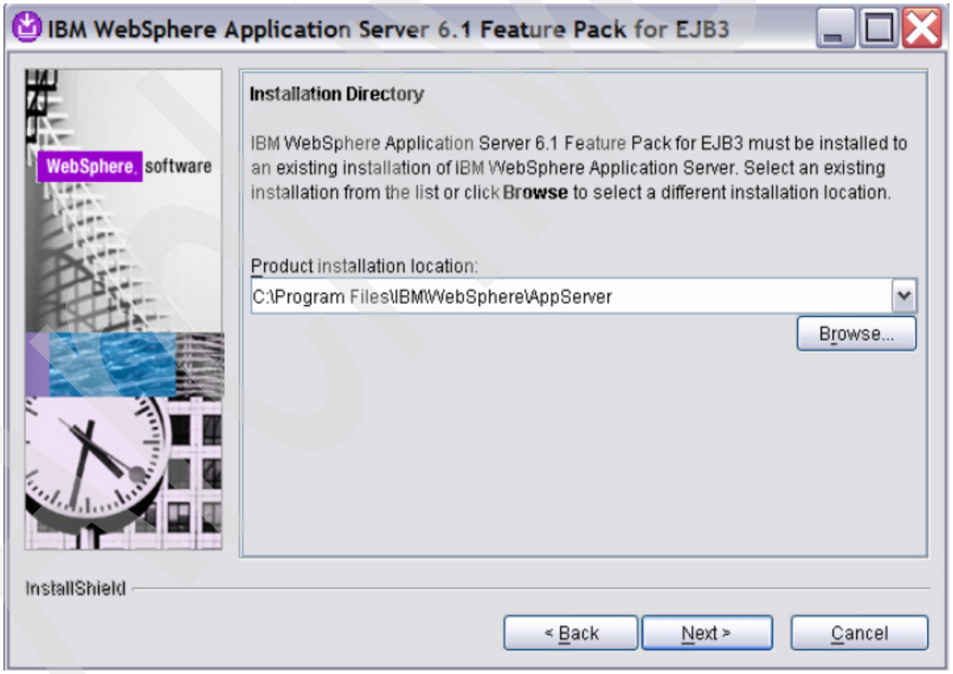


Figure B-8 Setting WebSphere location to install EJB 3.0 Feature Pack

6. The panel in Figure B-9 shows the summary of installation.

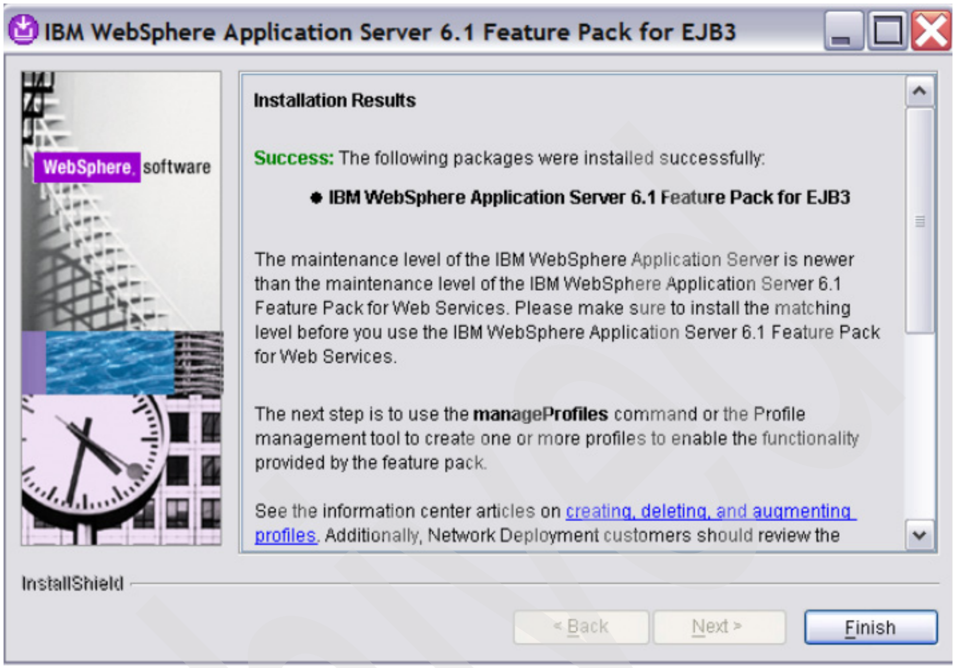


Figure B-9 Installation results

7. Run the Profile Management Tool (PMT) to create a new profile, application server instance, for installing EJB 3.0 based applications. All profiles have been enabled to use the Feature Pack beta (though we highly recommend that a new profile be created for Feature Pack exploitation). To do this, scroll down in the installation results step shown in Figure B-9 and enable the option to **Launch the Profile management tool**. See Figure B-10.

Figure B-10 Enable the Launch the Profile management tool option

8. From the panel shown in Figure B-10, you are guided to create a Profile. For more information about profile creation using the Profile Management Tool, see the following URL:

   http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/tpro_instancessaappserv.html

We give a short description of these steps only as an overview.

Using PMT: Overview

Next we provide an overview of using PMT:

1. As we described in step 8 on page 600, if you enable **Launch the Profile management tool** option, the panel shown in Figure B-11 is displayed.

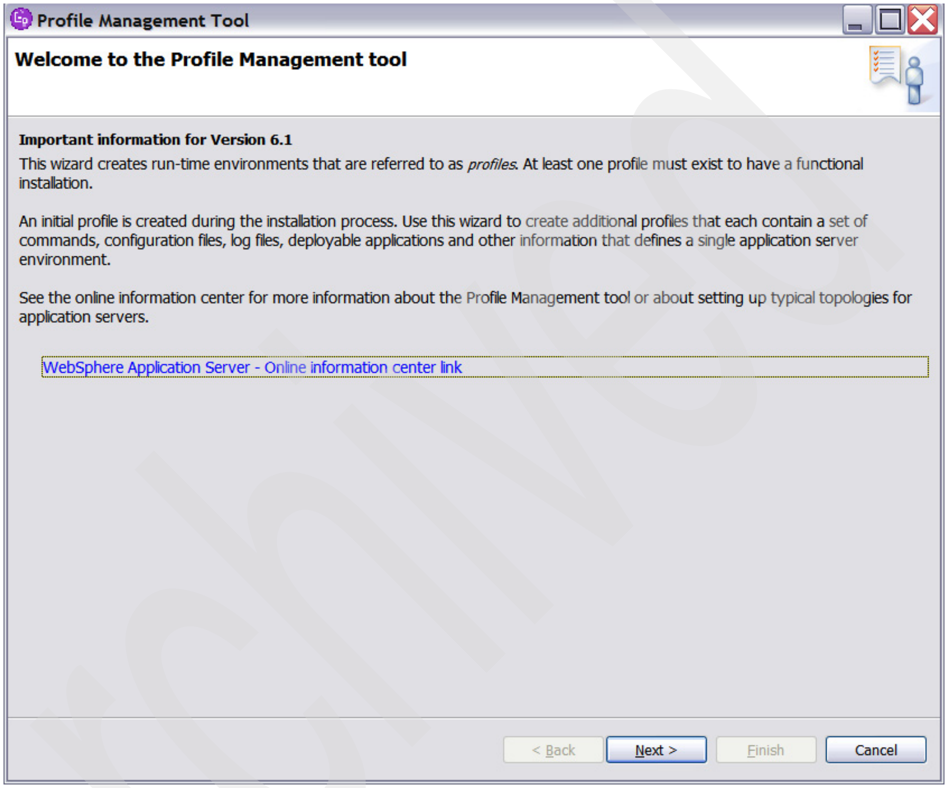


Figure B-11 Profile Management Tool first panel

2. In the panel shown in Figure B-12, we select a typical profile creation and click **Next**.

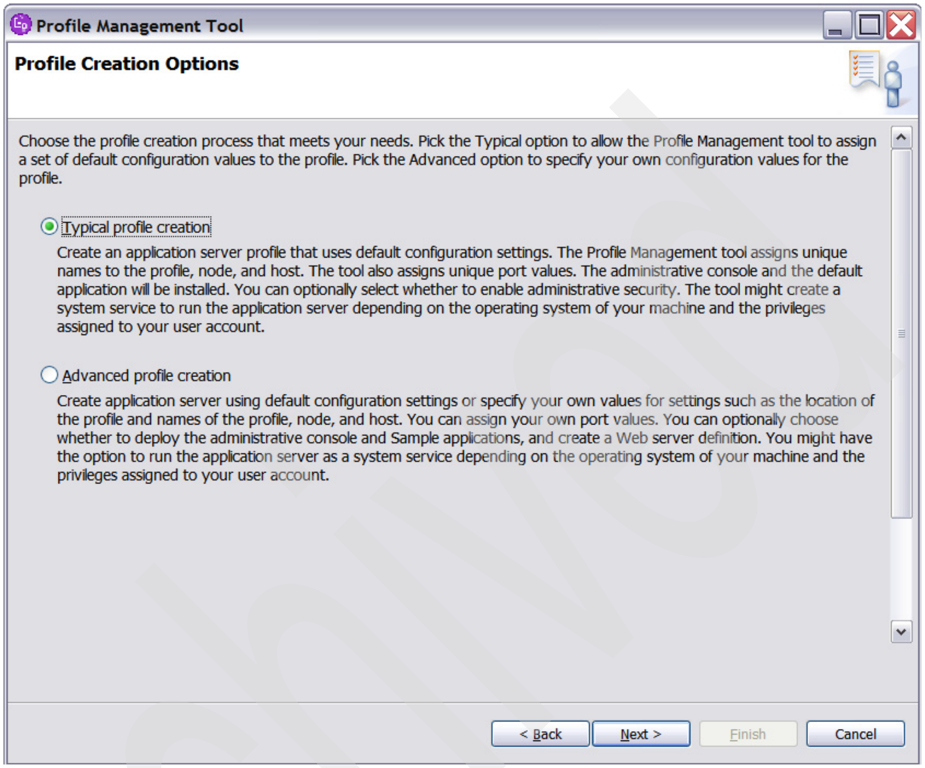


Figure B-12 Profile creation options

3. In the panel shown in Figure B-13, we enable administrative security and supply the required userid and password.

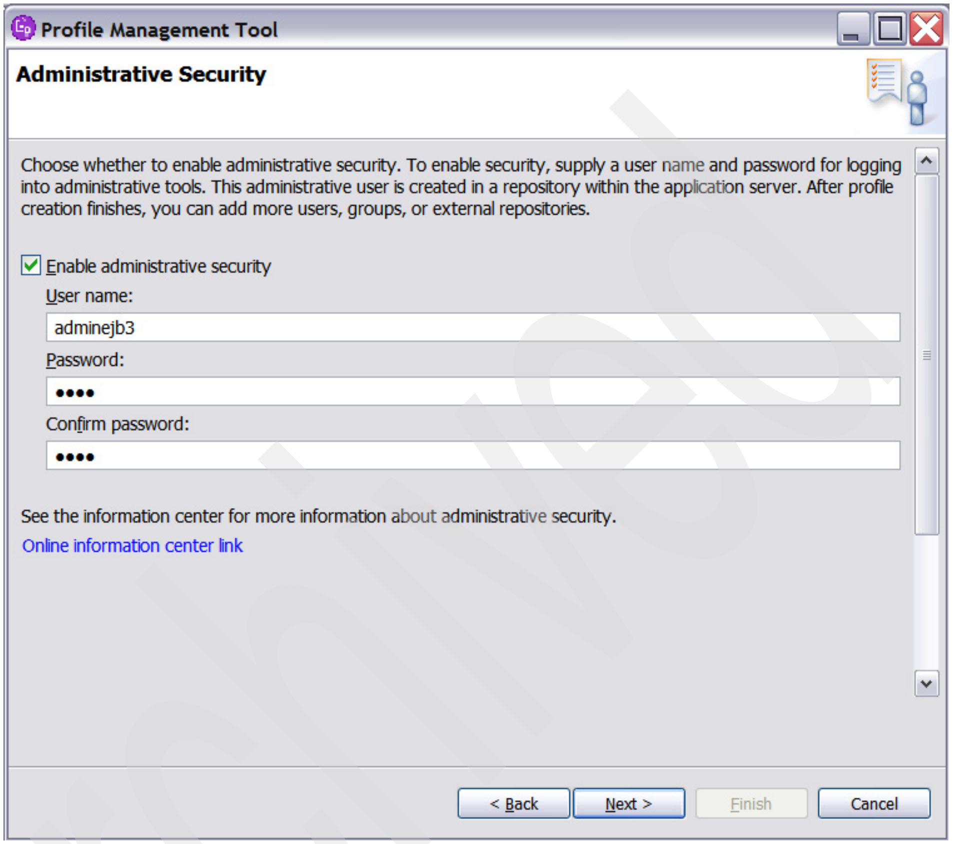


Figure B-13 Administrative security setup for profile

4. The panel shown in Figure B-14 gives a summary of choices before we begin the profile creation process. Click **Create** to begin the process.

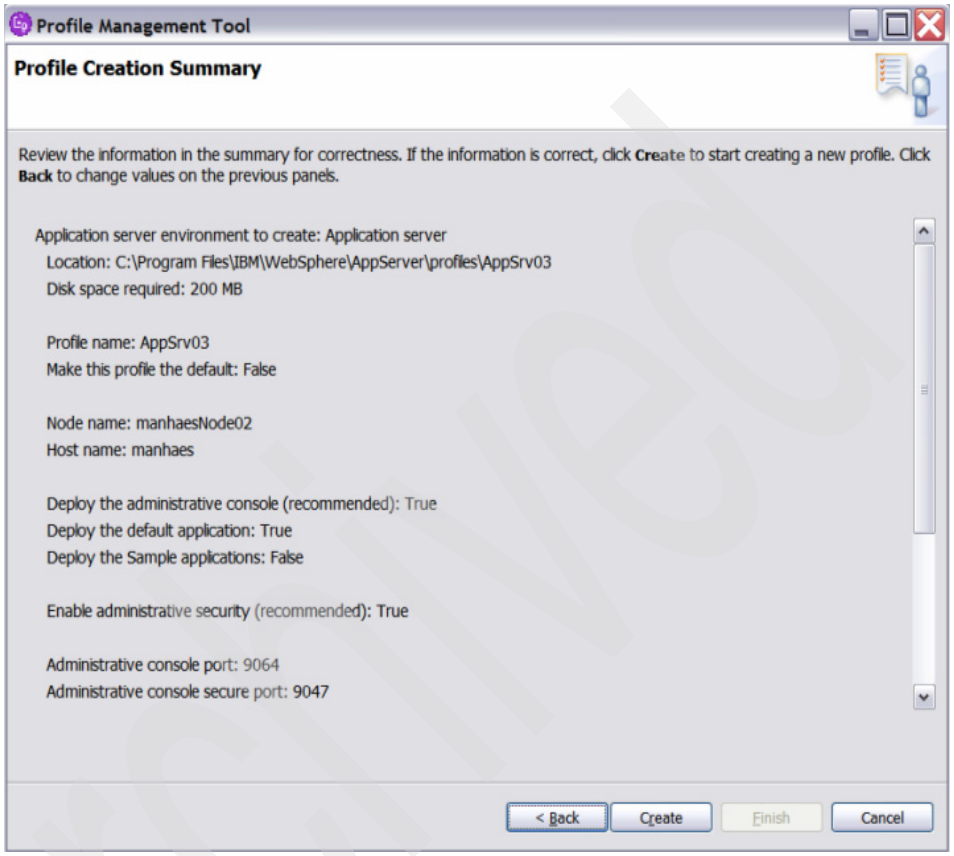


Figure B-14 Profile Creation Summary

Install verification

You can check if the EJB Feature Pack is installed in your environment using the following command:

```
<WAS_HOME>\bin\versionInfo.bat
```

In our environment, the command is in `C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv03\bin\versionInfo.bat`. See Figure B-15 for an example of the output.

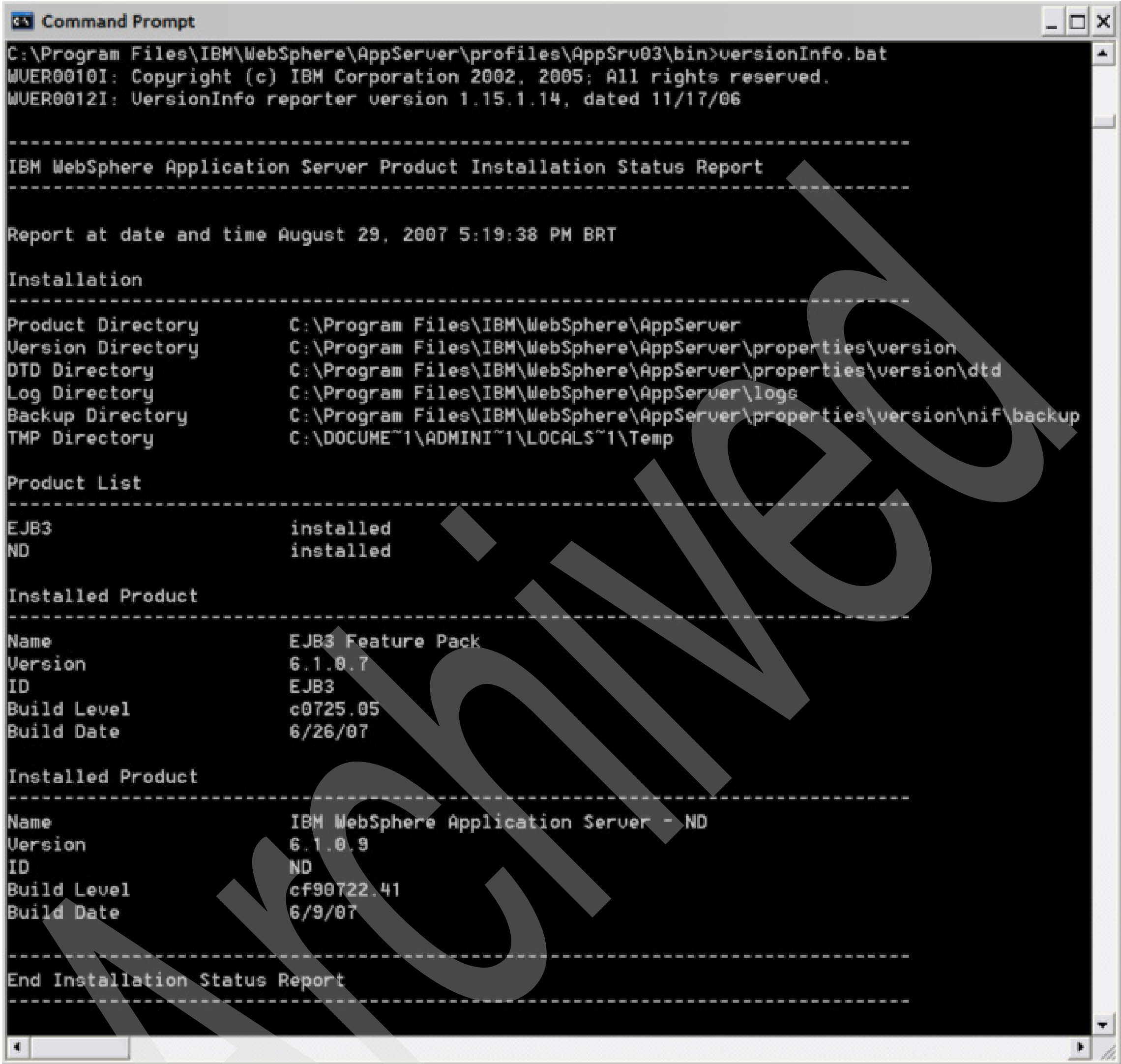


Figure B-15 VersionInfo command showing EJB 3.0 Feature Pack

You can see in this report that the EJB 3.0 Feature Pack is an installed product.

Application sample

In this section we install the application sample and test the application to validate EJB 3.0 Feature Pack installation.

Install application sample

The first step in this topic is to start the application server.

1. To start application server, go to `<WAS_HOME>\profiles\<PROFILE_NAME>\bin` and execute `startServer.sh server1` if not already started.
2. The next step is to open the console. For new users, this can be done using the firstSteps application that is on `<WAS_HOME>\profiles\firststeps\firststeps.bat`. See Figure B-16.

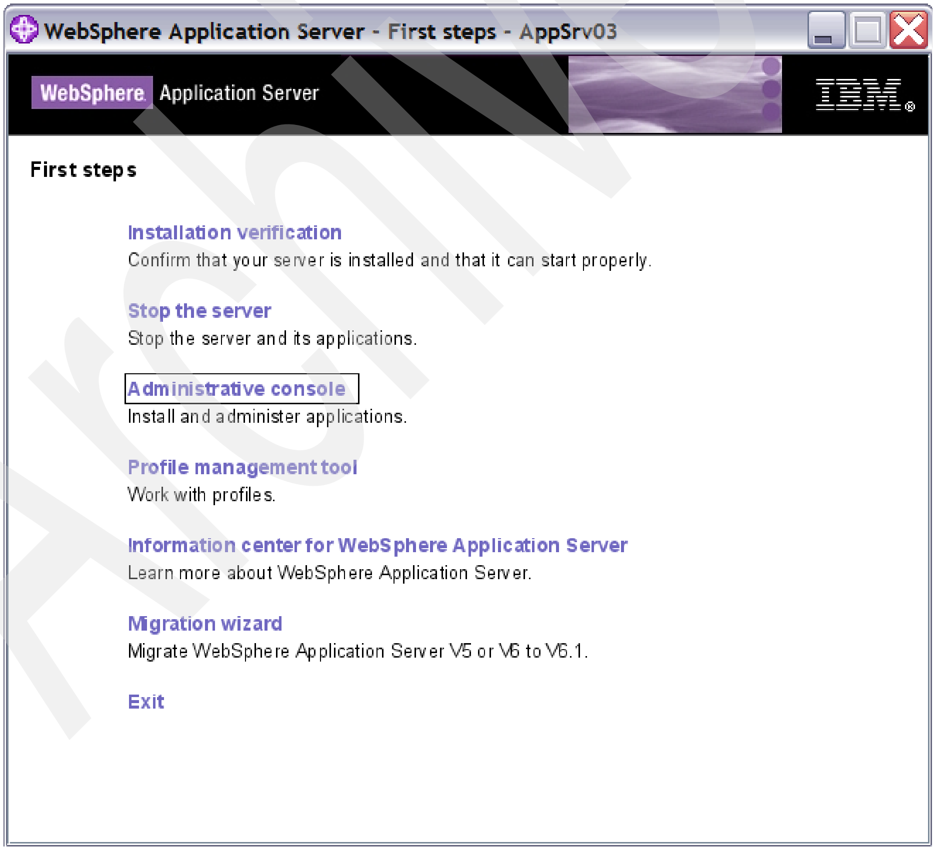


Figure B-16 First steps application

3. Selecting the Administrative console option shows the WebSphere console browser. If you configured security, you have to type user ID and password and click **Log in**. See Figure B-17.

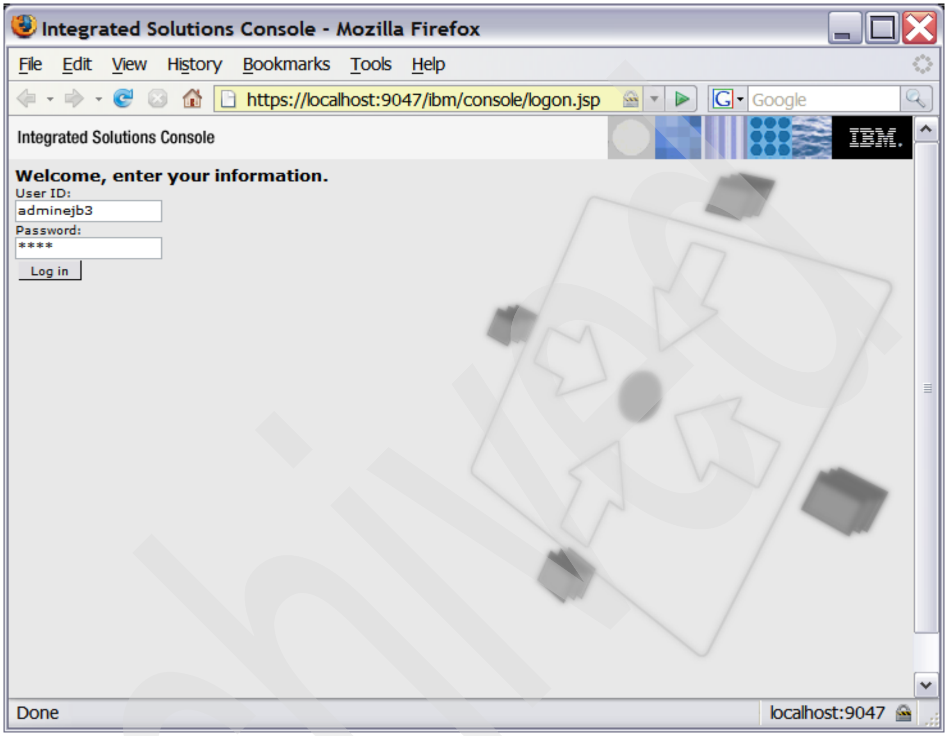


Figure B-17 WebSphere Application Server V6.1 console log in

4. On the console, display the **Enterprise Applications** panel by selecting **Applications** → **Enterprise Applications**.

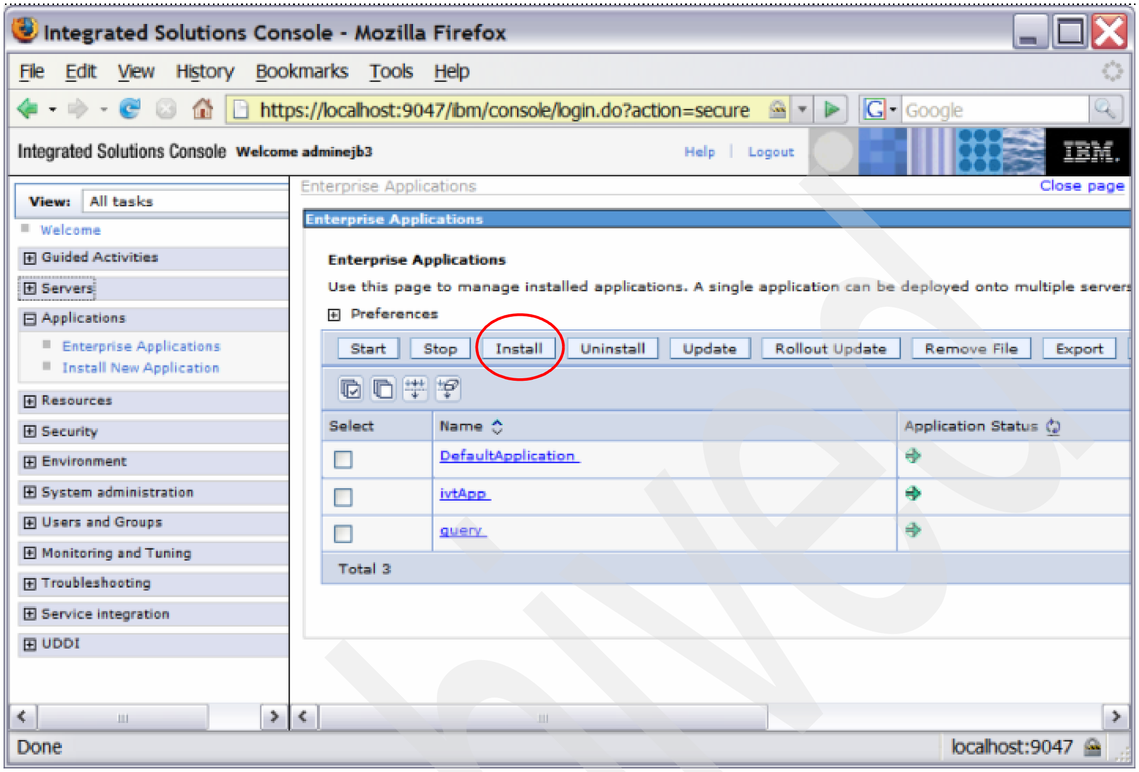


Figure B-18 Enterprise applications view

5. Click **Install** in the panel shown in Figure B-18.
6. Select the **Local file system** radio button under **Path to the new application**.
7. Select the **Browse** button to the right side of the **Full path** label. Browse your file system path to the `WAS_HOME/installableApps directory and select the EJB3CounterSample.ear` file. Following this, you do not have to provide any additional data to the application install process.
8. Select **Next** at the bottom of the current panel.
9. Again, select **Next** at the bottom of the current panel.
10. Again, select **Next** at the bottom of the current panel (for a total of three **Next** selections).
11. If you see the message **Application EJB3CounterSample installed successfully** then click the **Save** option to save your configuration changes back to the master configuration.
12. The panel shown in Figure B-19 is displayed.

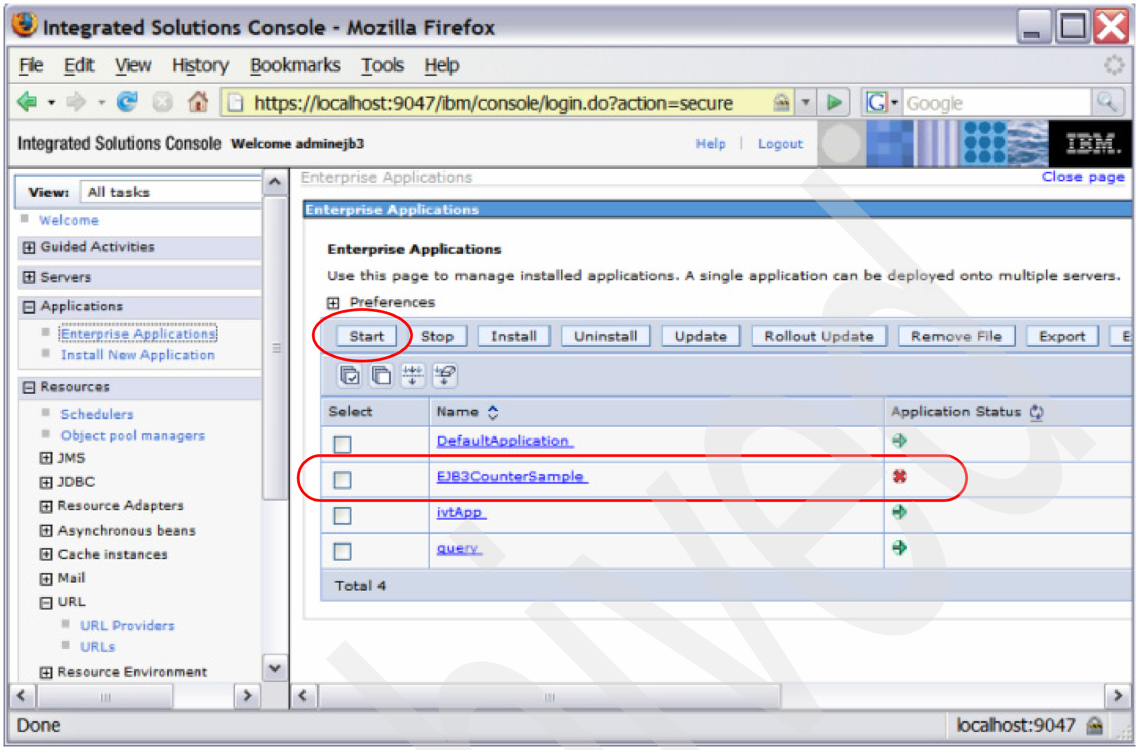


Figure B-19 EJB3CounterSample installed

13. To finish this step, select the check box just to the left of this application and select the **Start** button above the application.

Executing the sample

The sample can be executed either locally from the servlet or remotely using the client container. For our example, we focus on executing the sample using the servlet. For more details on client container execution, see the EJB 3.0 Feature Pack documentation.

Executing the sample from the servlet

The counter sample's web application is accessed by opening a web browser at URL `http://localhost:9080/ejb3sample/counter` on the system where you installed the Feature Pack.

Note: The port number in your configuration might different than 9080. However, 9080 is typical.

You should see the sample's Web page as follows.

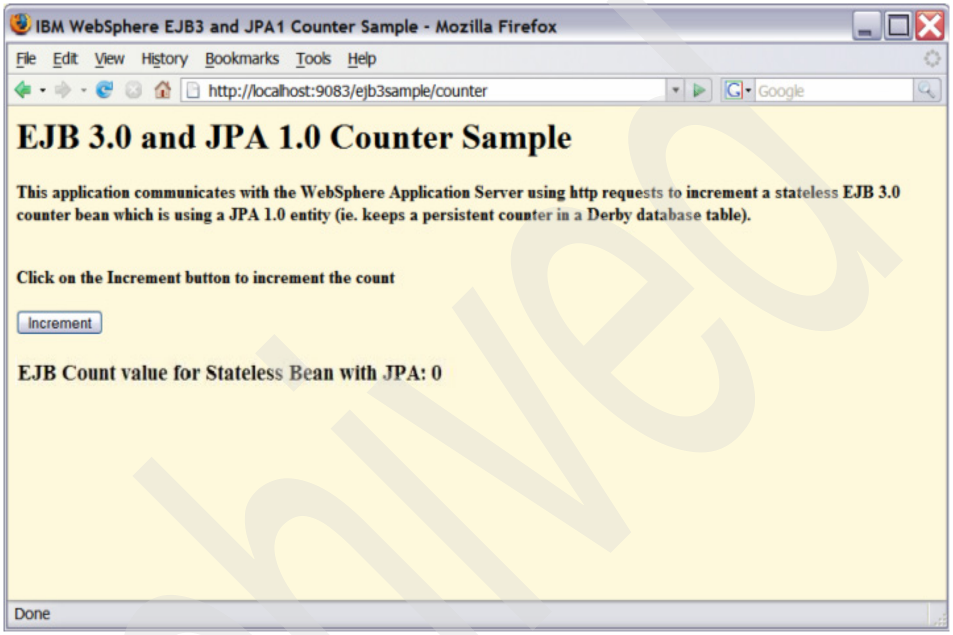


Figure B-20 EJB sample initial panel

Click the **Increment** button to cause the servlet to access the EJB. See Figure B-21.

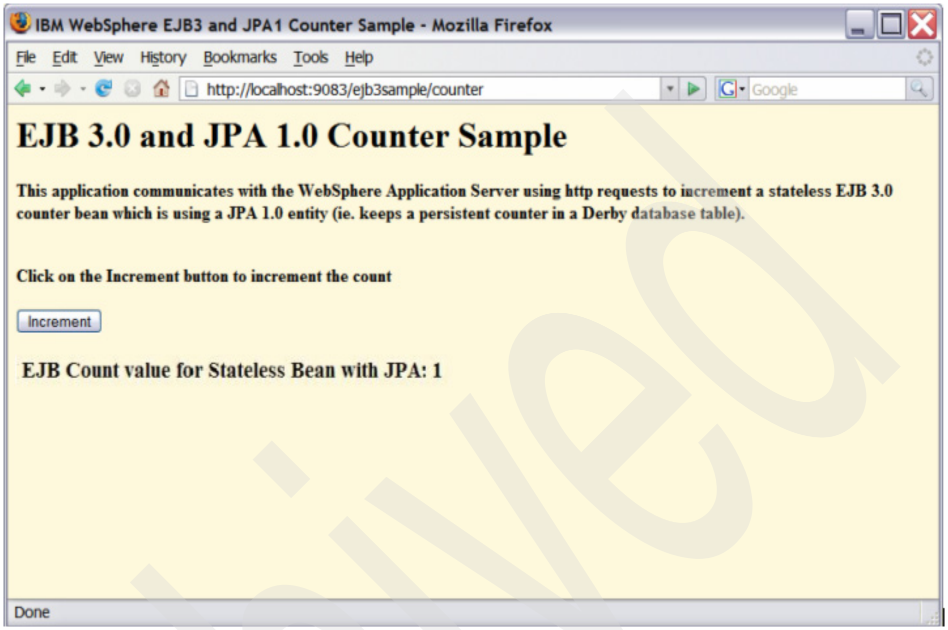
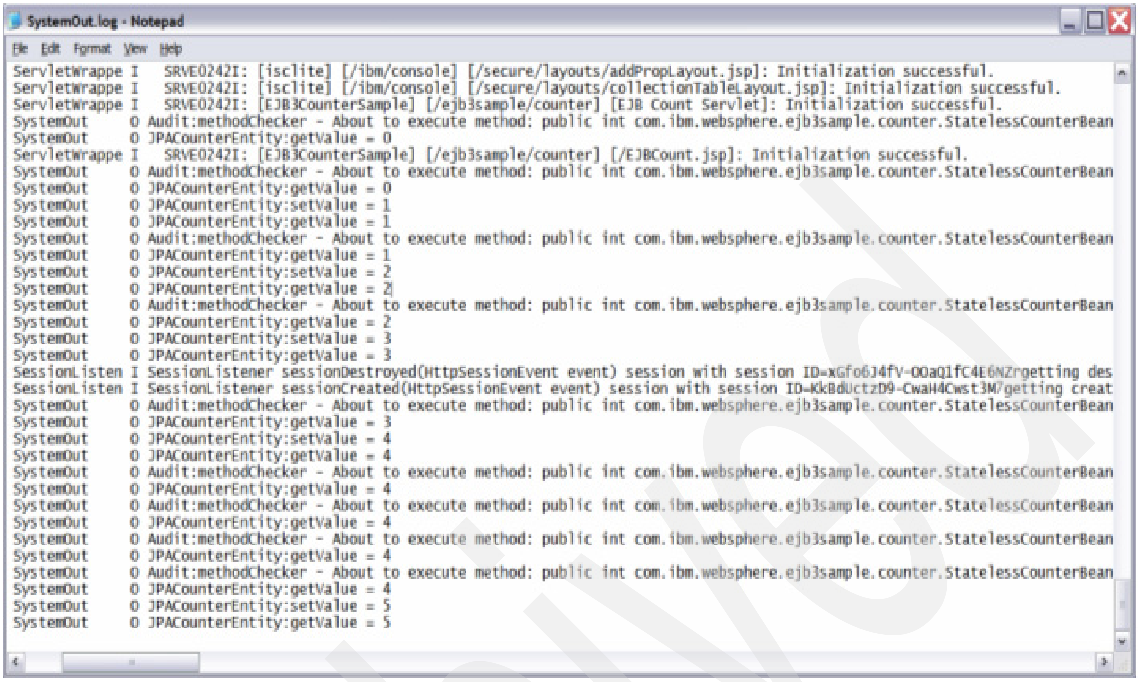


Figure B-21 Executing EJB Counter sample

To see what the sample application is doing within the WebSphere Application Server, navigate your file system to view the SystemOut.log file in the directory WAS_HOME/profiles/<your profile name>/logs/<your server name>/. See Figure B-22.




```
ServletWrappe I   SRVE0242I: [isclite] [/ibm/console] [/secure/layouts/addPropLayout.jsp]: Initialization successful.
ServletWrappe I   SRVE0242I: [isclite] [/ibm/console] [/secure/layouts/collectionTableLayout.jsp]: Initialization successful.
ServletWrappe I   SRVE0242I: [EJB3CounterSample] [/ejb3sample/counter] [EJB Count Servlet]: Initialization successful.
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 0
ServletWrappe I   SRVE0242I: [EJB3CounterSample] [/ejb3sample/counter] [/EJBCount.jsp]: Initialization successful.
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 0
SystemOut     O JPACounterEntity:setValue = 1
SystemOut     O JPACounterEntity:getValue = 1
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 1
SystemOut     O JPACounterEntity:setValue = 2
SystemOut     O JPACounterEntity:getValue = 2
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 2
SystemOut     O JPACounterEntity:setValue = 3
SystemOut     O JPACounterEntity:getValue = 3
SessionListen I SessionListener sessionDestroyed(HttpSessionEvent event) session with session ID=xGfo6J4fV-O0aQ1fC4E6NZrgetting des
SessionListen I SessionListener sessionCreated(HttpSessionEvent event) session with session ID=KkBdUctzD9-CwaH4Cwst3M7getting creat
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 3
SystemOut     O JPACounterEntity:setValue = 4
SystemOut     O JPACounterEntity:getValue = 4
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 4
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 4
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 4
SystemOut     O Audit:methodChecker - About to execute method: public int com.ibm.websphere.ejb3sample.counter.StatelessCounterBean
SystemOut     O JPACounterEntity:getValue = 4
SystemOut     O JPACounterEntity:setValue = 5
SystemOut     O JPACounterEntity:getValue = 5
```

Figure B-22 Analyzing application logs to view application actions

Configuring Eclipse for application sample development

These instructions assume that you have installed the WebSphere Application Server V6.1 Feature Pack for EJB 3.0 using the default location of `C:/Program Files/IBM/WebSphere/AppServer`. If you have used an alternate location, one small change is necessary in the `build.xml` file.

> **Note:** These instructions are using Eclipse V3.2 with a Java SE 5.0 compiler as the default. The Java SE 5.0 environment is a requirement for the EJB 3.0 and JPA programming models. Eclipse v3.3 was tested as well.

Setting up the workspace

The following steps document the requirements to set up the workspace:

1. Start Eclipse.

a. Open either a new or existing workspace. See Figure B-23.

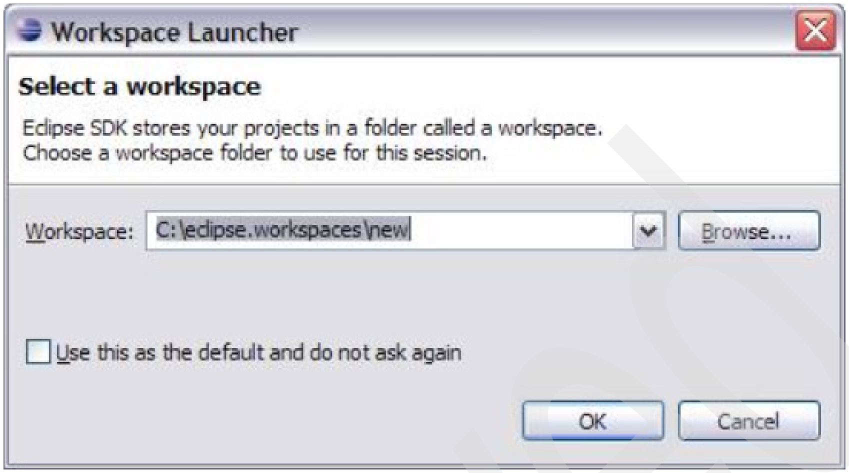


Figure B-23 Workspace launcher

b. Close the Welcome panel if it exists.

c. Switch to the Java Perspective as shown in Figure B-24.

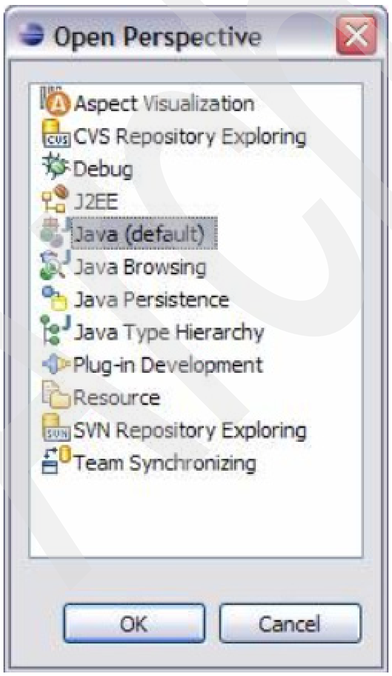


Figure B-24 Open perspective

2. Your Eclipse might be set up to use a Java 1.4 JDK and JRE™ by default. You have to update the preferences in order to compile Java 5 code. EJB 3.0 and JPA rely on Java 5 annotations and features:

   a. Select **Window** from the main menu, and then **Preferences.**

   b. Expand the **Java** section and select **Compiler**. Switch the Compiler compliance level to 5.0 as shown in Figure B-25.

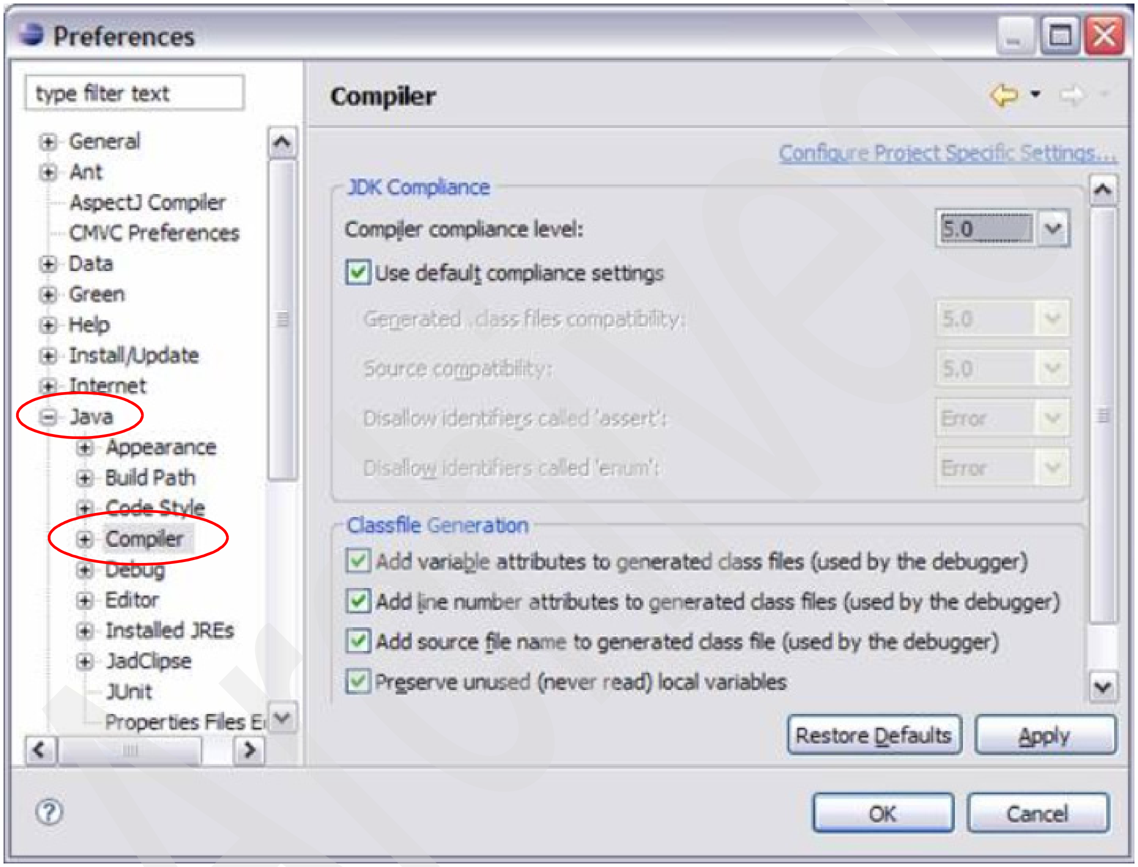


Figure B-25 Setting up compiler preferences

c. Select **Apply**. The following prompt in Figure B-26 might appear. If it does, just click **Yes** and continue.

Figure B-26 Compiler settings change confirmation

d. Next, switch to the Installed JREs and add the JRE that shipped with WebSphere Application Server V6.1 as shown in Figure B-27.

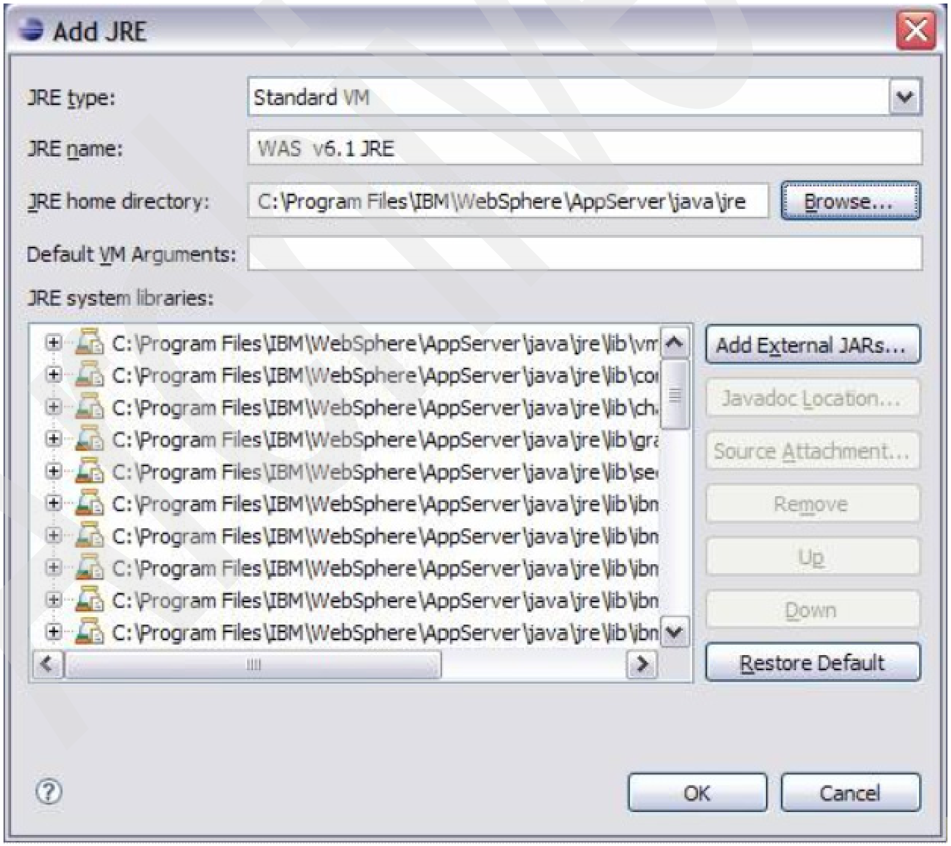


Figure B-27 Add JRE panel

e. Select **OK** in Figure B-28.

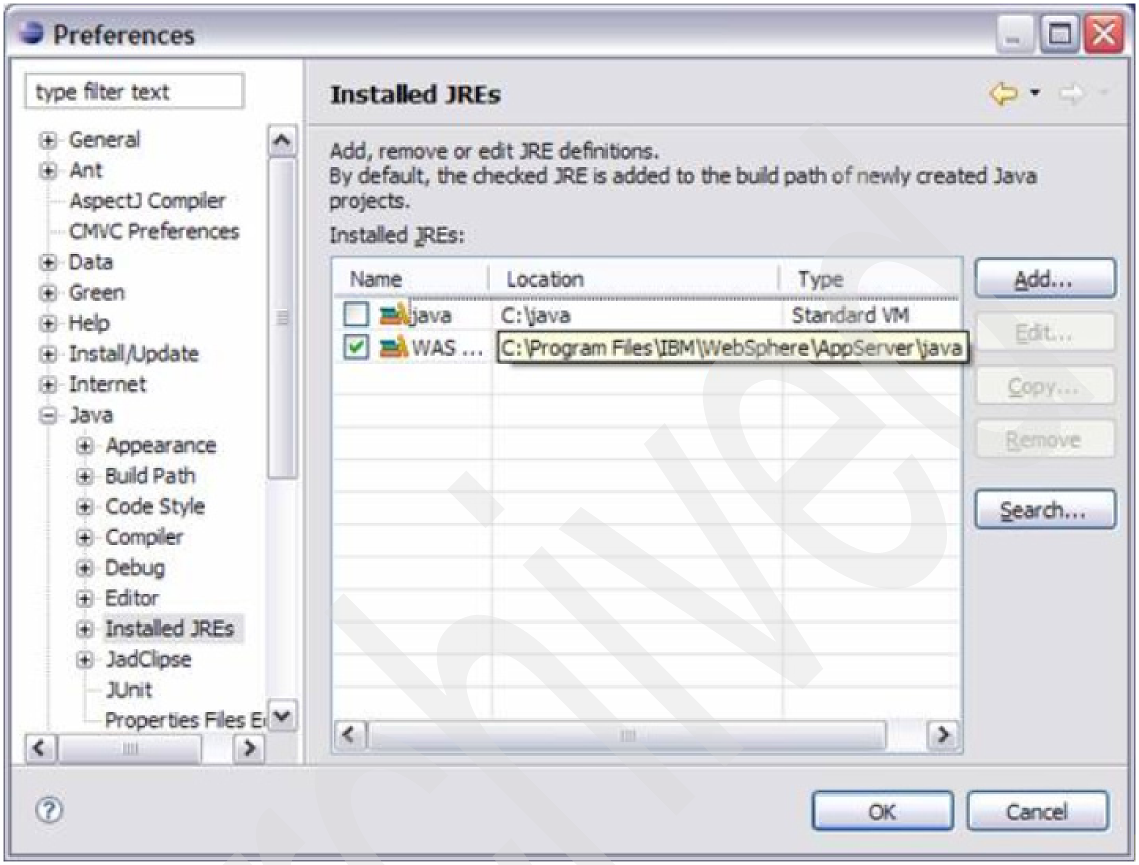


Figure B-28 Result of install was 6.1 JRE

f. Select **OK** to save the changes and select **Yes** to do a build. The prompt shown in Figure B-29 might appear. If so, select **Yes** and continue.

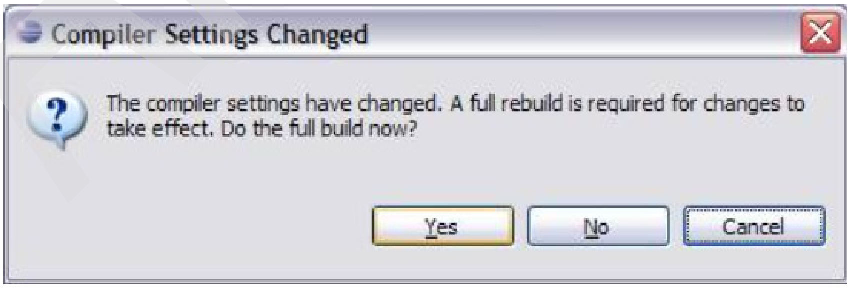


Figure B-29 Compiler settings confirmation changes

Creating a java project

In this section we walk through creating the sample Java Project.

1. First you have to create the Java Project via **File** → **New** → **Project**. Select **Java Project** and click **Next**. See Figure B-30.

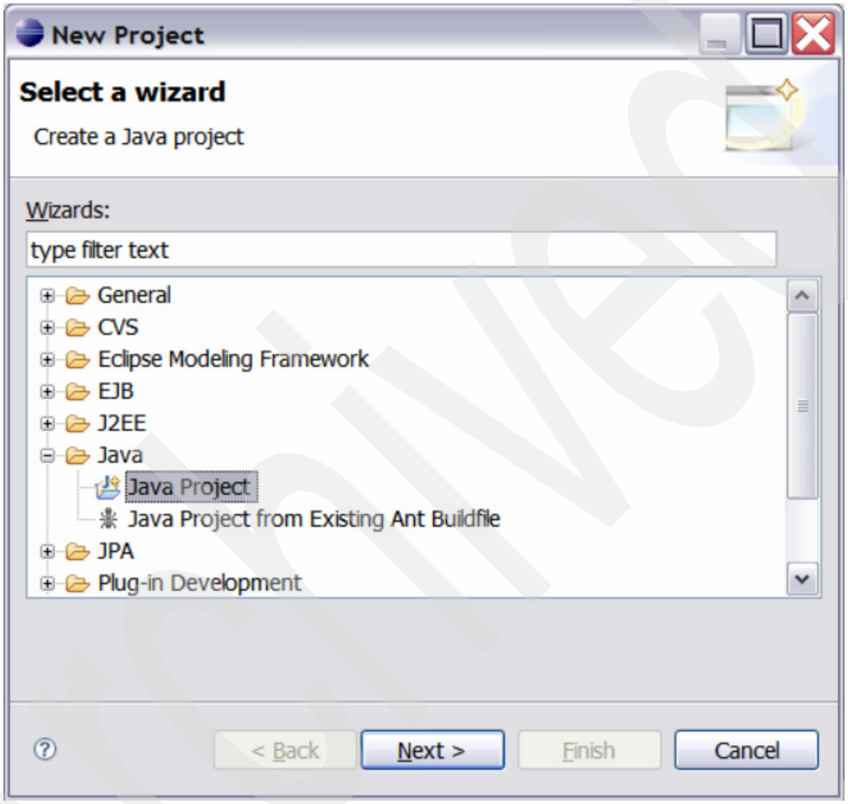


Figure B-30 Creating a new project

2. In Figure B-31, fill in the name for your project. Depending on the defaults for your Eclipse environment, you might also want to select the options as outlined below. Click **Next**.

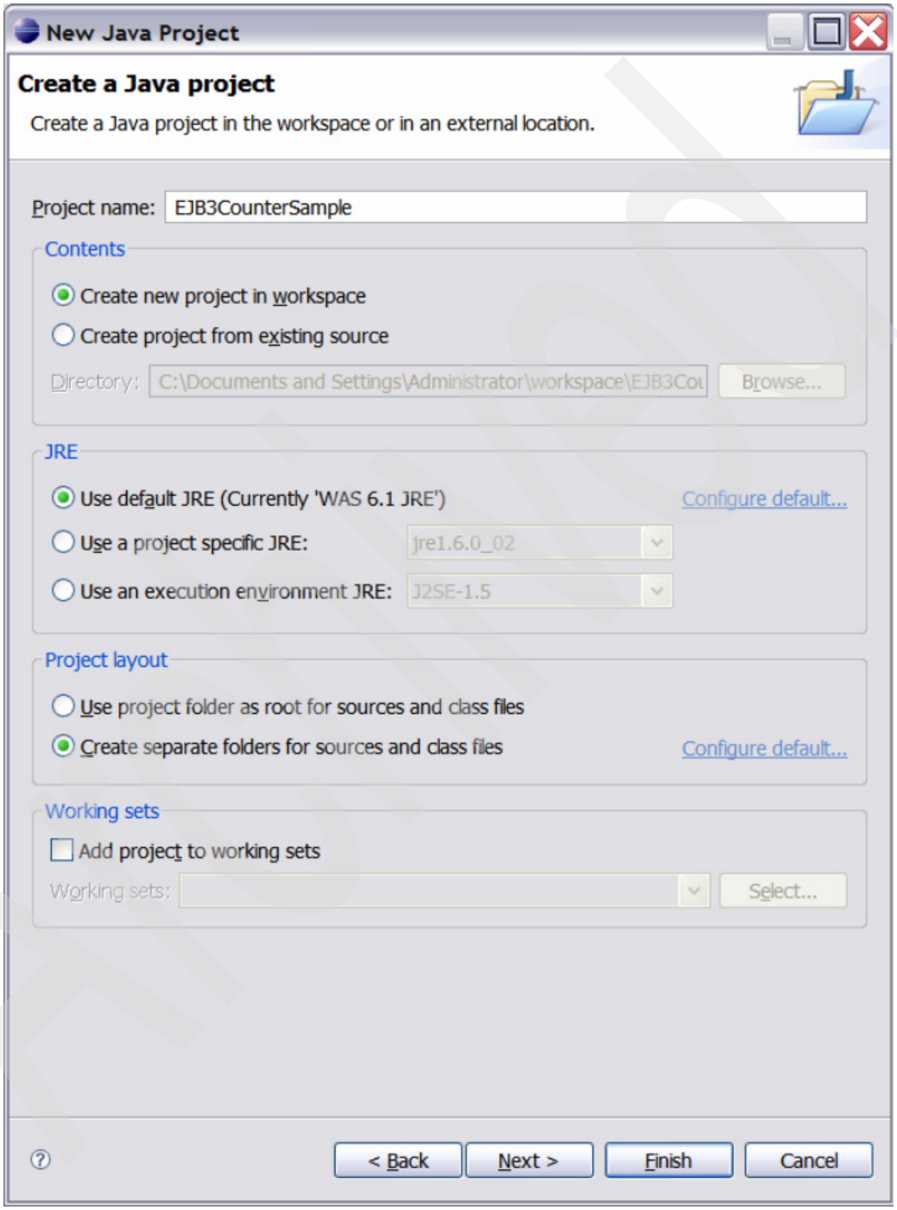


Figure B-31 Creating a java project

3. The panel shown in Figure B-32 shows the default values for the project. After reviewing these choices, click **Finish**. You have to adjust some of these project settings after you import the sample source. Your Eclipse project has been created.

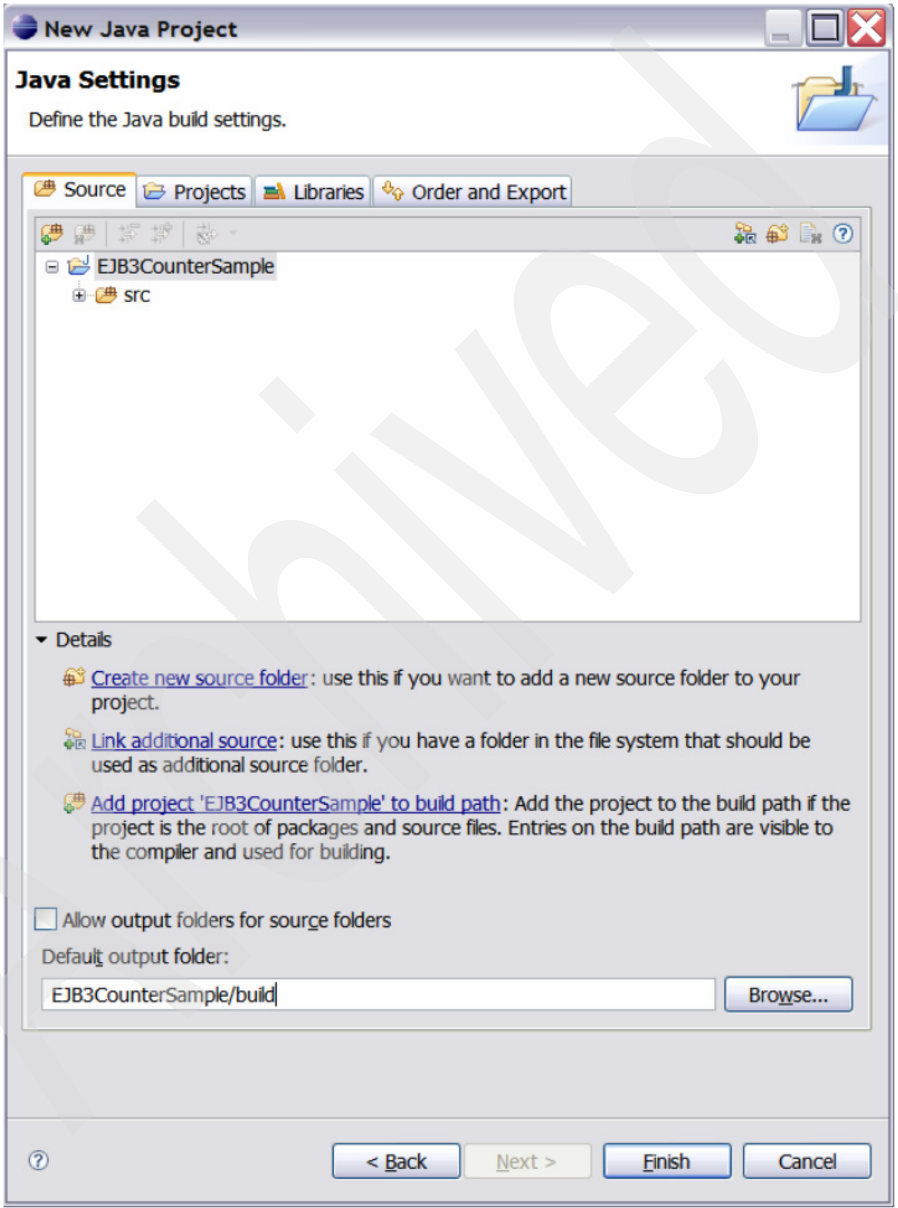


Figure B-32 Java settings for new java project

Adding Java EE Runtime Jar files to your project

In order to compile cleanly within Eclipse, the Java EE runtime jars from the Feature Pack installation have to be included as external jars for your Eclipse project.

1. Open your Project's **Properties** → **Java Build Path** and open the **Libraries** tab as shown in Figure B-33.

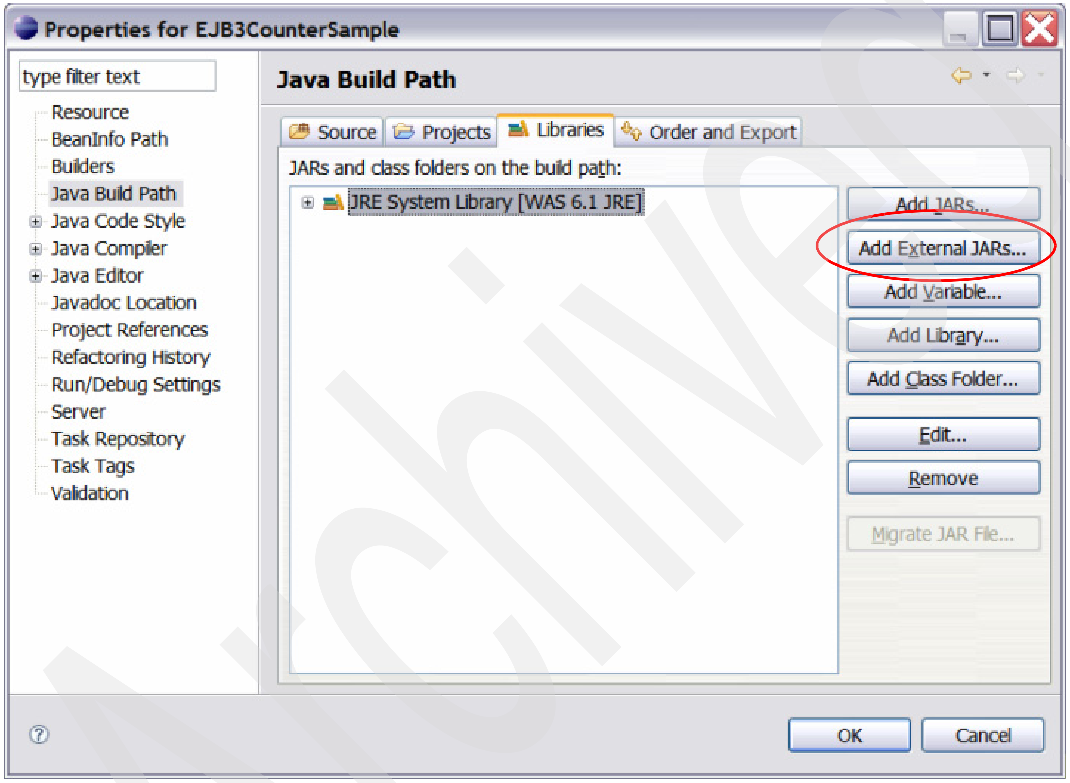


Figure B-33 EJBCounterSample library settings

2. Click **Add External JARs…** and navigate to the plugins directory within the install location for the WebSphere Application Server. In our example, it is within `C:\Program Files\IBM\WebSphere\AppServer\plugins`. Here, you have to select the `com.ibm.ws.jpa_7.0.0.jar` and `com.ibm.ws.runtime_6.1.0.jar` files. These files give you access to the JPA and EJB 3.0 runtime classes necessary for building your projects. See Figure B-34.

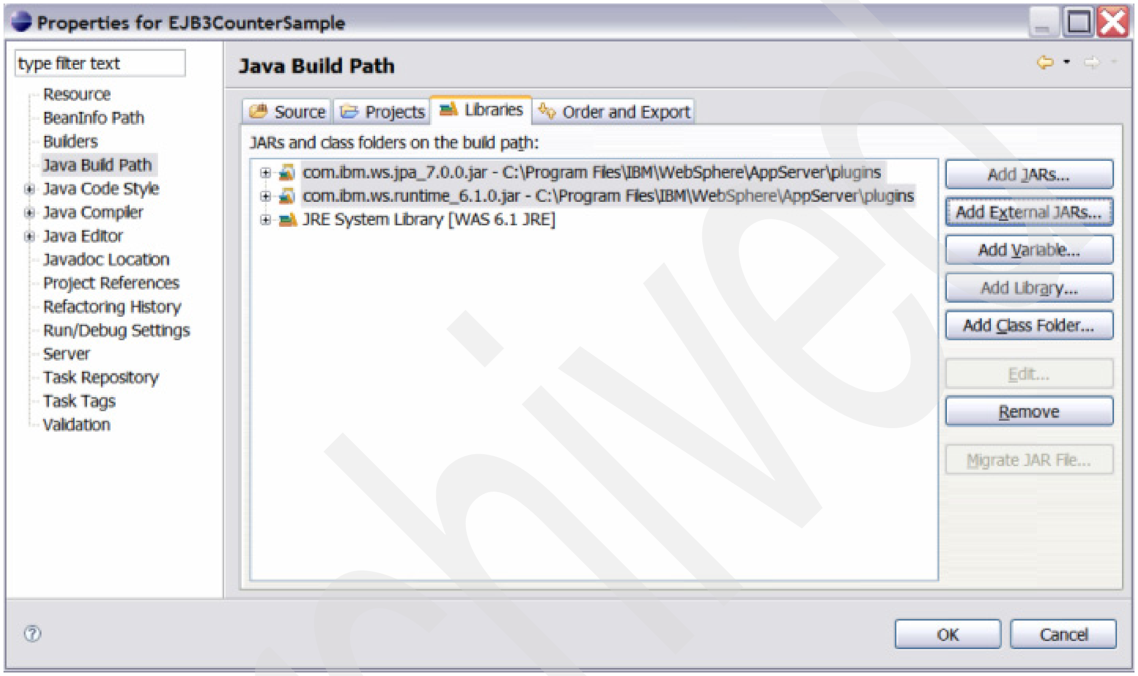


Figure B-34 EJB3CounterSample after addiction of JPA libraries

3. You also have to include the `j2ee.jar` from the lib directory; in our example, `C:\Program Files\IBM\WebSphere\AppServer\lib`.

Importing the sample source

Now that you have the workspace and project properly configured, you can import the source from the EJB3CounterSample.

1. Highlight your EJB3CounterSample project within Eclipse, right mouse click, and select **Import…** On the first Import panel, select **File System** and click **Next.** See Figure B-35.

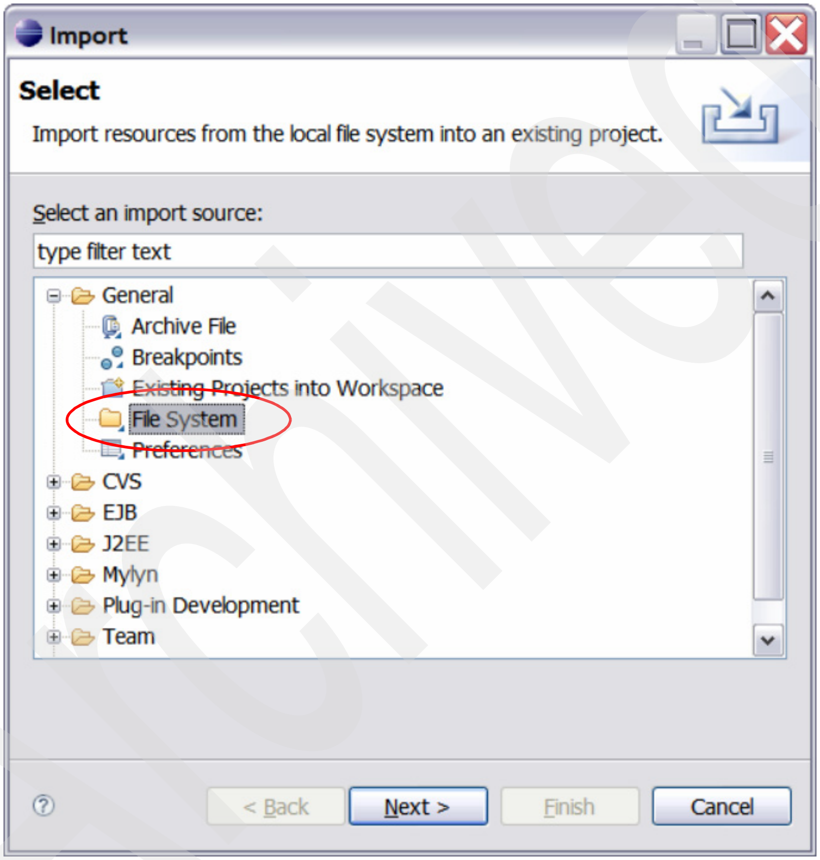


Figure B-35 Import example files

2. On the next panel (shown in Figure B-36), use the Browse function to find the samples subdirectory within your WebSphere Application Server installation. Only select the src directory, because the lib directory contains binaries that are not required for the Eclipse import. Click **Finish** to complete the import process.

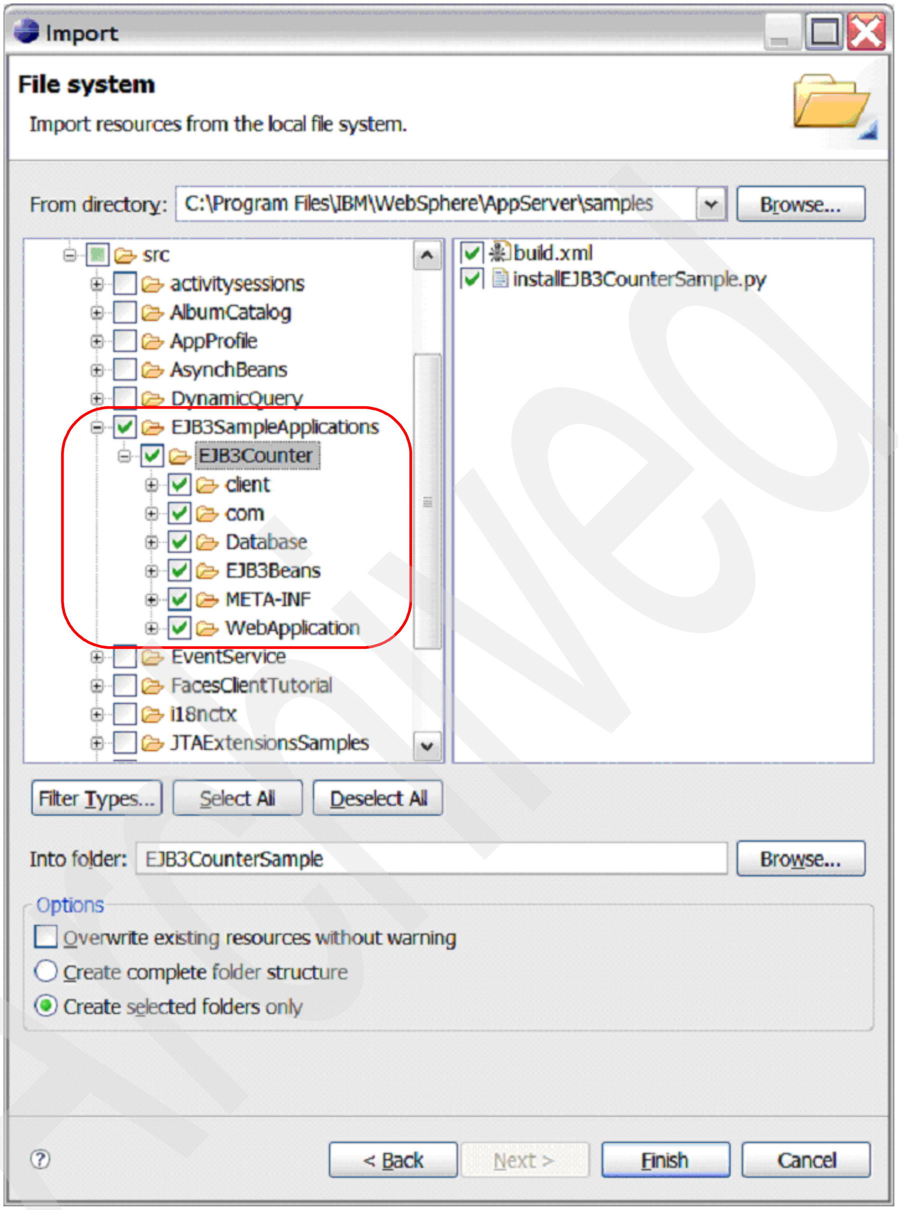


Figure B-36 Import process

3. After the import, you might notice some red x marks indicating build failures as shown in Figure B-37.

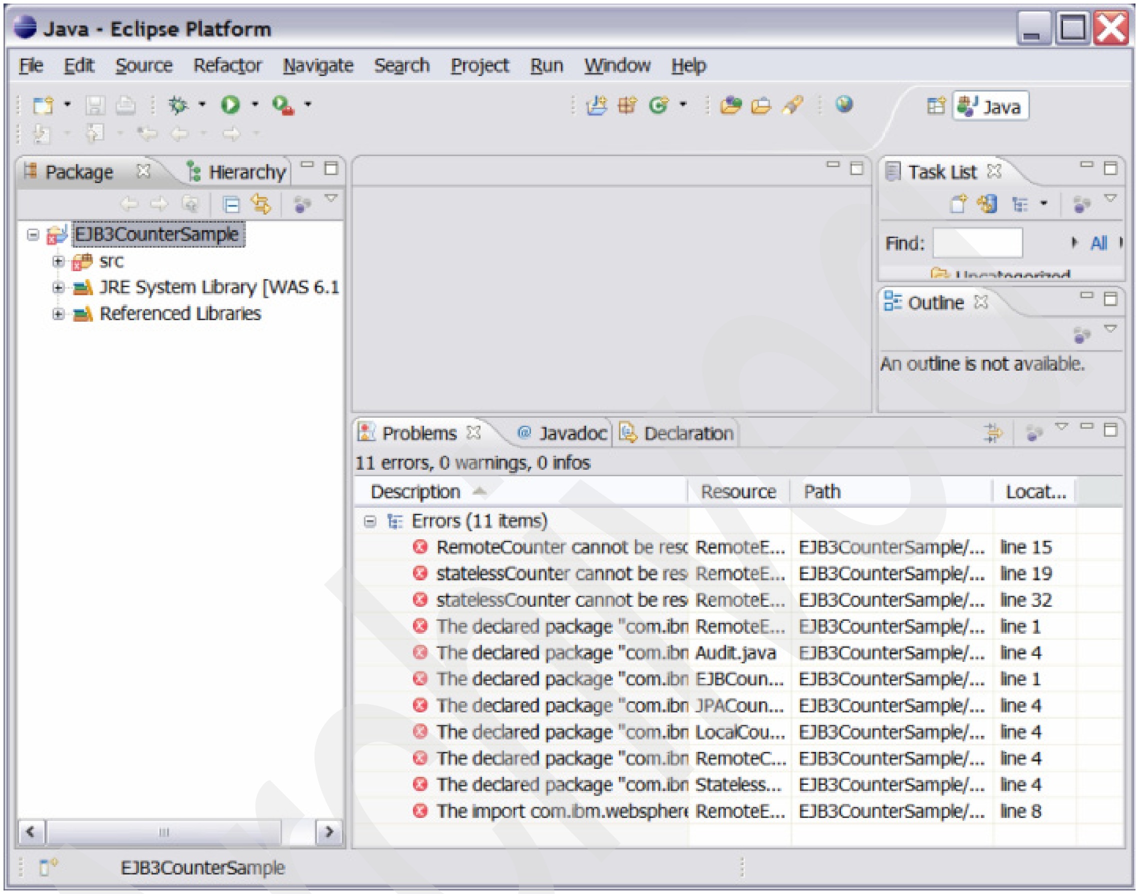


Figure B-37 Building failures

4. We have to configure the project's source directories. Once again, open the **Properties** panel for your EJB3CounterSample project and select the **Java Build Path**. This time, open the **Source** tab. See Figure B-38.

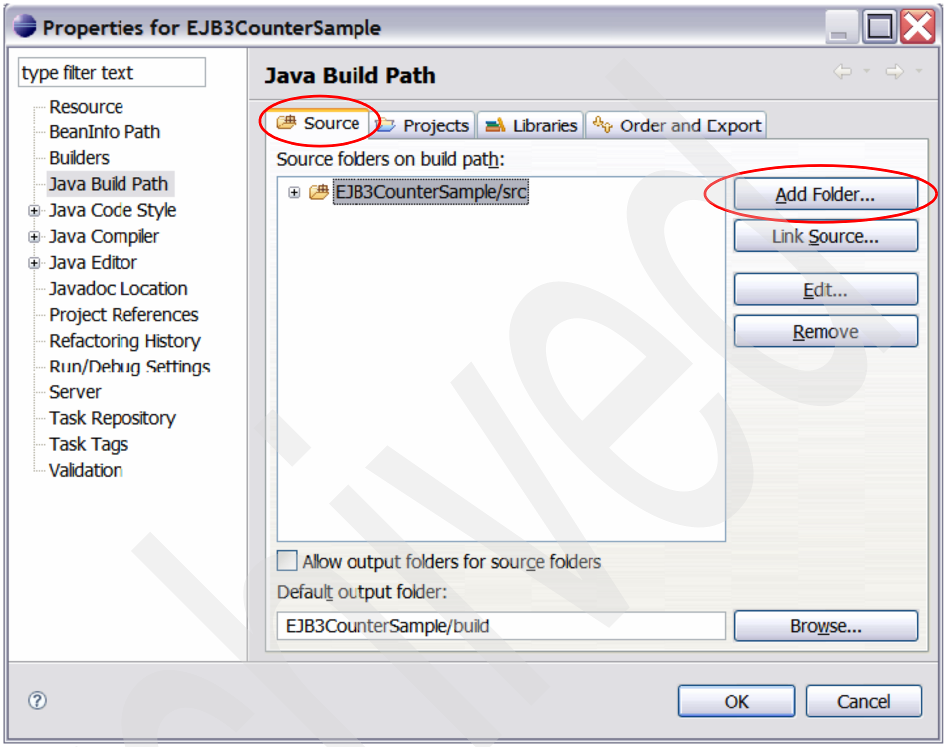


Figure B-38 Configuring project source

5. Click **Add Folder…** Expand the src folder and select the EJB3Counter directory and click **OK**. See Figure B-39.

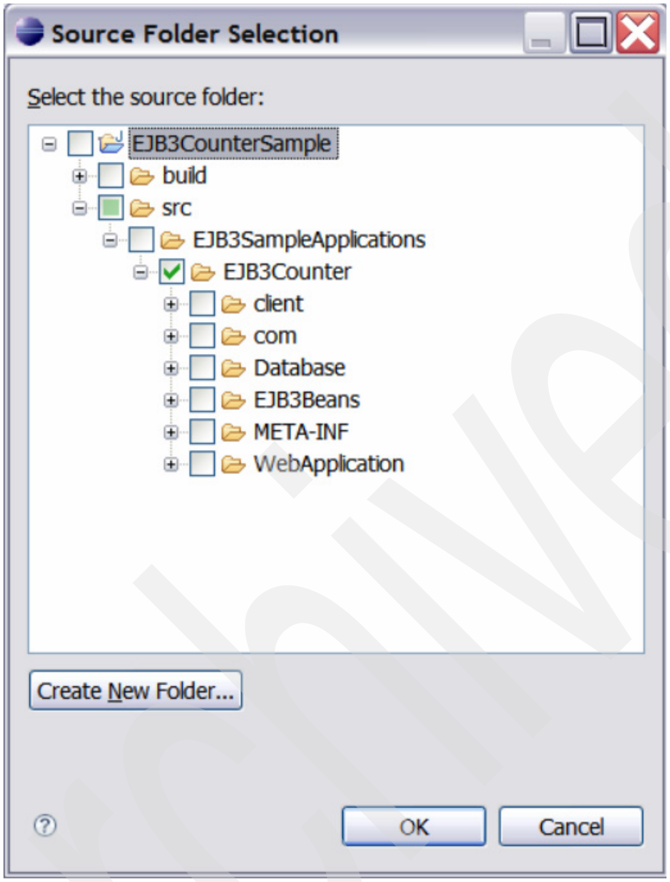


Figure B-39 Correcting the source file directory

6. You see an error about nesting source folders. Since you do not have any source files in the src directory, you have to remove this directory from the source folders. See Figure B-40.

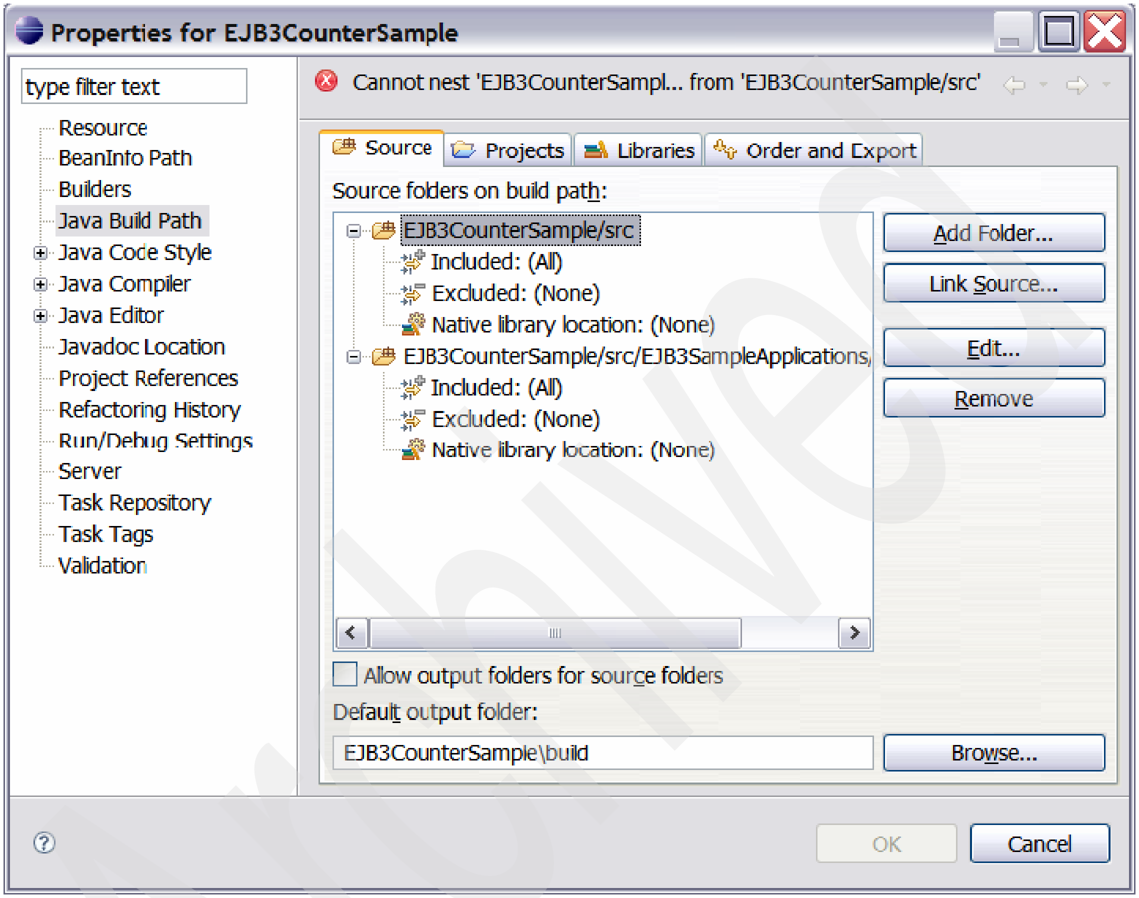


Figure B-40 Nested source errors in EJB3CounterSample

7. When you click **OK** after removing src directory on the panel shown in Figure B-40, your project should automatically re-build and the red x marks should disappear. See Figure B-41.

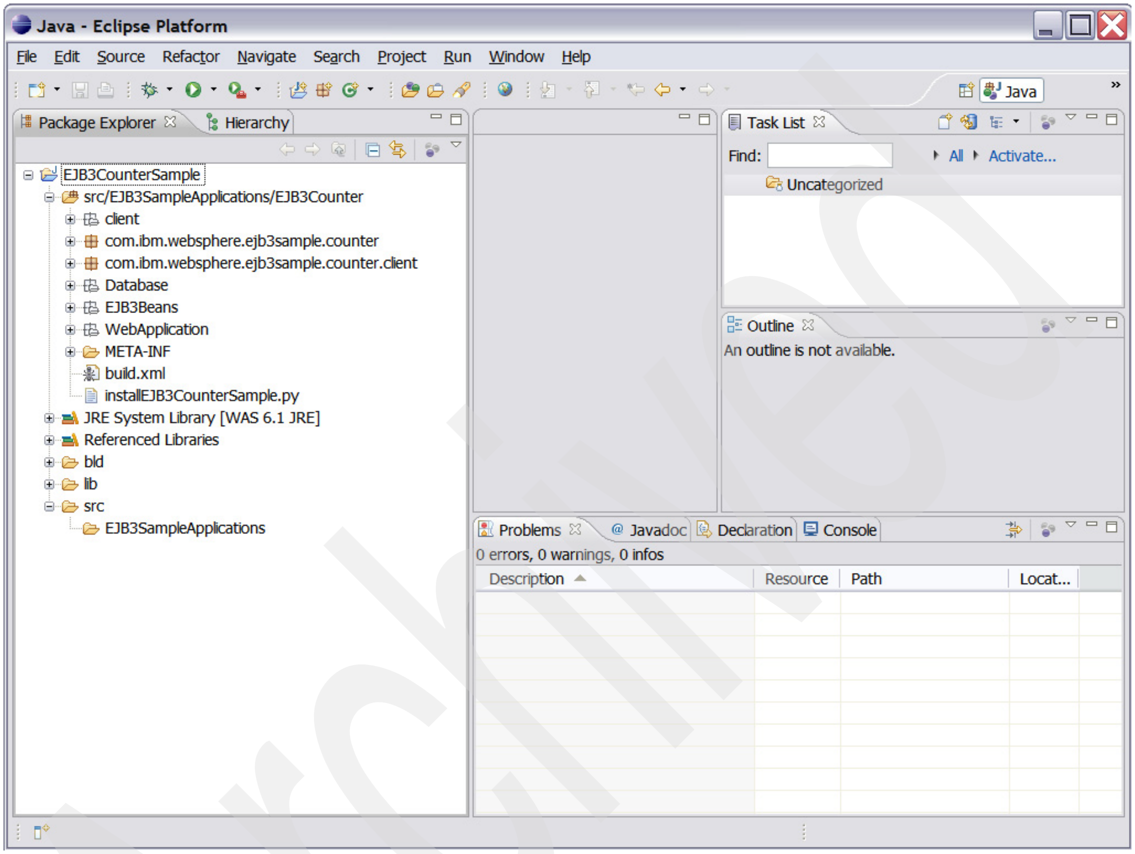


Figure B-41 Result of correct build sources

> **Note:** If some of the error codes do not disappear, rename only the package **client.** Rename **com.ibm.websphere.ejb3sample.counter.client** to **com.ibm.websphere.ejb3sample.counter.client**

Building the EJB3CounterSample Application

With the above configuration, basic compiling and building can be done via Eclipse. But, the packaging of the application into an enterprise application archive (ear) requires the use of an ant build.xml script. See the code snippet of build.xml in Example B-1.

Example: B-1 Build.xml code snippet

```
<?xml version="1.0"?>
<!--
  "This program may be used, executed, copied, modified and distributed 
without
  royalty for the purpose of developing, using, marketing, or 
distributing."
-->

<project name="EJB3" default="all" basedir=".">
<echo message="basedir ${basedir}"/>
  <!--
  Need to update was.home to point at your WebSphere install location.
  Default value is "c:/Program Files/IBM/WebSphere/AppServer"
  -->
  <property name="was.home" location="c:/Program 
Files/IBM/WebSphere/AppServer"/>
```

If you have installed WebSphere Application Server in a different directory from the default (`c:/Program Files/IBM/WebSphere/AppServer`), you have to update the was.home property to the appropriate value.

After the changes, it is necessary to kick off the build to create the ear file. Highlight the build.xml, right mouse click, select **Run As..** → **Ant Build**. Your Console window should look similar to Figure B-42.

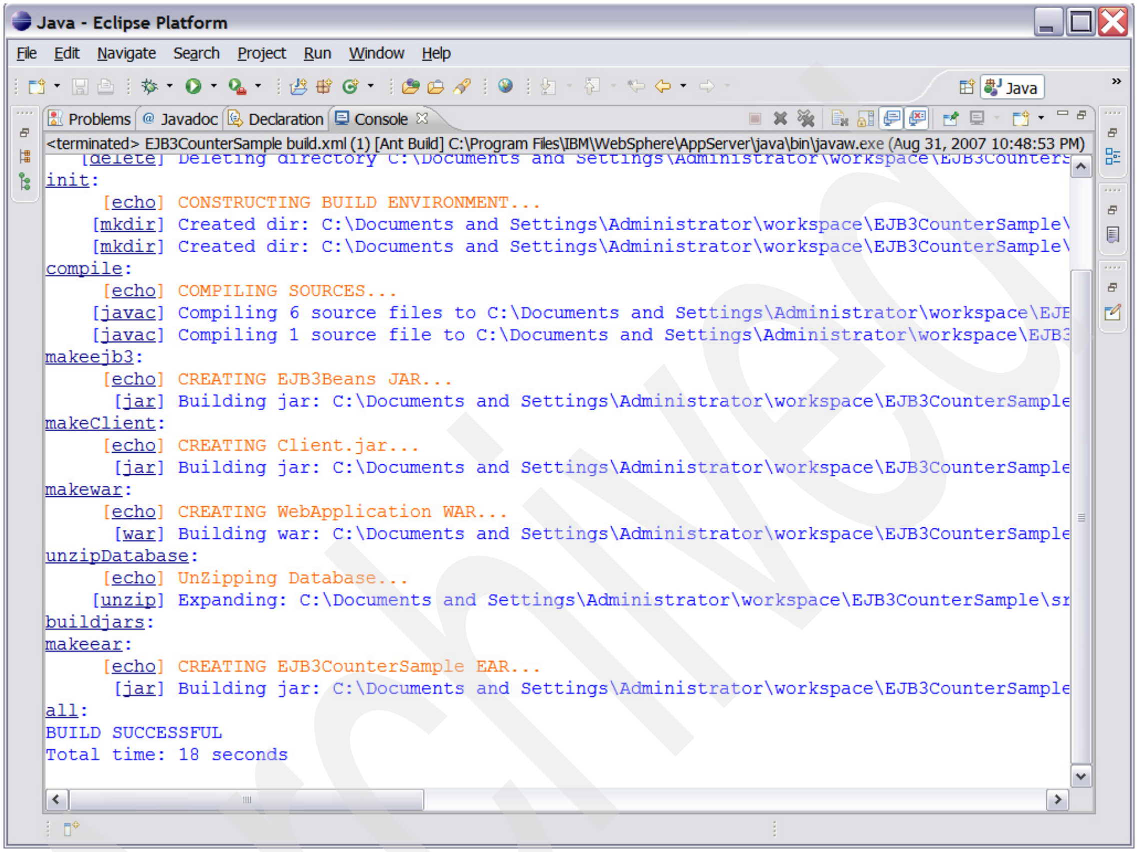


Figure B-42 Results of build.xml execution

Go back to your Package Explorer (Figure B-41 on page 628), highlight the EJB3CounterSample, and select Refresh. We now notice that the EJB3Beans.jar, WebApplication.war, and EJB3CounterSample.ear files have been created in the lib directory. See Figure B-43.

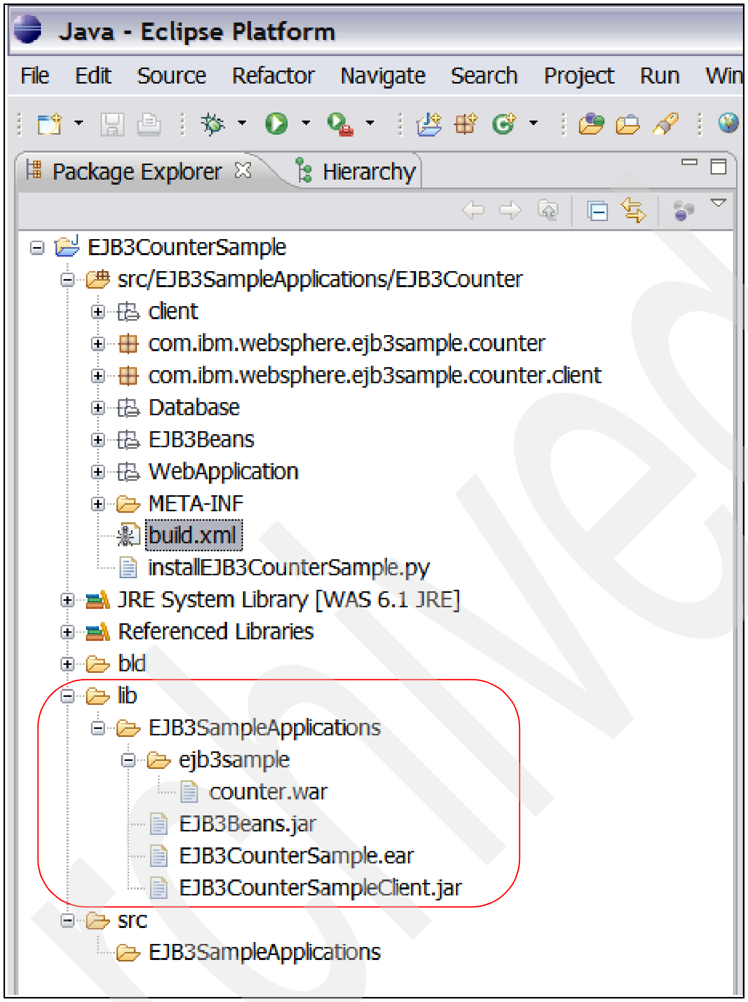


Figure B-43 Files generated after build.xml running with Ant

The EJB3CounterSample.ear file can now be installed into your WebSphere Application Server environment.

Known limitations with sample

The EJBCounterSample application is known to have installation and configuration problems with WAS ND due to the embedded database being located in a different location on the filesystem.

The Derby database, which is used in Embedded mode, does not permit removal of the database files with the WebSphere Application Server process is active. If you uninstall and reinstall the sample application using the admin console support (that is, while the server process is started) you see a warning message in the SystemOut.log, concerning Derby database files that are locked and cannot be deleted. Then, when you reactivate the sample, the counter value in the Derby database table is still at the value left over from any previous execution of the sample.

Understanding some parts of the sample code

In this section we explain some parts of the EJB3Counter code sample.

JPACounterEntity

This class represents the EntityBean of code. See Figure B-44.

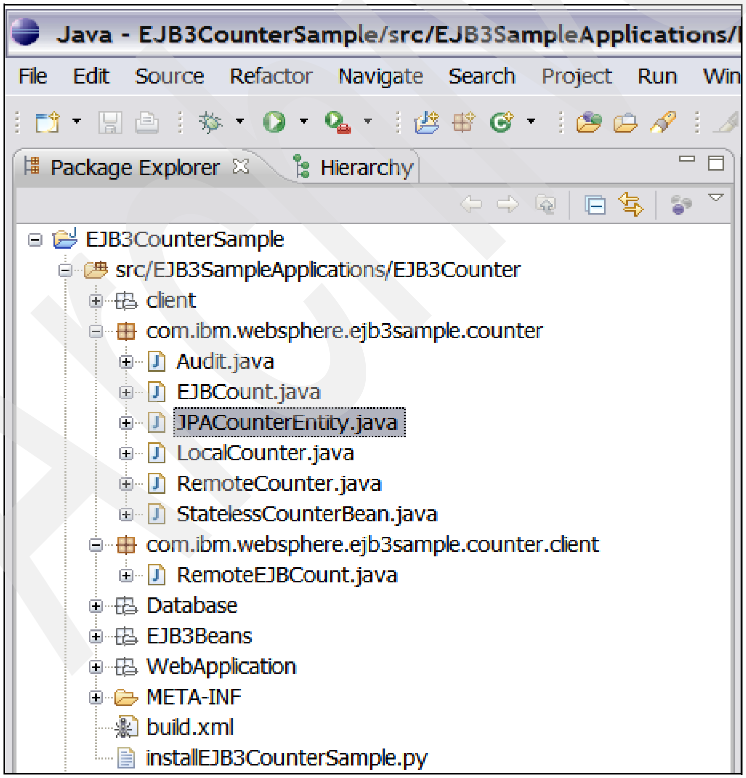


Figure B-44 JPACounterEntity

Consider the JPACounterEntity code in Example B-2.

Example: B-2 JPAConterEntity

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or
distributing.

package com.ibm.websphere.ejb3sample.counter;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="EJB3COUNTERTABLE")

public class JPACounterEntity {

    @Id
    private String primarykey = "PRIMARYKEY";

    private int value = 0;

    public void setValue( int newValue )
    {
        System.out.println ("JPACounterEntity:setValue = " + newValue);
        value = newValue;
    }

    public int getValue()
    {
        System.out.println ("JPACounterEntity:getValue = " + value);
        return value;
    }

    public void setPrimaryKey( String newKey )
    {
        System.out.println ("JPACounterEntity:setPrimaryKey = '" +
newKey + "'");
        primarykey = newKey;
    }

    public String getPrimaryKey()
    {
```

```
        System.out.println ("JPACounterEntity:getPrimaryKey = '" +
primarykey + "'");
        return primarykey;
    }
}
```

In the foregoing example, before the class declaration, we have the annotation @Entity that describes class as an Entity, and the table name that represents the O/R mapping to the table EJB3COUNTERTABLE. The @id annotation assigns a value of PRIMARYKEY to the attribute private String primarykey. The attribute called value is persisted by default and no special annotation is required. Also note the sets and gets for each Entity field. Now note the persistence.xml located in EJB3Beans\META-INF. See Figure B-45.

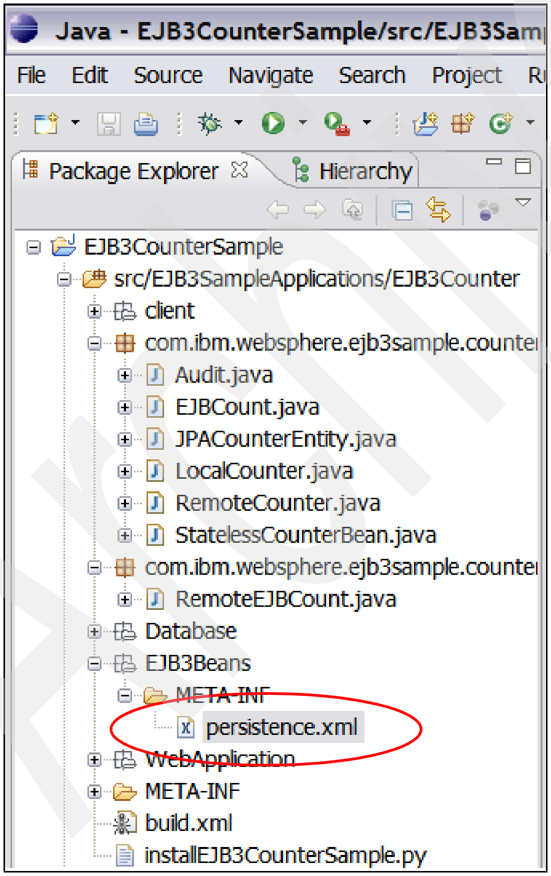


Figure B-45 Persistence.xml location

Study the persistence.xml code in Example B-3.

Example: B-3 Persistence.xml code

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.0"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
        <persistence-unit name="Counter">
<jta-data-source>jdbc/EJB3SampleDatasource</jta-data-source>
<class>com.ibm.websphere.ejb3sample.counter.JPACounterEntity</class>
<exclude-unlisted-classes>true</exclude-unlisted-classes>
        </persistence-unit>
</persistence>
```

You can see that JPACounterEntity is mapped to a data source called jdbc/EJB3SampleDatasource. This configuration is required for JPA run. The configuration of data source alias is mapped to resouces.xml that is deployed in EAR file of application. You can find resouce.xml in the project. See Figure B-46.

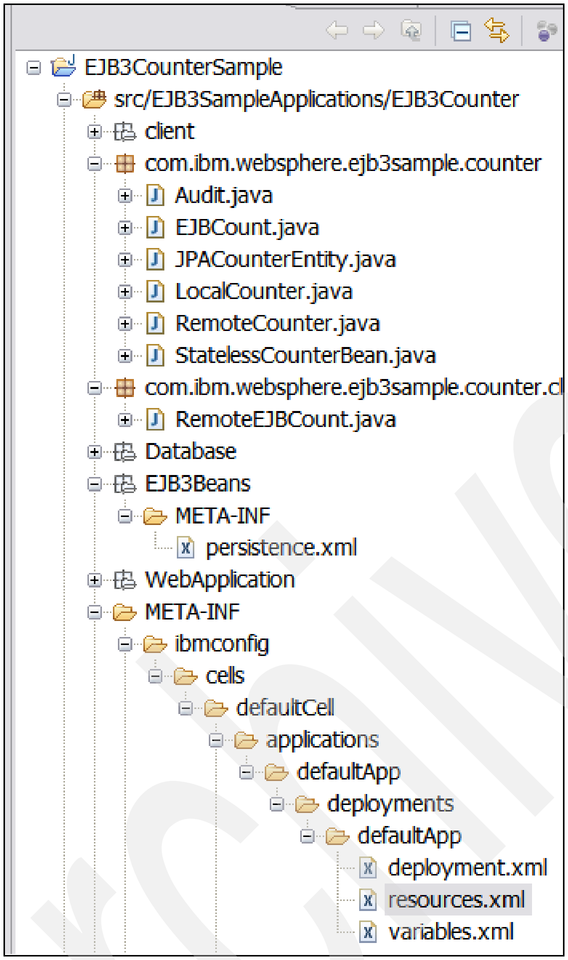


Figure B-46 resouces.xml location in java project

See the code snippet of resources.xml in Example B-4.

Example: B-4 resource.xml code snippet

```
<?xml version="1.0" encoding="UTF-8"?>
....
<resources.jdbc:JDBCProvider xmi:id="builtin_jdbcprovider" name="Derby
JDBC Provider (XA)" description="Built-in Derby JDBC Provider (XA)"
providerType="Derby JDBC Provider (XA)"
implementationClassName="org.apache.derby.jdbc.EmbeddedXADataSource"
xa="true">
    <classpath>${DERBY_JDBC_DRIVER_PATH}/derby.jar</classpath>
```

```
    <factories xmi:type="resources.jdbc:DataSource"
xmi:id="DataSource_1163624113079" name="EJB3SampleDatasource"
jndiName="jdbc/EJB3SampleDatasource" description="Derby datasource for
EJB3 sample application" category=""
relationalResourceAdapter="builtin_rra" statementCacheSize="10"
datasourceHelperClassname="com.ibm.websphere.rsadapter.DerbyDataStoreHe
lper">
      <propertySet xmi:id="J2EEResourcePropertySet_1163624113079">
       <resourceProperties xmi:id="J2EEResourceProperty_1163624113086"
name="databaseName" type="java.lang.String"
value="${APP_INSTALL_ROOT}/${CELL}/EJB3CounterSample.ear/database/EJB3S
ampleDB" description="adminRequired=true - This is a required property.
This property must be set and it identifies which database to access.
For example, g:/db/wombat." required="true"/>
```

Note in the snippet that the data source alias is mapped to Derby Embedded database, which is distributed in the EAR file of the application.

Stateless Counter Bean

In this example, we use a session bean façade to reach the EntityBean JPACounterEntity. The application has to be reached using servlets and a client in a remote JVM using J2EE Client container. Two interfaces are defined: LocalCounter and RemoteCounter. See the code in Example B-5.

Example: B-5 LocalCounter interface

```
package com.ibm.websphere.ejb3sample.counter;

import javax.ejb.Local;

@Local
public interface LocalCounter {
    public int increment();
    public int getTheValue();
}
```

In this coding example, the important feature is to define Session Bean methods to be reached by Servlets in the same JVM as EntityBean. To accomplish this, we have to declare @Local annotation (Example B-6).

Example: B-6 RemoteCounter interface

```
package com.ibm.websphere.ejb3sample.counter;

import javax.ejb.Remote;

@Remote
public interface RemoteCounter {
    public int increment();
    public int getTheValue();
}
```

This coding is required for access by another JVM environment using client code inside a J2EE Client Container. Also, we see the methods of SessionBean and the @Remote annotation for EJBContainer as a remote interface. Note that the main difference between the interfaces is the annotation setup.

Finally, we can see the session bean code in Example B-7.

Example: B-7 StatelessCounterBean code

```
package com.ibm.websphere.ejb3sample.counter;

import javax.ejb.Stateless;
import javax.interceptor.Interceptors;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
@Interceptors ( Audit.class )

public class StatelessCounterBean implements LocalCounter,
RemoteCounter {

    private static final String CounterDBKey = "PRIMARYKEY";

    // Use container managed persistence - inject the EntityManager
    @PersistenceContext (unitName="Counter")
    private EntityManager em;

    public int increment()
    {
```

```
        int result = 0;

        try {

            JPACounterEntity counter = em.find(JPACounterEntity.class,
CounterDBKey);

            if ( counter == null ) {
                counter = new JPACounterEntity();
                counter.setPrimaryKey(CounterDBKey);
                em.persist( counter );
            }

            counter.setValue( counter.getValue() + 1 );
            em.flush();
            em.clear();

            result = counter.getValue();

        } catch (Throwable t) {
           System.out.println("StatelessCounterBean:increment - caught
unexpected exception: " + t);
            t.printStackTrace();
        }

        return result;
    }


    public int getTheValue()
    {
        int result = 0;

        try {
            JPACounterEntity counter = em.find(JPACounterEntity.class,
CounterDBKey);

            if ( counter == null ) {
                counter = new JPACounterEntity();
                em.persist( counter );
                em.flush();
            }

            em.clear();
```

```
            result = counter.getValue();
        } catch (Throwable t) {
           System.out.println("StatelessCounterBean:increment - caught
unexpected exception: " + t);
            t.printStackTrace();
        }

        return result;
    }
}
```

Analyzing this coding, we note that the @Stateless annotation before class declaration is required to indicate that is a session bean. The session bean also implements LocalCounter and RemoteCounter at the same time. As the interfaces use the same method signature, there is no necessity to duplicate increment() and getTheValue() methods.

Note in the declaration of EntityManager that there is an annotation @PersistanceContext (unitName="Counter") that maps to persistence.xml <persistence-unit name="Counter"> described in Example B-3 on page 635. Also note that EntityManager is not created by the code, however the container creates the EntityManager automatically doing an injection approach. If you use EntityManager out of a EJB Container, the main difference is that you have to instantiate the EntityManager.

And finally, the EntityBean instance is not created in each access. To increment the counter, EntityManager gets the same EntityBean instance doing a em.find(JPACounterEntity.class,counterDBKey) where counterDBKey uses the same value of primarykey that is "PRIMARYKEY" value.

C

WebSphere Application Server Toolkit 6.1

The WebSphere Application Server Toolkit helps you create, test, and deploy applications with WebSphere Application Server Version 6.1.x. All the tools are integrated into a workbench to simplify the development process, but you can also use command line tools. Wizards for creating Java, J2EE, EJB, and Portlet applications help you quickly get started by creating projects with a basic set of files. Editors provide code assist and validation to improve productivity. Integration with WebSphere Application Server enables you to quickly test and deploy applications from the workbench.

The information in this appendix is a subset of material from the WebSphere Application Server InfoCenter. For more information, go to the following URL:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

Application Server Toolkit

Here we describe a typical scenario for developing large applications using WebSphere Application Server Toolkit:

A team of Application Developers write code on their personal workstation. WebSphere Application Server Toolkit supports a variety of technologies such as J2EE 1.4, Enterprise JavaBeans 2.1, Web Services, XML, and portlets.

Application Developers check their code into a source control system allowing them to manage, share, and synchronize resources. The workbench can be configured to work with CVS, Clearcase, or other source control systems.

Application Developers deploy Web applications to a unit test server where they can test and debug their code.

A Build Engineer generates scripts for building, packaging, and deploying the application. They run daily or weekly builds of the application. A Jython editor simplifies the development and maintenance of wsAdmin scripts.

Test Engineers perform functional testing on the application.

The Solution Deployer deploys the completed application onto a production server.

If you have used previous versions of WebSphere Application Toolkit, you should read What's New, to understand what features have been added this release. If you are new to WebSphere Application Toolkit, you should familiarize yourself with the Navigating and customizing the workbench topics in the online help before learning more advanced programming tasks.

Starting WebSphere Application Server Toolkit

Before you can assemble code artifacts into modules, you must install Application Server Toolkit. The Application Server Toolkit is available on CD-ROM in the WebSphere Application Server CD-ROM package. To install Application Server Toolkit, follow the installation instructions that is on its CD-ROM. Install Application Server Toolkit on either one of the supported operating systems: Linux® Intel® or Windows platform.

Install WebSphere Application Server Toolkit using the installation guide. The installation guide for WebSphere Application Server Toolkit is available at `x:/readme/readme_install_ast.html` where x is the installation directory of WebSphere Application Server Toolkit.

After that perform the following procedures:

1. Run the executable of the application:
   - From a command prompt, change to the installation directory of WebSphere Application Server toolkit directory and type **ast**.
   - In Windows: **Click Start** → **Programs** → **IBM WebSphere** → **Application Server Toolkit Vx** → **Application Server Toolkit**.
   - In Linux: If you are using the GNOME desktop environment, open the main menu and click **Programming** → **Application Server Toolkit Vx**. If you are using the K Desktop Environment, open the main menu and click **IBM WebSphere** → **Application Server Toolkit Vx** → **Application Server Toolkit**.
2. In the Workspace Launcher dialog, specify the workspace directory and click **OK** to launch the integrated development environment (IDE).

In the Project Explorer view of the J2EE perspective, **Window** → **Open Perspective** → **Other** → **J2EE** → **OK** displays a hierarchical structure used to build the contents of a new module, or to work with the contents of an existing module.

Configuring Application Server Toolkit

When you first start WebSphere Application Server Toolkit, menu choices for the J2EE Perspective might not be enabled. To assemble code artifacts into J2EE modules that can be deployed onto an application server, you must work in the J2EE Perspective. This topic explains how to configure your assembly tool for work on J2EE modules and specify a target server supported by WebSphere Application Server.

Prerequisites

Install and start WebSphere Application Server Toolkit. See “Starting WebSphere Application Server Toolkit” on page 642.

Configuring WebSphere Application Server Toolkit consists of ensuring that menu choices for the J2EE Perspective are enabled and specifying a target server supported by WebSphere Application Server. When you first start an assembly tool, menu choices for the J2EE Perspective might not be enabled, meaning that you cannot assemble code artifacts into deployable J2EE modules. You perform the steps in this task when you cannot work in the J2EE Perspective or when you need to specify a new target server for your modules.

Enable menu choices for the J2EE perspective:

1. Click **Window** → **Customize Perspective**.
2. In the **Customize Perspective** dialog, select **J2EE**. Also select **EJB, Web**, and any other categories that you might need. Then, click **OK**.
3. Click **Window** → **Open Perspective** → **Other** → **J2EE** → **OK**.
4. Select the **Project Explorer** view for your work.
5. Click **Window** → **Show View** → **Other** → **General** → **Project Explorer** → **OK**. The Project Explorer view is displayed in a panel of the workbench.
6. Define a target server for your modules.

To work on Version 6.1.1 modules, select WebSphere Application Server v6.1 as the target server runtime.

Abbreviations and acronyms

AJAX	Asynchronous JavaScript and XML
ANSI	American National Standards Institute
API	application programming interface
APPC	Advanced Program to Program Communication
ASCII	American Standard Code for Information Interchange
AST	Application Server Toolkit
AWT	Abstract Window Toolkit
BLOB	binary large object
BMP	Bean Managed Persistence
BPEL	Business Process Execution Language
CCF	Common Connector Framework
CD-ROM	compact-disc read-only memory
CF	ConnectionFactory
CGI	Common Gateway Interface
CICS	Customer Information Control System
CLOB	character large object
CLP	command line processor
CMP	Container Managed Persistence
CMR	Container Managed Relationship
COBOL	Common Business Oriented Language
COMMAREA	communication area
CORBA	Common Object Request Broker Architecture
CPU	central processing unit
CSS	cascading style sheet channel subsystem
CVS	Concurrent Versions System
DAO	Data Access Object
DB	database
DBA	database administrator
DBMS	Database Management System
DBRM	Database Request Module
DMS	Data Mediator Service
DTO	Data Transfer Object
DWORD	double word
EAI	Enterprise Application Integration
EAR	enterprise archive
EBCDIC	Extended Binary Coded Decimal Interchange Code
ECI	external call interface
EGL	Enterprise Generation Language
EIS	Enterprise Information System
EJB	Enterprise JavaBeans
EJBQL	EJB Query Language
EL	Expression Language
ERP	Enterprise Resource Planning
FIFO	First In, First Out
GB	gigabyte
GC	garbage collector
GIOP	General Inter-ORB Protocol
GNOME	GNU Network Object Model Environment
GNU	GNU's Not Unix
GUI	graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP over SSL Hypertext Transfer Protocol Secure
I/O	input/output
IBM	International Business Machines Corporation
IC	initial context

ID	identifier
IDE	integrated development environment
IIOP	Internet InterORB Protocol
IMS	Information Management System
IO	input/output
IP	Internet Protocol
IT	information technology
ITSO	International Technical Support Organization
JAR	Java archive
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML Web Services
JCA	J2EE Connector Architecture
JDBC	Java Database Connectivity
JDK	Java Development Kit
JDO	Java Data Objects
JIT	Just-In-Time
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JPA	Java Persistence API
JPQL	EBJ Query Language
JRE	Java Runtime Environment
JSF	JavaServer Faces
JSP	JavaServer Pages
JSR	Java Specification Request
JSTL	JSP Standard Tag Library
JTA	Java Transaction API
JTS	Java Transaction Service
JVM	Java Virtual Machine
JVMPI	Java virtual machine profiler interface
KB	kilobyte
LDAP	Lightweight Directory Access Protocol
LRU	Least Recently Used
LT	line transmission termination logical terminal
MB	megabyte
MDB	message-driven beans
MOM	message oriented middleware
MQ	managers and queues
MS	message store management services
MVC	Model View Controller
ND	network deployment
NIO	new input/output
OASIS	Organization for the Advancement of Structured Information Systems
OO	opportunity ownership
ORB	Object Request Broker
ORM	object relational mappers
OS	operating system
PM	project manager project management
PMT	Profile Management Tool
POJO	Plain Old Java Object
POST	power-on self test
PTP	point-to-point model
RAD	rapid application development
RAID	redundant array of independent disk
RDBMS	Relational Database Management System
RFC	Request for Comments
RMI	Remote Method Invocation
RMI/IIOP	Remote Method Invocation over Internet InterORB Protocol
RPC	remote procedure calls
RRD	Remote Request Dispatcher
RRS	Resource Recovery Services
RSS	rich site summary
RUP	Rational Unified Process
SAM	Software Asset Management
SAP	service access point
SAR	SIP archive
SCA	Service Components Architecture

SCM	Source Code Management
SDK	software development kit
SDO	Service Data Objects
SE	Support Element IBM systems engineer
SIP	Session Initiation Protocol
SNA	Systems Network Architecture
SOA	service-oriented architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SQLJ	Structured Query Language for Java
SSL	Secure Sockets Layer
SWT	Standard Widget Toolkit
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TG	transmission group
TLD	tag library descriptor
UCM	Unified Change Management
UDB	Universal Database
UDDI	Universal Description Discovery and Integration Universal Description, Discovery, and Integration
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VO	virtual origin
VOB	versioned object base
VRMF	version, release, modification, fix
WAR	Web ARchive
WLM	Workload Management
WML	Wireless Markup Language
WSDL	Web Services Description Language
XA	extended architecture
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XSD	XML Schema
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see "How to get Redbooks" on page 651. Note that some of the documents referenced here may be available in softcopy only.

- *WebSphere Application Server V6 Scalability and Performance Handbook,* SG24-6392
- *Rational Application Developer V6 Programming Guide*, SG24-6449
- *WebSphere Application Server - Express V6 Developers Guide and Development Examples*, SG24-6500
- *Performance Monitoring and Best Practices for WebSphere on z/OS*, SG24-7269
- *WebSphere Application Server V6.1: System Management and Configuration*, SG24-7304
- *WebSphere Application Server V6.1: Planning and Design*, SG24-7305
- *Performance Monitoring and Best Practices for WebSphere on z/OS*, SG24-7269

Other publications

These publications are also relevant as further information sources:

- Design Patterns: Elements of Reusable Object-Oriented Software, ISBN:0201633612
- Core J2EE Patterns: Best Practices and Design Strategies. ISBN:0131422464
- EJB Design Patterns

  ISBN:0471208310

Online resources

These Web sites are also relevant as further information sources:

- EJB Best Practices:

  http://www.ibm.com/developerworks/java/library/j-ejbcol.html
- Core J2EE Design Patterns:

  http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html
- Core J2EE Patterns, Best Practice and Design Strategies:

  http://www.corej2eepatterns.com/Patterns2ndEd/
- IBM WebSphere Developer Technical Journal: The top Java EE best practices:

  http://www.ibm.com/developerworks/websphere/techjournal/0701_botzum/0701_botzum.html
- Five common PHP design patterns:

  http://www.ibm.com/developerworks/opensource/library/os-php-designptrns
- Developing and Deploying Modular J2EE Applications with WebSphere Studio Application Developer and WebSphere Application Server:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0206_robinson/robinson.html
- Writing efficient thread-safe classes:

  http://www.ibm.com/developerworks/java/library/j-threadsafe/
- IBM WebSphere Developer Technical Journal: Leveraging OpenJPA with WebSphere Application Server V6.1:

  http://www.ibm.com/developerworks/websphere/techjournal/0612_barcia/0612_barcia.html
- OpenJPA User's Guide:

  http://openjpa.apache.org/docs/openjpa-0.9.7-incubating/manual/manual.html
- IBM WebSphere Application Server 6.1 Info center:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads:

ibm.com/support

IBM Global Services:

ibm.com/services

Index

B

F

G

Q

R

X

IBM

Redbooks

Designing and Coding Applications for Performance and Scalability in WebSphere Application Server

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Designing and Coding Applications for Performance and Scalability in WebSphere Application Server

Application design

Performance considerations

Best practices

In many WebSphere Application Server environments, support personnel often get more questions on application performance and tuning than on WebSphere Application Server tuning.

While there is much documentation and guidance for installing, monitoring, and tuning the performance and scalability aspects of WebSphere Application Server, not a lot of guidance and recommendations are given for performance and scalability considerations when designing and coding applications that execute in the WebSphere Application Server environment.

This IBM Redbooks publication provides performance and scalability considerations to keep in mind when developing and coding WebSphere Application Server applications. In this book, we take a layered approach to application development covering performance and coding considerations for each layer in a separate chapter.

In addition, various application development tools and strategies are compared within each layer along with best practices to keep in mind when designing and developing applications.

The target audience for this book includes the application development team, especially architects and developers. It also includes developers with experience using various application development techniques and tools for the different layers in the application architecture.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7497-00 ISBN 073848864X