WebSphere® software

IBM

Experience Java EE! Using WebSphere Application Server Community Edition 2.1

On-ramp to Java EE development

Including EJB 3.0, Web services, JMS and Web 2.0

Detailed hands-on tutorials

Ueli Wahli
Charles P Brown
Steven Calello
Rafael Coutinho
Patrick Gan
Cedric Hurst
Maan Mehta

Redbooks

ibm.com/redbooks

International Technical Support Organization

Experience Java EE! Using WebSphere Application Server Community Edition 2.1

February 2009

SG24-7639-00

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

First Edition (February 2009)

This edition applies to WebSphere Application Server Community Edition (WASCE) Version 2.1, and Eclipse Ganymede, which was used as the integrated development environment for WASCE.

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

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

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

AIX®
CICS®
DataPower®
DB2®
developerWorks®
IBM®
POWER®
Rational®
Redbooks®
Redbooks (logo) ®
Tivoli®
WebSphere®

The following terms are trademarks of other companies:

Acrobat, Adobe Flash, Adobe Flex, Adobe, Flex Builder, Macromedia, MXML, and Portable Document Format (PDF) are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

SUSE, the Novell logo, and the N logo are registered trademarks of Novell, Inc. in the United States and other countries.

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

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

EJB, Enterprise JavaBeans, J2EE, J2SE, Java, Java runtime environment, JavaBeans, JavaFX, JavaMail, JavaScript, JavaServer, JDBC, JDK, JRE, JSP, JVM, Solaris, Sun, Sun Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

ActiveX, Internet Explorer, Microsoft, Silverlight, SQL Server, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Pentium 4, Pentium, 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

This IBM® Redbooks® publication is a hands-on guide to developing a comprehensive Java™ EE application using WebSphere® Application Server Community Edition v2.1 (WASCE), including core functions, security, Web services, and messaging.

Novice users are thus able to experience Java EE, and advance from theoretical knowledge gained by reading introductory material to practical knowledge gained by implementing a real-life application.

Experience is one stage in gaining and applying knowledge, but there are additional stages needed to complete the knowledge acquisition cycle. This book also helps in those stages:

- Before experiencing Java EE, you ***learn*** about the base specifications and intellectual knowledge of Java EE through brief descriptions of the theory and through links to other information sources.
- After experiencing Java EE, you ***explore*** advanced Java EE through previews of advanced topics and links to other information sources.

This publication is not intended to duplicate the numerous excellent documents, tutorials, and demonstrations that provide both basic Java EE introductions (Learn) and expert level information (Explore) on specific Java EE aspects. Many of the other resources are available at the following Web sites:

- IBM developerWorks® and WASCE:

  http://www.ibm.com/developerworks
  http://www.ibm.com/developerworks/websphere/zones/was/wasce.html

- IBM Redbooks publications:

  http://www.redbooks.ibm.com

- Sun™ Java™ 2EE:

  http://java.sun.com/javaee

The target audience is technical users who have minimal experience with Java EE and the IBM WebSphere Application Server product set, but who do have past experience in using an integrated development environment (IDE) and in architecting or designing enterprise applications.

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, San Jose Center.

Ueli Wahli is a Consultant IT Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO over 20 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide about WebSphere Application Server, and WebSphere and Rational® application development products. In his ITSO career, Ueli has produced more than 40 IBM Redbooks publications. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology.

Charles P Brown (Charlie Brown) is an IBM Senior Consulting IT Specialist in the WebSphere brand of products. He spent over 20 years with IBM working in robotics, software development, beta support programs (for IBM in the United Kingdom), and for the last six years supporting IBM partners. Charlie's focus is the practical application of product and technology, helping partners and clients develop skills, and then assisting them in being successful in their initial implementations. Charlie holds a Bachelors of Science in Mechanical Engineering (M.I.T., 1984) and a Masters of Science in Computer Science (Johns Hopkins, 1989). Charles was the author of an earlier version of this book, Experience J2EE™!

Steven Calello has been with IBM since 1999 and has worked on numerous projects in four different divisions. His experience includes Global Business Services, Pervasive Computing, Personal Systems, and Software Group. He is currently a Technical Sales Leader for the World Wide WebSphere Sales Organization. He routinely presents at conferences, client sites, and Fortune 500 companies, as well as guest lectures at several universities. He is working towards becoming a World Wide Application Infrastructure Sales Specialist. Steven holds a Bachelors of Science in Computer Information Systems from Clemson University.

Rafael Coutinho is an IBM Accredited IT Specialist working for Global Services since 2005. He has 6 years of professional experience in many areas covering IT and Telecom expertise from embedded to Platform based solutions, using multiple technologies. He is a certified Java enterprise architect and specializes in high-performance distributed applications on corporate and financial projects. Rafael is a computer engineer graduate from the State University of Campinas (Unicamp), Brazil, and has a degree in information technologies from the Centrale Lyon (ECL), France.

Patrick Gan is a Senior IT Specialist who works for IBM Global Services, Application Innovation Services Center of Excellence, US. He has eight years of experience and expertise in OOAD/Java Design best practices, J2EE development, SOA, software development methods, and tools. In his current capacity, Patrick's primary responsibility in customer facing engagements is solution design and delivery of custom enterprise solutions. In addition, he has also been involved in the design and development of IBM assets and has authored articles on developerWorks. Patrick has a Computer Science degree from California Polytechnic State University, Pomona.

Cedric Hurst is a Java EE Architect working as an independent consultant at his own company, Foosoft Inc. He graduated from the University of Illinois at Chicago with a degree in Information and Decision Sciences. He has spent over 12 years in the technical field, working primarily with Web and data-tier applications and utilizing a number of IBM, third-party, and open source frameworks. Cedric worked as a Technical Specialist for the IBM Innovation Center for Business Partners in Chicago, where he helped Independent Software Vendors port their applications to IBM platforms and technologies. He also worked with the IBM developerWorks team to develop and introduce WebSphere Community Edition after the Gluecode acquisition in 2005. Cedric has developed and led several hands-on workshops focusing on the Java Persistence API, EJB™ 3.0, JavaServer™ Faces, and WebSphere Community Edition at several major IBM conferences, such as IBM IMPACT and the WebSphere Services Technical Conference. He is also active on the Apache Geronimo mailing list, and within the Java Community Process (JCP), Groovy, Grails, and Dojo communities.

Maan Mehta is a WebSphere Portal Architect. He works as an independent consultant and owns Revecore, Inc., a WebSphere Services company. Maan has over 11 years of experience in leading and delivering enterprise application development projects and also WebSphere and Portal Infrastructure projects. Most of Maan's working experience has been with IBM Software Group (Toronto Lab) and IBM Global Services in Canada. He holds a degree in Computer Science from the Queen's University at Kingston, Canada, and an Electronics Sciences Degree from the University of Delhi, India. His areas of expertise include analysis and design specifically with technologies such as Portal, Java EE, Flex, and Ajax. He also has extensive experience in the administration and configuration of WebSphere Application Servers and WebSphere Portal Servers.

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

- Ron Staerker, IBM, World Wide Sales for WASCE
- Bill Stoddard, IBM, Apache Geronimo and WASCE development, and his group: David Blevins, Jun Jie Cai, Jaroslaw Gawor, David Jencks, Ted Kirby, Tim McConnell, Barrett Reed, Lin Sun, and Donald Woods
- Kevan Miller, IBM, Apache Geronimo and WASCE
- Kevin Sutter, IBM, Java Persistence API
- Robert Ringo and Paul Craton, IBM, WebSphere Virtual Enterprise

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 can have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts can help increase product acceptance and customer satisfaction. As a bonus, you can 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

Part 1

Preliminary activities

In Part 1, we perform the following activities:

- Read an introduction to both Java EE and the scenario used in this book
- Install and configure the prerequisite software
- Configure the development environment
- Create and populate the database that represents the legacy application

1

Introduction

In this chapter, we introduce you to the following items:

- The purpose and intent of this IBM Redbooks publication
- An overall view of Java EE
- The scenario that forms the basis for the remainder of this publication

1.1 Learn, Experience, Explore!

“We have developers who know C++ and PERL or Microsoft® .NET, but we have to get them up to speed quickly on Java EE. Is there a good introductory hands-on book—not Java EE for Dummies but something that provides a cohesive view of Java EE with enough information to allow them to develop a real-life application?

Is there a book that helps them to experience Java EE?”

This same basic question has been voiced over many technologies in the Information Technology (IT) industry, most recently with Java Platform, Enterprise Edition (Java EE) 5. Refer to 1.2, “Java EE simplified” on page 7 for a brief introduction to Java EE.

Java EE developers have access to the many Web sites, books, articles, and news groups that deal with Java EE and related technologies. Still, new Java EE developers find it very difficult to move from reading this mass of information to actually applying the technology to solve business issues.

The British have a saying that refers to something being *used in anger*. The usage is more than just reading or playing: Something is actually done. In this, case, how do developers effectively use *Java EE in anger*? What is an optimal way to obtain this knowledge, and what are they not finding?

A previous IBM Redbooks publication, *Experience J2EE! Using WebSphere Application Server V6.1*, SG24-7297, suggested that the process of acquiring and applying knowledge of a technology would be more effective if the learning process was separated into the following three phases:

- **Learn** The basic specifications, intellectual knowledge, and theory of the technology, through reading specifications and product documentation.
- **Experience** A simple but still real-life application of the technology, through classes, initial usage, and hands-on tutorials.
- **Explore** Advanced features, latest changes, and best practices, through reading articles, specifications, and bulletin boards.

That book suggested that generally available information on technology is lacking in the Experience phase. Most guided tutorials/examples are written independently to focus on a specific area. This can be helpful for someone who already knows other aspects of the technology, but it can be very frustrating for those trying to experience the overall technology.

- The reader is left with disconnected segments of knowledge that they are unable to apply together to develop complete cohesive enterprise applications. For example, after completing a tutorial on session Enterprise JavaBean (EJB) security, the reader might understand that subject area, but does not understand how it links with security for Web front ends, thick clients, Web services, and messaging (JMS). This is the adage of *not being able to see the forest for the trees*.
- The reader must perform repetitive basic configuration tasks in each scenario (because each guided tutorial/example uses a separate sample application). This increases the amount of time they must invest to learn the technology, and it detracts from the focus and clarity of the example (for example, what is being done as the basic configuration versus what is being done as part of learning the specific scenario).
- The reader is presented with as much detail as possible about the specific Java EE aspect (including the proverbial kitchen sink). This works well for experienced Java EE developers, who use the material as a reference, but it can overwhelm and confuse the Java EE novice.

Experience UNIX?

This problem is not unique to Java EE. Step back to the late 1980s and early 1990s and consider the desktop operating system battle that Microsoft Windows® ultimately won. The UNIX® operating systems and associated technologies (xWindows, Motif, NFS, Yellow Pages, and others) in the late 1980s contained functionality far superior to the various versions of Microsoft Windows, yet they lost. Why?

These products and their associated user communities were deficient in the Learn and Experience phases, and they did not attempt to cultivate new users. Instead, they focused on arcane knowledge, such as complex keystroke macros for the `vi` editor and cryptic commands such as **`awk`**, **`sed`**, and **`grep`**. The unusability was adopted as a badge of honor and defined the technology and the community. “Novice users need not apply!”

Ultimately, the UNIX community was not completely oblivious to these issues, as evidenced by the focus on Linux®, providing a user-friendly version of the UNIX operating system for the desktop. Better late than never!

The impetus for the previous book was an effort by an IBM team in 2003 to provide initial J2EE training using a series of lectures and labs that came from multiple independent sources. The team was constantly faced with the problems described before. One of the team members expressed frustration about this and said that if they had time, they should create a set of material that uses a single-integrated scenario to address these problems.

The result was the Experience J2EE! book, which enabled readers to *experience* the full spectrum of basic J2EE application development and deployment in a single document (that is not 2000 pages in length). This book was focused on J2EE v1.4 and WebSphere Application Server v6.1.

This book is the successor to the Experience J2EE! book, enabling readers to experience the full spectrum of basic Java EE application development in a single document using WebSphere Application Server Community Edition v2.1:

- This publication utilizes a single-integrated scenario across all of Java EE. You learn and set up a single application, and sequentially add the various Java EE technologies, thus you can focus only on the new technology in each section (and how it relates to the technologies covered in previous sections).

  Section 1.3, "Introduction to the scenario" on page 19 describes the scenario used in this book. The following IBM software products are used to implement the Java EE application:

Java EE runtime:	WebSphere Application Server Community Edition v2.1, abbreviated ***WASCE***
Java EE development tool:	Eclipse IDE for Java EE developers plus the WASCE Server Tools v2.1, abbreviated ***WASCE IDE***

- This book is kept to a reasonable length by providing links to other information sources for both beginning knowledge (*learning*) and advanced topics (*exploring*), thus avoiding overloading the user with useful but not directly relevant information. Following are some key sites referenced in this book:
  - IBM developerWorks:

    `http://www.ibm.com/developerworks`
  - IBM Redbooks:

    `http://www.redbooks.ibm.com`
  - Sun Java EE (aka J2EE):

    `http://java.sun.com/javaee`

The target readers are technical users, who have minimal experience with the Java EE product set, but who do have past experience in using an integrated development environment (IDE) and in architecting or designing enterprise applications.

You will not become a Java EE expert by reading and experiencing this book, but afterwards you should have enough knowledge for the following tasks:

- Develop, test, and deploy simple Java EE applications that contain necessary qualities of service such as 2-phase commit support and a consistent security model.
- Know where to look for additional information (both to learn and explore).
- Phrase the relevant questions that can help you expand your knowledge and apply the appropriate technology to business scenarios.

Use this IBM Redbooks publication as a self-guided tutorial, as demonstration scripts, or as hands-on labs for a workshop. We hope this book leads you to a successful Java EE experience!

1.2 Java EE simplified

“Java Platform, Enterprise Edition (Java EE) is the industry standard for developing portable, robust, scalable and secure server-side Java applications. Building on the solid foundation of Java SE, Java EE provides web services, component model, management, and communications APIs that make it the industry standard for implementing enterprise class service-oriented architecture (SOA) and Web 2.0 applications”

From the Java EE Frequently Asked Questions (FAQs), at the Web site:

`http://java.sun.com/javaee/overview/faq/javaee_faq.jsp`

What? We understand Java EE, and find this definition confusing! What is Java EE and why do you care?

The Sun Java site contains a good introduction that answers these questions. The introduction is called **Update: An Introduction to the Java EE 5 Platform** and is available at:

`http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5`

The Sun Java site also contains a comprehensive tutorial called **The Java EE 5 Tutorial,** located at:

`http://java.sun.com/javaee/5/docs/tutorial/doc`

If you find these to be confusing as well, following is our simplified description of Java EE. The key point to remember is that Java EE allows developers to create applications using commercial off the shelf (COTS) development software, and then provides these applications to others to install and execute on COTS runtime software.

1.2.1 Objective

> The Java EE objective is to allow developers to create applications that can be executed—without rewriting or recompiling—on runtime software products provided by a wide variety of software companies across a wide variety of operating systems and services (such as databases and back-end applications).

Java EE acts as the mediator between the application, the operating system, and the back-end services. Java EE masks the differences among the various operating systems and back-end services, providing a consistent set of interfaces and functions to the application. This allows a single version of the source code to support all environments. The term that Sun uses is WORA (Write Once, Run Anywhere).

Java EE, as indicated by the name, is implemented on the Java programming language. Java uses a bytecode compilation strategy, which is a hybrid between the pure interpretive languages, such as Basic, and pure compilation languages, such as C and Pascal. The Java compiler converts the source code into an intermediate bytecode format. This bytecode can then be executed by any operating system that has the Java Virtual Machine (JVM™) runtime environment installed. The complete specification is called Java 2 Standard Edition (J2SE™). The term that many in the industry use is CORA (Compile Once, Run Anywhere).

In theory, a developer could write an application and, without changing a single line of code, run it on a wide range of environments:

- Java EE runtime providers (IBM, BEA, Oracle®, Sun)
- Operating systems (Microsoft Windows, Unix, Linux, AIX®)
- Databases (DB2®, Oracle Database, Sybase, Microsoft SQL Server®)
- Back-end information systems (SAP®, PeopleSoft®, and others)

In practice, Java EE comes pretty close to meeting this objective. In some situations, applications run on different runtime providers (and operating systems, databases, back-end services) without changing a single line of code or packaging information. WORA and CORA! However, in the real world, most applications require minor adjustments to run in the various environments:

- Java EE specifications do not cover all technologies and functions, so developers are often forced to write to environment specific interfaces/functions that change when the application is moved to another environment.

- Java EE specifications are open to interpretations (in spite of the best intentions of those who define the specifications) leading to differences in the application program interfaces and packaging supported by the various Java EE runtime providers.
- The Java EE runtime cannot completely mask the differences between the various environments. For example, DB2 and Oracle databases have different requirements for table names.

We are not discrediting Java EE by asserting that it cannot support 100% code portability in all situations. Rather, we are trying to set realistic expectations.

Our experiences suggest that most Java EE applications exceed 99% code portability—minimal tweaking is needed when moving from one environment to another. This is certainly far better than writing applications directly to proprietary, platform specific interfaces that mandate almost total code re-write when moving to another environment.

1.2.2 Specifications

> Java EE meets this objective by providing a set of architectural specifications and features that define and support the relationships between the application, the runtime environments, and the common services.

Java EE contains the following four core components for your use:

- Development specifications that support the creation of software applications, in particular application programming interfaces (API) and packaging requirements.
- Execution specifications that define the environments for executing Java EE applications, in particular API implementations, common services, and how to process the software application packaging.
- Compatibility test suite for verifying that the runtime environment product complies with the Java EE platform standard.
- Reference implementation that provides a rudimentary implementation of the Java EE specifications. This is quite adequate for base learning but does not have extended quality of services such as an administration GUI, logging, clustering, and development tools.

Changes in Java EE standards

The Java EE standards were first formally released in 1999 and continue to evolve.

Java EE version	Based on J2SE	Final specification release date	JSR (see discussion)
2 (1.2)	1.3	12/1999	n/a
3 (1.3)	1.3	09/2001	n/a
4 (1.4)	1.4	11/2003	151
5	5	05/2006	244
6	6	draft 07/2007	316

Note that the naming structure changed with the fifth version from Java 2 Platform, Enterprise Edition (J2EE) v1.x to Java Platform, Enterprise Edition (Java EE) x. For example, the version 4 specification is formally called J2EE 1.4 (not Java EE 4), however, the version 5 specification is called Java EE 5.

The pace of releases slowed down as the specifications matured, reflecting the broader install base (more pressures on maintaining compatibility) and the much broader user community (the more people involved, the longer it takes to reach a consensus).

The specification updates are also changing in focus. Earlier Java EE releases added new functions and features, but many of the changes in Java EE 5 are driven from the desire for simplification. For example, one of the key changes is reducing the number of text descriptor files (so called *deployment descriptors*) by moving the information into *annotations* contained directly in the Java files. This reduces the number of overall artifacts and helps avoid the problem of having deployment descriptors out of sync with the Java files.

This switch of emphasis to simplification illustrates the maturity of the Java EE platform. The platform provides enough base functions for users to develop highly-functional and portable applications. Users are generally satisfied with the current functions, and now are looking instead at how it can be done better and simpler.

Sun maintains these specifications, and updates them using a process called the Java Community Process (JCP). The JCP Web site is at:

```
http://jcp.org
```

JCP works on a broad range of specifications called Java Specification Requests (JSRs). These JSRs might or might not be incorporated into new versions of Java EE and J2SE, based on input from the open community.

The Java EE 5 specification is formally called *JSR 244: JavaTM Platform, Enterprise Edition 5 (Java EE 5) Specification*, by Sun, and is available at the following Web site:

http://jcp.org/en/jsr/detail?id=244

Deployment descriptors versus annotations:

For example, an EJB session bean can be defined using an annotation or a deployment descriptor:

- The annotation is practical for programmers:

```
@Stateless
public class FundFacade implements FundFacadeInterface { .... }
```

- The deployment descriptor is practical for deployers:

```
<?xml version="1.0".....>
   <display-name>FundFacade</display-name>
   <enterprise-beans>
      <session>
         <ejb-name>FundFacade</ejb-name>
         <ejb-class>donate.ejb.impl.FundFacade</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
      </session>
   </enterprise-beans>
</ejb-jar>
```

In practice, many robust Java EE application will include both.

1.2.3 Implementations

Software companies use these specifications to create and sell development and runtime Java EE software products.

Java EE is owned by Sun and licensed to companies that must pass the included compatibility test suite before shipping a product with the Java EE logo.

For a list of current authorized licensees of the Java EE and J2EE platforms, visit the following Web site:

```
http://java.sun.com/j2ee/licensees.html
```

These companies have permission to ship products that are branded Java EE and that have passed the Java EE compatibility tests.

For the list of authorized licencees who have developed products that have passed the Java EE compatibility tests, visit the following Web site:

```
http://java.sun.com/javaee/overview/compatibility.jsp
```

The following table lists the current Java EE product providers (commercial and open source) as of the writing of this book:

Vendor	Runtime product
Apache	Apache Geronimo v2.1.2
BEA	WebLogic Server v10.0
IBM	**WebSphere Application Server Community Edition v2.1** **WebSphere Application Server v7**
Kingdee	Apusic Application Server v5.0
NEC	WebOTX Application Server v8.1
Oracle	Oracle Application Server v11
SAP	SAP NetWeaver® v7.1
Sun	Sun Java System Application Server Platform Edition v9
TmaxSoft	TmaxSoft JEUS v6

In theory, an application generated by any development product can run on any runtime product. In reality, there is some affinity between development products and runtimes products due to the following:

- Some of the minor variations in Java EE implementations, due to the specifications being incomplete or imprecise.
- Support for areas where the runtime product implements proprietary (but still valuable) extensions to the specifications.
- Choosing to implement draft or up level specifications that are required by the next level higher of Java EE.
- Testing and debugging tools for that particular runtime product.

WebSphere Application Server v7.0 and Rational Application Developer v7.5

IBM markets two J2EE/Java EE platform platforms, WebSphere Community Edition and WebSphere Application Server.

WebSphere Application Server is the IBM premier Java EE runtime that provides all the programming model specifications plus additional infrastructure support (scalability, administration, failover, logging, diagnostics, clustering) needed to support highly available, highly reliable solutions. WebSphere Application Server is a fee-based product, both for the initial purchase and for support after the first year of ownership (the first year is included in the purchase fee).

In contrast, WebSphere Application Server Community edition has a zero cost purchase, but support is fee-based.

The recently announced WebSphere Application Server v7.0 also supports Java EE 5, and a matching development product, Rational Application Developer v7.5, was announced at the same time. Both products are available as trial downloads:

http://www.ibm.com/developerworks/downloads/ws/was/?S_TACT=105AGX28&S_CMP=DLMAIN&S_CMP=rnav
http://www.ibm.com/developerworks/downloads/r/rad/?S_TACT=105AGX23&S_CMP=DWNL&S_CMP=rnav

1.2.4 Web services and messaging

Java EE provides both traditional Internet server and program-to-program interaction through Web services and messaging.

Java EE originated as a set of specifications to support dynamic interaction over hypertext transfer protocol (HTTP) between browsers, thick clients, and back-end systems such as databases. The typical *page* content was and is hypertext markup language (HTML), but the environment supports other formats, such as cHTML, VXML, and XML, as well.

Over time, Java EE continues to refine and extend this support, but it has also been extended to support program-to-program type interaction as well. The two key technologies are Web services (discussed in Part 3, “Web services” on page 321) and messaging (discussed in Part 4, “Messaging” on page 403).

1.2.5 Container architecture

The Java EE runtime environment consists of a series of containers that provide a set of common services that mediate between the application components and the actual runtime environment (operating system, database).

The developer can focus on the business logic or presentation interface, and the Java EE container manages the common services such as security, session state, transactions, database access, fail-over, and recovery.

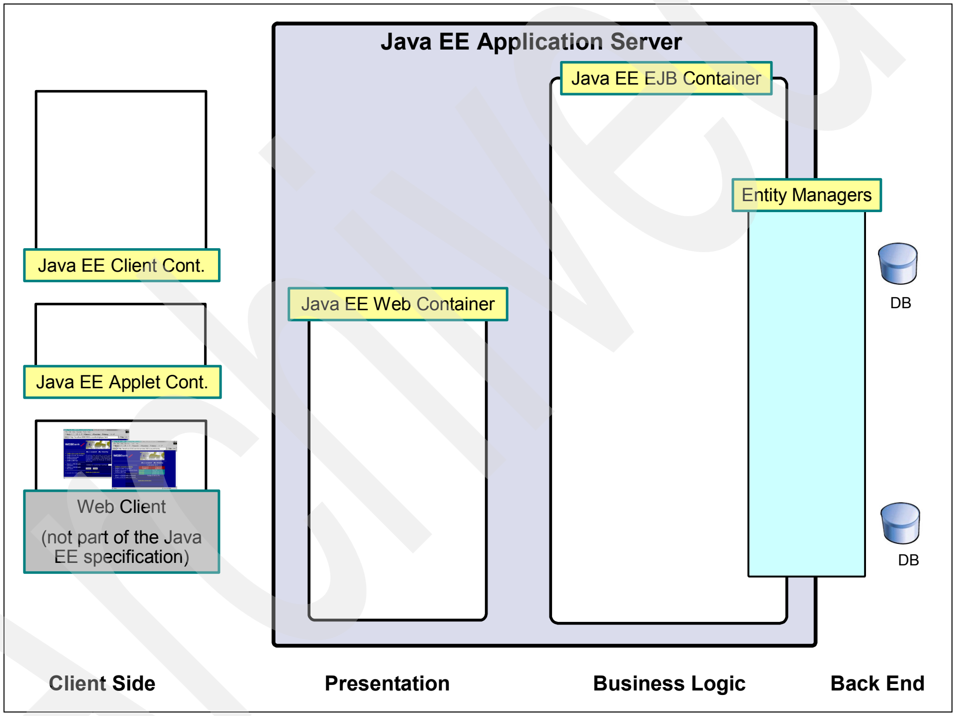


Figure 1-1 Java EE container architecture

The description of Figure 1-1 is meant to provide a brief introduction to the overall Java EE container environment and programming artifacts. The individual scenario chapters will provide more detail on the specific Java EE components (for example, servlets, JSPs, EJBs).

Java EE literature tends to describe the containers in the context of tiers or layers, where each tier or layer performs a specific function in the overall distributed application. We start from the left in Figure 1-1.

Client side tier

The client side tier represents the *glass* interaction with the user:

- The Web client is a browser: An application on a client system that supports interaction over HTTP, typically with a content type of HTML. This is a thin client: The user does not have to install any software on the client system (beyond the Web browser).

  Technically, this is not a Java EE container and the internal operations and capabilities of the browser are not part of the Java EE specifications and do not have access to the server side Java EE services.

  The only specific Java EE requirement is that the browser support HTML 3.2 or higher.

- The Java EE applet container supports the client side execution of Java programs in the context of a browser session. However, these Java programs have no direct access to server side Java EE services. Applets and client side scripting such as JavaScript™ have the same capabilities as they would in a non-Java EE environment.

- The Java EE Client container supports the client side execution of Java programs (for example, a Java class file with a main method) that have access to server side Java EE services, such as EJB, security, transactions, naming, and database access.

  This is a *thick* client: The user must install a Java EE Client container runtime environment on the system before executing the client program. Note that this can be hidden from the user by setting up a Web page that invokes an applet that downloads or installs the Java EE Client container; however, the end result is the same: The Java EE Client container is installed on the client system.

DMZ (not shown)

Many Java EE environments also insert a layer between the client tier and the presentation tier consisting of a traditional HTTP server that serves static content (HTML, images, and media).

Some call this the Web tier, but we do not like using that name because others use the name Web tier interchangeably with the presentation tier. We recommend calling this the de-militarized zone (DMZ) tier.

The DMZ is typically separated from the overall Internet through one firewall that typically lets port 80 traffic through. It is separated from the rest of the infrastructure through another firewall that typically lets through traffic only from the HTTP server to specific back-end servers. There are two motivations for having an HTTP server separate from the Java EE containers:

- Security: The DMZ, by virtue of the firewall configuration, is more susceptible to attack. If this system is compromised, the content can be wiped and the system reloaded with no loss of data (since all the content is static).
- Performance: HTTP servers are generally optimized to serve static content as fast and as efficiently as possible, and in most situations will do this much faster than a Java EE application server. Therefore, for large environments, implementing separate HTTP servers can lead to a smaller overall Java EE infrastructure.

Presentation tier

The presentation tier processes input from the client side tier, calls components in the business logic tier, and then sends a response back to the client side tier.

The Java EE Web container supports the execution of the servlet and JSP™ programming artifacts, manages the HTTP requests, and provides common services (session state, security, transactions, naming, and database access):

- Servlets provide *100% dynamic content*. They are Java classes that extend the `HttpServlet`, providing specific methods that support interaction with standard HTTP methods (get, post) and request/response objects.

  The user selects a submit action on a Web page that points to the Web address for the servlet, and the servlet is executed. The servlet can return the content directly to the browser by printing HTML statements using out.println statements or redirect to an HTML or JSP page.
- JavaServer Pages (JSPs) provide *partially dynamic content*. They are a combination of HTML and Java, and when invoked are compiled by the Web container and run—effectively as servlets. Therefore, the Web container runtime environment actually supports a single run-time programming model: servlets.

  The advantage of JSPs are that development tools can provide *what you see is what you get* (WYSIWYG) support. This provides a much more effective development process than the series of out.println statements that you code in a servlet.
- HTML and other static content can also be served up by the Web container. You can choose to do this in small environments (where you do not want to set up an external HTTP server in addition to the Java EE application server) or when you want to control access to the resources (using Java EE declarative security).

Web containers typically contain a minimal HTTP server that directly receives requests from the Web browsers. Therefore, the Java EE application server can be used either with or without an external HTTP server.

Servlets versus JSP

When should you use a servlet versus a JSP? Typical Java EE best practices suggest considering the balance between HTML and Java:

- If you are writing a servlet and find that you are creating many `out.println` statements for HTML, you probably should be using a JSP. (Some best practices go even further: If you are using any `out.println` statements, you should switch to using a JSP, or at least split out the `out.println` statements into a separate JSP file).
- If you are writing a JSP and find that you are including quite a bit of Java statements and are interacting with attributes of the `HttpServlet` interface, you probably should split that code out into a separate servlet.

Business logic tier

The business logic tier provides a framework for executing business logic and for accessing business data in a distributed transactional environment:

The Java EE EJB container provides the services needed to support the access to and execution of business logic. Business logic and data can be accessed concurrently by many different programs on many different systems. The Java EE EJB container provides the underlying transactional services and local/remote transparency needed to support the access to and execution of the business logic:

- Enterprise JavaBeans™ (EJBs) provide the Java EE implementation of the business logic, whether data (entities) or logic (session EJBs) or messaging interactions (message-driven beans).

In a non-Java EE environment, management of the transactional interactions and local/remote invocation routines can represent a large percentage of the user code. In a Java EE environment, the business logic tier manages this, dramatically reducing the amount of user written code.

Data access in Java EE versus J2EE:

In earlier versions of Java EE, back-end data was managed through the entity EJB artifact, which provided encapsulated and transactional access to a record-oriented data element with a primary key, such as a row in a relational database. Entity EJBs could be accessed locally by a direct program call within a JVM or remotely by remote method invocation (RMI) between JVMs.

Data was thus rooted in the business logic tier, and all access to data required an EJB element.

In Java EE, the entity EJB and the associated EJB container capabilities are replaced by entities (also called entity objects) and the Java Persistence API (JPA). Entity objects provide similar capabilities as entity EJBs, but they can also be accessed from the Web container and Application Client container.

The entity objects are coordinated by an architectural component called the entity manager, represented by the blue box to the right of the EJB container in Figure 1-1 on page 14.

Standard architectural practices still suggest that data access be maintained in the business logic tier (through session EJBs called facades), but developers now have the flexibility to directly access data from other tiers when appropriate.

Chapter 5, “Create the JPA entities” on page 99 provides detailed information on entities and JPA.

Back end tier

The back end tier represents existing business systems, whether databases or other software systems:

- The Java Connector Architecture (JCA) (not shown before) provides a common programmatic way of connecting with non-Java EE programs (SAP, Oracle, WebSphere MQ, IBM CICS®) with security, transactions, and performance attributes such as connection pooling.
- JDBC™ provides a specific JCA implementation for accessing databases. Earlier JDBC implementations were not implemented on JCA.

1.3 Introduction to the scenario

You are probably still confused after reading the introduction of this book and the *Simplified Guide to the Java 2 Platform, Enterprise Edition* located on the Sun Web site. A novice can still reasonably ask: How do I apply Java EE to a business need?

Consider this simple question: *"I am trying to write an application that accesses two back-end databases. How should I access them?"* Your current knowledge of Java EE could lead you to ask questions such as these:

- *Should I access the two back-end databases using database/JBDC?*
- *Should I access them using messaging/JMS?*
- *Should I access them using a Web service?*
- *Should I access them using an application connection such as JCA?*
- *Should I utilize data federation or business logic integration (using multiple EJBs)?*
- *Should I use stored procedures?*
- *Should I use XML?*
- *How do I manage the security context of this access?*
- *How do I manage the transactional integrity?*

The quandary: How do you effectively combine the appropriate Java EE technologies in a manner that provides a responsive, scalable, robust, and flexible solution? How do you learn how to do this in a reasonable period using a document of reasonable length? How do you *experience* Java EE?

This book helps you answer some of these questions by implementing the Donate sample application scenario:

- *The requirement is to integrate an existing VACATION database (containing an EMPLOYEE table with employee name, number, salary, and available vacation) into a new Donation application that allows employees to donate some of their vacation into a fund that can be used by other employees that need additional time off to deal with family emergencies.*
- *In the first phase this application should be available through a Web interface and a thick Java client, with security constraints to ensure that only authorized users use the application, and that information such as salary is restricted to management. In subsequent phases this application should be available over Web services and JMS (messaging).*

Donation sample application

Figure 1-2 represents the complete application, including the Web front end, the thick client, Web services, and messaging (JMS).

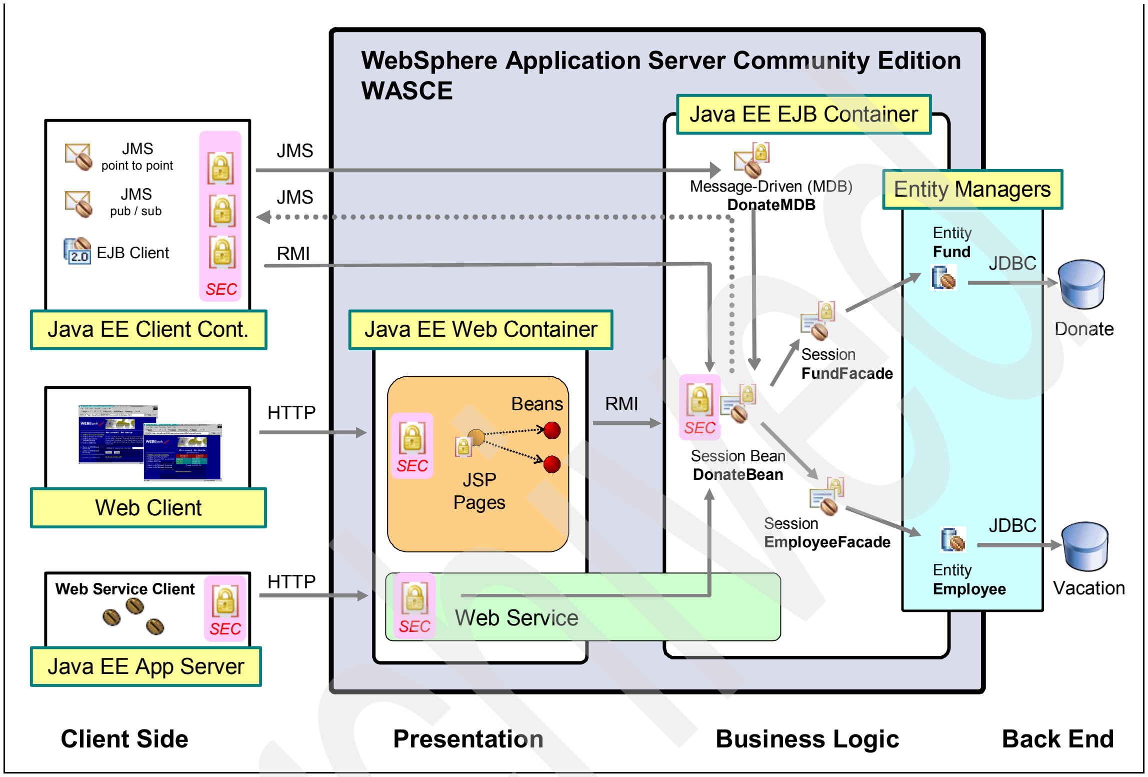


Figure 1-2 Donation sample application scenario

In this book, we implement the application by following the instructions in these parts and chapters:

Part 1, "Preliminary activities" on page 1: Install the required IBM software products and create the initial environment needed to support the application:

- Chapter 2, "Install and configure software" on page 25: Ensure that the prerequisite products are installed.
- Chapter 3, "Configure the development environment" on page 49: Prepare the development workspace and test server.
- Chapter 4, "Prepare the legacy application" on page 77: Extract the application samples needed for this book, and create and populate the Vacation database that represents the current application.

Part 2, "Core Java EE application" on page 97: Create the core Java EE artifacts (data access elements, application business logic elements, Web front end, and thick client):

- Chapter 5, "Create the JPA entities" on page 99: Create the Employee entity EJB (based on the Employee table in the existing Vacation database).
- Chapter 6, "Create the entity facade session beans" on page 165: Create the Funds entity EJB and the Funds table/database (based on the Funds entity EJB).
- Chapter 7, "Create the Donate session bean for the business logic" on page 209: Create the Donate session EJB to transfer vacation from an Employee entity EJB to a Funds entity EJB.
- Chapter 8, "Create the Web front end" on page 221: Create a Web application that provides access first to the EmployeeFacade session EJB, and then to the Donate session EJB.
- Chapter 9, "Create the application client" on page 273: Create a thick Java client that invokes the Donate session EJB.
- Chapter 10, "Implement core security" on page 285: Add the needed users to the user registry (in a file system based registry), configure the test server for Java EE application security, and implement both declarative security (Web and EJB, to restrict access to the overall Web page and EJB) and programmatic security (Web and EJB, to restrict access to the salary field).

Part 3, "Web services" on page 321: Extend the core Java EE application to support Web services:

- Chapter 11, "Create the Web service" on page 323: Use the tooling wizards to expose the Donate session EJB as a Web service, and create a Web service client to access this Web service.
- Chapter 12, "Implement security for the Web service" on page 367: Update the Web service and the Web service client to support Web services security using user ID/password for authentication.

Part 4, "Messaging" on page 403: Extend the core Java EE application to support messaging (JMS):

- Chapter 13, "Create the message-driven bean" on page 405: Create a message-driven bean to receive messages and to call the Donate session EJB. Create a JMS client to send a message to the message-driven bean.
- Chapter 14, "Add publication of results" on page 433: Update the Donate session EJB to publish the donation request results to a topic representing the employee number. Create a JMS client to subscribe to updates for an employee.

- Chapter 15, “Implement security for messaging” on page 445: Update the messaging artifacts to support security.

Part 5, “Advanced Java EE 5 and Web 2.0” on page 453: Describes advanced Java EE and non-Java EE capabilities and technologies that can be used to extend the core application:

- Chapter 16, “Develop a Web 2.0 client” on page 455: Describes how to develop a Web application using Web 2.0 technologies, such as Asynchronous JavaScript and XML (Ajax), Java Script Object Notation (JSON), Representational State Transfer (REST), and the Dojo toolkit.
- Chapter 17, “Advanced deployment with WebSphere Virtual Enterprise” on page 521: Describes how to optimize operations with WebSphere Virtual Enterprise.
- Chapter 18, “Integration with IBM HTTP Server v7.0” on page 531: Describes how to configure the IBM HTTP server as a front-end to WASCE.

1.4 Jump start

Jump start: *to start or restart rapidly or forcefully*

from Merriam-Webster on-line dictionary: `http://www.merriam-webster.com`

The intention is for most users to implement the Experience! sections of this book in a sequential manner: Starting with chapters in Part 1, “Preliminary activities” on page 1 and following in order.

However, readers who are already familiar with parts of Java EE might want to skip chapters or access in a random manner. Or, they might have problems in completing a specific chapter, but would like to continue with rest of the book.

For those readers, this book provides a “jump start” appendix that contains instructions to quickly configure WASCE, the WASCE IDE, and the workspace for each chapter:

- All readers have to complete the initial setup chapters in Part 1, “Preliminary activities” on page 1:
  - Chapter 2, “Install and configure software” on page 25
  - Chapter 3, “Configure the development environment” on page 49
  - Chapter 4, “Prepare the legacy application” on page 77
- Readers then follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for a chapter.

1.5 Linux variations

This book is intended for use with both Windows and Linux, and where needed contains alternate instructions when there will be significant differences in commands and usage on these two operating systems.

- Windows Indicates information unique for Windows
- Linux Indicates information unique for Linux

The most significant differences are in Chapter 2, “Install and configure software” on page 25.

However, this book would be significantly longer and more complex and confusing if it attempted to point out every minor difference between Windows and Linux.

Therefore, to minimize the added complexity, the Experience! sections assume that the reader is using Windows, and the sections are written using the file paths and operating system interactions that exist on Windows.

The most obvious differences between the two operating systems are the base directories used in this book.

Table 1 Directories on Windows and Linux

Path	Windows	Linux
WASCE	`C:\IBM\WASCE21`	`/opt/IBM/WASCE21`
Eclipse Ganymede/ WASCE IDE	`C:\Ganymede\eclipse`	`/opt/Ganymede/eclipse`
IBM Java SDK/JDK™	`C:\Program Files \IBM\Java50`	`/opt/ibm/java2-i386-50`
Samples directory	`C:\7639code`	`$HOME/7639code`
WASCE IDE workspace	`C:\workspaces\ WASCE21expjavaee`	`$HOME/workspaces /WASCE21expjee/`

`$HOME` is the home directory for the Linux user. For example, if the Linux user is `testuser`, the home directory is usually `/home/testuser`. Note that the home directory for root is usually `/root`.

Other differences include these:

- The Linux operating system command line requires the exact file name and normally does not include the current directory on the active search path.

  For example, when invoking the WASCE application client command from the WASCE bin directory in Chapter 9.6, “Test the application client” on page 280 the Windows command is:

  ```
  client ejee/DonateClient/1.0/car 1 DonationFund 1
  ```

  On Linux, you must indicate that the client command is in the current directory (./), and you must provide the command extension (.sh):

  ```
  ./client.sh ejee/DonateClient/1.0/car 1 DonationFund 1
  ```

- When browsing for files using the built-in search pop-ups, the selection button in Windows is **Open** and the selection button in Linux is **OK**.
- In Eclipse, the standard button for expanding and contracting lists (such in the Project Explorer) is a plus and minus sign on Windows, and is a triangle on Linux.

2

Install and configure software

In this chapter, we describe the basic steps to install the software used in this book:

- WebSphere Application Server Community Edition (WASCE v2.1.0.1) with the IBM SDK 5.0
- Eclipse IDE for Java EE Developers (Ganymede release)
- WASCE v2.1.0.1 WTP Server Adapter

2.1 Required software

The following software packages are used in this book:

- WebSphere Application Server Community Edition (WASCE v2.1.0.1) with the IBM SDK 5.0
- Eclipse IDE for Java EE Developers (Ganymede release)
- WASCE v2.1.0.1 WTP Server Adapter

You can skip this chapter if you already have these products installed. However, this book assumes that you install the products into the following directories:

Product	Installation directory used in this book	
WebSphere Application Server Community Edition (WASCE v2.1.0.1)	Windows Linux	`C:\IBM\WASCE21` `/opt/IBM/WASCE21`
Eclipse IDE for Java EE Developers (plus the WASCE v2.1 WTP Server Adapter)	Windows Linux	`C:\Ganymede\eclipse` `/opt/Ganymede/eclipse`

If you install the products to other locations, remember to adjust the various instructions to reflect the differences.

Detailed information about the requirements and supported operating systems for these products can be found at:

- Eclipse IDE for Java EE Developers: `readme_eclipse.html` file (found in the `readme` directory on the installed directory)
- WebSphere Application Server Community Edition v2.1 system requirements:

  http://www-306.ibm.com/software/webservers/appserv/community/sysreq/
- WebSphere Application Server Community Edition v2.1 detailed system requirements:

  http://www-1.ibm.com/support/docview.wss?uid=swg27006834

2.2 Hardware requirements

WASCE v2.1.0.1 and the Eclipse IDE for Java EE developers are supported on a variety of platforms, including x86-32, x86-64, SPARC, and POWER®. Refer to the support links in 2.1, “Required software” on page 26 for detailed information.

This book was developed and validated on x86-32.

The WASCE v2.1.0.1 product documentation states the following system requirements:

- **Disk***: 120 MB to hold the server's installation files, initial configuration, and log files (not including JDK)*
- **Memory***: 128 MB RAM (256 MB or more preferred and depends on deployed applications)*

These requirements refer to the runtime environment only. The actual disk space and memory for the overall environment is dependent on the type of applications that will be deployed.

The Eclipse IDE for Java EE Developers documentation does not list specific resource requirements. The basic installation required 143 MDB of disk, and the first IDE session required 151MB of memory.

Neither component (WASCE v2.1.0.1 and the Eclipse IDE for Java EE developers) list specific CPU recommendations.

However, these are generalized requirements and do not reflect the actual installation and usage that drives the true hardware requirements.

The following description summarizes the hardware recommendations resulting from the creation of this document:

- **Processor**: The scenarios and instructions in this book were developed on a variety of machines, but the main installation was done on a Pentium® 4 with 2.40 GHz processor, and performance was adequate.

  What is the true minimum to complete the scenario and instructions in this book? A Pentium III is probably an acceptable minimum, but with large software applications, the faster the processor, the better.

- **Memory**: 512 MB RAM is an acceptable minimum, but 1 GB is recommended. In this book scenario 2 GB RAM was used.

  The 1 GB RAM recommendation is a good minimum. However, the best value for your environment depends on what else is running on your system.

Therefore, your system should be able to generate acceptable performance when the total physical RAM in use is 512 MB higher than what is currently in use on your system.

For example, if your system is currently using around 300 MB of physical RAM, then in theory you should have at least 800 MB RAM. Because Windows performance degrades significantly when memory begins to swap, you would find performance to be marginal with 756 MB RAM and much better with 1 GB RAM.

- **Disk space**: The actual Eclipse IDE and WASCE v2.1.0.1 maximum disk space usage observed during the development of this book under both Microsoft Windows and Linux was approximately 455 MB.

Component	Directory (Windows)	Disk space
WASCE v2.1.0.1	`C:\IBM\WASCE21`	160 MB
IBM SDK/JRE™	`C:\Program Files\IBM\Java50\`	80 MB
Eclipse IDE for Java EE Developers	`C:\Ganymede\eclipse`	155 MB
WASCE v2.1.0.1 WTP Adapter	`C:\Ganymede\eclipse\plugins` `C:\Ganymede\eclipse\features`	5 MB
Workspace	`C:\Workspaces\WASCE21expjee`	55 MB
Total		**455 MB**

2.3 Software requirements

WASCE v2.1.0.1 and the Eclipse IDE for Java EE developers are supported on a variety of operating systems, including Microsoft Windows (Visa, XP), Linux distributions (Red Hat, SUSE®), AIX, Sun Solaris™, Apple Mac OS X 10.4 (Eclipse IDE only), and several others for WASCE v2.1.0.1. Refer to the support links referred to in 2.1, “Required software” on page 26 for detailed information.

This book was developed and validated on Microsoft Windows XP Professional (Service Pack 2) and Red Hat Enterprise Linux v5 (without updates).

Other requirements beyond the base operating system are as follows:

- Your system must be able to extract ZIP files.
- Your system must have an Internet browser (such as Microsoft Internet Explorer® 5 or higher or Explorer or Mozilla 1.7.5 or later).

- Most of this chapter and all subsequent chapters can be followed using a normal operating system user, with no specific user administrative privileges or roles. The following activities in this chapter do require elevated privileges:
  - Windows Installation of the IBM SDK/JRE in 2.5.1, “Install the IBM SDK/JRE (Windows)” on page 32 must be performed as a user that belongs to the local Administrators group.
  - Linux Installation of the IBM SDK/JRE in 2.5.2, “Install the IBM SDK/JRE (Linux)” on page 35 must be performed as the root user.
  - Linux Creation of the base installation directory for WASCE v2.1.0.1 in 2.5.3, “Install WASCE v2.1” on page 37 must be performed as the root user or a user with read write privileges to `/opt`.
  - Linux Creation of the base installation directory for the Eclipse IDE in 2.5.4, “Install the Eclipse IDE for Java EE Developers” on page 41 must be performed as the root user or a user with read write privileges to `/opt`.

2.4 Obtaining the software

The packages used in this book can be accessed as follows:

- The installation bundle of WebSphere Application Server Community Edition v2.1, with the IBM SDK, is available for download at:

  http://www.ibm.com/developerworks/downloads/ws/wasce/

  Ensure that you select the Server and IBM SDK link for your target operating system:

  - **Server and IBM SDK for Windows:**

    `wasce_ibm150sdk_setup-2.1.0.1-ia32win.zip (152MB)`

  - **Server and IBM SDK for Linux/Intel®:**

    `wasce_ibm150sdk_setup-2.1.0.1-ia32linux.tar.bz2 (138MB)`

Behind the scenes:

You need a valid universal IBM ID and password to download the install bundles. If you do not have a universal IBM ID, you can register for an ID using a link on the download sign in Web site.

Also, the download can be made using standard HTTP or using the Download Director Java Applet. This applet provides enhanced capabilities such as parallel connections (for faster download), restartable downloads (in case the network connection is interrupted), and a check-sum test (to ensure that the file was downloaded correctly.

This link provides more information on the IBM Download Director:

`http://www6.software.ibm.com/dldirector/doc/DDfaq_en.html`

- The installation of Eclipse IDE for Java EE Developers (Ganymede) can be downloaded from the following Web site:

  `http://www.eclipse.org/downloads/`

  Ensure that you download the **Eclipse IDE for Java EE Developers** and not the Eclipse IDE for Java Developers. Note that at the end of September 2008 Ganymede SR1 became available.

  – For Windows, the direct download link to Eclipse Ganymede is:

  http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/ganymede/R/eclipse-jee-ganymede-win32.zip
  http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/ganymede/**SR1**/eclipse-jee-ganymede-**SR1**-win32.zip

  – For Linux, the direct download link to Eclipse Ganymede is:

  http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/ganymede/R/eclipse-jee-ganymede-linux-gtk.tar.gz
  http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/ganymede/**SR1**/eclipse-jee-ganymede-**SR1**-linux-gtk.tar.gz

- The WASCE v2.1.0.1 WTP Server Adapter can be downloaded from the following Web site:

  `http://download.boulder.ibm.com/ibmdl/pub/software/websphere/wasce/updates/`

  This is the direct download link for both Windows and Linux:

  `http://download.boulder.ibm.com/ibmdl/pub/software/websphere/wasce/updates/wasce_eclipse-plugin-2.1.0.1-updatesite.zip`

  Extract the downloaded file into a suitable directory (for example, `C:\WASCE 2.1\adapter` on Windows).

Alternatives:

The WASCE v2.1.0.1 WTP Server Adapter can be installed via four different methods:

- From within the WASCE IDE, using the “Download additional server adapters” link when adding a server definition (automatically downloads over the Internet)
- Using the Eclipse Update Manager by searching against the WASCE Eclipse Update site (automatically downloads over the Internet)
- Using the Eclipse Update Manager by searching against a manually downloaded copy of the adapter update site image (manual download from a Web site)
- By directly expanding a deployable ZIP file in to the subdirectories of the installed Eclipse IDE for Java EE Developers (manual download from a Web site)

Most users typically use one of the first two options, but they will not work if your location has stringent firewall or proxy restrictions.

Therefore, to ensure that the instructions work for as many people as possible, this book uses the third option (manually downloaded copy of the adapter update site image).

Refer to this Web site for details:

```
http://publib.boulder.ibm.com/wasce/V2.1.0/en/eclipse.html#Eclipse-upd
atemgr
```

2.5 Install the software

In this section you install the software components that you downloaded before.

Note that the steps documented here are not the only way to install these products. These specific steps were documented to ensure that they work for as many users as possible, in particular for those users that might not have direct access to the Internet from their system, or who might have limited access due to firewall or proxy issues.

However, users with full Internet access could install in a more direct manner. For example, one alternative path is to:

- Manually download and install the SDK/JRE and the Eclipse IDE for Java EE Developers.

- Use the Eclipse Update Manager to install both WASCE Core Feature v2.1.1 and the WASCE v2.1.0.1 WTP Server Adapter over the Internet from the update site.

The only significant difference is that WASCE v2.1.0.1 would be installed to a different location.

2.5.1 Install the IBM SDK/JRE (Windows)

Perform the following steps to install the IBM SDK/JRE for Windows.

1. Extract the WASCE v2.1.0.1 image downloaded in 2.4, “Obtaining the software” on page 29 into any directory.
2. Log on as a user belonging to the Administrators group.
3. From the root of the extracted image, run `ibm-java2-sdk-50-win-i386.exe`.
4. In the Choose Setup Language pop-up, select the language for the installation and click **OK**.
5. In the Welcome pop-up, click **Next**.
6. In the License Agreement pop-up, click **Yes**.
7. In the Choose Destination Location pop-up, accept the default value (shown here) and click **Next**:
   – Windows `C:\Program Files\IBM\Java50`
8. In the Setup type pop-up, select **Typical**, and click **Next**.
9. In the Install this Java Runtime Environment as the system JVM pop-up, click **Yes**. (This is not required if you already have a system JVM. See the following box, **Behind the scenes**.)
   – If you see the Another Java Runtime Environment is currently installed as the system JVM pop-up, click **Yes**.

Behind the scenes:

WASCE v2.1.0.1 and the WASCE IDE both, by default, configure against the system JVM. For simplicity, this book has you install the IBM SDK/JRE as the system JVM to ensure that the overall scenario steps function as documented.

However, you do not have to install the IBM SDK/JRE as the system JRE, but then you have to ensure that you execute against the correct SDK/JRE. This book does include **SDK/JRE selection alert!** boxes to alert you to those situations.

See the box **IBM SDK/JRE versus Sun SDK/JRE** on the next page for further details.

10. In the Start Copying Files pop-up, click **Next**.
11. If you see a Browser Registration prompt:
    - Optionally select the browsers that you want to associate with this JVM.
    - Click **Next**.

Behind the scenes:

If you select any of the browsers, they will use the IBM SDK/JRE for Java applets.

Applets found in some browser based applications are sensitive to the Java runtime environment™ that they run in. If you want to add IBM SDK as a Java plug-in in the Internet Explorer, make sure that this will not cause any issues for other browser based applications.

12. In the InstallShield Wizard Complete pop-up, click **Finish**.
13. Log out as the user that belongs to the Administrators group, and log in as your normal, non-admin user.

IBM SDK/JRE versus Sun SDK/JRE

WASCE v2.1.0.1 can run on the following Java software development kits (SDK) and Java runtime environments (JRE):

- IBM Java SE5 SR6 or SR8 release (32 bit or 64 bit) (SR8 shipped with WASCE v2.1.0.1)
- Sun Java SE 5 1.5.0 Update 15 or later (32 bit) (must be downloaded separately)

From a technical perspective, both SDKs/JREs fully support the WASCE v2.1.0.1 capabilities. The key difference is support. Per the **WebSphere Application Server Community Edition detailed system requirements** page (reference the link in 2.1, “Required software” on page 26):

- With the IBM SDK/JRE, users are “ *...entitled to support, defect fixes or a reasonable alternative to address a reported problem ...*”.
- With the Sun SDK/JRE, users should except IBM to work with them *“...in a commercially reasonable manner to help solve the customer problem or determine that the problem source is outside of WebSphere Application Server Community Edition...IBM might, at our discretion, defer support to a more current release of WebSphere Application Server Community Edition or choose to reclassify the configuration as “Not Supported.”*
- This book was developed and validated using the IBM SDK/JRE. The material should function with the Sun SDK/JRE, but this has not been validated.

The only hard dependency on the IBM SDK/JRE is the callback handler used in 10.7.2, “Update the DonateClient Geronimo deployment descriptor” on page 315:

```
com.ibm.security.auth.callback.DialogCallbackHandler
```

This class is only included in the IBM SDK/JRE. With the Sun SDK/JRE, you can instead use:

```
com.sun.security.auth.callback.DialogCallbackHandler
```

2.5.2 Install the IBM SDK/JRE (Linux)

Perform the following steps to install the IBM SDK/JRE on Linux:

1. Extract the WASCE v2.1.0.1 image downloaded in 2.4, “Obtaining the software” on page 29 into any directory using the Linux tar command.

   For example, run the **tar** command to extract the image from the /src directory to the /tmp/img/wasce21 directory:

   ```
   tar -xvf /tmp/img/wasce_ibm150sdk_setup-2.1.0.1-ia32linux.tar.bz2 -C
   /tmp/img/fred
   ```

2. Log on as the root user, either as a full logon or by using the **su** command to switch to the root user.

3. From the root of the extracted image, install the IBM SDK/JRE using the Linux **rpm** command:

   ```
   rpm -ivh ibm-java2-i386-sdk-5.0-8a.0.i386.rpm
   ```

   The command should complete with output similar to the following:

   ```
   Preparing... ########################################### [100%]
      1:ibm-java2-i386-sdk ######################################### [100%]
   ```

4. (Optional) the installation can be verified by running the following command with the resulting output:

   ```
   [root@localhost Server]# rpm -qa | grep "ibm-java2"
   ibm-java2-i386-sdk-5.0-8a.0
   ```

5. Log out as the root user, and log in as non-root user.

Off course?

If the **rpm** installation command fails with output similar to this:

```
error: Failed dependencies:
        libstdc++.so.5 is needed by ibm-java2-i386-sdk-5.0-8a.0.i386
        libXp.so.6 is needed by ibm-java2-i386-sdk-5.0-8a.0.i386
```

Then the Linux operating system does not contain needed prerequisites. This specific error occurred during the development of this book on a system installed with the standard installation package of Red Hat Enterprise Linux V5 (no updates).

As an example, the following steps were used to install the required dependencies, after which the **rpm** install of the IBM SDK/JRE succeeded.

- The **su** root command was issued to switch to root (and the root password entered at the prompt).
- The RHEL V5 DVD was inserted into the CD/DVD ROM drive.
- The DVD was mounted:

```
[root@localhost /]# mount /dev/cdrom /media/cdrom
mount: block device /dev/cdrom is write-protected, mounting
read-only
```

- The command prompt was switched to the directory with the RPM files:

```
[root@localhost /]# cd /media/cdrom/Server/
```

- The first package was installed:

```
[root@localhost Server]# rpm -ivh libXp-1.0.0-8.i386.rpm
warning: libXp-1.0.0-8.i386.rpm: Header V3 DSA signature: NOKEY, key
ID 37017186
Preparing...
########################################### [100%]
1:libXp ########################################### [100%]
```

- The second package was installed:

```
[root@localhost Server]# rpm -ivh
compat-libstdc++-33-3.2.3-61.i386.rpm
warning: compat-libstdc++-33-3.2.3-61.i386.rpm: Header V3 DSA
signature: NOKEY, key ID 37017186
Preparing...
########################################### [100%]#
1:compat-libstdc++-33 ###################################### [100%]
```

- The root user was logged out (by typing exit).

2.5.3 Install WASCE v2.1

Perform the following steps to install WASCE v2.1.0.1:

1. Linux Log on as the root user, and create a directory under which the WASCE v2.1.0.1 package can be installed by the non-root user.
   - Run the following command to create the /opt/IBM directory:

     ```
     mkdir /opt/IBM
     ```

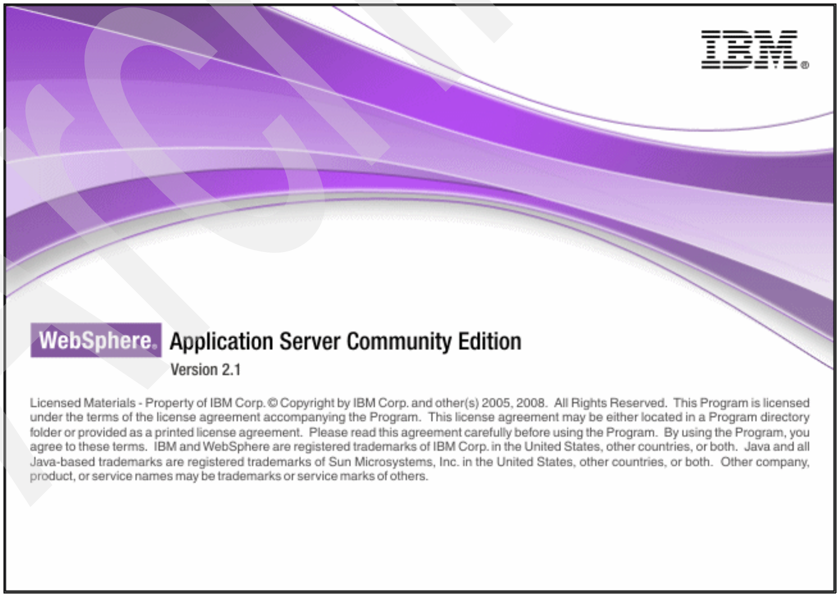
   - Run the following command to transfer ownership of this directory to the `users` group (or to whichever group your non-root user belongs to):

     ```
     chgrp users /opt/IBM
     ```

   - Run the following command to allow members of the users group to create and make changes inside of `/opt/IBM`:

     ```
     chmod g+w /opt/IBM
     ```

   - Log out as the root user, and log in as non-root user.
2. From the root of the extracted WASCE v2.1.0.1 image (extracted in the previous section), run:
   - Windows `wasce_setup-2.1.0.1-win.exe`
   - Linux `wasce_setup-2.1.0.1-unix.bin`
3. The installation starts with a splash screen.



WebSphere. Application Server Community Edition

Version 2.1

Licensed Materials - Property of IBM Corp. © Copyright by IBM Corp. and other(s) 2005, 2008. All Rights Reserved. This Program is licensed under the terms of the license agreement accompanying the Program. This license agreement may be either located in a Program directory folder or provided as a printed license agreement. Please read this agreement carefully before using the Program. By using the Program, you agree to these terms. IBM and WebSphere are registered trademarks of IBM Corp. in the United States, other countries, or both. Java and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

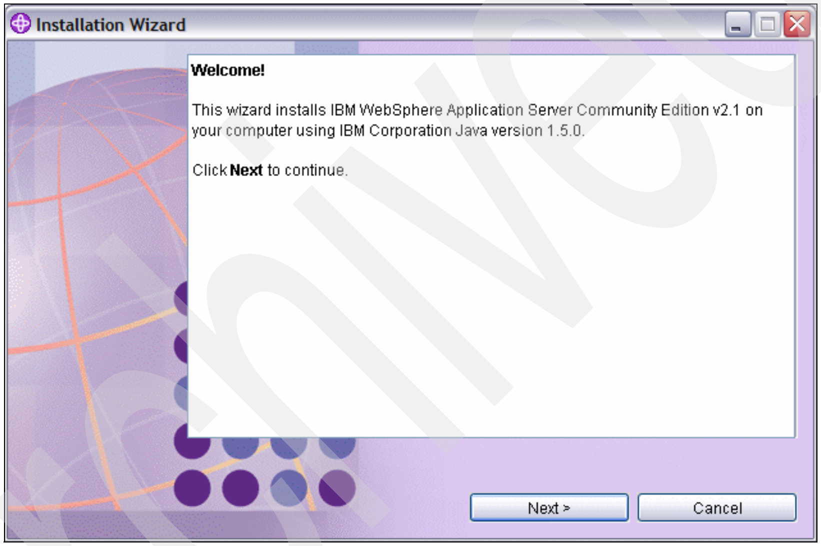
Off course?

The installer looks for a compatible Java JRE. If it finds one, the installation continues. If you get a message indicating that a suitable JVM cannot be found, ensure that you have installed a compatible SDK and use the `-is:javahome` option to start the installer.

```
wasce_setup-2.1.0.0-win.exe -is:javahome C:\<Java_Home>
```

Click **Next** to continue.

4. In the Welcome pop-up, click **Next**.

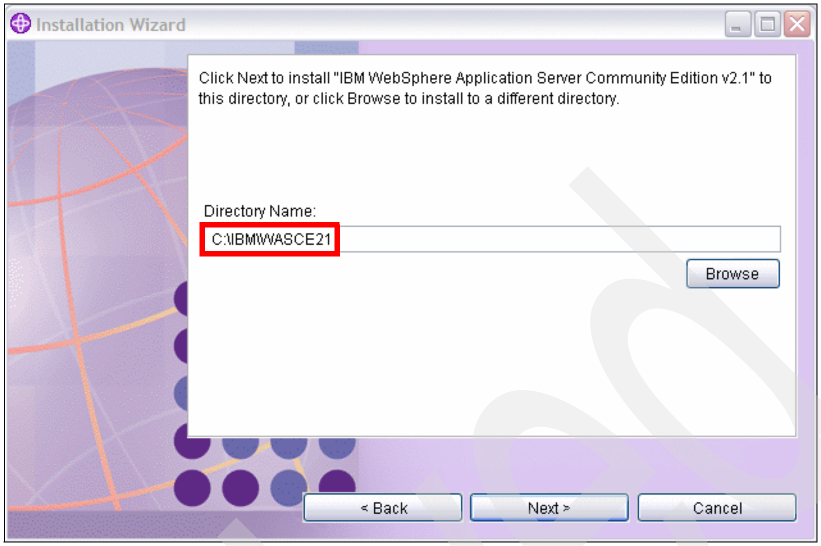


Welcome!

This wizard installs IBM WebSphere Application Server Community Edition v2.1 on your computer using IBM Corporation Java version 1.5.0.

Click **Next** to continue.

5. In the **Software License Agreement** pop-up, select **I accept the terms in the license agreement**, and click **Next**.
6. In the Click Next to install... pop-up, set the Directory Name as shown here, and click **Next**.
   - Windows `C:\IBM\WASCE21`
   - Linux `/opt/IBM/WASCE21`

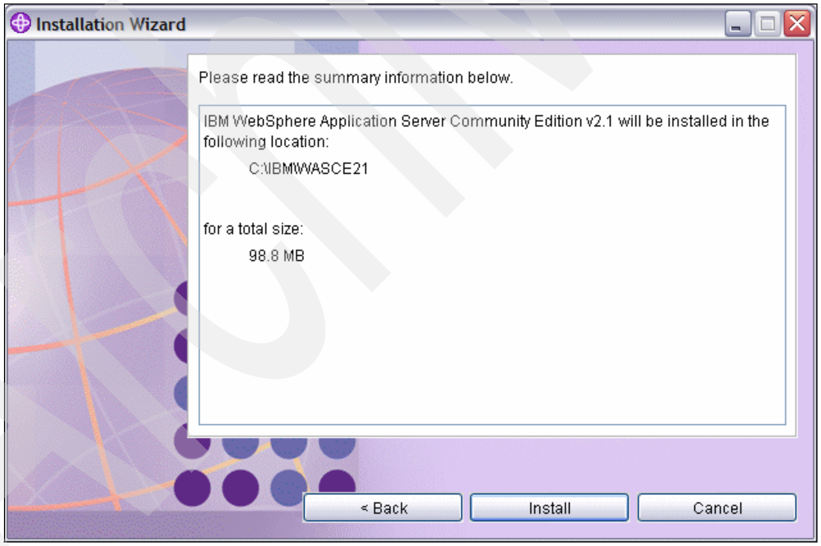


Click Next to install "IBM WebSphere Application Server Community Edition v2.1" to this directory, or click Browse to install to a different directory.

Directory Name:

C:\IBM\WASCE21

7. In the Please read the summary information below pop-up, click **Install**.



Please read the summary information below.

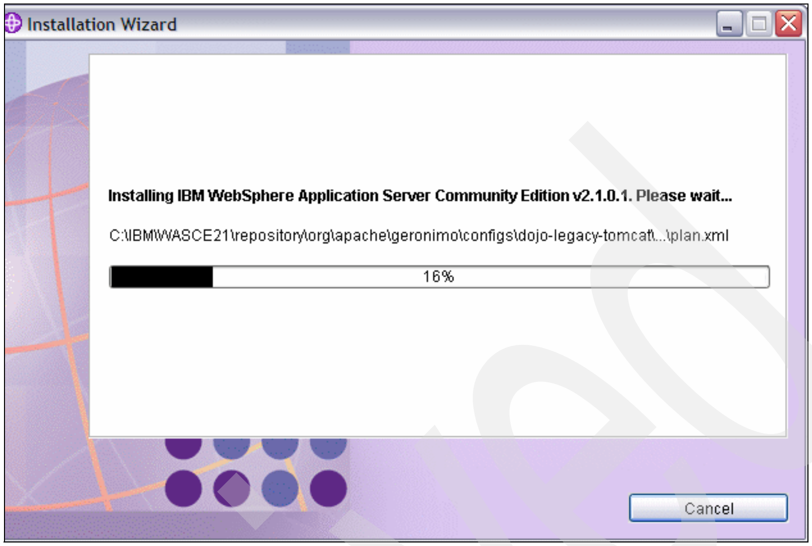
IBM WebSphere Application Server Community Edition v2.1 will be installed in the following location:

C:\IBM\WASCE21

for a total size:

98.8 MB

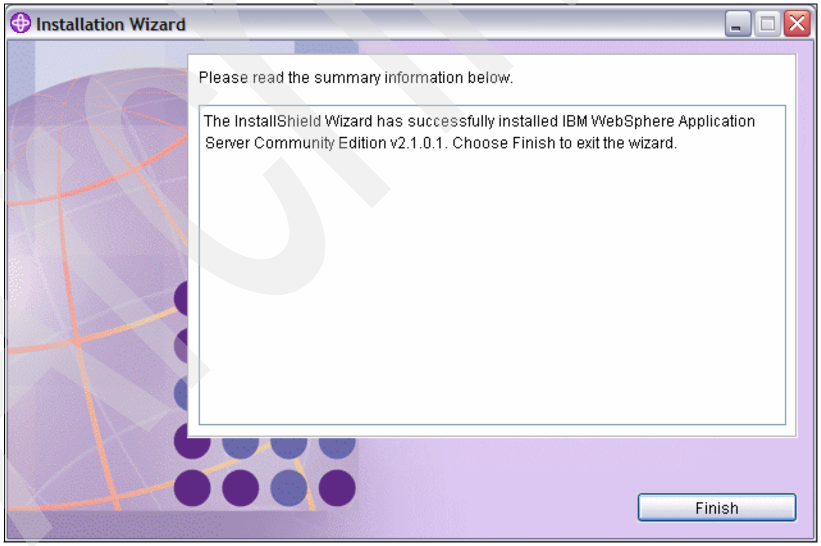
8. The installation starts and displays progress information.



Installing IBM WebSphere Application Server Community Edition v2.1.0.1. Please wait...

C:\IBM\WASCE21\repository\org\apache\geronimo\configs\dojo-legacy-tomcat\...\plan.xml

9. The final window indicates the success of the installation. Click **Finish** to complete the installation.



Please read the summary information below.

The InstallShield Wizard has successfully installed IBM WebSphere Application Server Community Edition v2.1.0.1. Choose Finish to exit the wizard.

SDK/JRE selection alert!

WASCE v2.1.0.1 should automatically detect and configure against the installed IBM SDK/JRE. You can verify this by checking for the following lines in `C:\IBM\WASCE21\bin\setenv.bat`:

```
@REM First instance of our WASCE's java home variable is initialized by
the installer...
set WASCE_JAVA_HOME=C:\Program Files\IBM\Java50
```

If there is a different value (such as `C:\Program Files\Java\jdk1.5.0_15`), then the installation program has detected and configured against the Sun SDK/JRE.

As discussed previously, this is a supported configuration because WASCE v2.1.0.1 is supported on the Sun SDK/JRE, but for the purpose of this book, we recommend that you change it to point to the IBM SDK/JRE.

2.5.4 Install the Eclipse IDE for Java EE Developers

For this document we used **Eclipse Ganymede**, which is an annual release of Eclipse projects. Like the 2007 Europa release, the 2008 Ganymede release is a coordinated release of different Eclipse project teams. Eclipse Ganymede includes 23 projects. By releasing these projects at the same time, the goal is to eliminate uncertainty about version compatibility and make it easier to incorporate multiple projects into your environment.

Perform the following steps to install the Eclipse IDE for Java EE Developers:

1. Linux Log on as the root user, and create a directory under which the WASCE v2.1.0.1 package can be installed by the non-root user.

   – Run the following command to create the /opt/IBM directory:

   ```
   mkdir /opt/Ganymede
   ```

   – Run the following command to transfer ownership of this directory to the `users` group (or to whichever group your non-root user belongs to):

   ```
   chgrp users /opt/Ganymede
   ```

   – Run the following command to allow member of the users group to create make changes inside of `/opt/Ganymede`:

   ```
   chmod g+w /opt/Ganymede
   ```

2. Extract the Eclipse IDE for Java EE Developers image downloaded in 2.4, “Obtaining the software” on page 29 into the following directory:

   – Windows C:\Ganymede
   – Linux /opt/Ganymede

   The Eclipse IDE ZIP file contains a root directory of \eclipse, so all the extracted files will be in C:\Ganymede\eclipse for Windows and /opt/Ganymede/eclipse for Linux.

3. [Optional, but recommended] create a short cut to start the Eclipse IDE:

   – Windows C:\Ganymede\eclipse\eclipse.exe
   – Linux opt/Ganymede/eclipse/eclipse

2.5.5 Install the WASCE Server Adapter v2.1

Perform the following steps to install the WASCE v2.1.0.1 WTP Server Adapter. This feature allows you to develop and test applications for WASCE v2.1.0.1 from within the Eclipse IDE:

1. Extract the WASCE v2.1.0.1 WTP Server Adapter image downloaded in 2.4, “Obtaining the software” on page 29 into any directory.

   Note that this image is packaged as a ZIP file, and the same file is used for both Windows and Linux. Therefore, on Linux you will typically use the **unzip** command instead of the **tar** command.

2. Launch the Eclipse IDE either using the shortcut created in the previous section, or by opening an operating system command prompt and executing the following command:

   – Windows C:\Ganymede\eclipse\eclipse.exe
   – Linux /opt/Ganymede/eclipse/eclipse

Off course?

If the **eclipse** command executes with no output, and the workspace prompt described in the next step never appears, then Eclipse is unable to locate the SDK. This might occur if you did not install the IBM SDK/JRE as the system JVM in 2.5.1, "Install the IBM SDK/JRE (Windows)" on page 32.

Or, if an incompatible JVM is detected, you might see this error:

```
Version 1.4.2 of the JVM is not suitable for this product. Version:
1.5 or greater is required.
```

In either case, the workaround is to invoke Eclipse with an optional command line parameter that identifies the SDK/JRE home directory:

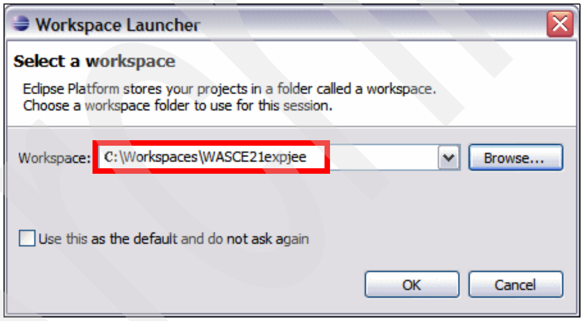
- Windows :

  ```
  C:\Ganymede\eclipse\eclipse.exe -vm "c:\Program Files\IBM\Java50\bin"
  ```

- Linux :

  ```
  /opt/Ganymede/eclipse/eclipse  -vm /opt/ibm/java2-i386-50/bin
  ```

3. In the Workspace Launcher pop-up:



Workspace Launcher

Select a workspace

Eclipse Platform stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Workspaces\WASCE21expjee

☐ Use this as the default and do not ask again

– Set the Workspace path as follows (for example):

  - Windows `C:\workspaces\WASCE21expjee`
  - Linux `$HOME/workspaces/WASCE21expjee`

  Where `$HOME` is the home directory for the Linux user. For example, if the Linux user is testuser, the home directory is usually `/home/testuser`. Note that the home directory for root is usually `/root`.

– Do **NOT** select **Use this as the default and do not ask again**.

– Click **OK**.

SDK/JRE selection alert!

If you did not install the IBM SDK/JRE as the system JVM, you must specify the SDK/JRE home directory as described in the **Off course?** box.

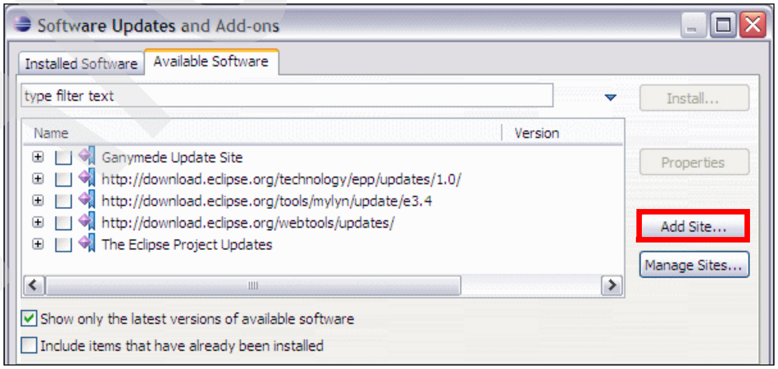
If you did not do this, and the workspace prompt still appeared, this means that you have another SDK/JVM installed as the system JVM. This might or might not cause issues as you follow this book. Our recommendation is that you start Eclipse using the IBM SDK/JRE.

If you are unsure which SDK/JRE Eclipse is using, you can check when Eclipse is started as follows:

- From the WASCE IDE action bar, select **Help** → **About Eclipse Platform**.
- In the About Eclipse Platform pop-up, click **Configuration Details**.
- In the Configuration Details pop-up, scroll down and look for the `java.home` variable definition. For Windows, the value would be:
  - IBM SDK/JRE: `java.home=c:\Program Files\IBM\Java50\jre`
  - Sun SDK/JRE: `java.home=C:\Program Files\Java\jre1.5.0_15`

  In 3.4, "Configure the IBM SDK/JRE" on page 57, we describe how to configure the IBM SDK as the default for Eclipse Ganymede.

4. In the resulting Eclipse IDE editor action bar, select **Help** → **Software Updates**.
5. In the Software Updates and Add-ons pop-up:
   - Switch to the **Available Software** tab, and click **Add Site**.

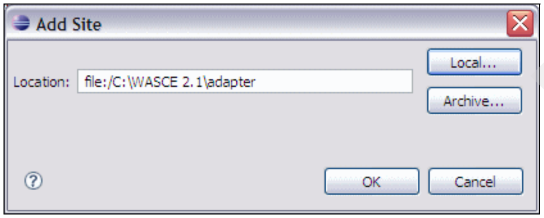


Name	Version
Ganymede Update Site	
http://download.eclipse.org/technology/epp/updates/1.0/	
http://download.eclipse.org/tools/mylyn/update/e3.4	
http://download.eclipse.org/webtools/updates/	
The Eclipse Project Updates	

☑ Show only the latest versions of available software

☐ Include items that have already been installed

- In the Add Site pop-up, click **Local**, and in the resulting pop-up, browse to the directory where you extracted the WASCE v2.1.0.1 WTP Server Adapter image, and click **OK**.



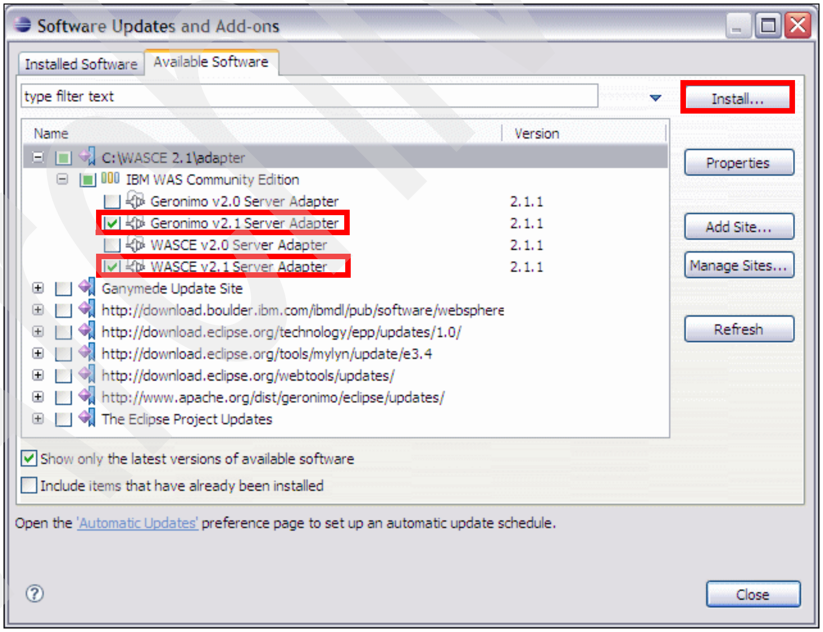
Add Site

Location: file:/C:\WASCE 2.1\adapter

- If you are not connected to the Internet, you might get an error pop-up. Click **OK**.

6. Back in the Software Updates and Add-ons pop-up:

   - Expand the feature tree to <adapter directory> → **IBM WAS Community Edition**, and select both **Geronimo V2.1 Server Adapter** and **WASCE v2.1 Server Adapter.**
   - On the right, click **Install**.



Software Updates and Add-ons

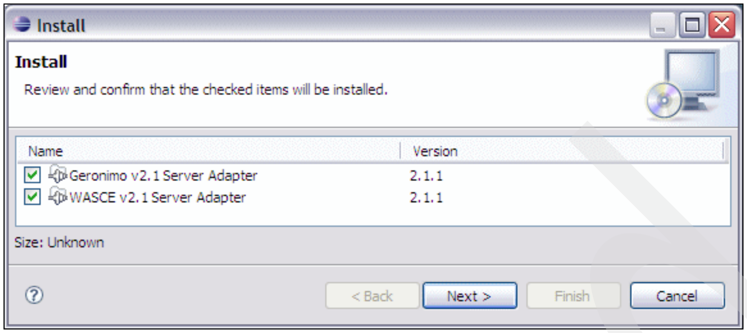
Name	Version
⊟ ■ C:\WASCE 2.1\adapter	
⊟ ■ IBM WAS Community Edition	
□ Geronimo v2.0 Server Adapter	2.1.1
☑ Geronimo v2.1 Server Adapter	2.1.1
□ WASCE v2.0 Server Adapter	2.1.1
☑ WASCE v2.1 Server Adapter	2.1.1
⊞ □ Ganymede Update Site	
⊞ □ http://download.boulder.ibm.com/ibmdl/pub/software/websphere	
⊞ □ http://download.eclipse.org/technology/epp/updates/1.0/	
⊞ □ http://download.eclipse.org/tools/mylyn/update/e3.4	
⊞ □ http://download.eclipse.org/webtools/updates/	
⊞ □ http://www.apache.org/dist/geronimo/eclipse/updates/	
⊞ □ The Eclipse Project Updates	

☑ Show only the latest versions of available software

□ Include items that have already been installed

Open the 'Automatic Updates' preference page to set up an automatic update schedule.

7. In the Install pop-up, verify that both adapters are in the list, and click **Next**.



Install

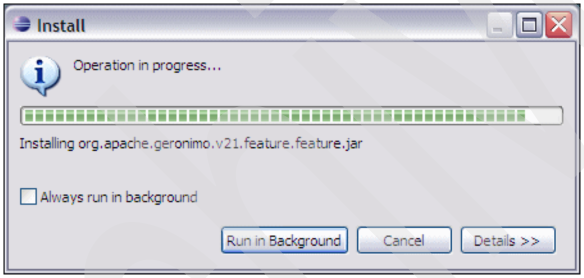
Review and confirm that the checked items will be installed.

Name	Version
☑ Geronimo v2.1 Server Adapter	2.1.1
☑ WASCE v2.1 Server Adapter	2.1.1

Size: Unknown

8. In the Review Licenses pop-up, select **I accept the terms and conditions of the license agreements**, and click **Finish**.

9. Wait for the Install pop-up to complete.

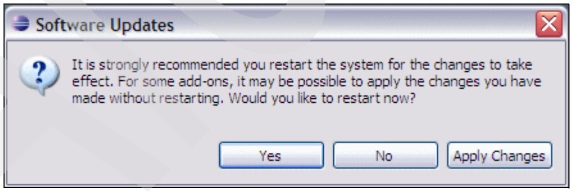


Operation in progress...

Installing org.apache.geronimo.v21.feature.feature.jar

☐ Always run in background

10. At the Software Updates pop-up, click **Yes** to restart the Eclipse IDE. This only restarts the Eclipse IDE, and does not reboot the computer.



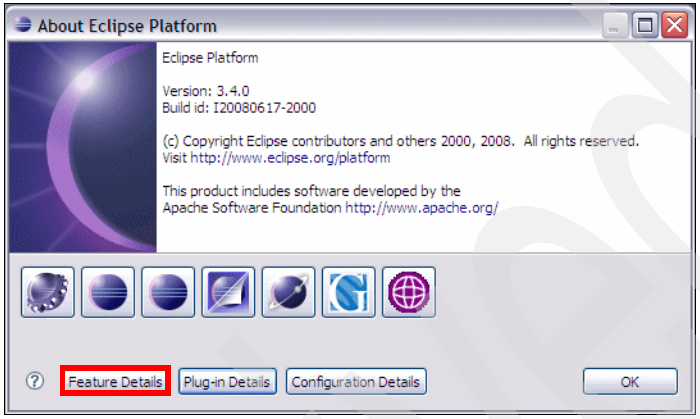
It is strongly recommended you restart the system for the changes to take effect. For some add-ons, it may be possible to apply the changes you have made without restarting. Would you like to restart now?

11. In the Select a workspace prompt, click **OK**.

2.5.6 Verify the installation

Next we verify the installed features and plug-ins:

1. From the Eclipse IDE editor action bar, select **Help** → **About Eclipse Platform**.



About Eclipse Platform

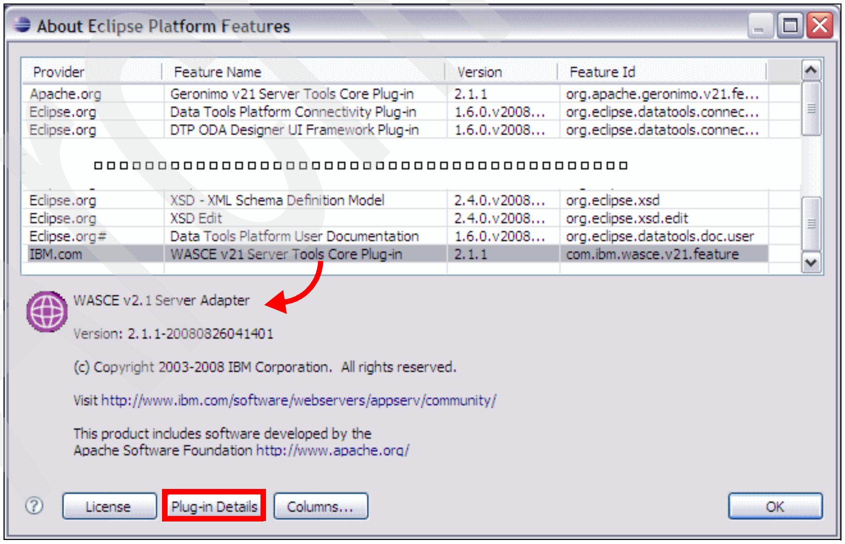
Eclipse Platform

Version: 3.4.0
Build id: I20080617-2000

(c) Copyright Eclipse contributors and others 2000, 2008. All rights reserved.
Visit http://www.eclipse.org/platform

This product includes software developed by the
Apache Software Foundation http://www.apache.org/

2. From the resulting About Eclipse Platform pop-up, click **Feature Details**. The Geronimo v21 Server Tools Core Plug-in is listed at the top, and the WASCE v21 Server Tools Core Plug-in is listed at the bottom.



About Eclipse Platform Features

Provider	Feature Name	Version	Feature Id
Apache.org	Geronimo v21 Server Tools Core Plug-in	2.1.1	org.apache.geronimo.v21.fe...
Eclipse.org	Data Tools Platform Connectivity Plug-in	1.6.0.v2008...	org.eclipse.datatools.connec...
Eclipse.org	DTP ODA Designer UI Framework Plug-in	1.6.0.v2008...	org.eclipse.datatools.connec...
Eclipse.org	XSD - XML Schema Definition Model	2.4.0.v2008...	org.eclipse.xsd
Eclipse.org	XSD Edit	2.4.0.v2008...	org.eclipse.xsd.edit
Eclipse.org#	Data Tools Platform User Documentation	1.6.0.v2008...	org.eclipse.datatools.doc.user
IBM.com	WASCE v21 Server Tools Core Plug-in	2.1.1	com.ibm.wasce.v21.feature

WASCE v2.1 Server Adapter

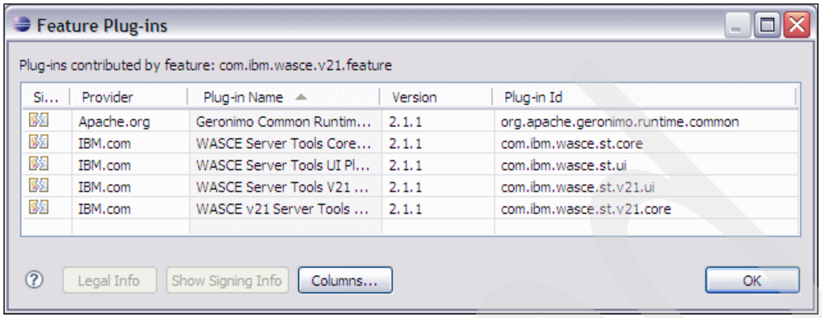
Version: 2.1.1-20080826041401

(c) Copyright 2003-2008 IBM Corporation. All rights reserved.

Visit http://www.ibm.com/software/webservers/appserv/community/

This product includes software developed by the
Apache Software Foundation http://www.apache.org/

3. From the About Eclipse Platform Features pop-up, scroll down and select one of the two features mentioned previously. Click **Plug-in Details** to open the Feature Plug-ins pop-up with the details of the feature.



Plug-ins contributed by feature: com.ibm.wasce.v21.feature

Si...	Provider	Plug-in Name	Version	Plug-in Id
	Apache.org	Geronimo Common Runtim...	2.1.1	org.apache.geronimo.runtime.common
	IBM.com	WASCE Server Tools Core...	2.1.1	com.ibm.wasce.st.core
	IBM.com	WASCE Server Tools UI Pl...	2.1.1	com.ibm.wasce.st.ui
	IBM.com	WASCE Server Tools V21 ...	2.1.1	com.ibm.wasce.st.v21.ui
	IBM.com	WASCE v21 Server Tools ...	2.1.1	com.ibm.wasce.st.v21.core

4. You can also click **Plug-in Details** in the About Eclipse Platform pop-up to list all the plug-ins.
5. Click **OK** to close the About Eclipse Platform pop-up.

Behind the scenes:

In Eclipse terminology, a plug-in is a specific function or capability, and a feature is a group of plug-ins. For the WASCE v2.1.0.1 WTP Server Adapter, you installed two features (Geronimo v2.1 Server Adapter, and the WASCE v2.1 Server Adapter) each with a set of plug-ins.

These features provide support for:

- WASCE Deployment Plan Editor
- Ability to create a WASCE server, and manage its state and deploy applications to it
- Project facet for WASCE applications (WASCE Deployment 1.2)

2.6 Post configuration in Eclipse

After installing the server adapter, it is necessary to configure the WASCE v2.1.0.1 server runtime in Eclipse. We do this when we configure the development environment in Chapter 3, “Configure the development environment” on page 49.

3

Configure the development environment

In this chapter, we configure the WASCE IDE development environment to support the Experience Java EE! scenario.

3.1 Learn!

The WASCE IDE is a combination of features that together provide an integrated development environment (IDE) for WASCE (Table 3-1). By the documentation, WASCE requires:

- Eclipse V3.4 Classic SDK
- Eclipse Web Tools Platform (WTP) V3.0
- Eclipse Modeling Framework (EMF) V2.4
- Eclipse Graphical Editing Framework (GEF) V3.4
- Eclipse Data Tools Platform (DTP) V1.6
- WebSphere Application Server Community Edition Server Adapter

The last component is provided with WASCE. All others are open source downloads, and for this book are downloaded in one integrated package called the **Eclipse IDE for Java EE Developers**. This package is given a different name each year. The 2007 release was called *Europa*, and the 2008 release that this book uses is called ***Ganymede***.

Table 3-1 Features of Eclipse IDEs

Feature	Eclipse Classic	Web Tools Platform (WTP)	Eclipse IDE for Java EE Developers (Ganymede)
Core Eclipse	Y	Y	Y
CVS	Y	Y	Y
Java Development Tools (JDT)	Y	Y	Y
Plug-in Development Environment (PDE)	Y	Y	Y
Web Standard Tools (WST)		Y	Y
Java EE Tools (JEE)		Y	Y
Eclipse Modeling Framework (EMF)			Y
Graphical Editing Framework (GEF)			Y
Data Tools Platform (DTP)		Y	Y
Mylan			Y

WebSphere Application Server Community Edition Server Adapter is a component of the WebSphere Application Server Community Edition v2.1 software package and it installs into an Eclipse environment to provide a basic integrated development environment (IDE) for WASCE.

WASCE IDE is based on evolving open-standard IDE initiatives:

- **Eclipse** (v3.4) provides the overall IDE framework, Java (Java 5 or higher) development, and test tools. The following quote is from the WASCE IDE online help:

  “*Eclipse is an award-winning, open source platform for the construction of powerful software development tools and rich desktop applications. Leveraging the Eclipse plug-in framework to integrate technology on the desktop saves technology providers time and money by enabling them to focus their efforts on delivering differentiation and value for their offerings. Eclipse is a multi-language, multi-platform, multi-vendor supported environment that is built by an open source community of developers and it is provided royalty-free by the Eclipse Foundation. Eclipse is written in the Java language, includes extensive plug-in construction toolkits and examples, and can be extended and deployed on a range of desktop operating systems including Windows, Linux, QNX and Macintosh OS X.*”

- **Web Tools Platform** (WTP) (v3.0) provides Eclipse-based tooling for developing Java EE applications. The following quote is from the Web Tools Platform Web site:

  “*The Eclipse Web Tools Platform (WTP) project extends the Eclipse platform with tools for developing Web and Java EE applications. It includes source and graphical editors for a variety of languages, wizards, and built-in applications to simplify development, and tools and APIs to support deploying, running, and testing apps.*”

- **WASCE Server Adapter** (v2.1) provides the linkage between the WASCE IDE and the WASCE server. This enables a developer to start and stop the server, run WASCE administrative scripts, and to deploy application to the server.

The combined Eclipse, WTP, and WASCE Server Adapter package provides both development and test capabilities:

- Users can create and edit Java EE specific artifacts.
- Users can test the artifacts against Java EE test servers (in this case WASCE).

The user sees a single-integrated product and is generally not aware or not concerned about which component a specific feature or function came from.

Both Eclipse and WTP were seeded with code submissions from IBM that were based on actual IBM products:

- Eclipse was based on the tooling platform initially released with WebSphere Studio Application Developer v4.x.
- WTP was based on the J2EE tooling in Rational Application Developer v6.x.

Additional Learn resources are available at the following Web sites:

- Eclipse.org home page:

  `http://www.eclipse.org`

- Eclipse Web Tools Platform Project:

  `http://www.eclipse.org/webtools`

3.2 Extract the sample code file

The `7639code.zip` file (refer to Appendix B, "Additional material" on page 571 for download instructions) contains various directories and files that are used in the scenarios (Table 3-2).

Table 3-2 Sample code content: 7639code.zip

Directory or file	Description
`7639code`	Master directory
`../jumpstart`	Zip files that can be used to set up the projects in the WASCE IDE for the various development chapters.
`../plans`	Deployment plans for the WASCE server to set up the database pools and the JMS configuration.

`../snippets`
`experiencejavaee_snippets.xml`
	Code snippets used to create various portions of the Java EE application. The snippets are available in text format for copy/paste, and in an XML file that can be imported into the WASCE IDE (see 3.7, "Import the sample code snippets" on page 69).

Extract the samples file **7639code.zip**:

Windows Unzip to `C:\` to create `C:\7639code`.

Linux Unzip to `$HOME/` to create `$HOME/7639code`.

`$HOME` is the home directory for the Linux user. For example, if the Linux user is `testuser`, the home directory is usually `/home/testuser`.

3.3 Create the WASCE IDE workspace

The WASCE IDE (and any Eclipse based product) operates against a collection of resources called a workspace that provides a contained and isolated environment. A workspace typically contains the following components:

- **Resources**, which represent specific artifacts that the developer will create and test.

  A resource can represent a discreet file, such as a Java class or an XML data file, or an abstract construct, such as an EJB or a Web service.

- **Projects** that allow the grouping of resources.

  Projects can be simple, which means they can contain any artifact, or they can be specific to a technology or standard such as an EJB project or an enterprise application project.

- **Capabilities** to manipulate the various projects and resources.

  Capabilities are provided through plug-ins. Products such as the WASCE IDE come with a built-in set of plug-ins, but you can also download and install plug-ins from other sources as well. The Eclipse Foundation manages the following plug-in download site:

  `http://www.eclipseplugincentral.com`

- **Test capabilities** that allow the testing and debugging of projects and resources.

  These can include built-in language environments such as Java, standard transformations such as XSL, and specific runtime test environments such as Java EE application servers.

A workspace is implemented as a file system subdirectory structure, but it does allow you to link in artifacts that exist in other locations in the file system.

In this section we create a single workspace to use for all scenarios in this book:

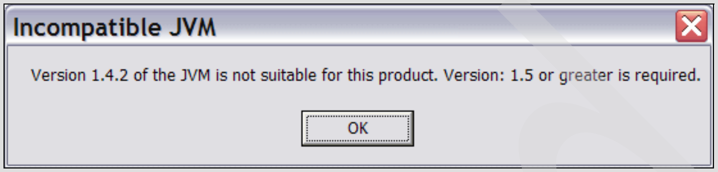
1. The WASCE IDE should already be running as a result of the steps completed in 2.5.5, “Install the WASCE Server Adapter v2.1” on page 42. However, if it is not, you can start it as follows:

   – Open an operating system command prompt and execute the following command:

     - Windows `C:\Ganymede\eclipse\eclipse.exe`
     - Linux `/opt/Ganymede/eclipse/eclipse`

Off course?

If Eclipse fails to initialize (if the Workspace Launcher pop-up described here never starts), or if Eclipse fails with an error pop-up indicating an incompatible JRE:



Incompatible JVM

Version 1.4.2 of the JVM is not suitable for this product. Version: 1.5 or greater is required.

Try running the command with the VM argument to explicitly define the Java runtime location:

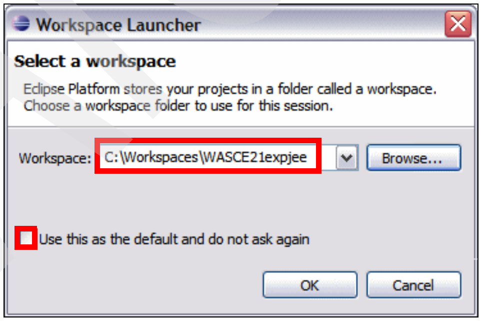
Windows `eclipse -vm "c:\Program Files\IBM\Java50\bin"`

Linux `eclipse -vm /opt/ibm/java2-i386-50/bin`

– In the Workspace Launcher pop-up:

- Set the Workspace path as follows:
  - Windows `C:\workspaces\WASCE21expjee`
  - Linux `$HOME/workspaces/WASCE21expjee`

  Note: Accepting the default workspace on Windows under *Document and Settings* causes problems when running some wizards (such as creating a Web service client) due to spaces in the directory path.
- Do **NOT** select **Use this as the default and do not ask again**.
- Click **OK**.



Workspace Launcher

Select a workspace

Eclipse Platform stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Workspaces\WASCE21expjee

□ Use this as the default and do not ask again

If the specified folder does not exist, Eclipse creates it and initializes the workspace by adding a `.metadata` folder, where metadata information will be stored.

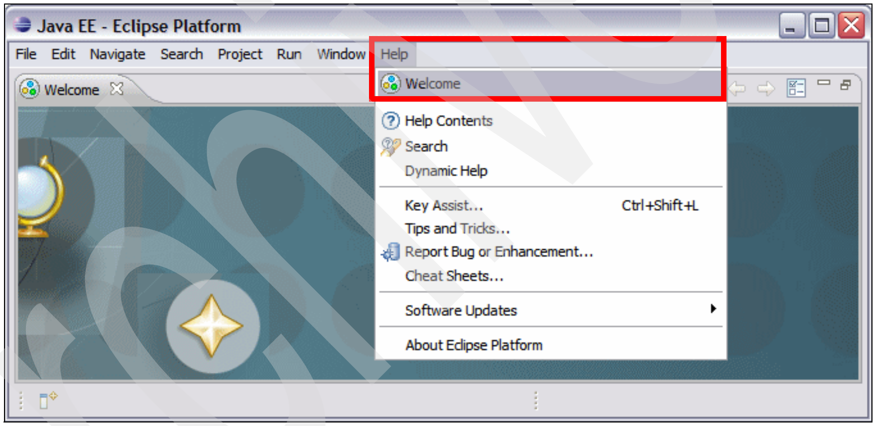
Behind the scenes:

You did not select **Use this workspace as the default and do not ask again**, because it would do exactly what it says: It would no longer give you the option to set the workspace directory.

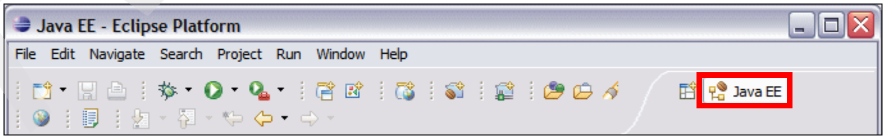
If you accidentally set this, you can re-enable the workspace prompt:

- From the WASCE IDE, select **Window** → **Preferences**.
- In the Preferences pop-up, select **General** → **Startup and Shutdown**, then select **Prompt for workspace on startup** on the right side, and click **OK**.

2. After the workspace initializes, close the **Welcome** view by clicking the **X** next to Welcome. Note that you can open the Welcome view at any time by selecting **Help** → **Welcome** from the menu bar.



3. The workspace title bar in the upper left (shown above) and the perspective icon in the upper right (shown below) both indicate the current perspective. With the Ganymede distribution, the default perspective is **Java EE**.



Behind the scenes:

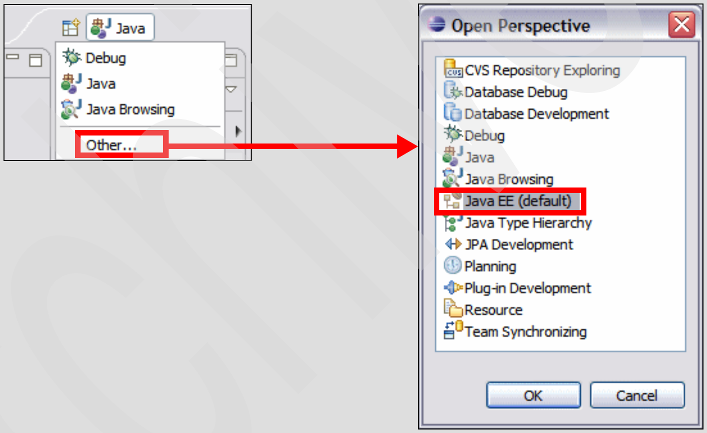
Eclipse-based products contain one or more perspectives that contain a set of default views and capabilities. Each perspective is tailored for a specific development role (in this case Java EE development), only showing views and capabilities that apply for that role.

Typical perspectives used for this document are Java EE, Web, and Java.

Off course?

If the current perspective is not **Java EE**, you can switch to the perspective through two different options:

- Click the **Perspective** icon [ ], located at the upper right, and select **Other** from the pull-down.



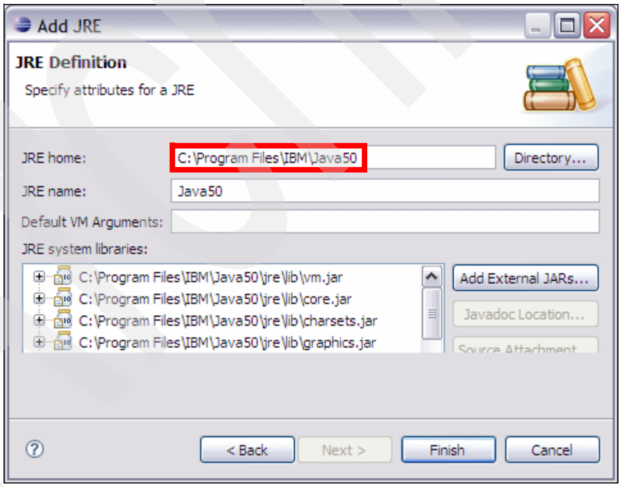
  – In the Open Perspective pop-up, select **Java EE (default)** and click **OK**.

- In the action bar, select **Window** → **Open Perspective** → **Other**, and in the Open Perspective pop-up, select **Java EE**.

3.4 Configure the IBM SDK/JRE

You can have more than one Java SDK/JRE installed on your machine. Eclipse will pick one of the SDKs/JREs for the workbench, and it might not pick the IBM SDK/JRE:

1. In the action bar, select **Window** → **Preferences**.
2. In the Preferences pop-up, on the left select **Java** → **Installed JREs**.
3. In the Preferences pop-up, on the right, ensure that the IBM SDK/JRE (`C:\Program Files\IBM\Java50` for Windows and `/opt/ibm/java2-i386-50` for Linux) is defined and selected as the default JRE through the check box next to the name [✓].
4. If the IBM SDK/JRE is not defined, click **Add**:
   – In the Add JRE pop-up, select **Standard VM** and click **Next**.
   – In the resulting Add JRE pop-up:
     - Set the JRE home directory to the IBM SDK/JRE installation directory (for example, `C:\Program Files\IBM\Java50` on Windows).
     - Set the JRE name to a value that will be meaningful to you, such as **Java50** (default) or **IBMJRE**.
     - After the JRE system libraries list is filled in, click **Finish**.



JRE home: C:\Program Files\IBM\Java50

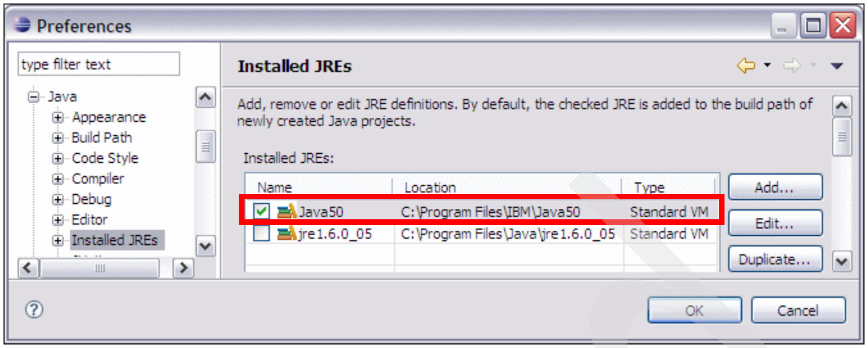
JRE name: Java50

Default VM Arguments:

JRE system libraries:

```
C:\Program Files\IBM\Java50\jre\lib\vm.jar
C:\Program Files\IBM\Java50\jre\lib\core.jar
C:\Program Files\IBM\Java50\jre\lib\charsets.jar
C:\Program Files\IBM\Java50\jre\lib\graphics.jar
```

5. If the IBM SDK/JRE is defined, but not selected as the default, click the check box next to the name to make it the default SDK.JRE.
6. Verify that the Preferences pop-up contains the IBM JRE. Click **OK** to save and close the Preferences pop-up.



Installed JREs

Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.

Installed JREs:

	Name	Location	Type
☑	Java50	C:\Program Files\IBM\Java50	Standard VM
□	jre1.6.0_05	C:\Program Files\Java\jre1.6.0_05	Standard VM

3.5 Add the ExperienceJavaEE test server definition

The WASCE IDE provides the capability to develop and package applications, but it needs to reference an actual WASCE runtime environment to test and debug applications:

- WASCE must be installed on the same system as the WASCE IDE. You did this step in Chapter 2, “Install and configure software” on page 25.

  In theory, the WASCE IDE and the Eclipse IDE also provide techniques for installing WASCE. This can be done in two ways:

  – WASCE can be installed as a separate product using the standard WASCE installation program (2.5.3, “Install WASCE v2.1” on page 37).

  – WASCE can be installed when you add the server to the WASCE IDE.

  In this book we used the first method, which we recommend, because doing this places WASCE into the standard file system location where most users are accustomed to finding the product. Otherwise, WASCE is installed into the WASCE IDE subdirectory structure.

- The server runtime must be added to the list of installed runtimes in the WASCE IDE. We do this in 3.5.1, “Add the WASCE server runtime definition” on page 59.

- The server instance must be defined in the Server view in the Java EE perspective in the WASCE IDE. This can be done in two places:

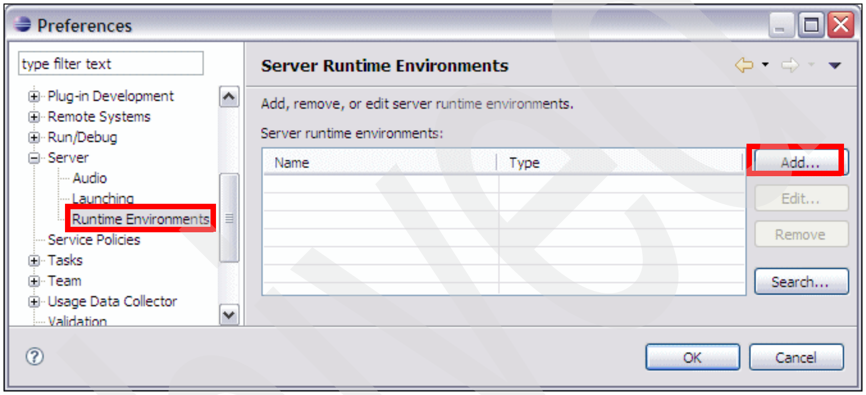
  – You can have this instance automatically defined when you add the server runtime definition (as we do in 3.5.1, “Add the WASCE server runtime definition” on page 59).

  – You can manually define this instance in the Servers view as described in 3.5.2, “Create the ExperienceJavaEE Server definition” on page 61.

3.5.1 Add the WASCE server runtime definition

Here we define the WASCE server runtime in WASCE IDE:

1. Navigate to the Installed Runtimes preferences:
   - Select **Window** → **Preferences**.
   - In the Preferences pop-up, scroll down and select **Server** → **Runtime Environments**. The Server Runtime Environments pane displays on the right.

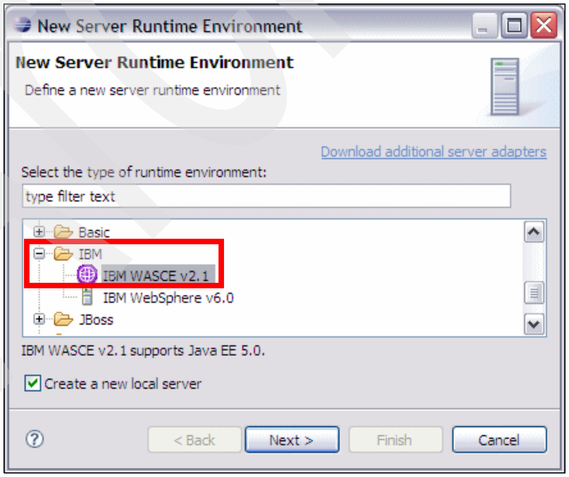


Server Runtime Environments

Add, remove, or edit server runtime environments.

Server runtime environments:

2. In the Server Runtime Environments pane, click **Add**.
   - In the New Server Runtime Environment pop-up, expand to select **IBM** → **IBM WASCE v2.1**, select **Create a new local server**, and click **Next**.



New Server Runtime Environment

Define a new server runtime environment

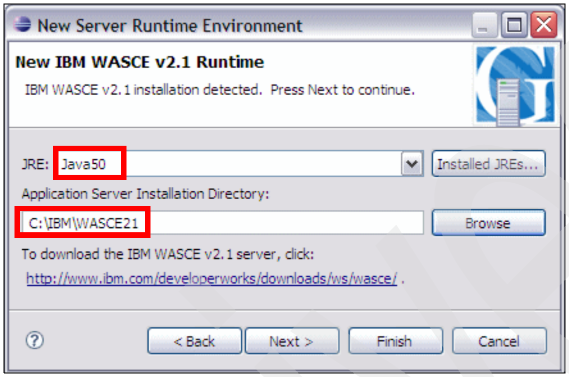
Select the type of runtime environment:

IBM WASCE v2.1 supports Java EE 5.0.

☑ Create a new local server

– In the New IBM WASCE v2.1 Runtime pop-up:

  - For JRE, select (if it is not already selected) the JRE that maps to the IBM SDK/JRE.
  - For Application Server Installation Directory, click **Browse** and locate `C:\IBM\WASCE21` for Windows and `/opt/IBM/WASCE21` for Linux.
  - Click **Finish**.



New Server Runtime Environment

New IBM WASCE v2.1 Runtime

IBM WASCE v2.1 installation detected. Press Next to continue.

JRE: Java50

Application Server Installation Directory: C:\IBM\WASCE21

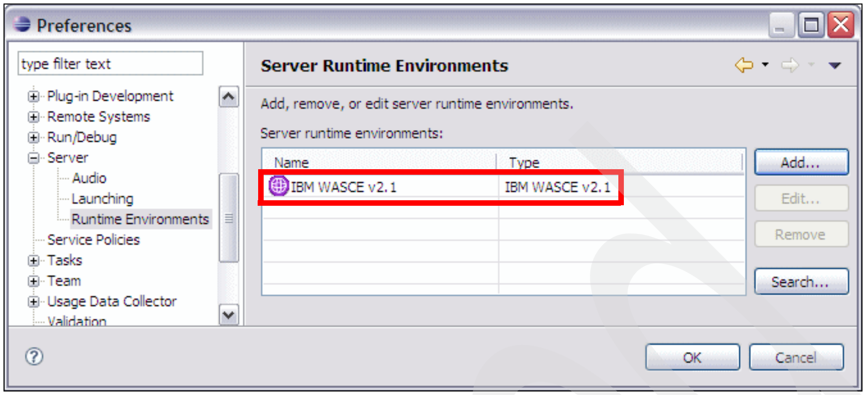
To download the IBM WASCE v2.1 server, click: http://www.ibm.com/developerworks/downloads/ws/wasce/ .

Off course?

If you are not sure about the JRE, close the New IBM WASCE v2.1 Runtime pop-up and view the **Java** → **Installed JREs**.

The IBM SDK/JRE will have an installation location of `C:\Program Files\IBM\Java50` and on a system with no other SDKs/JREs, this is usually named `Java50`.

3. Back in the Preferences pop-up, note the new entry.



Server Runtime Environments

Add, remove, or edit server runtime environments.

Server runtime environments:

Name	Type
IBM WASCE v2.1	IBM WASCE v2.1

4. Close the Preferences pop-up by clicking **OK**.

3.5.2 Create the ExperienceJavaEE Server definition

The workspace can contain the definitions of multiple test servers that identify the runtime environments to which you want to deploy, test, and debug applications.

Local server versus remote server:

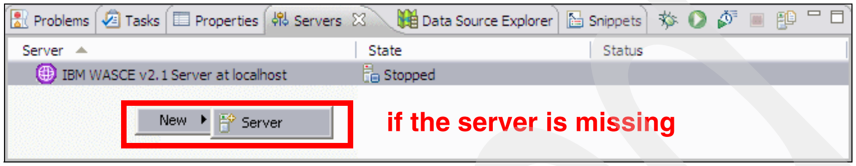
WTP (and the WASCE IDE) allows the definition of local test servers and remote test servers:

- A local test server runs on the same system as the IDE.
- A remote test server runs on a different system then the IDE, and its host name is something other than `localhost` or `127.0.0.1`.

From a functional standpoint, a remote test server is almost equivalent to a local test server. The only significant difference is that you cannot start a remote test server from the IDE. Instead, you must start the test server on the remote system, and then you can access it from the IDE.

In this section, we verify that the test server definition was automatically added by the previous section. If it was not, we manually add it here:

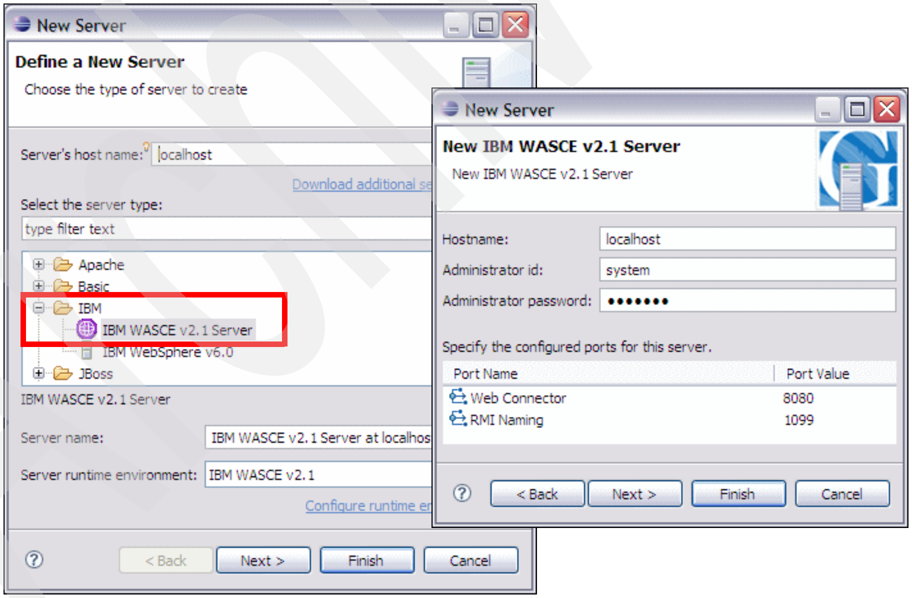
1. Switch to the Servers view in the lower-right pane. If you selected **Create new local server** in step 2 on page 59, a server with the name `IBM WASCE v2.1 Server at localhost` is already defined.
2. If you missed that selection, right-click the white space, and select **New** → **Server**.



Server	State	Status
IBM WASCE v2.1 Server at localhost	Stopped	

if the server is missing

3. In the New Server/Define a New Server pop-up, ensure that the server type is set to **IBM** → **IBM WASCE v2.1 Server** and click **Next**.
4. In the New IBM WASCE v2.1 Server pop-up, accept the defaults, and click **Finish**.



Define a New Server

Choose the type of server to create

Server's host name: localhost

Select the server type:

type filter text

IBM WASCE v2.1 Server

Server name: IBM WASCE v2.1 Server at localhos

Server runtime environment: IBM WASCE v2.1

New IBM WASCE v2.1 Server

New IBM WASCE v2.1 Server

Hostname: localhost

Administrator id: system

Administrator password: •••••••

Specify the configured ports for this server.

Port Name	Port Value
Web Connector	8080
RMI Naming	1099

Behind the scenes:

By default WASCE is installed with security and an administrative user ID of **system** and password **manager**. The Geronimo v2.1 documentation describes how you can change these:

```
http://cwiki.apache.org/GMOxDOC21/installation-and-configuration.html#
Installationandconfiguration-Changingtheusernameandpassword
```

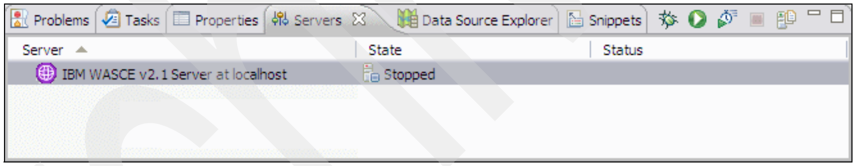
The default Web port of `8080` is used for application testing, and the RMI naming port of `1099` for RMI connections.

The ports used by the server are visible and can be configured in:

```
C:\IBM\WASCE21\var\config\config-substitutions.properties
```

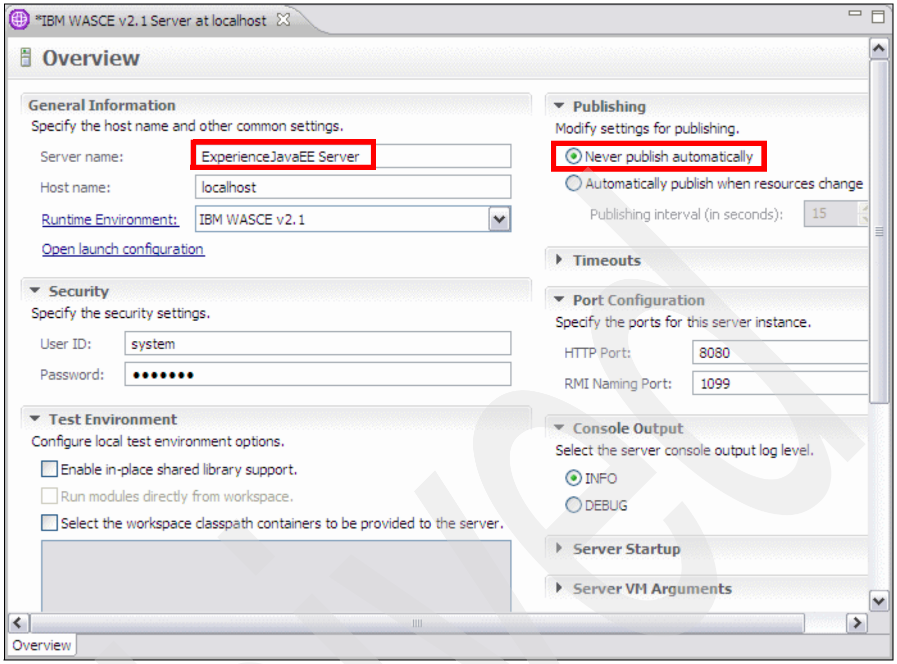
If the server fails to start because of port conflicts, open this file and change the offending ports.

5. The Servers view should now contain the new entry called **IBM WASCE v2.1 Server at localhost**. You can move the vertical bar between the Server and State columns to see the complete name.



Server	State	Status
IBM WASCE v2.1 Server at localhost	Stopped	

6. Double-click the new server to open in a configuration editor.
7. From the resulting IBM WASCE v2.1 Server at localhost editor:
   - In the General section, change the Server name field to **ExperienceJavaEE Server**.
   - In the Automatic Publishing section (on the upper right side of the pane), select **Never Publish Automatically**.



*IBM WASCE v2.1 Server at localhost

Overview

General Information
Specify the host name and other common settings.
Server name: ExperienceJavaEE Server
Host name: localhost
Runtime Environment: IBM WASCE v2.1
Open launch configuration

Security
Specify the security settings.
User ID: system
Password: •••••••

Test Environment
Configure local test environment options.
☐ Enable in-place shared library support.
☐ Run modules directly from workspace.
☐ Select the workspace classpath containers to be provided to the server.

Publishing
Modify settings for publishing.
● Never publish automatically
○ Automatically publish when resources change
Publishing interval (in seconds): 15

Timeouts

Port Configuration
Specify the ports for this server instance.
HTTP Port: 8080
RMI Naming Port: 1099

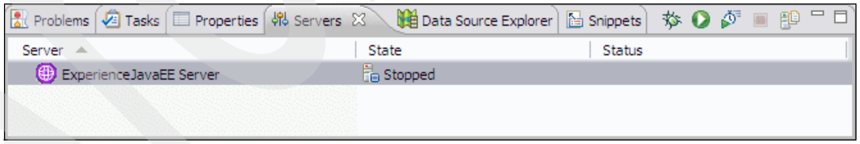
Console Output
Select the server console output log level.
● INFO
○ DEBUG

Server Startup

Server VM Arguments

- Save and close the changes using one of the following methods:
  - Click ☒ next to the name, and click **Yes** in the Save Resource pop-up.
  - Ensure that the editor window is active, and select **File** → **Save** (or Ctrl+S) and then **File** → **Close** (or Ctrl+F4).

8. The server list is now updated to show **ExperienceJavaEE Server**.



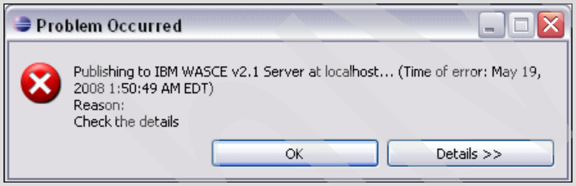
Server	State	Status
ExperienceJavaEE Server	Stopped	

Behind the scenes:

The server configuration editor contains a combination of configuration options for how WASCE is defined and to control how the WASCE IDE interacts with WASCE.

For example, automatic publishing:

- The default publishing setting is automatic, which instructs the WASCE IDE to automatically propagate updates for deployed applications to the test server at the specified interval (the default is 15 seconds). This occurs each time the WASCE IDE detects a change to the application. The key benefit of Automatic Publishing is that WASCE always has the latest version of your application.
- Automatic publishing does have drawbacks and limitations, such as the performance overhead for large projects, and the situation when your projects contain problems that cause automatic publishing to fail with an error pop-up:



Publishing to IBM WASCE v2.1 Server at localhost... (Time of error: May 19, 2008 1:50:49 AM EDT)
Reason:
Check the details

There are at least two different approaches for dealing with these situations:

1. Increase the automatic publishing interval to several minutes, thereby reducing the number of pop-ups.
2. Turn off automatic publishing, which will require you to manually publish the updated application.

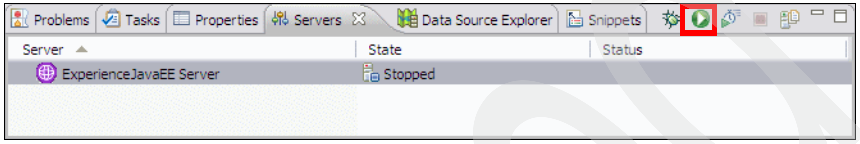
Which is the better or best approach? It depends on personal preference. For this book we turn off automatic publishing.

The Servers view shows the current publishing status, and displays **Republish** in the Status field if WASCE is out of sync with the workspace.

3.6 Start the ExperienceJavaEE Server test server

With the WASCE IDE and the WASCE runtime environment, most configuration changes are picked up automatically and the test server does not have to be restarted. Therefore, you can start the test server now, and leave it running for most of the subsequent scenarios:

1. In the Servers view, select **ExperienceJavaEE Server**.
2. Click the **Start** icon [▶] in the action bar.



Server	State	Status
ExperienceJavaEE Server	Stopped	

The server status changes to **Starting….**, and then to **Started** when the startup is complete. Manually switch back to the Servers view to see this final change in status.

If you receive any firewall prompts, allow access by `java` or `javaw`.

Off course?

If the server does not start and the Console shows these messages:

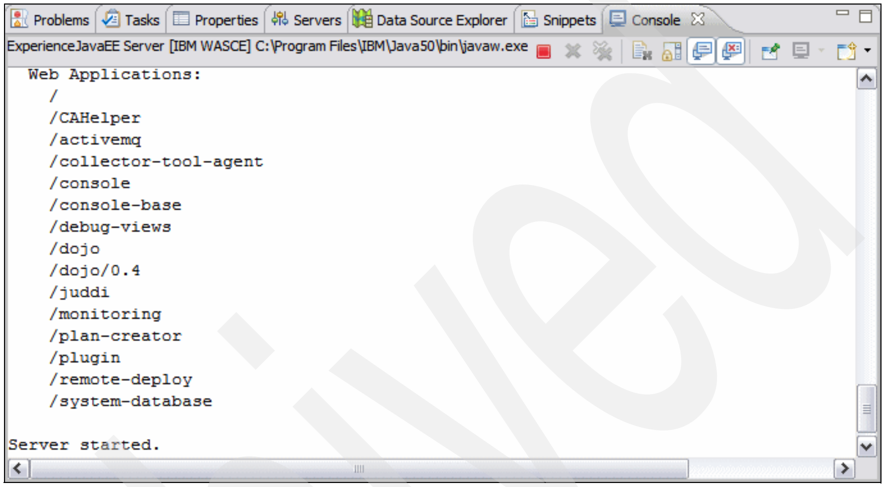
```
JVMJ9TI003E Agent_OnLoad failed for library instrument
java.lang.instrument/-javaagent: cannot create native agent.
JVMJ9VM015W Initialization error for library j9jvmti23(-3):
JVMJ9VM009E J9VMDllMain failed
Could not create the Java virtual machine.
```

- This indicates that the server is started with the wrong JVM.
- Close the WASCE IDE.
- Restart the WASCE IDE with the -vm argument:

```
......\eclipse.exe -vm "c:\Program Files\IBM\Java50\bin" (Windows)
....../eclipse -vm /opt/ibm/java2-i386-50/bin            (Linux)
```

- Make this the default way to start the WASCE IDE, using a short-cut or a command file.

3. The WASCE IDE should automatically switch to the Console view in the lower right (if it does not, click the tab for the Console view), and then you can view the startup messages in the log file (which will show as **ExperienceJavaEE Server [IBM WASCE]...**). Verify that startup occurred normally (for example, that there are no red messages), and look for the Geronimo Application Server started message.



ExperienceJavaEE Server [IBM WASCE] C:\Program Files\IBM\Java50\bin\javaw.exe

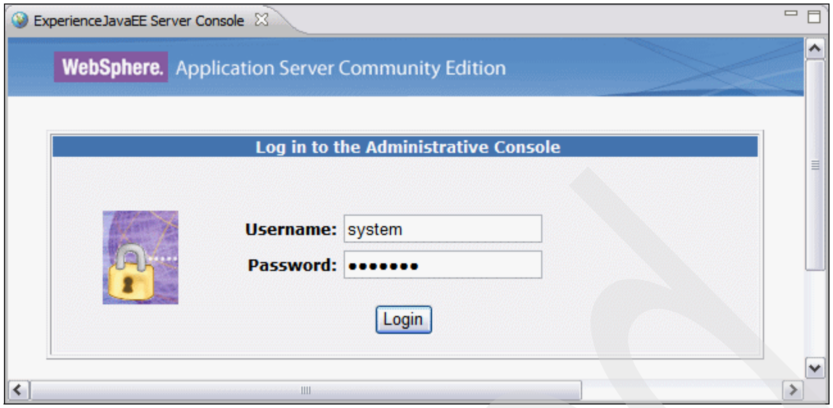
```
  Web Applications:
    /
    /CAHelper
    /activemq
    /collector-tool-agent
    /console
    /console-base
    /debug-views
    /dojo
    /dojo/0.4
    /juddi
    /monitoring
    /plan-creator
    /plugin
    /remote-deploy
    /system-database

Server started.
```

4. If you do not see the messages described in step 3, use the Console switch selection [🖥 ▾] to ensure that the Console view is showing the log for **ExperienceJavaEE Server**.

3.6.1 Start the WASCE Administrative Console

Next we perform configuration of the server with data sources and JMS resources by using the WASCE Console:

1. In the Servers view, right-click the **ExperienceJavaEE Server** and select **Launch WASCE Console**.
2. The console opens within the Eclipse IDE for login. Type the **system** user ID and password, by default, **manager**. Click **Login** (Figure 3-1).


ExperienceJavaEE Server Console
WebSphere. Application Server Community Edition
Log in to the Administrative Console
Login

Username: system

Password: •••••••

Figure 3-1 Login to the WASCE Administrative Console

3. The WASCE Console opens. Explore the different sections in the Console Navigation on the left:

   – Server
   – Services
   – Applications
   – Security
   – Debug Views
   – Embedded DB

4. You might want to set the level of information that is displayed in the WASCE Console. Select **Server** → **Server Logs**. Set the Log Level to **ERROR** and click **Update**, to suppress warnings. You can change this at any time while the server is running.

5. You have no configuration tasks for now, so you can exit the console by:

   – Clicking **Logout** to log out of the WASCE Console.

   – Closing the **ExperienceJavaEE Server Console** window.

> **Behind the scenes:** You can control which browser is used by WASCE IDE. Select **Window** → **Web Browser** to select the default internal browser, or an external browser (Internet Explorer or Firefox).

3.6.2 Control the server outside of WASCE IDE

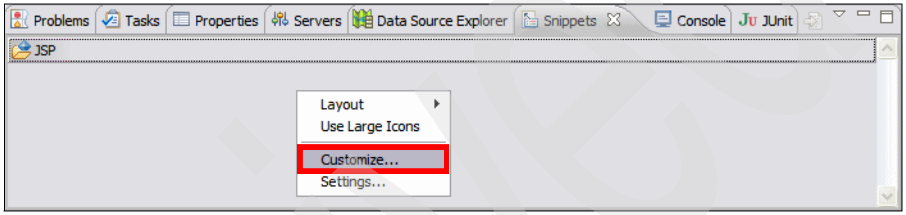
The installation creates a menu with icons to start and stop the server, and to start the WASCE Console in a browser. You can use these icons to control the server and perform administrative tasks. The WASCE IDE notices if the server is running or not.

3.7 Import the sample code snippets

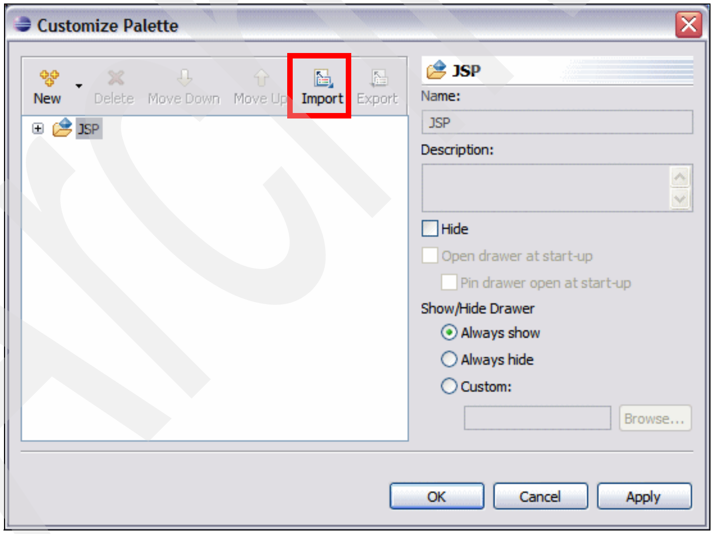
The WASCE IDE allows you to configure a set of reusable samples or templates called snippets. You can insert these snippets into your artifacts either as a simple cut and paste, or with advanced features such as variable replacement.

In this case, you use snippets as a simple cut and paste mechanism when adding code to various Java classes and Web pages, instead of manually typing in the code or copying from a text file:

1. Switch to the Snippets view in the lower right pane.
2. Right-click anywhere inside the view, and select **Customize**.



3. In the Customize Palette pop-up, click **Import**.



JSP

Name: JSP

Description:

□ Hide

□ Open drawer at start-up

□ Pin drawer open at start-up

Show/Hide Drawer

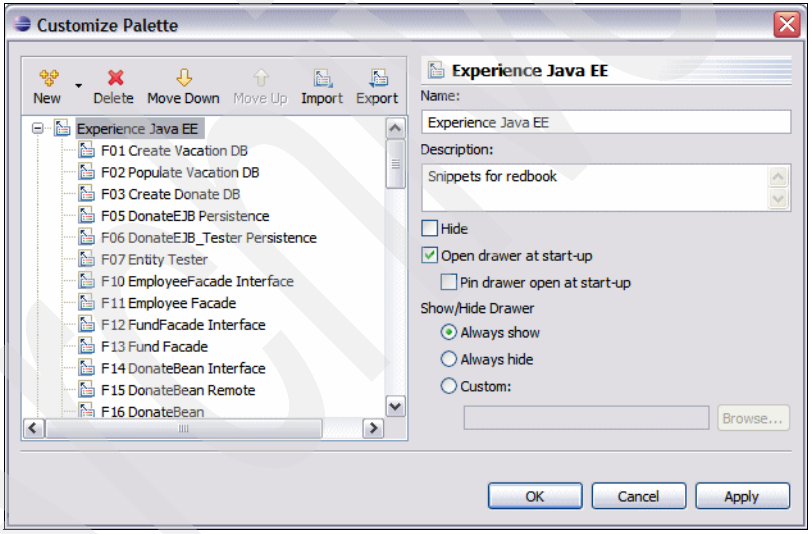
● Always show

○ Always hide

○ Custom:

Browse...

4. In the directory browser pop-up, perform the following actions:
   - Windows Navigate to the directory `C:\7639code\snippets`, select **experiencejavaee_snippets.xml**, and click **Open**.
   - Linux Navigate to the directory `$HOME/7639code/snippets`, select **experiencejavaee_snippets(.xml)**, and click **OK**.
5. In the Customize Palette pop-up, move the newly created category to the top of the list, by performing the following:
   - Scroll to the bottom of the list (if not already there).
   - Select the **Experience Java EE** category.
   - Click **Move Up** until the **Experience Java EE** category is at the top of the list.
6. Expand the **Experience Java EE** category by clicking the plus expansion symbol located next to it (⊞ on Windows or ▷ on Linux). Now you can view the list of imported snippets for this category.



Experience Java EE
- F01 Create Vacation DB
- F02 Populate Vacation DB
- F03 Create Donate DB
- F05 DonateEJB Persistence
- F06 DonateEJB_Tester Persistence
- F07 Entity Tester
- F10 EmployeeFacade Interface
- F11 Employee Facade
- F12 FundFacade Interface
- F13 Fund Facade
- F14 DonateBean Interface
- F15 DonateBean Remote
- F16 DonateBean

Experience Java EE

Name: Experience Java EE

Description: Snippets for redbook

☐ Hide

☑ Open drawer at start-up

☐ Pin drawer open at start-up

Show/Hide Drawer

● Always show

○ Always hide

○ Custom:

7. Select a snippet and its code is shown on the right side.
8. Click **OK** to close the pop-up.

3.8 Configure user libraries in Eclipse

To test stand-alone access to the databases from Java Persistence API (JPA) entities through JUnit tests, we have to define user libraries for accessing the Derby databases. Similarly, to connect a JUnit test case to EJB session beans, we require an EJB user library.

We could defer the definition of these user libraries to the chapters where they are used, but it is more efficient to perform all setup activities before starting with the application code:

1. In the WASCE IDE, select **Window** → **Preferences**.
2. In the Preferences pop-up, select **Java** → **Build Path** → **User Libraries**.
3. In the User Libraries page, click **New**.
4. In the New User Library pop-up, type **Derby Net Client** as User library name, and click **OK**.
5. With **Derby Net Client** selected, click **Add JARs**.
6. In the JAR Selection pop-up, browse to:

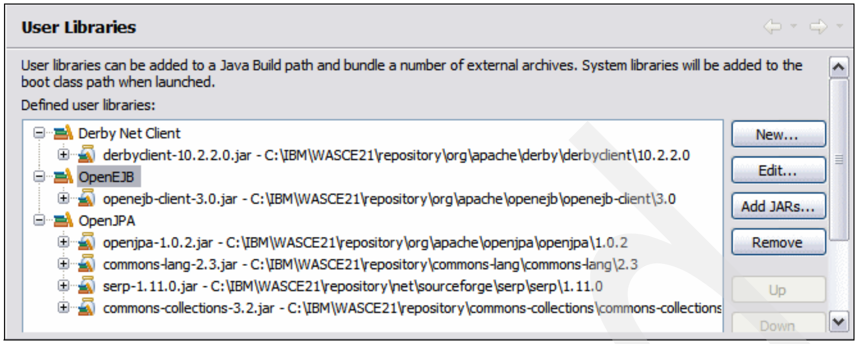
   – `C:\IBM\WASCE21\repository\org\apache\derby\derbyclient\10.2.2.0\derbyclient-10.2.2.0.jar`

   This is the JAR file that WASCE uses on the server-side to access other Derby databases over a network.
7. Back in the Preferences/User Libraries pop-up, repeat the steps defined previously to create an **OpenJPA** user library, and add the following JARs from the WASCE installation repository (`C:\IBM\WASCE21\repository`):

   ```
   org/apache/openjpa/openjpa/1.0.2/openjpa-1.0.2.jar
   commons-lang/commons-lang/2.3/commons-lang-2.3.jar
   net/sourceforge/serp/serp/1.11.0/serp-1.11.0.jar
   commons-collections/commons-collections/3.2/commons-collections-3.2.jar
   ```
8. Repeat the steps to create an **OpenEJB** user library, and add the following JAR from the WASCE installation repository (`C:\IBM\WASCE21\repository`):

   ```
   org/apache/openejb/openejb-client/3.0/openejb-client-3.0.jar
   ```

9. Verify that the JAR files are listed in their respective hierarchy, and click **OK**.



User Libraries

User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.

Defined user libraries:

- Derby Net Client
  - derbyclient-10.2.2.0.jar - C:\IBM\WASCE21\repository\org\apache\derby\derbyclient\10.2.2.0
- OpenEJB
  - openejb-client-3.0.jar - C:\IBM\WASCE21\repository\org\apache\openejb\openejb-client\3.0
- OpenJPA
  - openjpa-1.0.2.jar - C:\IBM\WASCE21\repository\org\apache\openjpa\openjpa\1.0.2
  - commons-lang-2.3.jar - C:\IBM\WASCE21\repository\commons-lang\commons-lang\2.3
  - serp-1.11.0.jar - C:\IBM\WASCE21\repository\net\sourceforge\serp\serp\1.11.0
  - commons-collections-3.2.jar - C:\IBM\WASCE21\repository\commons-collections\commons-collections

Behind the scenes:

We define the user libraries here to consolidate the creation of user libraries in one location in this book.

The Derby Net Client and OpenJPA libraries will be used for JUnit testing of the JPA entities in 5.9.1, “Create the JUnit project” on page 149.

The OpenEJB library will be used in 6.7.1, “Add the EJB dependency and the OpenEJB library” on page 203.

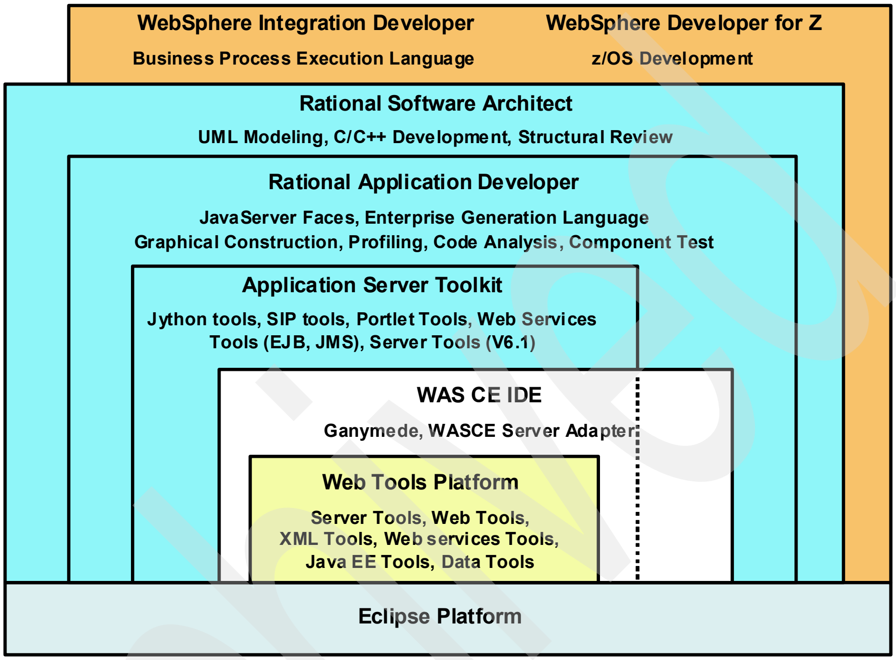
3.9 IDE preferences

There are a few other preferences that can be set for our advantage. Open the Preferences dialog by selecting **Window** → **Preferences**. In the Preferences dialog:

1. Select **General** → **Editors** → **Text Editors**, and select **Show line numbers**. This setting is practical because error messages sometime refer to line numbers.
2. Select **Java** → **Code Style** → **Formatter** to see the preferred formatting style for Java code. Note that you can tailor your own style as well.

3.10 Explore!

Figure 3-2 shows the various integrated development environments that IBM provides for development of J2EE and Java EE applications.



WebSphere Integration Developer
Business Process Execution Language
WebSphere Developer for Z
z/OS Development
Rational Software Architect
UML Modeling, C/C++ Development, Structural Review
Rational Application Developer
JavaServer Faces, Enterprise Generation Language
Graphical Construction, Profiling, Code Analysis, Component Test
Application Server Toolkit
Jython tools, SIP tools, Portlet Tools, Web Services Tools (EJB, JMS), Server Tools (V6.1)
WAS CE IDE
Ganymede, WASCE Server Adapter
Web Tools Platform
Server Tools, Web Tools, XML Tools, Web services Tools, Java EE Tools, Data Tools
Eclipse Platform

Figure 3-2 IBM integrated development environments

The WASCE IDE tool configured in this chapter is the only Eclipsed-based development platform for WASCE. However, just as IBM has another Java runtime environment with WebSphere Application Server v6.x and v7.0, IBM also has other Eclipsed-based development products to support WebSphere Application Server v6.x and v7.0.

- **WebSphere Application Server Toolkit v6.1** (AST) provides similar functions as WASCE IDE, but for the basic WebSphere Application Server v6.x:
  - AST is built on top of WTP.
  - AST includes Portlet tools and Jython administrative scripting.

  Note: AST is not available for WebSphere Application Server v7.0, which is shipped with a copy of Rational Application Developer Assembly and Deploy, which provides support for creating, building, testing, and deploying J2EE 1.4 and Java EE 5 applications.

- **Rational Application Developer v7.0** provides a comprehensive Java/J2EE oriented IDE, with additional capabilities:
  - Visual editors for Web development (HTML, JSP, Struts, JSF)
  - Support for other J2EE servers (including older WebSphere Application Server releases, Apache Tomcat, and BEA WebLogic)
  - Visual editors for EJB development
  - Integrated WebSphere Portal development/testing
  - Basic Unified Modeling Language (UML) support
  - Code analysis and validation tools (profiling)
  - Support for service-oriented architecture (SOA) extensions, such as Service Data Objects (SDOs)
  - Enterprise Generation Language (EGL) support, available through a separate installable package called IBM Rational Business Developer Extension
- **Rational Application Developer for WebSphere Software v7.5** is the next version of Application Developer and provides a comprehensive Java/Java EE development environment for WebSphere Application Server v7.0. Refer to the following link for more information:

  `http://www.ibm.com/software/awdtools/developer/application/index.html`
- **Rational Software Architect v7.x** provides UML based modeling and design tools intended for use by non-programmers such as architects and systems analysts. The models can be used to generate the skeleton Java EE artifacts that then can be completed and extended by WASCE IDE and Rational Application Developer.
- **WebSphere Integration Developer v6.1** provides for the development and testing of non-J2EE artifacts such as Enterprise Service Bus (ESB) mediations and WS-BPEL based workflows. Refer to 19.3, “Enterprise Service Bus” on page 548 and 19.4, “Business Process Execution Language” on page 550 for more information.

Collectively these products are all based off the same core foundation of Eclipse, the Web Tools Platform (WTP), and targeted server adapters. Thus, the general skills and concepts learned in the WASCE IDE can be leveraged when using these other products.

One key consideration is that these products are based on different versions of Eclipse (v3.2, v3.3, and v3.4) and the WTP (v1.5, v2.0, v3.0). This reflects the reality of the Eclipse environment, where new releases and capabilities are being released each year. 18 month old products (such as the WebSphere Application Server Toolkit v6.1) can easily be two levels behind the latest releases of the underlying open source packages.

The positive side is that these releases represent evolutionary changes and not revolutionary changes. The core functions and capabilities are the same across releases, and are changed to provide improved usability, improved features, and to support the latest technical specification. Someone using Eclipse v3.2 with WTP v1.5 should be able to easily ramp up on using Eclipse v3.4 with WTP v3.0.

Additional Explore resources are available at the following Web sites:

- WASCE announcement letter:

  http://www.ibm.com/common/ssi/fcgi-bin/ssialias?infotype=an&subtype=ca&appname=GPA&htmlfid=897/ENUS206-076

- IBM developerWorks: Recommended reading list: J2EE and WebSphere Application Server:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0305_issw/recommendedreading.html

- Rational home page, Rational Application Developer, and Rational Software Architect:

  http://www.ibm.com/software/rational
  http://www.ibm.com/software/awdtools/developer/application/index.html
  http://www.ibm.com/software/awdtools/architect/swarchitect/index.html

- IBM developerWorks Rational home page:

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

- WebSphere Integration Developer home page:

  http://www-306.ibm.com/software/integration/wid/

4

Prepare the legacy application

In this chapter, we configure the initial application database that will be used in the Experience Java EE! scenario.

4.1 Learn!

Derby is a Java-based “load and go” database that supports standard structured query language (SQL) and JDBC. It provides a full-featured, robust, small-footprint database server that is simple to deploy and that reduces the cost of applications.

Derby does not have to be installed like WASCE or WASCE IDE. Instead, Derby only requires that the various JAR and configuration files are copied onto the target system and available through the classpath configuration. This was done as part of the WASCE or WASCE IDE installation, because they both ship with Derby; however, we could just as easily go to the Derby Web site and download the files.

Derby contains the following two different run-time versions:

- **Embedded** provides direct access to a database to a single user. This version provides simple *load and go support*: Point the JDBC drivers to the database, and invoke the access. It also has low overhead because it runs in the same JVM as the caller.

  The primary disadvantage of the embedded is that you cannot have concurrent access (for example, by both WASCE IDE running in one JVM and WASCE running in another JVM). If you want to see if an entity updated a database record, you must first shutdown WASCE, before you can view the database with WASCE IDE

- **Network Server** provides network access to a database to multiple users for concurrent operation. This version requires that an administrator configure and start the database server. The database client points the JDBC drivers to this network location and database name, and then invokes the access. This version runs in a separate JVM.

WASCE is configured with a Derby Network Server, and the WASCE IDE will support both. This book, for simplicity of configuration, uses the Derby Network Server in WASCE.

For the purpose of this book, Derby allows you to develop and test an application that uses a database without installing a full-function database, such as Oracle Database or DB2.

Derby was seeded with code submission from IBM based on a product feature called Cloudscape.

Additional Learn resources are available at the following Web sites:

- Apache Derby home page:

  http://db.apache.org/derby

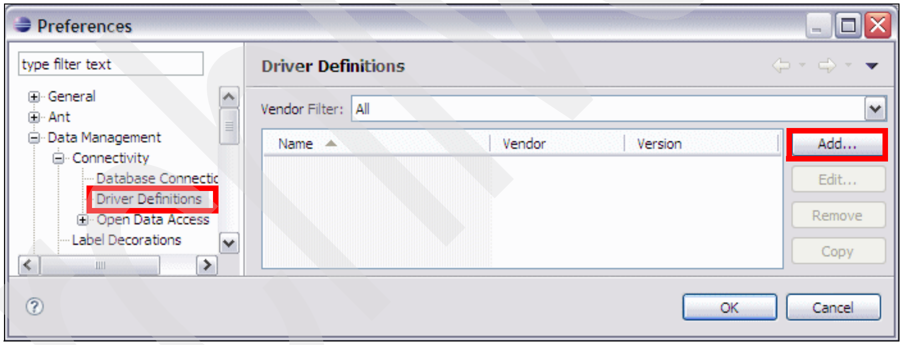
- IBM developerWorks: Open Source: Apache Derby project resources:

  http://www.ibm.com/developerworks/opensource/top-projects/derby-community.html

4.2 Configure the Derby JDBC driver

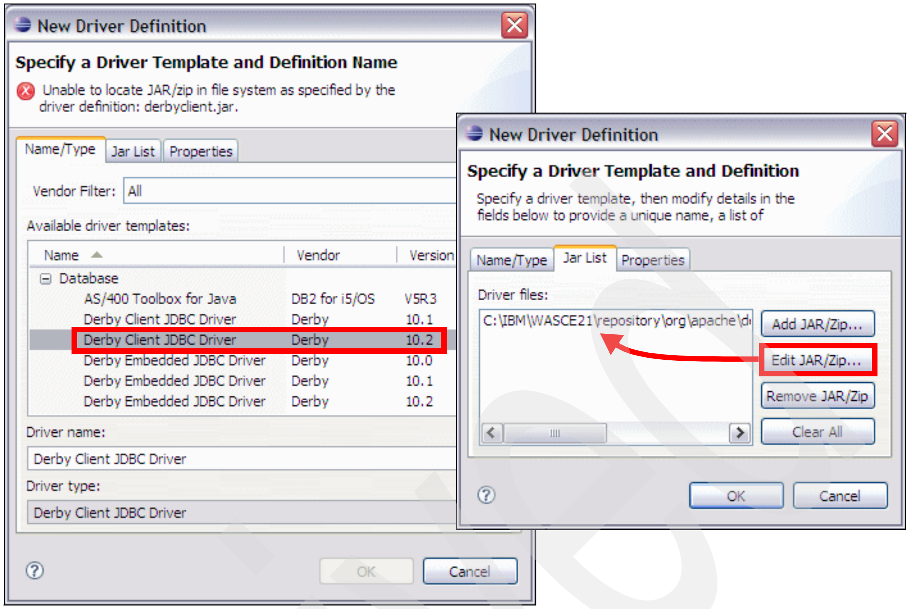
In this section we update the Eclipse IDE Data Management definition to reference the Derby JDBC drivers contained in WASCE:

1. At the application action bar, select **Window** → **Preferences**.
2. In the Preferences pop-up, scroll down and select **Data Management** → **Connectivity** → **Driver Definitions**. The Driver Definitions pane displays on the right.



3. On the right of the Driver Definitions pane, click **Add**.
4. In the resulting New Driver Definition pop-up:
   - In the Name/Type tab, under name, select **Database** → **Derby Client JDBC Driver** (version 10.2).
   - In the Jar List tab:
     - Under Driver files, select **derbyclient.jar**, and click **Edit JAR/Zip**.
     - In the Select the file pop-up, browse to and select the following file, and click **Open**.

       ```
       C:\IBM\WASCE21\repository\org\apache\derby\derbyclient\10.2.2.0\derbyclient-10.2.2.0.jar
       ```



New Driver Definition

Specify a Driver Template and Definition Name

Unable to locate JAR/zip in file system as specified by the driver definition: derbyclient.jar.

Vendor Filter: All

Available driver templates:

Name	Vendor	Version
Database		
AS/400 Toolbox for Java	DB2 for i5/OS	V5R3
Derby Client JDBC Driver	Derby	10.1
Derby Client JDBC Driver	Derby	10.2
Derby Embedded JDBC Driver	Derby	10.0
Derby Embedded JDBC Driver	Derby	10.1
Derby Embedded JDBC Driver	Derby	10.2

Driver name: Derby Client JDBC Driver

Driver type: Derby Client JDBC Driver

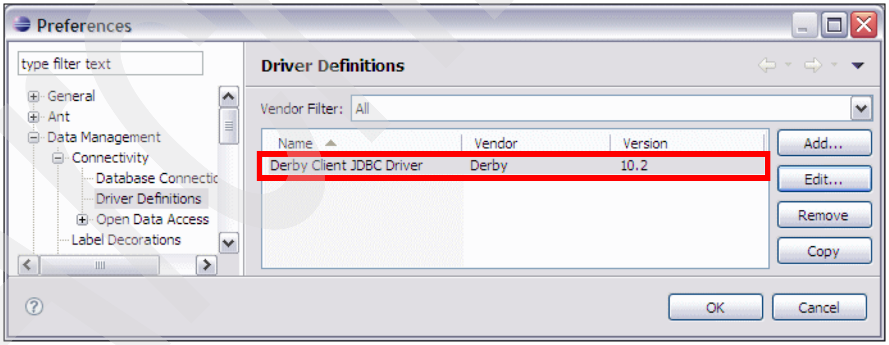
New Driver Definition

Specify a Driver Template and Definition

Specify a driver template, then modify details in the fields below to provide a unique name, a list of

Driver files: C:\IBM\WASCE21\repository\org\apache\d

5. Click **OK** to close the New driver definition pop-up.
6. Back in the Preferences pop-up, note the new driver definition, and click **OK** to close.



Preferences

Driver Definitions

Vendor Filter: All

Name	Vendor	Version
Derby Client JDBC Driver	Derby	10.2

4.3 Create the Vacation database

The Vacation database simulates the legacy application. Derby does not have a graphical administration tool, so instead we use the Data tooling in the WASCE IDE to create the database.

1. Switch to the **Database Development** perspective:
   - Click the **Select Perspective** icon [ ] located at the upper right, and select **Other** from the pull-down.
   - In the Open Perspective pop-up, select **Database Development** and click **OK**.

Behind the scenes:

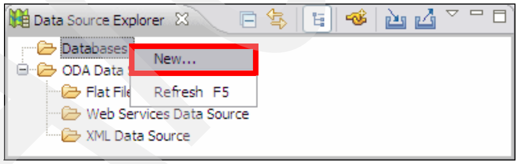
Perspectives are tailored for user roles. Typically a database specialist would work with databases in the Database Development perspective, whereas a Java EE developer would work in the Java EE perspective.

It is possible to add views that are displayed in one perspective to another perspective, where that view is not open by default. Also, users sometimes close views by mistake. Here are instructions on how to deal with views:

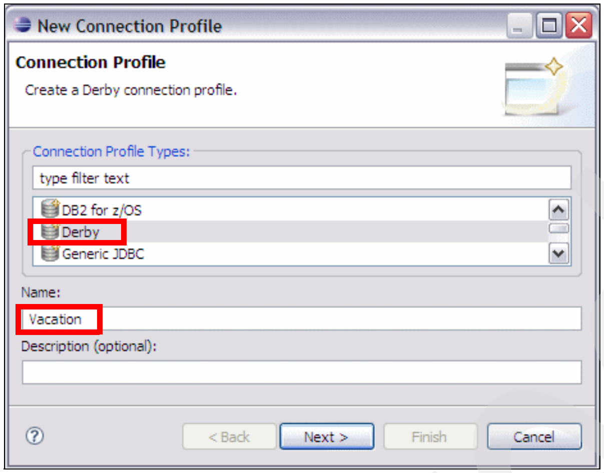
- To add a view to a perspective, in the WASCE IDE action bar, select **Window** → **Show View** → **Other**. Navigate to the view, and click **OK**.

  For example, if the Data Source Explorer view is not visible, in the Show View pop-up, select **Data** → **Data Source Explorer**, and click **OK**.
- To reset a perspective to display the default set of views, select **Window** → **Reset Perspective**.
- You can also save the current combination of panes and views by selecting **Window** → **Save Perspective As**. You can save as a new perspective or change an existing perspective.

2. In the Data Source Explorer view in the WASCE IDE Database Development perspective, right-click **Databases** and select **New**.



3. In the New Connection Profile/Connection Selection Page pop-up:
   - Under Connection Profile Types, select **Derby**.
   - Set name to **Vacation**.
   - Click **Next**.



New Connection Profile

Connection Profile

Create a Derby connection profile.

Connection Profile Types:

type filter text

- DB2 for z/OS
- Derby
- Generic JDBC

Name: Vacation

Description (optional):

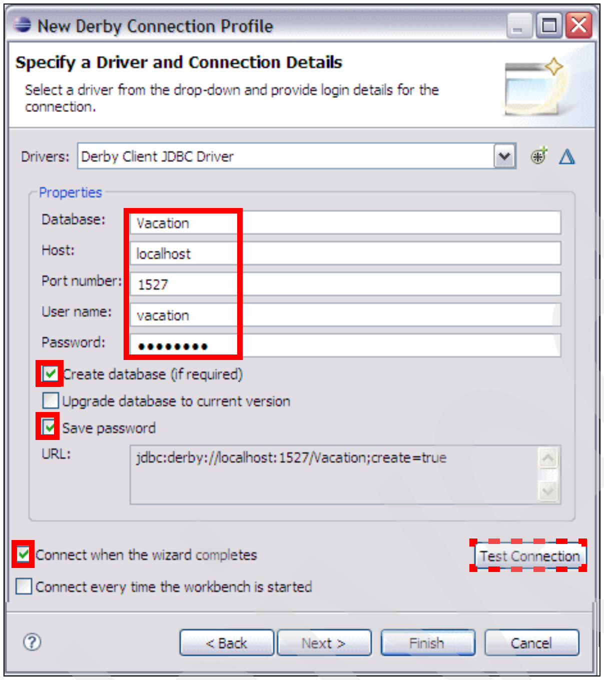
4. In the New Derby Connection Profile/Specify a Driver and Connection Details pop-up:
   - From the Drivers pull-down, select **Derby Client JDDC Driver**.
   - Under Properties:
     - Set Database to **Vacation**.
     - Set Host to **localhost**.
     - Set Port number to **1527**.
     - Set Username to **vacation**.
     - Set Password to any value (such as `vacation`).
   - Select **Create database (if required)**.

Behind the scenes:

These fields combine to form the URL string that is used when connecting to the Derby Network server that is running inside of WASCE on port 1527:

```
jdbc:derby://localhost:1527/Vacation;create=true
```

   - Select **Save password**.
   - Select **Connect when the wizard completes**.


New Derby Connection Profile

Specify a Driver and Connection Details

Select a driver from the drop-down and provide login details for the connection.

Drivers: Derby Client JDBC Driver

Properties

Database: Vacation

Host: localhost

Port number: 1527

User name: vacation

Password: ••••••••

☑ Create database (if required)

□ Upgrade database to current version

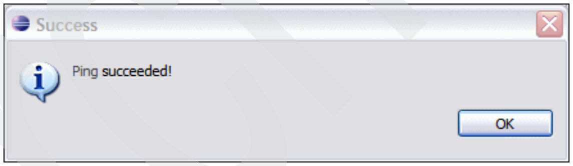
☑ Save password

URL: jdbc:derby://localhost:1527/Vacation;create=true

☑ Connect when the wizard completes

□ Connect every time the workbench is started

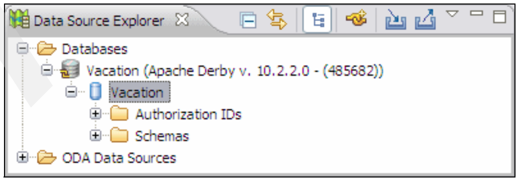
– Click **Test Connection** (bottom right) to verify that the database properties were correctly specified. In the Success pop-up, click **OK**.


Success

Ping succeeded!

– Click **Finish**.

5. The Data Source Explorer view contains the `Vacation` database connection, initially connected.


Data Source Explorer

```
Databases
  Vacation (Apache Derby v. 10.2.2.0 - (485682))
    Vacation
      Authorization IDs
      Schemas
ODA Data Sources
```

Behind the scenes:

The **create=true** parameter on the URL connection string instructs the Derby Server to automatically create the database. By default the database is created as directory `C:\IBM\WASCE21\var\derby\Vacation`.

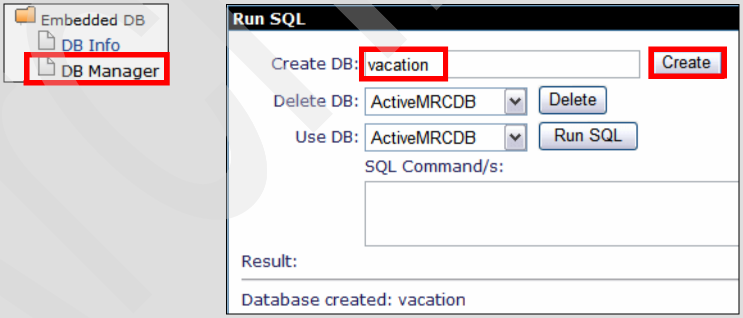
Alternatives:

This step uses the **Create the database if required** option in the New Connection wizard to create the database. You can also create the database externally using the Derby command line interface or using the WASCE database management function:

- From the Derby **`ij`** command line (`C:\IBM\WASCE21\bin\ij.bat`), run the following commands to create the database:

```
connect 'jdbc:derby://localhost:1527/vacation;create=true';
exit;
```

- From the WASCE Console (right-click the server in the Servers view and select **Launch WASCE Console**). On the left select **Embedded DB** → **DB Manager** and on the right:
  - Next to Create DB: enter the database name: **vacation**
  - Click **Create**.
  - A message is displayed under Results.



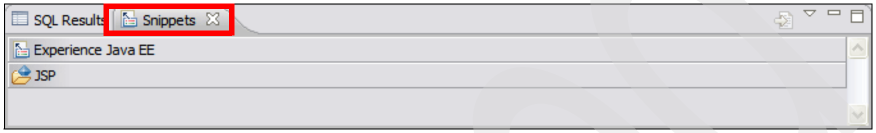
Database created: vacation

  - The `Vacation` database is listed on the top under Database List.

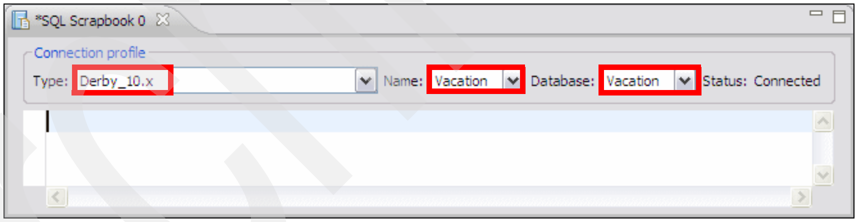
4.4 Create the employee table in the Vacation database

In this section we use the data tooling in WASCE IDE to create and populate the `EMPLOYEE` table that contains the application data:

1. In the action bar, select **Window** → **Show View** → **Other**.
   - In the Show View pop-up, select **General** → **Snippets** (or start typing snippet to find the view).
   - Click **OK** and the Snippets view is visible in the lower right pane.



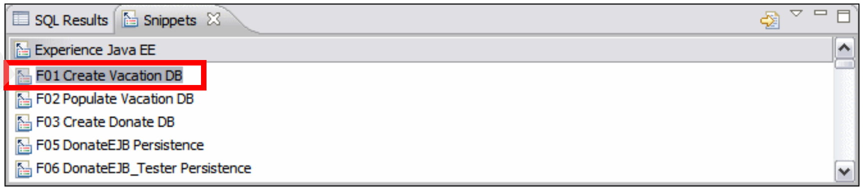
2. In the action bar, click the **SQL Scrapbook** icon [ ].
3. In the resulting **SQL Scrapbook 0** editor (in the upper right):
   - Type: **Derby_10.x**
   - Name: **Vacation**
   - Database: **Vacation**


*SQL Scrapbook 0

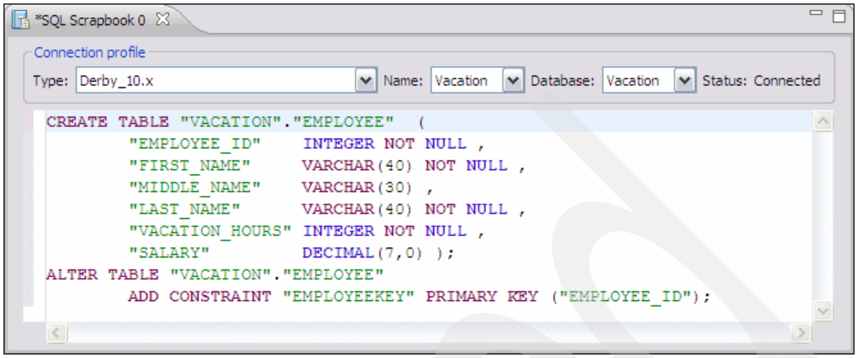
Connection profile

Type: Derby_10.x Name: Vacation Database: Vacation Status: Connected

   - Select anywhere on the white space of the scrapbook.
   - In the Snippets view (lower right), expand the **Experience Java EE** category, and double-click on **F01 Create Vacation DB** to insert the statements into the editor.



– The SQL scrapbook now contains the SQL statements that create the table and the primary key.

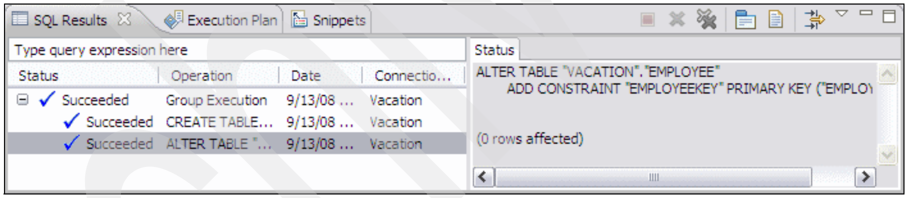


Type: Derby_10.x Name: Vacation Database: Vacation Status: Connected

```
CREATE TABLE "VACATION"."EMPLOYEE"  (
        "EMPLOYEE_ID"     INTEGER NOT NULL ,
        "FIRST_NAME"      VARCHAR(40) NOT NULL ,
        "MIDDLE_NAME"     VARCHAR(30) ,
        "LAST_NAME"       VARCHAR(40) NOT NULL ,
        "VACATION_HOURS"  INTEGER NOT NULL ,
        "SALARY"          DECIMAL(7,0) );
ALTER TABLE "VACATION"."EMPLOYEE"
        ADD CONSTRAINT "EMPLOYEEKEY" PRIMARY KEY ("EMPLOYEE_ID");
```

– Right-click anywhere on the editor, and select **Execute All**.

4. The WASCE IDE executes the statements and opens an SQL Results view in the lower pane to display the output messages for the commands (create table, alter table, create records).



Status	Operation	Date	Connectio...
⊟ ✓ Succeeded	Group Execution	9/13/08 ...	Vacation
✓ Succeeded	CREATE TABLE...	9/13/08 ...	Vacation
✓ Succeeded	ALTER TABLE "...	9/13/08 ...	Vacation

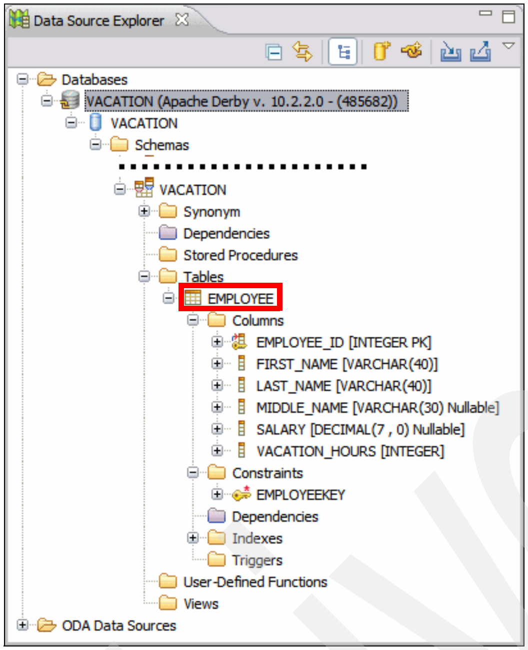
```
Status
ALTER TABLE "VACATION"."EMPLOYEE"
  ADD CONSTRAINT "EMPLOYEEKEY" PRIMARY KEY ("EMPLOY

(0 rows affected)
```

Behind the scenes:

The commands are standard DDL (data definition language) statements that create the EMPLOYEE table, and define the primary key.

After executing the DDL statements the EMPLOYEE table is defined in the Vacation database.

5. In the Data Source Explorer view, expand **Databases** → **Vacation (Apache Derby v. 10.2.2.0 - (485682))** → **Vacation**, right-click, and select **Refresh** to see the new VACATION schema and the EMPLOYEE table under Databases → Vacation (Apache Derby v. 10.2.2.0 - (485682)) → Vacation → Schemas → VACATION → Tables → EMPLOYEE table.



Data Source Explorer

```
Databases
  VACATION (Apache Derby v. 10.2.2.0 - (485682))
    VACATION
      Schemas
        ........................
        VACATION
          Synonym
          Dependencies
          Stored Procedures
          Tables
            EMPLOYEE
              Columns
                EMPLOYEE_ID [INTEGER PK]
                FIRST_NAME [VARCHAR(40)]
                LAST_NAME [VARCHAR(40)]
                MIDDLE_NAME [VARCHAR(30) Nullable]
                SALARY [DECIMAL(7 , 0) Nullable]
                VACATION_HOURS [INTEGER]
              Constraints
                EMPLOYEEKEY
              Dependencies
              Indexes
              Triggers
          User-Defined Functions
          Views
ODA Data Sources
```

The **Refresh** action reloads the database definition and ensures that the VACATION schema and the EMPLOYEE table appear.

Alternatives:

As before, you can also create the table through the Derby command line interface or the WASCE database management function:

- From the Derby **ij** command line, connect, and run the commands:

```
connect 'jdbc:derby://localhost:1527/Vacation';
run 'createEmployee.sql'
exit;
```

- From the WASCE Console, select **Embedded DB** → **DB Manager** and on the right:
  - Next to Use DB: Select the Vacation database.
  - In the SQL Command/s area, paste the **F01 Create Vacation DB** snippet contents.
  - Click **Run SQL**.
  - At the top of the screen in the Database List section, click the **Application** link next to the Vacation database.
  - At the resulting screen, note the VACATION.EMPLOYEE table.

4.5 Populate the Vacation database

In this section we use the Data tooling in WASCE IDE to populate the database table with the initial data:

1. In the SQL Scrapbook 0 editor, delete the existing code, and insert the **F02 Populate Vacation DB** snippet from the Snippets view:

```
INSERT INTO "VACATION"."EMPLOYEE" ( "EMPLOYEE_ID", "FIRST_NAME",
             "MIDDLE_NAME", "LAST_NAME", "VACATION_HOURS")
   VALUES (1, 'Charles', 'Patrick', 'Brown', 20);
INSERT INTO "VACATION"."EMPLOYEE" ( "EMPLOYEE_ID", "FIRST_NAME",
             "MIDDLE_NAME", "LAST_NAME", "VACATION_HOURS", "SALARY")
   VALUES (2, 'Neil', '', 'Armstrong', 50, 100000);
INSERT INTO "VACATION"."EMPLOYEE" ( "EMPLOYEE_ID", "FIRST_NAME",
             "MIDDLE_NAME", "LAST_NAME", "VACATION_HOURS", "SALARY")
   VALUES (3, 'Rich ', 'ReallyRich', 'Rich', 90, 500000);
```

   – Select anywhere on the editor, and right-click **Execute All**.

2. The WASCE IDE executes the statements and displays the results in the SQL Results view.

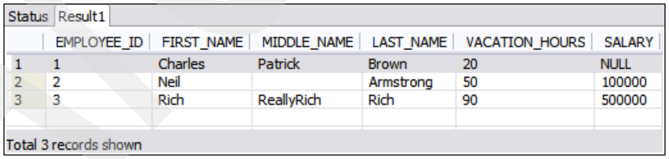
3. Close the **SQL Scrapbook 0** editor (without saving).

   – In the Save Resource pop-up click **No** (to close the editor without saving).

4. Verify that the data was loaded into the `EMPLOYEE` table:

   – In the Data Source Explorer view, select **Databases** → **Vacation (Apache Derby v. 10.2.2.0 - (485682))** → **Vacation** → **Schemas** → **Vacation** → **Tables** → **Employee**, right-click and select **Data** → **Sample Contents**.

   – In the resulting SQL Results view in the lower right, switch to the Result1 tab on the right and verify that the three rows were added.



	EMPLOYEE_ID	FIRST_NAME	MIDDLE_NAME	LAST_NAME	VACATION_HOURS	SALARY
1	1	Charles	Patrick	Brown	20	NULL
2	2	Neil		Armstrong	50	100000
3	3	Rich	ReallyRich	Rich	90	500000

Total 3 records shown

Alternatives:

As in the previous sections, the rows could be added through the Derby **`ij`** command line or through the WASCE database management function. In the WASCE Console you can click **Application** in the database list, and **View Contents** for the table, and the data is displayed.

4.6 Connect to the Donate database in the WASCE IDE

The second database in the application is the `Donate` database that holds the `FUND` table for vacation donations of employees. We create a connection to the database and also create the database. The connection will be used by the JPA tooling for synchronization when the database table has been generated.

This step could be deferred to 5.7, "Create the Fund entity" on page 142, but we perform the step now because we just completed similar steps in 4.3, "Create the Vacation database" on page 80, and because placing this step here simplifies the jump start process:

1. In the Data Source Explorer, right-click **Databases** and select **New**.
2. In the New Connection Profile/Connection Selection Page pop-up:
   - Under Connection Profile types, select **Derby**.
   - Set name to **Donate**.
   - Click **Next**.
3. In the New Derby Connection Profile/Specify a Driver and Connection Details pop-up:
   - From the Drivers pull-down, select **Derby Client JDDC Driver**.
   - Under Properties:
     - Set Database to **Donate**.
     - Set Host to **localhost**.
     - Set Port number to **1527**.
     - Set Username to **DONATE**.
     - Set Password to any value (such as `donate`).
   - Select **Create database** (at first access).
   - Select **Save password**.
   - Select **Connect when the wizard completes**.
   - Further down, click **Test Connection** to verify that the database properties were correctly specified. This step creates the `Donate` database in `c:\IBM\WASCE21\var\derby\Donate`.
   - Click **Finish**.
4. The Data Source Explorer view contains the `Donate` database connection.

4.7 Create the WASCE Vacation database data source

The Vacation database will be accessed by our sample application, and requires a data source definition in the WASCE server:

1. Open or switch back to the Java EE perspective, and select the Server view (in the lower right pane).

   – You can also close the Database Development perspective because you will not use it again in this book.

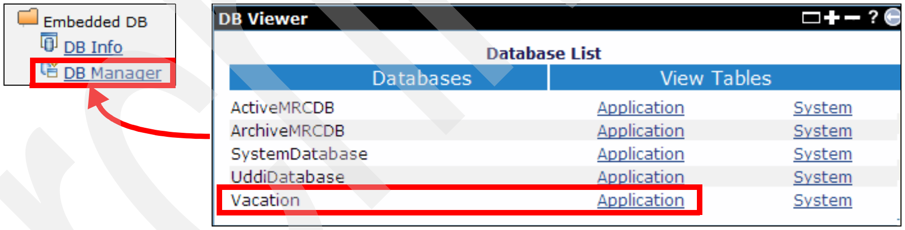
2. If the ExperienceJavaEE Server shows a state of stopped, start it using the steps described in 3.6, "Start the ExperienceJavaEE Server test server" on page 66.

   The configuration of the data source is performed in the WASCE Console and thus requires that the test server be started.

3. To start the WASCE Console, right-click the **ExperienceJavaEE Server** and select **Launch WASCE Console**.

4. The console opens for login. Type the **system** user ID and password, by default, **manager**, and click **Login**.

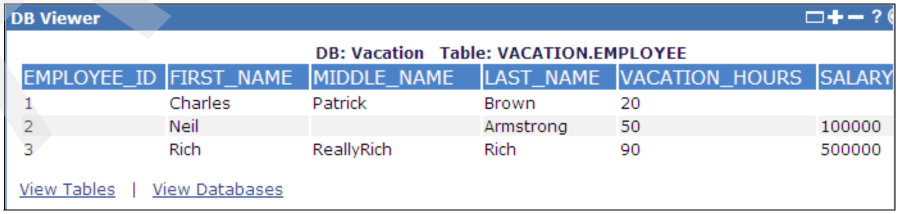
5. In the Console Navigation, under **Embedded DB**, select **DB Manager**. The list of databases is displayed on the right.



Database List

Databases	View Tables	
ActiveMRCDB	Application	System
ArchiveMRCDB	Application	System
SystemDatabase	Application	System
UddiDatabase	Application	System
Vacation	Application	System

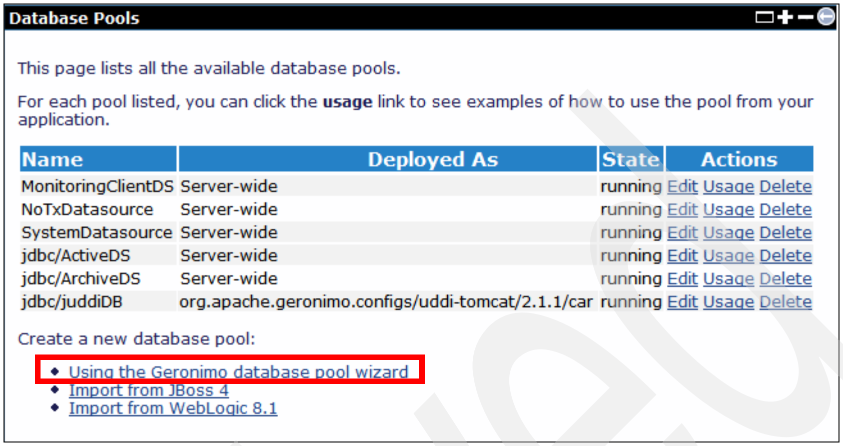
6. The Vacation database is in the list because we already defined it. Click **Application** to see the list of tables. The EMPLOYEE table is listed. Click **View Contents** to see the three employees.



DB: Vacation Table: VACATION.EMPLOYEE

EMPLOYEE_ID	FIRST_NAME	MIDDLE_NAME	LAST_NAME	VACATION_HOURS	SALARY
1	Charles	Patrick	Brown	20	
2	Neil		Armstrong	50	100000
3	Rich	ReallyRich	Rich	90	500000

7. In the Console Navigation, under **Services**, select **Database Pools**. The list of pools is displayed on the right.



Database Pools

This page lists all the available database pools.

For each pool listed, you can click the **usage** link to see examples of how to use the pool from your application.

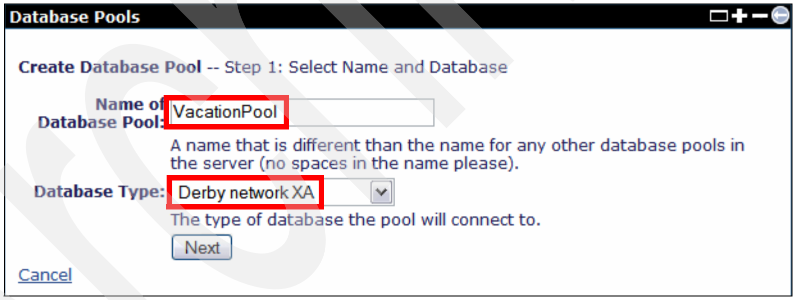
Name	Deployed As	State	Actions
MonitoringClientDS	Server-wide	running	Edit Usage Delete
NoTxDatasource	Server-wide	running	Edit Usage Delete
SystemDatasource	Server-wide	running	Edit Usage Delete
jdbc/ActiveDS	Server-wide	running	Edit Usage Delete
jdbc/ArchiveDS	Server-wide	running	Edit Usage Delete
jdbc/juddiDB	org.apache.geronimo.configs/uddi-tomcat/2.1.1/car	running	Edit Usage Delete

Create a new database pool:

- Using the Geronimo database pool wizard
- Import from JBoss 4
- Import from WebLogic 8.1

8. To define a database pool for the `Vacation` database, click **Using the Geronimo database pool wizard**.

9. The wizard opens. Type **VacationPool** as name and select **Derby network XA** as database type. Click **Next**.



Database Pools

Create Database Pool -- Step 1: Select Name and Database

Name of Database Pool: VacationPool

A name that is different than the name for any other database pools in the server (no spaces in the name please).

Database Type: Derby network XA

The type of database the pool will connect to.

10. Enter the details for the database pool:
   - Driver JAR: Select the JAR that is listed.
   - Database Name: **Vacation**
   - Password: **password**
   - Server Name: Accept **localhost**
   - User Name: **VACATION** (this defines the schema for tables)
   - Create Database: **true**
   - Port Number: Accept **1527**
   - Leave the other fields empty to accept the defaults.

11. Click **Show Plan** to see the XML definition of the database pool. This XML file can be saved to easily define this pool in another WASCE server.

   – During the development of this book, a copy of all the generated deployment plans were saved in `c:\7639code\plans`. You should find that the information on this screen is identical to the `VacationPool-plan.xml` file found in that directory.

Behind the scenes:

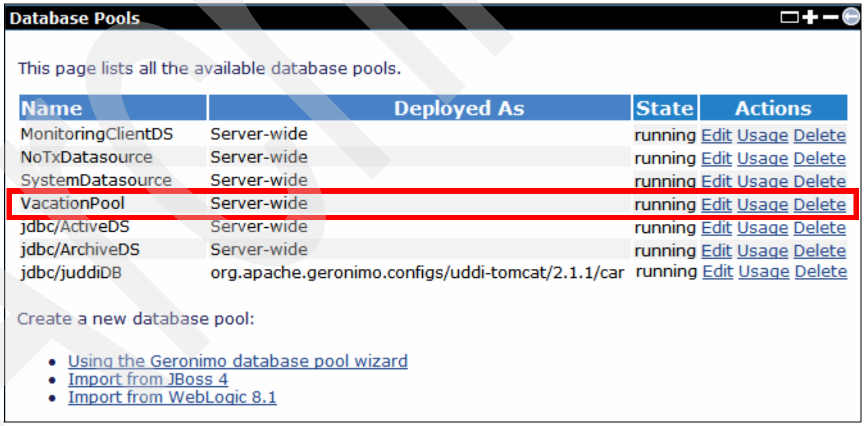
The `VacationPool-plan.xml` file can be used to configure the necessary WASCE database resources for the `Vacation` database by running the following command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory:

```
deploy -u system -p manager deploy
   C:\7639code\plans\VacationPool-plan.xml
   ..\repository\org\tranql\tranql-connector-derby-client-xa\1.4
                         \tranql-connector-derby-client-xa-1.4.rar
```

The command completes with the following message:

```
Deployed console.dbpool/VacationPool/1.0/rar
```

12. Click **Deploy Pool**. The `VacationPool` is added to the list.



Database Pools

This page lists all the available database pools.

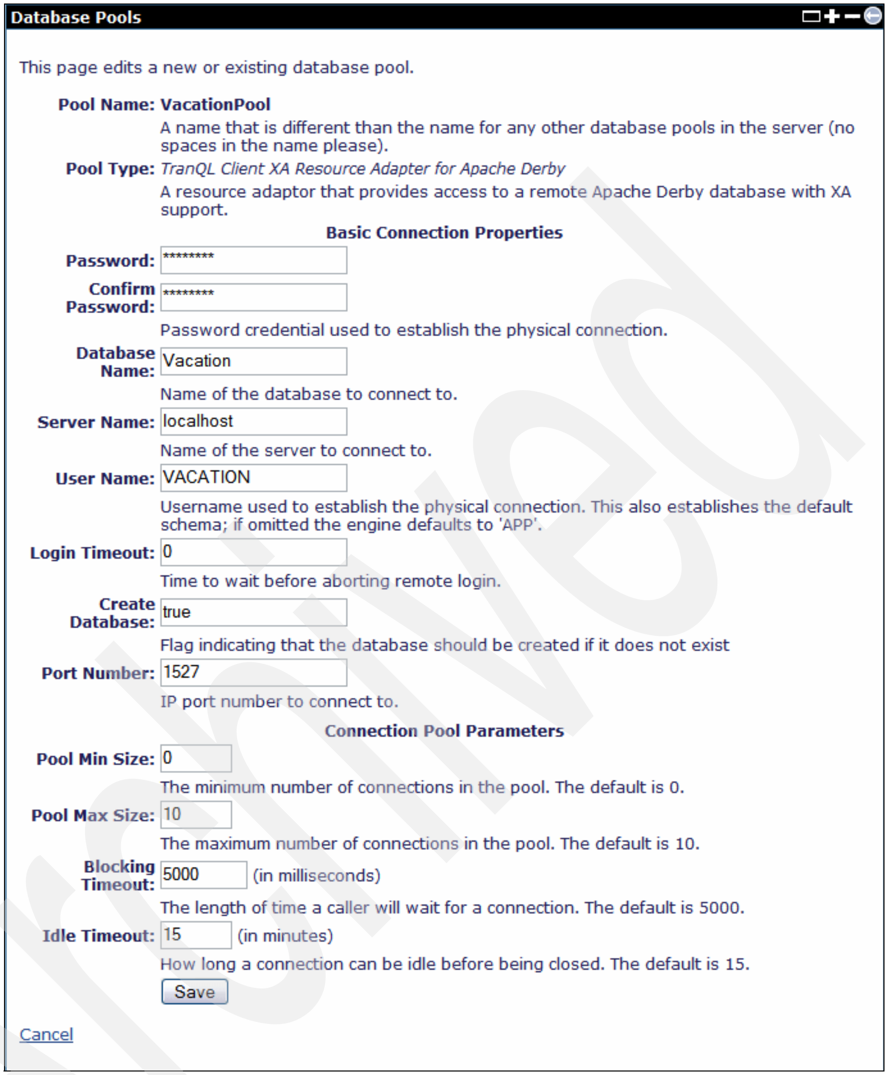
Name	Deployed As	State	Actions
MonitoringClientDS	Server-wide	running	Edit Usage Delete
NoTxDatasource	Server-wide	running	Edit Usage Delete
SystemDatasource	Server-wide	running	Edit Usage Delete
VacationPool	Server-wide	running	Edit Usage Delete
jdbc/ActiveDS	Server-wide	running	Edit Usage Delete
jdbc/ArchiveDS	Server-wide	running	Edit Usage Delete
jdbc/juddiDB	org.apache.geronimo.configs/uddi-tomcat/2.1.1/car	running	Edit Usage Delete

Create a new database pool:

- Using the Geronimo database pool wizard
- Import from JBoss 4
- Import from WebLogic 8.1

13. Click **Edit** for the `VacationPool` in the list to see the details or to make changes.

   – Click **Cancel** at the bottom of the screen when you have finished.



Database Pools

This page edits a new or existing database pool.

Pool Name:	**VacationPool** A name that is different than the name for any other database pools in the server (no spaces in the name please).
Pool Type:	*TranQL Client XA Resource Adapter for Apache Derby* A resource adaptor that provides access to a remote Apache Derby database with XA support.
	Basic Connection Properties
Password:	********
Confirm Password:	******** Password credential used to establish the physical connection.
Database Name:	Vacation Name of the database to connect to.
Server Name:	localhost Name of the server to connect to.
User Name:	VACATION Username used to establish the physical connection. This also establishes the default schema; if omitted the engine defaults to 'APP'.
Login Timeout:	0 Time to wait before aborting remote login.
Create Database:	true Flag indicating that the database should be created if it does not exist
Port Number:	1527 IP port number to connect to.
	Connection Pool Parameters
Pool Min Size:	0 The minimum number of connections in the pool. The default is 0.
Pool Max Size:	10 The maximum number of connections in the pool. The default is 10.
Blocking Timeout:	5000 (in milliseconds) The length of time a caller will wait for a connection. The default is 5000.
Idle Timeout:	15 (in minutes) How long a connection can be idle before being closed. The default is 15.

14. Click **Usage** for the `VacationPool` in the list to get information on how to use the database pool in a Web application:

 – Define a resource reference in the `web.xml` file:

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

– Define a dependency in the geronimo-web.xml file:

```
<web-app xmlns=........>
    <environment>
        <moduleId>
            <artifactId>MyWebApp</artifactId>
        </moduleId>
        <dependencies>
            <dependency>
                <groupId>console.dbpool</groupId>
                <artifactId>VacationPool</artifactId>
            </dependency>
        </dependencies>
    </environment>
    <context-root>/MyWebApp</context-root>
    <resource-ref>
        <ref-name>jdbc/MyDataSource</ref-name>
        <resource-link>VacationPool</resource-link>
    </resource-ref>
</web-app>
```

– Click **Return to list** when done.

15. The Vacation database is now ready for access by applications running in the WASCE server.

Behind the scenes:

The database pool wizard makes two changes to the WASCE configuration:

- Creates the complete VacationPool definition in the C:\IBM\WASCE21\repository\console\dbpool\VacationPool subdirectory.
- Adds the VacationPool to the WASCE runtime configuration file C:\IBM\WASCE21\var\config\config.xml:

```
<module name="console.dbpool/VacationPool/1.0/rar"/>
```

4.8 Create the WASCE Donate database data source

Our sample application also uses a Donate database to store information about donations funds:

1. In the Console Navigation, under **Services**, select **Database Pools**. The list of pools is displayed on the right.
2. To define a database pool for the DONATE database, click **Using the Geronimo database pool wizard**.

3. The wizard opens. Type **DonatePool** as name and select **Derby network XA** as database type. Click **Next**.
4. Enter the details for the database pool:
   - Driver JAR: Select the JAR that is listed
   - Database Name: **Donate**
   - Password: **password**
   - Server Name: Accept **localhost**
   - User Name: **DONATE** (this defines the schema for tables)
   - Create Database: **true**
   - Port Number: Accept **1527**
   - Leave the other fields empty to accept the defaults
5. Click **Show Plan** to see the XML definition of the database pool.
   - As before, you will find the equivalent plan generated during the development of this book in `c:\7639code\plans\DonatePool-plan.xml`.
6. Click **Deploy Pool**. The `DonatePool` is added to the list.
7. Click **Logout** to log out of the WASCE Console.
8. Close the **ExperienceJavaEE Server Console** window.

4.9 Explore!

Additional Explore resources are available at the following Web site:

- Derby administration guide (containing information on the Network Server):

  `http://db.apache.org/derby/docs/dev/adminguide`

Part 2

Core Java EE application

In Part 2, we describe how to implement a traditional Java EE application, starting with a single back-end relational database called `Vacation`, which we created in Chapter 4, “Prepare the legacy application” on page 77, and ending with a multi-tier application that implements the core Java EE elements of persistent entities, session EJBs, and JSPs, supporting both Web browsers and a thin Java client.

5

Create the JPA entities

In this chapter, we create two JPA entities that coordinate and mediate access with two Derby databases (Figure 5-1):

- The first entity, `Employee`, is created in a *bottom-up scenario* from the existing `EMPLOYEE` table in the `Vacation` database.
- The second entity, `Fund`, is created in a *top-down scenario* and a matching `FUND` table in the `Donate` database is created for it.

5.1 Learn!

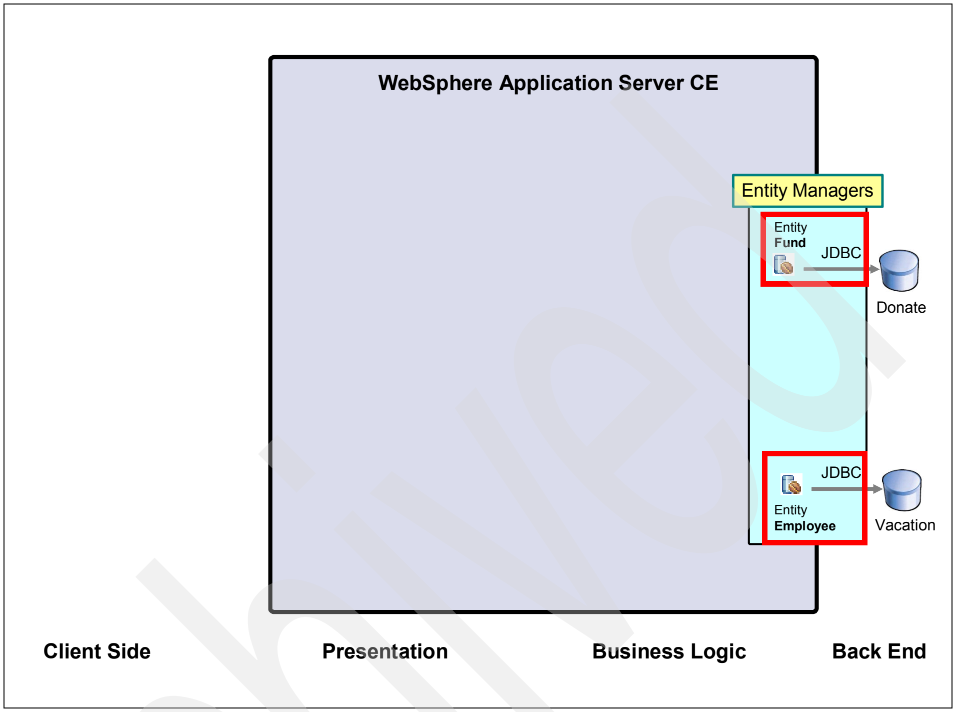


Figure 5-1 Donate application: JPA entities and database mapping

The Learn section of this chapter is an extract from the IBM Redbooks publication, *WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0*, SG24-7611, *Chapter 2, Introduction to JPA*.

The persistence layer in a typical J2EE application that interacts with a relational database has been implemented over the last years in several ways:

- EJB 2.x entity beans
- Data access object (DAO) pattern
- Data mapper frameworks (such as IBatis)
- Object-relational mapping (ORM) frameworks both commercial (such as Oracle Toplink) or from the open source world (such as Hibernate)

Data mapper and ORM frameworks gained a great approval among developers communities, because they give a concrete answer to the demand of simplification of the design of the persistence layer, and let developers concentrate on the business related aspects of their solutions.

However, even if these frameworks overcome the limitations of EJB 2.x CMP entity beans, one of the common concerns related to the adoption of such frameworks was that they were not standardized.

One of the main innovative concepts introduced by EJB 3.0 is the provisioning of a single persistence standard for the Java platform that can be used in both the Java EE and Java SE environments, and that can be used to build the persistence layer of a Java EE application. Furthermore, it defines a pluggable service interface model, so that you can plug in different provider implementations, without significant changes to your application.

The Java Persistence API provides an object relational mapping facility to Java developers for managing relational data in Java applications. Java persistence consists of three areas:

- Java Persistence API
- Object-relational mapping metadata
- Query language

5.1.1 Entities

In the EJB 3.0 specification, entity beans have been substituted by the concept of entities, which are sometimes called entity objects to clarify the distinction between EJB 2.1 entity beans and JPA entities (entity objects).

A JPA entity is a Java object that must match the following rules:

- It is a plain old Java object (POJO) that does not have to implement any particular interface or extend a special class.
- The class must not be declared final, and no methods or persistent instance variables must be declared final.
- The entity class must have a no-argument constructor that is public or protected. The entity class can have other constructors as well.
- The class must either be annotated with the @Entity annotation or specified in the `orm.xml` JPA descriptor. We use annotations in our examples.
- The class must define an attribute that is used to identify in an unambiguous way an instance of that class (it corresponds to the primary key in the mapped relational table).

- Both abstract and concrete classes can be entities, and entities can extend non-entity classes (this is a significant limitation with EJB 2.x).

> **Note:** Entities have overcome the traditional limitation that is present in EJB 2.x entity beans: They can be transferred *on the wire* (for example, they can be serialized over RMI-IIOP). Of course, you must remember to implement the `java.io.Serializable` interface in the entity.

A simple entity example

Example 5-1 shows a simple `Customer` entity with a few fields.

Example 5-1 Simple entity class with annotations

```
package itso.bank.entities;

@Entity
public class Customer implements java.io.Serializable {

   @Id
   private String ssn;

   private String title;
   private String firstname;
   private String lastname;

   public String getSsn() { return this.ssn; }
   public void setSsn(String ssn) { this.ssn = ssn; }

   // more getter and setter methods
}
```

- The **@Entity** annotation identifies a Java class as an entity.
- The **@Id** annotation is used to identify the property that corresponds to the primary key in the mapped table.
- The class is conforming to the JavaBean specification.

5.1.2 Mapping the table and columns

To specify the mapping of the entity to a database table, we use @Table and @Column annotations (Example 5-2).

Example 5-2 Entity with mapping to a database table

```
@Entity
@Table (schema="ITSO", name="CUSTOMER")
public class Customer implements java.io.Serializable {
```

```
@Id
@Column (name="SSN")
private String ssn;
@Column (name="NAME")
private String lastname;
private String title;
private String firstname;
......
```

- The **@Table** annotation provides information related to which table and schema the entity corresponds to.
- The **@Column** annotation provides information related to which column is mapped by an entity property. By default, properties are mapped to columns with the same name, and the @Column annotation is used when the property and column names differ.

> **Note:** Entities support two types of persistence mechanisms:
>
> - Field-based persistence: The entity properties must be declared as public or protected and instruct the JPA provider to ignore getter/setters.
> - Property-based persistence: You must provide getter/setter methods.
>
> We recommend using the property-based approach, because it is more adherent to the Java programming guidelines.

The @Column annotation has further attributes that can be used to completely specify the mapping of an field/property (Table 5-1).

Table 5-1 @Column attributes

Attribute	Description
name	The name of the column
unique	True for UNIQUE columns
nullable	False for IS NOT NULL columns
insertable	True if the column is inserted on a create call
updatable	True if the column is updated when the field is modified
columnDefinition	SQL to create the column in a CREATE TABLE statement
table	Specified if the column is stored in a secondary table
length	Default length for a VARCHAR for CREATE TABLE

Attribute	Description
precision	Default length for a number definition for CREATE TABLE
scale	Default scale for a number definition for CREATE TABLE

The fields or properties supported by entities are:

- Java primitive types (`byte`, `int`, `short`, `long`, `char`).
- `java.lang.String`
- Other serializable types, including:
  - Wrappers of Java primitive types (`Byte`, `Integer`, `Short`, `Long`, `Character`)
  - `java.math.BigInteger`, `java.math.BigDecimal`
  - `java.util.Date`, `java.util.Calendar`, `java.sql.Date`
  - `java.sql.Time`, `java.sql.TimeStamp`
  - `byte[]`, `Byte[]`, `char[]`, `Character[]`
  - User-defined serializable types
- Enumerated types
- Other entities and/or collections of entities, such as:
  - `java.util.Collection`
  - `java.util.Set`
  - `java.util.List`
  - `java.util.Map`
- Embeddable classes

Note on inheritance: Entities can extend another entity. In this case all the inherited properties are automatically persisted as well. This is not true if the entity extends a class that is not declared as an entity.

Basic fields

The term *basic* is associated with a standard value persisted as-is to the data store.

```
@Basic
@Column(name="NAME")
private String lastname;
```

You can use the **@Basic** annotation on persistent fields of the following types:

- Primitives and primitive wrappers
- `java.lang.String`
- `byte[]`, `Byte[]`, `char[]`, `Character[]`
- `java.math.BigDecimal`, `java.math.BigInteger`

- `java.util.Date`, `java.util.Calendar`, `java.sql.Date`
- `java.sql.Timestamp`
- Enums
- Serializable types

This annotation is used very often when you want to specify a fetch policy.

Transient fields

If you model a field inside an entity that you do not want to be persistent, you can use the @Transient annotation on the field/getter method (or simply using the transient modifier:

```
@Transient
private String fieldNotPersistent1;

transient String fieldNotPersistent2;
```

5.1.3 Entity identity

Every entity that is mapped to a relational database must have a mapping to a primary key in the table.

The JPA specification supports three different ways to specify the entity identity:

- @Id annotation
- @IDClass annotation
- @EmbeddedId annotation

@Id annotation

The @Id annotation offers the simplest mechanism to define the mapping to the primary key.

You can associate the @Id annotation to fields/properties of these types:

- Primitive Java types and their wrapper classes
- Arrays of primitive or wrapper types
- Strings: `java.lang.String`
- Large numeric types: `java.math.BigDecimal`, `java.math.BigInteger`
- Temporal types: `java.util.Date`, `java.sql.Date`

We discourage the usage of floating point types (float and double, and their wrapper classes) for decimal data, because you can have rounding errors and the result of the equals operator is unreliable in these cases. Use `BigDecimal` instead. Temporal types can have similar problems due to time-based inconsistencies.

The @Id annotation fits very well in scenarios where a natural primary key is available, or when database designers use *surrogate primary keys* (typically an integer) that has no descriptive value and is not derived from any application data in the database.

On the other end, composite keys are useful when the primary key of the corresponding database table consists of more than one column. Composite keys can be defined by the @IdClass or @EmbeddedId annotation.

@IDClass

In this section we illustrate how to model a composite key using the @IDClass annotation.

To map this composite key, follow these steps:

- Create a class that models the primary key, for example, CustomerKey:

  ```
  public class CustomerKey implements Serializable {
     private int keyfield1;
     private String keyfield2;
     ......
  }
  ```

  - The key class must implement the java.io.Serializable interface.
  - It overrides the equals and hashCode methods in order to use organizationalUnit and userName properties for the definition of Employee identity.
  - It has no setter methods, because once it has been constructed using the primary key values, it cannot be changed.

- Specify in the entity that we want to use the key class as our identity provider:

  ```
  @IdClass(value=CustomerKey.class)
  @Entity
  public class Customer implements java.io.Serializable {
     @id
     private int keyfield1;
     @id
     private String keyfield2;
     ......
  }
  ```

@EmbeddedId

Using the **@EmbeddedId** annotation in conjunction with the @Embeddable annotation, the definition of the composite key is moved inside the entity:

```
@Entity
public class Customer implements Serializable {
```

```
    @EmbeddedId
    private CustomerKey key;
    ......
}
@Embeddable class CustomerKey {
    ......
}
```

The main feature of this refactoring are as follows:

- The key class does not have to implement `Serializable`.
- The key class is now nested (embedded) inside the entity class and therefore can be used only in conjunction with the entity.
- The entity does not redeclare the fields associated with the composite key.

> **Note:** Generally speaking, the @Embeddable annotation is used to model persistent objects that have no identity of their own, because they are nested inside another entity.

Entity automatic identity generation

Very often applications do not want to care explicitly about the identity of their managed entities, but instead require that identifier values be automatically generated for them. By default, the JPA persistence provider manages the provision of unique identifiers for entity primary keys using the @Id annotation.

This is where the **@GeneratedValue** annotation helps. JPA supports four different strategies for automatic identity generation (Table 5-2).

Table 5-2 identity generation strategies

Generation strategy	Description
`GenerationType.IDENTITY`	Specifies that the persistence provider uses a database identity column.
`GenerationType.SEQUENCE`	Specifies that the persistence provider use a database sequence (in conjunction with the @SequenceGenerator annotation).
`GenerationType.AUTO`	Specifies that the persistence provider should select a primary key generator that is most appropriate for the underlying database.

Generation strategy	Description
`GenerationType.TABLE`	Specifies that the persistence provider assigns primary keys for the entity using an underlying database table to ensure uniqueness (in conjunction with the @TableGenerator annotation).

ID generation using database identity

Some databases support a primary key identity column sometimes referred to as an autonumber column. In such cases, an identity column provides a way for these RDBMS to automatically generate a unique numeric value for each row in a table. A table can have a single column that is defined with the identity attribute.

ID generation using database sequence

A sequence is a database object that allows the automatic generation of numeric values.

While the identity column feature also allows automatic generation of numeric values, it is essentially an attribute of a column and thus exists for the period of existence of the table containing that column. On the other hand, a sequence exists outside of a table and is not tied to a column of a table. This allows more flexibility in using the sequence values in SQL operations.

Automatic ID generation

With this strategy, the persistence provider uses any strategy to generate identifiers.

ID generation using a table generator

Table generator technique uses a table in the database to generate unique IDs. The table has two columns, one stores the name of the sequence, the other stores the last ID value that was assigned.

There is a row in the sequence table for each sequence object. Each time a new ID is required the row for that sequence is incremented and the new ID value is passed back to the application to be assigned to an object.

> **Note:** Table generator is the most portable solution because it just uses a regular database table, so unlike sequence and identity can be used on any RDBMS.

In JPA the @TableGenerator annotation is used to define a sequence table. The table generator defines `pkColumnName` for the column used to store the name of the sequence, `valueColumnName` for the column used to store the last ID allocated, and `pkColumnValue` for the value to store in the name column (normally the sequence name).

5.1.4 Callback methods and listeners

JPA provides callback methods for performing actions at different stages of persistence operations. The callback methods can be annotated as follows:

- @PostLoad
- @PrePersist
- @PostPersist
- @PreUpdate
- @PostUpdate
- @PreRemove
- @PostRemove

For these annotations, the JSR 220 specification indicates:

- The @PrePersist and @PreRemove callback methods are invoked for a given entity before the respective persist and remove operations for that entity are executed by the persistence manager.
- The @PostPersist and @PostRemove callback methods are invoked for an entity after the entity has been made persistent or removed. These callbacks are also invoked on all entities to which these operations are cascaded. The @PostPersist and @PostRemove methods will be invoked after the database insert and delete operations respectively.
- The @PreUpdate and @PostUpdate callbacks occur before and after the database update operations to entity data. These database operations can occur at the time the entity state is updated, or they can occur at the time state is flushed to the database (which might be at the end of the transaction). Note that it is implementation-dependent as to whether @PreUpdate and @PostUpdate callbacks occur when an entity is persisted and subsequently modified in a single transaction, or when an entity is modified and subsequently removed within a single transaction. Portable applications should not rely on such behavior.
- The @PostLoad callback is invoked after the entity has been loaded into the current persistence context from the database or after the refresh operation has been applied to it. The @PostLoad method is invoked before a query result is returned or accessed or before an association is traversed.

These callback methods can be inserted in the entity class itself, or in a separate class and reference them in the entity class with a @EntityListeners annotation.

5.1.5 Relationships

Before starting our discussion of entity relationships, it is useful to refresh how the concept of relationships is defined in object-oriented and in relational database worlds (Table 5-3).

Table 5-3 Relationship concept in two different worlds

Java/JPA	RDBMS
A relationship is a reference from one object to another. Relationships are defined through object references (pointers) from a source object to the target object.	Relationships are defined through foreign keys.
If a relationship involves a collection of other objects, a collection or array type is used to hold the contents of the relationship.	Collections are either defined by the target objects have a foreign key back to the source object's primary key, or by having an intermediate join table to store the relationships.
Relationships are always unidirectional, in that if a source object references a target object, it is not guaranteed that the target object also has a relationship to the source object.	Relationships are defined through foreign keys and queries, such that the inverse query always exists.

JPA defines the following relationships: one-to-one, many-to-one, one-to-many, and many-to-many.

One-to-one relationship

In this type of relationship each entity instance is related to a single instance of another entity. The @OneToOne annotation is used to define this single value association, for example, a `Customer` is related to a `CustomerRecord`:

```
@Entity
public class Customer {

   @OneToOne
   @JoinColumn(
      name="CUSTREC_ID", unique=true, nullable=false, updatable=false)
   public CustomerRecord getCustomerRecord() {
      return customerRecord;
   }
   ....
}
```

In many situations the target entity of the one-to-one has a relationship back to the source entity. In our example, `CustomerRecord` could have a reference back to the `Customer`. When this is the case, we call it a bidirectional one-to-one relationship.

There are two rules for bi-directional one-to-one associations:

- The @JoinColumn annotation must be specified in the entity that is mapped to the table containing the join column, or the owner of the relationship.
- The `mappedBy` element should be specified in the @OneToOne annotation in the entity that does not define a join column, that is, the inverse side of the relationship.

Many-to-one and one-to-many relationships

Many-to-one mapping is used to represent simple pointer references between two Java objects. In Java, a single pointer stored in an attribute represents the mapping between the source and target objects. Relational database tables implement these mappings using foreign keys.

On the other side, one-to-many mapping is used to represent the relationship between a single source object and a collection of target objects. This relationship is usually represented in Java with a collection of target objects, but is more difficult to implement using relational databases (where you retrieve related rows through a query).

For example, an `Account` entity object can be associated with many `Transaction` entity objects:

```
@Entity
@Table (schema="ITSO", name="ACCOUNT")
public class Account implements Serializable {
   @Id
   private String id;
   private BigDecimal balance;

   @OneToMany(mappedBy="account")
   private Set<Transaction> transactionCollection;

   ....
}

@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
public class Transaction implements Serializable {
   @Id
   private String id;
   ......
   @ManyToOne
   @JoinColumn(name="ACCOUNT_ID")
   private Account account;
   ....
}
```

Using the @JoinColumn annotation

In the database, a relationship mapping means that one table has a reference to another table. The database term for a column that refers to a key (usually the primary key) in another table is a foreign key column.

In the Java Persistence API we call them join columns, and the @JoinColumn annotation is used to configure these types of columns.

> **Note:** If you do not specify @JoinColumn, then a default column name is assumed. The algorithm used to build the name is based on a combination of both the source and target entities. It is the name of the relationship attribute in the Transaction source entity (the **`account`** attribute), plus an underscore character (_), plus the name of the primary key column of the target Account entity (the **`id`** attribute).
>
> Therefore a foreign key named **`ACCOUNT_ID`** is expected inside the `TRANSACTION` table. If this is not applicable, you must use @JoinColumn to override this automatic behavior.
>
> The @JoinColumn annotation also applies to one-to-one relationships.

Many-to-many relationship

When an entity A references multiple B entities, and other As might reference some of the same Bs, we say there is a many-to-many relation between A and B. To implement a many-to-many relationship there must be a distinct join table that maps the many-to-many relationship. This is called an association table.

For example, a `Customer` entity object can be associated with many `Account` entity objects, and an `Account` entity object can be associated with many `Customer` entity objects using the @ManyToMany annotation:

```
@Entity
public class Customer implements Serializable {
   ......

   @ManyToMany
   @JoinTable(name="ACCOUNTS_CUSTOMERS", schema="ITSO",
      joinColumns=@JoinColumn(name="CUSTOMERS_SSN"),
      inverseJoinColumns=@JoinColumn(name="ACCOUNTS_ID") )
   private Set<Account> accountCollection;
   ......
}

@Entity
public class Account implements Serializable {
   ......

   @ManyToMany(mappedBy="accountCollection")
   private Set<Customer> customerCollection;
....
}
```

The @JoinTable annotation is used to specify a table in the database that associates customers with accounts. The entity that specifies the @JoinTable is the owner of the relationship, so in this case the `Customer` entity is the owner of the relationship with the `Account` entity.

The join column pointing to the owning side is described in the `joinColumns` element, while the join column pointing to the inverse side is specified by the `inverseJoinColumns` element.

> **Note:** Neither the `CUSTOMER` nor the `ACCOUNT` table contains a foreign key. The foreign keys are in the association table. Therefore, the `Customer` or the `Account` entity can be defined as the owning entity.

Fetch modes

When an entity manager retrieves an entity from the underlying database, it can use two types of strategies:

- **Eager mode**: When you retrieve an entity from the entity manager or by using a query, you are guaranteed that all of its fields (with relationships too) are populated with data store data.
- **Lazy mode**: This is a hint to the JPA runtime that you want to defer loading of the field until you access it. Lazy loading is completely transparent, when you attempt to read the field for the first time, the JPA runtime will load the value from the data store and populate the field automatically:

```
@OneToMany(mappedBy="accounts", fetch=FetchType.LAZY)
private Set<Transaction> transactionCollection;
```

Note: The use of eager mode can greatly impact the performance of an application, especially if entities have many and recursive relationships, because all the entity will be loaded at once.

On the other hand, if the entity after that has been read by the entity manager, is detached and sent over the network to another layer, you usually should assure that all the entity attributes have been read from the underlying data store, or that the receiver does not require related entities.

The default value of the fetch strategy changes according to where it is used (Table 5-4).

Table 5-4 Default fetch strategies

Context	Default fetch mode
@Basic	EAGER
@OneToMany	LAZY
@ManyToOne	EAGER
@ManyToMany	LAZY

Cascade types

When the entity manager is reading, updating, or deleting an entity, you can instruct it to automatically cascade the operation to the entities held in a persistent field with the *cascade* property of the metadata annotation. This process is recursive:

```
@OneToMany(mappedBy="accounts",
           cascade={CascadeType.PERSIST,CascadeType.REMOVE})
private Set<Transaction> transactionCollection;
```

The cascade property accepts an array of `CascadeType` enum values (Table 5-5).

Table 5-5 Cascade rules

Cascade rule	Description
`CascadeType.PERSIST`	When persisting an entity, also persist the entities held in this field. This rule should always been used if you want to guarantee all relationships linked to be automatically persisted when a entity field/attribute is modified.
`CascadeType.REMOVE`	When deleting an entity, also delete the entities held in this field.

Cascade rule	Description
`CascadeType.REFRESH`	When refreshing an entity, also refresh the entities held in this field.
`CascadeType.MERGE`	When merging entity state, also merge the entities held in this field.
`CascadeType.ALL`	This rule acts as a shortcut for all of the foregoing values.

5.1.6 Entity inheritance

For several years we heard the term *impedance mismatch*, which describes the difficulties in bridging the object and relational worlds. Unfortunately, there is no natural and efficient way to represent an inheritance relationship in a relational database.

JPA introduces three strategies to support inheritance:

- Single table
- Joined tables
- Table per class

Single table inheritance

This strategy maps all classes in the hierarchy to the base class table. This means that the table contains the superset of all the data contained in the class hierarchy.

The main advantage of this strategy is that is the fastest of all inheritance models, because it never requires a join to retrieve a persistent instance from the database. Similarly, persisting or updating a persistent instance requires only a single INSERT or UPDATE statement.

However, if the hierarchy model becomes wide or deep, the mapped table must have columns for every field in the entire inheritance hierarchy, which results in designing tables with nullable columns, that will be mostly empty.

For example, if the `Transaction` entity has two subclasses `Credit` and `Debit`, and the underlying table has a *discriminator* column named `transtype` to distinguish between the subclasses, then we specify single table inheritance using the @Inheritance, @DiscriminatorColumn, and @DiscriminatorValue annotations:

```
@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="transtype",
                        discriminatorType=DiscriminatorType.STRING, length=32)
public abstract class Transaction implements Serializable {
...
}

@Entity
@DiscriminatorValue("Debit")
public class Debit extends Transaction {
...
}

@Entity
@DiscriminatorValue("Credit")
public class Credit extends Transaction {
...
}
```

Joined tables inheritance

With this strategy, the top level entry in the entity hierarchy is mapped to a table that contains columns common to all the entities, while each of the other entities down the hierarchy are mapped to a table that contain only columns specific to that entity.

To reassemble an instance of any of the subclasses, the tables of the subclasses must be joined together with the superclass tables. That is why this strategy is called the joined strategy.

For each subclass, we use the @Table annotation to specify the joined table, the @DiscrimininatorValue annotation to specify the value in the discriminator column, and the @PrimaryKeyJoinColumn annotation to specify the foreign key column:

```
@Entity
@Table(schema="ITSO", name="CREDIT")
@DiscriminatorValue(value="Credit")
@PrimaryKeyJoinColumn(name="TRANS_ID")
public class Credit extends Transaction {
...
}
```

Table per class inheritance

With this strategy both the superclass and subclasses are stored in their own table and no relationship exists between any of the tables. Therefore, all the entity data are stored in their own tables.

Because every entity is mapped to a separate table, you must specify the @Table annotation for each of them.

The table-per-class strategy is very efficient when operating on instances of a known class, because you do not have to join to superclass or subclass tables. Reads, joins, inserts, updates, and deletes are all efficient in the absence of polymorphic behavior. Also, as in the joined strategy, adding additional classes to the hierarchy does not require modifying existing class tables.

However, with this strategy, when the concrete subclass is not known, and therefore the related object could be in any of the subclass tables, you cannot use joins, because there are no relationships. This issue involves identity lookups and queries too, and these operations require multiple SQL SELECTs (one for each possible subclass), or a complex UNION, and therefore you might have performance problems as well.

Because of these drawbacks, the table per class inheritance implementation is not used much in practice.

5.1.7 Persistence units

A persistence unit defines a set of all entity classes that are managed by entity manager (described hereafter) instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the `persistence.xml` configuration file. The JAR file or directory whose `META-INF` directory contains `persistence.xml` is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit's root.

Example 5-3 shows an extract of the `persistence.xml` file.

Example 5-3 Extract of a persistence.xml file

```
<persistence version="1.0" ..........>
   <persistence-unit name="EJB3Bank" transaction-type="JTA">
      <jta-data-source>jdbc/ejb3bank</jta-data-source>
      <non-jta-data-source>jdbc/itsoejb30nojta</non-jta-data-source>
      <class>itso.bank.entities.Account</class>
      <class>itso.bank.entities.Customer</class>
      <class>itso.bank.entities.Transactions</class>
   </persistence-unit>
</persistence>
```

Each persistence unit must be identified with a name that is unique to the persistence unit's scope. Persistent units can be packaged as part of a WAR or EJB JAR file, or can be packaged as a JAR file that can then be included in a WAR or EAR file. If you package the persistent unit as a set of classes in an EJB JAR, the `persistence.xml` file should be put in the EJB JAR's `META-INF` directory.

In Java EE, the root of a persistence unit can be one of the following items:

- An EJB JAR file
- The `WEB-INF/classes` directory of a WAR file
- A JAR file in the `WEB-INF/lib` directory of a WAR file
- A JAR file in the root of the EAR
- A JAR file in the EAR library directory
- An application client JAR file

Table 5-6 illustrates the meaning of the main elements of this file.

Table 5-6 Main elements in the persistence.xml file

Element	Description
`name`	Identifies the persistence unit and is used to refer the persistence unit with annotation injection or deployment descriptors. The name must be unique within the scope of the persistence unit packaging.
`provider`	Specifies the persistence provider; this element is optional and usually is not specified, so that the default implementation provided by the Java EE application server is used.
`transaction-type`	Specifies if the persistence unit will participate in a JTA transaction managed by the container or it will managed by the `EntityTransaction` interface. It assumes two values: ► `JTA` ► `RESOURCE_LOCAL` Usually you do not have to specify this element, because it is automatically detected by the runtime environment. The default value is based on whether the application is using container-managed entity managers (`JTA`) or application-managed entity managers (`RESOURCE_LOCAL`). So, if you are moving from a unit-test development environment to a more production environment, these default values can come into play.
`jta-data-source`	(optional) Identifies the global JNDI name of the data source that can be used in JTA transactions.
`non-jta-data-source`	(optional) Non-JTA-enabled data source to be used by the persistence provider for accessing data outside a JTA. These non-jta data sources are the defined mechanisms for accessing the database for ID generation (table and sequence).
`class`	Identifies the list of Entities defined inside this persistence unit; if this attribute is not specified, the EJB container automatically detects all the entities marked with @Entity annotation.

Element	Description
mapping-file	(optional) Name of an XML mapping file containing persistence information to be included in this persistence unit.
properties	(optional) Defines the set of vendor-specific attributes passed to the persistence provider.

Note: The persistence.xml file is mandatory and cannot be substituted with the use of annotations.

5.1.8 Object-relational mapping through orm.xml

As we have seen in this chapter, the object-relational (o/r) mapping of an entity can be done through the use of annotations. As an alternative, you can specify the same information in an external file (called **orm.xml**) that must be packaged in the META-INF directory of the persistence module, or in a separate file packaged as a resource and defined in persistence.xml with the mapping-file element.

Example 5-4 shows an extract of an orm.xml file that maps the Account entity to the ITSO.ACCOUNT table.

Example 5-4 Extract of an orm.xml file to define an entity mapping

```
<entity-mappings ........>
   <entity class="itso.bank.entity.Account" metadata-complete="true"
            name="Account">
      <description>Account of ITSO Bank</description>
      <table name="ACCOUNT" schema="ITSO"></table>
      <attributes>
         <id name="accountNumber"><column name="id"/></id>
         <basic name="balance"></basic>
         <one-to-many name="transactionCollection"></one-to-many>
      </attributes>
   </entity>
   ......
</entity-mappings>
```

When using the orm.xml file with the metadata-complete="true" value, no annotations should be present in the entity classes and the complete mapping must be specified in the orm.xml file. For more information on the content of the orm.xml file, refer to the JPA specification.

5.1.9 Persistence provider

Persistence providers are implementations of the Java Persistence API (JPA) specification and can be deployed in the Java EE compliant application server that supports JPA persistence.

Apache OpenJPA persistence provider

WASCE includes the Apache OpenJPA persistence provider to support the open source implementation of JPA.

5.1.10 Entity manager

Entities cannot persist themselves on the relational database; annotations are used only to declare a POJO as an entity or to define its mapping and relationships with the corresponding tables on the relational database.

JPA has defined the `EntityManager` interface for this purpose to let applications manage and search for entities in the relational database.

The `EntityManager` primary definition is:

- An API that manages the life cycle of entity instances.
- Each `EntityManager` instance is associated with a *persistence context*.
- A persistence context defines the scope under which particular entity instances are created, persisted, and removed through the APIs made available by an `EntityManager`.
- An object that manages a set of entities defined by a persistence unit.

The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database. After a persistence context is closed, all managed entity object instances become detached from the persistence context and its associated entity manager, and are no longer managed.

> **Managed and unmanaged entities**: An entity object instance is either *managed* (attached) by an entity manager or *unmanaged* (detached):
>
> - When an entity is attached to an entity manager, the manager monitors any changes to the entity and synchronizes them with the database whenever the entity manager decides to flush its state.
> - When an entity is detached, and therefore is no more associated with a persistence context, it is unmanaged, and its state changes are not tracked by the entity manager and synchronized with the database.

The main operations that can be performed by an entity manager are shown in Table 5-7.

Table 5-7 PersistenceManager API

Operation (method)	Description
`persist`	► Insert a new entity instance into the database. ► Save the persistent state of the entity and any owned relationship references. ► Entity instance becomes managed.
`find`	Obtain a managed entity instance with a given persistent identity (primary key), return null if not found.
`remove`	Delete a managed entity with the given persistent identity from the database.
`merge`	► State of detached entity gets merged into a managed copy of the detached entity. ► Managed entity that is returned has a different Java identity than the detached entity.
`refresh`	Reload the entity state from the database.
`lock`	Set the lock mode for an entity object contained in the persistence context.
`flush`	Force synchronization with database.
`contains`	Determine if an entity is contained by the current persistence context.
`createQuery`	Create a query instance using dynamic Java Persistent Query Language.
`createNamedQuery`	Create instance of a predefined query.
`createNativeQuery`	Create instance of an SQL query.

Two types of entity manager are available:

- Container-managed entity manager
- Application-managed entity manager

Container-managed entity manager

The most common and widely used entity manager in a Java EE environment is the container-managed entity manager. In this mode, the container is responsible for the opening and closing of the entity manager and thus, the life cycle of the persistence context (this is transparent to the application).

A container-managed entity manager is also responsible for transaction boundaries. A container-managed entity manager is obtained in an application through dependency injection or through JNDI lookup, and the container manages interaction with the entity manager factory transparently to the application.

A container-managed entity manger requires the use of a JTA transaction, because its persistence context will automatically be propagated with the current JTA transaction, and the entity manager references that are mapped to the same persistence unit will provide access to this same persistence context within the JTA transaction.

This propagation of persistence context by the Java EE container avoids the need for the application to pass entity manager references from one component to another.

Container-managed persistence contexts can be defined to have one of two different scopes:

- Transaction persistence scope
- Extended persistence scope

Transaction persistence scope

Within this scope, contexts live as long as a transaction and are closed when the transaction completes. When the transaction-scoped persistence context is destroyed, all managed entity object instances become detached. Transaction scoped persistence is the most common within a container.

> **Note:** Entity manager instances are not thread safe. This means that you cannot inject it inside a servlet. You must instead use JNDI lookup and assign the result to a local variable (not an instance variable), otherwise you could have unpredictable results in your Java EE application.

Extended persistence context

Persistence contexts can be configured to live longer than a transaction. This is called an *extended persistence* context. Entity object instances that are attached to an extended context remain managed, and attached to the associated entity manager, even after a transaction is complete.

A container-managed persistence context that has an extended life time begins when a stateful session bean is created and ends when the stateful session bean is removed from the container.

Note: Only stateful session beans can have a container-managed, extended persistence context.

One of the important distinctions between a transaction-scoped persistence context and that of an extended persistence context is the state of the entities after a transaction completes:

- In the case of a transaction-scoped persistence context, the entities become detached, that is, they are no longer managed.
- In the case of an extended persistence context, the entities remain managed. In other words, all operations, such as persist, that are done outside of a transaction are queued and committed when the persistence context is attached to a transaction (and when it commits). This scenario is used for all the workflows/use cases that span multiple transactions.

Application-managed entity manager

An application-managed entity manager allows you to control the entity manager in application code. When using such an entity manager, you should be aware of the following fundamental aspects:

- With application-managed entity managers the persistence context is not propagated to application components, and the life cycle of entity manager instances is managed by the application.
- This means that the persistence context is not propagated with the JTA transaction across entity manager instances in a particular persistence unit.
- Furthermore, the entity manager, and its associated persistence context, is created and destroyed explicitly by the application.

Taking into account these peculiarities, this entity manager is usually used in two different scenarios:

- In Java SE environments, where you want to access a persistence context that is stand-alone, and not propagated along with the JTA transaction across the entity manager references for the given persistence unit.
- Inside a Java EE container, when you want to gain a very fine-grained control over the entity manager life cycle.

Application-managed entity manager in a Java SE scenario

If you want to use JPA outside a Java EE container, you must use resource-local transactions (because JTA is not available) through the APIs provided by the `EntityTransaction` interface.

5.1.11 Entity life cycle

An entity manager instance is associated with a persistence context. Within this persistence context, the entity instances and their life cycle are managed and can be accessed though the operations described in Table 5-7 on page 121.

Entity instances become unmanaged and detached when a transaction scope or extended persistence context ends. A very important consequence is that detached entities can be serialized and sent across the network to a remote client. The client can make changes remotely to these serialized object instances and send them back to the server to be merged back and synchronized with the database.

> **Note:** This behavior is very different from the EJB 2.1 entity model, where entities are always managed by the container. In EJB 3.0 you must always remember that you are working with entities that are POJOs.
>
> This can be considered as a real innovative (and, above all, simplified) model in designing Java EE applications, because you are not forced any more to use patterns, such as data transfer objects (DTO), between the business logic layer (session beans) and the persistence layer.
>
> Whether to propagate this new approach to clients of an EJB 3.0 business layer and (for instance a Web MVC controller or a Rich Eclipse Platform application) has been long debated, both inside the communities and in literature as well.
>
> Here (without going into too deep details), we can remark that before the coming of EJB 3.0, two fundamental approaches have been used to model Java EE applications:
>
> - EJB facade pattern, where session beans are used as facades and coordinate the access to the back-end enterprise information system.
> - POJO facade, where the facade pattern is directly implemented by POJOs and transactional and remote services are provided by frameworks such as Spring.
>
> These are two models that were sharply separated, but find their natural convergence in EJB 3.0, because session beans are POJOs as well.

5.1.12 JPA query language

The Java persistence query language (JPQL) is used to define searches against persistent entities independent of the mechanism used to store those entities. As such, JPQL is *portable*, and not constrained to any particular data store.

The Java persistence query language is an extension of the Enterprise JavaBeans query language, EJB QL, and is designed to combine the syntax and simple query semantics of SQL with the expressiveness of an object-oriented expression language:

- The application creates an instance of the `javax.persistence.EntityManager` interface.
- The `EntityManager` creates an instance of the `javax.persistence.Query` interface, through its public methods, for example, `createNamedQuery`.
- The `Query` instance executes a query (to read or update entities).

Query types

`Query` instances are created using the methods exposed by the `EntityManager` interface (Table 5-8).

Table 5-8 How to create a Query instance

Method name	Description
`createQuery(String qlString)`	Create an instance of Query for executing a Java Persistence query language statement.
`createNamedQuery (String name)`	Create an instance of Query for executing a named query (in the Java Persistence query language or in native SQL).
`createNativeQuery (String sqlString)`	Create an instance of Query for executing a native SQL statement, for example, for update or delete.
`createNativeQuery (String sqlString, Class resultClass)`	Create an instance of Query for executing a native SQL query that retrieves a single entity type.
`createNativeQuery (String sqlString, String resultSetMapping)`	Create an instance of Query for executing a native SQL query statement that retrieves a result set with multiple entity instances.

Query basics

A simple query that retrieves all the `Customer` entities from the database is shown here:

```
EntityManager em = ...
Query q = em.createQuery("SELECT c FROM Customer c");
List<Customer> results = (List<Customer>)q.getResultList();
```

A JPQL query has an internal name space declared in the from clause of the query. Arbitrary identifiers are assigned to entities so that they can be referenced elsewhere in the query. In the previous query example, the identifier c is assigned to the Customer entity.

The where condition is used to express a logical condition:

```
EntityManager em = ...
Query q = em.createQuery("SELECT c FROM Customer c
                          where c.ssn='111-11-1111'");
List<Customer> results = (List<Customer>)q.getResultList();
```

Operators

JPQL provides several operators; the most important are:

- Logical operators: NOT, AND, OR
- Relational operators: =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
- Arithmetic operators: +, -, /, *

Named queries

JPQL defines two types of queries:

- **Dynamic queries**: These queries are created on the fly.
- **Named queries**: These queries are intended to be used in contexts where the same query is invoked several times. Their main benefits include the improved reusability of the code, a minor maintenance effort, and finally better performance, because they are evaluated once.

;

> **Note:** From this point of view, there is a strong similarity between dynamic/named queries and JDBC Statement/PreparedStatements. However, named queries are stored in a global scope, which enables them to be accessed by different EJB 3.0 components.

Defining a named query

Named queries are defined using the @**NamedQuery** annotation:

```
@Entity
@Table (schema="ITSO", name="CUSTOMER")
@NamedQuery(name="getCustomerBySSN",
             query="select c from Customer c where c.ssn = ?1")
public class Customer implements Serializable {
     ...
}
```

The `name` attribute is used to uniquely identify the named query, while the `query` attribute defines the query. We can see how this syntax resembles the syntax used in JDBC code with `jdbc.sql.PreparedStatement` statements.

Instead of a positional parameter (?1), the same named query can be expressed using a named parameter:

```
@NamedQuery(name="getCustomerBySSN",
            query="select c from Customer c where c.ssn = :ssn")
```

Completing a named query

Named queries must have all their parameters specified before being executed. The `javax.persistence.Query` interface exposes two methods:

```
public void setParameter(int position, Object value)
public void setParameter(String paramName, Object value)
```

A complete example that uses a named query is shown here:

```
EntityManager em = ...
Query q = em.createNamedQuery ("getCustomerBySSN");
q.setParameter(1, "111-11-1111");
//q.setParameter("ssn", "111-11-1111"); // for named parameter
List<Customer> results = (List<Customer>)q.getResultList();
```

Defining multiple named queries

If there are more than one named query for an entity, they are placed inside an **@NamedQueries** annotation, which accepts an array of one or more @NamedQuery annotations:

```
@NamedQueries({
   @NamedQuery(name="getCustomers",
               query="select c from Customer c"),
   @NamedQuery(name="getCustomerBySSN",
               query="select c from Customer c where c.ssn =?1"),
   @NamedQuery(name="getAccountsBySSN",
               query="select a from Customer c, in(c.accountCollection) a
                     where c.ssn =?1 order by a.accountNumber")
})
```

Updating and deleting instances

Of course, JPQL can be used to update or delete entities.

Retrieving a single entity

If you want to retrieve a single instance, the `Query` interface provides the `getSingleResult` method. This method has some significant characteristics:

- It throws a `NoResultException` if there is no result.

- It throws a `NonUniqueResultException` if more than one result.
- It throws an `IllegalStateException` if called for a Java Persistence query language UPDATE or DELETE statement.

> **Note:** Even if these three exceptions are unchecked, they do not cause the provider to roll back the current transaction. These exceptions must be managed by the application code.

Relationship navigation

Relations between objects can be traversed using Java-like syntax:

```
SELECT t FROM Transaction t  WHERE t.account.id = '001-111001'
```

There are other ways to use queries to traverse relationships. For example, if the `Account` entity has a property called `transacationCollection` that is annotated as a @OneToMany relationship, then this query retrieves all the `Transaction` instances of one `Account`:

```
@NamedQuery(name="getTransactionsByID",
            query="select t from Account a, in(a.transactionsCollection) t
                    where a.id =?1 order by t.transtime")
```

Query paging

Query paging is a very common design pattern in applications that work with a corporate database, which generally contains a huge amount of data. The design of these applications must take into account not to generate queries that could read a very large number of records, because this could easily consume the necessary limited physical resources of the operating environment (RDBMS and Java EE application servers first).

The `Query` interface has two specific methods to limit the number of returned entities:

- Set the maximum number of results to retrieve:

  ```
  setMaxResults(int maxResults)
  ```

- Set the position of the first result to retrieve:

  ```
  setFirstPosition (int firstPosition)
  ```

Flush mode and queries

When queries retrieve data inside a transaction, the returned entities can be deeply impacted by the pending operations that are currently ongoing in the same transaction (within the same persistence context).

The `Query` interface supports a specific method `setFlushMode (int flushMode)` to fine control the flushing of the persistence context before a specific query is executed. The `flushMode` can assume two values:

- `FlushModeType.AUTO` (default value): The persistence provider is responsible for ensuring that all updates to the state of all entities in the persistence context, which could potentially affect the result of the query, are visible to the processing of the query. The persistence provider implementation can achieve this by flushing those entities to the database or by some other means.
- `FlushModeType.COMMIT`: The effect of updates made to entities in the persistence context upon queries is not detailed by the JPA specification, and depends on how the persistence provider implements the query integrity support.[1]

Polymorphic queries

JPQL queries are polymorphic, which means that the from clause of a query includes not only instances of the specific entity class to which it refers, but all subclasses of that class as well. The instances returned by a query include instances of the subclasses that satisfy the query conditions.

5.2 Jump start

For most chapters of this book, you can jump start to a chapter by importing the code of the previous chapters as a ZIP file. However, the jump start concept does not apply for this chapter, because it is the first chapter in the Donate application scenario. By definition, you must complete the preceding chapters before starting this chapter:

- Chapter 2, “Install and configure software” on page 25
- Chapter 3, “Configure the development environment” on page 49
- Chapter 4, “Prepare the legacy application” on page 77

5.3 Create the database connection pool

To create JPA entities from an existing database, we must have the database and tables defined. The `Vacation` database was created in Chapter 4, “Prepare the legacy application” on page 77, and a data source for the database was defined in 4.7, “Create the WASCE Vacation database data source” on page 90.

[1] OpenJPA and TopLink® update the database at commit regardless of the FlushModeType setting, whereas Hibernate saves changes before a query is executed.

5.4 Create an enterprise application project

In this section, we create the DonateEAR enterprise application project that we use in this scenario:

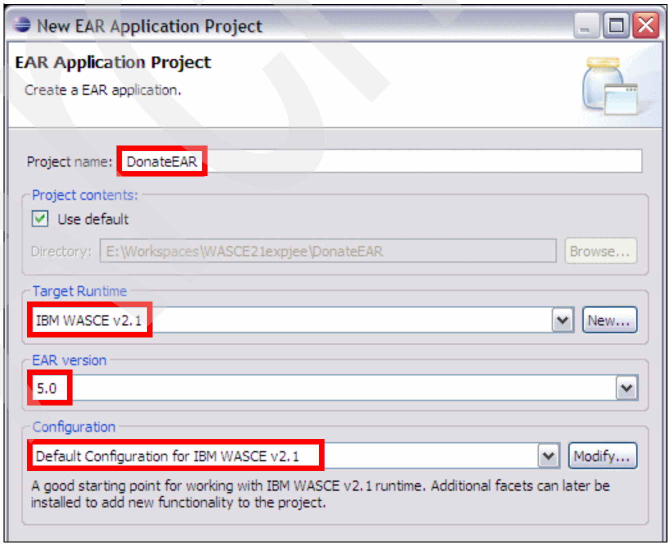
1. From the WASCE IDE, select **File** → **New** → **Project**.
2. In the New Project pop-up, select **Java EE** → **Enterprise Application Project**. Click **Next**.

Alternatives:

You can also create the Enterprise Application Project through several other methods.

- In the Project Explorer, right-click in the white space and select **New** → **Enterprise Application Project**.
- In the Java EE perspective action bar, click the **Create an Enterprise Application** icon ( ).

3. In the New EAR Application Project/EAR Application Project pop-up:
   - Set Project name to **DonateEAR**.
   - Set Target Runtime to **IBM WASCE v2.1**.
   - Set EAR version to **5.0**.
   - Set Configuration to **Default Configuration for IBM WASCE v2.1**.
   - Click **Next**.



New EAR Application Project

EAR Application Project

Create a EAR application.

Project name: DonateEAR

Project contents:

☑ Use default

Directory: E:\Workspaces\WASCE21expjee\DonateEAR

Target Runtime: IBM WASCE v2.1

EAR version: 5.0

Configuration: Default Configuration for IBM WASCE v2.1

A good starting point for working with IBM WASCE v2.1 runtime. Additional facets can later be installed to add new functionality to the project.

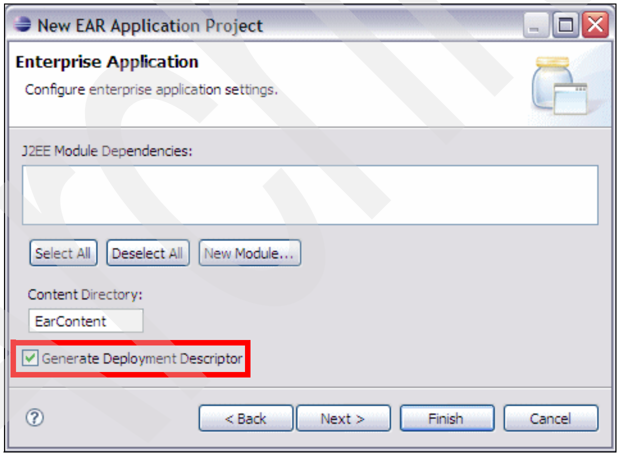
Behind the scenes:

- EAR version refers to the Java EE (J2EE) version. WASCE v2.1 supports both J2EE 1.4 (1.4) and Java EE 5 (5.0), and in this book you are using Java EE 5.
- The Configuration setting allows you to override the specific functions that capabilities that are valid for the combination of this project and the target runtime. A user could create a custom configuration that provides a reduced set of facets (functions) or an extended set.

  Project facets are a feature of WTP that allows the developer to include optional characteristics and requirements in a project. For example, in this case DonateEAR is defined as being an enterprise application (EAR) that runs on WASCE v2.1, supporting base Java EE capabilities (co-existence) and WASCE Extensions, such as the deployment plans.

  You can change the list of project facets after a project is created by opening the project properties (right-click the project, and select **Properties**) and adding or removing facets on the Project Facets tab.

4. In the New EAR Application Project/Enterprise Application pop-up, select **Generate Deployment Descriptor** and click **Next**.



New EAR Application Project

Enterprise Application

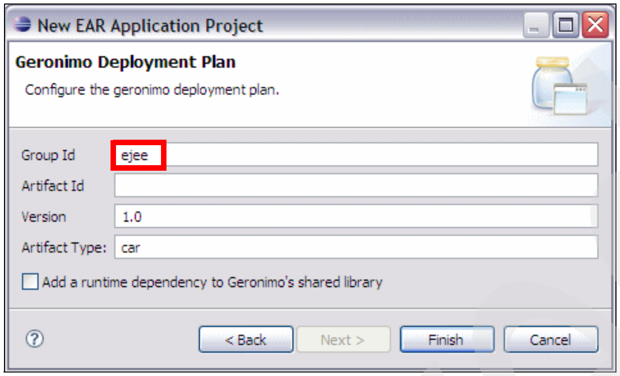
Configure enterprise application settings.

J2EE Module Dependencies:

Content Directory: EarContent

☑ Generate Deployment Descriptor

5. In the New EAR Application Project/Geronimo Deployment Plan pop-up, set the Group Id as **ejee**, optionally set the Artifact Id as **DonateEAR** (the default is the project name), accept the other two fields, and click **Finish**.



Geronimo Deployment Plan

Configure the geronimo deployment plan.

Group Id: ejee

Artifact Id:

Version: 1.0

Artifact Type: car

□ Add a runtime dependency to Geronimo's shared library

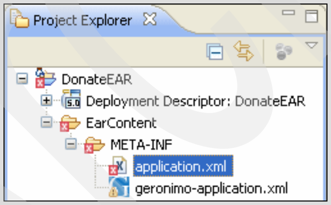
Note that we will use **ejee** as Group Id for all the projects that we add to this EAR.

6. Verify that the DonateEAR project is visible in the Project Explorer view.

Behind the scenes:

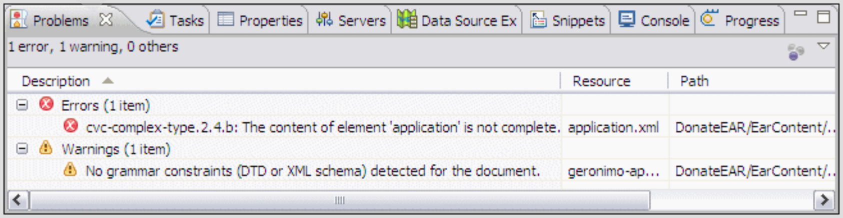
You receive an error in the `application.xml` file at this point because we have not added any projects to the EAR yet. This error will be resolved when you add the DonateEJB project to this EAR in 6.3.1, “Create the Project” on page 185.

- The error is visible in the Project Explorer through red error flags [☒] in the project tree hierarchy.



```
DonateEAR
  Deployment Descriptor: DonateEAR
  EarContent
    META-INF
      application.xml
      geronimo-application.xml
```

- The error is listed in the Problem view (tab) in the lower right pane.



1 error, 1 warning, 0 others

Description	Resource	Path
Errors (1 item)		
cvc-complex-type.2.4.b: The content of element 'application' is not complete.	application.xml	DonateEAR/EarContent/..
Warnings (1 item)		
No grammar constraints (DTD or XML schema) detected for the document.	geronimo-ap...	DonateEAR/EarContent/..

5.5 Create and configure a JPA project

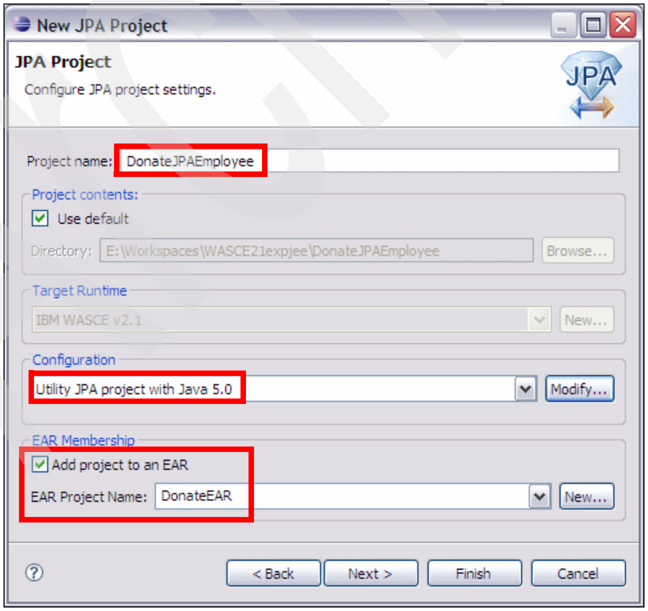
In this section, we create the DonateJPAEmployee project to be contained in the EAR project. A JPA project generally contains JPA entities, which are packaged as a JAR file. These JAR files are re-packaged in an EAR file for deployment.

In our case, the JPA project will contain the `Employee` entity that maps to the `VACATION.EMPLOYEE` table. We use reverse engineering to create the `Employee` entity from the database table.

5.5.1 Create the project

Follow these steps to create the project:

1. In the Project Explorer, right-click and select **New** → **Other**, and from the pop-up, select **JPA** → **JPA Project**.
2. In the New JPA Project/JPA Project pop-up:
   - Set the project name as **DonateJPAEmployee**.
   - Accept **Utility JPA project with Java 5.0** for Configuration. Optionally click **Modify** to see the project facets: Java, Java Persistence, and Utility Module.
   - Select **Add project to an EAR** and accept **DonateEAR** as EAR project.
   - Click **Next**.



New JPA Project

JPA Project

Configure JPA project settings.

Project name: DonateJPAEmployee

Project contents:

☑ Use default

Directory: E:\Workspaces\WASCE21expjee\DonateJPAEmployee

Target Runtime

IBM WASCE v2.1

Configuration

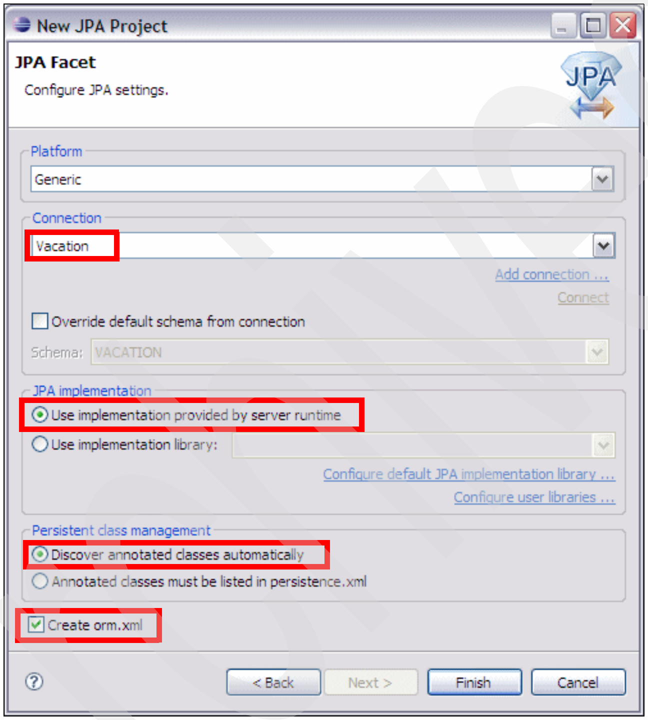
Utility JPA project with Java 5.0

EAR Membership

☑ Add project to an EAR

EAR Project Name: DonateEAR

3. In the New JPA Project/JPA Facet pop-up:
   - Select the **Vacation** connection that we defined in 4.3, “Create the Vacation database” on page 80 (Platform **Generic**, Connection **Vacation**).
   - Select **Use implementation provided by server runtime**.
   - Select **Discover annotated classes automatically**.
   - Select **Create orm.xml**.
   - Click **Finish**.



New JPA Project

JPA Facet

Configure JPA settings.

Platform: Generic

Connection: Vacation

□ Override default schema from connection

Schema: VACATION

JPA implementation

● Use implementation provided by server runtime

○ Use implementation library:

Persistent class management

● Discover annotated classes automatically

○ Annotated classes must be listed in persistence.xml

☑ Create orm.xml

JPA
Add connection ...
Connect
Configure default JPA implementation library ...
Configure user libraries ...
< Back
Next >
Finish
Cancel

4. When prompted, switch to the JPA perspective. The JPA perspective provides special views for JPA development, such as JPA Structure (top right) and JPA Details (bottom right).

5. Verify that the DonateJPAEmployee project is visible in the Project Explorer. Notice the `persistence.xml` and `orm.xml` files under `src/META-INF`:
   - The `persistence.xml` file defines the persistence unit, which by default is the project name.
   - The `orm.xml` file can be used to define explicit mappings of JPA entities to database tables.

Behind the scenes:

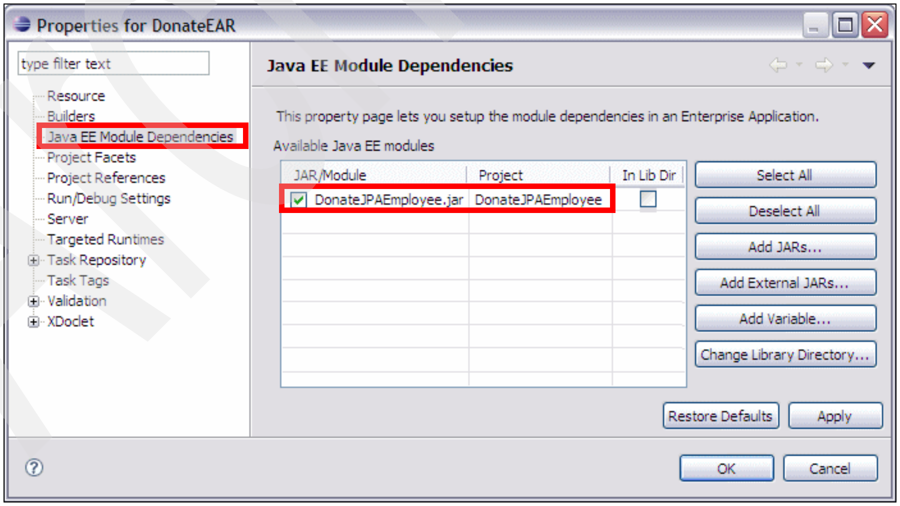
Views in Eclipse are typically used to navigate information related to the Workbench in some sort of hierarchical fashion. The Project Explorer view is used to navigate resources as they relate logically to a project. In addition to the source code directories, the Project Explorer will display any libraries added to the classpath of a project, such as the JRE System Library and IBM WASCE 2.1 as in our DonateJPAEmployee project. It can also hide various working directories and files (for example, `.settings` and `.classpath`). You can click the Menu icon in the view (top right ▽), and select **Customize View** to filter what is visible in the Project Explorer.

To view a strict representation of the physical project structure, you can enable the Navigator view by selecting **Window** → **Show View** → **Navigator**.

5.5.2 Project dependencies

The DonateEAR deployment descriptor, `application.xml`, still shows errors. JPA projects are not Java EE modules, and are not added to `application.xml`. However, a project dependency has been added:

1. Right-click DonateEAR and select **Properties**.
2. In the Properties pop-up, select **Java EE Module Dependencies**, and you can see that `DonateJPAEmployee.jar` has been added.



Java EE Module Dependencies

This property page lets you setup the module dependencies in an Enterprise Application.

Available Java EE modules

JAR/Module	Project	In Lib Dir
☑ DonateJPAEmployee.jar	DonateJPAEmployee	□

5.5.3 Add dependencies to the EAR deployment plan

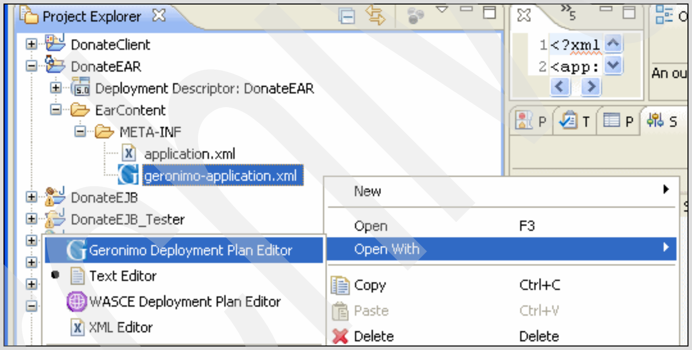
When we created the DonateEAR project, a WASCE Deployment plan, `META-INF/geronimo-application.xml`, was created as one of the project facets. We now edit this deployment plan to declare a dependency to the database connection pools that we configured in 5.3, “Create the database connection pool” on page 129:

1. In the Project Explorer, double-click **geronimo-application.xml** (located in `DonateEAR/EarContent/META-INF`) to open in an editor.
2. In the editor pane, select the **Deployment** tab (bottom).

Off course?

If you do not see a deployment tab, most likely the file was opened with an editor other than the default Geronimo Deployment Plan Editor.

To resolve this, close the editor, highlight **geronimo-application.xml** and right-click **Open With** → **Geronimo Deployment Plan Editor**.

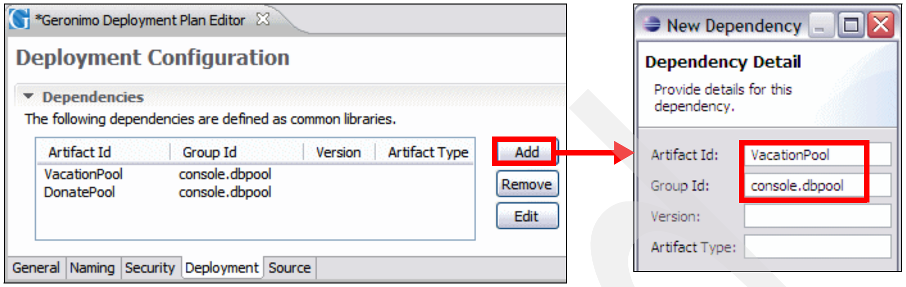


Eclipse remembers the last editor you used with a specific file, and will continue using that editor until you open with another editor.

3. Click **Add** next to the Dependencies column.
4. In the New Dependency/Dependency Details pop-up, set the Artifact Id to **VacationPool**, the Group Id to **console.dbpool**, and click **Finish**.

Behind the scenes: Most artifacts created through the WASCE Console are grouped together under the *console* directory of the WASCE repository. Thus, the group Id for all database pools created in the WASCE Console is `console.dbpool`. Similarly, the group Id for JMS will be `console.jms`.

5. Repeat this process to add a dependency to the **DonatePool** artifact with the same Group Id (**console.dbpool**).



The following dependencies are defined as common libraries.

Artifact Id	Group Id	Version	Artifact Type
VacationPool	console.dbpool		
DonatePool	console.dbpool		

Provide details for this dependency.

Artifact Id: VacationPool
Group Id: console.dbpool
Version:
Artifact Type:

6. Save the file (Ctrl+S), then switch to the **Source** tab.

Behind the scenes: Changes made in any single graphical editor tab (such as the Deployment tab) must be saved (**File** → **Save** or Ctrl+S) before switching to another graphical editor tab or to the source view. Otherwise, changes made within the editor might not be preserved and saved to the source file.

WORKAROUND:

The Geronimo Deployment Plan Editor creates incorrect XML that prevents the EAR file from deploying. This is a known defect that will be fixed in a future release.

Select the **Source** tab.

- Delete these lines under both dependency definitions.

```
<dep:version></dep:version>
<dep:type></dep:type>
```

- Switch the values for the `dep:groupid` and `dep:artificatid` attributes:
  - Switch the values `console.dbpool` and `DonatePool`.
  - Switch the values `console.dbpool` and `VacationPool`.

The corrected `geronimo-application.xml` content is shown here:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app:application ......>
    <dep:environment>
        <dep:moduleId>
            <dep:groupId>ejee</dep:groupId>
            <dep:artifactId>DonateEAR</dep:artifactId>
            <dep:version>1.0</dep:version>
            <dep:type>car</dep:type>
        </dep:moduleId>
        <dep:dependencies>
            <dep:dependency>
                <dep:groupId>console.dbpool</dep:groupId>
                <dep:artifactId>VacationPool</dep:artifactId>
            </dep:dependency>
            <dep:dependency>
                <dep:groupId>console.dbpool</dep:groupId>
                <dep:artifactId>DonatePool</dep:artifactId>
            </dep:dependency>
        </dep:dependencies>
    </dep:environment>
</app:application>
```

7. Save and close the Geronimo Deployment Plan Editor (`geronimo-application.xml`) using one of the following methods:
   - Click ☒ next to the name, and click **Yes** in the Save Resource pop-up.
   - Select **File** → **Save** (or Ctrl+S) and then **File** → **Close** (or Ctrl+F4).

5.6 Generate the Employee entity from a table

In this section we first create the Employee entity and then verify the persistence.xml file.

5.6.1 Create the Employee entity

Here we create the `Employee` entity from the existing `EMPLOYEE` table in the `Vacation` database, using the *bottom-up scenario*:

1. Start the **ExperienceJavaEE Server** if it is not running.

2. Switch to the JPA perspective (it should be open already):

   – Click the **Select Perspective** icon [ ] located at the upper right, and select **Other** from the pull-down.

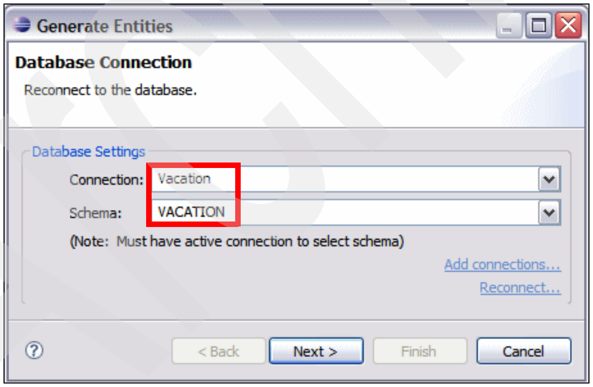
   – In the Open Perspective pop-up, select **JPA** click **OK**.

3. In the Data Source Explorer view, ensure that the **Vacation** database has an active connection.

> **Off course?**
>
> ► If the `Vacation` database is not in the list of connections, repeat the steps in 4.3, “Create the Vacation database” on page 80.
>
> ► If the `Vacation` connection cannot be expanded, then right-click **Vacation** and select **Connect**.
>
> These issues can occur if you closed the view or restarted the workspace after following the steps in Chapter 4, “Prepare the legacy application” on page 77.

4. In the Project Explorer, right-click the **DonateJPAEmployee** project and select **JPA Tools** → **Generate Entities**.

5. In the Generate Entities/Database Connection pop-up, make sure that the **Vacation** connection and **VACATION** schema are selected, and click **Next**.

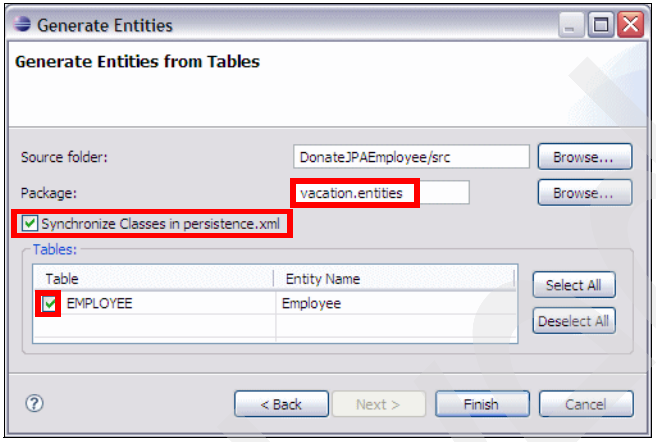


Connection: Vacation

Schema: VACATION

(Note: Must have active connection to select schema)

6. In the **Generate Entities/Generate Entities from Tables** pop-up, set the Package to `vacation.entities`, select **Synchronize Classes in persistence.xml**, and select the **EMPLOYEE** table. Click **Finish**.



Generate Entities

Generate Entities from Tables

Source folder: DonateJPAEmployee/src

Package: vacation.entities

☑ Synchronize Classes in persistence.xml

Tables:

Table	Entity Name
☑ EMPLOYEE	Employee

7. In the Project Explorer, open **Employee.java** (in `DonateJPAEmployee/ejbModule/vacation.entities`) to inspect the contents.

   – Notice that the JPA tooling has generated the Java class fields and accessor methods automatically. These fields correspond to the columns of the `EMPLOYEE` table in the database. Example 5-5 shows an extract of the generated `Employee` class.

8. Close `Employee.java`.

Example 5-5 Employee entity (extract)

```
@Entity
public class Employee implements Serializable {
   @Id
   @Column(name="EMPLOYEE_ID")
   private int employeeId;

   @Column(name="LAST_NAME")
   private String lastName;

   private BigDecimal salary;

   ......

   public int getEmployeeId() {
      return this.employeeId;
   }
```

```
    public void setEmployeeId(int employeeId) {
        this.employeeId = employeeId;
    }
    ......
}
```

Behind the scenes:

In addition to fields and accessor methods, the JPA tooling also adds JPA annotations, such as @Entity, @Id, and @Column automatically.

Because SQL database field names are case-insensitive, the JPA tooling assumes that words are separated by underscores. Thus, column names (`FIRST_NAME`) are mapped to Java fields (`firstName`) using the lowerCamelCase convention of Java. For these fields, a @Column annotation is added to map the field to the database column. The `salary` field does not require a @Column annotation because the column name is the same.

The @Id annotation defines the primary key of the entity.

5.6.2 Verify the persistence.xml file

Because we selected **Synchronize Classes in persistence.xml**, the Employee entity is added to the `persistence.xml` file:

1. In the Project Explorer, open **persistence.xml** (in `DonateJPAEmployee /ejbModule/META-INF`).

   Note that a persistence unit named `DonateJPAEmployee` has been created, with the `Employee` entity contained in it.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <persistence ......>
       <persistence-unit name="DonateJPAEmployee">
           <class>
           vacation.entities.Employee</class>
       </persistence-unit>
   </persistence>
   ```

2. Close `persistence.xml`.

5.7 Create the Fund entity

In 5.6, “Generate the Employee entity from a table” on page 138, we used an existing database table to generate a JPA entity. In this section, we create a JPA entity from scratch. This class will be used to generate a database table in the Donate database using the *top-down scenario*.

5.7.1 Create a JPA project for the Fund entity

Data for donation funds will be stored in the Donate database. We create a JPA entity named Fund for this purpose.

We already have one JPA project for entities of the Vacation database, and we could create the Fund entity in the same DonateJPAEmployee project. This would result in two persistence units in the same project (in the same persistence.xml file).

The JPA runtime architecture does support having multiple persistence units in the same persistence.xml. However, one side effect is that the persistence manager will create tables for all the persistence unit in each and every data source. In our case, this would result in an extra VACATION.FUND table in the Vacation database, and an extra DONATE.EMPLOYEE table in the Donate database. These tables are created with the wrong schema and are in fact never used or accessed by the application.

In addition, the JPA tooling does not formally support multiple persistence units in the same persistence.xml:

- The tooling displays the following warning whenever validating a persistence.xml with multiple persistence units:

  ```
  Multiple persistence units defined - tooling only supports 1 persistence
  unit per project
  ```

- JPA validation only works against one database connection and thus in theory would generate errors, because only one table would be found. In reality, due to the duplicate table problem described previously, validation would work because both tables exist in both databases.

Therefore, to avoid these issues, we create a second JPA project named DonateJPAFund.

1. Follow the same instructions as in 5.5, “Create and configure a JPA project” on page 133 to create this project:
   – Select **New** → **Project** and in the resulting pop-up select **JPA** → **JPA Project**.

   - For the project name type **DonateJPAFund**.
   - Select **Add project to an EAR** and select **DonateEAR**.
   - Click **Next**.
   - Select **Donate** for the connection.
   - Click **Finish**.

5.7.2 Create the Fund entity

Follow these steps to create the Fund entity:

1. In the **Project Explorer**, right-click **DonateJPAFund** and select **New** → **Class**.
2. In the New Java Class/Java Class pop-up:
   - Set the Package to **donate.entities** and the Name to **Fund**.
   - For Interfaces, click **Add**. In the Implemented Interfaces Selection pop-up, search and select **java.io.Serializable** and click **OK**.
3. Click **Finish** to create and open the `Fund` class in the Java Editor.
4. In the class body, declare two Java fields:

   ```
   protected String name;
   protected int balance;
   ```

5. While in the editor, select **Source** → **Generate Getters and Setters**. In the Generate Getters and Setters pop-up, click **Select All** and click **OK**.
6. Leave the editor open because we will make further changes in the next section.

Alternatives:

The JPA specification does not explicitly require accessor methods for its entities, but they are generally recommended. Not only do they provide encapsulation of any business logic such as data validation, they also pave the way for loose coupling.

However, there are a few cases where you might not want to use getters and/or setters. For example, if you wanted to ensure that an entity was immutable (cannot be changed after it is created), you can elect to omit setter methods. If you omit accessor methods, JPA will try to access the fields directly. Thus, you must declare the fields public so they are visible to JPA.

5.7.3 Add JPA annotations to the Fund entity

The JPA tooling provides the JPA Details view (bottom right) that allows you to add annotated metadata to the classes, fields and definitions associated with JPA entities. We first use this tooling to declare the `Fund` class as a JPA entity, then specify the `name` field as the identity field:

1. In the editor of the `Fund` class, add the @Entity annotation:

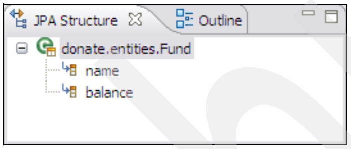
   ```
   @Entity
   public class Fund implements Serializable {
   ```

2. Select **Source** → **Organize Imports** (or press Ctrl+Shift+O) to resolve the import.

3. Select **Fund** in the class definition:

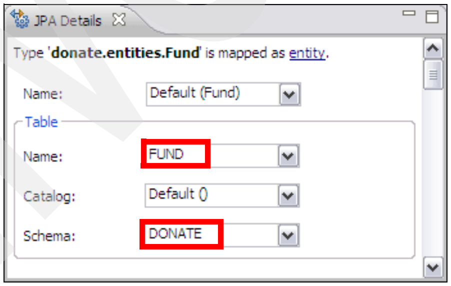
   ```
   public class Fund implements Serializable {
   ```

4. Notice the JPA Structure view (top right) and the JPA Details view (bottom right). Overtype the JPA Details for the Table with **FUND** as table name, and **DONATE** as table schema.



JPA Structure ⊠ | Outline

⊟ donate.entities.Fund
- name
- balance



JPA Details ⊠

Type 'donate.entities.Fund' is mapped as entity.

Name: Default (Fund)

Table

Name: FUND

Catalog: Default ()

Schema: DONATE

5. Notice that a @Table annotation is added to the source:

   ```
   @Table(schema="DONATE", name = "FUND")
   ```

> **Off course?**
>
> If you were to save the file at this point, Eclipse would report an error because we have declared a JPA entity with no primary key. In the next series of steps, we declare the `name` field as our primary key and this error message would go away.

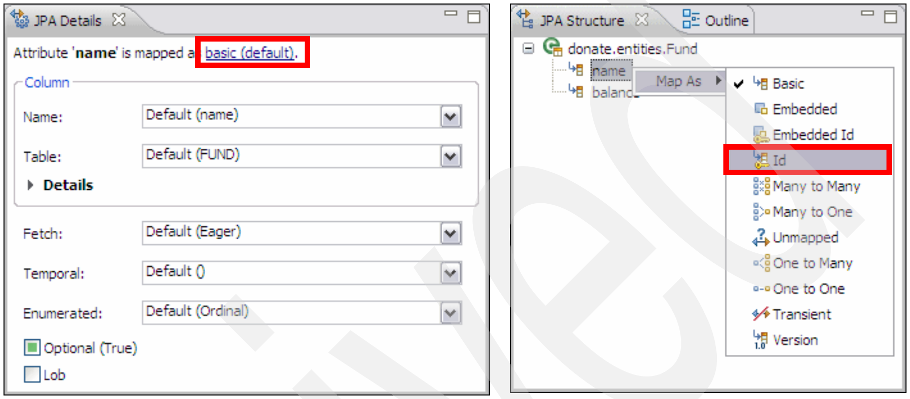
6. The name field is the key in the database table. In the editor, select **name** in the field declaration:

   ```
   protected String name;
   ```

7. We can map the fund `name` as the key field (@Id annotation) in two ways:
   - In the JPA Details view, you can see the text:

     ```
     Attribute 'name' is mapped as basic (default).
     ```

   - Click **basic (default)**, then select **Id** and click **OK**.
   - In the JPA Structure view, right-click **name** and select **Map As** → **Id**.



JPA Details

Attribute '**name**' is mapped as basic (default).

Column

Name: Default (name)

Table: Default (FUND)

Details

Fetch: Default (Eager)

Temporal: Default ()

Enumerated: Default (Ordinal)

☑ Optional (True)

□ Lob

8. An @Id annotation is added to the `name` field.
9. Save `Fund.java` but leave the editor open because we will make further changes in the next section.
10. Ignore any potential errors that the table and columns cannot be resolved. We have not yet defined the `FUND` table in the `Donate` database.

Alternatives:

To keep things simple, we are using the fund name as the primary key. Most database administrators would likely frown on this in the real world, though, because the fund name could change. JPA supports a number of automatic identity generators. For more information on how this works, refer to “ Entity automatic identity generation” on page 107.

5.7.4 Add a named query to the Fund entity

In our application we want to retrieve all the funds that exist. We can do this best by using the JPA query language (JPA QL), where we can execute dynamic queries or predefined queries. Here we use a predefined named query:

1. In the action bar, select **Window** → **Show View** → **Other**.

- In the Show View pop-up, select **General** → **Snippets** (or start typing snippet to find the view).
- Click **OK** and the Snippets view is visible in the lower right pane.

2. In the code editor, insert a blank line above the @Entity annotation and position the cursor on this line.
3. From the Snippets view insert the **F04 Fund NamedQuery** snippet. This should add the code for the named query to the Fund class:

```
@NamedQuery(name="getAllFunds", query="Select f FROM Fund f")
@Entity
public class Fund implements Serializable {
```

4. To resolve the NamedQuery, select **Source** → **Organize Imports** (or press **Ctrl+Shift+O**).
5. Optionally format the code by selecting **Source** → **Format** (or press **Ctrl+Shift+F**).
6. If you scroll down in the JPA Details view, you can see the getAllFunds query listed under Queries.
7. Save and close Fund.java.

5.8 Customize the persistence.xml files

As mentioned in 5.1.7, “Persistence units” on page 117, the persistence.xml file provides the JPA implementation with details about how to connect to the underlying data sources. These connection details are grouped into *persistence units*.

We are using two distinct data sources in our application, VacationPool and DonatePool. Thus, we require two distinct persistence units, one in each JPA project. The JPA tooling created a persistence unit automatically in each project, and it contains a reference to the Employee class in the DonateJPAEmployee project.

In this section, we add the Fund class to the persistence.xml file in the DonateJPAFund project, and then configure the link to the data source.

5.8.1 Add the Fund entity to the persistence.xml

To add the Fund entity to the persistence.xml file, perform these steps:

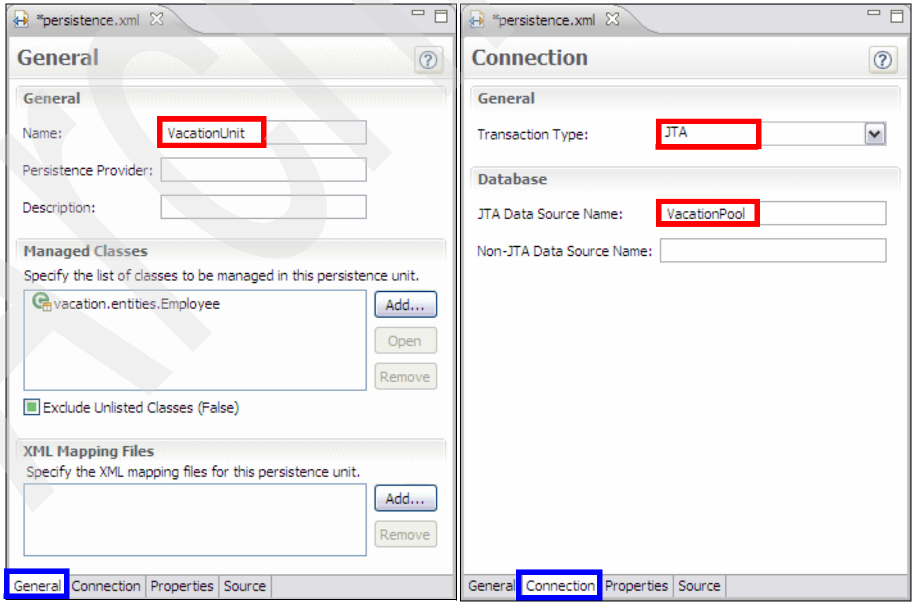
1. In the Project Explorer, open **persistence.xml** (in DonateJPAFund /ejbModule/META-INF). Note that it currently has no classes defined.

2. Right-click on **persistence.xml** and select **JPA Tools** → **Synchronize Classes**. This adds a reference to the `Fund` class into the persistence unit.
3. Review the reference in the **General** and in the **Source** tabs.
4. Save and close `persistence.xml` when done.

5.8.2 Add data source references to the persistence units

Working with JPA Entities inside an application server allows you to leverage several features of Java EE. First, you have access to container-managed database pools, such as our `VacationPool` and `DonatePool`. You are also able to use the container to automatically manage transactions against a JTA data source. Because we execute transactions that require transaction management, we configure a JTA data source for each persistence unit. When we define unit tests later in this chapter, we work with JPA entities outside of our application server and do not require any data source metadata.

1. In the Project Explorer, open **persistence.xml** (in `DonateJPAEmployee/src/META-INF`).
2. In the **General** tab, change the name of the persistence unit to **VacationUnit** (we connect to the `Vacation` database).
3. In the **Connection** tab, select **JTA** for Transaction Type, and type **VacationPool** as JTA Data Source Name.


*persistence.xml

General

General

Name: VacationUnit

Persistence Provider:

Description:

Managed Classes

Specify the list of classes to be managed in this persistence unit.

vacation.entities.Employee

□ Exclude Unlisted Classes (False)

XML Mapping Files

Specify the XML mapping files for this persistence unit.

*persistence.xml

Connection

General

Transaction Type: JTA

Database

JTA Data Source Name: VacationPool

Non-JTA Data Source Name:

4. Verify that the Source shows:

```
<persistence-unit name="VacationUnit" transaction-type="JTA">
   <jta-data-source>VacationPool</jta-data-source>
   <class>
   vacation.entities.Employee</class>
</persistence-unit>
```

5. Save and close `persistence.xml`.
6. In the Project Explorer, open **persistence.xml** (in `DonateJPAFund/src/META-INF`).
7. In the **General** tab, change the name of the persistence unit to **DonateUnit** (we connect to the `Donate` database).
8. In the **Connection** tab, select **JTA** for Transaction Type, and type **DonatePool** as JTA Data Source Name.
9. Verify the source code, and save and close `persistence.xml`.

```
<persistence-unit name="DonateUnit" transaction-type="JTA">
   <jta-data-source>DonatePool</jta-data-source>
   <class>
   donate.entities.Fund</class>
</persistence-unit>
```

Behind the scenes:

Our application accesses the database only through JTA transactions. We could also define a `<non-jta-data-source>` for access by non-transactional programs.

5.9 Test the JPA entities with JUnit

In this section, we will use JUnit to unit test the entities.

The Ganymede Eclipse IDE includes the Eclipse Test & Performance Tools Platform (TPTP), which extends the Eclipse Hyades Tool Project. It contains monitoring, tracing, profiling, and testing tools. JUnit is one of the testing tools and can be used for automated component testing.

5.9.1 Create the JUnit project

To test the entities, we create a JUnit project, which is basically a Java project:

1. Switch to the Java perspective:
   - Click the **Select Perspective** icon [ ] located at the upper right, and select **Other** from the pull-down.
   - In the Open Perspective pop-up, select **Java** and click **OK**.
2. Select **File** → **New** → **Java Project**.
3. In the New Project/Create a Java project pop-up, set the project name to **DonateEJB_Tester**, and click **Next**.
4. In the New Java Project/Java Settings pop-up:
   - Select the **Projects** tab and click **Add**. In the Required Project Selection pop-up, select **DonateJPAEmployee** and **DonateJPAFund**, and click **OK**.
   - Back in the New Java Project/Java Settings pop-up, select the **Libraries** tab and click **Add Library**:
     - In the Add Library pop-up, select **User Library** and click **Next**.
     - In the Add Library/User Library pop-up, select both **Derby Net Client** and **OpenJPA**, and click **Finish**.
   - Back in the New Java Project/Project Settings pop-up, click **Add Library** again:
     - In the Add Library pop-up, select **Server Runtime** and click **Next**.
     - In the Add Library/Server Library pop-up, select **IBM WASCE v2.1** and click **Finish**.
   - Back in the New Java Project/Java Settings pop-up, click **Finish**.

Behind the scenes:

Remember that the user libraries were defined in 3.8, “Configure user libraries in Eclipse” on page 71.

The libraries are added to the DonateEJB_Tester project. Expand the project in the Project Explorer to see the libraries. Expand a library to see all the JAR files contained in the library.

The JUnit test cases in this project will run in the Java SE environment, outside of the WASCE server. Therefore, this project has a dependency to the JPA projects to have access to the JPA entities.

5.9.2 Create the persistence units for the Java SE profile

The Java Persistence API came out of the framework of the JSR 220 Expert Group (EJB 3.0). However, the Java Persistence API can be used outside the context of EJBs, including in a Web module running inside the application server or even in a standalone Java application, such as JUnit, running outside the application server.

Because the JUnit tests are running outside of the WASCE server, the `persistence.xml` file must define the JPA implementation to be used in the Java SE environment. We could modify the existing `persistence.xml` files, but it is much better to provide a `persistence.xml` file in the JUnit project itself. This `persistence.xml` file takes precedence over the files in the related JPA projects.

We follow these steps:

1. In the Project Explorer, expand the **DonateEJB_Tester** project.
2. Right-click **src** and select **New** → **Folder**. Type **META-INF** as name and click **Finish**.
3. Right-click **META-INF** (in `DonateEJB_Tester/src`) and select **New** → **File**. Type **persistence.xml** as name and click **Finish**.
4. Add the **Snippets** view to the Java perspective (**Window** → **Show View** → **Other** and then select **General** → **Snippets**).
5. Select anywhere in the white space of the `persistence.xml` editor.
6. In the Snippets view (lower right), expand the **Experience Java EE** category, and double-click on **F06 DonateEJB_Tester Persistence** to populate the new `persistence.xml` file with the required configuration settings.
7. Save and close `persistence.xml`.

Behind the scenes:

This persistence.xml file has two persistence units, one for each database. Instead of data source connections, properties are used to define OpenJPA connections to the two databases.

For example, to connect to the Vacation database:

```
<persistence-unit name="VacationUnit">
   <class>vacation.entities.Employee</class>
   <properties>
      <property name="openjpa.ConnectionURL"
            value="jdbc:derby://localhost:1527/Vacation" />
      <property name="openjpa.ConnectionDriverName"
            value="org.apache.derby.jdbc.ClientDriver" />
      <property name="openjpa.ConnectionUserName" value="VACATION" />
      <property name="openjpa.Log" value="SQL=TRACE" />
   </properties>
</persistence-unit>
```

Note the openjpa.Log property:

```
<property name="openjpa.Log" value="SQL=TRACE" />
```

With the SQL=TRACE value, all the SQL statements that are executed are displayed in the Console. This enables you to monitor database access.

Note that for the DonateUnit, a new database named Donate is created:

```
      <property name="openjpa.ConnectionURL"
            value="jdbc:derby://localhost:1527
                        /Donate;create=true" />
```

For the DonateUnit we use *runtime forward mapping*. We use one of these properties to tell OpenJPA to automatically generate any database tables that do not already exist. By enabling this feature, we are able to automatically generate the FUND table in the Donate database when its JPA entity is first accessed.

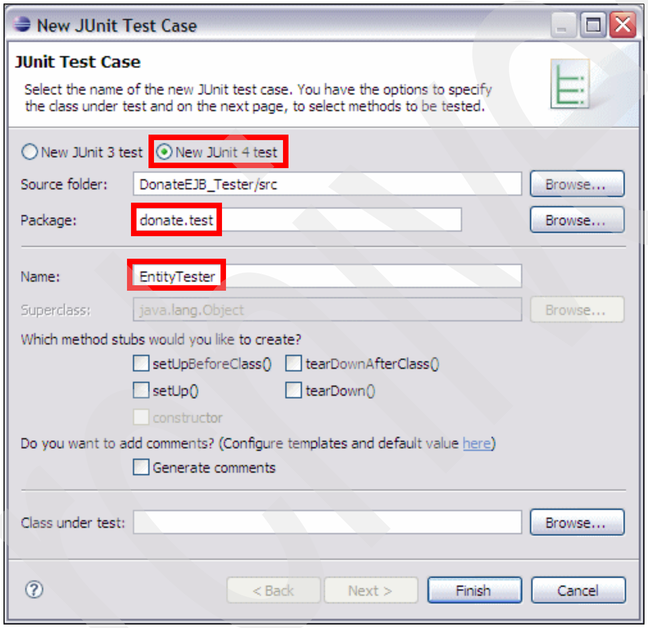
```
      <property name="openjpa.jdbc.SynchronizeMappings"
            value="buildSchema(ForeignKeys=true)" />
```

Because this persistence.xml file is in the DonateEJB_Tester project, the runtime uses it instead of the files in the JPA projects.

5.9.3 Create the JUnit test case

Next we create the test case to test the entities:

1. In the Project Explorer, right-click **src** (in DonateEJB_Tester) and select **New** → **Other** and then from the pop-up select **Java** → **JUnit** → **JUnit Test Case**.
2. In the New JUnit Test Case/JUnit Test Case pop-up:
   – Select **New JUnit 4 test**.
   – Set the Package to **donate.test**.
   – Set the Name to **EntityTester**.
   – Click **Finish**.



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

○ New JUnit 3 test ● New JUnit 4 test

Source folder: DonateEJB_Tester/src

Package: donate.test

Name: EntityTester

Superclass: java.lang.Object

Which method stubs would you like to create?

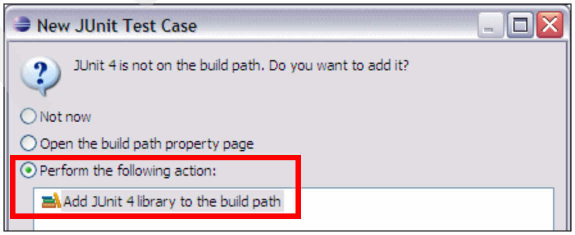
□ setUpBeforeClass() □ tearDownAfterClass()
□ setUp() □ tearDown()
□ constructor

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Class under test:

3. At the resulting New JUnit Test Case pop-up, click **OK** to have the wizard add the JUnit 4 library to the build path.



New JUnit Test Case

JUnit 4 is not on the build path. Do you want to add it?

○ Not now
○ Open the build path property page
● Perform the following action:

Add JUnit 4 library to the build path

4. The `EntityTest` class opens in the editor:
   - Delete the existing lines.
   - In the Snippets view (lower right), expand the **Experience Java EE** category, and double-click on **F07 EntityTester** to insert the complete JUnit test case.
5. Save and close `EntityTester.java`.

Behind the scenes:

- The @BeforeClass annotation for the `init` method is used to initialize the entity managers for the two persistence units:

```
public static void init() {
   emVacationUnit = Persistence
      .createEntityManagerFactory("VacationUnit")
      .createEntityManager();
   ....
}
```

- The @Test annotated `getEmployee` method retrieves an employee using the `find` method of `EntityManager`:

```
public void getEmployee() {
   Employee e = emVacationUnit.find(Employee.class, 2);
   ......
}
```

- The @Test annotated `createFund` method creates a fund using the `persist` method of `EntityManager`:

```
public void createFund() {
   int random = Math.abs(new Random().nextInt());
   Fund f = new Fund();
   f.setName("Test Fund #" + random);
   f.setBalance(20);
   emDonateUnit.getTransaction().begin();
   emDonateUnit.persist(f);
   emDonateUnit.getTransaction().commit();
}
```

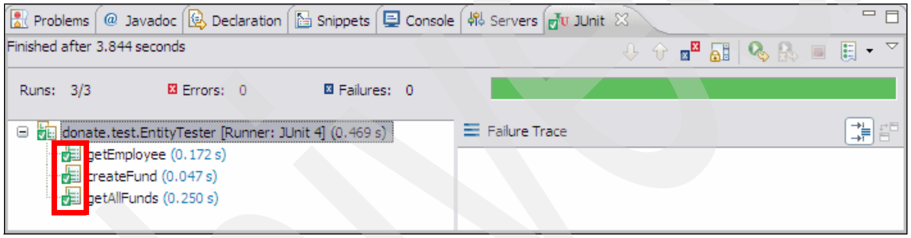
- The @Test annotated `getAllFunds` method retrieves all the funds using the `getAllFunds` named query:

```
public void getAllFunds() {
   List<Fund> funds = emDonateUnit.createNamedQuery("getAllFunds")
                                           .getResultList();
   assertTrue("No funds found", funds.size() > 0);
   ......
}
```

5.9.4 Run the JUnit test

We are now ready to run the JUnit test:

1. In the Servers view of the Java EE perspective, ensure that the ExperienceJavaEE Server is started. The server is required to connect to the Derby databases.
2. In the Project Explorer, right-click **EntityTester.java** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.
3. The JUnit view opens. If the view opens in an inconvenient place, just drag the view to the bottom right.
4. Verify that all tests were successful, that is, the tests are marked by a green check mark.



Finished after 3.844 seconds

Runs: 3/3 Errors: 0 Failures: 0

donate.test.EntityTester [Runner: JUnit 4] (0.469 s)
getEmployee (0.172 s)
createFund (0.047 s)
getAllFunds (0.250 s)

5. The Console view displays the detailed results of the test (Example 5-6).

Example 5-6 JUnit test output with SQL statement trace

```
Creating EntityManagers
==============================
31  VacationUnit  INFO   [main] openjpa.Runtime - Starting OpenJPA 1.0.2
156  VacationUnit  INFO   [main] openjpa.jdbc.JDBC - Using dictionary ...
1593  VacationUnit  INFO   [main] openjpa.Enhance - Creating subclass ...
VacationUnit EntityManager successfully created

0  DonateUnit  INFO   [main] openjpa.Runtime - ...
1359  DonateUnit  TRACE  [main] openjpa.jdbc.SQL - <t 1090273532, conn
1194084140> executing stmnt 1366970746 CREATE TABLE Fund (name VARCHAR(255)
NOT NULL, balance INTEGER, PRIMARY KEY (name))
1406  DonateUnit  TRACE  [main] openjpa.jdbc.SQL - <...> [47 ms] spent
1422  DonateUnit  INFO   [main] openjpa.Enhance - Creating subclass ...
DonateUnit EntityManager successfully created

Retrieving info for employee 2
==============================
3953  VacationUnit  TRACE  [main] openjpa.jdbc.SQL - <...> executing ...
SELECT t0.FIRST_NAME, t0.LAST_NAME, t0.MIDDLE_NAME, t0.salary,
t0.VACATION_HOURS FROM Employee t0 WHERE t0.EMPLOYEE_ID = ? [params=(int)2]
```

```
Employee info successfully retrieved!
Employee: org.apache.openjpa.enhance.vacation$entities$Employee$...
Employee Id: 2
First Name: Neil
Last Name:
Last Name: Armstrong
Salary: 100000
Vacation Hours: 50

Persisting new Fund
==============================
1766  DonateUnit  TRACE  [main] openjpa.jdbc.SQL - <...> executing ...
INSERT INTO Fund (name, balance) VALUES (?, ?) [params=(String) Test Fund
#xxxxxxxxxx, (int) 20]
Fund successfully persisted!

Retrieving Funds from database
==============================
1969  DonateUnit  TRACE  [main] openjpa.jdbc.SQL - <...> executing ...
SELECT t0.name, t0.balance FROM Fund t0

Name                     Balance
Test Fund #xxxxxxxx      20
```

Behind the scenes:

Here are some notes on the output:

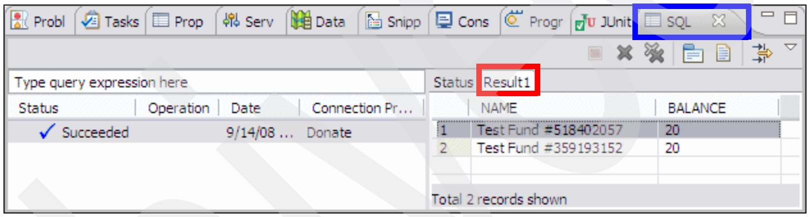
- We enabled trace in the `persistence.xml` file:

  `<property name="openjpa.Log" value="SQL=TRACE" />`

- The trace output shows the SQL statements that are executed and the elapsed time of execution.
- The `FUND` table is created in the `Donate` database. We will verify that the table was created in the next step.
- The employee is retrieved and displayed.
- A new fund is created with 20 hours.
- All the funds (only one the first time) are displayed.
- Rerun the test and two funds are displayed.

6. Verify the database activities using the WASCE IDE Data Source Explorer. Note that you can also do these same steps from the WASCE Console under **Embedded DB** → **DB Manager**.

- Switch to the Data Source Explorer view in the lower right in the Java EE perspective.
- Refresh the `Donate` connection by selecting **Databases** → **Donate (Apache Derby xxx)** → **Donate** and right-click **Refresh**.
- Verify that the `FUND` table was created and that a record was created.
  - Expand to **Databases** → **Donate (Apache Derby xxx)** → **Donate** → **Schemas** → **DONATE** → **Tables** → **FUND.** This will verify that the `FUND` database and table was created.
  - Select the `FUND` table and right-click **Data** → **Sample Contents**.
  - Switch to the SQL Results view (you might have to resize it), and then select the **Result1** tab on the left. An entry is displayed for every time the test was run, thus verifying the JUnit test.



Type query expression here

Status	Operation	Date	Connection Pr...
✓ Succeeded		9/14/08 ...	Donate

Status Result1

	NAME	BALANCE
1	Test Fund #518402057	20
2	Test Fund #359193152	20

Total 2 records shown

> **Off course?**
>
> If the JUnit test fails, verify that the server is started. Verify that the `Vacation` database exists.

5.10 Handle errors and warning

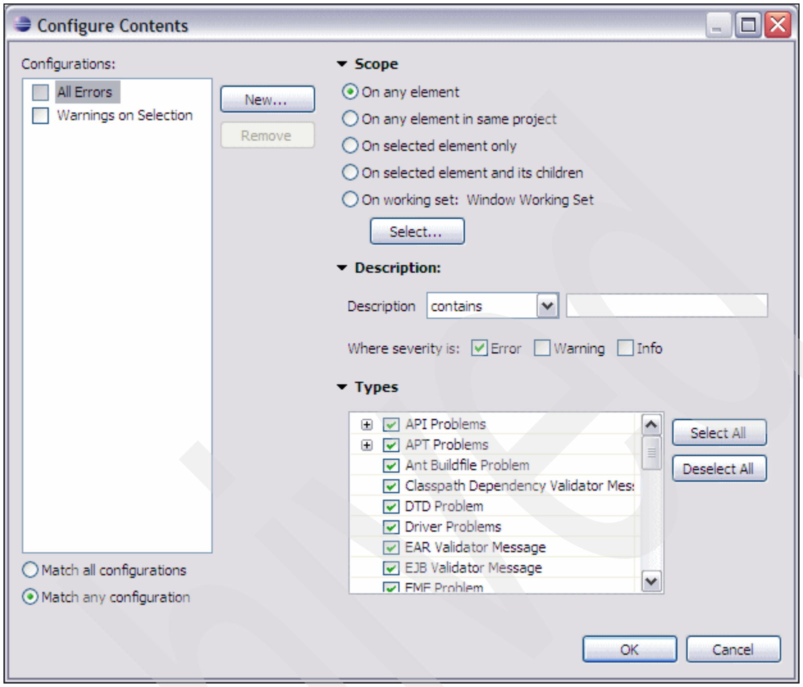
By now the Problems view shows one error for the DonateEAR enterprise application, because we have no modules defined yet. In addition, we see a number of warnings, such as:

- XML warning: No grammar constraints (DTD or XML schema) are detected for the document (`geronimo-application.xml`).
- Java warning: The serializable class `Fund` does not declare a static final serialVersionUID field of type long (`Fund.java`).

We can tailor the Problems view in a number of ways:

1. By default, errors and warnings of all projects are shown. You can limit the Problems view to only show problems of one project:

- Click the **Show Menu** icon ▽ (top right of the view), and select **Configure Contents**.



Configurations:

□ All Errors

□ Warnings on Selection

○ Match all configurations

● Match any configuration

Scope

● On any element

○ On any element in same project

○ On selected element only

○ On selected element and its children

○ On working set: Window Working Set

Description:

Description contains

Where severity is: ☑ Error □ Warning □ Info

Types

☑ API Problems

☑ APT Problems

☑ Ant Buildfile Problem

☑ Classpath Dependency Validator Mes:

☑ DTD Problem

☑ Driver Problems

☑ EAR Validator Message

☑ EJB Validator Message

☑ EMF Problem

- Select **On any element in same project** to limit the view to a selected project.xx.
- Select **All Errors** (on the left) to remove all the warnings.
- Select **Cancel** to close this pop-up without saving the changes. There are many other ways to be more selective about the problems shown in the Problems view. For now we will leave the default of showing all errors and warnings.

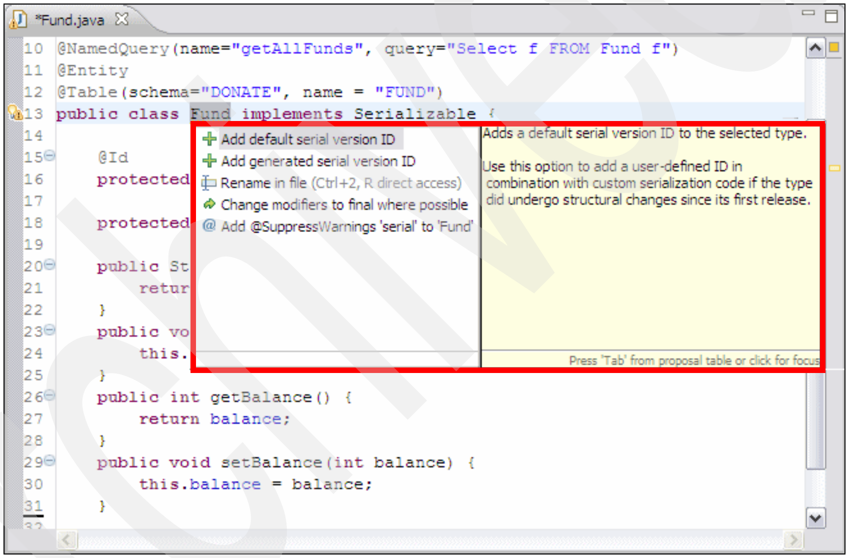
2. Resolve the warning: `No grammar constraints (DTD or XML schema) detected for the document ...` by disabling this type of warning globally (for all schema files that lack grammar constraints):

   - From the action bar, select **Window** → **Preferences**.
   - In the Preferences pop-up, perform the following steps:
     - On the left, select **XML** → **XML Files**.
     - On the right (at the resulting XML Files pane), change the Validating files/Indicate when no grammar is specified from **Warning** to **Ignore**.
     - Click **OK**.

Behind the scenes:

XML validation errors can only be disabled globally. There is no way to turn this type of warning for a single instance or project.

3. Resolve the warning: `The serializable class Fund does not declare a static final serialVersionUID field of type long` (`Fund.java`) by updating the code so that the warning is no longer issued.
   - Double-click the warning, and the `Fund` class opens.
   - Right-click on the warning icon at the left border, and select **Quick Fix**.
   - Select **Add default serial version ID** (the first suggestion).


*Fund.java
Add default serial version ID
Add generated serial version ID
Rename in file (Ctrl+2, R direct access)
Change modifiers to final where possible
Add @SuppressWarnings 'serial' to 'Fund'
Press 'Tab' from proposal table or click for focus

```
@NamedQuery(name="getAllFunds", query="Select f FROM Fund f")
@Entity
@Table(schema="DONATE", name = "FUND")
public class Fund implements Serializable {

    @Id
    protected

    protected

    public St
        retur
    }
    public vo
        this.
    }
    public int getBalance() {
        return balance;
    }
    public void setBalance(int balance) {
        this.balance = balance;
    }
```

Adds a default serial version ID to the selected type.

Use this option to add a user-defined ID in combination with custom serialization code if the type did undergo structural changes since its first release.

   - Double-click the first entry, and a serial version ID is created:

```
private static final long serialVersionUID = 1L;
```

   - Save and close the class, and the warning disappears.

Alternatives:

We could also have removed this warning by:

- Accepting the quick fix recommendation to suppress this warning for this instance using a @SuppressWarnings annotation:

  ```
  @SuppressWarnings("serial")
  @NamedQuery(name="getAllFunds", query="Select f FROM Fund f")
  @Entity
  @Table(schema="DONATE", name = "FUND")
  public class Fund implements Serializable {
  ```

- Globally disabling this warning for all instances through the Java compiler settings:
  - Open the Preferences pop-up (**Windows** → **Preferences**) and expand to **Java** → **Compiler** → **Errors/Warnings.**
  - In the Errors/Warnings pane (on the right), expand **Potential programming problems** and change Serializable class without serialVersionUID from **Warning** to **Ignore**.

Over time, you probably will use all of these approaches to fix or control the display of errors and warnings:

- Updating the code to resolve the error or warning (as we did here) generally is considered the best practice.
- You might not be able to easily update the code to resolve the warning, or perhaps in that specific situation the code is correct. In that case, you could choose to suppress that single instance.
- You might find that the particular error or warning exists in many of your files (or in the auto-generated files) and that the simplest approach is to ignore all of errors or warnings of that type. In that case, you can update the WASCE IDE to globally ignore the error or warning.

4. Optionally resolve the warning: `Type safety: Unchecked cast from List to List<Fund>` by adding the **@SuppressWarnings("unchecked")** annotation before the `getAllFunds` method.

5.11 Explore!

In this section we provide a few advanced topics for persistence.

5.11.1 Advanced topics for persistent entities

The object-oriented paradigm is largely used in application development in the industry. It models the world into application objects defining its behavior and state. Application objects needs to be persisted in case of a system crash or a power loss. Databases are used as the persistence layer of most of the applications, however, even if there are some object oriented databases systems on the market, relational databases are more pervasive. It means that the object oriented systems must map their objects to relational model. This mapping can be done ad hoc for each application, or a using a object-relational mapping (O/RM) framework.

Using an O/RM framework or API reduces the coupling between the application and the relational persistence layer by making it transparent. It also allows you to improve performance by having smart fetching, better inheritance mapping strategies, automatic dirty checking, and other mapping techniques.

The complex approach that was used to map the objects in EJB 2.x prompted the creation of a simpler persistence framework and APIs, such as Hibernate, iBatis, and Oracle TopLink.

Java EE 5 specifications defined EJB 3.0 specification (JSR 220), and its major enhancement is the addition of the Java Persistence API (JPA). JPA defines how Java objects that have to be persisted (persistent entities) are mapped to relational data to keep their state.

JPA standardizes the object-relational mapping. It makes large use of annotations, and still allows deployment descriptors that override the annotations, making development easier, and allows deployers to customize the application without changing the code.

For further O/RM, JPA, and EJB 3.0 references, refer to the following Web addresses:

- Design enterprise applications with the EJB 3.0 Java Persistence API:

  http://www.ibm.com/developerworks/java/library/j-ejb3jpa.html

- Get to know Java EE 5:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0707_barcia/0707_barcia.html

- The Java Persistence API - A Simpler Programming Model for Entity Persistence:

  http://java.sun.com/developer/technicalArticles/J2EE/jpa/

5.11.2 Experience advanced entities with JPA

In this chapter, we created object mappings for the simple entities (Employee and Fund). In this section we describe a more advanced object model and we show how to map it to a relational database using JPA.

We extended the object model defined in the core section to better illustrate the JPA mappings. The extensions were done in the DONATE database only, because based on the business model defined in the core chapter, the VACATION database is a legacy database and cannot be modified.

Original mapping

The original mapping has the following entities:

Employee Models the employee object; the employee can donate vacations to a vacation fund.

Fund Models the fund to receive any donation of vacation hours from an employee.

The two entities are completely independent.

Potential new mapping

The following entities are added:

FundAction This abstract class models any action done on a Fund.

Donation This class is a FundAction that represents any donation done by an employee to a fund.

Withdrawal This class is a FundAction that represents a withdrawal made by an employee from a fund.

Approval This is class that represents the information about the approval of a withdrawal.

The relationship between these classes are represented in a class diagram (Figure 5-2).

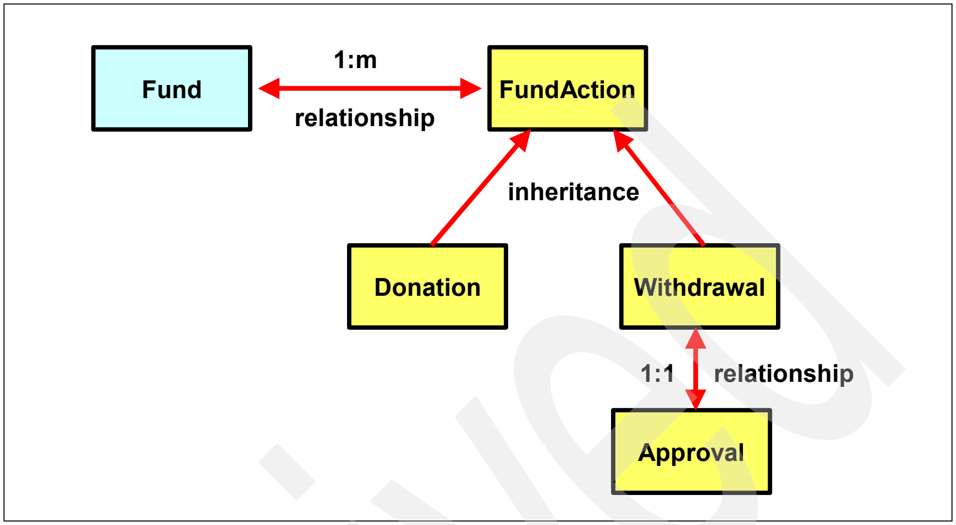


Figure 5-2 Advanced entity model with relationships and inheritance

Entity relationships

JPA support relationships between entities. It supports one-to-one, one-to-many, many-to-one, and many-to-many relationships. Entity objects define other entities as collection members.

In our example, we have a one-to-many relationship between `Fund` and `FundAction`, that is, a fund can have many actions. We also have a many-to-one relationship between `FundAction` and `Fund`, that is, an action is for one fund.

These relationship can be expressed using collections and annotations in the `Fund` and `FundAction` classes (Example 5-7).

Example 5-7 Relationship annotations

```
@Entity
public class Fund implements Serializable {

   @Id
   protected String name;

   protected int balance;

   @OneToMany(mappedBy="fund")
   private Set<FundAction> fundActionCollection;
   ......
```

```
==============================================================================

@Entity
public class FundAction implements Serializable {

   @Id
   private String id;

   int employeeId;

   private int hours;

   @ManyToOne
   private Fund fund;

   ......
```

Inheritance

A fund action (`FundAction` class) can either be a donation or a withdrawal. This can be expressed using inheritance, that is, the `Donation` and the `Withdrawal` classes are subclasses of `FundAction`.

As discussed in 5.1.6, "Entity inheritance" on page 115, inheritance can be implemented using a single table, or multiple tables. In a single table approach, a discriminator columns is used to distinguish between the different entity types.

Example 5-8 shows how a `FUNDACTION` table with a column named `action` can be used to map the three entities.

Example 5-8 Inheritance annotations

```
@Entity
@Table (name="FUNDACTION")
@Inheritance
@DiscriminatorColumn(name="action",\
                      discriminatorType=DiscriminatorType.STRING, length=8)
@DiscriminatorValue("FundAction")
public class FundAction implements Serializable {

      @Id
   private String id;

   private int hours;

   @ManyToOne
   private Fund fund;
   ......
================================================================================

@Entity
@Table (name="FUNDACTION")
@Inheritance
@DiscriminatorValue("Donation")
public class Donation extends FundAction {

   ......

================================================================================

@Entity
@Table (name="FUNDACTION")
@Inheritance
@DiscriminatorValue("Withdraw")
public class Withdrawal extends FundAction {

   String reason;

   @OneToOne
   public Approval getApproval() {
      return approval
   };

   ......
```

Note: We did not implement this scenario.

For examples using relationship and inheritance, refer to the IBM Redbooks publication, *WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0*, SG24-7611, or *Rational Application Developer V7.5 Programming Guide*, SG24-7672.

6

Create the entity facade session beans

In this chapter, we create we create the EJB 3.0 session beans, `EmployeeFacade` and `FundFacade`, that are facades to the two JPA entities, `Employee` and `Fund` (Figure 6-1).

6.1 Learn!

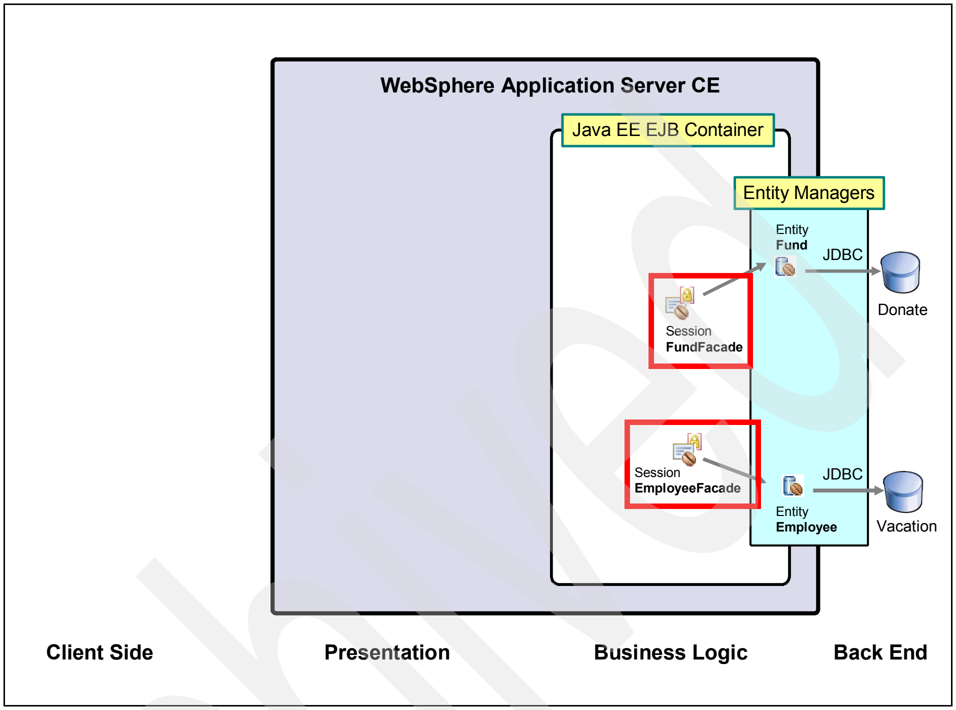


Figure 6-1 Donate application: Session facades for JPA entities

JPA entity objects, such as those you created in the preceding chapter, encapsulate data into a record-oriented format. The next challenge is, how do you link these objects together to form a business function that acts against one or more of these entities?

Typically, the application clients should not access JPA entities directly. The answer is a session EJB that provides an execution environment for the business logic. The EJB container provides the core services such as transactional support and local/remote invocation, allowing the user-written code to focus on the business-level functions.

The end application accesses the session EJB using local and remote invocation mechanisms. The user-written session EJB methods use the JPA entities to access the underlying databases.

The session EJB method is run as a single transaction, and when the method completes all actions against the various JPA entities (such as creating or deleting, or changing values) all are committed or rolled back. The user does not have to insert any code to handle this transactional activity.

The typical usage pattern is for client programs to access session EJBs using the remote interface. This provides the fail-over and workload management that exist when using a Java EE application server in a cluster environment. In this book, we are using a single-instance version of WASCE, so failover and workload management are not applicable.

Session EJBs, through the user-written code, access the associated JPA entities through an entity manager.

Session EJBs can also be used in situations with no entities to execute other transactional (JMS messaging, JCA adapters, user-written code with transactional contexts) and non-transactional (JavaBeans or Web services) artifacts. You still benefit from the following features:

- Local and remote invocation, along with the associated failover and workload management.
- Common access and security control across all technologies (Web front end, Java EE application client, Web services, and messaging).

Finally, there are two types of session EJBs:

- **Stateful**, where the EJB maintains state across a series of invocations, thus making the results of one call dependent on a previous call.

  For example, the first call can initialize a variable that is used in subsequent calls.
- **Stateless**, where the EJB does not maintain state, thus making each call completely independent from any other call.

Typical EJB best practices almost always recommend against stateful session EJBs, because the complexity they add to workload management and failover far outweigh their benefits and because most practical applications do not need this support.

If you do have an application that has to maintain state or conversational context across session EJB invocations, an alternate design pattern is to store the state information in an entity and to have the stateless session EJB return an index or key to the requester on the first call. The requester provides this index/key on subsequent calls, and the stateless session EJB can use this to retrieve the state from the entity.

Additional Learn resources are available at the following Web sites:

- Java EE Enterprise JavaBeans technology:

  http://java.sun.com/products/ejb/
- Mastering Enterprise JavaBeans 3.0:

  http://www.theserverside.com/tt/books/wiley/masteringEJB3/index.tss

6.1.1 EJB 3.0 specification

In this section we learn about the EJB 3.0 specifications that describes session EJBs and message-driven EJBs. This section is an extract from the IBM Redbooks publication *WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0*, SG24-7611, *Chapter 1, Introduction to EJB 3.0*.

6.1.2 EJB 3.0 simplified model

There are many publications that discuss the complexities and differences between the old EJB programming model and the new. For this reason, this book focuses on diving right into the new programming model. To overcome the limitations of the EJB 2.x, the new specification introduces a really new simplified model, whose main features are as follows:

- Entity EJBs are now plain old Java objects (POJO) that expose regular business interfaces (POJI), and there is no requirement for home interfaces.
- Removal of the requirement for specific interfaces and deployment descriptors (deployment descriptor information can be replaced by annotations).
- A complete new persistence model (based on the JPA standard), that supersedes EJB 2.x entity beans (see Chapter 5, “Create the JPA entities” on page 99).
- Interceptor facility to invoke user methods at the invocation of business methods or at life cycle events.
- Default values are used whenever possible (“configuration by exception” approach).
- Reduction of the requirements for usage of checked exception.

Figure 6-2 shows how the model of J2EE 1.4 has been completely reworked with the introduction of the EJB 3.0 specification.

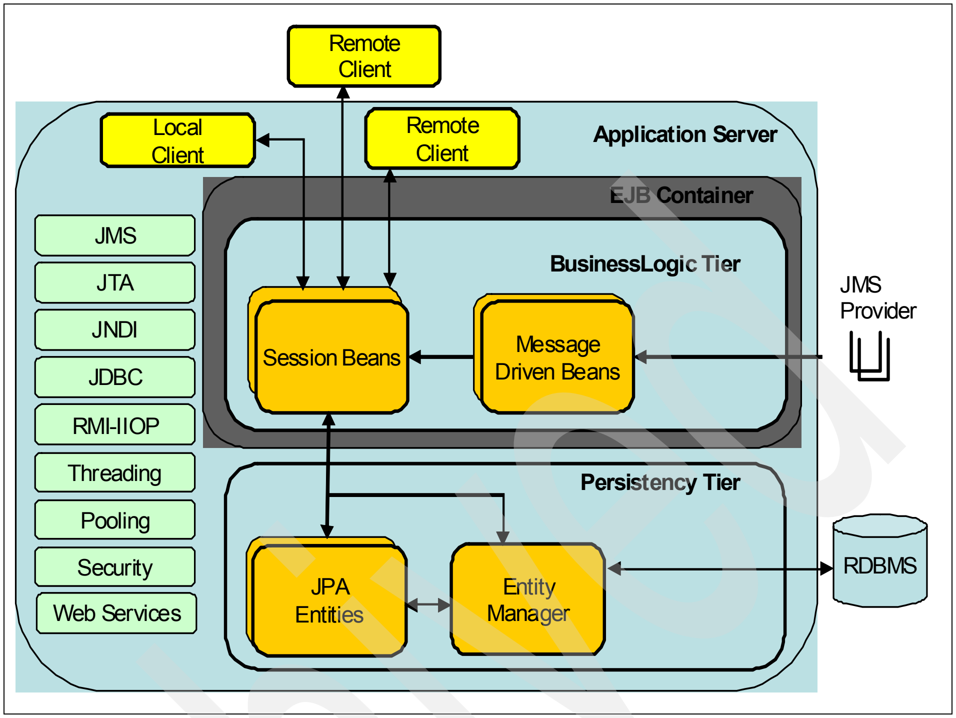


Figure 6-2 EJB 3.0 architecture

In this chapter, we introduce these features.

Metadata annotations

EJB 3.0 uses *metadata annotations*, as part of Java SE 5.0.

Annotations are similar to XDoclet, a powerful open-source framework extensible, metadata-driven, attribute-oriented framework that is used to generate Java code or XML deployment descriptors. However unlike XDoclet, that requires pre-compilation, annotations are syntax checked by the Java compiler at compile-time, and sometimes compiled into classes.

By specifying special annotations, developers can create POJO classes that are EJB components.

In the sections that follow we illustrate the most popular use of annotations and EJB 3.0 together.

6.1.3 EJB types and their definition

In EJB 3.0 there are two types of EJBs:

- Session beans (stateless and stateful)
- Message-driven beans (MDB)

> **Note:** EJB2.x entity beans have been replaced by JPA entities, which are discussed in Chapter 5, “Create the JPA entities” on page 99.

Annotations can be used to declare session beans and message-driven beans (MDB).

Stateless session EJBs

Session stateless EJBs have always been used to model a task being performed for a client code that invokes it. They implement business logic or rules of a system, and provide coordination of those activities between beans, such as a banking service, that allows for a transfer between accounts.

A stateless session bean is generally used for business logic that spans a single request and therefore cannot retain client-specific state among calls.

Because a stateless EJB does not maintain a conversational state, all the data exchanged between the client and the EJB must be passed either as input parameters, or as return value, declared on the EJB business method interface.

To better appreciate the simplification effort done by the EJB 3.0 specification, consider this comparison of the steps involved in the definition of an EJB according to 2.x and 3.0.

Steps to define a stateless session bean in EJB 2.x

To define a stateless session EJB for EJB 2.x, you have to define the following components:

- **EJB component interface**: Used by an EJB client to gain access to the capabilities of the bean. This is where the business methods are defined. The component interface is called the **EJB object**. There are two types of component interfaces (Figure 6-3):
  - Remote component (`EJBObject`): Used by a remote client to access the EJB through the RMI-IIOP protocol.
  - Local component (`EJBLocalObject`): Used by a local client (that runs inside the same JVM) to access the EJB.

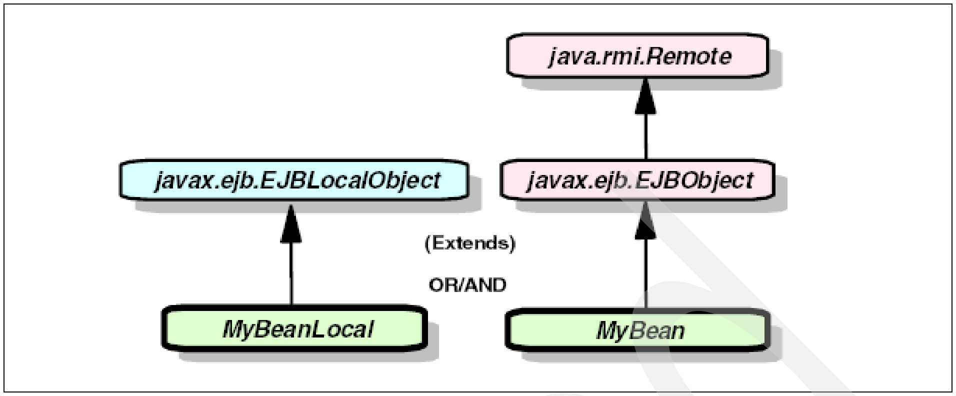


Figure 6-3 EJB 2.x component interfaces

- **EJB home interface**: Used by an EJB client to gain access to the bean. Contains the bean life cycle methods of create, find, or remove. The home interface is called the **EJB home**. The `EJBHome` object is an object which implements the home interface, and as in `EJBObject`, is generated from the container tools during deployment, and includes container specific code. At startup time, the EJB container instantiates the `EJBHome` objects of the deployed enterprise beans and registers the home in the naming service. An EJB client accesses the `EJBHome` objects using the Java Naming and Directory Interface (JNDI). There are two types of home interfaces (Figure 6-4):
  - Remote home interface (`EJBHome`): Used by a remote client to access the EJB through the RMI-IIOP protocol.
  - Local home interface (`EJBLocalHome`): Used by a local client (that runs inside the same JVM) to access the EJB.

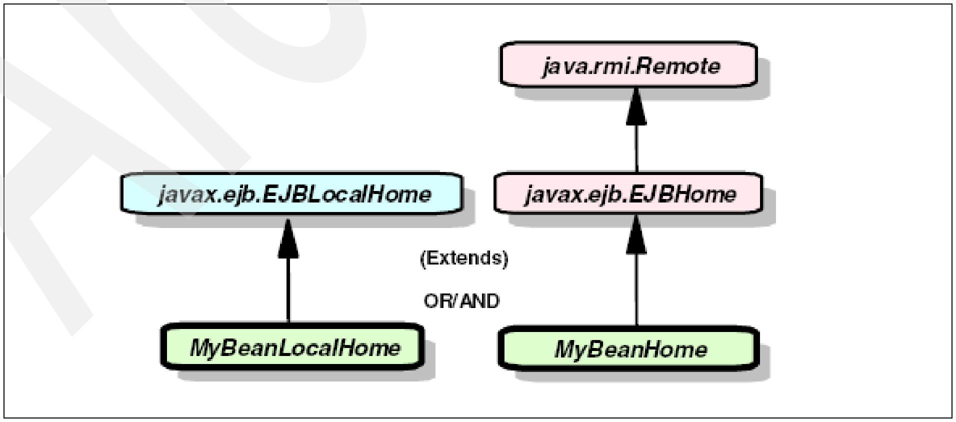


Figure 6-4 EJB 2.x home interfaces

- **EJB bean class**: Contains all of the actual bean business logic. Is the class that provides the business logic implementation. Methods in this bean class associate to methods in the component and home interfaces.

Steps to define a stateless session bean in EJB 3.0

To declare a session stateless bean, according to EJB 3.0 specification, you can simply define a POJO:

```
@Stateless
public class MyFirstSessionBean implements MyBusinessInterface {

   // business methods according to MyBusinessInterface
   .....
}
```

Consider the new, revolutionary aspects of this approach:

- `MyFirstSessionBean` is a POJO that exposes a plain old Java interface (POJI), in this case `MyBusinessInterface`. This interface will be available to clients to invoke the EJB business methods.
- The **@Stateless** annotation indicates to the container that the given bean is stateless session bean so that the proper life cycle and runtime semantics can be enforced.
- By default, this session bean is accessed through a local interface.

This is all you need to set up a session EJB! There are no special classes to extend and no interfaces to implement. The very simple model of EJB 3.0 is shown in Figure 6-5.

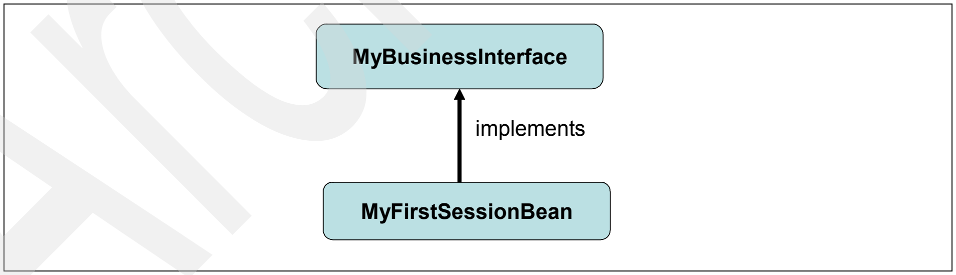


Figure 6-5 EJB is a POJO exposing a POJI

On the other hand, if we want to expose the same bean on the remote interface, we would use the **@Remote** annotation:

```
@Remote(MyRemoteBusinessInterface.class)
@Stateless
public class MyBean implements MyRemoteBusinessInterface {
   // ejb methods
   .....
}
```

Tip: If the session bean implements only one interface, you can also just code @Remote (without a class name).

Stateful session EJBs

Stateful session EJBs are usually used to model a task or business process that spans multiple client requests, therefore, a stateful session bean retains state on behalf of an individual client. The client, on the other hand, has to store the handle to the stateful EJB, so that it always accesses the same EJB instance.

Using the same approach we adopted before, to define a stateful session EJB is to declare a POJO with the @Stateful annotation:

```
@Stateful
public class MySecondSessionBean implements MyBusinessStatefulInterface {

   // ejb methods
   .....
}
```

The **@Stateful** annotation indicates to the container that the given bean is stateful session bean so that the proper life cycle and runtime semantics can be enforced.

Business interfaces

EJBs can expose different business interfaces, according to the fact that the EJB could be accessed either from a local or remote client. We recommend that common behaviors to both local and remote interfaces should be placed in a super-interface, as shown in Figure 6-6.

Please take care of the following aspects:

- A business interface cannot be both a local and a remote business interface of the bean.
- If a bean class implements a single interface, that interface is assumed to be the business interface of the bean. This business interface will be a *local* interface, unless the interface is designated as a remote business interface by use of the @Remote annotation or by means of the deployment descriptor.

This gives you a great flexibility during the design phase, because you can decide which methods are visible to local and remote clients.

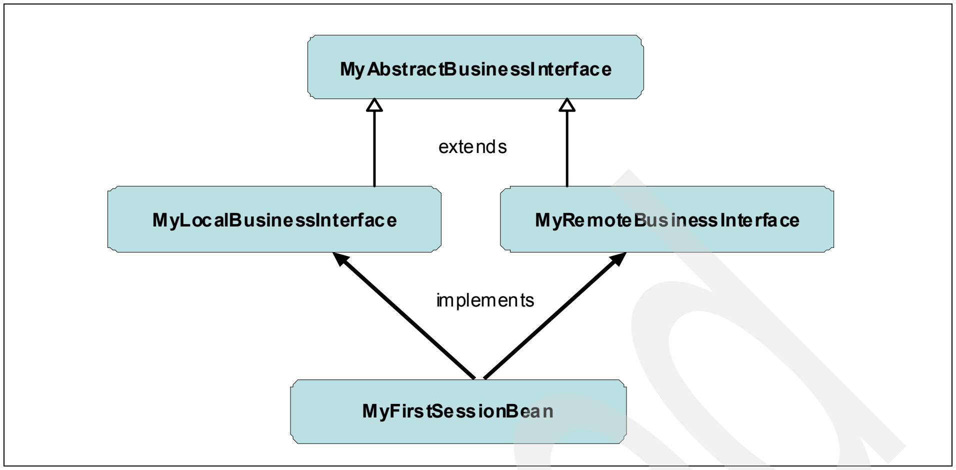


Figure 6-6 How to organize the EJB component interface

Using these guidelines, our first EJB is refactored as shown here:

```
@Stateless
public class MyFirstSessionBean
      implements MyLocalBusinessInterface, MyRemoteBusinessInterface {

   // implementation of methods declared in MyLocalBusinessInterface
   ....
   ....

   // implementation of methods declared in MyRemoteBusinessInterface
   ....
   ....
}
```

The `MyLocalBusinessInterface` is declared as interfaces with a @Local or @Remote annotation:

```
@Local
public interface MyLocalBusinessInterface
                              extends MyAbstractBusinessInterface {
   // methods declared in MyLocalBusinessInterface
   ......
}

@Remote
public interface MyRemoteBusinessInterface
                               extends MyAbstractBusinessInterface {
   // methods declared in MyRemoteBusinessInterface
   ......
}
```

Another techniques to define the business interfaces exposed either as local or remote is to specify **@Local** or **@Remote** annotations with the full class that implements these interfaces:

```
@Stateless
@Local(MyLocalBusinessInterface.class)
@Remote(MyRemoteBusinessInterface.class)
public class MyFirstSessionBean implements MyLocalBusinessInterface,
                                           MyRemoteBusinessInterface {

   // implementation of methods declared in MyLocalBusinessInterface
   ....
   ....

   // implementation of methods declared in MyRemoteBusinessInterface
   ....
   ....
}
```

You can declare arbitrary exceptions on the business interface, but take into account the following rules:

- Do not use `RemoteException`.
- Any runtime exception thrown by the container is wrapped into an `EJBException`.

6.1.4 Best practices for developing EJBs

An EJB 3.0 developer must be adherent to the following basic rules:

- Each session bean must be a POJO, the class must be concrete (therefore neither abstract or final), and must have a no-argument constructor (if not present, the compiler will insert a default constructor).
- The POJO must implement *at least* one POJI. We stress at least, because you can have different interfaces for local and remote clients.
- If the business interface is @Remote annotated, all the values passed through the interface must implement `java.io.Serializable`. Typically the declared parameters are defined serializable, but this is not required as long as the actual values passed are serializable.

6.1.5 Message-driven beans

Exposing EJB 3.0 beans as message-driven beans (MDBs) using the @MessageDriven annotation is covered in Chapter 13, “Create the message-driven bean” on page 405, where we show how to implement an MDB that makes calls to EJB 3.0 session beans.

6.1.6 Web services

Exposing EJB 3.0 beans as Web services using the @WebService annotation is covered in Chapter 11, “Create the Web service” on page 323, where we show how to implement a Web service from an EJB 3.0 session bean.

6.1.7 Life cycle events

Another very powerful use of annotations is to *mark* callback methods for session bean life cycle events.

EJB 2.1 and prior releases required implementation of several life cycle methods, such as `ejbPassivate`, `ejbActivate`, `ejbLoad`, and `ejbStore`, for every EJB, even if you do not need these methods.

Because we use POJOs in EJB 3.0, the implementation of these life cycle methods has been made optional. Only if you implement any callback method in the EJB, would the container invoke that specific method.

The life cycle of a session bean can be categorized into several phases or events. The most obvious two events of a bean life cycle are creation and destruction for stateless session beans.

After the container creates an instance of a session bean, the container performs any *dependency injection* (described in the section that follows), and then invokes the method annotated with **@PostConstruct** (if there is one).

The client obtains a reference to a session bean, and then invokes a business method.

> **Note:** The actual life cycle of a **stateless** session bean is independent of when a client obtains a reference to it. For example, the container can hand out a reference to the client, but not create the bean instance until some later time, for example, when a method is actually invoked on the reference. Or, the container can create a number of instances at startup time, and match them up with references at a later time.

At the end of the life cycle, the EJB container calls the method annotated with **@PreDestroy**, if there is one. The bean instance is then ready for garbage collection.

A stateless session bean with the two callback methods is shown here:

```
@Stateless
public class MyStatelessBean implements MyBusinessLogic {
    // .. bean business method

    @PostConstruct
    public void initialize() {
        // initialize the resources uses by the bean
    }

    @PreDestroy
    public void cleanup() {
        // deallocates the resources uses by the bean
    }
}
```

All stateless and stateful EJBs go through these two phases.

In addition, stateful session beans go through the passivation/activation cycle. Because an instance of a stateful bean is bound to a specific client (and therefore it cannot be reused among different requests), and the EJB container must manage the amount of physical available resources, the EJB container might decide to deactivate, or passivate, the bean by moving it from memory to secondary storage.

In correspondence with this more complex life cycle, we have further callback methods, specific to stateful session beans:

- The EJB container invokes the method annotated with **@PrePassivate**, immediately before passivating it.
- If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean by calling the method annotated with **@PostActivate**, if any, and then moves it to the ready stage.
- At the end of the life cycle, the client explicitly invokes a method annotated with **@Remove**, and the EJB container in turns calls the callback method annotated **@PreDestroy**.

Note: Because a stateful bean is bound to a particular client, it is best practice to correctly design session stateful beans to minimize their footprint inside the EJB container, and to correctly un-allocate it at the end of its life cycle, by invoking the method annotated with @Remove.

Stateful session beans have a *time-out* value. If the stateful session bean has not been used in the time-out period, it is marked inactive and is eligible for automatic deletion by the EJB container. Of course, it is still best practice for applications to remove the bean when the client is through with it, rather than relying on the time-out mechanism to do this.

Developers can explicitly invoke only the life cycle method annotated with @Remove, the other methods are invoked automatically by the EJB container.

6.1.8 Interceptors

The EJB 3.0 specification defines the ability to apply custom made interceptors to the business methods of both session and message driven beans. Interceptors take the form of methods annotated with the @AroundInvoke annotation (Example 6-1).

Example 6-1 Applying an interceptor

```
@Stateless
public class MySessionBean implements MyBusinessInterface {

    @Interceptors(LoggerInterceptor.class)
    public Customer getCustomer(String ssn) {
      ...
   }
    ......
}

public class LoggerInterceptor {
   @AroundInvoke
   public Object logMethodEntry(InvocationContext invocationContext)
                                                       throws Exception {
      System.out.println("Entering method: "
                            + invocationContext.getMethod().getName());
      Object result = invocationContext.proceed();
      // could have more logic here
      return result;
   }
}
```

We have the following notes for this example:

- The @Interceptors annotation is used to identify the session bean method where the interceptor should be applied;
- The `LoggerInterceptor` interceptor class defines a method (`logMethodEntry`) annotated with @AroundInvoke.
- The `logMethodEntry` method contains the advisor logic (in this case very simply logs the invoked method name), and invokes the `proceed` method on the `InvocationContext` interface to advice the container to proceed with the execution of the business method.

The implementation of interceptor in EJB 3.0 is a bit different from the analogous implementation of this aspect-oriented programming (AOP) paradigm that you can find in frameworks such as Spring or AspectJ, because EJB 3.0 does not support *before* or *after* advisors, but only *around* interceptors. However, *around* interceptors can act as *before* or *after* interceptors, or both. Interceptor code before the `invocationContext.proceed` call is run before the EJB method, and interceptor code after that call is run after the EJB method.

A very common use of interceptors is to provide preliminary checks (validation, security, and so forth) before the invocation of business logic tasks, and therefore they can throw exceptions. Because the interceptor is called together with the session bean code at run-time, these potential exceptions are sent directly to the invoking client.

In this sample we have seen an interceptor applied on a specific method; actually, the @Interceptors annotation can be applied at class level. In this case the interceptor will be called for every method.

Furthermore, the @Interceptors annotation accepts a list of classes, so that multiple interceptors can be applied to the same object.

Note: To give further flexibility, EJB 3.0 introduces the concept of a default interceptor that can be applied on every session (or MDB) bean contained inside the same EJB module. A default interceptor cannot be specified using an annotation, instead you should define it inside the deployment descriptor of the EJB module.

Note the following execution order in which interceptors are run:

- Default interceptor
- Class interceptors
- Method interceptors

To disable the invocation of a default interceptor or a class interceptor on a specific method, you can use the @ExcludeDefaultInterceptors and @ExcludeClassInterceptors annotations, respectively.

6.1.9 Dependency injection

The new specification introduces a powerful mechanism for obtaining Java EE resources (JDBC data source, JMS factories and queues, EJB references) and to inject them into EJBs, entities, or EJB clients.

In EJB 2.x the only way to obtain these resources was to use JNDI lookup using resource references, with a piece of code that could become cumbersome and vendor specific, because very often you had to specify properties related to the specific J2EE container provider.

EJB 3.0 adopts a *dependency injection* (DI) pattern, which is one of best ways to implement loosely coupled applications. It is much easier to use and more elegant than older approaches, such as dependency lookup through JNDI or container callbacks.

The implementation of dependency injection in the EJB 3.0 specification is based on annotations or XML descriptor entries, which allow you to inject dependencies on fields or setter methods.

Instead of complicated XML EJB references or resource references, you can use the @EJB and @Resource annotations to set the value of a field or to call a setter method within the beans with anything registered within JNDI. With these annotations, you can inject EJB references and resource references such as data sources and JMS factories.

In this section we show the most common usage of dependency injection in EJB 3.0.

@EJB annotation

The @EJB annotation is used for injecting session beans into a client. This injection is only possible within managed environments, such as another EJB, or a servlet. We cannot inject an EJB into a JSF managed bean or Struts action, for example.

The parameters for the @EJB annotation are optional. The annotation parameters are:

- `name`: Specifies the JNDI name that is used to bind the injected EJB in the environment naming context (`java:comp/env`).
- `beanInterface`: Specifies the business interface to be used to access the EJB. By default, the business interface to be used is taken from the Java type of the field into which the EJB is injected. However, if the field is a supertype of the business interface, or if method-based injection is used rather than field-based injection, the `beanInterface` parameter is typically required, because the specific interface type to be used might be ambiguous without the additional information provided by this parameter.
- `beanName`: Specifies a *hint* to the system of the `ejb-name` of the target EJB that should be injected. It is analogous to the `<ejb-link>` stanza that can be added to an `<ejb-ref>` or `<ejb-local-ref>` stanza in the XML descriptor.

To access a session bean from a Java servlet we use the code shown in Example 6-2.

Example 6-2 Injecting an EJB reference inside a servlet

```
import javax.ejb.EJB;
public class TestServlet extends javax.servlet.http.HttpServlet
                        implements javax.servlet.Servlet {

   // inject the remote business interface
   @EJB(beanInterface=MyRemoteBusinessInterface.class)
   MyAbstractBusinessInterface serviceProvider;

   protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                        throws ServletException, IOException {
      // call ejb method
      serviceProvider.myBusinessMethod();
      ......
   }
}
```

Here are some remarks on this example:

- We have specified the `beanInterface` attribute, because the EJB exposes two business interfaces (`MyRemoteBusinessInterface` and `MyLocalBusinessInterface`).
- If the EJB exposes only one interface, you are not required to specify this attribute, however, it can be useful to make the client code more readable.

@Resource annotation

The @Resource annotation is the main annotation that can be used to inject resources in a managed component. In the following we show the most common usage scenarios of this annotation.

We show here how to inject a typical resource such as a data source inside a session bean.

Field injection technique

Example 6-3 shows how to inject a data source inside a property that is used in a business method:

```
   @Resource (name="jdbc/dataSource")
```

Example 6-3 Field injection technique for a data source

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

   @Resource (name="jdbc/dataSource")
   private DataSource ds;

   public void businessMethod1() {
      java.sql.Connection c=null;
      try {
          c = ds.getConnection();
          // .. use the connection
      } catch (java.sql.SQLException e) {
         // ... manage the exception
      } finally {
         // close the connection
         if(c!=null) {
            try { c.close(); } catch (SQLException e) { }
         }
      }
   }
}
```

All parameters for the @Resource annotation are optional. The annotation parameters are:

- `name`: Specifies the component-specific internal name—the resource reference name—within the **`java:comp/env`** name space. It does not specify the global JNDI name of the resource being injected. A binding of the reference reference to a JNDI name is necessary to provide that linkage, just as it is in J2EE 1.4.
- `type`: Specifies the resource manager connection factory type.
- `authenticationType`: Specifies whether the container or the bean is to perform authentication.
- `shareable`: Specifies whether resource connections are shareable or not.
- `mappedName`: Specifies a product specific name that the resource should be mapped to. WebSphere does not make any use of `mappedName`.
- `description`: Description.

Setter method injection

Another technique is to inject a setter method. The setter injection technique is based on JavaBeans property naming conventions (Example 6-4).

Example 6-4 Setter method injection technique for a data source

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

   private Datasource ds;

   @Resource (name="jdbc/dataSource")
   public void setDatasource(DataSource datasource) {
      this.ds = datasource;
   }
   ...
   public void businessMethod1() {
     ...
   }
}
```

Here are some remarks on these two examples:

- In this example, we directly used the data source inside the session bean. This is not good practice, because you should put JDBC code in specific components, such as data access objects.
- We recommend that you use the setter injection technique, which gives more flexibility:
  - You can put initialization code inside the setter method.
  - The session bean can be easily tested as a stand-alone component.

Other interesting usages of @Resource are as follows:

- To obtain a reference to the EJB session context:

  ```
  @Stateless
  public class CustomerSessionBean implements CustomerServiceInterface {
     ....
     @Resource javax.ejb.SessionContext ctx;

  }
  ```

- To obtain the value of an environment variable, which is configured inside the deployment descriptor with env-entry:

  ```
  @Stateless
  public class CustomerSessionBean implements CustomerServiceInterface {
     ....
     @Resource String myEnvironmentVariable;

  }
  ```

- Injection of JMS resources, such as JMS factories or queues.

6.1.10 Using deployment descriptors

Up to now we have seen how to define an EJB, how to inject resources into it, or how to specify its life cycle event with annotations. We can get the same result by specifying a deployment descriptor (`ejb-jar.xml`) with the necessary information in the EJB module.

6.1.11 EJB 3.0 application packaging

Session beans and MDBs can be packaged in Java standard JAR file. This can be achieved by two strategies:

- Using a Java project or Java Utility project
- Using an EJB project

If you use the first approach you have to add the Java Project to the EAR project, by editing the deployment descriptor (`application.xml`), and adding the lines:

```
<module>
   <ejb>MyEJB3Module.jar</ejb>
</module>
```

When using the second approach, the IDE can automatically update the `application.xml` file, and set up an `ejb-jar.xml` inside the EJB project. However, in EJB 3.0 you are not required to define the EJBs and related resources in an `ejb-jar.xml` file, because they are usually defined through the use of annotations. The main usage of deployment descriptor files is to override or complete behavior specified by annotations.

6.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed these steps:

- Defined the two databases (`Vacation` and `Donate`)
- Defined the matching database pools (`VacationPool` and `DonatePool`)
- Created the JPA entities (`Employee` and `Fund`)
- Created and run the JUnit test case (`EntityTester`)

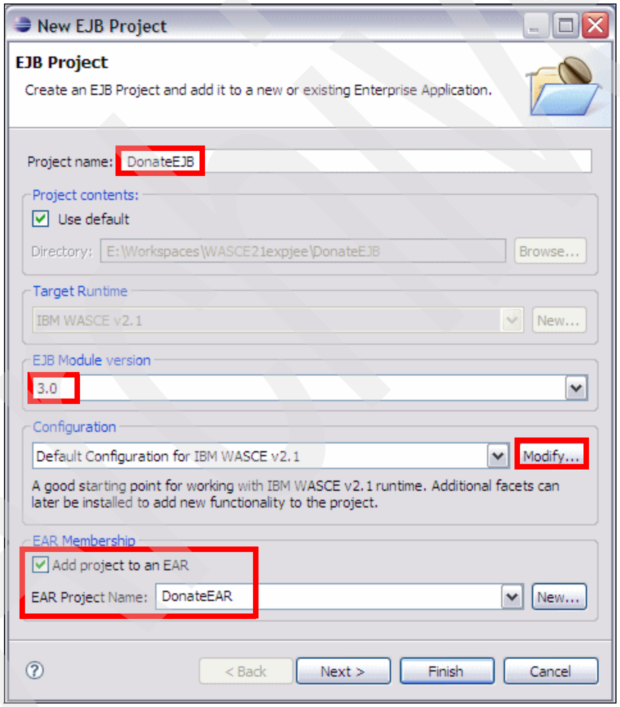
6.3 Create and configure an EJB 3.0 project

In this section, we create the DonateEJB project to be contained in the DonateEAR project. An EJB3.0 project generally contains session beans, message-driven beans, and JPA entities, all packaged together as a JAR file.

In our design, we created the JPA entities in separate JPA projects, DonateJPAEmployee and DonateJPAFund. The DonateEJB project only contains session beans and a message-driven bean.

6.3.1 Create the Project

1. In the Java EE perspective, Project Explorer, right-click and select **New** → **EJB Project**.
2. In the New EJB Project/EJB Project pop-up:



New EJB Project

EJB Project

Create an EJB Project and add it to a new or existing Enterprise Application.

Project name: DonateEJB

Project contents:

☑ Use default

Directory: E:\Workspaces\WASCE21expjee\DonateEJB

Target Runtime: IBM WASCE v2.1

EJB Module version: 3.0

Configuration: Default Configuration for IBM WASCE v2.1

A good starting point for working with IBM WASCE v2.1 runtime. Additional facets can later be installed to add new functionality to the project.

EAR Membership

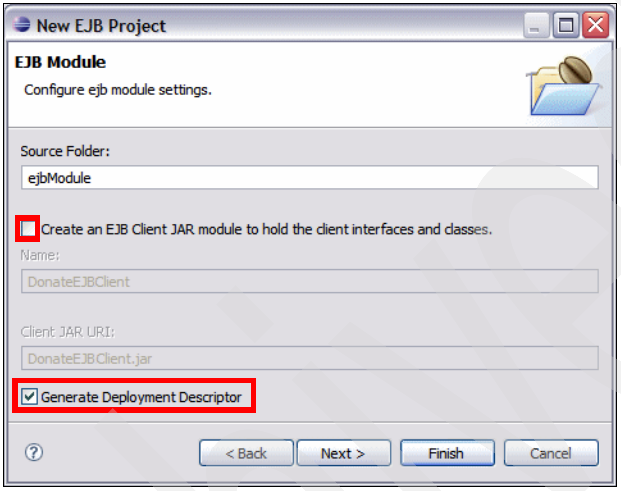
☑ Add project to an EAR

EAR Project Name: DonateEAR

- Set the project name as **DonateEJB**.
- Accept the default of 3.0 for EJB Module version.
- Next to Configuration, click **Modify**. Verify that EJB Module, Java, and WASCE Deployment are selected. Click **OK**.

- Back in the New EJB Project/EJB Project pop-up, select **Add project to an EAR** and select **DonateEAR** as the EAR Project Name.
- Click **Next**.

3. In the New EJB Project/EJB Module pop-up, clear **Create an EJB Client JAR module to hold the client interfaces and classes**, select **Generate Deployment Descriptor**, and click **Next**.



New EJB Project

EJB Module

Configure ejb module settings.

Source Folder: ejbModule

□ Create an EJB Client JAR module to hold the client interfaces and classes.

Name: DonateEJBClient

Client JAR URI: DonateEJBClient.jar

☑ Generate Deployment Descriptor

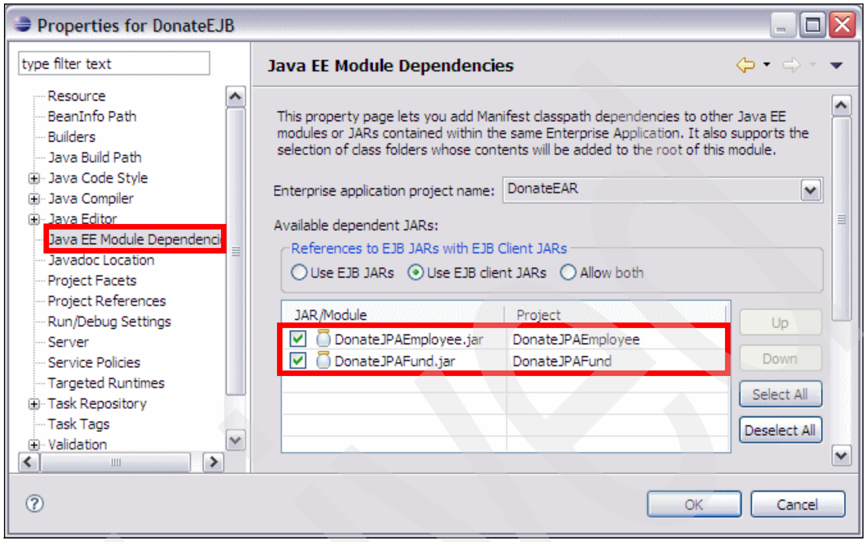
4. In the New EJB Project/Geronimo Deployment Plan pop-up, set the Group Id as **ejee** (to match the enterprise application), leave the Artifact Id empty (otherwise it overwrites the EAR application), accept the other fields, and click **Next**.
5. Verify that the DonateEJB project is visible in the Project Explorer. Notice the EJB deployment descriptor (`ejb-jar.xml`) and the Geronimo deployment plan (`openejb-jar.xml`), both in `ejbModule/META-INF`.
6. Notice that the error in DonateEAR disappeared. The EJB module has been added to the `application.xml` file.

6.3.2 Add project dependencies

The DonateEJB project requires access to the JPA entities to interact with the databases. Remember that the references to the database plans were made in the DonateEAR deployment pan (refer to 5.5.3, “Add dependencies to the EAR deployment plan” on page 136).

1. Right-click **DonateEJB** and select **Properties**.

2. In the Properties pop-up, select **Java EE Module Dependencies**, and select the two JPA projects, **DonateJPAEmployee.jar** and **DonateJPAFund.jar**. Click **OK**.



Properties for DonateEJB

type filter text

- Resource
- BeanInfo Path
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Java EE Module Dependenci
- Javadoc Location
- Project Facets
- Project References
- Run/Debug Settings
- Server
- Service Policies
- Targeted Runtimes
- Task Repository
- Task Tags
- Validation

Java EE Module Dependencies

This property page lets you add Manifest classpath dependencies to other Java EE modules or JARs contained within the same Enterprise Application. It also supports the selection of class folders whose contents will be added to the root of this module.

Enterprise application project name: DonateEAR

Available dependent JARs:

References to EJB JARs with EJB Client JARs

○ Use EJB JARs ● Use EJB client JARs ○ Allow both

JAR/Module	Project
☑ DonateJPAEmployee.jar	DonateJPAEmployee
☑ DonateJPAFund.jar	DonateJPAFund

6.4 Create the entity facade interfaces and session beans

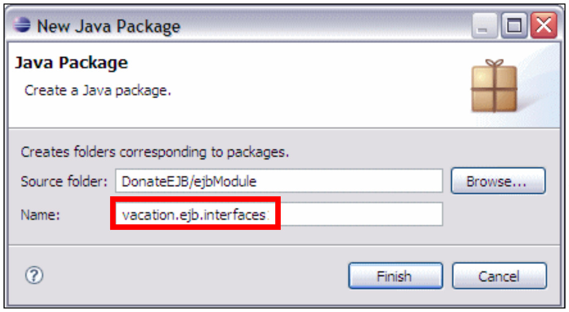
We do not want clients (such as servlets) to use the JPA entities directly. Therefore, we provide a session facade for each entity, and a business session bean for general application logic.

In this section we create the session bean facade for the `Employee` and the `Fund` entities.

6.4.1 Define the business interface for the employee facade

First we define the business interface that is implemented by the employee facade session bean. Open the Java EE perspective:

1. In the Project Explorer, expand the DonateEJB project, right-click **ejbModule** and select **New** → **Package**.
2. In the New Java Package/Java Package pop-up, type a name of **vacation.ejb.interfaces** and click **Finish**.



New Java Package

Java Package

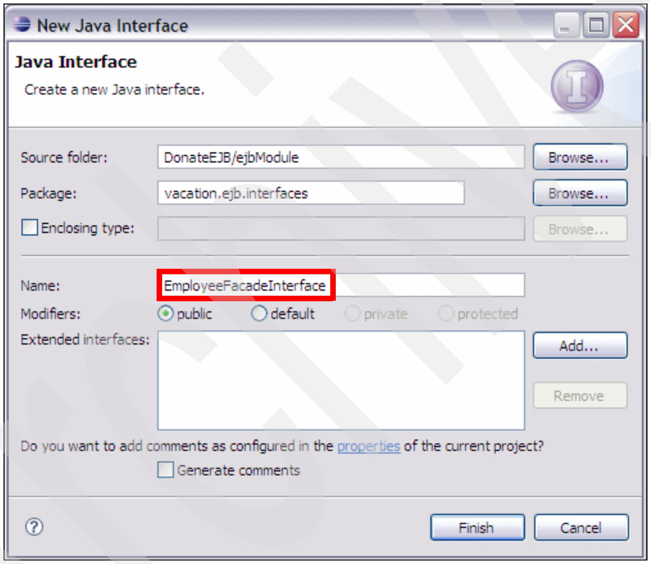
Create a Java package.

Creates folders corresponding to packages.

Source folder: DonateEJB/ejbModule

Name: vacation.ejb.interfaces

3. In the Project Explorer, right-click **vacation.ejb.interfaces** (in `DonateEJB/ejbModule`) and select **New** → **Interface**. Type **EmployeeFacadeInterface** as name, and click **Finish**.



New Java Interface

Java Interface

Create a new Java interface.

Source folder: DonateEJB/ejbModule

Package: vacation.ejb.interfaces

□ Enclosing type:

Name: EmployeeFacadeInterface

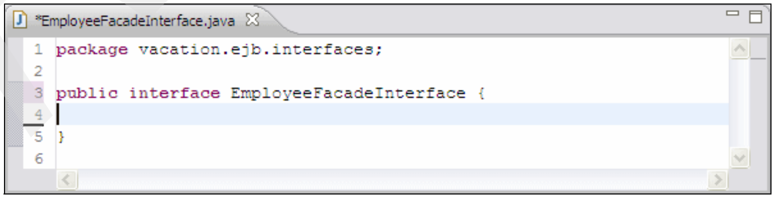
Modifiers: ● public ○ default ○ private ○ protected

Extended interfaces:

Do you want to add comments as configured in the properties of the current project?

□ Generate comments

4. In the Java Editor, move the cursor to the interface body.



*EmployeeFacadeInterface.java

```
package vacation.ejb.interfaces;

public interface EmployeeFacadeInterface {

}

```

5. In the Snippets view, double-click the **F10 EmployeeFacade Interface** snippet to add the method signatures to the interface.
6. You should now see the method signatures added to the interface body:

```
public interface EmployeeFacadeInterface {
   public Employee findEmployeeById(int id) throws Exception;
   public Employee addVacationHours(Employee employee, int hours);
   public Employee deductVacationHours(Employee employee, int hours)
                                                throws Exception;
}
```

7. Organize the imports (select **Source** → **Organize Imports** or **Ctrl+Shift+O**).

```
import vacation.entities.Employee;
```

8. Save and close `EmployeeFacadeInterface.java`.

Behind the scenes:

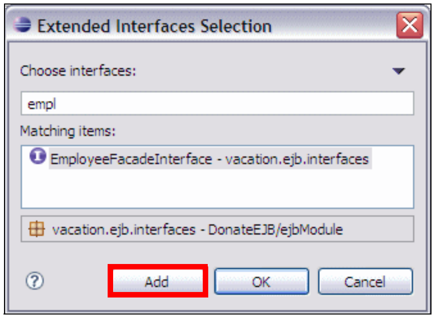
The business interface, by default, is a local interface. We could add the @Local annotation to the interface definition to explicitly state that this interface is intended to be used as a local interface, or we could define a separate local interface that extends the business interface.

In the section that follows, we define a remote interface that extends the business interface.

6.4.2 Define the remote interface for the employee facade

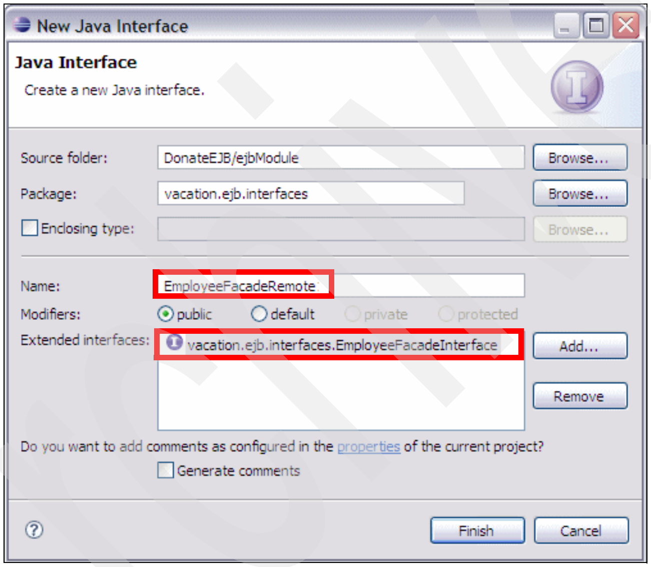
The session bean will be accessed from remote clients. Therefore, we also define a remote interface for the session bean facade:

1. In the Project Explorer, right-click **vacation.ejb.interfaces** (in `DonateEJB/ejbModule`) and select **New** → **Interface**.
2. In the New Java Interface pop-up, type a name of **EmployeeFacadeRemote**, and click **Add** next to the Extended interfaces list.
3. In the Extended Interfaces Selection pop-up, begin typing **EmployeeFacadeInterface** until the matching item appears. Select the matching item, click **Add** and click **OK**.



Extended Interfaces Selection
Choose interfaces:
empl
Matching items:
EmployeeFacadeInterface - vacation.ejb.interfaces
vacation.ejb.interfaces - DonateEJB/ejbModule
Add
OK
Cancel

4. The `EmployeeFacadeInterface` is now listed in the Extended interfaces list. Click **Finish**.



Source folder: DonateEJB/ejbModule

Package: vacation.ejb.interfaces

□ Enclosing type:

Name: EmployeeFacadeRemote

Modifiers: ● public ○ default ○ private ○ protected

Extended interfaces: vacation.ejb.interfaces.EmployeeFacadeInterface

Do you want to add comments as configured in the properties of the current project?

□ Generate comments

5. In the Java Editor, add a **@Remote** annotation above the method signature:

```
@Remote
public interface EmployeeFacadeRemote extends EmployeeFacadeInterface {
}
```

The @Remote annotation makes this interface available to remote clients.

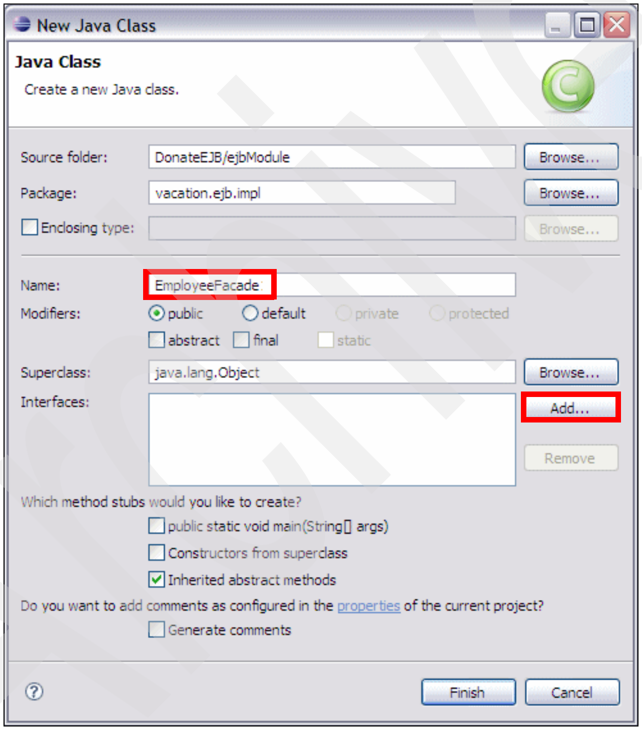
6. Organize imports (**Ctrl+Shift+O**) to add the import for `javax.ejb.Remote`.

7. Save and close `EmployeeFacadeRemote.java`.

6.4.3 Create the session bean facade for the employee entity

In this section we create the session bean facade (`EmployeeFacade`) for the `Employee` entity:

1. In the Project Explorer, expand the DonateEJB project, right-click on the **ejbModule** source folder and select **New** → **Package**.
2. In the New Java Package/Java Package pop-up, type a name of **vacation.ejb.impl** and click **Finish**.
3. Right-click **vacation.ejb.imp** (in `DonateEJB/ejbModule`) and select **New** → **Class**.
4. In the New Java Class/Java Class pop-up, type a name of **EmployeeFacade**, and click **Add** next to the Interfaces list.



New Java Class

Java Class

Create a new Java class.

Source folder: DonateEJB/ejbModule

Package: vacation.ejb.impl

□ Enclosing type:

Name: EmployeeFacade

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

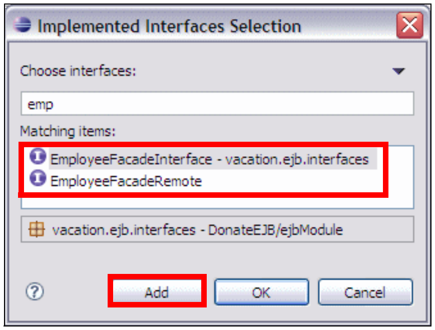
□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments as configured in the properties of the current project?

□ Generate comments

5. In the Implemented Interfaces Selection pop-up, begin typing **EmployeeFacade** until the two interfaces you created appear. Select each of these interfaces and click **Add**. Click **OK**.



Implemented Interfaces Selection

Choose interfaces:

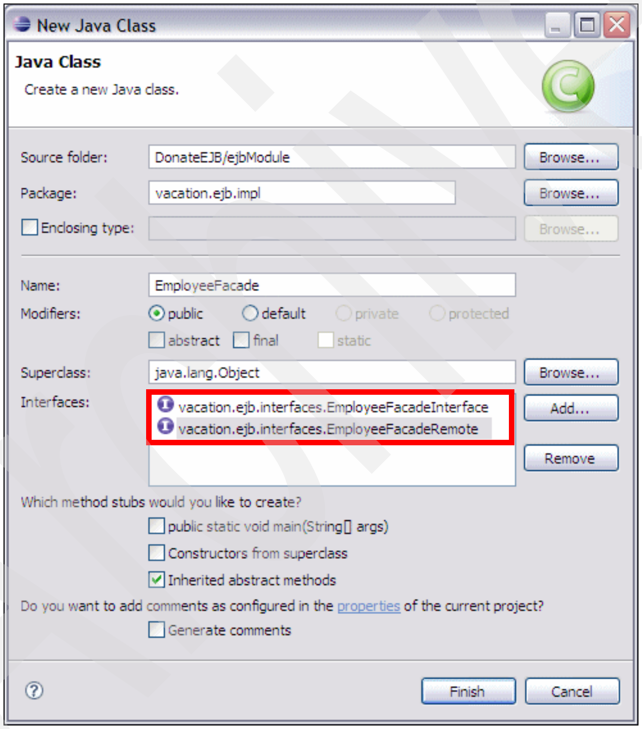
emp

Matching items:

EmployeeFacadeInterface - vacation.ejb.interfaces
EmployeeFacadeRemote

vacation.ejb.interfaces - DonateEJB/ejbModule

6. In the New Java Class/Java Class pop-up, the two selected interfaces are now listed. Click **Finish**.



New Java Class

Java Class

Create a new Java class.

Source folder: DonateEJB/ejbModule

Package: vacation.ejb.impl

□ Enclosing type:

Name: EmployeeFacade

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:
vacation.ejb.interfaces.EmployeeFacadeInterface
vacation.ejb.interfaces.EmployeeFacadeRemote

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments as configured in the properties of the current project?

□ Generate comments

7. A skeleton session bean is created (Example 6-5).

Example 6-5 Skeleton employee facade session bean

```
package vacation.ejb.impl;

import vacation.ejb.interfaces.EmployeeFacadeInterface;
import vacation.ejb.interfaces.EmployeeFacadeRemote;
import vacation.entities.Employee;

public class EmployeeFacade implements EmployeeFacadeInterface,
      EmployeeFacadeRemote {

   public Employee addVacationHours(Employee employee, int hours) {
      // TODO Auto-generated method stub
      return null;
   }

   public Employee deductVacationHours(Employee employee, int hours)
         throws Exception {
      // TODO Auto-generated method stub
      return null;
   }

   public Employee findEmployeeById(int id) throws Exception {
      // TODO Auto-generated method stub
      return null;
   }

}
```

Behind the scenes:

Note that the three methods skeletons were created from the three method signatures in the `EmployeeFacadeInteface` (`addVacationHours`, `deductVacationHours`, and `findEmployeeById`).

8. In the Java Editor view, add two annotations directly above the class signature:

```
@Stateless
@ApplicationException(rollback = true)
public class EmployeeFacade implements EmployeeFacadeInterface,
                                       EmployeeFacadeRemote {
   ......
```

Behind the scenes: The @Stateless annotation declares this Java class as a stateless session EJB, and the @ApplicationException annotation defines that a rollback is executed on application exceptions.

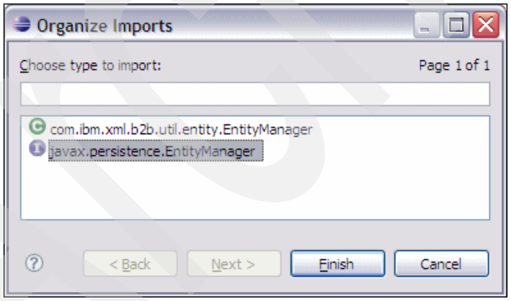
9. Now, add an `EntityManager` to this class by adding the following code to the class body, directly above the method signatures:

```
public class EmployeeFacade implements EmployeeFacadeInterface,
                                       EmployeeFacadeRemote {

   @PersistenceContext(unitName = "VacationUnit")
   private EntityManager em;

   public Employee addVacationHours(Employee employee, int hours) {
      .....
```

Behind the scenes: The `EntityManager` will coordinate accessing information associated with the `VacationUnit` persistence unit.

Recall that we defined the `VacationUnit` persistence unit in the `persistence.xml` file in 5.8, “Customize the persistence.xml files” on page 146.

10. Organize imports (**Ctrl+Shift+O**). If prompted to specify which `EntityManager` class to use, select **javax.persistence.EntityManager**.



11. In the Java Editor, delete the generated skeleton methods, then insert the **F11 Employee Facade** snippet from the Snippets view. Refer to the following **Behind the scenes** box for details about the inserted methods.

12. Save and close `EmployeeFacade.java`.

This concludes the creation of the employee facade session bean.

Behind the scenes:

The **F11 Employee Facade** snippet inserted completed method definitions for the three required methods (from `EmployeeFacadeInterface`) along with a common helper method:

- An `addHours` helper method is defined:

  ```
  public Employee addHours(Employee employee, int hours) {
     employee = em.find(Employee.class, employee.getEmployeeId());
     employee.setVacationHours(employee.getVacationHours() + hours);
     em.merge(employee);
     return employee;
  }
  ```

  This method retrieves an employee and adds the specified number of hours to the `vacationHours` property.

  The employee object that is passed into this method could have been created outside of this transaction context and thus might be stale, that is containing an older version of the employee record. Therefore, this method transactionally retrieves the current employee element and updates only the `vacationHours` field.

- The `addVacationHours` method calls the `addHours` method:

  ```
  public Employee addVacationHours(Employee employee, int hours) {
     return addHours((Employee) employee, hours);
  }
  ```

- The `deductVacationHours` method verifies that enough hours are available before subtracting the specified number of hours (using the `addHours` method):

  ```
  public Employee deductVacationHours(Employee employee, int hours)
                                   throws Exception {
     if (employee.getVacationHours() < hours)
        throw new Exception
           ("Employee does not have sufficient hours to deduct from");
     return addHours(employee, -hours);
  }
  ```

- The `findEmpoyeeById` method retrieves the employee and throws an exception if the employee is not found:

  ```
  public Employee findEmployeeById(int id) throws Exception {
     Employee employee = em.find(Employee.class, id);
     if (employee == null)
        throw new Exception("Employee with id " + id + " not found");
     return employee;
  }
  ```

6.4.4 Define the business interface for the fund facade

We define the business interface that is implemented by the fund facade session bean in the same way as for the employee facade (see 6.4.1, “Define the business interface for the employee facade” on page 187):

1. In the Project Explorer, expand the DonateEJB project, and create a package named **donate.ejb.interfaces**.
2. In the **donate.ejb.interfaces** package, create an interface named **FundFacadeInterface**.
3. In the Java Editor, move the cursor to the interface body and add the **F12 Fund Facade Interface** snippet from the Snippets view:

   ```
   public interface FundFacadeInterface {
      public abstract List<Fund> getAllFunds();
      public abstract Fund findFundByName(String name) throws Exception;
      public abstract Fund createNewFund(String name) throws Exception;
      public void removeFund(Fund fund);
      public abstract Fund addToBalance(Fund fund, int amount);
      public abstract Fund deductFromBalance(Fund fund, int amount)
                                                        throws Exception;
   }
   ```

4. Organize imports (**Ctrl+Shift+O**) to resolve these references:

   ```
   import java.util.List;
   import donate.entities.Fund;
   ```

5. Save and close FundFacadeInterface.java.

6.4.5 Define the remote interface for the fund facade

For access by remote clients, we also define a remote interface for the session bean facade in the same way as for the remote employee facade (see 6.4.2, “Define the remote interface for the employee facade” on page 189):

1. In the **donate.ejb.interfaces** package, create an interface named **FundFacadeRemote** that extends the FundFacadeInterface interface:
2. In the Java Editor, add a **@Remote** annotation above the method signature:

   ```
   @Remote
   public interface FundFacadeRemote extends FundFacadeInterface {
   }
   ```

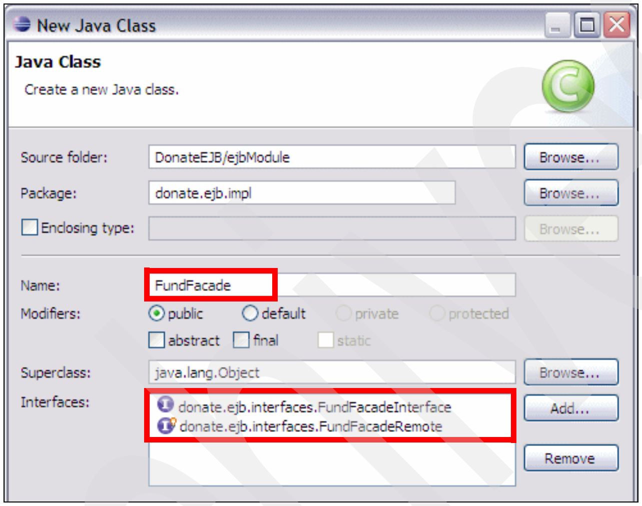
   The @Remote annotation makes this interface available to remote clients.

3. Organize imports (**Ctrl+Shift+O**), which adds javax.ejb.Remote.
4. Save and close FundFacadeRemote.java.

6.4.6 Create the session bean facade for the fund entity

In this section we create the session bean facade (`FundFacade`) for the `Fund` entity in the same way as for the `Employee` entity (see 6.4.3, “Create the session bean facade for the employee entity” on page 191):

1. In the Project Explorer, create a new package named **donate.ejb.impl**.
2. In the **donate.ejb.impl** package, create a new class named **FundFacade**, which implements the `FundFacadeInterface` and `FundFacadeRemote` interfaces.



New Java Class

Java Class

Create a new Java class.

Source folder: DonateEJB/ejbModule

Package: donate.ejb.impl

□ Enclosing type:

Name: FundFacade

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:
donate.ejb.interfaces.FundFacadeInterface
donate.ejb.interfaces.FundFacadeRemote

3. A skeleton session bean is created (Example 6-6).

Example 6-6 Skeleton fund facade session bean

```
package donate.ejb.impl;

import java.util.List;

import donate.ejb.interfaces.FundFacadeInterface;
import donate.ejb.interfaces.FundFacadeRemote;
import donate.entities.Fund;

public class FundFacade implements FundFacadeInterface, FundFacadeRemote {

   public Fund addToBalance(Fund fund, int amount) {
      // TODO Auto-generated method stub
      return null;
   }
   public Fund createNewFund(String name) throws Exception {
      // TODO Auto-generated method stub
      return null;
   }

   public Fund deductFromBalance(Fund fund, int amount) throws Exception {
      // TODO Auto-generated method stub
      return null;
   }

   public Fund findFundByName(String name) throws Exception {
      // TODO Auto-generated method stub
      return null;
   }

   public List<Fund> getAllFunds() {
      // TODO Auto-generated method stub
      return null;
   }

   public void removeFund(Fund fund) {
      // TODO Auto-generated method stub

   }

}
```

Behind the scenes:

Note that the six methods skeletons were created from the six method signatures in the `FundFacadeInterface` (`createNewFund`, `deductFromBalance`, `findFundByName`, `getAllFunds`, and `removeFund`).

4. In the Java Editor view, add two annotations directly above the class signature:

```
@Stateless
@ApplicationException(rollback = true)
public class FundFacade implements FundFacadeInterface, FundFacadeRemote
   ......
```

5. Now, add an `EntityManager` to this class by adding the following code to the class body, directly above the method signatures:

```
@PersistenceContext(unitName = "DonateUnit")
private EntityManager em;
```

6. Organize imports (**Ctrl+Shift+O**).

7. In the Java Editor, delete the generated skeleton methods, then insert the **F13 Fund Facade** snippet from the Snippets view. Refer to the following **Behind the scenes** box for details about the inserted methods.
8. Optionally add the annotation **@SuppressWarnings("unchecked")** before the `getAllFunds` method (this removes the warning).
9. Save and close `FundFacade.java`.

This concludes the creation of the fund facade session bean.

Behind the scenes:

The **F13 Fund Facade** snippet inserted completed method definitions for the six required methods (from `FundFacadeInterface`) along with a common helper method:

- The `getAllFunds` method uses the `getAllFunds` named query of the `Fund` entity to retrieve all the funds:

```
public List<Fund> getAllFunds() {
   return (List<Fund>)em.createNamedQuery("getAllFunds")
                                         .getResultList();
}
```

- The `findFundByName` method retrieves one fund and throws an exception if the fund is not found:

```
public Fund findFundByName(String name) throws Exception {
   Fund fund = em.find(Fund.class, name);
   if (fund == null)
      throw new Exception
                ("Fund not found with name \"" + name + "\"");
   return fund;
}
```

- The createNewFund method verifies that the fund does not exist already, then adds the new fund:

```
public Fund createNewFund(String name) throws Exception {
   Fund fund = null;
   try {
      fund = findFundByName(name);
   } catch(Exception e) { }
   if (fund != null)
      throw new Exception("Fund name already exists");
   fund = new Fund();
   fund.setName(name);
   fund.setBalance(0);
   em.persist(fund);
   return fund;
}
```

- The removeFund method removes a fund:

```
public void removeFund(Fund fund) {
   fund = em.find(Fund.class, fund.getName());
   em.remove(fund);
}
```

- The addToBalance method adds donated hours to the fund:

```
public Fund addToBalance(Fund fund, int amount) {
   fund = em.find(Fund.class, fund.getName());
   fund.setBalance(fund.getBalance() + amount);
   em.merge(fund);
   return fund;
}
```

- The deductFromBalance method subtracts hours from the fund if enough hours exist:

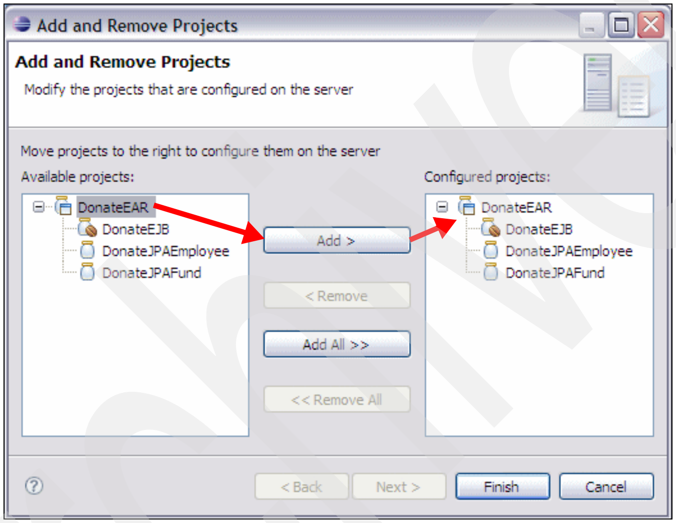
```
public Fund deductFromBalance(Fund fund, int amount)
                                       throws Exception {
   if (fund.getBalance() < amount)
      throw new Exception
         ("Fund does not have sufficient amount to deduct from");
   return addToBalance(fund, -amount);
}
```

6.5 Deploy the Donate enterprise application

As a preparation for testing with JUnit, we deploy the DonateEAR enterprise application to the server and verify that deployment succeeds:

1. In the Servers view of the Java EE perspective, ensure that the ExperienceJavaEE Server is started.
2. Right-click the **ExperienceJavaEE Server** and select **Add and Remove Projects**. Select the **DonateEAR** project on the left and click **Add**.

   Notice that the DonateEAR application contains the DonateEJB, DonateJPAEmployee, and DonateJPAFund projects.



Add and Remove Projects

Modify the projects that are configured on the server

Move projects to the right to configure them on the server

3. Click **Finish** to deploy the enterprise application to the server.
4. Right-click the **ExperienceJavaEE Server** and select **Publish**. Remember that we set automatic publishing to **Never publish automatically**.

Off course?

If publishing fails:

- Verify that the deployment location in `WASCE21/repository/ejee` contains a `DonateEAR` folder with a `DonateEAR-1.0.car` folder, with the three JAR files of the dependent projects.
- If these folders and files are missing, refer to 6.6, “Handle publishing failures” (next) for possible solutions.

6.6 Handle publishing failures

There are a number of situations where publishing can fail. Here we describe some that we encountered:

1. If you receive a pop-up that publishing fails, read the detailed messages. They can contain hints on what to look for.
2. Verify **geronimo-application.xml** (in `DonateEAR/EarContent/META-INF`). It must contain these tags only:

```
<app:application ...........>
    <dep:environment>
        <dep:moduleId>
            <dep:groupId>ejee</dep:groupId>
            <dep:artifactId>DonateEAR</dep:artifactId>
            <dep:version>1.0</dep:version>
            <dep:type>car</dep:type>
        </dep:moduleId>
        <dep:dependencies>
            <dep:dependency>
                <dep:groupId>console.dbpool</dep:groupId>
                <dep:artifactId>VacationPool</dep:artifactId>
            </dep:dependency>
            <dep:dependency>
                <dep:groupId>console.dbpool</dep:groupId>
                <dep:artifactId>DonatePool</dep:artifactId>
            </dep:dependency>
        </dep:dependencies>
    </dep:environment>
</app:application>
```

   - Additional tags can prevent the publishing from succeeding. Note that we will add more tags later, for security.
   - Verify the spelling of the group IDs and artifact names.

3. If the failure message indicates that you already have a duplicate deployed application, follow these steps:
   - Remove the **DonateEAR** project from the server using **Add and Remove Projects**.
   - Right-click the server and select **Publish**.
   - Delete the **ejee** folder from `WASCE21/repository`.
   - Add the **DonateEAR** project to the server using **Add and Remove Projects**.
   - Right-click the server and select **Publish**.

4. If you find a **default** folder with parts of the application under `WASCE21/repository`, verify that the group ID name is set to **ejee** in both DonateEAR (`geronimo-application.xml`) and DonateEJB (`openejb-jar.xml`), and that the artifact IDs are `DonateEAR` and `DonateEJB`.
5. The last resort is to restart the server:
   - Remove all applications, and publish.
   - Delete the `ejee` folder from `WASCE21/repository`.
   - Stop the server and restart the server.
   - Add the application to the server.
   - Publish the server.

6.7 Test the session facades with JUnit

In this section we test the session facade beans with JUnit. We use a similar approach as in 5.9, “Test the JPA entities with JUnit” on page 148, and we reuse the same testing project, DonateEJB_Tester.

6.7.1 Add the EJB dependency and the OpenEJB library

To test the session EJBs, we have to add a dependency to the DonateEJB project, and we have to add the OpenEJB library to the DonateEJB_Tester project:

1. Right-click the **DonateEJB_Tester** project and select **Properties**.
2. In the Properties pop-up, select **Java Build Path**,
3. Select the **Projects** tab and click **Add**. In the Required Project Selection pop-up, select **DonateEJB** and click **OK**.
4. Select the **Libraries** tab and click **Add Library**:
   - In the Add Library/Add Library pop-up, select **User Library** and click **Next**.
   - In the Add Library/User Library pop-up, select **OpenEJB** and click **Finish**.

Behind the scenes:

The OpenEJB user library was defined in 3.8, “Configure user libraries in Eclipse” on page 71.

5. Click **OK** to save and close the DonateEJB_Tester properties.

6.7.2 Create the employee facade tester

To test the employee facade, we create a JUnit test in the DonateEJB_Tester project. Make sure that you are in the Java perspective:

1. In the Project Explorer, right-click **donate.test** (in `DonateEJB_Tester/src`) and select **New** → **JUnit Test Case**. Note that in the Java EE perspective, you have to select **New** → **Other** and then from the pop-up select **Java** → **JUnit** → **JUnit Test Case**.
2. In the New JUnit Test Case/JUnit Test Case pop-up:
   - Select **New JUnit 4 test**.
   - Set the Package to **donate.test**.
   - Set the Name to **EmployeeFacadeTester**.
   - Click **Finish**.
3. Replace the entire contents of **EmployeeFacadeTester.java** with the **F21 EmployeeFacade Tester** snippet from the Snippets view.
4. Save and close `EmployeeFacadeTester.java`.

Behind the scenes:

- The @BeforeClass annotated `init` method finds the employee facade through the remote interface (`EmployeeFacadeRemote`):

```
@BeforeClass
@SuppressWarnings("unchecked"
public static void init() throws Exception {
   Hashtable props = new Hashtable();
   props.put("java.naming.factory.initial",
               "org.openejb.client.RemoteInitialContextFactory");
   ctx = new InitialContext(props);
   employeeFacade = (EmployeeFacadeRemote)
                        ctx.lookup("EmployeeFacadeRemote");
}
```

- Two helper methods (`getTestEmployee`) retrieve a test employee with an `employeeId` of 1.

```
public static Employee getTestEmployee
      (EmployeeFacadeInterface employeeFacade) throws Exception {
   return employeeFacade.findEmployeeById(1);
}
public Employee getTestEmployee() throws Exception {
   return getTestEmployee(employeeFacade);
}
```

- The @Test annotated `findEmployeeById` method retrieves the employee with `employeeId` 1 through the session facade:

```
public void findEmployeeById() throws Exception {
   Employee e = employeeFacade.findEmployeeById(1);
   assertNotNull(e);
}
```

- The @Test annotated `addHours` method donates 100 hours for the test employee, and verifies the result (`assertEquals`):

```
public void addHours() throws Exception {
   Employee e = getTestEmployee();
   int initialHours = e.getVacationHours();
   e = employeeFacade.addVacationHours(e, 100);
   assertEquals(e.getVacationHours(), initialHours+100);
}
```

- The @Test annotated `deductHours` method adds 1000 vacation hours, then removes 500 hours twice. Finally, the method tries to remove more hours than the employee has accumulated, and verifies that the request fails.

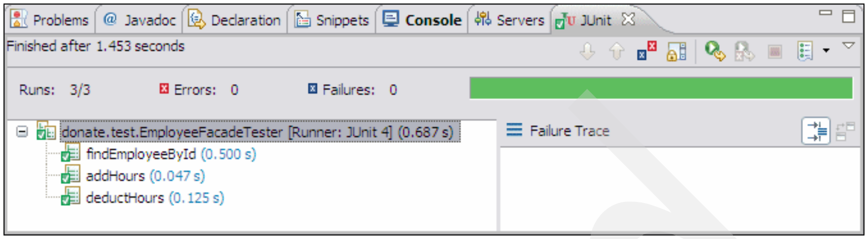
```
public void deductHours() throws Exception {
   Employee e = getTestEmployee();
   int initialHours = e.getVacationHours();
   e = employeeFacade.addVacationHours(e, 1000);
   try {
      e = employeeFacade.deductVacationHours(e, 500);
      e = employeeFacade.deductVacationHours(e, 500);
   } catch (Exception ex) {
      fail("Unexpected exception returned: " + ex.getMessage());
   }
   try {
      e = employeeFacade.deductVacationHours(e, 1+initialHours);
      fail("Deduction fails for negative hours");
   } catch(Exception ex) {
      System.out.println("deductHours: " + ex.getMessage());
   }
}
```

6.7.3 Run the employee facade tester

To run the test, follow the steps outlined in 5.9.4, “Run the JUnit test” on page 154:

1. In the Project Explorer, right-click **EmployeeFacadeTester.java** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.

2. Verify that all tests were successful in the JUnit view. Successful tests are marked by a green check mark.



```
Finished after 1.453 seconds
Runs:  3/3        Errors:  0        Failures:  0
donate.test.EmployeeFacadeTester [Runner: JUnit 4] (0.687 s)
    findEmployeeById (0.500 s)
    addHours (0.047 s)
    deductHours (0.125 s)
```

6.7.4 Create the fund facade tester

To test the fund facade we create a **FundFacadeTester** JUnit test case in the DonateEJB_Tester project:

1. Create the `FundFacadeTester` test case in the same way as for the employee.
2. Replace the skeleton code with the **F22 FundFacade Tester** snippet.

Behind the scenes:

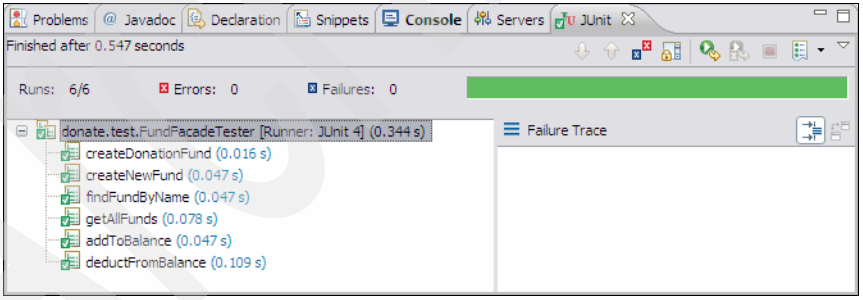
- The @BeforeClass annotated `init` method finds the fund facade through the remote interface (`FundFacadeRemote`).
- The @AfterClass annotated `tearDown` method removes the funds in the `cleanup` list that are created for testing.
- The two `createMockFund` helper methods create a random fund with the name `Fund #xxxxx` and add it to the `cleanup` list.
- The @Test annotated `createDonationFund` method creates the **DonationFund** with an initial balance of 1000 hours. The `DonationFund` will be used in subsequent chapters as the main fund for vacation donations.

- The @Test annotated `createNewFund` method creates a random mock fund, then tries to create the same fund again, and verifies that it fails.
- The @Test annotated `findFundByName` method creates a random mock fund, then retrieves the new fund, and verifies that the data matches.
- The @Test annotated `getAllFunds` method retrieves all the funds, then adds two random mock funds, and verifies that there are two more funds in total.
- The @Test annotated `addToBalance` method creates a random mock fund, then adds vacation hours, and verifies that the balance is correct.
- The @Test annotated `deductFromBalance` method creates a random mock fund, then adds and subtracts vacation hours and verifies the balance. Finally, the methods tries to deduct more hours than the balance, and verifies that the request fails.

6.7.5 Run the fund facade test

To run the test follow the same steps as for the employee facade.

1. In the Project Explorer, right-click **FundFacadeTester.java** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.
2. Verify that all tests were successful in the JUnit view. Successful tests are marked by a green check mark.



Finished after 0.547 seconds

Runs: 6/6 Errors: 0 Failures: 0

```
donate.test.FundFacadeTester [Runner: JUnit 4] (0.344 s)
    createDonationFund (0.016 s)
    createNewFund (0.047 s)
    findFundByName (0.047 s)
    getAllFunds (0.078 s)
    addToBalance (0.047 s)
    deductFromBalance (0.109 s)
```

3. Note that when you rerun the `FundFacadeTester`, the `createDonationFund` test fails because the `DonationFund` already exists in the database:

```
java.lang.Exception: Fund name already exists
   at donate.ejb.impl.FundFacade.createNewFund(FundFacade.java:38)
   ......
```

The other tests always succeed.

6.8 Verify the databases

After running the JUnit test cases for the entity facade session beans, the `FUND` table is created in the `Donate` database, and it contains the `DonationFund` and some random mock funds.

We verify these database updates using the DB Manager facility in the WASCE Console. Note that we could also have validated this using the Data Source Explorer in the WASCE IDE (as we did in 5.9.4, “Run the JUnit test” on page 154):

1. Open the WASCE Console (right-click the server and select **Launch WASCE Console**).
2. In the WASCE Console, select **Embedded DB** → **DB Manager**.
3. Click **Application** for the `Donate` database.
   - You can see one table, `DONATE.FUND`.
   - Click **View Contents** for the table (and then click **View Tables** to return). The `FUND` table contains the `DonationFund` (plus mock funds).
   - Click **View Databases** to return to the database list.
4. Click **Application** for the `Vacation` database.
   - You can see one table, `VACATION.EMPLOYEE`.
   - Click **View Contents** for the table (and then click **View Tables** to return). The `EMPLOYEE` table contains the three employees.

6.9 Explore!

Additional Explore resources about EJB 3.0 session beans are available in the IBM Redbooks publication, *WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0*, SG24-7611.

Create the Donate session bean for the business logic

In this chapter, we create a new session EJB that contains the business logic of the application. The session EJB transfers vacation hours from an employee record to a funds record (Figure 7-1).

7.1 Learn!

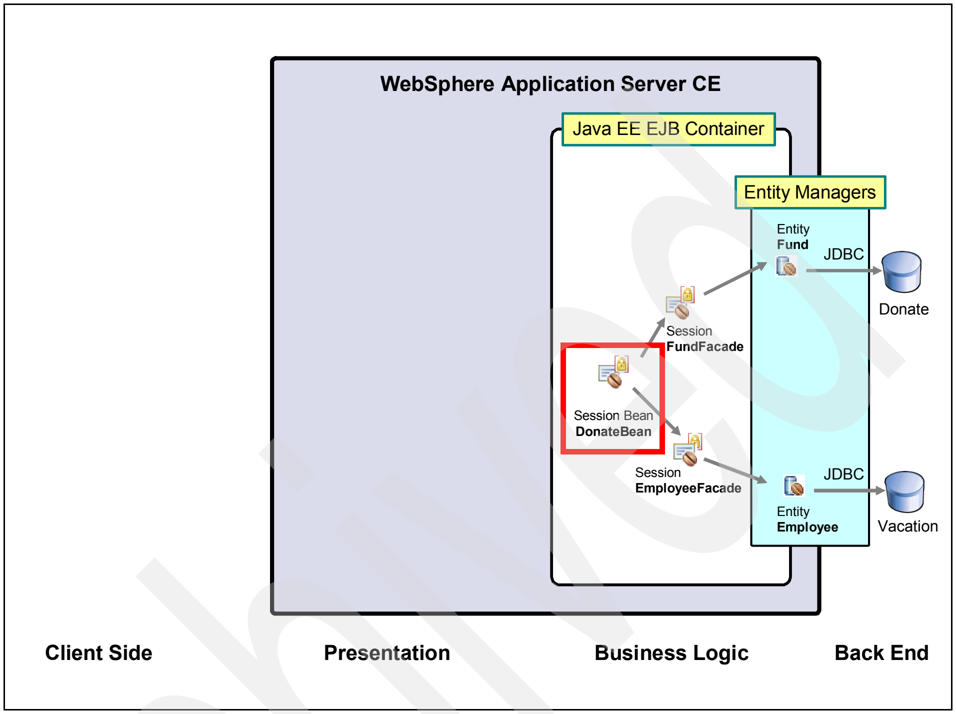


Figure 7-1 Donate application: Session bean for business logic

The Learn resources for this chapter are described in Chapter 6, “Create the entity facade session beans” on page 165.

7.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed these steps:

- Defined the two databases (`Vacation` and `Donate`)
- Defined the matching database pools (`VacationPool` and `DonatePool`)
- Created the JPA entities (`Employee` and `Fund`)
- Created the two session facade beans (`EmployeeFacade` and `FundFacade`)
- Created and run the JUnit test cases for JPA entities and the facades

7.3 Define the Donate session EJB

In this section, we define the Donate session EJB in the existing DonateEJB project. We follow the same approach as in 6.4, “Create the entity facade interfaces and session beans” on page 187.

7.3.1 Define the business interface for the Donate session bean

We define the business interface that is implemented by the Donate session bean in the same way as for the employee facade (see 6.4.1, “Define the business interface for the employee facade” on page 187):

1. In the Java EE perspective, Project Explorer, expand the DonateEJB project. We use the same package, **donate.ejb.interfaces**, as for the fund facade.
2. In the **donate.ejb.interfaces** package (in `DonateEJB/ejbModule`) create an interface named **DonateBeanInterface**.
3. In the Java Editor, move the cursor to the interface body and add the **F14 DonateBean Interface** snippet from the Snippets view:

```
public interface DonateBeanInterface {
   public void donateToFund(Employee employee, Fund fund, int hours)
                                                throws Exception;
}
```

4. Organize imports (**Ctrl+Shift+O**) to resolve:

```
import vacation.entities.Employee;
import donate.entities.Fund;
```

5. Save and close `DonateBeanInterface.java`.

7.3.2 Define the remote interface for the Donate session bean

For access by remote clients, we also define a remote interface for the session bean (see 6.4.2, “Define the remote interface for the employee facade” on page 189):

1. In the **donate.ejb.interfaces** package, create an interface named **DonateBeanRemote** that extends the `DonateBeanInterface` interface.
2. In the Java Editor, add a **@Remote** annotation above the method signature:

```
@Remote
public interface DonateBeanRemote extends DonateBeanInterface {
}
```

The @Remote annotation makes this interface available to remote clients.

3. In the Java Editor, move the cursor to the interface body and add the **F15 DonateBean Remote** snippet from the Snippets view:

```
public interface DonateBeanRemote extends DonateBeanInterface {
   public String donateToFund(int employeeId, String fundName,
                                              int hours);
}
```

Behind the scenes:

Note that the method unique to the remote interface uses simple argument types:

```
   public String donateToFund(int employeeId, String fundName, int hours)
```

Where the method in the base interface (valid for both local and remote access) takes complex types:

```
   public void donateToFund(Employee employee, Fund fund, int hours)
```

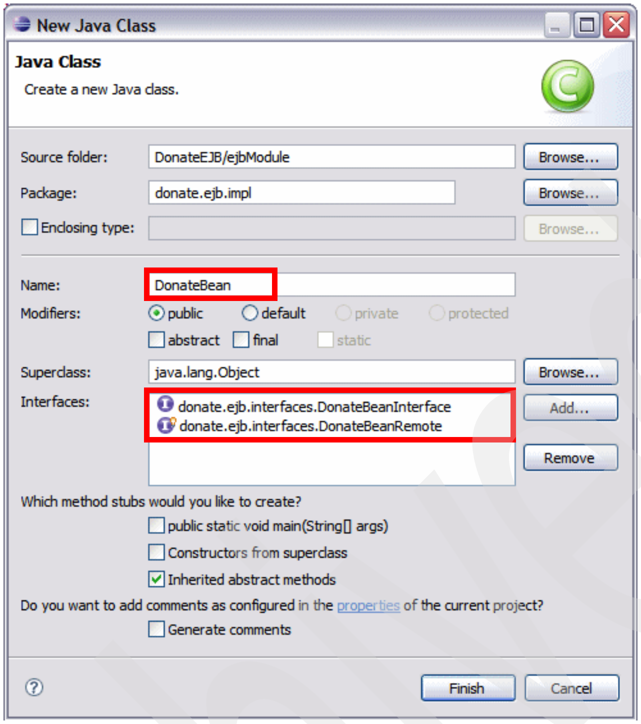
Remote callers (running outside of the Java EE container) such as Web clients, Web services, or message-driven beans are less likely to have access to the full object definition. Therefore, it is preferable to allow them to send the least amount of information required to perform the operation.

4. Organize imports (**Ctrl+Shift+O**) to resolve `javax.ejb.Remote`.
5. Save and close `DonateBeanRemote.java`.

7.3.3 Create the Donate session bean

In this section we create the Donate session bean in the same way as we created the session facades (see 6.4.3, “Create the session bean facade for the employee entity” on page 191):

1. In the **donate.ejb.impl** package (in `DonateEJB/ejbModule`), create a new class named **DonateBean**, which implements the `DonateBeanInterface` and `DonateBeanRemote` interfaces.



New Java Class

Java Class

Create a new Java class.

Source folder: DonateEJB/ejbModule

Package: donate.ejb.impl

□ Enclosing type:

Name: DonateBean

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:
donate.ejb.interfaces.DonateBeanInterface
donate.ejb.interfaces.DonateBeanRemote

Which method stubs would you like to create?

□ public static void main(String[] args)

□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments as configured in the properties of the current project?

□ Generate comments

2. A skeleton session bean is created (Example 7-1).

Example 7-1   Skeleton donate session bean

```
package donate.ejb.impl;
import vacation.entities.Employee;
import donate.ejb.interfaces.DonateBeanInterface;
import donate.ejb.interfaces.DonateBeanRemote;
import donate.entities.Fund;

public class DonateBean implements DonateBeanInterface, DonateBeanRemote {

   public void donateToFund(Employee employee, Fund fund, int hours)
         throws Exception {
      // TODO Auto-generated method stub

   }
   public String donateToFund(int employeeId, String fundName, int hours) {
      // TODO Auto-generated method stub
      return null;
   }
}
```

Behind the scenes:

Note that two different `donateToFund` method skeletons are created, one from the base interface and one from the remote interface.

3. In the Java Editor view, add two annotations directly above the class signature:

```
@Stateless
@ApplicationException(rollback = true)
public class DonateBean implements DonateBeanInterface, DonateBeanRemote
   ......
```

4. In the Java Editor, delete the generated skeleton methods, then insert the **F16 DonateBean** snippet from the Snippets view.
5. Organize imports (**Ctrl+Shift+O**).
6. Save and close `DonateBean.java`.

This concludes the creation of the `DonateBean` session bean.

Behind the scenes:

- The @EJB annotated interfaces define references to the business interfaces of the two facade session bean, `EmployeeFacade` and `FundFacade`, which are used to access the databases:

```
@EJB
private EmployeeFacadeInterface employeeFacade;

@EJB
private FundFacadeInterface fundFacade;
```

  The @EJB annotation uses EJB 3.0 injection to define the references to the other session beans, without using deployment descriptors.

- The first `donateToFund` method deducts the donated hours from the employee and adds the hours to the fund. The parameters are the `Employee` and `FUND` JPA entity objects. Exceptions are returned to the caller.

```
public void donateToFund(Employee employee, Fund fund, int hours)
      throws Exception {
   employeeFacade.deductVacationHours(employee, hours);
   fundFacade.addToBalance(fund, hours);
}
```

- The second `donateToFund` method performs the same function, but first has to retrieve the employee and fund objects using the given keys. This method handles any exceptions and returns an error message.

```
public String donateToFund(int employeeId, String fundName,
                                                 int hours) {
   String return_string = null;
   try {
      Employee employee = employeeFacade
                              .findEmployeeById(employeeId);
      Fund fund = fundFacade.findFundByName(fundName);
      donateToFund(employee, fund, hours);
      return_string = "Transfer successful";
   } catch (Exception e) {
      e.printStackTrace();
      return_string = "Error: " + e.getMessage();
   }
   return return_string;
}
```

7.4 Test the session bean with JUnit

In this section we test the session facade beans with JUnit. We use a similar approach as in 6.7, "Test the session facades with JUnit" on page 203, and we reuse the same testing project, DonateEJB_Tester.

7.4.1 Create the Donate bean test case

To test the `DonateBean` session bean, we create a JUnit test in the DonateEJB_Tester project. Make sure that you are in the Java perspective:

1. Expand the DonateEJB_Tester project, right-click on the **donate.test** package, and select **New** → **JUnit Test Case**. Note that in the Java EE perspective, you have to select **New** → **Other** and from the pop-up, select **Java** → **JUnit** → **JUnit Test Case**.
2. In the New JUnit Test Case/JUnit Test Case pop-up:
   - Select **New JUnit 4 test**.
   - Set the Package to **donate.test**.
   - Set the Name to **DonateBeanTester**.
   - Click **Finish**.
3. Replace the entire contents of **DonateBeanTester.java** with the **F23 DonateBean Tester** snippet from the Snippets view.
4. Save and close `DonateBeanTester.java`.

Behind the scenes:

- The @BeforeClass annotated `init` method finds the session bean and the entity facades through the remote interfaces (`DonateBeanRemote`, `FundFacadeRemote`, and `EmployeeFacadeRemote`):

```
public static void init() throws Exception {
   Hashtable props = new Hashtable();
   props.put("java.naming.factory.initial",
            "org.openejb.client.RemoteInitialContextFactory");
   ctx = new InitialContext(props);
   donateBean = (DonateBeanRemote)ctx.lookup("DonateBeanRemote");
   fundFacade = (FundFacadeRemote)ctx.lookup("FundFacadeRemote");
   employeeFacade = (EmployeeFacadeRemote)ctx.lookup
                                          ("EmployeeFacadeRemote");
}
```

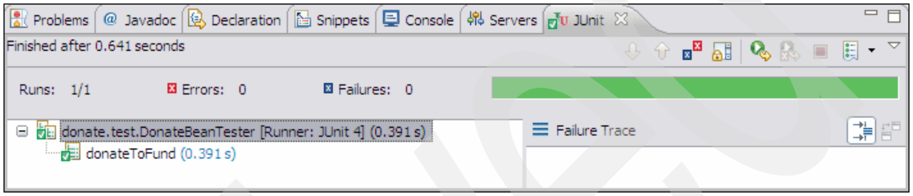
- The @Test annotated `donateToFund` method creates a mock fund, retrieves an employee through the session facade, adds 1000 hours to the employee, donates 1000 hours to the mock fund, and verifies that the employee and fund have been updated:

```
public void donateToFund() {
   try {
      Fund fund = FundFacadeTester.createMockFund(fundFacade, cleanup);
      Employee employee = EmployeeFacadeTester
                               .getTestEmployee(employeeFacade);
      employee = employeeFacade.addVacationHours(employee, 1000);
      donateBean.donateToFund(1, fund.getName(), 1000);
      assertEquals(
         fundFacade.findFundByName(fund.getName()).getBalance(),
         fund.getBalance()+1000
      );
      assertEquals(
         employeeFacade.findEmployeeById
                  (employee.getEmployeeId()).getVacationHours(),
         employee.getVacationHours()-1000
      );
   } catch(Exception e) {
      fail("Unexpected exception thrown: " + e.getMessage());
   }
}
```

7.4.2 Run the Donate bean tester

To run the test, follow the steps outlined in 5.9.4, “Run the JUnit test” on page 154:

1. In the Servers view of the Java EE perspective, publish the DonateEAR application. Right-click the **ExperienceJavaEE Server** and select **Publish**.
2. In the Project Explorer, right-click **DonateBeanTester.java** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.
3. Verify that all tests were successful in the JUnit view.



Finished after 0.641 seconds

Runs: 1/1 Errors: 0 Failures: 0

donate.test.DonateBeanTester [Runner: JUnit 4] (0.391 s)
donateToFund (0.391 s)

4. Note that every test run increases the balance in the `DonationFund` by one hour. You can verify that in the WASCE Console using the DB Manager.

7.5 Explore!

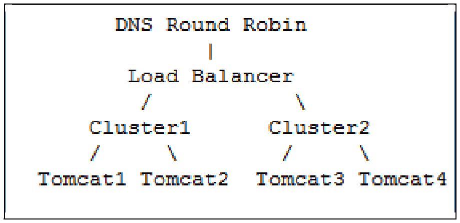
In this section, we provide information about advances in session state management.

7.5.1 Clustering and state management

The term clustering tends to be used loosely in the software industry and can be confusing. An application server cluster is a group of servers that transparently provide enterprise services, such as servlet and JavaServer Pages (JSP) support, as if it was a single server.

The servers, typically running on separate systems, exchange messages to synchronize data, allowing any individual node to process requests for a distributed application and to take over a user's session data when its node fails. Configuring multiple servers into a cluster is commonly called clustering.

WebSphere Application Server Community Edition implements HTTP session replication using memory-to-memory communication technology provided by Apache Tomcat. Every node transmits its session data to every node in the cluster. This algorithm is only efficient when the clusters are small. If the clusters grow too large, the overhead in storage utilization and network traffic becomes excessive. To avoid excessive overhead, consider dividing the nodes into several smaller clusters.



```
          DNS Round Robin
                 |
           Load Balancer
            /           \
        Cluster1       Cluster2
       /     \        /      \
  Tomcat1 Tomcat2  Tomcat3 Tomcat4
```

Image copied from Apache:

http://tomcat.apache.org/tomcat-5.5-doc/cluster-howto.html

HTTP session data is replicated among the nodes in the cluster using a multicast broadcast. All nodes in the cluster must be on the same physical subnet and multicast broadcast must be supported by that subnet.

7.5.2 Recommendations

It is important to also note that WASCE does not support EJB, JMS, or any other type of clustering/failover capabilities. This is acceptable for some applications. Otherwise, the IBM recommendation is to either use database backed persistent failover or data replication service (memory-to-memory) with our WebSphere Application Server Network Deployment product.

Notes:

The use of HTTP sessions can be a complicated subject that can lead into heated I/T discussions. The overhead and complexities of EJB 2.1 and earlier releases are regarded as major contributors to its widespread use. Many I/T organizations have steered away from using EJBs and instead opted for alternative, easier to build, stateful implementations. Many organizations have opted to implement conversational session state through HTTP session. Even though widely used, managing state through HTTP sessions is generally not recommended for building highly available scalable applications because all Java objects have to be serialized, which can cause a large overhead.

State should be implemented using frameworks and standards such as EJB. The new EJB 3.0 programming model uses annotations. The result makes state management radically simpler. Developers can now re-think implementing applications on the Java EE 5 programming model.

For more information, refer to *Best Practices for using HTTP sessions* in the IBM Information Center:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

Alternatively, an emerging and viable alternative for state management is to use grid or data fabric based solutions. The IBM solution in this area is called WebSphere eXtreme Scale (WXS). WXS provides application acceleration and transaction processing for objects obtained from back-end systems or stored within the WXS grid. WXS is a software layer that sits behind an application, providing the application its data from WXS itself or from back-end systems. The layer can be embedded in the same Java Virtual Machine as the application or can be a distributed layer where it makes sense for the application. WXS also provides a foundation for hosting XTP (extreme transaction processing) applications, transaction processing applications that must provide high performance, and availability and scalability requirements, ensuring fault tolerance and data integrity.

This description was taken from I*BM WebSphere eXtreme Scale - An Introduction and Overview* at:

ftp://ftp.software.ibm.com/software/webservers/WebSphere_eXtreme_Scale_wp.pdf

8

Create the Web front end

In this chapter, we present two Web applications, Basic and JSF (Figure 8-1).

In 8.3, “Basic Web application” on page 230, you gain hands-on experience in creating servlets and JSPs, which will interact with the `EmployeeFacade` and `DonateBean` session EJBs created in the previous chapters. The Basic Web application section is not required for any other chapters in this book, but completing this section illustrates how a Web application connects to EJB 3.0 session beans through simple injection.

In 8.4, “JSF Web application” on page 241, you gain hands-on experience in creating a JSF application, which will interact with the `EmployeeFacade` and `DonateBean` session EJBs created in the previous chapters.

8.1 Learn!

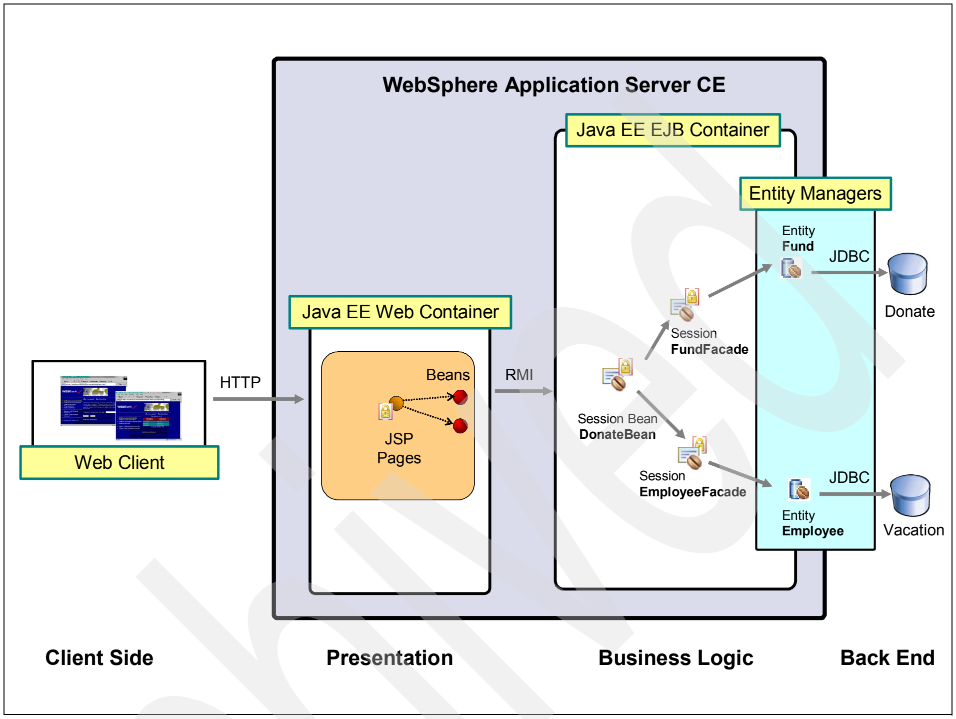


Figure 8-1 Donate application: Web front end

In general, an overall *Web front end* in Java EE usually consists of **Web components**, which are the actual artifacts that are invoked when a user types a Web address into a browser. In a Java EE environment. the addressable Web components can include these components:

- Static content, such as HTML pages or image files. Note that these are not included in the Java EE specifications, but they certainly provide a core portion of an overall Web front end, and they can be utilized side-by-side with the actual Java EE artifacts.
- Servlets, which provide 100% *dynamic* content.
- JavaServer Pages (JSPs), which provide for *partial dynamic* content. The JSP consists of both HTML statements and Java code.
- Session store, which maintains persistence information between the invocation of various Web components.
- Controller, which acts as an intermediary between the request and the actual final page.

8.1.1 What is a servlet?

A servlet is a Java EE specification which is used to extend the capabilities of servers that host applications accessed via a request-response programming model. In theory, servlets can respond to any type of request. The most common usage of servlets is to extend the applications hosted by Web servers. In this case, the underlying protocol with which the servlet interacts is HTTP.

A servlet receives a request, returns a response, or forwards processing to another servlet or a JSP for the response. In the request, data is passed to the servlet through parameters and attributes. Parameters are strings, whereas attributes can be any objects. The response is typically written in HTML format for rendering in a browser. A conversation can be maintained with a user by saving data (objects) as session data that is available in subsequent requests.

Servlet life cycle

In order to fully understand servlets, it is important to understand their life cycle. Like all Java EE container-based technologies, a servlet's life cycle is controlled by a container (in this case, a Web container). These are the life cycle steps of a servlet:

- **Start up**: If an instance of the servlet class does not exist, the Web container creates an instance of the servlet class. As a part of this step, the Web container also invokes the servlet's `init` method.
- **Handling the request**: An in-bound request (regardless of whether a `GET` or a `POST`) is handled by the servlet's `service` method. Typically the service method forwards the request to either the `doGet` or `doPost` method. A sample `doGet` method is shown here:

```
protected void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
   PrintWriter out = response.getWriter();
   out.println("<body>");
   out.println("<h1>Hello World</h1>");
   out.println("</body>");
}
```

- **Shut down**: When the servlet instance is ready to be removed (or recycled), the container invokes the destroy method of the servlet.

For more in-depth discussion on the servlet specification, refer to JSR 53: Servlet 2.3 and JSP 1.2 specification at:

http://jcp.org/en/jsr/detail?id=53

8.1.2 What is a JSP?

A JavaServer Page (JSP) is a text-based document that contains two types of text:

- Static template data, which can be expressed in any text-based format, such as HTML, SVG, WML, and XML
- JSP elements, which construct dynamic content. At runtime, JSPs are compiled into servlets.

8.1.3 Session store

The session store maintains information between the client (a browser) and the server:

- On the client side, the session ID is typically identified through a cookie (`JSESSIONID`) that is stored in the browser and transmitted in the HTTP header.
- On the server side, the session information is maintained in a persistence store (typically in memory, but also potentially in a relational database or in a publish/subscribe mechanism).

The first artifact invocation initializes the store and returns the session ID to the browser. On subsequent artifact invocations, this session ID is used to retrieve any stored information from the session store.

8.1.4 Controller

In basic Web development, there are initially two general Web development paradigms:

- **Model 1**: The Model 1 approach, the simpler of the two paradigms, is based on the notion that a request is made to a JSP or servlet. The JSP or servlet then handles all responsibilities for the request, including processing the request, validating data, handling the business logic, and generating a response.
- **Model 2**: The Model 2 approach employs the concept of separation of concerns. Specifically, requests from the client browser are passed to the controller, which is a servlet. The controller then decides which view (JSP) to pass the request to create the response.

In general, small applications can be written using the Model 1 approach. Larger enterprise-ready applications are better served using a Model 2 approach. The Model 2 approach is called *model-view-controller* (MVC).

8.1.5 Model-view-controller

Model-view-controller (MVC) is an architectural pattern. The pattern is grounded on the concept of isolating business logic layer from user interface layer (through a controller layer). The main benefit of employing such an approach is the application is not monolithic (and therefore easier to modify and maintain). It also enforces delegation of responsibilities between team members in a development team:

- **Model**: The model represents the information (the data) of the application and the business rules used to manipulate the data. The model is implemented as Java components, such as EJBs and JavaBeans, where EJBs provide the business logic and JavaBeans are the data transfer objects (DTO). The model encapsulates application state, responds to state queries, exposes application functionality, and notifies the view of changes.
- **View**: The view corresponds to elements of the user interface such as text and images, and it typically implemented as JSPs. The view renders the model, requests updates from the model, sends user input to the controller, and allows the controller to select the view.
- **Controller**: The controller handles all the requests, manages the communication to the model, and passes control to the view JSPs for the response to the client. The controller defined application behavior, maps user actions to model updates, and selects the view for the response.

Figure 8-2 shows a typical implementation of the model-view-controller architecture using servlets, JSPs, and EJBs.

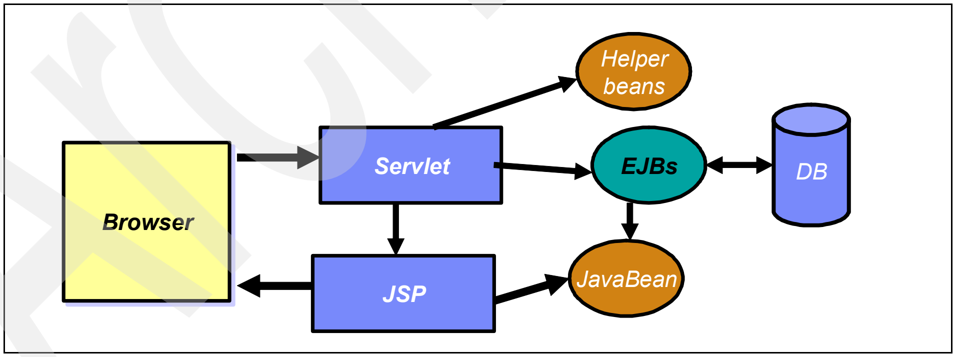


Figure 8-2 Model-view-controller architecture using servlets and JSPs

Note that out-of-the-box, the MVC programming paradigm is not enforced in servlet/JSP programming. MVC is an architectural approach: Java EE provides the required technical components, but the developer still must manually create quite a bit of code to coordinate the overall application.

The open source community developed a framework called Struts that defined the artifacts and required runtime support needed for a flexible MVC oriented Web application. This framework has been very popular and is shipped with many Java EE runtimes and Java EE IDEs. Note that Struts is not a standard or specification: It is simply an implementation that has been widely adopted. Other MVC frameworks are Spring MVC and Tapestry.

Java EE includes the JavaServer Faces (JSF) specification, which provides similar support.

8.1.6 What is a JSF?

In this section we describe the JavaServer Faces technology and develop a JSF-based Web application. JSF is part of the J2EE and Java EE specification and addresses many shortcomings of previous Web technologies.

JavaServer Faces technology, which is built on top of servlet/JSP technology, was created to simplify Web development, while enforcing best practices (such as promoting re-use and separation of concerns). Furthermore, JSF provides a component-based platform for Web development.

8.1.7 JSF application 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.

The focus of this section is to highlight the JSF application architecture shown in Figure 8-3:

- **Faces JSP pages**: Pages are built from JSF components, where each component is represented by a server-side class.
- **Faces servlet**: One servlet (`FacesServlet`) controls the execution flow.
- **Configuration file**: An XML file (`faces-config.xml`) that contains the navigation rules between the JSPs, validators, and managed beans.
- **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.
- **Managed beans**: JavaBeans defined in the configuration file to hold the data from JSF components. Managed beans represent the data model and are passed between business logic and user interface. JSF moves the data between managed beans and user interface components.

- **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.

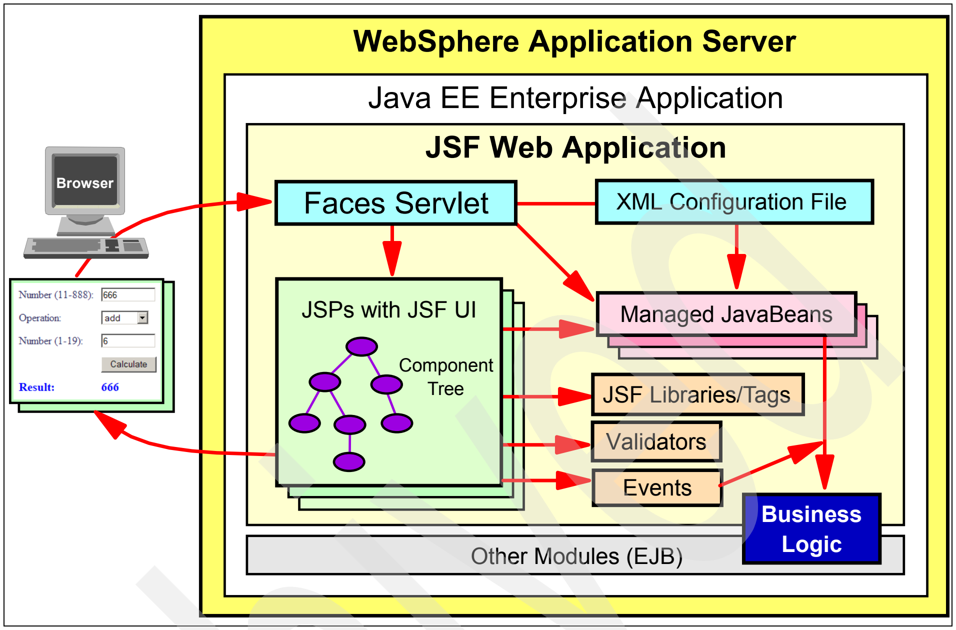
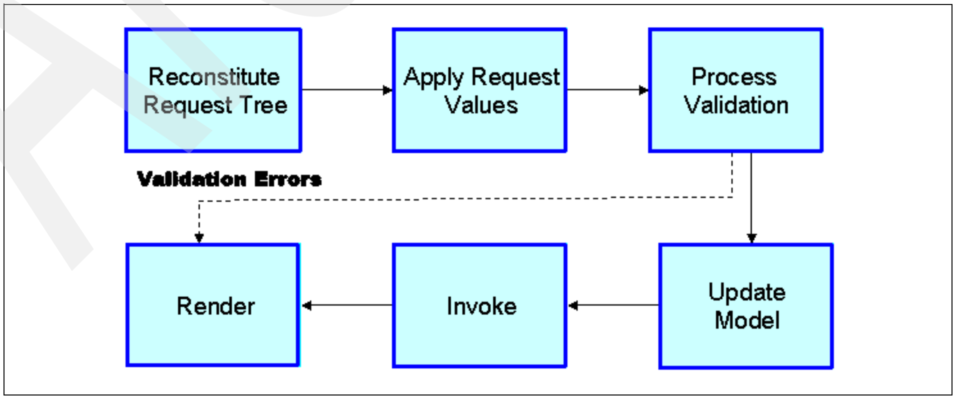


Figure 8-3 JSF application architecture

8.1.8 JSF life cycle

In an attempt to reduce complexity around servlets and provide a specification-backed standard framework, JSF was created to plug holes in servlets. Based on servlets, JSF embodies a more complex life cycle processing that addresses typical processing in Web applications.



- **Start up** (not shown): If an instance of the JSF servlet class does not exist, the Web container creates an instance of the JSF servlet class. As a part of this step, the Web container invokes the JSF servlet's `init` method.
- **Reconstitute Request Tree**: In this phase, an inbound request is handled by the Faces servlet. The controller extracts the view ID (from request), which is determined by the name of the JSP page. The Faces servlet uses the view ID to identify the component(s) for the current view. This phase of the life cycle presents three view instances: New, initial, and postback, with each being handled differently. New and initial view are related to creation and the initial loading the view. The postback view is primarily for restoring views in cases of failed user interface validations.
- **Apply Request Values**: In this phase, each component discovers its the current state by retrieving current values from either request parameters, cookies or headers. This phase also supports an immediate event handling property. This property of JSF is to handle events that do not require form validations.
- **Process Validation**: In this phase, each component will have its values validated against the application's validation rules. Should a validation fail, an error message is added to the Faces context, and the component is marked invalid. Thereafter, JSF advances to the render response phase, which will display the current view with the validation error messages. If there are no validation errors, JSF advances to the next phase.
- **Update Model**: In this phase, the values collected are applied to the managed beans. Only bean properties that are bound to a component's value will be updated.
- **Invoke**: In this phase, we can invoke the business logic. In addition, we can also specify the next logical view (or navigation) for a given sequence or number of possible sequences.
- **Render**: In the last phase, the results of the model are displayed.
- **Shut down** (not shown): When the JSF servlet instance is ready to be removed (or recycled), the container invokes the destroy method of the JSF servlet.

For more detailed accounting on JSF, refer to the JSR 127 specification at:

`http://jcp.org/en/jsr/detail?id=127`

8.1.9 Goals of JSF

The obvious next question now is, if JSF is built on top of servlets/JSPs, then what is the value-add of JSF (why do we need it)?

To better understand the motivation behind JSF, consider some of the gaps that exist in the Web application development space using other Java EE technologies.

In no particular order, here are some gaps in available technologies:

- There are no standards around building custom widgets for building a complex Web based user interface. For example, consider a graphical user interface that uses a tree hierarchy view. UI components would need to be designed and built to support this type of interface. In addition, the design of these components should be extensible and reusable in order to accommodate any future functionality and maintenance of the product.
- The same lack of standards in building custom widgets also makes it difficult for Tool providers to build IDEs that provide developers with an easier development experience and make Java EE technologies more accessible to a wide variety of development roles.
- There is also a lack of standards around support for best practices in Web development. Specifically, JSF enforces the MVC architecture. Since the inception of Java EE, we have experienced the pain-points of the Model 1 approach. Various Open Source frameworks have tried to address this gap by implementing the MVC architecture. Like all facets of Java EE, a standard based technology helps reduces complexity.

8.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed these steps:

- Defined the two databases and the matching database pools
- Created the JPA entities and the three session beans
- Created and run the JUnit test cases

8.3 Basic Web application

Optional!

Creating the complete JSF Web application as described in 8.4, "JSF Web application" on page 241 can be take quite a bit of time and might provide more information than needed if your primary focus is in back-end development (JPA, EJB, Web services, or messaging).

Therefore, you can optionally implement this Basic Web application with access to an employee record. This provides you with basic experience with Web development and how to connect a Web application to EJB 3.0 session beans.

However, if you complete this section (and do not complete the JSF Web application), you still must complete or jump start the JSF Web application before starting Chapter 10, "Implement core security" on page 285.

In this section, we create a basic Web application using servlets and JSP. For the purpose of this example, we use out-of-the-box servlet and JSP functionality. Therefore, we do not follow MVC architecture in this example, though at a minimum, we follow some separation of concerns.

Because we are developing two Web applications (one simple and one JSF-based), the approach we take for the simple Web application is to only implement the Find Employee use case.

To further illustrate and clarify our approach, the Web application structure is shown in Figure 8-4.

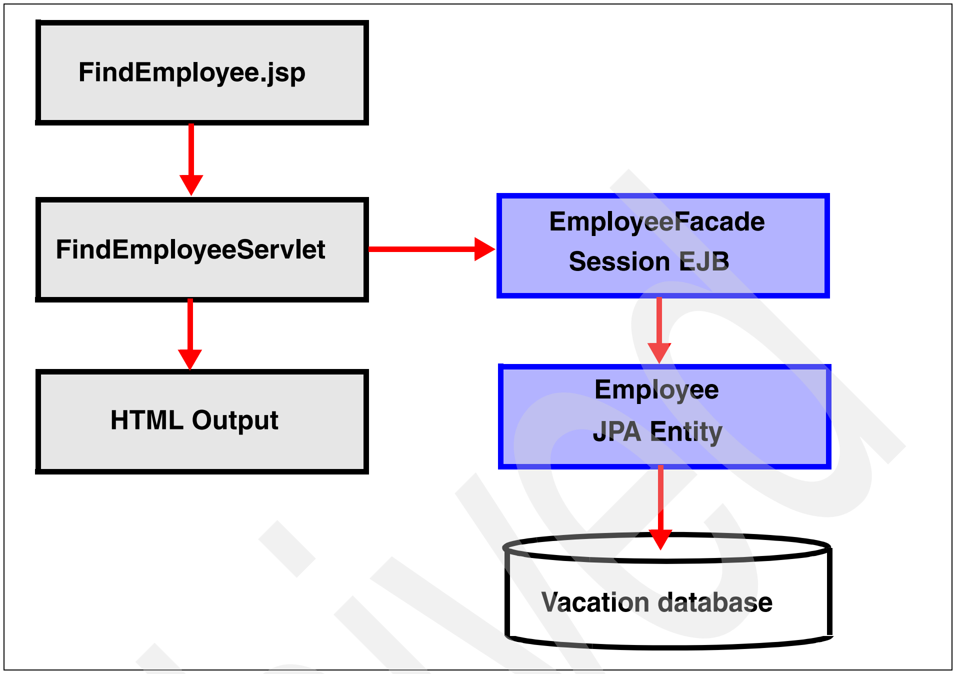
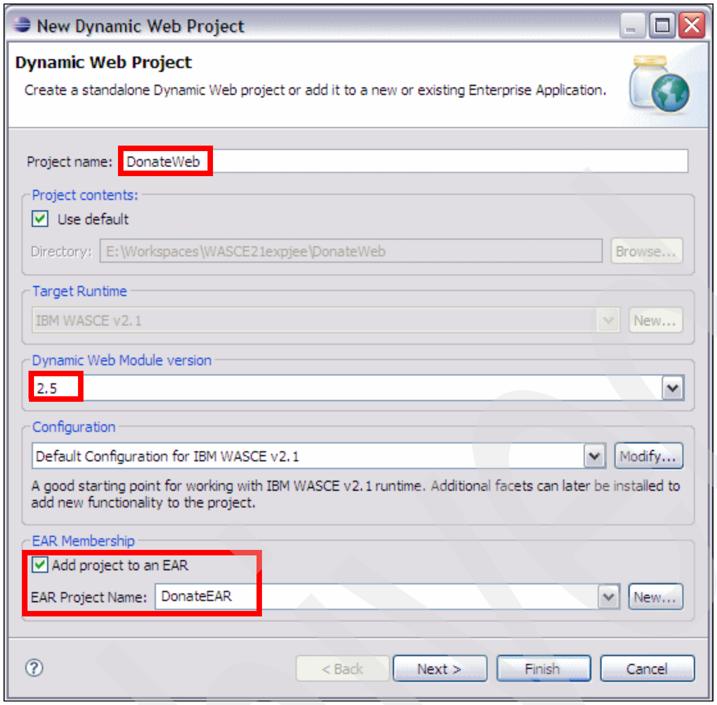


Figure 8-4 Web application structure

8.3.1 Create and configure the Web project

To begin this exercise, create a dynamic Web project called **DonateWeb**:

1. In the Java EE perspective, Project Explorer, select **File** → **New** → **Dynamic Web Project**.
2. In the Dynamic Web Project pop-up, specify the following:
   - Project Name: **DonateWeb**.
   - Target IBM Runtime: **IBM WASCE v2.1**.
   - Dynamic Web Module version: **2.5**.
   - Configurations: **Default Configurations for IBM WASCE v2.1**.
     - Click **Modify** to see the project facets. Verify that WASCE Deployment is selected.
     - Click **OK** or **Cancel**.
   - Select **Add project to an EAR**.
   - EAR Project Name: **DonateEAR**.
   - Click **Next**.



New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: DonateWeb

Project contents:
☑ Use default
Directory: E:\Workspaces\WASCE21expjee\DonateWeb

Target Runtime: IBM WASCE v2.1

Dynamic Web Module version: 2.5

Configuration: Default Configuration for IBM WASCE v2.1

A good starting point for working with IBM WASCE v2.1 runtime. Additional facets can later be installed to add new functionality to the project.

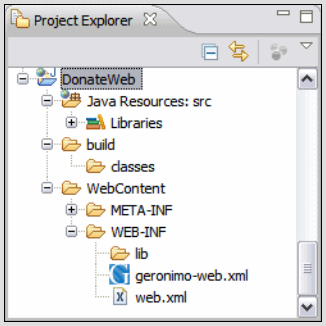
EAR Membership
☑ Add project to an EAR
EAR Project Name: DonateEAR

3. In the Web Module pop-up, select **Generate Deployment Descriptor**, and click **Next**.

4. In the Geronimo Deployment Plan pop-up, set the Group Id to **ejee** (to match the EAR), leave the Artifact Id empty, and click **Finish**.

Behind the scenes:

The structure of the Web project shows `Java Resources: src`, `build`, and `WebContent`.



- **Java Resources: src**: The `src` folder contains Java packages with the Java source files.
- **build**: compiled class files
- **WebContent**: The `WebContent` folder contains Web-related artifacts, such as HTML files, JSPs, images, and control files.
- **lib**: This folder holds JAR files required by other components.
- **geronimo-web.xml**: The deployment descriptor for the server contains the deployment information and the Web context root (`/DonateWeb`).
- **web.xml**: The `web.xml` file (or Web deployment descriptor) is the primary configuration file of the Web container. Items such as servlet, servlet filters, listeners, and tag library definitions are configured in this file.

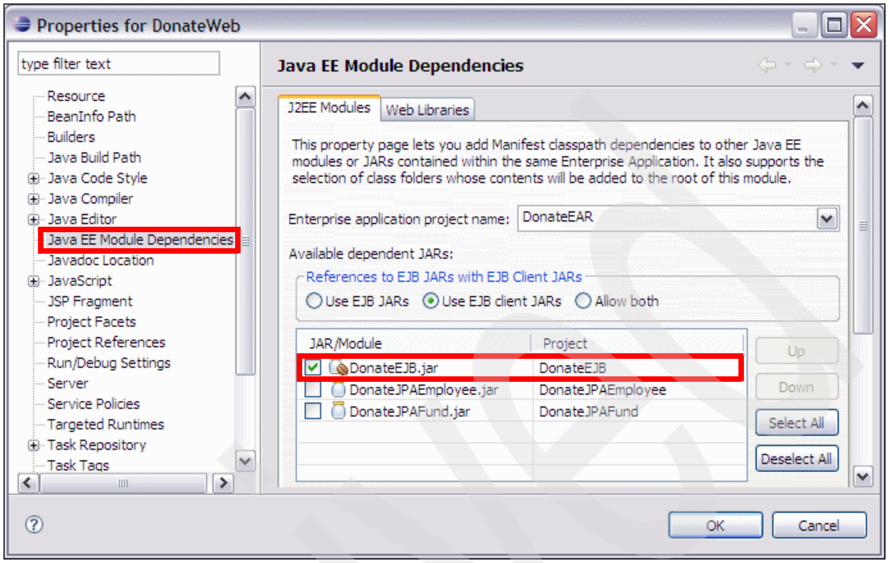
8.3.2 Add the dependency to the EJB project

In the Web project we access the EJB session beans. Therefore, we have to setup the dependency of the Web project to the EJB project.

By adding the DonateWeb project to the DonateEAR enterprise application, a dependency has been created in DonateEAR. Now we can set up further dependencies between projects contained in DonateEAR:

1. Right-click the **DonateWeb** project and select **Properties**.

2. In the properties pop-up, select **Java EE Module Dependencies**, and select the **DonateEJB.jar** file. Click **OK**.



Properties for DonateWeb

type filter text

- Resource
- BeanInfo Path
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Java EE Module Dependencies
- Javadoc Location
- JavaScript
- JSP Fragment
- Project Facets
- Project References
- Run/Debug Settings
- Server
- Service Policies
- Targeted Runtimes
- Task Repository
- Task Tags

Java EE Module Dependencies

J2EE Modules | Web Libraries

This property page lets you add Manifest classpath dependencies to other Java EE modules or JARs contained within the same Enterprise Application. It also supports the selection of class folders whose contents will be added to the root of this module.

Enterprise application project name: DonateEAR

Available dependent JARs:

References to EJB JARs with EJB Client JARs

○ Use EJB JARs ● Use EJB client JARs ○ Allow both

JAR/Module	Project
☑ DonateEJB.jar	DonateEJB
□ DonateJPAEmployee.jar	DonateJPAEmployee
□ DonateJPAFund.jar	DonateJPAFund

8.3.3 Create the JSP

Next, we create the page:

1. In the Project Explorer, right-click **WebContent** (in `DonateWeb`) and select **New** → **JSP**.
2. In the New JavaServer Page pop-up, type **FindEmployee.jsp** for the File name and click **Finish**.
3. `FindEmployee.jsp` opens in the editor (Example 8-1):
   – Insert the **F31 Web Front Employee JSP** snippet into the JSP code between the `<body></body>` tags.
   – Change the `<title>` to **Find Employee**.

Example 8-1 FindEmployee JSP

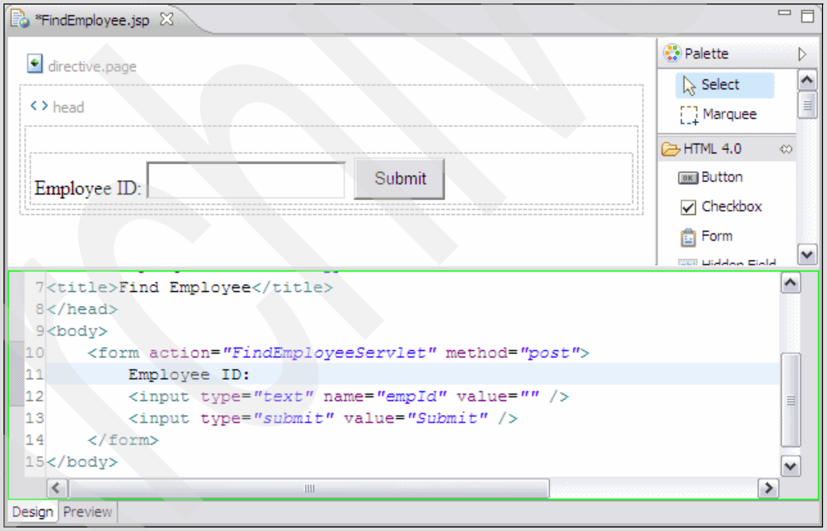
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Find Employee</title>
```

```
</head>
<body>
   <form action="FindEmployeeServlet" method="post">
      Employee ID:
      <input type="text" name="empId" value="" />

      <input type="submit" value="Submit" />
   </form>
</body>
</html>
```

4. Save and close `FindEmployee.jsp`.

Alternative editor:

Note that you can also open the JSP with a graphical editor. Right-click the `FindEmployee.jsp` and select **Open With** → **Web Page Editor**. The editor is split into two panes, showing the graphical representation at the top, and the source code at the bottom.



```
<title>Find Employee</title>
</head>
<body>
  <form action="FindEmployeeServlet" method="post">
    Employee ID:
    <input type="text" name="empId" value="" />
    <input type="submit" value="Submit" />
  </form>
</body>
```

You can edit in the graphical pane using the Palette (click the small arrow to show the Palette), and you can edit the source code. The two panes are kept synchronized.

We will make use of the graphical editor when we develop the JSF application.

Note that the last used editor becomes the default editor for double-click.

Behind the scenes:

Although it is labeled a JSP, `FindEmployee` contains only HTML tags. These tags defines an input filed that permits the user to enter the employee ID as a text string, and then click **Submit** to redirect the page (with the input value) to the `FindEmployeeServlet` using the post method.

```
<form action="FindEmployeeServlet" method="post">
   Employee ID:
   <input type="text" name="empId" value="" />
   <input type="submit" value="Submit" />
</form>
```

8.3.4 Create the servlet

Next we create the servlet, which provides the invocation of the EJB session bean:

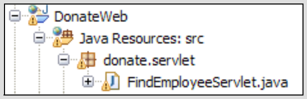
1. In the Project Explorer, right-click **DonateWeb** project, and select **New** → **Servlet**.
2. In the Create Servlet pop-up, provide the following data:
   - Java package: **donate.servlet**
   - Class name: **FindEmployeeServlet**
   - Superclass: Accept `javax.servlet.http.HttpServlet`
   - Click **Finish** (or go through the dialog pages using **Next** to see other options).

Behind the scenes:

In creating a servlet, a servlet class and a corresponding mapping in the `web.xml` are created.

A servlet (like any Java EE component) is a container-managed component. The container must be made aware of the Java EE component. In the case of a servlet, the Web container is made aware of the servlet, and knows what URL is used to invoke the servlet.

- The servlet class is created in the `donate.servlet` package.



```
DonateWeb
  Java Resources: src
    donate.servlet
      FindEmployeeServlet.java
```

- The corresponding entries are inserted into the Web deployment descriptor (`DonateWeb/WebContent/WEB-INF/web.xml`).

```
<servlet>
  <description></description>
  <display-name>FindEmployeeServlet</display-name>
  <servlet-name>FindEmployeeServlet</servlet-name>
  <servlet-class>donate.servlet.FindEmployeeServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>FindEmployeeServlet</servlet-name>
  <url-pattern>/FindEmployeeServlet</url-pattern>
</servlet-mapping>
```

3. Complete the `FindEmployeeServlet` code that invokes the session bean.

   – Delete all the generated method skeletons (default constructor, `doGet`, and `doPost`), leaving the base class definition and the `serialVersionUID`.

```
public class FindEmployeeServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

}
```

   – Insert the **F32 Web Front Employee Servlet** snippet before the closing brace.

   – Organize imports (**Ctrl+Shift+O**).

   – Save and close `FindEmployeeServlet.java` (Example 8-2).

Example 8-2 FindEmployeeServlet

```
public class FindEmployeeServlet extends javax.servlet.http.HttpServlet
                                    implements javax.servlet.Servlet {
   static final long serialVersionUID = 1L;

   @EJB
   EmployeeFacadeInterface employeeFacade;

   protected void doGet(HttpServletRequest request, HttpServletResponse
                        response) throws ServletException, IOException {
      doPost(request, response);
   }

   protected void doPost(HttpServletRequest request, HttpServletResponse
                        response) throws ServletException, IOException {

      String empId = (String) request.getParameter("empId");
      Employee obj = null;
      try {
         obj = employeeFacade.findEmployeeById(Long.parseLong(empId));
      } catch(Exception e) {
         e.printStackTrace();
      }

      if (obj == null) {
         response.getWriter().println("null object");
      } else {
         response.getWriter().println(formatEmployeeInfo(obj));
      }
   }

   public String formatEmployeeInfo(Employee obj) {
      StringBuffer sb = new StringBuffer();
      sb.append("<html><body>");
      sb.append("employee id = " + obj.getEmployeeId() + " <br><b>");
      sb.append("employee first name = " + obj.getFirstName() + "<br>");
      sb.append("employee middle name = " + obj.getMiddleName() + "<br>");
      sb.append("employee last name = " + obj.getLastName() + "<br>");
      sb.append("Salary = " + obj.getSalary() + " </b></body></html>");
      return sb.toString();
   }
}
```

Behind the scenes:

Notice that the code in the servlet is implemented in the **doPost** method. There is a difference in method submissions of HTML forms. The two types are GET and POST:

- **GET**: Request parameters are appended to the URL query string, and the end user can save such a URL with all the parameters.
- **POST**: Request parameters are written in the server input stream, and are not visible to the end user.

The key elements in this servlet are as follows:

- The @EJB annotation provides an instance of the employee facade session bean's business interface (EmployeeFacadeInterface). This is the injection technique of EJB 3.0, much simpler than using JNDI.

```
@EJB
EmployeeFacadeInterface employeeFacade;
```

- The application code required is the same for both a doPost and a doGet, so we redirect all doGet requests to the doPost method.

```
protected void doGet(HttpServletRequest request, HttpServletResponse
                    response) throws ServletException, IOException {
   doPost(request, response);
}
```

- The doPost method uses the input parameter to look up the employee record from the EmployeeFacade session facade, and then calls the formatEmployeeInfo method (not shown here) to format the result.

```
protected void doPost(HttpServletRequest request, HttpServletResponse
                    response) throws ServletException, IOException {

   String empId = (String) request.getParameter("empId");
   Employee employee = null;
   try {
      employee = employeeFacade.findEmployeeById
                                       (Integer.parseInt(empId));
   } catch(Exception e) {
      e.printStackTrace();
   }
   if (employee == null) {
      response.getWriter().println("Employee not found");
   } else {
      response.getWriter().println(formatEmployeeInfo(employee));
   }
}
```

8.3.5 Test the Web application

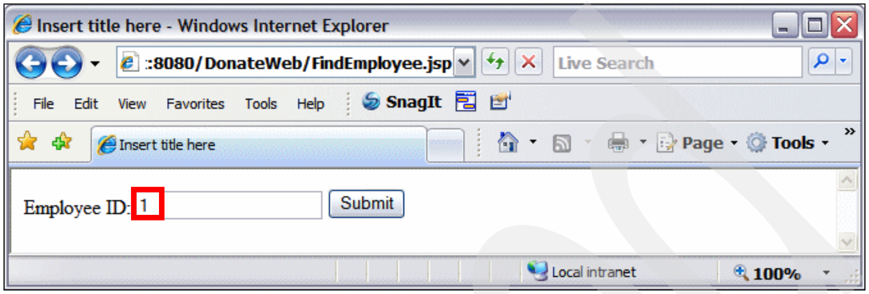
To test the Web application, we deploy it to the server and run the servlet:

1. Make sure that the **ExperienceJavaEE Server** is started:
   - If the server is not running, select the server in the Servers view and click the **Start** icon ▶ or right-click the server and select **Start**.
   - If the server instance starts successfully, you will find a message **Server started** in the Console view.
2. If the DonateEAR application is already deployed to the server, select the **ExperienceJavaEE Server** and select **Publish**:
   - If the DonateEAR application is not deployed to the server, right-click the server, select **Add and Remove Projects**, and add the **DonateEAR** to the configured projects. Click **Finish**, and then publish the project as described previously.

3. Test the code. Open an instance of Internet Explorer and submit this URL (you can also use Firefox):

```
http://localhost:8080/DonateWeb/FindEmployee.jsp
```

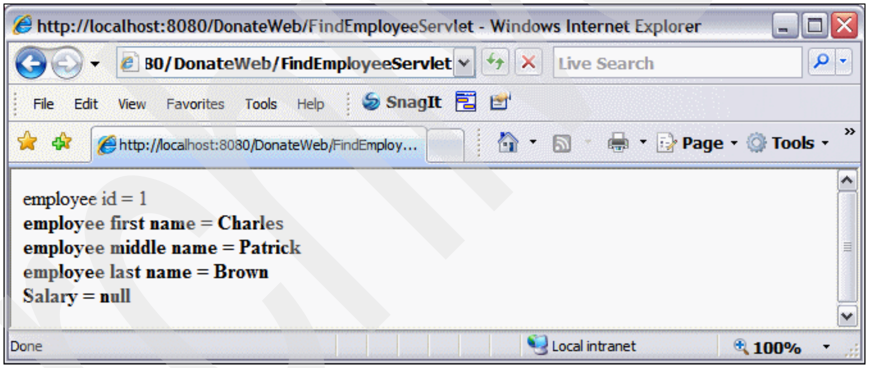
4. The initial Web page is displayed.


Insert title here - Windows Internet Explorer
:8080/DonateWeb/FindEmployee.jsp
Live Search
File Edit View Favorites Tools Help
SnagIt
Insert title here
Page
Tools
Local intranet
100%

Employee ID: 1 Submit

5. In the Employee ID field, type the number **1** and click **Submit**.

6. The employee details page is displayed.


http://localhost:8080/DonateWeb/FindEmployeeServlet - Windows Internet Explorer
80/DonateWeb/FindEmployeeServlet
Live Search
File Edit View Favorites Tools Help
SnagIt
http://localhost:8080/DonateWeb/FindEmploy...
Page
Tools
Done
Local intranet
100%

```
employee id = 1
employee first name = Charles
employee middle name = Patrick
employee last name = Brown
Salary = null
```

7. You can also try employee ID 2 or 3. Any other ID gets an error message.

Behind the scenes:

If you are used to Rational Application Developer or other IBM integrated IDEs, you would right-click the JSP in the Project Explorer and select **Run As** → **Run on Server**. Unfortunately, this action does not work in the WASCE IDE.

8.3.6 Recap

In this section, we have gained practical experience in developing a simple Web-based application using JSP and servlet technology. Because we were using out-of-the-box functionality from JSP and servlet technology, we only partially followed the Model 2 approach. We did not have a clear separation of concerns between the view and the model layers.

As the Java EE specifications evolved, the adoption of full Model 2 (MVC) paradigm (in addition to Web component-based programming) was eventually included in the specification. These features were included in the JavaServer Faces (JSF) specification. For more information on JSF and its capabilities, please read the rest of this chapter.

8.4 JSF Web application

In the remainder of this chapter we develop a JSF Web application with multiple pages.

We are developing an application that facilitates the donation of vacation days to employees who are in need of extra vacation days. The following are the use cases that we implement for this application:

- **Employee search**: A page where employee id is provided.
- **Employee detail**: A page where employee details are displayed.
- **Vacation donation**: A page where vacation donation can be specified

Taking a further step, we build a JSF-based front end to the Donate application by implementing the following pages:

- **Find an employee**: This page will provide the search functionality for an employee by employee ID.
- **Employee details**: This page displays the employee details. The page also contains a form where vacation hours donation can be specified and submitted.
- **Donation confirmation**: This page confirms the vacation hours donation specified in the employee details page.

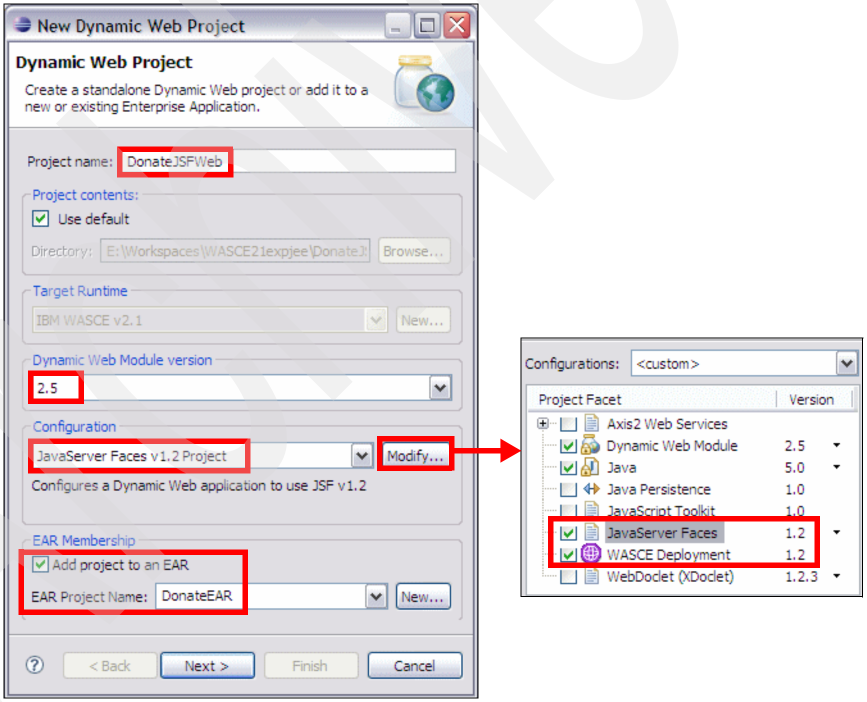
8.4.1 Create the JSF Web project

We create a Dynamic Web project named **DonateJSFWeb** for JSF development, and add the Web project to the DonateEAR enterprise application.

1. In the Java EE perspective, Project Explorer, select **File** → **New** → **Dynamic Web Project**.
2. In the Dynamic Web Project pop-up, specify the following parameters:
   - Project Name: **DonateJSFWeb**
   - Target IBM Runtime: **IBM WASCE v2.1**
   - Dynamic Web Module version: **2.5**
   - Configuration: **JavaServer Faces v1.2 Project**:

     Click **Modify**, and in the resulting Project Facets pop-up:

     - Select **JavaServer Faces** (preselected at Version 1.2).
     - Select **WASCE Deployment**.
     - Click **OK**. The configuration changes to `<custom>`.
   - EAR Membership: Select **Add project to an EAR**, and set the EAR Project Name to **DonateEAR**.



New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: DonateJSFWeb

Project contents:

☑ Use default

Directory: E:\Workspaces\WASCE21expjee\DonateJ

Target Runtime

IBM WASCE v2.1

Dynamic Web Module version

2.5

Configuration

JavaServer Faces v1.2 Project

Configures a Dynamic Web application to use JSF v1.2

EAR Membership

☑ Add project to an EAR

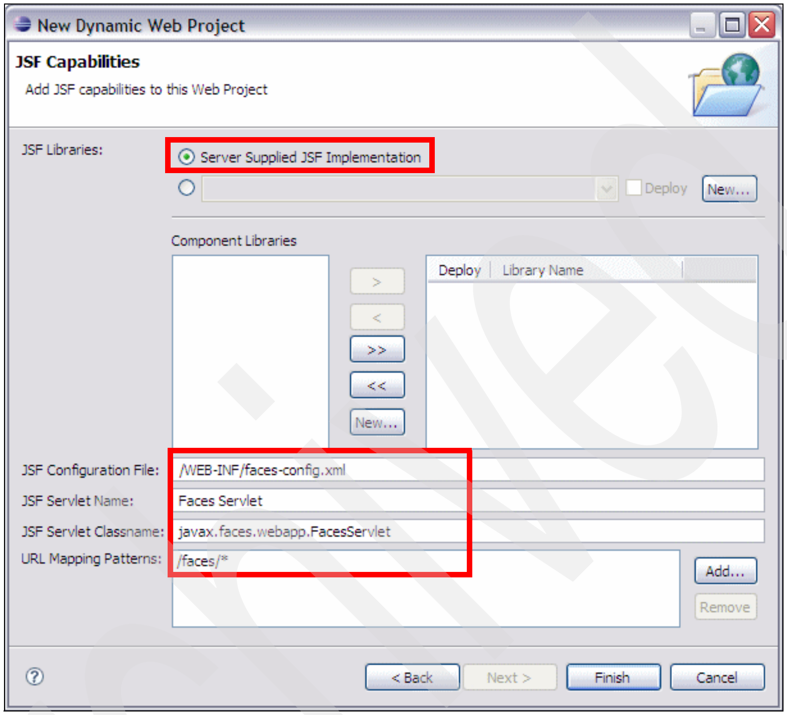
EAR Project Name: DonateEAR

Configurations: <custom>

	Project Facet	Version
□	Axis2 Web Services	
☑	Dynamic Web Module	2.5
☑	Java	5.0
□	Java Persistence	1.0
□	JavaScript Toolkit	1.0
☑	JavaServer Faces	1.2
☑	WASCE Deployment	1.2
□	WebDoclet (XDoclet)	1.2.3

   - Click **Next**.
3. In the Web Module pop-up, select **Generate Deployment Descriptor**, and click **Next**.

4. In the Geronimo Deployment Plan pop-up, set the Group Id to **ejee** (to match the EAR), leave the Artifact Id empty, and click **Next**.
5. In the JSF Capabilities pop-up, select **Server Supplied JSF Implementation**, and accept the defaults.



New Dynamic Web Project

JSF Capabilities

Add JSF capabilities to this Web Project

JSF Libraries: ● Server Supplied JSF Implementation

JSF Configuration File: /WEB-INF/faces-config.xml

JSF Servlet Name: Faces Servlet

JSF Servlet Classname: javax.faces.webapp.FacesServlet

URL Mapping Patterns: /faces/*

6. Click **Finish**.

Behind the scenes:

The layout of the JSF Web project is the same as for the simple Web project, with these additions:

- A Faces configuration file (`WEB-INF/faces-config.xml`) has been created. The Faces configuration file contains information about navigation between pages, validators, and managed beans. You can open the file, but it is basically empty now.
- A Faces servlet is predefined. The Faces servlet handles all the requests. You can see the servlet in the Web deployment descriptor (`web.xml`).
- The URL mapping of **/faces/*** directs all such requests to the Faces servlet. JSF request are of the form `http://<host>:8080/faces/xxxxx.jsp`.

8.4.2 Create the JSPs

The approach that we take to develop the JSF front end is page by page, from the finding an employee page (`FindEmployee.jsp`), to the employee details page (`EmployeeDetails.jsp`), to the donation confirmation page (`DonationConfirmation.jsp`).

First we create the empty JSPs:

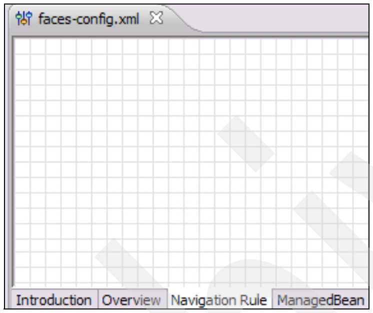
1. Right-click **WebContent** (in `DonateJSFWeb`), and select **New** → **JSP**.
2. In the JavaServer Page pop-up, type **FindEmployee.jsp** as File name, and click **Next**.
3. Select **Use JSP Template**, and select **New JavaServer Faces (JSF) Page (html)**, then click **Finish**.
4. In the editor that opens, notice the two JSF tag libraries (`<%@ taglib>`) that are added to the JSP source code. Close the editor.
5. Repeat this sequence for the other two JSPs (`EmployeeDetails.jsp` and `DonationConfirmation.jsp`).

8.4.3 Define the navigation

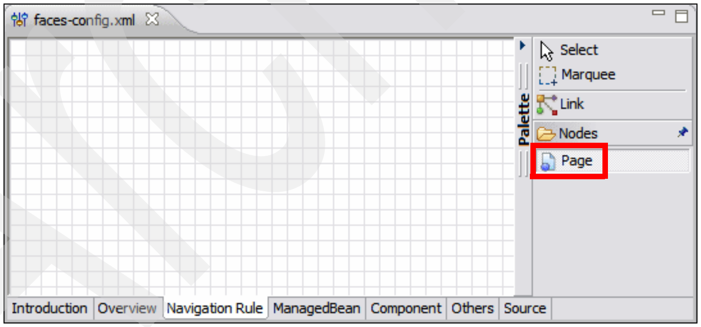
In servlet programming, we do not have any provisions on how to specify navigation throughout an application. Typically, the navigation through an application is implemented programmatically within the servlet.

In JSF, we define navigation declaratively through an external configuration file. The advantage of declarative programming is that page navigation is externalized from code (easily accessible), which eases development and maintenance:

1. Open **faces-config.xml** (in `DonateJSFWeb/WebContent/WEB-INF`), and select the **Navigation Rule** tab.



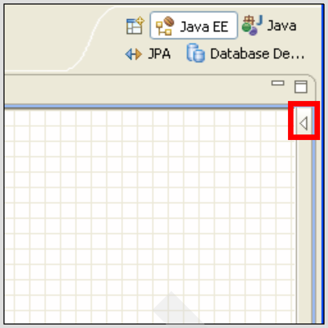
2. On the far right-hand side, click **Palette** and select the **Page** icon. We begin to construct the page flow of the DonateJSFWeb application.



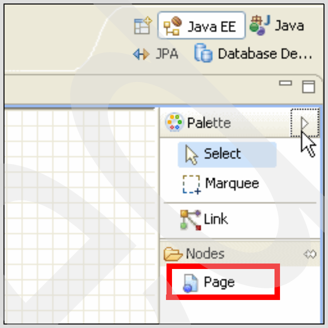
Off course?

If you do not see the Palette, it is probably minimized. Look for the left facing arrow on the upper right side of the faces-config.xml editor and click the arrow to open the palette.

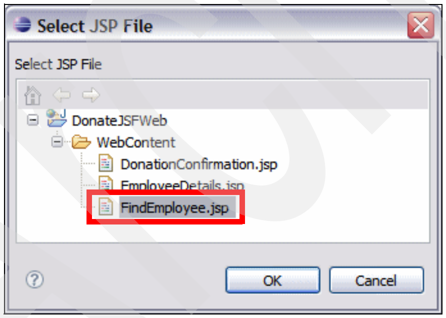
- Hidden palette:



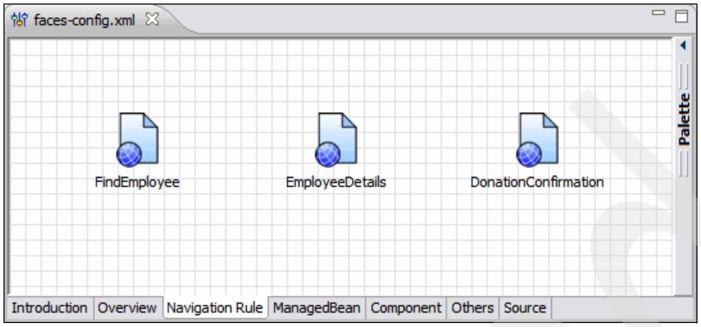
- Visible palette:



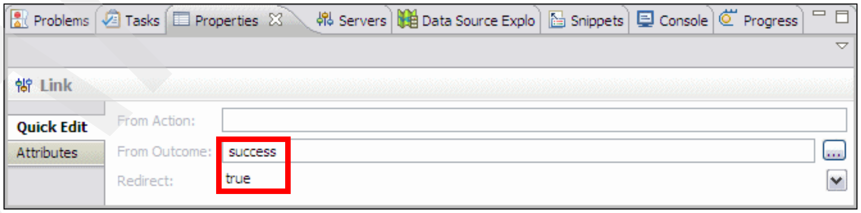
3. Select the **Page** icon and click into the empty space. The Select JSP File pop-up open:
   - Expand **DonateJSFWeb** → **WebContent** and select **FindEmployee.jsp**.
   - Click **OK** and the FindEmployee page is added to the diagram.



4. Repeat these steps to add the other two pages (`EmployeeDetails.jsp` and `DonationConfirmation.jsp`).



5. Create the navigation links between the pages:
   – In the **Palette**, select the **Link** icon.
   – Click on **FindEmployee** and drag the link to **Employee Details.** A link line is drawn between the pages.
   – Repeat this and draw a link from **EmployeeDetails** to **DonationConfirmation**.
6. Next we label each navigation link:
   – In the **Palette,** select the **Select** icon.
   – Right-click the navigational link from `FindEmployee` to `EmployeeDetails`.
   – In the lower right pane, select the **Properties** view. If the Properties view is not available, select **Show View** → **Properties** to open the view).
   – In the Properties view, **Quick Edit** tab:
     - Set the From Outcome to **success**.
     - Set Redirect to **true** using the pull-down menu.



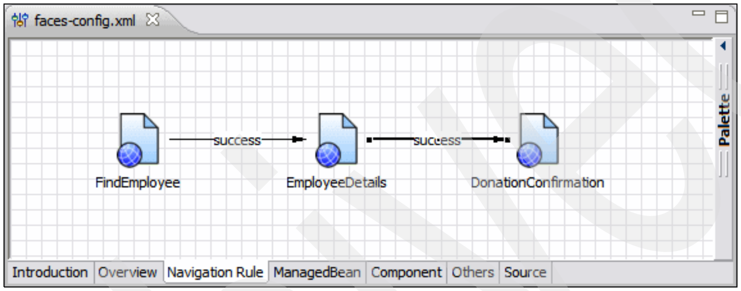
From Action:
From Outcome: success
Redirect: true

   – Repeat this for the link from `EmployeeDetails` to `DonationConfirmation` (using **success** for the From Outcome and leave Redirect as **false**). The Navigation Rule page shows the navigation names.

Behind the scenes:

Refer to the Behind the scenes box (on the next page) and to 10.5.4, “Restrict access to Web pages through declarative security” on page 300, for a description of why we set redirect to true for the first link, but to false for the second link.

7. View the updated content in both the Navigation Rule tab (graphical view) and in the Source tab (Example 8-3).
8. Save and close `faces-config.xml` when done.



Example 8-3 Navigation rules in faces-config.xml

```
<navigation-rule>
   <display-name>FindEmployee</display-name>
   <from-view-id>/FindEmployee.jsp</from-view-id>
   <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/EmployeeDetails.jsp</to-view-id>
      <redirect />
   </navigation-case>
</navigation-rule>
<navigation-rule>
   <display-name>EmployeeDetails</display-name>
   <from-view-id>/EmployeeDetails.jsp</from-view-id>
   <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/DonationConfirmation.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

Behind the scenes:

- The XML element **<from-view-id>** represents the current page. The element **<from-outcome>** represents the indicator of the outcome of the invocation of the business logic. The element **<to-view-id>** represents the page to forward or redirect to based on the outcome of the invocation of the business logic.

  In our example this means: If the `FindEmployee.jsp` returns the outcome **success**, then the `EmployeeDetails.jsp` is invoked.

- **Request redirection** is used instead of **request forwarding** to ensure that Java EE role-based security can be applied to unique pages.

  ```
  <to-view-id>
  /EmployeeDetails.jsp</to-view-id>
  <redirect/>
  ```

  Java EE role-based security for Web artifacts are controlled at the URI level:

  – Request forwarding controls the page navigation on the server side, so the URI submitted from the client side does not change, and would therefore restrict us to a single security constraint (targeted for `DonateJSFWeb/FindEmployee.jsp`) but applying to all subsequent pages accessed through request forwarding.

  – Request redirection passes the new URI back to the client side, and therefore allows separate security constraints for each page.

- So, why use request forwarding and why is it the default? The answer is to prevent the user of the Web application from knowing the URLs of secondary pages (`EmployeeDetails` or `DonationConfirmation`). This ensures that they always start at the primary entry point into the Web application (`FindEmployee.jsp`).

  With request forwarding, the user can see the secondary Web pages and can bookmark one of them. What happens when the user reloads that bookmark (after the session information has expired)? The Faces Managed bean (which we create in the next section) will no longer have a value, and therefore you must put extra logic into the Web application to handle this condition of incorrect page navigation.

- The choice presented so far is not pleasant: Use request forwarding (to allow multiple Java EE declarative role-based security policies but allow the potential for incorrect page navigation, or use request redirection and limit Java EE declarative role-based security to a single policy for the entire Web application, but reduce the potential for incorrect page navigation.

- There is a third approach: Stick with request forwarding (and a single Java EE declarative role-based security policy), but then use Java EE programmatic role-based security to control the transition from one page to another. See 10.6.4, “Implement programmatic security in a Web application” on page 312 for more details on programmatic security.

8.5 Create the managed beans

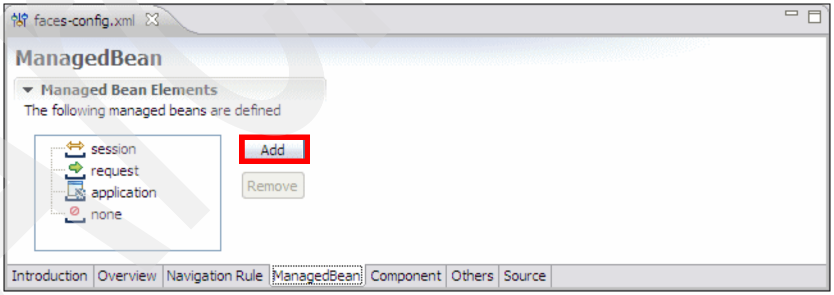
In this section, we create the managed beans that hold the data and action code for the pages. We use a package named **donate.jsf.bean** to hold two managed beans:

- **`FindEmployeeManagedBean`**: Holds the employee data and the action logic to find an employee.
- **`DonateHoursManagedBean`**: Holds the data and action logic to donate hours to the specified donation fund.

8.5.1 Define the managed beans

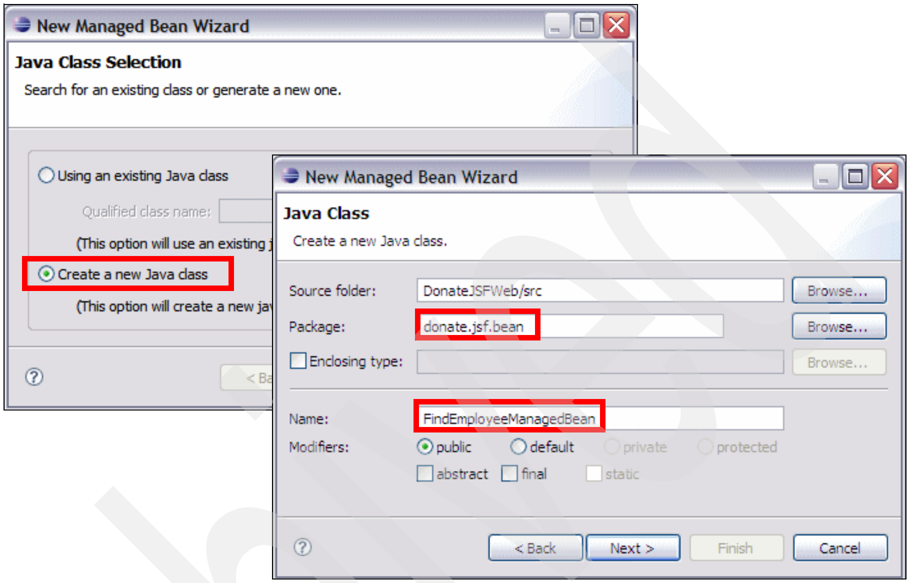
Managed beans are defined in the Faces configuration file. They are managed (that is, allocated) at execution time when needed for pages:

1. In the Project Explorer, open **faces-config.xml** (in `DonateJSFWeb/WebContent/WEB-INF`).
2. Select the **ManagedBean** tab and click **Add**.



3. In the New Managed Bean Wizard, select **Create a new Java class**, and click **Next**.

4. In the New Managed Bean Wizard pop-up, set the package to **donate.jsf.bean**, the name to **FindEmployeeManagedBean**, and click **Next**.



New Managed Bean Wizard

Java Class Selection

Search for an existing class or generate a new one.

○ Using an existing Java class

Qualified class name:

(This option will use an existing j

● Create a new Java class

(This option will create a new ja

New Managed Bean Wizard

Java Class

Create a new Java class.

Source folder: DonateJSFWeb/src

Package: donate.jsf.bean

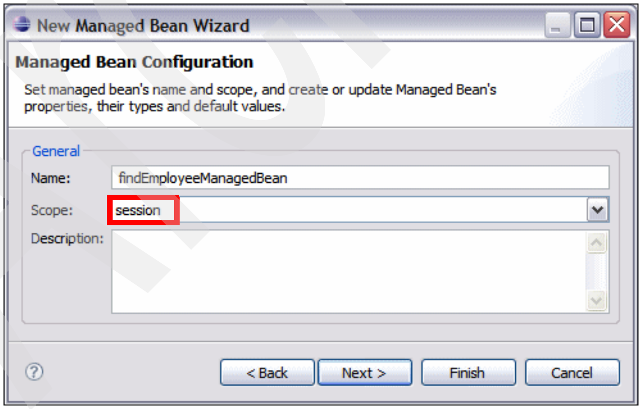
□ Enclosing type:

Name: FindEmployeeManagedBean

Modifiers: ● public ○ default ○ private ○ protected

□ abstract □ final □ static

5. In the Managed Bean Configuration pop-up, accept the name, and for Scope select **session**. Click **Finish**.



New Managed Bean Wizard

Managed Bean Configuration

Set managed bean's name and scope, and create or update Managed Bean's properties, their types and default values.

General

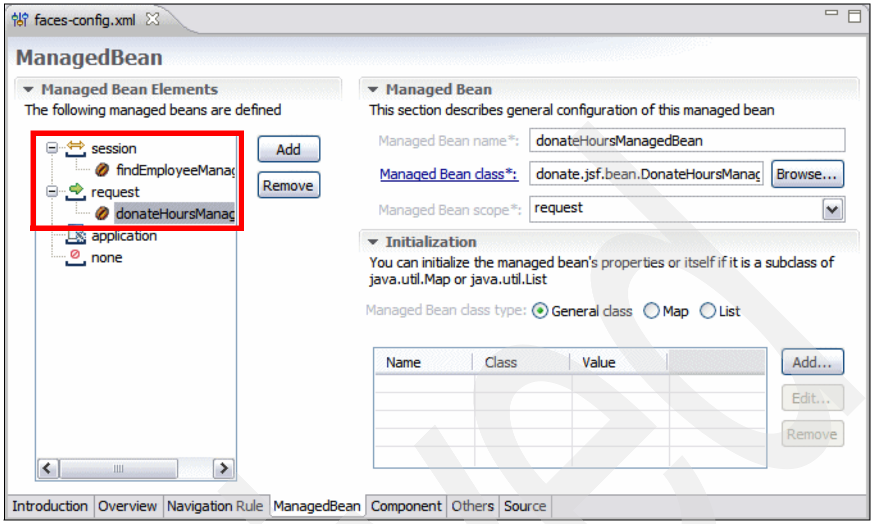
Name: findEmployeeManagedBean

Scope: session

Description:

6. Repeat this sequence for the **DonateHoursManagedBean** class, but set the scope to **request**.

7. Save `faces-config.xml`, but do not close it.


faces-config.xml

ManagedBean

Managed Bean Elements

The following managed beans are defined

Managed Bean

This section describes general configuration of this managed bean

Managed Bean name*: donateHoursManagedBean

Managed Bean class*: donate.jsf.bean.DonateHoursManag

Managed Bean scope*: request

Initialization

You can initialize the managed bean's properties or itself if it is a subclass of java.util.Map or java.util.List

Managed Bean class type: ● General class ○ Map ○ List

Name	Class	Value

8. Select the **Source** tab. Notice that the `faces-config.xml` has two mappings for managed beans (Example 8-4).

Example 8-4 Manage beans in the Faces configuration file

```
<managed-bean>
   <managed-bean-name>findEmployeeManagedBean</managed-bean-name>
   <managed-bean-class>donate.jsf.bean.FindEmployeeManagedBean</man...>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
   <managed-bean-name>donateHoursManagedBean</managed-bean-name>
   <managed-bean-class>donate.jsf.bean.DonateHoursManagedBean</man...>
   <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

9. Close `faces-config.xml`.

Behind the scenes:

Notice that two managed beans were created. One managed bean resides in request scope. The other managed bean resides in session scope:

- Request scope means that all attributes and parameters are stored in the HTTP request and are only good for one HTTP request/response pair.
- Session scope means that all attributes and parameters are stored in the HTTP session and persist until explicitly removed.

So, when do we store values in session versus request? Ideally, most values should live in the request scope. Values that might be expensive to re-create and that are needed across pages within an application can be stored in the session scope. However, excess storage of objects in the session scope causes session bloat, which can negatively influence application performance.

8.5.2 Add the dependency to the EJB project

Because the JSF managed beans require access to the classes of the session EJBs, we have to create a dependency of the DonateJSFWeb project to the DonateEJB project:

1. Right-click the **DonateJSFWeb** project and select **Properties**.
2. In the Properties pop-up, select **Java EE Module Dependencies** on the left, and select the **DonateEJB.jar** module on the right.
3. Click **OK**.

8.5.3 Implement the FindEmployeeManagedBean class

In 8.6.2, “Bind FindEmployee to a managed bean” on page 259, we will bind fields in `FindEmployeeManagedBean` for an input field and a Submit button to this managed bean:

```
Input field:    value="#{findEmployeeManagedBean.empId}"
Submit button:  action="#{findEmployeeManagedBean.findEmployee}"
```

Therefore, we have to define an `empId` property to hold the input field value, and a `findEmployee` method to hold the action code to access the `EmployeeFacade` session bean and retrieve the employee:

1. Open **FindEmployeeManagedBean** (in `DonateJSFWeb/Java Resources: src/donate.jsf.bean`) and place the cursor before the closing brace.

2. Define two properties:

```
private String empId;
private Employee employee;
```

3. Right-click in the source code and select **Source** → **Generate Getters and Setters**. Select both properties and click **OK**. This creates the get and set methods for the two properties.
4. To access the `EmployeeFacade` session bean, add a definition of the session bean to the class:

```
@EJB EmployeeFacadeInterface employeeFacade;
```

5. Create the `findEmployee` method by inserting the **F38 JSF Employee Managed Bean findEmployee** snippet from the Snippets view at the end of the class (Example 8-5).

Example 8-5 Action code to retrieve an employee

```
public String findEmployee() {
   Employee employee = null;
   try {
      employee = employeeFacade
                      .findEmployeeById(Integer.parseInt(getEmpId()));
      setEmployee(employee);
      return "success";
   }
   catch(Exception e) {
      e.printStackTrace();
   }
   return "";  // stay on the same page
}
```

6. Organize imports (**Ctrl+Shift+O**).
7. Save and close `FindEmployeeManagedBean.java`

Behind the scenes:

- The `findEmployee` method takes the value from the `empId` property, which is mapped to the input field in the `FindEmployee` JSP.
- The method then uses the `EmployeeFacade` session bean to retrieve the employee, and stores the employee.
- Finally, the method returns `success` to pass control to the `EmployeeDetails` JSP.

Remember that the whole bean is stored in session scope, therefore, the employee is available to the next JSP.

8.5.4 Implement the DonateHoursManagedBean class

The `DonateHoursManagedBean` is used for the logic to donate hours from an employee to a fund. Therefore, we have to define an `employee` property to hold the employee data, and a `donateHours` method to hold the action code to access the `DonateBean` session bean and perform the donation of vacation hours:

1. Open **DonateHoursManagedBean** (in `DonateJSFWeb/Java Resources: src/donate.jsf.bean`) and place the cursor before the closing brace.
2. Create three properties and the name of the default fund:

   ```
   private String resultMessage;
   private int hoursDonated;
   private Employee employee;

   private final String fundname = "DonationFund";
   ```

3. Create getters and setters for the three properties (not for the `fundname`).
4. Create references for the two session beans that are used in the action code:

   ```
   @EJB DonateBeanRemote donateBean;
   @EJB EmployeeFacadeInterface employeeFacade;
   ```

   Note that we have to use the remote interface of the `DonateBean` session bean, because the business interface does not have the `donateToFund` method that returns a result.

5. Create the `donateHours` method by inserting the **F39 JSF Donate Managed Bean donateHours** snippet from the Snippets view at the end of the class (Example 8-6).

Example 8-6 Action code to donate vacation hours

```
public String donateHours() {
   FacesContext ctx = FacesContext.getCurrentInstance();
   HttpServletRequest request = (HttpServletRequest)
                              ctx.getExternalContext().getRequest();
   FindEmployeeManagedBean employeeManagedBean =
                (FindEmployeeManagedBean) request.getSession()
                            .getAttribute("findEmployeeManagedBean");
   String empId = employeeManagedBean.getEmpId();
   try {
      int employeeId = Integer.parseInt(empId);
      String msg = donateBean.donateToFund(employeeId, fundname,
                                                  getHoursDonated());
      setResultMessage(msg);
      setEmployee(employeeFacade.findEmployeeById(employeeId));
      return "success";
   } catch (Exception e) {
      e.printStackTrace();
```

```
        }
        return "";
    }
```

6. Organize imports (**Ctrl+Shift+O**).
7. Save and close DonateHoursManagedBean.java.

Behind the scenes:

- The FacesContext is a standard JSF class used to retrieve elements of the page, such as the request block.
- From the request block we retrieve the session data (the managed bean).
- We execute donateToFund in the DonateBean session bean to donate a number of hours from an employee to the default fund.
- We store the result in a property, that will be mapped to the page.
- We retrieve the employee again to get the remaining vacation hours.

Recapitulation:

Consider what we have done for the business logic:

- We created and implemented two managed beans and configured them in faces-config.xml, including their scope (session or request).
- We provided the properties to hold input and output values of the JSPs.
- We implemented the controller logic for each action.

So how is this different from servlet programming? In servlet programming:

- The equivalent of the managed beans are the servlets. Servlets are configured in the web.xml. The servlets contain the controller logic.
- Managed beans have to declaratively specify scope, because they contain form data that JSF automatically retrieves and sets in either request or session. In servlet programming, retrieval of form data from either request or session is done programmatically.
- The navigation across a Web application is also declarative in JSF. We got a taste of how this is achieved when we configured the navigation rules in faces-config.xml. In servlets, navigation across a Web application is programmatic.

8.6 Implement the JSP pages

In this section we complete the pages that were defined in 8.4.2, “Create the JSPs” on page 244.

For reference, the final code of the pages is available in the Snippets view:

- F35 JSF Find Employee JSP
- F36 JSF Employee Details JSP
- F37 JSF Donation Confirmation JSP

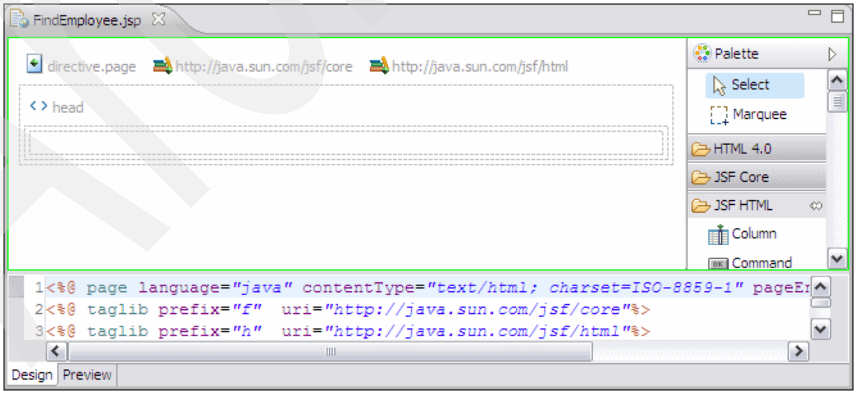
8.6.1 Implement the FindEmployee page

For the `FindEmployee.jsp`, we use the JSF Palette to build the JSP. The JSP has three widgets on the page:

- Input field for the employee ID (Text Input component).
- Label of the input field (Output Text component).
- Submit button to invoke the execution of the employee search function (Command Button component).

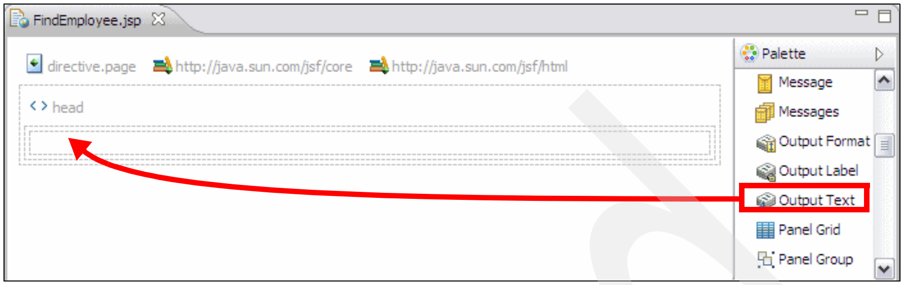
To implement the `FindEmployee` page, follow these steps:

1. Right-click **FindEmployee.jsp** (in `DonateJSFWeb/WebContent`), and select **Open With** → **Web Page Editor**. Note the split editor with a design view on top, and the source below.
2. Ensure that the Palette is expanded on the right (by clicking on the left facing arrow), and click **JSF HTML** to expand that category of components.



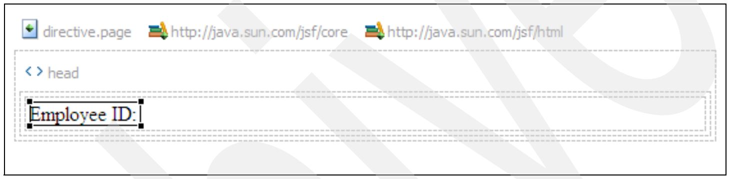
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEr
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
```

3. Select the **Output Text** component and drag it into the page into the dotted rectangle (which is the `<f:view>` tag inside the `<body>`).

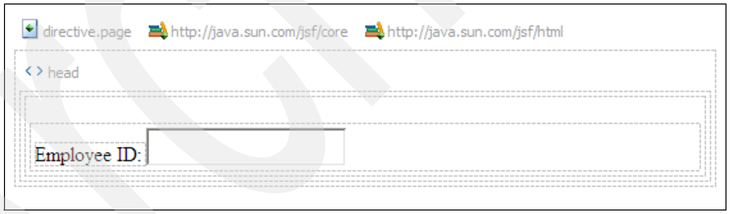


4. In the Properties view, set the Value to **Employee ID:** (with a trailing space). The source tag becomes:

```
<h:outputText value="Employee ID: "> </h:outputText>.
```



5. Drag a **Text Input** component behind the output text. Then move the output text (Employee ID:) into the form that is created.



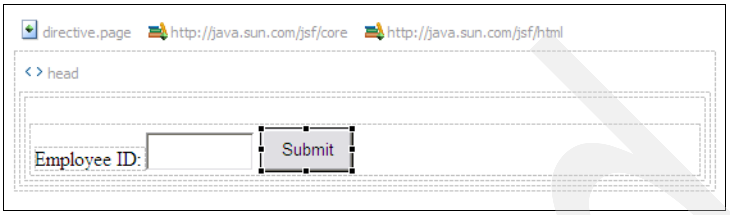
6. Select the input text field. In the Properties view, set the ID to **empid**. In the source add maximum length and size, so that the tag becomes:

```
<h:inputText id="empid" maxlength="6" size="10"></h:inputText>
```

Note: You can also set the `maxlength` and `size` attributes in the Properties view, **Attributes** tab (scroll down to HTML).

7. Drag a **Command Button** behind the input field. In the Properties view, set id as **submit**, value as **Submit**, and type as **submit**. The tag becomes:

```
<h:commandButton id="submit" type="submit"
                             value="Submit"></h:commandButton>
```



8. Set the title tag to `<title>Donate - Search</title>`.
9. Leave the editor open because we will make further changes in the next section.

8.6.2 Bind FindEmployee to a managed bean

The `FindEmployee` page stores the employee ID into the managed bean named `FindEmployeeManagedBean`, and the Submit button executes a method in that JavaBean. We defined this bean in 8.5.1, “Define the managed beans” on page 250.

Behind the scenes:

The managed beans do not have to exist prior to defining the binding in the JSP file. However, if a referenced managed bean does not exist, you will see a warnings in the Problems view:

```
findEmployeeManagedBean cannot be resolved
```

Therefore, in general it is considered to a best practice to define the managed beans before completing the JSPs. However, in some situations you might find it appropriate to complete the JSP definitions first.

We follow these steps:

1. Select the input field in the JSP, and in the Properties view set the Value to **#{findEmployeeManagedBean.empId}**. This instructs JSF to place the user input into a property named `empId` in the managed bean.
2. Select the **Submit** button, and in the Properties view set the Action to **#{findEmployeeManagedBean.findEmployee}**. This instructs JSF to run the `findEmployee` method in the managed bean.

3. Save and close FindEmployee.jsp.

Tip: You can use context assist to help with the binding:

- Type **#{}**.
- Place the cursor between the braces and press **Ctrl+Space**.
- Select the managed bean (findEmployeeManagedBean) from the list.
- Type a period and press **Ctrl+Space** again.
- Select the property (empId).

Example 8-7 shows the source of the FindEmployee page.

Example 8-7 FindEmployee.jsp code

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Donate - Search</title>
</head>
<body>
<f:view>
   <h:form>
      <h:outputText value="Employee ID: "></h:outputText>
      <h:inputText id="empid" maxlength="6" size="10"
         value="#{findEmployeeManagedBean.empId}"></h:inputText>
      <h:commandButton id="submit" type="submit" value="Submit"
         action="#{findEmployeeManagedBean.findEmployee}"></h:commandButton>
   </h:form>
</f:view>
</body>
</html>
```

Behind the scenes:

Notice the <%@ taglib> tags that define the JSF libraries. The body of the page is a JSF <view> with an embedded <form> that submits data to the server.

8.6.3 Implement the EmployeeDetails page

The `EmployeeDetails` JSP contains a table with the employee data, and an input field to donate hours of vacation time.

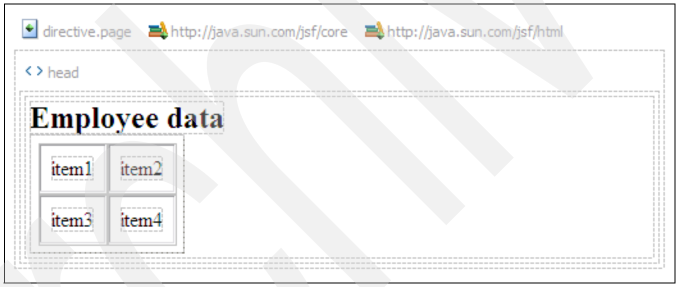
To implement the `EmployeeDetails` page, follow these steps:

1. Open **EmployeeDetails.jsp** (in `DonateJSFWeb/WebContent`) with the Web Page Editor.
2. Set the title tag to `<title>Donate - Employee</title>`.
3. Drag an **Output Text** component from the JSF HTML Palette into the page, and in the Properties view, set the text to **Employee data**.
4. In the Properties view (for this output component), click **Edit** to set the style.

   In the CSS Style Definition pop-up, select **24** for Size and **bold** for Weight. Click **OK** and the source tag is change to:

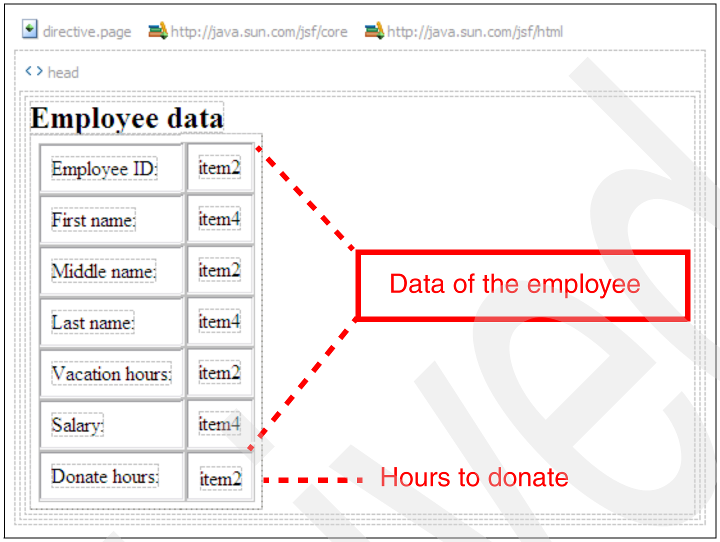
   ```
   <h:outputText value="Employee data" style="font-weight: bold; font-size: 24px"
   ```

5. Drag a **Panel Grid** component into the page. By default two rows and two columns are created.



6. In the Source view, duplicate the tags of the rows to create seven rows.

7. Select each **Output Text** component in the first cell of each row and set the values to: `Employee ID:`, `First name:`, `Middle name:`, `Last name:`, `Vacation hours:`, `Salary:`, and `Donate hours:`.



directive.page http://java.sun.com/jsf/core http://java.sun.com/jsf/html

head

Employee data

Employee ID:	item2
First name:	item4
Middle name:	item2
Last name:	item4
Vacation hours:	item2
Salary:	item4
Donate hours:	item2

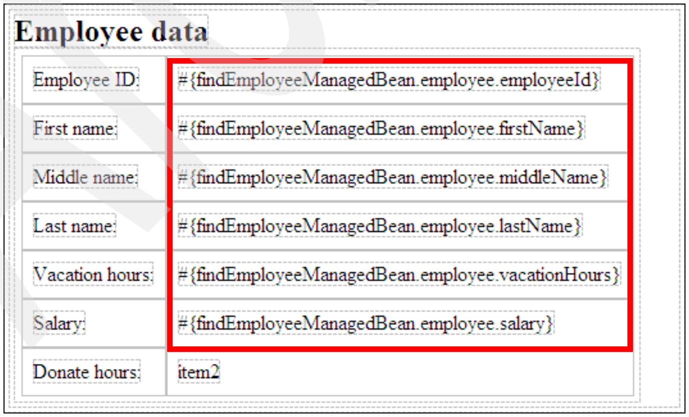
Data of the employee

Hours to donate

8. Select the **Panel Grid** component, and in the Properties view set border to **0**.

9. Select each **Output Text** component in the second cell of each row and set the values to results stored in the `findEmployeeManagedBean` managed bean, `employee` properties: `employeeId`, `firstName`, `middleName`, `lastName`, `vacationHours`, and `salary`, for example:

```
findEmployeeManagedBean.employee.employeeId
```



Employee data

Employee ID:	#{findEmployeeManagedBean.employee.employeeId}
First name:	#{findEmployeeManagedBean.employee.firstName}
Middle name:	#{findEmployeeManagedBean.employee.middleName}
Last name:	#{findEmployeeManagedBean.employee.lastName}
Vacation hours:	#{findEmployeeManagedBean.employee.vacationHours}
Salary:	#{findEmployeeManagedBean.employee.salary}
Donate hours:	item2

10. Delete the **item2** Output Text component in the last cell.

11. Drag a **Text Input** component into the last cell, set the length in the source code, and bind the value field in the Properties view.

    ```
    <h:inputText maxlength="3" size="5"
       value="#{donateHoursManagedBean.hoursDonated}”></h:inputText>
    ```

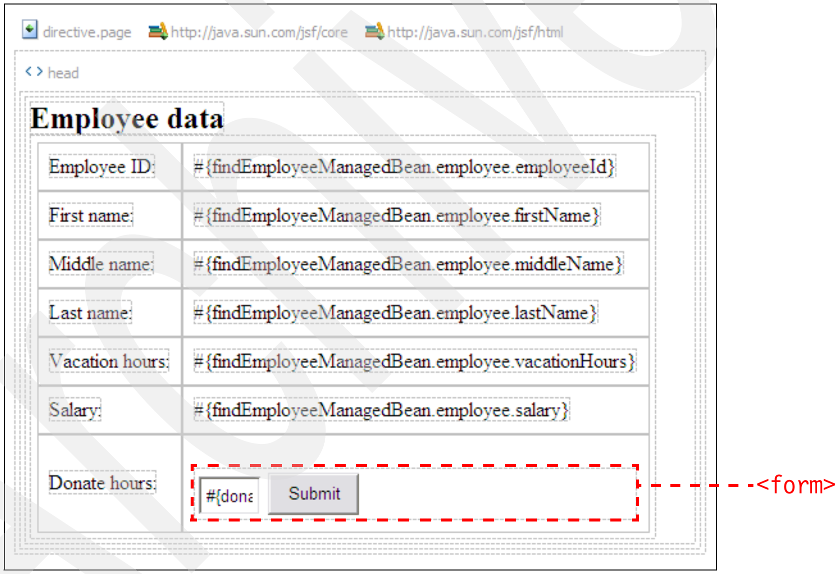
    Note: You can also set the `maxlength` and `size` attributes in the Properties view, **Attributes** tab.

    A `<form>` is automatically created for an input field.

12. Drag a **Command Button** after the input field. In the Properties set the value to **Submit**, select **submit** for type. and bind the Action to the `donateHours` method:

    ```
    <h:commandButton value="Submit" type="submit"
       action="#{donateHoursManagedBean.donateHours}"></h:commandButton>
    ```

13. The final design is shown here.



Employee data

Employee ID:	#{findEmployeeManagedBean.employee.employeeId}
First name:	#{findEmployeeManagedBean.employee.firstName}
Middle name:	#{findEmployeeManagedBean.employee.middleName}
Last name:	#{findEmployeeManagedBean.employee.lastName}
Vacation hours:	#{findEmployeeManagedBean.employee.vacationHours}
Salary:	#{findEmployeeManagedBean.employee.salary}
Donate hours:	#{dona Submit

14. Save and close `EmployeeDetails.jsp`.

Example 8-8 shows the source of the `EmployeeDetails` page.

Example 8-8   EmployeeDetails.jsp code

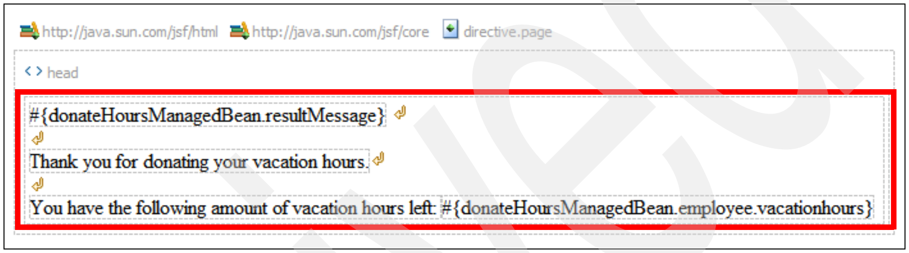
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
                                pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                                "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Donate - Employee</title>
</head>
<body>
<f:view>
   <h:outputText value="Employee data"
                   style="font-weight: bold; font-size: 24px"></h:outputText>
   <h:panelGrid border="0" columns="2">
      <h:outputText value="Employee ID:"></h:outputText>
      <h:outputText value="#{findEmployeeManagedBean.employee.employeeId}">
         </h:outputText>
      <h:outputText value="First name:"></h:outputText>
      <h:outputText value="#{findEmployeeManagedBean.employee.firstName}">
         </h:outputText>
      <h:outputText value="Middle name:"></h:outputText>
      <h:outputText value="#{findEmployeeManagedBean.employee.middleName}">
         </h:outputText>
      <h:outputText value="Last name:"></h:outputText>
      <h:outputText value="#{findEmployeeManagedBean.employee.lastName}">
         </h:outputText>
      <h:outputText value="Vacation hours:"></h:outputText>
      <h:outputText value="#{findEmployeeManagedBean.employee.vacationHours}">
         </h:outputText>
      <h:outputText value="Salary:"></h:outputText>
      <h:outputText value="#{findEmployeeManagedBean.employee.salary}">
         </h:outputText>
      <h:outputText value="Donate hours:"></h:outputText>
      <h:form>
         <h:inputText  maxlength="3" size="5"
            value="#{donateHoursManagedBean.hoursDonated}"></h:inputText>
         <h:commandButton value="Submit" type="submit"
            action="#{donateHoursManagedBean.donateHours}"></h:commandButton>
      </h:form>
   </h:panelGrid>
</f:view>
</body>
</html>
```

8.6.4 Implement the DonationConfirmation page

The `DonationConfirmation` JSP contains the result of donating vacation hours.

To implement the `DonationConfirmation` page, follow these steps:

1. Open **DonationConfirmation.jsp** (in `DonateJSFWeb/WebContent`) with the Web Page Editor.
2. Set the title tag to `<title>Donate - Result</title>`.
3. Create the confirmation page with four output text fields. We bind two of the fields to the `DonateHoursManagedBean` managed bean.



#{donateHoursManagedBean.resultMessage}

Thank you for donating your vacation hours

You have the following amount of vacation hours left: #{donateHoursManagedBean.employee.vacationhours}

   - Drag an **Output Text** component into the page and set the value to **#{donateHoursManagedBean.resultMessage}**. We bind the value to the result string in the managed bean.
   - With the cursor behind the output field, press **Enter** twice.
   - Drag an **Output Text** component into the page (after the new lines) and set the value to **Thank you for donating your vacation hours**.
   - With the cursor behind the output field, press **Enter** twice.
   - Drag an **Output Text** component underneath and set the value to **You have the following amount of vacation hours left:** (with a space).
   - Drag an **Output Text** component behind the text and set the value to **#{donateHoursManagedBean.employee.vacationHours}**.

4. Save and close `DonationConfirmation.jsp`.

Example 8-9 shows the source of the `DonationConfirmation` page.

Example 8-9 DonationConfirmation.jsp code

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
                                   pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```
                                   "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Donate - Result</title>
</head>
<body>
<f:view>
   <h:outputText value="#{donateHoursManagedBean.resultMessage}">
      </h:outputText><br><br>
   <h:outputText value="Thank you for donating your vacation hours.">
      </h:outputText><br><br>
   <h:outputText
         value="You have the following amount of vacation hours left: ">
      </h:outputText>
   <h:outputText value="#{donateHoursManagedBean.employee.vacationHours}">
      </h:outputText>
</f:view>
</body>
</html>
```

Recapitulation

In the preceding steps:

- We defined the three pages of the JSF application.
- We wired the navigation path throughout the application using the `faces-config.xml`.
- We also provided the invocation of the business functionality behind each page, by providing two managed beans with data and action code.
- We used JSF value binding #{...} to bind JSF input and output fields to properties of the managed beans.
- We used JSF action binding to bind Submit buttons to methods in the managed beans.

8.7 Test the JSF Web application

Implementation-wise, we have completed the functionality for pages in this application. Now, it is time to see our implementation in action:

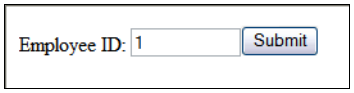
1. In the Servers view, start the server if it is not running.
2. Deploy the DonateEAR enterprise application by using **Publish** or **Add and Remove Projects** on the **ExperienceJavaEE Server**.

Sometimes it is necessary to remove an application and add it again if new modules have been added. So if you get deployment errors, remove and add the DonateEAR application.

3. To run the application, start a browser and use the URL:

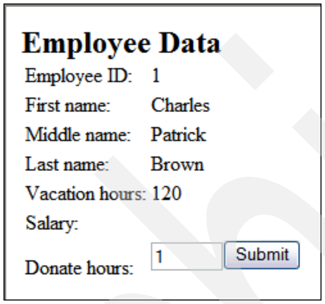
   `http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp`

4. The initial page is displayed (note that it takes a while the first time to compile the JSP). Enter an employee number and click **Submit**.



Employee ID: 1

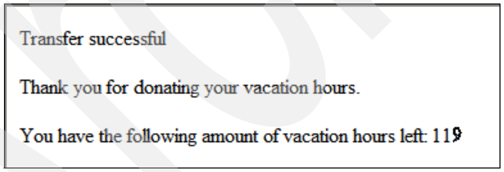
- The employee is retrieved and the employee data is displayed. Enter a number of hours (**1**) to donate, and click **Submit**.



Employee Data

Employee ID: 1
First name: Charles
Middle name: Patrick
Last name: Brown
Vacation hours: 120
Salary:
Donate hours: 1

- The donation is executed and the result is displayed.



Transfer successful

Thank you for donating your vacation hours.

You have the following amount of vacation hours left: 119

8.8 Optional extensions to DonateJSFWeb

The preceding sections provide a basic JSF application and all that is needed to complete Chapter 10, “Implement core security” on page 285.

The following sections show how to extend DonateJSFWeb to provide additional capabilities that are useful in a more robust application.

8.8.1 Error handling

The current JSF application does not provide any error handling, for example, when an employee is not found.

JSF error message components

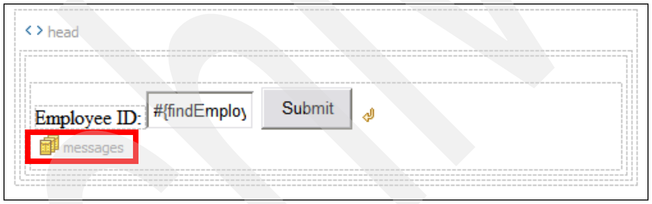
JSF provides two components to display errors:

- Message: Displays messages that are attached to another component, for example, an input field.
- Messages: Displays all error messages in a list or table.

Adding an error message to find an employee

To add an error message for an invalid employee ID, perform these steps:

1. Provide a field for error messages in the `FindEmployee` page:
   - Open **FindEmployee.jsp** (in `DonateJSFWeb/WebContent`)
   - Drag and drop a **Messages** component (JSF HTML) under the current data.



   - Configure the Messages component in the Properties view, Quick Edit tab:
     - For layout, select **table (an HTML table)** using the pull-down on the right.
     - For style, click **Edit** and set the color to red.
     - The source tag becomes:

```
<h:messages layout="table" style="color: #FF0000"></h:messages>
```

   - Save and close `FindEmployee.jsp`.
2. Insert the error message into the messages field by the action code in the managed bean:
   - Open **FindEmployeeManagedBean.java** (in `DonateJSFWeb/Java Resources: src/donate.jsf.bean`).

– Add the code in bold to the `findEmployee` method. The **F40 JSF Employee Managed Bean findEmployee error** snippet can be used to insert the two lines (Example 8-10).

Example 8-10 Displaying a Faces error message

```
public String findEmployee() {
   Employee employee = null;
   try {
      employee = employeeFacade.findEmployeeById
                                (Integer.parseInt(getEmpId()));
      setEmployee(employee);
      return "success";
   }
   catch(Exception e) {
      e.printStackTrace();
      FacesContext ctx = FacesContext.getCurrentInstance();
      ctx.addMessage("", new FacesMessage("Employee not found"));
   }
   return "";  // stay on the same page
}
```

– Organize imports (**Ctrl+Shift+O**).

– Save and close **FindEmployeeManagedBean.java**.

Behind the scenes:

- This code creates a Faces message when an employee is not found, and adds it to the Faces context. JSF displays all messages added to the Faces context in the Messages component.
- You can also provide the id of a component in the `addMessage` method to display the message in a specific Message component. The empty id ("") directs the message to the Messages component.

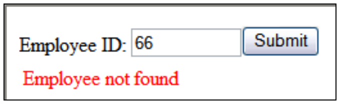
3. Test the FindEmployee error message:

– Publish the updated application.

– Start a browser and use the URL:

`http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp`

– Verify that an error message is displayed for an invalid employee ID:



Employee ID: 66

Employee not found

Adding an error message when vacation donation fails

The DonateBean session bean returns an error message string when not enough hours are available. We can turn this into a JSF error message as well:

1. Add a Messages component to **EmployeeDetails.jsp** (in DonateJSFWeb /WebContent) under the Panel Grid, using the same steps as described for FindEmployee.jsp.
2. Open **DonateHoursManagedBean** (in DonateJSFWeb/Java Resources: src/donate.jsf.bean):
   – Add the code in bold to the donateHours method in the DonateHoursManagedBean. The **F41 JSF Donate Managed Bean donateHours error** snippet can be used to replace the complete donateHours method (Example 8-11).

   Example 8-11 Faces error messages for donation

```
public String donateHours() {
   FacesContext ctx = FacesContext.getCurrentInstance();
   HttpServletRequest request = ....
   ......
   try {
      int employeeId = Integer.parseInt(empId);
      String msg = donateBean.donateToFund
                      (employeeId, fundname, getHoursDonated());
      setResultMessage(msg);
      if (msg.substring(0,6).equals("Error:")) {
         ctx.addMessage("", new FacesMessage(msg));
         return "";
      }
      setEmployee(employeeFacade.findEmployeeById(employeeId));
      return "success";
   } catch (Exception e) {
      e.printStackTrace();
      ctx.addMessage("", new FacesMessage("Execution error"));
   }
   return "";
}
```

   – Organize imports (**Ctrl+Shift+O**).
   – Save and close DonateHoursManagedBean.java.
3. Test the donation error message:
   – Publish the updated application and start a browser with the URL:

   http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp

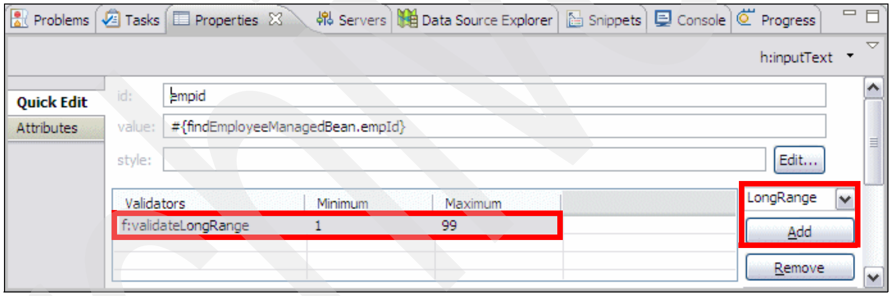
- Enter a valid employee ID, and then attempt to donate more vacation hours than are currently available. The donation fails with the error message:

```
Error: Employee does not have sufficient hours to deduct from
```

8.8.2 Validation

JSF provides elaborate validation of data, for example, when typing values of the wrong data type. In addition, we can add custom validation by value range, for example, for the employee ID:

1. Open **FindEmployee.jsp** (in `DonateJSFWeb/WebContent`):
   - Select the input field and go to the Properties view.
   - For Validators, select **LongRange** from the pull-down menu on the right side, and click **Add**.
   - Type **1** as minimum, and **99** as maximum.



id: empid
value: #{findEmployeeManagedBean.empId}
style:

Validators	Minimum	Maximum
f:validateLongRange	1	99

   - In the Source, you can see that a `<f:validateLongRange>` tag:

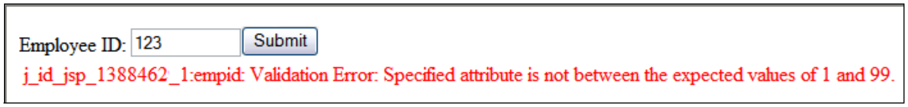
```
<f:validateLongRange minimum="1" maximum="99"></f:validateLongRange>
```

   - Save and close `FindEmployee.jsp`.

2. Test the find employee validation:
   - Publish the updated application.
   - Start a browser and use the URL:

     `http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp`

   - Verify that an error message is displayed when an invalid employee ID is entered.



j_id_jsp_1388462_1:empid: Validation Error: Specified attribute is not between the expected values of 1 and 99.

– Type an alphabetic employee ID and you receive the JSF error message:

```
j_id_jsp_1871129070_1:empid: Validation Error: Value is not of the
correct type.
```

3. Test standard JSF validation:

– Enter a valid employee number, and in the resulting employee details page, enter an alphabetic value for the vacation amount, and click **Submit.** You receive the error message:

```
_id_jsp_1067239513_16:j_id_jsp_1067239513_17: 'x' must be a number
consisting of one or more digits.
```

Behind the scenes:

These error messages are from a standard JSF validation. The input field vacation amount is bound to an integer field in the managed bean, and JSF cannot convert the alphabetic value to an integer.

8.9 Explore! JSF

Now that we have gotten a taste of JSF architecture and development, the next steps are to explore the various supporting JSF projects and technologies.

For more resources on JSF, consult these IBM Redbooks publications:

- *Rational Application Developer V7 Programming Guide*, SG24-7501, Chapter 14, *Develop Web applications using JSF and SDO*. Note that instead of Service Data Objects (SDO) we use JPA entities to access the database.
- *Rational Application Developer V7.5 Programming Guide*, SG24-7672, Chapter 16, *Develop Web applications using JSF,* where we also use JSF with JPA entities.

9

Create the application client

In this chapter, we create simple non-graphical client applications that can interact with the server side `DonateBean` session EJB (Figure 9-1):

- Java EE application client that provides *thick client* access to the session EJB.
- Standalone client that provides a *thinner* client access to the session EJB.

9.1 Learn!

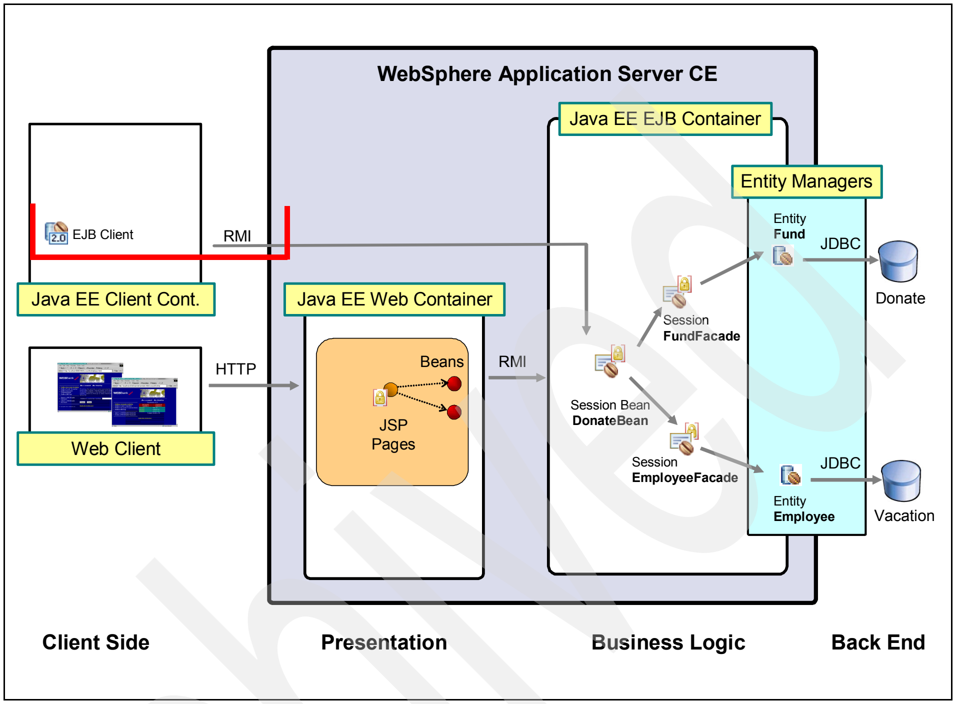


Figure 9-1 Donate application: Java EE application client

The traditional Web application created in the previous chapter renders all content on the client's system using a Web browser. This is called a *thin* client, which means that no unique application code is required. However, this type of client is not appropriate in certain situations, such as these:

- The client system requires sub-second response time. For example, customer service applications often require very fast response times to ensure that customer calls are processed as quickly as possible. It is often unacceptable to wait—for even one second—for a Web browser to send the request to the server to display an alternate page.
- The client system requires high graphics resolution or complex graphics display capabilities that cannot be accommodated by the browser.
- The client system requires asynchronous notification from the server side, using messaging (JMS), for example. A standard browser is driven by the user, which means that nothing happens until the user presses Enter or until the browser page is automatically refreshed using a timer.

For these situations, Java EE provides a *thick client* called the Java EE application client that consists of a runtime infrastructure that must be installed on the client system along with the actual application. The user can invoke this application, which then interacts with the Java EE server-side environment using resources such as JNDI, EJBs, JDBC, and JMS.

The one drawback of the application client is that it is a thick client: It requires the installation (or download) of a Java EE runtime environment that consists of Java libraries and configuration files. This is called the Java EE Application Client container. In the case of WASCE, this is actually the entire server installation environment (over 150 MB).

As a result, many developers prefer to create a smaller informal runtime environment that consists only of the required JAR files and configuration files that can be easily downloaded and executed on demand. This is sometimes called a standalone client.

The Java EE specification also defines the Applet container which supports the execution of Java programs in the context of a browser session. Although this is formally defined in Java EE, in reality applets have no direct access to server-side Java EE services and thus can be treated as standalone clients.

The examples used in this chapter are simple, non-graphical clients. However, most real-world application clients are implemented using a graphical display capability such as Swing, Abstract Window Toolkit (AWT), or Standard Widget toolkit (SWT).

There are no specific Learn resources for this chapter.

9.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

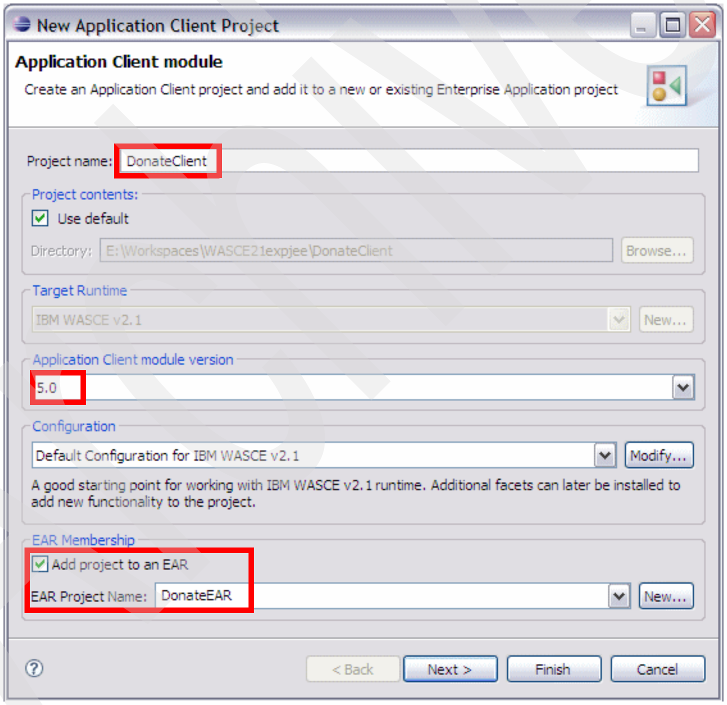
For this chapter to work, make sure that you have completed these steps:

- Defined the two databases and the matching database pools
- Created the JPA entities and the three session beans
- Created and run the JUnit test cases
- Optionally created and tested the Web application

9.3 Create the application client project

In this section you create the application client project that will contain the main Java class and any other required class path definitions/artifacts:

1. In the Java EE Perspective, Project Explorer, right-click **DonateEAR** and select **New** → **Application Client Project**.
2. In the New Application Client Project/Application Client module pop-up:
   - Set the Project name to **DonateClient**.
   - Set Application Client module version to **5.0**.
   - Set Configuration to **Default Configuration for IBM WASCE v2.1**.
   - Under EAR membership, select **Add project to an EAR**, and select **DonateEAR**.
   - Click **Next**.



New Application Client Project

Application Client module

Create an Application Client project and add it to a new or existing Enterprise Application project

Project name: DonateClient

Project contents:

☑ Use default

Directory: E:\Workspaces\WASCE21expjee\DonateClient

Target Runtime

IBM WASCE v2.1

Application Client module version

5.0

Configuration

Default Configuration for IBM WASCE v2.1

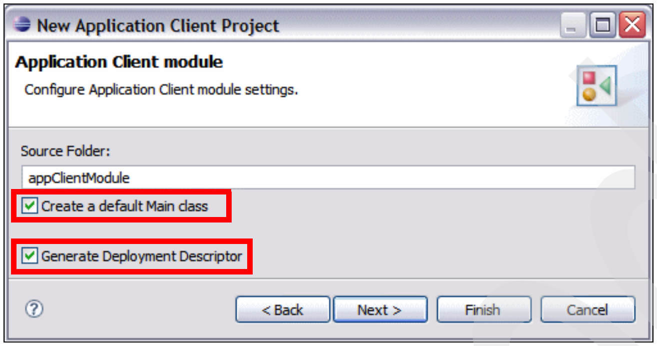
A good starting point for working with IBM WASCE v2.1 runtime. Additional facets can later be installed to add new functionality to the project.

EAR Membership

☑ Add project to an EAR

EAR Project Name: DonateEAR

3. In the New Application Client Project/Application Client module pop-up, select **Create a default Main class**, **Generate Deployment Descriptor**, and click **Next**.



New Application Client Project

Application Client module

Configure Application Client module settings.

Source Folder: appClientModule

☑ Create a default Main class

☑ Generate Deployment Descriptor

4. In the New Application Client/Geronimo Deployment Plan pop-up, set the Group Id to **ejee** (to match the enterprise application), leave the Artifact Id empty, and click **Finish**.

Behind the scenes:

The application client project would be in a different enterprise application in most real world examples. We keep it simple here and use one enterprise application for the complete example.

Careful: if you enter `DonateClient` as artifact id, the enterprise application deployment file is overwritten as well with this name.

9.3.1 Workaround: Update Geronimo deployment descriptor

The tooling currently generates a Geronimo deployment descriptor that produces errors when deployed to the server. We have to remove the offending lines:

1. In the Project Explorer, open **geronimo-application-client.xml** (in `DonateClient/appClientModule/META-INF`).
2. In the resulting editor:
   - Delete the `<name:service-ref>` tag lines before the closing tag.

```
~~<name:service-ref>~~
   ~~<name:service-ref-name>ServiceRefName</name:service-ref-name>~~
~~</name:service-ref>~~
</client:application-client>
```

   - Save and close `geronimo-application-client.xml`.

9.4 Configure the application client project

We must add DonateEJB to the manifest for DonateClient. This is required so that the Main class has access to the donateToFund method that we created in the DonateBean session EJB (in Chapter 7, “Create the Donate session bean for the business logic” on page 209):

1. In the Project Explorer, right-click **DonateClient** and select **Properties**.
2. In the resulting Properties for DonateClient pop-up:
   - On the left, select **Java EE Module Dependencies**.
   - On the right, select **DonateEJB.jar**.
   - Click **OK** to save the changes.

Behind the scenes:

The DonateEJB.jar file is added to the MANIFEST.MF file, and makes the classes in the EJB project available to the client project.

9.5 Code the main method

As with any standalone Java application, a main method is required to start the application. The Main class was created in the default package. Use the following instructions to code the main method:

1. In the Project Explorer, open **Main.java** (in DonateClient/appClientModule/(default package)). Note that the class currently contains the following main method:

```
public static void main(String[] args) {
      // TODO Auto-generated method stub
}
```

2. In the Main editor, and insert the following lines, before the main method:

```
@EJB
static DonateBeanRemote aDonate;
```

3. In the main method replace the // TODO line with a blank line, and insert the **F45 App Client Main** snippet from the Snippets view (Example 9-1).

Example 9-1 Application client main method

```
public static void main(String[] args) {

   if (args.length == 3) {
      int employeeId = Integer.parseInt((args[0]));
      String fundName = args[1];
      int hours = (new Integer(args[2])).intValue();

      System.out.println("Employee Number = " + employeeId);
      System.out.println("Fund            = " + fundName);
      System.out.println("Donation Amount = " + hours);

      System.out.println("Donation result = " +
         aDonate.donateToFund(employeeId, fundName, hours));
   } else {
      System.out.println("Program requires 3 command line arguments.");
   }
}
```

4. Organize the imports (**Ctrl+Shift+O**)
5. Save and close `Main.java`.

Behind the scenes:

- Initialize the session bean instance through annotation:

```
@EJB
static DonateBeanRemote aDonate;
```

Note how much easier this is with EJB 3.0, instead of having to locate the home of the session bean through JNDI.

- Verify that the appropriate arguments were passed to the main method:

```
if (args.length == 3) {
   int employeeId = Integer.parseInt((args[0]));
   String fundName = args[1];
   int hours = (new Integer(args[2])).intValue();

   System.out.println("Employee Number = " + employeeId);
   System.out.println("Fund            = " + fundName);
   System.out.println("Donation Amount = " + hours);
...
...
} else {
   System.out.println("Program requires 3 command line arguments.");
```

- Call the `donatetoFund` method and print out the results:

```
System.out.println("Donation result = " +
         aDonate.donateToFund(employeeId, fundName, hours));
```

9.6 Test the application client

Java EE application clients require a configured runtime environment called the Application Client container. This is similar to the Web container used for Servlets and JSPs and the EJB container used for EJBs.

With WASCE, the following considerations apply:

- Unlike other Java EE application servers, WASCE does not provide a unique Application Client container. Instead, you must install the full server package if you want to run an application client.
- This is compliant with the Java EE specification, which does not require that you provide a unique installation process for the Application Client container (the specification only requires that it exists). Also, because WASCE has a very small server footprint of around 150 MB, the net disk space savings for a special Application Client container most likely outweighs the realized benefit of disk space.

The WASCE IDE does not provide tooling for running and debugging the application client, as you would run and debug a servlet or EJB. Instead, you must execute the client from the command line using the **client** command:

1. In the Servers view, if the DonateEAR application is not deployed to the ExperienceJavaEE Server, right-click **ExperienceJavaEE Server**, select **Add and Remove Projects**, and publish the **DonateEAR** application.
2. If the application is already deployed, right-click **ExperienceJavaEE Server** and select **Publish**.

Behind the scenes:

The DonateClient application client module has been added to the DonateEAR enterprise application. Therefore, we must republish the EAR.

3. Run the **client** command in an operating system prompt from the C:\IBM\WASCE21\bin directory:

```
client ejee/DonateClient/1.0/car 1 DonationFund 1
```

Behind the scenes:

Deployment has created a repository/ejee/DonateClient directory, where the client code is stored (DonateClient.jar), together with the EJB dependency code (DonateEJB.jar), and the two JPA project JAR files. This code is executed by the **client** command.

4. The command generates output similar to this:

```
Using GERONIMO_BASE:   C:\IBM\WASCE21
Using GERONIMO_HOME:   C:\IBM\WASCE21
Using GERONIMO_TMPDIR: var\temp
Using JRE_HOME:        C:\Program Files\IBM\Java50\jre
    Client Build:  V2.1.0.0-20080825
    Java Version:  1.5.0
Employee Number = 1
Fund            = DonationFund
Donation Amount = 1
Donation result = Transfer successful
```

Off course?

If the client command fails, verify that the `geronimo-application-client.xml` deployment descriptor specifies **ejee** as Group Id. If you did not overwrite `default` with `ejee`, deployment seems successful, but the code does not work.

In that case, edit all the deployment descriptor files and reset the Group Id to `ejee`:

- `DonateClient: geronimo-application-client.xml`
- `DonateEAR:    geronimo-application.xml`
- `DonateEJB:    openejb-jar.xml`

9.7 Create and test a standalone client

In this section we create a standalone Java client that accesses the `DonateBean` session bean. Instead of starting with a new project, we can reuse the DonateEJB_Tester project and add a `main` method to the `DonateBeanTester` class:

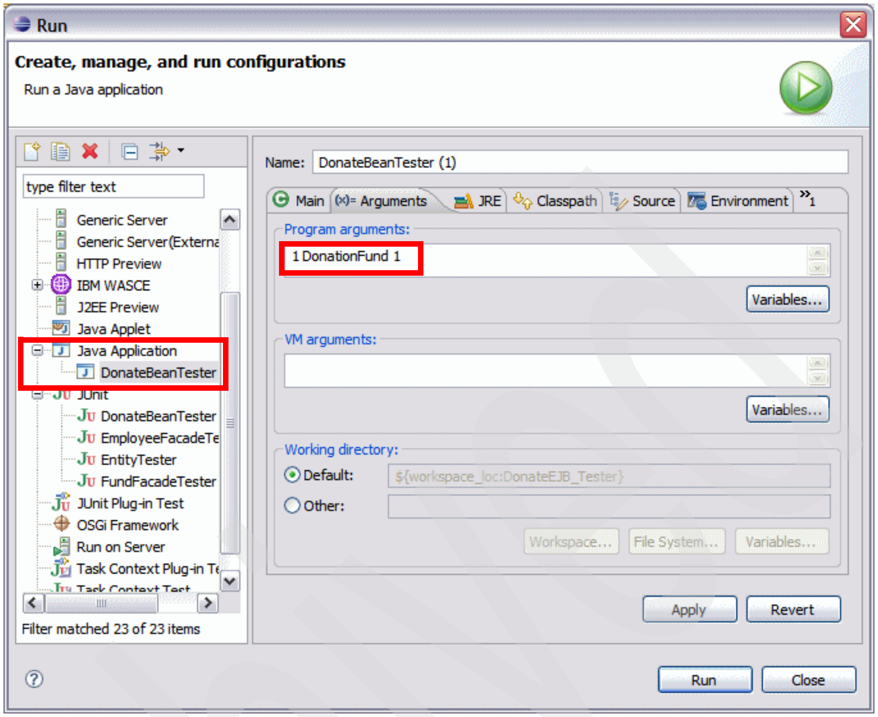
1. In the Project Explorer, open **DonateBeanTester.java** (in `DonateEJB_Tester/src/donate.test`).
   – Insert the **F46 Standalone Client Main** snippet into the class before the closing bracket (Example 9-2).
   – Save and close `DonateBeanTester.java`.

Example 9-2 Standalone client main method

```
public static void main(String[] args) throws Exception {
   DonateBeanTester.init();
   if (args.length == 3) {
      int employeeId = Integer.parseInt((args[0]));
      String fundName = args[1];
      int hours = (new Integer(args[2])).intValue();

      System.out.println("Employee Number = " + employeeId);
      System.out.println("Fund            = " + fundName);
      System.out.println("Donation Amount = " + hours);

      System.out.println("Donation result = " +
            donateBean.donateToFund(employeeId, fundName, hours));
   } else {
      System.out.println("Program requires 3 command line arguments.");
   }
   DonateBeanTester.tearDown();
}
```

> **Behind the scenes:**
>
> - Initialize the session bean instance by calling the JUnit initialization method (which looks up the EJB directly from OpenEJB).
>
>   ```
>   DonateBeanTester.init();
>   ```
>
> - Verify that the appropriate arguments were passed to the `main` method:
> - Call the `donatetoFund` method and print out the results:
>
>   ```
>   System.out.println("Donation result = " +
>      donateBean.donateToFund(employeeId, fundName, hours));
>   ```

2. In the Project Explorer, right-click **DonateBeanTester.java** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **Run Configurations**.
3. In the Create, manage, and run configurations pop-up, perform the following actions:
   – In the left pane, right-click **Java Application** and select **New**.
   – In the right pane, select the **Arguments** tab and:
      - Set Program arguments to **1 DonationFund 1**
      - Click **Apply** to save the changes.
      - Click **Run** to execute the standalone client.



Name: DonateBeanTester (1)

Program arguments: 1 DonationFund 1

Working directory: ● Default: ${workspace_loc:DonateEJB_Tester}

4. The Java application results are displayed in the Console view:

```
DonateBeanTester: Start
Employee Number = 1
Fund            = DonationFund
Donation Amount = 1
Donation result = Transfer successful
TearDown: number of funds=0
DonateBeanTester: End
```

5. You can rerun the client by selecting **Run** → **Run Configurations**, selecting the **Java Application** → **DonateBeanTester (1)**, changing the arguments, clicking **Apply**, and clicking **Run**.

9.8 Explore!

As extra credit, you can run the standalone client from a Java command line:

1. Export the DonateEJB_Tester project to a JAR file:
   - In the Project Explorer, right-click **DonateEJB_Tester** and select **Export**.
   - In the Export/Select pop-up, select **Java** → **JAR file** and click **Next**.
   - In the JAR Export/Jar File Specification, set Jar file to **C:\7639code\appclient\DonateEJB_Tester.jar** and click **Finish**.
2. Export the DonateEJB project to a JAR file by selecting **Export** → **EJB JAR file**, set the name to `C:\7639code\appclient\DonateEJB.jar`, and click **Finish**.
3. Export the DonateJPAEmployee and DonateJPAFund projects each to a JAR file by selecting **Export** → **EJB JAR file**, set the names to `C:\7639code\appclient\DonateJPAEmployee.jar`, and `C:\7639code\appclient\DonateJPAFund.jar`.
4. Run the following command in an operating system prompt:

```
java -cp
DonateEJB_Tester.jar;DonateEJB.jar;DonateJPAEmployee.jar;DonateJPAFund.jar;
C:\ibm\WASCE21\repository\org\apache\openejb\openejb-client\3.0\openejb-cli
ent-3.0.jar;C:\ibm\WASCE21\repository\org\apache\geronimo\specs\geronimo-ej
b_3.0_spec\1.0.1\geronimo-ejb_3.0_spec-1.0.1.jar
donate.test.DonateBeanTester 1 DonationFund 1
```

Note: A sample command file to run the standalone client in a command prompt is provided in `7638code/appclient/run.bat`. The exported JAR files are available in `7638code/appclient/solutionJARs`.

10

Implement core security

In this chapter, we secure Web and Java EE application client access to the application (Figure 10-1).

We define a security realm in the server, and implement groups and users. Then we define declarative security to restrict access to the session EJB and a JSF Web page. Using programmatic security, we hide sensitive salary information in the session EJB and in the JSF Web application. Finally, we update the Java EE application client to force a security login.

10.1 Learn!

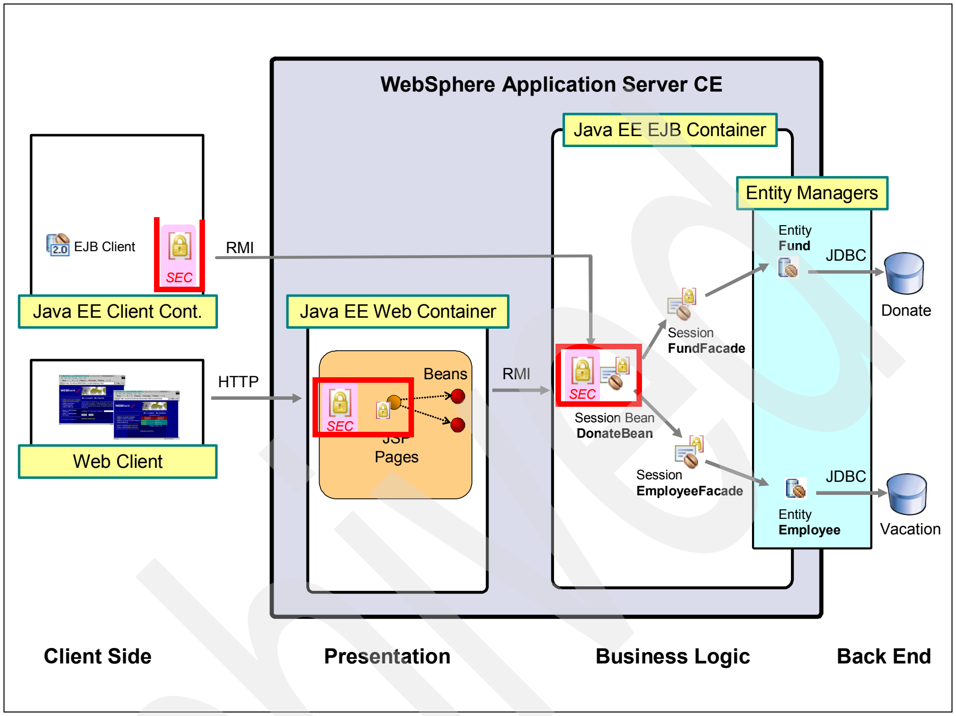


Figure 10-1 Donate application: Java EE application security

Java EE contains a multi-level security architecture for applications. This chapter focus on the following aspects:

- User registry, which defines the valid users and groups within a Java EE environment
- Application role based security, which restricts access to Java EE resources based on roles, which are then mapped to users and groups
- Administrative security

Following are other security aspects that are briefly discussed in the Explore section:

- Single sign-on
- Java 2 Security
- Java Authentication and Authorization Service (JAAS)
- Java Authorization Contract for Containers (JACC)

Additional Learn resources include these:

- The Java EE 5 Tutorial: Chapter 28: Introduction to Security in the Java EE Platform:

  http://java.sun.com/javaee/5/docs/tutorial/doc/bnbwj.html
- The Java EE 5 Tutorial: Chapter 29: Securing Java EE Applications:

  http://java.sun.com/javaee/5/docs/tutorial/doc/bnbyk.html
- The Java EE 5 Tutorial: Chapter 30: Securing Web Applications:

  http://java.sun.com/javaee/5/docs/tutorial/doc/bncas.html
- IBM developerWorks: Configuring Web application security in WebSphere Application Server Community Edition V2.0:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0709_vamsi/0709_vamsi.html

10.1.1 User registry

Any Java EE application server that utilizes application role-based security or administrative security (which is discussed in the subsequent sections) requires a mechanism for authenticating users and groups.

The Java EE mechanism for this type of security is a user registry, and WASCE supports the following to store/access user IDs and passwords:

- Database
- Lightweight Directory Access Protocol (LDAP)
- Properties file
- Certificates (X.509 certificates)

Access to a secured Java EE resource results in a request against the user registry to validate the presented credentials (user ID/password).

Note that the user registry by itself does not enforce any security policies or constraints. It simply validates the supplied user, group, and password information that is presented to it by the Java EE authentication mechanisms.

10.1.2 Application role-based security

Application role-based security allows the developer to secure access to both Web and EJB resources based on roles.

Often the developer has no knowledge of the users and groups that will exist at runtime, so how does the developer refer to the set of users and groups that should have access to a specific artifact?

The answer is through *roles*. Java EE allows the developer to define an arbitrary set of roles and to restrict access to artifacts based on these roles. Later, the developer (or an administrator) can map these roles to actual users and groups.

Java EE has two forms of role-based security:

- **Declarative** security, where the security constraints limiting access are defined in the deployment descriptors, and the Java EE runtime is the mechanism that manages access to the Java EE artifact. Consider this as coarse-grained security, where the user will see all or none of the employee data.
- **Programmatic** security, where the user code queries if a user is in a role, and then executes different statements depending on whether the user is in the role or not. Consider this as fine-grained security, where depending on their role, the user might only see a subset of the employee data.

10.1.3 Administrative security

All Java EE application servers should have secure access to their administrative interfaces (such as the WASCE Console and the **`deploy`** command). In the case of WASCE, security consists of a single security role named `admin`. Any user in the `admin` group in the `geronimo-admin` security realm is mapped to this role.

The installation creates a single administrative user, `system`. Users can add additional users in the WASCE Console by selecting **Security** → **Users and Groups**:

- Create additional users
- Add these additional users to the `admin` group

10.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed these steps:

- Defined the two databases and the matching database pools
- Created the JPA entities and the three session beans
- Created and run the JUnit test cases
- Created the Web applications
- Created the application client

10.3 Create user and group files

In this section you create the users and groups for use in demonstrating application role based security:

Group	User members
DUSERS	DNARMSTRONG. DCBROWN
DMANAGERS	DNARMSTRONG
ALLAUTHENTICATED	DGADAMS, DNARMSTRONG, DCBROWN

We define three groups and three users to implement security using two simple properties files. Security properties files are stored in the directory `<WASCE_HOME>/var/security`. We will implement a security realm that points to these two files. Note that the `var/security` folder contains already two other sets of groups and users:

1. Copy **`C:\7639code\security\experiencejavaeegroups.properties`** into **`C:\IBM\WASCE21\var\security`**

```
## Experience Java EE Groups
##
DUSERS=DCBROWN,DNARMSTRONG
DMANAGERS=DNARMSTRONG
ALLUSERS=DCBROWN,DNARMSTRONG,DGADAMS
```

> **Behind the scenes:**
>
> The `ALLAUTHENTICATED` group exists because one of the Java EE declarative roles is intended to allow all authenticated users to access the resource (Web page).
>
> Some J2EE/Java EE Application servers (such as WebSphere Application Server) allow special role mappings to all authenticated users, so for those servers, this group would not be needed.
>
> WASCE does not support special role mappings such as this, so you must explicitly define a group that contains all users.

2. Copy **`C:\7639code\security\experiencejavaeeusers.properties`** into **`C:\IBM\WASCE21\var\security`**.

   Optionally edit this file and change all occurrences of **`password`** to the password value that you want to use. However, there is no advantage of changing the passwords (our code uses `password`):

```
## Experience Java EE Users
##
DNARMSTRONG=password
DCBROWN=password
DGADAMS=password
```

Behind the scenes:

Passwords can be encrypted, but the encryption used must be that defined in the WASCE security realm. Refer to 10.4, “Configure the security realm”.

10.4 Configure the security realm

What is a security realm? A realm determines the scope of security data. A realm is the region to which a security ID or permission applies. A user defined as John in one realm is treated as different from John in a second realm, even if these two IDs represent the same human user.

A security realm inside of WASCE is a user registry definition, containing users and groups that can map to Java EE roles that protect resources (Web pages, EJBs). Realms can be defined at the server level (and thus are available and visible to all applications) or can be defined as part of an individual application.

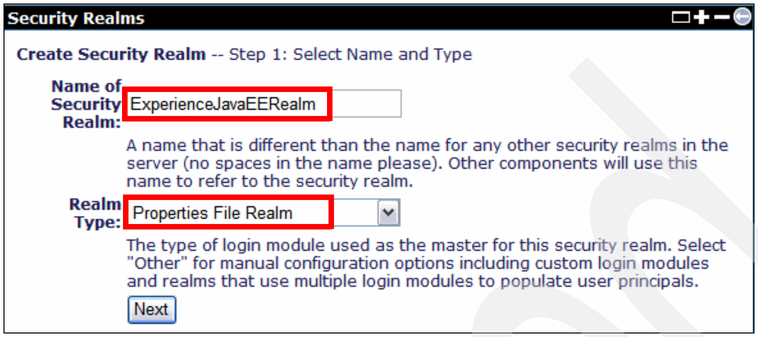
The default WASCE installation contains the geronimo-admin security realm. This realm protects the administrative interfaces, and maps to a properties file based registry. Administrators can choose to use this default registry, or define additional security realms and configure application and application roles against these realms.

One practical reason for using multiple realms is that most production installations will use an LDAP registry instead of the provided file-based registry.

For the purpose of this book, we create a new file-based registry, but we could have easily used the existing geronimo-admin realm instead:

1. Start the WASCE Console (in the Servers view, right-click the **ExperienceJavaEE Server** and select **Launch WASCE Console**).
   – Recall that at the WASCE Console login screen, you use the default **system** user ID and password (**manager**) that the WASCE server was installed with.
2. On the left under Console Navigation, select **Security** → **Security Realms**.
3. Under **Security Realms**, click **Add new security realm**.

4. In the Create Security Realm -- Step 1: Select Name and Type panel, type **ExperienceJavaEERealm** for the name, select **Properties File Realm** for the type, and click **Next**.



Create Security Realm -- Step 1: Select Name and Type

Name of Security Realm: ExperienceJavaEERealm

A name that is different than the name for any other security realms in the server (no spaces in the name please). Other components will use this name to refer to the security realm.

Realm Type: Properties File Realm

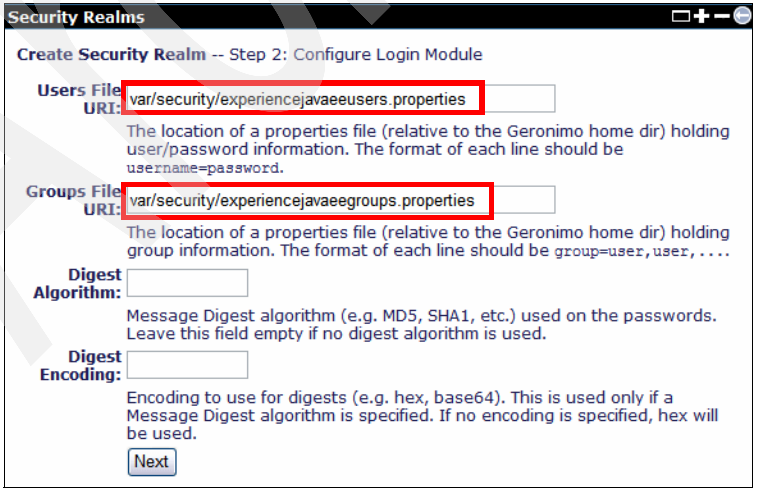
The type of login module used as the master for this security realm. Select "Other" for manual configuration options including custom login modules and realms that use multiple login modules to populate user principals.

Behind the scenes:

Another type of realm would be an LDAP realm, where groups and users are defined in an LDAP server. LDAP would typically be used in a production environment. You can also write your own login module. Here we used simple properties files.

5. In the Create Security Realm -- Step 2: Configure Login Module panel:

   – Users File URI: **`var/security/experiencejavaeeusers.properties`**
   – Groups File URI: **`var/security/experiencejavaeegroups.properties`**
   – Leave Digest Algorithm and Digest Encoding empty.
   – Click **Next**.



Create Security Realm -- Step 2: Configure Login Module

Users File URI: var/security/experiencejavaeeusers.properties

The location of a properties file (relative to the Geronimo home dir) holding user/password information. The format of each line should be `username=password`.

Groups File URI: var/security/experiencejavaeegroups.properties

The location of a properties file (relative to the Geronimo home dir) holding group information. The format of each line should be `group=user,user,....`

Digest Algorithm:

Message Digest algorithm (e.g. MD5, SHA1, etc.) used on the passwords. Leave this field empty if no digest algorithm is used.

Digest Encoding:

Encoding to use for digests (e.g. hex, base64). This is used only if a Message Digest algorithm is specified. If no encoding is specified, hex will be used.

Behind the scenes:

Digest simply means that the password is stored in an encrypted form instead of in clear text:

- **Algorithm** indicates type of encryption that will be used to create the binary representation of the password. The supported algorithms are contained within the SDK/JDK and thus differ depending on the specific SDK/JDK that you are using.

  ```
  IBM Java 5 SDK: MD5OLD, MD2, SHA2, MD4, SHA3, MD5, SHA5, SHA
  Sun Java 5 SDK: SHA-512, MD2, SHA, SHA-256, MD5, SHA-384
  ```

  The algorithms available on your specific SDK/JDK can be determined by running this code fragment in any Java class:

  ```
  Set<String> algs = Security.getAlgorithms("MessageDigest");
  Iterator<String> algI = algs.iterator();
  while (algI.hasNext()) {
     System.out.println(algI.next());
  }
  ```

- **Encoding** indicates the text format that will be used to store the binary representation. The two supported formats are HEX and BASE64.

The IBM developerWorks article *Configuring Web application security in WebSphere Application Server Community Edition V2.0* mentioned in 10.1, “Learn!” on page 286 contains the `encode.jar` utility file that can be used to encrypt an input password, and the encrypted output can replace the clear text password in the user properties file:

```
C:\IBM\WASCE21\var\security>java -jar encode.jar xxxxxxxx MD5 HEX
0b0cfc07fca81c956ab9181d8576f4a8
```

6. In the Create Security Realm -- Step 3: Advanced Configuration panel, click **Skip Test and Show Plan**, or do the extra credit first.

Extra credit:

Click **Test a Login**. Enter a username (in upper case) and the password that you set. Click **Next**. The login should work and display the principals. Click **Show Plan** to continue.

7. In the Create Security Realm -- Show Deployment Plan panel:

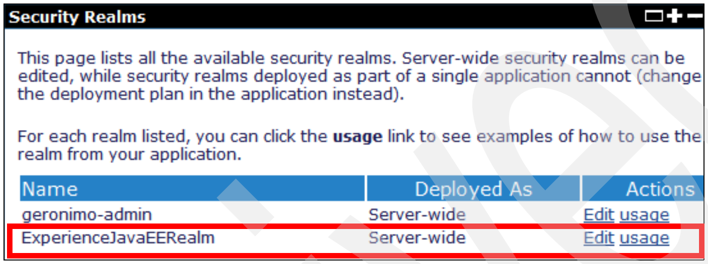
   – Save the deployment plan as `c:\7639code\plans\SecurityRealm-plan.xml`.

   – Click **Deploy Realm**.

Behind the scenes:

The `SecurityRealm-plan.xml` file contains the command script representation of the security realm configuration. We can use this file to recreate the security realm using the **`deploy`** command, as described in step 4 on page 558 in A.1, “Common jump start activities”.

8. The new security realm is now visible in the Security Realms page of the WASCE Console.



This page lists all the available security realms. Server-wide security realms can be edited, while security realms deployed as part of a single application cannot (change the deployment plan in the application instead).

For each realm listed, you can click the **usage** link to see examples of how to use the realm from your application.

Name	Deployed As	Actions
geronimo-admin	Server-wide	Edit usage
ExperienceJavaEERealm	Server-wide	Edit usage

9. Click **Usage** to get instructions on how to configure security in Web applications. Then click **Logout**, and close the WASCE Console.

Behind the scenes:

Deploying the realm added one line to the WASCE `config.xml` file (in `C:\IBM\WASCE21\var\config`):

```
<module name="console.realm/ExperienceJavaEERealm/1.0/car"/>
```

In addition, the security realm configuration information is stored in `C:\IBM\WASCE21\repository\console\realm\ExperienceJavaEERealm`.

The security realm is inaccessible until it is linked to an application. We do that in several sections:

- 10.5.3, “Test the Donate session EJB declarative security” on page 297
- 10.5.4, “Restrict access to Web pages through declarative security” on page 300
- 10.7.2, “Update the DonateClient Geronimo deployment descriptor” on page 315

10.5 Implement declarative security

Declarative security restricts access to URLs (servlets, JSPs, or even HTTP files or images served by the Web container) or EJBs. This is sometimes called coarse grained access because access is either allowed or disallowed:

- URL access is controlled through deployment descriptors in the Web application
- EJB access is controlled through annotations in the EJB class or through deployment descriptors in the EJB application.

The mapping of specific users and groups to the roles is defined in deployment descriptors, and this can be in the Web, EJB, or EAR deployment descriptor.

10.5.1 Configure DonateEAR roles and role mapping

In this section, we configure the overall set of roles and mappings to users and groups in the DonateEAR enterprise application.

1. Update the DonateEAR Geronimo application deployment descriptor:
   - In the Project Explorer, open **geronimo-application.xml** (in `DonateEAR/EarContent/META-INF`).
   - In the resulting Geronimo Deployment Plan Editor, switch to the **Source** tab.
   - Insert a blank line before the closing `</app:application>` tag, and insert the **F52 Donate EAR Role Mapping** snippet from the Snippets view (Example 10-1).

Example 10-1 Security role mapping

```
......
   <security doas-current-caller="true" >
      <role-mappings>
         <role role-name="ALLAUTHENTICATED_ROLE">
            <principal name="ALLUSERS" designated-run-as="true"
               class="org.apache.geronimo.security.realm.providers
                                            .GeronimoGroupPrincipal" />
         </role>
         <role role-name="DMANAGERS_ROLE">
            <principal name="DMANAGERS" designated-run-as="true"
               class="org.apache.geronimo.security.realm.providers
                                            .GeronimoGroupPrincipal" />
         </role>
         <role role-name="DUSERS_ROLE">
            <principal name="DUSERS" designated-run-as="true"
               class="org.apache.geronimo.security.realm.providers
```

```
                                        .GeronimoGroupPrincipal" />
        </role>
      </role-mappings>
    </security>
</app:application>
```

– Save and close `geronimo-application.xml`.

Behind the scenes:

The role mapping binds the Java EE role to the specified users (or groups) in the active security realm for this application.

We could have defined some of this information (the role names), in the Security tab in the Geronimo Deployment Plan Editor, but other information, such as the group mapping, can only be entered in the source tab.

Note that we have not yet linked the ExperienceJavaEERealm security realm to the application. We will do that in other files (and program code) in:

- 10.5.3, “Test the Donate session EJB declarative security” on page 297
- 10.5.4, “Restrict access to Web pages through declarative security” on page 300
- 10.7.2, “Update the DonateClient Geronimo deployment descriptor” on page 315

Important note for deployment failures:

When adding projects to the DonateEAR enterprise application, sometimes the **geronimo-application.xml** file gets overwritten. If publishing to the server fails, always open the `geronimo-application.xml` file and verify:

- The Group Id is set to **ejee**.
- The `<security>` tags are in the file. If these tags are missing, insert the **F52 Donate EAR Role Mapping** snippet before the `</app:application>` tag.

10.5.2 Restrict access to the Donate session EJB (declarative)

In this section, we use declarative security to restrict access to the DonateBean session EJB created in Chapter 7, “Create the Donate session bean for the business logic” on page 209:

1. In the Project Explorer, open **DonateBean** (in DonateEJB/ejbModule/donate.ejb.impl).
2. In the resulting DonateBean editor:
   - Before the DonateBean class definition insert the following line:

     ```
     @DeclareRoles( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
     public class DonateBean implements DonateBeanInterface, ...... {
     ```

     Behind the scenes:

     This annotation statement defines the Java EE roles that will be used for both declarative and programmatic security in this class. Recall that you defined the mapping of these roles to actual users and groups in 10.5.1, “Configure DonateEAR roles and role mapping” on page 294.

   - Before both donateToFund methods insert a @RolesAllowed annotation:

     ```
     @RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
     public void donateToFund(Employee employee, Fund fund, int hours) ...
        ......

     @RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
     public String donateToFund(int employeeId, String fundName, int hours) {
        ......
     ```

     Behind the scenes:

     These annotation statements restrict access to the donateToFund methods to users belonging to the DMANAGES_ROLE and DUSERS_ROLE roles. Thus, user DADAMS will no longer be authorized to run these methods.

   - Organize imports (**Ctrl+Shift+O**).
   - Save and close DonateBean.java.

Deployment descriptor alternatives:

We inserted three annotation statements into the session EJB:

```
@DeclareRoles( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
@RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
```

These statements could be replaced by the following statements in **ejb-jar.xml** (in `DonateEJB/ejbModule/META-INF`) immediately before the closing `</ejb-jar.xml>` tag:

```
<assembly-descriptor>
   <security-role>
      <role-name>DMANAGERS_ROLE</role-name>
   </security-role>
   <security-role>
      <role-name>DUSERS_ROLE</role-name>
   </security-role>
   <method-permission>
      <role-name>DUSERS_ROLE</role-name>
      <role-name>DMANAGERS_ROLE</role-name>
      <method>
         <ejb-name>DonateBean</ejb-name>
         <method-name>donateToFund</method-name>
      </method>
   </method-permission>
</assembly-descriptor>
```

The potential value of using the deployment descriptor instead of the annotations is that the security definitions can be changed at deployment time, without having access to the source code. Deployment descriptors always overwrite annotations.

10.5.3 Test the Donate session EJB declarative security

We can test declarative security for the session bean using our JUnit test case:

1. If the DonateEAR application is not deployed to the server, deploy it now (using **Add and Remove Projects**).
2. In the Servers view, right-click **ExperienceJavaEE Server**, and select **Publish**.
3. Prepare the JUnit test case in the DonateEJB_Tester project:
   - Right-click the DonateEJB_Tester project and select **Properties**.
   - In the resulting pop-up, on the left select **Java Build Path**, and on the right select the **Libraries** tab.

- Click **Add External JARs**, and select the `geronimo-security-2.1.1.jar` (from `C:\IBM\WASCE21\repository\org\apache\geronimo\framework\geronimo-security\2.1.1`).
- Click **OK** to save and close.

Behind the scenes:

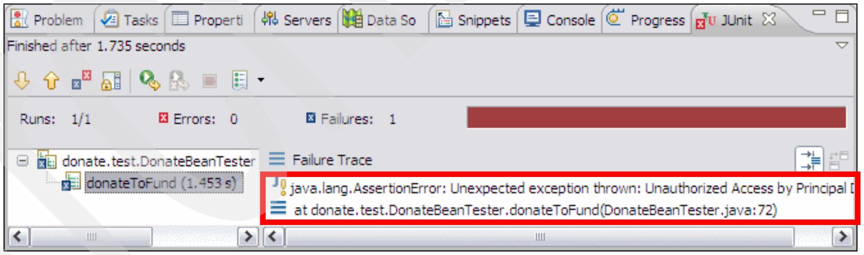
This JAR file contains classes that are returned to the client when initializing to the server with a user ID identity.

Without this JAR file, the execution of the updated `DonateBeanTester` (with a user ID and password) will fail in step 6 with the following error:

```
java.lang.ClassNotFoundException:
        org.apache.geronimo.security.SubjectId
```

4. Verify that the access fails with the unaltered JUnit test, which accesses the `DonateBean` as unauthenticated.
   - In the Project Explorer, right-click **DonateBeanTester.java** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.
   - In the JUnit view, verify that the `donateToFund` fails with the following error:

```
java.lang.AssertionError: Unexpected exception thrown:
                                Unauthorized Access by Principal Denied
    at donate.test.DonateBeanTester.donateToFund(...)
```



Finished after 1.735 seconds

Runs: 1/1 Errors: 0 Failures: 1

donate.test.DonateBeanTester
donateToFund (1.453 s)

Failure Trace

```
java.lang.AssertionError: Unexpected exception thrown: Unauthorized Access by Principal D
at donate.test.DonateBeanTester.donateToFund(DonateBeanTester.java:72)
```

Off course?

If the access succeeds, then the method is not protected by a @RolesAllowed annotation. Verify the changes made in 10.5.2, “Restrict access to the Donate session EJB (declarative)” on page 296.

5. Change DonateBeanTester to access the DonateBean as DNARMSTRONG.
   - In the Project Explorer, open **DonateBeanTester.java**.
   - In the init method, insert blank lines before:

     ```
     ctx = new InitialContext(props)
     ```

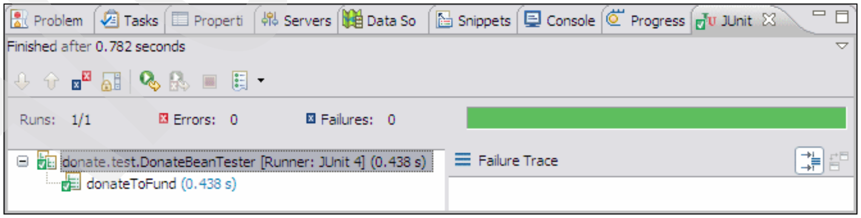
   - Insert the **F53 DonateBeanTester Security** snippet (Example 10-2).

Example 10-2 Security properties for JUnit test

```
public static void init() throws Exception {
   Hashtable props = new Hashtable();
   props.put("java.naming.factory.initial",
            "org.openejb.client.RemoteInitialContextFactory");

   props.put(Context.SECURITY_PRINCIPAL, "DNARMSTRONG");
   props.put(Context.SECURITY_CREDENTIALS, "password");
   props.put("openejb.authentication.realmName","ExperienceJavaEERealm");

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

   - Organize imports (**Ctrl+Shift+O**) to resolve javax.naming.Context.
   - Change the "password" to match the password that you used for DNARMSTRONG in 10.3, "Create user and group files" on page 289.
   - Save and close DonateBeanTester.java.

6. Verify that the access succeeds with the updated JUnit test.
   - In the Project Explorer, right-click **DonateBeanTester** (in DonateEJB_Tester/src/donate.test) and select **Run As → JUnit Test**.
   - In the JUnit view, verify that the call to donateToFund succeeded.



Finished after 0.782 seconds

Runs: 1/1 Errors: 0 Failures: 0

donate.test.DonateBeanTester [Runner: JUnit 4] (0.438 s)
donateToFund (0.438 s)

Off course?

If the access fails, then the method does not contain the DUSERS_ROLE or the DMANAGERS_ROLE in one of the @RolesAllowed annotations in the DonateBean session EJB.

Extra credit:

Change the user ID and password in the `Context.SECURITY_*` properties in the `DonateBeanTester` JUnit test case to `DGADAMS`, and verify that the access fails (because `DGADAMS` does not belong to `DUSERS_ROLE` nor `DUSERS_MANAGERS`).

10.5.4 Restrict access to Web pages through declarative security

In this section, we implement Web declarative security to restrict access to the Web pages created in Chapter 8, “Create the Web front end” on page 221:

1. Update the DonateJSFWeb Java EE Web deployment descriptor to define the security realm.
   - In the Project Explorer, open **web.xml** (in `DonateJSFWeb/WebContent/WEB-INF`).
   - In the editor, switch to the **Source** tab.
   - Insert a blank line before the closing `</web-app>` tag, and insert the **F54 JSF Web Login Config** snippet from the Snippets view (Example 10-3).

Example 10-3 Login configuration for the Web application

```
......
   <login-config>
       <auth-method>BASIC</auth-method>
       <realm-name>ExperienceJavaEERealm</realm-name>
   </login-config>
</web-app>
```

Behind the scenes:

These statements link any login prompts to the ExperienceJavaEERealm defined in 10.4, “Configure the security realm” on page 290 using the BASIC authentication method.

The **BASIC** authentication mechanism and is the default option for requesting authentication information from a browser-based client. The authentication information is encrypted using base64 encoding.

Other options include these:

- **FORM**: Requests the authentication information through a customized logon page provided by the application developer.
- **Client_CERT**: Uses digital certificates passed over an SSL connection.
- **Digest**: Similar to BASIC, but the password is transmitted using a custom encryption mechanism.

Additional information is available in the SUN J2EE tutorial in the section *Understanding Login Authentication* at the following Web site:

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security5.html#wp182253

– Insert a blank line before the closing `</web-app>` tag, and insert the **F55 JSF Web Roles** snippet from the Snippets view (Example 10-4).

Example 10-4 Security roles for the Web application

```
......
   <security-role>
      <role-name>ALLAUTHENTICATED_ROLE</role-name>
   </security-role>
   <security-role>
      <role-name>DMANAGERS_ROLE</role-name>
   </security-role>
   <security-role>
      <role-name>DUSERS_ROLE</role-name>
   </security-role>
</web-app>
```

Behind the scenes:

These are the same roles as defined in the DonateEAR deployment descriptor, but in the context of the Web application.

– Insert a blank line before the closing `</web-app>` tag, and insert the **F56 JSF Web Constraint 1** snippet from the Snippets view (Example 10-5).

Example 10-5 Security constraint for the FindEmployee page

```
......
   <security-constraint>
      <display-name>Secure Base Page</display-name>
      <web-resource-collection>
```

```
         <web-resource-name>Base Resources</web-resource-name>
         <url-pattern>/faces/FindEmployee.jsp</url-pattern>
         <http-method>GET</http-method>
         <http-method>PUT</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <description>
         </description>
         <role-name>ALLAUTHENTICATED_ROLE</role-name>
      </auth-constraint>
   </security-constraint>
</web-app>
```

Behind the scenes:

These statements define the security constraint for the initial page (`FindEmployee.jsp`). The key elements are:

- Definition of which page and HTTP methods to secure:

  ```
  <url-pattern>/faces/FindEmployee.jsp</url-pattern>
  <http-method>GET</http-method>
  <http-method>PUT</http-method>
  <http-method>POST</http-method>
  ```

- Definition of which role(s) have access to the URL/methods:

  ```
  <role-name>ALLAUTHENTICATED_ROLE</role-name>
  ```

– Insert a blank line before the closing `</web-app>` tag, and insert the **F57 JSF Web Constraint 2** snippet from the Snippets view (Example 10-6).

Example 10-6 Security constraint for EmployeeDetails page

```
......
   <security-constraint>
      <display-name>Secure Result Pages</display-name>
      <web-resource-collection>
         <web-resource-name>Result Resources</web-resource-name>
         <url-pattern>/faces/EmployeeDetails.jsp</url-pattern>
         <url-pattern>/faces/DonationConfirmation.jsp</url-pattern>
         <http-method>GET</http-method>
         <http-method>PUT</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <description>
         </description>
            <role-name>DUSERS_ROLE</role-name>
```

```
            <role-name>DMANAGERS_ROLE</role-name>
      </auth-constraint>
   </security-constraint>
</web-app>
```

Behind the scenes:

Similar to the previous step, these statements restrict access to the `EmployeeDetails.jsp` and `DonationConfirmation.jsp` pages to users belonging to the `DUSERS_ROLE` and `DMANAGERS_ROLE`.

– Save and close `web.xml`.

2. Update the DonateJSFWeb Geronimo Web deployment descriptor:

   – In the Project Explorer, open **geronimo-web.xml** (in `DonateJSFWeb /WebContent/WEB-INF`).

   – In the editor, switch to the **Source** tab.

   – Before the closing `</web-app>` tag, add the following line to define the login properties (**F58 JSF Web Security Realm Geronimo** snippet)):

```
   ......
   <security-realm-name>ExperienceJavaEERealm</security-realm-name>
</web-app>
```

Behind the scenes:

This statement links the Web application security to the ExperienceJavaEERealm defined in 10.4, "Configure the security realm" on page 290 using the BASIC authentication method.

This statement does duplicate information that you added to the Java EE Web deployment descriptor back in step 1 on page 300. However, if you omit this statement, publishing fails with an error similar to this:

```
org.apache.geronimo.common.DeploymentException: web.xml for web
app DonateJSFWeb.war includes security elements but Geronimo
deployment plan is not provided or does not contain
<security-realm-name> element necessary to configure security
accordingly.
```

   – Save and close `geronimo-web.xml`.

10.5.5 Test the Web declarative security

In this section you verify that the security mechanisms operate correctly. Table 10-1 summarizes the assignments that you made in the previous sections and links the Java EE roles to the file registry groups and the file registry users.

Table 10-1 Summary of previous assignments

Java EE role	Group	User members
DUSERS_ROLE	DUSERS	DCBROWN, DNARMSTRONG
DMANAGERS_ROLE	DMANAGERS	DNARMSTRONG
ALLAUTHENTICATED_ROLE	ALLUSERS	DCBROWN, DNARMSTRONG, DGADAMS

Testing occurs through an external browser because the WASCE IDE browser keeps credentials once established, making it difficult to validate security:

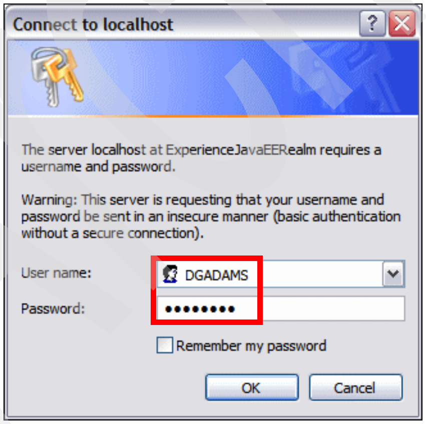
1. Test the access using the DGADAMS user ID:

   – In the Servers view, right-click the **ExperienceJavaEE Server** and select **Publish**.

   – Open a browser and type the following Web address:

   `http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp`

   – In the Connect to localhost pop-up, login as DGADAMS using the password defined in 10.3, "Create user and group files" on page 289.



Connect to localhost

The server localhost at ExperienceJavaEERealm requires a username and password.

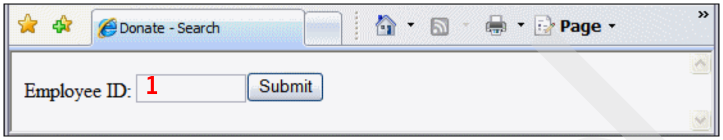
Warning: This server is requesting that your username and password be sent in an insecure manner (basic authentication without a secure connection).

User name: DGADAMS

Password: ••••••••

□ Remember my password

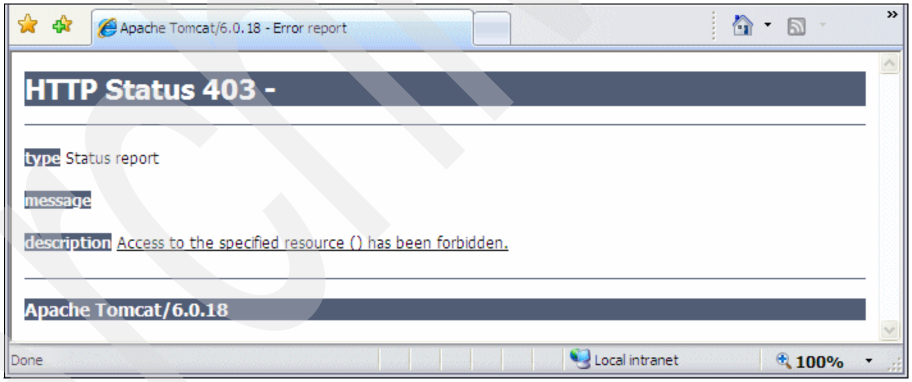
– The initial access to FindEmployee.jsp succeeds because the declarative security for this page allows access by anyone belonging to the ALLAUTHENTICATED_ROLE, and the group ALLUSERS is assigned to that role (and all users are assigned to the ALLUSERS group).



Off course?

If you do not see the login pop-up, perform the following troubleshooting task. Ensure that you correctly created the Web declarative security in 10.5.4, “Restrict access to Web pages through declarative security” on page 300 for the base pages.

2. Enter a valid employee number (such as 1) in the Employee ID field, and retrieve an employee record (using DGADAMS) by clicking **Submit**. The following failure message appears in the browser.



HTTP Status 403 -

type Status report

message

description Access to the specified resource () has been forbidden.

Apache Tomcat/6.0.18

The access to the secondary elements, in this case the EmployeeDetails.jsp page, fails because that resource allows access by anyone belonging to the DUSERS_ROLE and DMANAGERS_ROLE roles, and DGADAMS is assigned to neither of these roles.

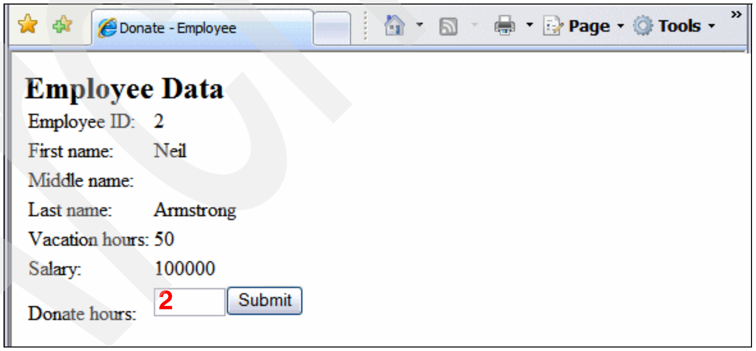
Off course?

If this access works, and you in fact see the `EmployeeDetails.jsp` page, ensure that you correctly created the Web declarative security in 10.5.4, “Restrict access to Web pages through declarative security” on page 300 for the Result pages.

3. Test the access using the `DNARMSTRONG` user ID:
   - Close the browser.
   - Open a new browser and use the same Web address:

     `http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp`
   - Login as `DNARMSTRONG`.

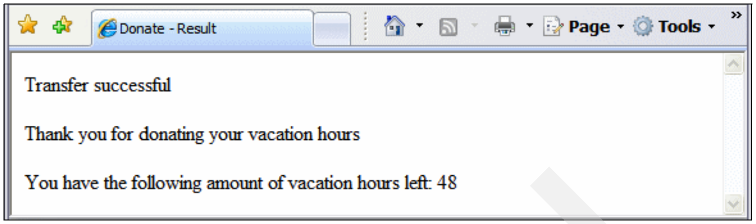
     As previously, the initial access to `FindEmployee.jsp` succeeds because the declarative security for this page allows access by anyone belonging to the `ALLAUTHENTICATED_ROLE` role.
4. Test access to the `DonationConfirmation` page.
   - Enter an employee number of **2**, and click **Submit**. The access to the secondary element (`EmployeeDetails.jsp`) succeeds because this resource allows access by anyone belonging to the `DUSERS_ROLE` and `DMANAGERS_ROLE` roles, and `DNARMSTRONG` is assigned to the `DMANAGERS` group that is assigned to the `DMANAGERS_ROLE` role.



Employee Data

Employee ID:	2
First name:	Neil
Middle name:	
Last name:	Armstrong
Vacation hours:	50
Salary:	100000
Donate hours:	2

   - Type a Donation hours amount (for example, **2**), and click **Submit**. The access to the secondary elements (the `DonateBean` session EJB and `DonationConfirmation.jsp`) succeeds because they both allow access by anyone belonging to the `DUSERS_ROLE` and `DMANAGERS_ROLE` role, and `DNARMSTRONG` is assigned to both of these roles.



Transfer successful

Thank you for donating your vacation hours

You have the following amount of vacation hours left: 48

Off course?

If you receive an error on the donation action, ensure that you correctly created the EJB declarative security in 10.5.2, “Restrict access to the Donate session EJB (declarative)” on page 296.

Extra credit:

Optionally try the foregoing steps using the `DCBROWN` user ID. The access should succeed because `DCBROWN` belongs to the `DUSERS_ROLE` role.

10.6 Implement and test programmatic security

Programmatic security is implemented in user code. It references the role definitions in the deployment descriptors:

- To illustrate programmatic security in a session EJB, we hide the employee salary for non-managers in the `EmployeeFacade` session EJB.
- To illustrate programmatic security in a Web page, we hide the display of the salary for non-managers in the `EmployeeDetails.jsp`.

10.6.1 Add clone method to the Employee entity

We want to mask (hide) the salary of an employee to non-managers. When we retrieve an employee object in the `EmployeeFacade`, we have to ensure that masking the salary is not propagated to the back-end database.

For example, if we change this field directly in the employee object in the `findEmployeeById` method, the change would be automatically committed to the database.

We have several options:

- Create a new object, for example, `EmployeePrime`, and copy all the fields.
- In the `EmployeeFacade`, instantiate a new `Employee` object and copy all the fields.
- In the `Employee` class, add a `clone` method (to create a copy of the object). Then, in the `EmployeeFacade`, call this method to create a copy and mask/null the one field (salary). This involves changes in two methods, but the least amount of code.

Here, we use the `clone` method:

1. In the Project Explorer, open **Employee** (in `DonateJPAEmployee/src /vacation.entities`).
2. In the `Employee` editor:
   - Add the **Cloneable** interface to the class definition:

     ```
     public class Employee implements Serializable, Cloneable {
     ```

   - Implement the `clone` method (required by the interface) by adding the **F61 Employee Clone Method** snippet from the Snippets view after the class definition.

     ```
     public class Employee implements Serializable, Cloneable {

        public Employee clone() throws CloneNotSupportedException {
           return (Employee) super.clone();
        }
        ...
     ```

   - Save and close `Employee.java`.

Behind the scenes:

The `Cloneable` interface copies the entire content of the original object to the new object. This new object is not linked to the original object, so any changes to this object are not reflected in the back-end database.

10.6.2 Implement programmatic security in the EmployeeFacade

In this section, we update the `EmployeeFacade` session EJB to restrict access to the salary field to members of the `DMANAGERS_ROLE` role:

1. In the Project Explorer, open **EmployeeFacadeBean** (in `DonateEJB /ejbModule/vacation.ejb.impl`).
2. In the `EmployeeFacadeBean` editor:
   - Add the following line above the class definition to define the `DMANAGERS` role within this class:

```
@Stateless
@ApplicationException(rollback = true)
@DeclareRoles( { "DMANAGERS_ROLE" })
public class EmployeeFacade implements EmployeeFacadeInterface, ... {
   ......
```

   - Add the following lines below the class definition to define the session context object that will be used to determine if the user is in the `DMANAGERS_ROLE` role:

```
public class EmployeeFacade implements EmployeeFacadeInterface,
                                       EmployeeFacadeRemote {
   @Resource
   private SessionContext ctx;
```

   - Locate the `findEmployeebyId` method, comment the `return` statement and add the **F62 EmployeeFacade Salary Hide** snippet from the Snippets view:

```
public Employee findEmployeeById(int id) throws Exception {
   Employee employee = em.find(Employee.class, id);
   if (employee == null)
      throw new Exception("Employee with id " + id + " not found");
   // return employee;
   System.out.println("EmployeeFacade: User id="
                     + ctx.getCallerPrincipal().getName());
   System.out.println("EmployeeFacade: isCallerInRole="
                     + ctx.isCallerInRole("DMANAGERS_ROLE"));
   Employee clone = employee.clone();
   if (!ctx.isCallerInRole("DMANAGERS_ROLE"))
      clone.setSalary(null);
   return clone;
}
```

   - Format the code (**Ctrl+Shift+F)** and organize imports (**Ctrl+Shift+O**).
   - Save and close `EmployeeFacade.java`.

Behind the scenes:

The salary field is cleared from the employee object if the user does not belong to the DMANAGERS_ROLE.

Note that this has implications in how the returned employee object is used. Because the salary field has been cleared (null), any future calls that use this object would store the null value in the database.

For example, the current addHours method in the EmployeeFacade is coded correctly to avoid propagating a null salary field to the database:

```
public Employee addHours(Employee employee, int hours) {
   employee = em.find(Employee.class, employee.getEmployeeId());
   employee.setVacationhours(employee.getVacationhours() + hours);
   em.merge(employee);
   return employee;
}
```

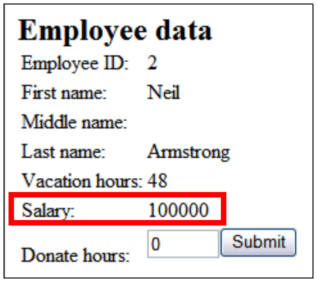
10.6.3 Test programmatic security in a session EJB

We use the JSF Web application to test programmatic security in the session bean:

1. Publish the DonateEAR enterprise application (in the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**).
2. Test the access using the DNARMSTRONG user ID:
   - Open a new browser, and use the following URL:

     http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp
   - Log in as DNARMSTRONG.
   - In the resulting page, retrieve record 2.

   Note that salary information is visible because DNARMSTRONG belongs to the DMANAGERS group, which is assigned to the DMANAGERS_ROLE role that has visibility into this information.



Employee data

Employee ID: 2

First name: Neil

Middle name:

Last name: Armstrong

Vacation hours: 48

Salary: 100000

Donate hours: 0

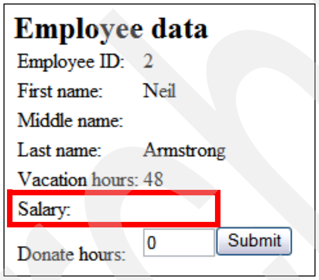
3. Test the access using the DCBROWN user ID:

   – Close the browser, and open a new browser with the same URL:

   – Log in as DCBROWN.

   – In the resulting page, retrieve record 2.

   Note that salary information is masked because DCBROWN belongs to the DUSERS group, which is assigned to the DUSERS_ROLE role, and this role does not have visibility to this information.



Employee data

Employee ID: 2

First name: Neil

Middle name:

Last name: Armstrong

Vacation hours: 48

Salary:

Donate hours: 0

4. Close the browser.

5. Notice the test output in the Console:

```
EmployeeFacade: User id=DNARMSTRONG
EmployeeFacade: isCallerInRole=true
EmployeeFacade: User id=DCBROWN
EmployeeFacade: isCallerInRole=false
```

10.6.4 Implement programmatic security in a Web application

The EJB programmatic security eliminates the value of the salary field for non-authorized users, but the front-end JSPs still contain the name of the field.

We can use programmatic security in the `EmployeeDetails.jsp` to eliminate the display of the salary field for non-authorized users. This helps address the security concern where knowledge of the field is considered to be a security concern, in addition to the actual values:

1. In the Project Explorer, open **DonateHoursManagedBean** (in `DonateJSFWeb /Java Resources: src/donate.jsf.bean`).
2. In the `DonateHoursManagedBean` editor:
   – Add an `isManager` method to the end of the class, using the **F63 Donate Managed Bean isManager** snippet from the Snippets view:

```
public boolean isManager() {
   FacesContext context = FacesContext.getCurrentInstance();
   return context.getExternalContext().isUserInRole("DMANAGERS_ROLE");
}
```

   – Save and close `DonateHoursManagedBean.java`.

Behind the scenes:

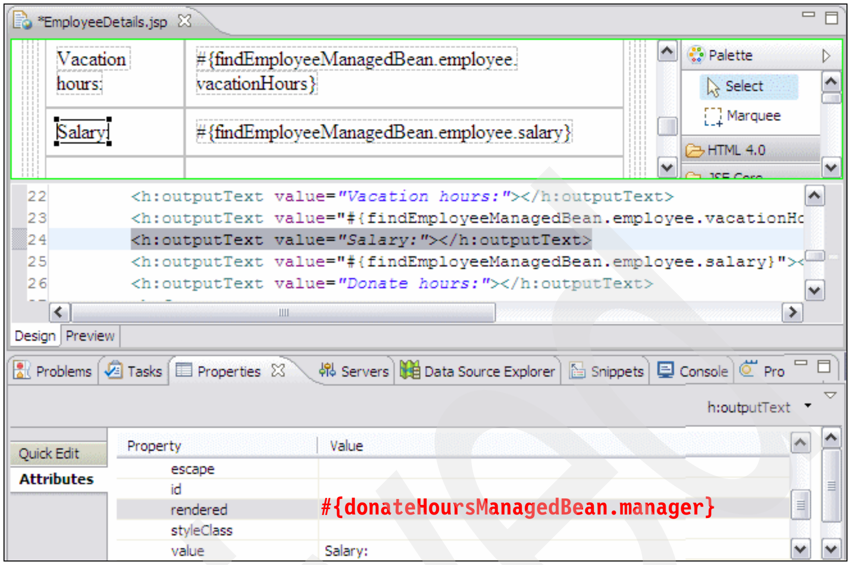
Similar to the `isCallerInRole` method used with EJBs, the `isUserInRole` checks to see if the user is in the specified role and returns the appropriate true or false value.

This method is put into a Faces managed bean because the page does not have direct access to the `FacesContext` instance that contains this method.

3. In the Project Explorer, right-click **EmployeeDetails.jsp** (in `DonateJSFWeb /WebContent`) and select **Open With → Web Page Editor**.
4. In the `EmployeeDetails.jsp` editor, Design tab:
   – Click on **Salary:** to locate the source code:

```
<h:outputText value="Salary:"></h:outputText>
<h:outputText value="#{findEmployeeManagedBean.employee.salary}">
   </h:outputText>
```

   – Go to the Properties view, and select the **Attributes** tab.



Vacation hours:	#{findEmployeeManagedBean.employee.vacationHours}
Salary:	#{findEmployeeManagedBean.employee.salary}

```
<h:outputText value="Vacation hours:"></h:outputText>
<h:outputText value="#{findEmployeeManagedBean.employee.vacationHc
<h:outputText value="Salary:"></h:outputText>
<h:outputText value="#{findEmployeeManagedBean.employee.salary}"><
<h:outputText value="Donate hours:"></h:outputText>
```

Property	Value
escape	
id	
rendered	#{donateHoursManagedBean.manager}
styleClass	
value	Salary:

- Select the **rendered** property, and for the value type:

  ```
  #{donateHoursManagedBean.manager}
  ```

  This is a reference to the `isManager` method in the managed bean. Press **Enter** to make the value appear in the source.

- This value changes the source to:

  ```
  <h:outputText value="Salary:"
     rendered="#{donateHoursManagedBean.manager}"></h:outputText>
  ```

- Repeat this for the output field of the salary:

  ```
  <h:outputText value="#{findEmployeeManagedBean.employee.salary}"
     rendered="#{donateHoursManagedBean.manager}"></h:outputText>
  ```

- Save and close `EmployeeDetails.jsp`.

Behind the scenes:

If the caller belongs to the `DMANAGERS_ROLE` role, then the salary field is displayed as before. If the user does not belong to the `DMANAGERS_ROLE` role, then the salary field is skipped.

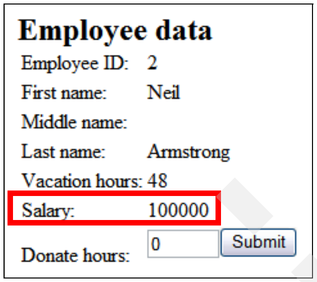
The Faces `rendered` property binding of `#{managedbean.<someproperty>}` indicates that a method named `is<SomeProperty>` is called to determine if the field is displayed. In our case, `isManager` in `DonateHoursManagedBean` is invoked.

10.6.5 Test programmatic security in a Web application

To test the programmatic security in the Web application, repeat the test described in 10.6.3, “Test programmatic security in a session EJB” on page 310:

1. Publish the application in the server.
2. Run the Web application with user ID `DNARMSTRONG` (always close the browser and open a new browser). The salary label and field are displayed.
3. Run the Web application with user ID `DCBROWN` and the salary is not displayed.

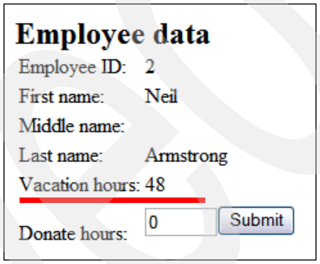
DNARMSTRONG



Employee data

Employee ID: 2
First name: Neil
Middle name:
Last name: Armstrong
Vacation hours: 48
Salary: 100000
Donate hours: 0

DCBROWN



Employee data

Employee ID: 2
First name: Neil
Middle name:
Last name: Armstrong
Vacation hours: 48
Donate hours: 0

10.7 Update the Java EE application client for security

The Java EE application client is subject to the same authentication and authorization requirements as the Web front end.

Here are two ways to supply the authentication information for the application client:

- Through a pop-up prompt, configured in the Geronimo deployment descriptor (our example)
- Through programmatic login

10.7.1 Test the DonateClient (unaltered)

Security is enabled in the EJBs, so the application client should fail when executed:

1. Run the **`client`** command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory:

```
client ejee/DonateClient/1.0/car 1 DonationFund 1
```

2. The command fails with the following output:

```
Employee Number = 1
Fund            = DonationFund
Donation Amount = 1
13:40:56,031 ERROR [CommandLine] Client failed with exception:
javax.ejb.EJBAccessException: Unauthorized Access by Principal Denied
  at org.apache.openejb.core.stateless.StatelessContainer.invoke(Stateless
Container.java:144)
......
```

10.7.2 Update the DonateClient Geronimo deployment descriptor

Normally we would configure an application client to directly utilize the security domain defined within the Java EE application server. However, this does not work with access to EJB resources with WASCE application clients, because the resulting user principal from a standard login is not propagated down to the EJB layer (OpenEJB) when you invoke an EJB session bean. The user identity in the EJB still shows as unauthenticated.

Therefore, as a workaround, we instead configure a login mechanism in the client that logs into OpenEJB, which in turn logs into WASCE:

1. In the Project Explorer, open **geronimo-application-client.xml** (in `DonateClient/appClientModule/META-INF`).
2. In the resulting editor:
   - Switch to the **Source** tab.
   - Insert the **F64 App Client Security Geronimo** snippet from the Snippets view before the closing `</client:application>` tag (Example 10-7).

Example 10-7 Security configuration for application client

```
......
   <client:realm-name>remote-openejb-realm</client:realm-name>
   <client:callback-handler>
      com.ibm.security.auth.callback.DialogCallbackHandler
   </client:callback-handler>
   <gbean name="remote-openejb-realm"
        class="org.apache.geronimo.security.realm.GenericSecurityRealm">
      <attribute name="realmName">remote-openejb-realm</attribute>
      <xml-reference name="LoginModuleConfiguration">
         <lc:login-config
         xmlns:lc="http://geronimo.apache.org/xml/ns/loginconfig-2.0">
            <lc:login-module control-flag="REQUIRED">
               <lc:login-domain-name>remote-openejb-realm
               </lc:login-domain-name>
               <lc:login-module-class>
```

```
                  org.apache.geronimo.openejb.OpenejbRemoteLoginModule
               </lc:login-module-class>
               <lc:option name="RemoteSecurityRealm">
                  ExperienceJavaEERealm</lc:option>
               <lc:option name="ServerURI">
                  ejbd://localhost:4201</lc:option>
            </lc:login-module>
         </lc:login-config>
      </xml-reference>
      <reference name="ServerInfo">
         <name>ServerInfo</name>
      </reference>
   </gbean>
</client:application-client>
```

3. Save and close `geronimo-application-client.xml`.

SDK/JRE selection alert!

The callback handler (`DialogCallbackHandler`) comes with the IBM JDK in `C:\Program Files\IBM\Java50\jre\lib\security.jar`.

If you use the Sun JDK, you have to change the reference to `com.sun.security.auth.callback.DialogCallbackHandler`.

Otherwise, the client execution in the next step will fail with an error similar to:

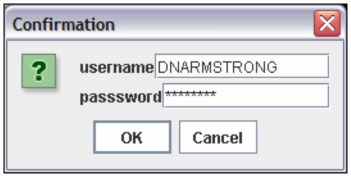
```
13:51:48,250 ERROR [CommandLine] Client failed with exception:
java.lang.InstantiationException: Can't load class
com.ibm.security.auth.callback.DialogCallbackHandler in classloader:
[org.apache.geronimo.kernel.classloader.
JarFileClassLoader id=ejee/DonateClient/1.0/car]
```

10.7.3 Test the updated application client

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.
2. Run the **`client`** command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory:

```
client ejee/DonateClient/1.0/car 1 DonationFund 1
```

3. In the Confirmation pop-up, login as DNARMSTRONG using the password defined in 10.3, “Create user and group files” on page 289.



username DNARMSTRONG
passsword ********

4. The command should succeed with the following output:

```
Employee Number = 1
Fund            = DonationFund
Donation Amount = 1
Donation result = Transfer successful
```

10.8 Explore!

In this section we describe a few Explore activities.

10.8.1 Java EE Security

Java EE security contains many aspects, and addressing all of them would require a separate book (and in fact that was done; see the reference list in the Learn section). Some of the other security aspects include:

- **Java 2 Security**: This is a security mechanism provided by the core J2SE environment to manage an application’s access to operating system resources, such as files and network resources. The policies are defined in a text policy file, based on the calling code (not users), and are enforced by the runtime.

  Note that Java 2 Security grants access. After Java 2 Security is enabled, by default all access is denied unless explicitly granted by a policy file. This is the opposite of Java EE application role based security, where by default access is allowed unless explicitly denied.

- **Java Authorization Contract for Containers (JACC)**: JACC provides a common interface for connecting Java EE application servers with an external authorization provider (such as IBM Tivoli® Access Manager). This allows the Java EE application server authorization definitions to be managed by a centralized authorization system instead of by an internal Java EE unique mechanism.

JACC requires that Java EE containers be configurable to point to an external authorization provider, both for propagating policy information when installing and removing a Java EE application and when making runtime authorization decisions.

Additional Learn resources are available at the following Web addresses:

- WASCE InfoCenter: Implementing single sign-on to minimize Web user authentications:

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.doc/info/aes/ae/tsec_msso.html`

- WASCE InfoCenter: Java 2 Security:

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.doc/info/aes/ae/csec_rsecmgr2.html`

- WASCE InfoCenter: JACC support in WebSphere Application Server:

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.doc/info/aes/ae/csec_jaccsupport.html`

10.8.2 Geronimo deployment descriptor files

In this chapter, we worked quite a bit with Geronimo deployment descriptor files. Why are there IBM (Geronimo) specific binding files?

The Java EE specification is very extensive, but is by no means complete and defers the specification of some areas to the Java EE runtime providers, such as the mapping of security roles to users and groups.

The following excerpt is from *Designing Enterprise Applications with the J2EE Platform, Second Edition*, which you can find at the following Web address:

`http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/deployment/deployment6.html`

> ***7.5.2.1 Vendor-Specific Deployment Information***
>
> *The J2EE platform specification defines deployment unit requirements for each of the four J2EE module types and for a J2EE application itself. Each specification defines how the archive file must be structured to operate correctly with a J2EE deployment tool. In addition to application code and a deployment descriptor, an application requires a certain amount of additional vendor- or environment-specific binding information. For example, when the J2EE reference implementation receives an application EAR file from a deployer, it also needs the following information:*

- A JNDI name for each enterprise bean's home interface
- A mapping of the application's security roles to user and group names
- JNDI lookup names and account information for all databases
- JavaMail™ session configuration information

As a result, the implementation differs between Java EE providers, and the Java EE application must be updated when deploying on different Java EE application servers. For example, the BEA WebLogic server puts the binding information into `weblogic.xml`, `weblogic-ejb.xml`, or `weblogic-application.xml`, and the Sun Java System Application Server uses a similar convention.

Part 3

Web services

In Part 3, we extend the core Java EE application to support Web services:

- Chapter 11, “Create the Web service” on page 323: Use the tooling wizards to expose the DonateBean session EJB as a Web service, and create a Web service client to access this Web service.
- Chapter 12, “Implement security for the Web service” on page 367: Update the Web service and the Web service client to support Web services security using user ID and password for authentication.

Note that WASCE does not support the Web services security (WS-Security) specification.

11

Create the Web service

In this chapter, we discuss Web services and the support provided by Java EE (Figure 11-1).

We develop a bottom-up Web service from the `DonateBean` session bean, and create a Web service client for testing.

We also provide examples of a Web services with complex data types, fault handling, and develop a top-down Web service.

11.1 Learn!

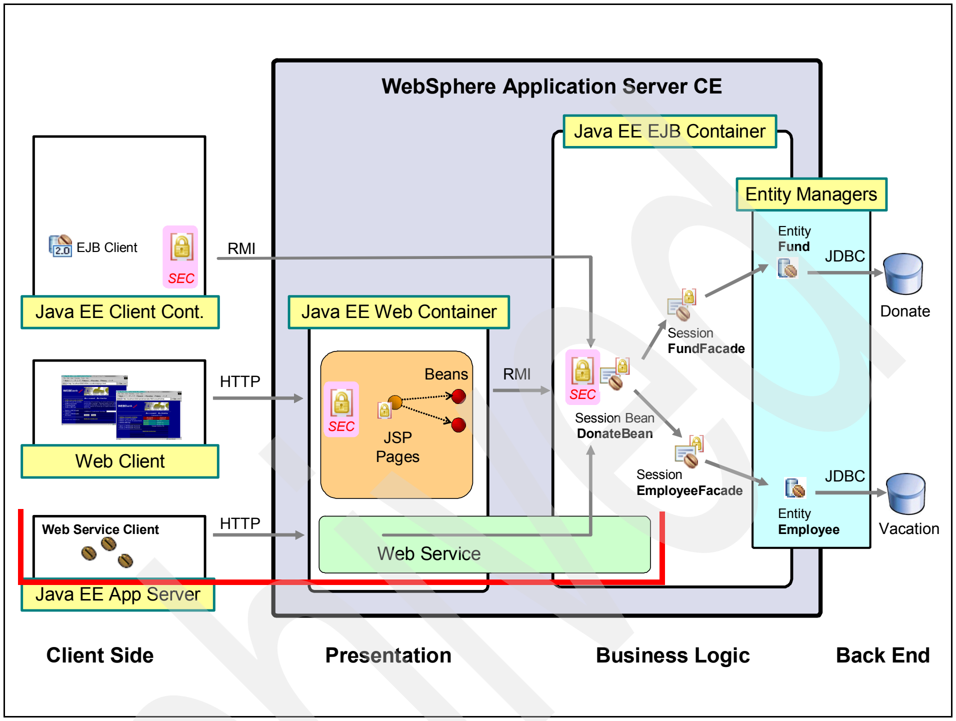


Figure 11-1 Donate application: Web service

Web services provide a standards-based implementation for process-to-process communication between two presumably disparate systems: one providing access to a function through a service, and one consuming this function through a client. Key standards include these:

- Hypertext Transfer Protocol (**HTTP**) is the most common transport protocol to connect the service and the client.

  However, current specifications allow providing services over alternate protocols (other than HTTP) such as messaging, which effectively extends Web services to provide “asynchronous” operation.

- eXtensible Markup Language (**XML**) describes the overall content (being delivered over HTTP) in a structured format that is recognized by both the service and the client.

- **SOAP**[1] messages describe the remote operation and the associated request or response message data in an XML based format. Runtime environments such as Java EE and .NET are the primary producers and consumers of SOAP messages, which are transferred over HTTP.

  Thus, a SOAP message is the runtime instance of a Web service request or response.

- Web Services Description Language (**WSDL**) files describe the remote operation and the associated request or response messages in an XML based format. Development tools are the primary producers and consumers of WSDL documents, which are stored in files.

  Thus, a WSDL file is the development time representation of a Web service.

- Universal Description, Discovery, and Integration (**UDDI**) is an optional component that stores and retrieves WSDL documents. This allows the developers of Web services to publish WSDL documents to a directory, and the developers of Web services clients to search this directory for the desired service.

  UDDI can also be used at runtime to dynamically discover part or all of the information about a Web service:

  – *Partial dynamic* operation is querying the UDDI registry for a list of endpoints that implement the Web service, and then selecting one of these. The endpoint is effectively the Web address for the Web service, updated for the hostname on which the Web service is deployed.

  – *Full dynamic* operation is querying the UDDI registry for a complete WSDL that represents a desired type of service, and then dynamically creating and executing a Web service client, potentially selecting different messages, operations, or transports (HTTP or messaging).

  Practically, most implementations use *partial dynamic* operation. They do so because although *full dynamic* operation is technically feasible, it provides very poor performance (because you generate, compile, and execute a new Web service client each time), it is complex to code to, and it can potentially generate a large number of runtime errors.

  The UDDI specifications supported a publicly accessible Universal Business Registry (UBR) in which a naming system was built around the UDDI-driven service broker. IBM, Microsoft, and SAP announced they were closing their public UDDI nodes in January 2006.

[1] Note that SOAP once stood for *Simple Object Access Protocol*, but this acronym was dropped with Version 1.2 of the standard, as it was considered to be misleading. Version 1.2 became a W3C Recommendation on June 24, 2003. The acronym is sometimes confused with SOA (service-oriented architecture), however SOAP is quite different from SOA.

In many companies, UDDI has been replaced by private registries, such as the IBM WebSphere Service Registry and Repository (WSRR), which helps maximize the business value of your service oriented architecture (SOA). WSRR is an industry leading solution that enables you to easily and quickly publish, find, enrich, manage, and govern services and policies in your SOA

An alternative to UDDI is the Web Services Inspection Language (WSIL), which defines WSDL file locations through XML documents. However, this capability is rarely used because for low-end registry support, most users find it simpler and almost as effective to post their WSDL files on a managed HTTP server.

The technical implementation and capabilities inherent in Web services is not unique, and it has been possible to implement these basic capabilities using off the shelf technologies for years.

However, the value and appeal of Web services in today's environment is the almost universal implementation that enables application developers to implement this capability with minimal coding, supporting access from a wide variety of clients:

- Almost every major application development vendor has tooling to generate Web services and Web services clients.
- Almost every major infrastructure vendor, such as those for application servers and databases, has runtimes that support these Web services and Web service clients.

11.1.1 Web services and Java EE 5

Java EE 5 provides new features for the development of Web services. It allows to use annotations that can set up the Web service or its client with no need of a deployment descriptor. By simply adding the **@WebService** annotation to a session bean or a POJO, it is enough to expose that as a Web service. This allows the developers to easily define and deploy Web services. However, Java EE 5 keeps the advantages of using deployment descriptors, they can be used by a deployer to augment or override the behavior of the annotations.

These new features are ruled by the following specifications:

JAX-WS 2.0 The Java API for XML-Based Web Services (JAX-WS), JSR 224, is the fundamental technology for developing Web services in Java EE 5 and is the evolution of JAX-RPC 1.1.

JAXB 2.0 The mapping between Java artifacts and XML documents are defined by the Java Architecture for XML Binding (JAXB), JSR 222. Differently of the JAX-RPC, JAX-WS 2.0 has delegated the XML binding module to JAXB.

Annotations The Web Services Metadata for the Java Platform, JSR 181, specifies the annotations used to define a Web services and its artifacts.

Java API for XML-Based Web Services (JAX-WS 2.0)

JAX-WS 2.0 (JSR 224) is the core technology for Web services in Java EE 5 specification. The JAX-WS 2.0 programming model simplifies application development by supporting annotations to define Web services applications and clients from simple POJOs, allowing them to be deployed without deployment descriptors.

JAX-WS supports WS-I Basic Profile 1.1, which ensures that Web services developed with the JAX-WS stack can be consumed by any Web service client that adheres to the WS-I Basic Profile standard.

JAX-WS is the evolution of JAX-RPC 1.1 (JSR 101) and Web Services for J2EE (JSR 109). The main difference regarding JAX-RPC is the message-oriented programming model, that replaces the remote procedure call programming model. It allows JAX-WS to be protocol and transport independent, allowing a direct XML/HTTP binding.

Because JAX-RPC is widely used in the industry, JAX-WS maintains backward compatibility with the JAX-RPC 1.1 specifications, with possible exceptions such as the optional SOAP encoding.

Note: The main problem Web services had in its beginning was the development of interoperable services. JAX-RPC was a Java community effort to eliminate this problem and provide a well-known application programming interface for Web service implementations on both the client and server side.

It was broadly adopted in the current Java enterprise servers since 2002. Therefore, JAX-WS backward compatibility is very important to allow new services using the large number of services developed with JAX-RPC.

Table 11-1 provides a list of the main technologies supported by JAX-WS 2.0 and its predecessor JAX-RPC 1.1.

Table 11-1 JAX-RPC versus JAX-WS 2.0 technologies

Technology	JAX-RPC 1.1	JAX-WS 2.0
SOAP	1.1	1.2
XML/HTTP	not supported	supported
WSDL	1.0	1.1

Technology	JAX-RPC 1.1	JAX-WS 2.0
WS-I Basic Profile	1.0	1.1
Annotations	not supported	supported
Mapping model	Internal mapping	Delegated to JAXB 2.0

WebSphere Community Edition 2.1 supports JAX-WS 2.0 Web services. It uses Apache Axis/Axis2 to provide the Web services engine. By using Axis and Axis2, WebSphere Community Edition allows to deploy older Web services implementations.

Additional Learn resources are available at the following Web sites:

- Design and develop JAX-WS 2.0 Web services:

  `http://www.ibm.com/developerworks/edu/ws-dw-ws-jax.html`

- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1:

  `http://www.ibm.com/developerworks/webservices/library/ws-tip-jaxwsrpc.html`

- Java Technology and Web services:

  `http://java.sun.com/javaee/technologies/webservices/`

- JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0:

  `http://jcp.org/en/jsr/detail?id=224`

- JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0:

  `http://jcp.org/en/jsr/detail?id=222`

- JSR 181: Web Services Metadata for the JavaTM Platform:

  `http://jcp.org/en/jsr/detail?id=181`

11.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed all the applications of Part 2, “Core Java EE application” on page 97.

11.3 Disable Java EE EJB declarative security

This chapter does not contain the application code required to support a secure session EJB. Therefore, before starting this chapter, we have to disable the Java EE EJB declarative security. We will re-enable EJB security at the start of Chapter 12, “Implement security for the Web service” on page 367.

1. Disable declarative security in the `DonateBean` session EJB:
   - In the Project Explorer, open **DonateBean** (in `DonateEJB/ejbModule /donate.ejb.impl`).
   - Before the two `donateToFund` methods, comment out the `@RolesAllowed` statement and add a `@PermitAll` annotation.

```
//@RolesAllowed( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
@PermitAll
public String donateToFund(Employee employee, ......)
   ......

//@RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
@PermitAll
public String donateToFund(int employeeId, ......)
```

   - Save and close `DonateBean.java`.

Workaround:

We should not have to add the @PermitAll annotation, because removing the @RolesAllowed annotation should leave the method unprotected.

However, because there are still partial security definitions in this class, access to the Web service that we create fails with the following error, unless you add the @PermitAll annotation:

```
[Axis2WebServiceContainer] Exception occurred while trying to invoke
service method doService()
org.apache.axis2.AxisFault: javax.ejb.EJBAccessException: Unauthorized
Access by Principal Denied: Unauthorized Access by Principal Denied
```

2. We do not test that declarative security is disabled. We could change and run the `DonateBeanTester` JUnit test to verify the changes:

```
//props.put(Context.SECURITY_PRINCIPAL, "DNARMSTRONG");
//props.put(Context.SECURITY_CREDENTIALS, "password");
//props.put("openejb.authentication.realmName","ExperienceJavaEERealm");
```

   However, we will get errors when testing the Web service if the changes were not made properly.

11.4 Create the bottom-up EJB Web service

Web services can be created using two methods: top-down development and bottom-up development. Bottom-up Web service development involves creating a Web service from a Java bean or enterprise bean.

When creating a Web service using a bottom-up approach, first you create a Java bean or EJB and then annotate this Java artifact to expose it as a Web service. JAX-WS 2.0 defines that the Java EE 5 implementation must support mapping Java to WSDL 1.1, which means that the WSDL is automatically generated by the Java EE 5 container.

In this section you create a bottom-up EJB-based Web service from the `Donate` session bean by adding annotations to the bean class. You only expose the method `donateFund` that uses only primitive types as parameters:

1. In the Project Explorer, open **DonateBean** (in `DonateEJB/ejbModule/donate.ejb.impl`).
2. In the editor, add the annotations to the `DonateBean` class (Example 11-1).
   - Before the class definition, add a **@WebService** annotation.
   - Before the `donateToFund(int,String,int)` method, add a **@WebMethod** annotation. Be sure to set it at the second `donateToFund` method.
   - Organize the imports (Ctrl+Shift+O).
   - Save and close `DonateBean.java`.

Example 11-1 Web service annotations for the session bean

```
@Stateless
@ApplicationException(rollback = true)
@DeclareRoles( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
@WebService
public class DonateBean implements DonateBeanInterface, DonateBeanRemote {
   ......

   //@RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
   @PermitAll
   @WebMethod
   public String donateToFund(int employeeId, String fundName, int hours) {
      ...
   }
}
```

Behind the scenes:

The annotations have simplified the complexity to expose session beans or POJO Java classes as Web services. The annotations used to expose the service are:

@WebService The `javax.jws.WebService` annotation can be used on a Java interface or a Java class. When used in a Java class, it defines a Web service. When used in a Java interface, it allows the separation of Web service interface and implementation. If no method is explicitly annotated to be exposed as a Web service, by default all the public methods of the annotated class compose the service endpoint interface (SEI).

@WebMethod The `javax.jws.WebMethod` annotation is used to explicitly define which methods of a class are exposed as Web services.

For example, if the @WebMethod annotation was not added to the `DonateBean` session bean, all its public methods would be exposed as services. Because you have explicitly added the annotation to one method only, that method is defined as a service.

Deployment descriptors versus annotations:

The @WebService annotation is mandatory per JAX-WS 2.0 specifications to expose an EJB session bean as a Web service. Customizations of this annotation, such as Web service endpoint URL, security definitions, the virtual host used by the Web service can also be defined in the OpenEJB deployment descriptor. The deployment descriptor overrides the annotations, allowing the deployer to easily change how the Web service is deployed.

3. Make sure that the ExperienceJavaEE Server is started, and the DonateEAR application is deployed.
4. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.

Behind the scenes:

Per JAX-WS 2.0 specification, the annotations provide the information to the Java EE 5 container to create the mapping from the Java artifact to the WSDL, and to create the bindings. The WSDL file is only created when the application is deployed to a Java EE container. In the WASCE environment, the bindings are created during deployment.

The generation can be monitored in the server logs (by setting it to the `DEBUG` level) as shown in the following messages:

```
14:22:55,980 INFO  [config] Loaded Module: ejee/DonateEAR/1.0/car
14:22:57,081 INFO  [JAXWSServiceBuilder] Configuring EJB JAX-WS Web
Service: DonateBean at /DonateBeanService/DonateBean
```

No Web project is needed to route the HTTP requests to this EJB- based Web service. In JAX-RPC, a Web router project was required.

WASCE implements JAX-WS 2.0 by generating the WSDL file (`DonateBeanService.wsdl`), an XSD file (`DonateBeanService_schema1.xsd`), and two Java helper classes (`DonateToFund` and `DonateToFundResponse`), during the deployment of the application. These files are generated under the deployment directory of the application in the repository folder. The path to the Web services related generated files is:

```
<WASCE_HOME>/repository/ejee/DonateEAR/1.0/DonateEAR-1.0.car/<random>/
```

In that directory you can find:

- `DonateBeanService.wsdl`
- `DonateBeanService_schema1.xsd`
- `donate/ejb/impl/jaxws` with helper classes (`DonateToFund`, ...)

The WSDL file includes a variable for the target address

```
<soap:address location="REPLACE_WITH_ACTUAL_URL"/>
```

5. Open the following URL in a browser to view the WSDL file that was generated by the container during the application deployment. Note that the WSLD file will be rendered differently depending on the browser that you are using. Example 11-2 shows the text-view of the WSDL file:

   `http://localhost:8080/DonateBeanService/DonateBean?wsdl`

Example 11-2 Generated WSDL file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://impl.ejb.donate/"
      name="DonateBeanService"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://impl.ejb.donate/"
      xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://impl.ejb.donate/"
                          schemaLocation="DonateBeanService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="donateToFund">
    <part name="parameters" element="tns:donateToFund"/>
  </message>
  <message name="donateToFundResponse">
    <part name="parameters" element="tns:donateToFundResponse"/>
  </message>
  <portType name="DonateBean">
    <operation name="donateToFund">
      <input message="tns:donateToFund"/>
      <output message="tns:donateToFundResponse"/>
    </operation>
  </portType>
  <binding name="DonateBeanPortBinding" type="tns:DonateBean">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                                             style="document"/>
    <operation name="donateToFund">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="DonateBeanService">
    <port name="DonateBeanPort" binding="tns:DonateBeanPortBinding">
      <soap:address
         location="http://localhost:8080/DonateBeanService/DonateBean"/>
    </port>
  </service>
</definitions>
```

Behind the scenes:

In the WSDL displayed in Example 11-2, we discuss the XSD import, the method exposed, and the service endpoint URL at the end. For example:

- The WSDL imports the XSD file that contains the Java/XML bindings for this service:

```
<xsd:schema>
<xsd:import namespace="http://impl.ejb.donate/"
   schemaLocation="http://localhost:8080/DonateBeanService/DonateBean
                              ?xsd=DonateBeanService_schema1.xsd"/>
</xsd:schema>
```

- The XSD file contains the bindings for each Java artifact annotated to be serialized with JAXB. For example, the binding for the `donateToFund` method is shown here:

```
<xs:complexType name="donateToFund">
   <xs:sequence>
      <xs:element name="arg0" type="xs:long" />
      <xs:element minOccurs="0" name="arg1" type="xs:string" />
      <xs:element name="arg2" type="xs:int" />
   </xs:sequence>
</xs:complexType>
```

- The service endpoint URL is also defined in the WSDL file. The WSDL file on disk contains the string "`REPLACE_WITH_ACTUAL_URL`", and the WSDL loaded in a browser shows the current endpoint URL for that deployed instances of the Web service.

```
<service name="DonateBeanService">
   <port binding="tns:DonateBeanPortBinding" name="DonateBeanPort">
      <soap:address
      location="http://localhost:8080/DonateBeanService/DonateBean"/>
   </port>
</service>
```

Alternative:

The service endpoint in the foregoing WSDL is automatically generated. A custom service endpoint URL can be specified by changing the `openejb-jar.xml` deployment descriptor:

```
<ejb:openejb-jar ......
   xmlns:ejb="http://openejb.apache.org/xml/ns/openejb-jar-2.2">
   <ejb:enterprise-beans>
      <ejb:session>
         <ejb:ejb-name>DonateBean</ejb:ejb-name>
         <ejb:web-service-address>/DonateBean/ejbws
         </ejb:web-service-address>
      </ejb:session>
   </ejb:enterprise-beans>
</ejb:openejb-jar>
```

The Web service endpoint is now:

```
http://localhost:8080/DonateBean/ejbws
```

11.5 Test the Web service using the Web Services Explorer

Eclipse WTP contains a generic test tool called the Web Services Explorer that enables you to load a WSDL file, either within the workspace or available anywhere over HTTP, and manually test the referenced Web service:

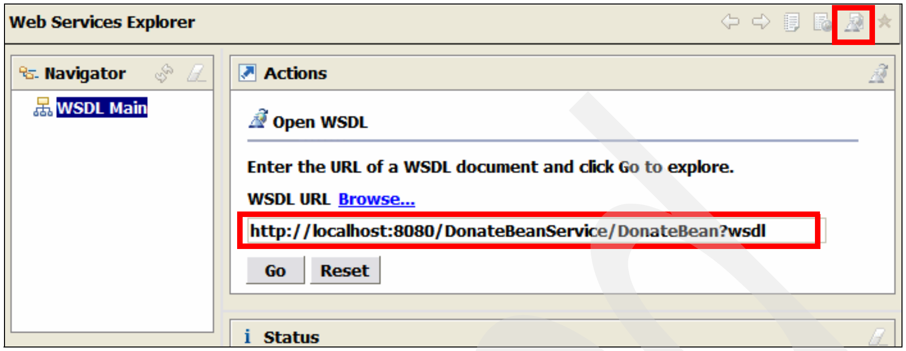
1. In the Java EE perspective, select **Run** → **Launch the Web Services Explorer**.

   Tip: You can use an embedded browser or an external browser (Internet Explorer or Firefox). Select **Window** → **Web Browser** (or **Window** → **Preferences** and from the pop-up select **General** → **Web Browser**) and select your preference.

2. In the Web Services Explorer browser, click the **WSDL Page** icon [ ] in the upper right.
3. On the left side, in the Navigator pane, select **WSDL Main**.

4. On the right side, in the Actions pane, enter the following URL and click **Go**:

```
http://localhost:8080/DonateBeanService/DonateBean?wsdl
```



Enter the URL of a WSDL document and click Go to explore.

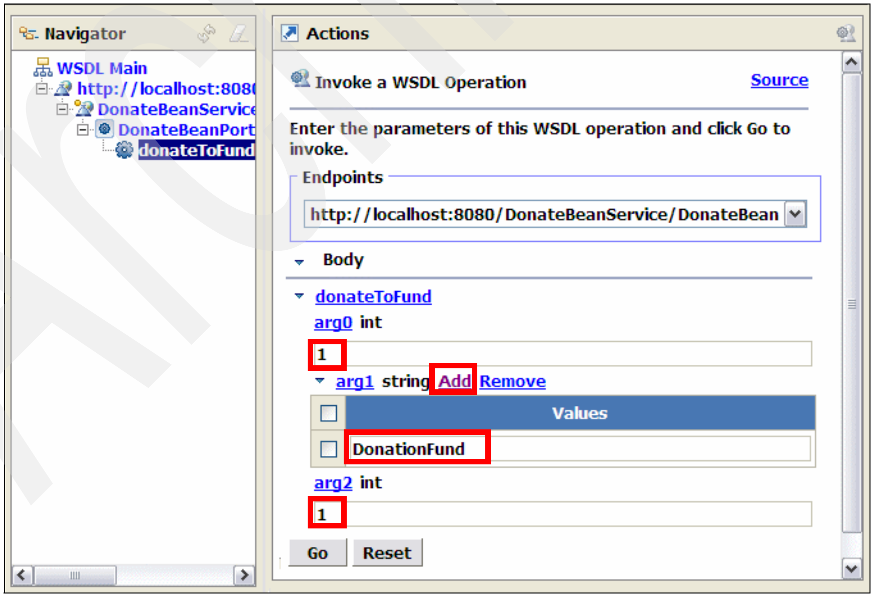
WSDL URL Browse...

http://localhost:8080/DonateBeanService/DonateBean?wsdl

5. In the Navigator pane, select **WSDL Main** → **http://localhost...** → **DonateBeanService** → **DonateBeanPortBinding** → **donateToFund**. You can also select the **donateToFund** operation in the Actions pane.

6. In the Actions pane, which now displays Invoke a WSDL Operation, complete the following fields:

   – arg0 (employee ID): **1**
   – Next to arg1 (fund name): Click **Add**, and type **DonationFund**.
   – arg2 (donation amount): **1**



Enter the parameters of this WSDL operation and click Go to invoke.

Endpoints: http://localhost:8080/DonateBeanService/DonateBean

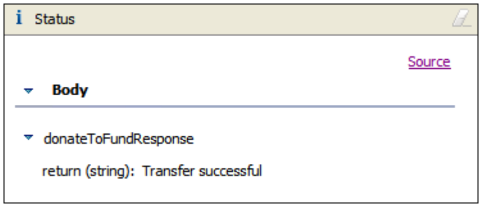
arg0 int: 1

arg1 string Add Remove

□	Values
□	DonationFund

arg2 int: 1

7. Click **Go**. The Web service is invoked and the result is displayed in the Status pane.



Status

Source

Body

donateToFundResponse

return (string): Transfer successful

8. In the Status pane, click **Source** to see the SOAP input and output messages:

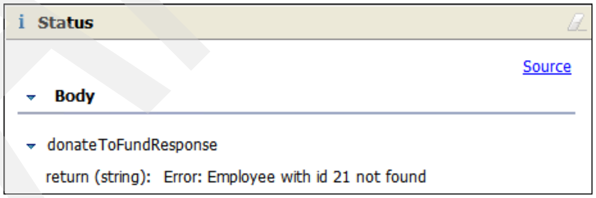
   SOAP Request Envelope:

```
<soapenv:Envelope>
  <soapenv:Body>
    <q0:donateToFund>
      <arg0>1</arg0>
      <arg1>DonationFund</arg1>
      <arg2>1</arg2>
    </q0:donateToFund>
  </soapenv:Body>
</soapenv:Envelope>
```

   SOAP Response Envelope:

```
<soapenv:Envelope>
  <soapenv:Body>
    <donateToFundResponse>
      <return>Transfer successful</return>
    </donateToFundResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

9. In the Actions pane, invoke the Web service with an employee ID of **21**. Note the error message in the Status pane.



Status

Source

Body

donateToFundResponse

return (string): Error: Employee with id 21 not found

10. Close the Web Services Explorer.

Off course?

If the Web service invocation fails with the fault:

```
<soapenv:Fault xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <faultcode>soapenv:Server</faultcode>
  <faultstring>javax.ejb.EJBAccessException: Unauthorized Access by
Principal Denied: Unauthorized Access by Principal Denied</faultstring>
```

Go back to 11.3, “Disable Java EE EJB declarative security” on page 329 and verify that the DonateBean was changed correctly.

11.6 Create and test a Web service client

The WSDL is published to allow other systems to implement Web services clients (Java EE or non-Java EE) of the published service. In this section we create and test a Web service client.

In this section we create a second enterprise application project, run the **`jaxws-tools`** command to generate helper code, create a client servlet, and then test it.

11.6.1 Create an enterprise application project for the client

In this section we create a second enterprise application project that contains the generated Web service client:

1. In the Project Explorer, right-click **DonateEAR** and select **New** → **Enterprise Application Project** (or select **File** → **New** → **Project** and from the pop-up select **Java EE** → **Enterprise Application Project**).
2. In the New EAR Application Project/EAR Application Project pop-up, set the Project Name to **DonateWSClientEAR**, accept the other defaults, and click **Next**.
3. In the New EAR Application Project/Enterprise Application pop-up, click **New Module**.
   - In the resulting Create default Java EE modules pop-up, select only **Web module**, set the Web module name to **DonateWSClientWeb**, and click **Finish**.
4. Back in the New EAR Application Project/Enterprise Application pop-up, ensure that **DonateWSClientWeb** is selected. Select **Generate Deployment Descriptor** and click **Next**.

5. In the New EAR Application Project/Geronimo Deployment Plan pop-up, set the Group Id to **ejee**, leave the Artifact Id empty, and click **Finish**.
6. Open **geronimo-web.xml** (in `DonateWSClientWeb/WebContent/WEB-INF`).
   - Change the group Id from `default` to **ejee**.
   - Save and close `geronimo-web.xml`.

11.6.2 Run the jaxws-tools command

We use the **`jaxws-tools`** command, provided with WASCE, to create the portable Java artifacts that are used by the Web services client:

1. Run the following command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory:

   ```
   jaxws-tools wsimport -s <OUTPUT_DIR>
            http://localhost:8080/DonateBeanService/DonateBean?wsdl
   ```

   Where <`OUTPUT_DIR`> is the directory where the stubs should be created, for example, `C:\7639code\ws-create\stubs`.

   ```
   jaxws-tools wsimport -s C:\7639code\ws-create\stubs
                  http://localhost:8080/DonateBeanService/DonateBean?wsdl
   ```

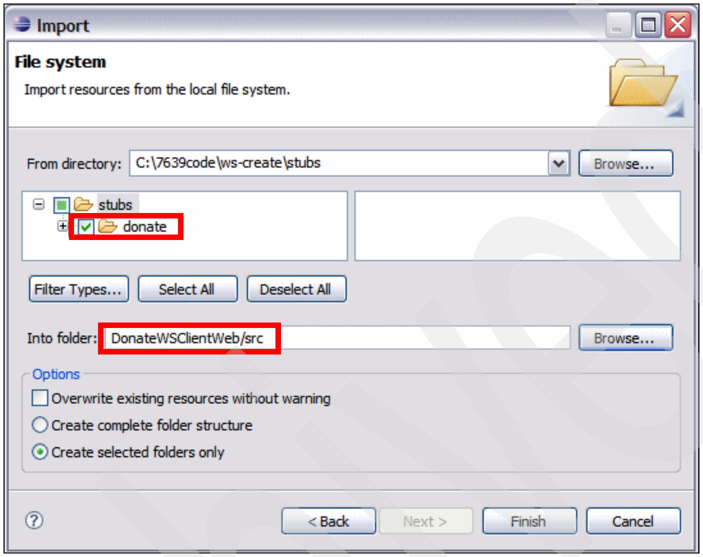
   Behind the scenes:

   When executing the **`jaxws-tools wsimport`** command, the portable artifacts to be used by the client are generated from the WSDL file:

   - Service endpoint interface (SEI): `DonateBean` (refer to Example 11-3 on page 340)
   - Service class: `DonateBeanService` (refer to Example 11-4 on page 341)
   - Exception class mapped from `wsdl:fault` (if any)
   - Async response bean derived from response `wsdl:message` (if any)
   - JAXB generated value types (mapped Java classes from schema types, if any): `DonateToFund` and `DonateToFundResponse`
   - Helper classes: `ObjectFactory` and `package-info`

2. Import the generated stubs into the DonateWSClientWeb project:
   - In the Project Explorer, select **File** → **Import**.
   - In the Import/Select pop-up, select **General** → **File System** and click **Next**.

- In the Import/File system pop-up:
  - Set the From directory to the <OUTPUT_DIR> used in the command.
  - Expand and select the **donate** folder.
  - Set the Into folder to **DonateWSClientWeb/src**.
  - Click **Finish**.



File system

Import resources from the local file system.

From directory: C:\7639code\ws-create\stubs

- stubs
  - ☑ donate

Into folder: DonateWSClientWeb/src

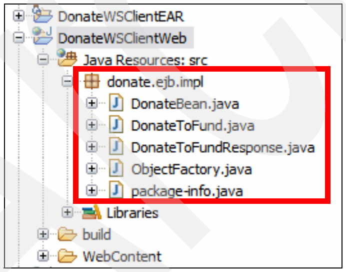
Options

☐ Overwrite existing resources without warning

○ Create complete folder structure

● Create selected folders only

3. Back in the Project Explorer, expand the imported **donate.ejb.impl package** and note that the Java classes were imported.



```
DonateWSClientEAR
DonateWSClientWeb
  Java Resources: src
    donate.ejb.impl
      DonateBean.java
      DonateToFund.java
      DonateToFundResponse.java
      ObjectFactory.java
      package-info.java
    Libraries
  build
  WebContent
```

Example 11-3 Generated service endpoint interface (SEI): DonateBean

```
@WebService(name = "DonateBean", targetNamespace="http://impl.ejb.donate/")
public interface DonateBean {

   @WebMethod
   @WebResult(targetNamespace = "")
   @RequestWrapper(localName = "donateToFund",
                   targetNamespace = "http://impl.ejb.donate/",
                   className = "donate.ejb.impl.DonateToFund")
   @ResponseWrapper(localName = "donateToFundResponse",
                    targetNamespace = "http://impl.ejb.donate/",
                    className = "donate.ejb.impl.DonateToFundResponse")
   public String donateToFund(
      @WebParam(name = "arg0", targetNamespace = "")
      int arg0,
      @WebParam(name = "arg1", targetNamespace = "")
      String arg1,
      @WebParam(name = "arg2", targetNamespace = "")
      int arg2);
}
```

Example 11-4 Generated client service class: DonateBeanService

```
@WebServiceClient(name = "DonateBeanService",
   targetNamespace = "http://impl.ejb.donate/",
   wsdlLocation = "http://localhost:8080/DonateBeanService/DonateBean?wsdl")
public class DonateBeanService extends Service {
    private final static URL DONATEBEANSERVICE_WSDL_LOCATION;
    static {
        URL url = null;
        try {
            url = new
                URL("http://localhost:8080/DonateBeanService/DonateBean?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        DONATEBEANSERVICE_WSDL_LOCATION = url;
    }

    public DonateBeanService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public DonateBeanService() {
        super(DONATEBEANSERVICE_WSDL_LOCATION,
                new QName("http://impl.ejb.donate/", "DonateBeanService"));
    }

   @WebEndpoint(name = "DonateBeanPort")
    public DonateBean getDonateBeanPort() {
        return (DonateBean)super
             .getPort(new QName("http://impl.ejb.donate/", "DonateBeanPort"),
                       DonateBean.class);
    }
}
```

11.6.3 Create a client servlet

In this section we create a servlet that uses the @WebServiceRef annotation to use the artifacts generated by the `jaxws-tools` command:

1. In the Project Explorer, right-click **DonateWSClientWeb** and select **New** → **Servlet**.
2. In the Create Servlet pop-up, set the Java package name to **donate.client**, the Class name to **ClientServlet**, and click **Finish**. The `ClientServlet` class opens in the editor.

Behind the scenes:

A new servlet is created and mapped to a `/ClientServlet` URL. The wizard creates the `HTTPServlet` Java class in the project. It also creates the definition of the servlet and the following mapping in the `web.xml` deployment descriptor:

```
<servlet>
    <description></description>
    <display-name>ClientServlet</display-name>
    <servlet-name>ClientServlet</servlet-name>
    <servlet-class>donate.client.ClientServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ClientServlet</servlet-name>
    <url-pattern>/ClientServlet</url-pattern>
</servlet-mapping>
```

3. After the class definition, insert the following lines:

```
public class ClientServlet extends HttpServlet{

   @WebServiceRef
   DonateBeanService service;

   @WebServiceRef(value=DonateBeanService.class)
   DonateBean mybean;
```

Behind the scenes:

Referencing a Web service was been simplified in Java EE 5. The annotation used to reference the service is:

@WebServiceRef The `javax.xml.ws.WebServiceRef` annotation is used to declare a reference to a Web service. It can be used to reference a generated service class or a generated service endpoint interface (in that case the generated service class type parameter must be always present).

In the client servlet we use both references. First we use a reference to the generated service class (`DonateBeanService`), and the second reference binds to a generated service endpoint interface (`DonateBean`).

Note that each technique works on its own.

4. In the `doGet` method, delete the method body, and insert the **F71 Web Service Client Servlet** snippet from the Snippets view (Example 11-5).

Example 11-5 Web service client servlet

```
protected void doGet(......) throws .... {
   PrintWriter out = response.getWriter();
   response.setContentType("text/plain");
   out.println("Calling DonateBean Web service... ");
   try {
      DonateBean bean = service.getDonateBeanPort();
      out.print("Response using service class: ");
      out.println(bean.donateToFund(1, "DonationFund", 1));
      out.print("Response using SEI directly: ");
      out.println(mybean.donateToFund(1, "DonationFund", 1));
   } catch (Exception e) {
      out.println("EXCEPTION: " + e.getLocalizedMessage());
   }
}
```

5. Replace the `doPost` method body with `doGet(request, response);`.

```
protected void doPost(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
   doGet(request,response);
}
```

6. Organize imports (**Ctrl+Shift+O**).
7. Save and close `ClientServlet.java`.

Behind the scenes:

In the `ClientServlet` we reference the Web service in two ways:

1. We inject the service class.

   Using the service class we have to get the service proxy, also known as a port. The method to get a service proxy is usually named `get<proxyname>Port`. Here are the lines to retrieve the service proxy for the DonateBean Web service and invoke the method:

   ```
   DonateBean bean = service.getDonateBeanPort();
   bean.donateToFund(1, "DonationFund", 1)
   ```

2. In the second reference, we provide the value argument that contains the service class type. In this case, the container injects a SEI by instantiating the service class and retrieving the port. We can call the method directly:

   ```
   mybean.donateToFund(1, "DonationFund", 1)
   ```

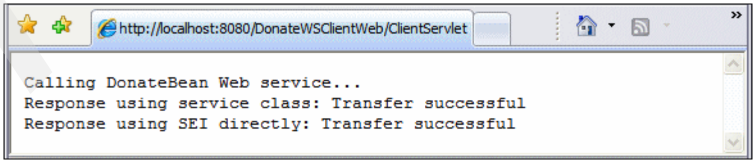
   The advantage of using the second reference is that the code is reduced, and we do not have to call the method to return the Web service port. However, in some cases the service class might be customized and the method name to return the Web service port might have a different name.

11.6.4 Run the client servlet

Now we can run the servlet by publishing the application:

1. In the Servers view, right-click the **ExperienceJavaEE Server** and click **Add and Remove Projects.**
   - Select the DonateWSClientEAR project on the left and click **Add.**
   - Click **Finish** to deploy the enterprise application to the server.
2. Right-click the ExperienceJavaEE Server and select **Publish.**
3. Start a browser and open the following URL:

   `http://localhost:8080/DonateWSClientWeb/ClientServlet`
4. The result of the Web service client is displayed.



```
Calling DonateBean Web service...
Response using service class: Transfer successful
Response using SEI directly: Transfer successful
```

11.6.5 Recapitulation

The preceding steps are sufficient to perform the experiences in Chapter 12, “Implement security for the Web service” on page 367, but we only created a very basic Web service.

The **optional sections** that follow cover other Web services topics that users might encounter when developing Web services.

11.7 Create a top-down Web service

In 11.4, “Create the bottom-up EJB Web service”, we created a Web service from the Java code. In this section we create a Web service based on the WSDL file, having first the contract and then creating the logic to implement it.

Here we create a Web service based on a POJO in a Web project and implement the service defined in the WSDL file generated in the bottom-up example. We use a new Web project to serve this Web service. Then we use the `DonateBean` session bean to execute the operations defined in the WSDL.

11.7.1 Create a Web project for the top-down Web service

Here we create the top-down Web service in a new Web project:

1. Right-click in Project Explorer and select **New** → **Dynamic Web Project**.
2. In the New Dynamic Web Project pop-up, type **DonateTopDownWeb** as Project name. Select **Add project to an EAR** and select the **DonateEAR**. Click **Next**.
3. For Web Module, select **Generate Deployment Descriptor**, and click **Next**.
4. For Geronimo Deployment Plan, type **ejee** as Group Id, and click **Finish**.
5. Right-click the **DonateTopDownWeb** project and select **Properties**. In the Properties pop-up, **Java EE Module Dependencies** page, select **DonateEJB.jar** and click **OK**.

11.7.2 Map the WSDL service to Java artifacts

First we have to map the WSDL service and types to Java artifacts. The **`wsimport`** tool does that by automatically generating the Java portable artifacts from the WSDL.

For this example, we generate from a WSDL file located in the file system, differently than what is described in 11.6, “Create and test a Web service client”, where we used the WSDL from the server.

The WSDL file is created with the DonateEAR application that contains the Web service deployed on the server. We saved the WSDL and matching XSD file in:

```
c:\7639code\ws-create\wsdl\DonateBean.wsdl
c:\7639code\ws-create\wsdl\DonateBeanService_schema1.xsd
```

To create the portable artifacts, do the following:

1. Run the **`jaxws-tools`** command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory, replacing `<OUTPUT_DIR>` by the path where the generated stubs should be created (`C:\7639code\ws-create\stubstopdown`):

   ```
   jaxws-tools wsimport c:\7639code\ws-create\wsdl\DonateBean.wsdl
                -s <OUTPUT_DIR>
   ```

   The generated output is identical to what we generated in 11.6.2, “Run the jaxws-tools command” on page 339. See Example 11-3 on page 340 for the service endpoint interface.

2. Copy the generated stubs into the DonateTopDownWeb source files directory (similar to the steps you did in 11.6, “Create and test a Web service client” on page 338).

Workaround:

The DonateTopDownWeb project is included in the DonateEAR enterprise archive. So the classpath of them is shared. This causes a conflict because the generated service endpoint interface has the same full qualified name as the `DonateBean` session bean:

```
donate.ejb.impl.DonateBean
```

To resolve this, we rename the package name of the generated artifacts:

1. Right-click the generated package `donate.ejb.impl` and select **Refactor** → **Rename**.
2. In the Rename Package pop-up, set the new name to **donate.ejb.impl.ws**, and click **OK**.

11.7.3 Create a Java class that implements the SEI

Now we create a POJO class that implements the SEI, by using annotations:

1. In the Project Explorer, right-click the **DonateTopDownWeb** project and select **New** → **Class**.

2. In the New Java Class/Java Class pop-up:
   - Set Package to **donate.web**.
   - Set Name to **DonateWSImpl**.
   - Next to Interfaces, click **Add**. Start typing **DonateBean**, and select **DonateBean - donate.ejb.impl.ws**. Click **OK**.
   - Back in the New Java Class/Java Class pop-up, click **Finish**.
3. In the resulting `DonateWSImpl` editor:
   - Add the following annotation before the `DonateWSImpl` class definition:

```
@WebService(endpointInterface="donate.ejb.impl.ws.DonateBean",
                              targetNamespace = "http://impl.ejb.donate/")
```

   - Add an @EJB annotation after the class to have the `DonateBean` session bean injected (the remote interface has the simple `donateToFund` method):

```
@EJB DonateBeanRemote donateBean;
```

   - Complete the `donateToFund` method:

```
System.out.println("Top-down Web service: Forwarding to session EJB");
return donateBean.donateToFund(arg0,arg1,arg2);
```

   - Organize imports (**Ctrl+Shift+O**).
   - Save and close `DonateWSImp.java`.

Behind the scenes:

The completed `DonateWSImpl` class is shown here (snippet **F76 Web Service Top Down**):

```
@WebService(endpointInterface="donate.ejb.impl.ws.DonateBean",
      targetNamespace = "http://impl.ejb.donate/")
public class DonateWSImpl implements DonateBean {

   @EJB DonateBeanRemote donateBean;

   public String donateToFund(int arg0, String arg1, int arg2) {
      System.out.println
         ("Top-down Web service: Forwarding to session EJB");
      return donateBean.donateToFund(arg0,arg1,arg2);
   }
}
```

The `endpointInterface` attribute points to the generated `DonateBean` interface. This attribute is used to define the abstract WSDL contract (port type and bindings).

4. In the Project Explorer, open **web.xml** (in `DonateTopDownWeb/WebContent /WEB-INF`).
5. In the `web.xml` editor:
   - Switch to the **Source** tab.
   - Before the closing `</web-app>` tag, insert the **F77 Web Service Top Down XML** snippet from the Snippets view (Example 11-6).
   - Save and close `web.xml`.

Example 11-6 DonateTopDownWeb web.xml servlet definition

```
   ......
   <servlet>
      <servlet-name>DonateWSImpl</servlet-name>
      <servlet-class>donate.web.DonateWSImpl</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>DonateWSImpl</servlet-name>
      <url-pattern>/DonateTopDownWS</url-pattern>
   </servlet-mapping>
</web-app>
```

Behind the scenes:

This service is being exposed from a Web project, the URL mapping to the service is done through a servlet URL mapping.

This mapping is not necessary for simple JAX-WS Web service deployments. If `web.xml` is not found in the Web project, the Web service automatically generates the mapping based on the implementation class name, concatenated with `Service`. If you did not have the `web.xml` deployment descriptor, the URL would be:

`http://localhost:8080/DonateJSFWeb/DonateWSImplService?wsdl`

11.7.4 Run the top-down Web service

To test the Web service, republish the application in the server and use the Web Services Explorer as described in 11.9.3, “Deploy and test the Web service” on page 355:

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.

2. Display the WSDL or XSD generated by the Web service:

```
http://localhost:8080/DonateTopDownWeb/DonateTopDownWS?wsdl
http://localhost:8080/DonateTopDownWeb/DonateTopDownWS?xsd=DonateWSImplS
ervice_schema1.xsd
```

3. This service can be tested by using the Web Services Explorer as described in 11.6, “Create and test a Web service client” on page 338, using the WSDL:

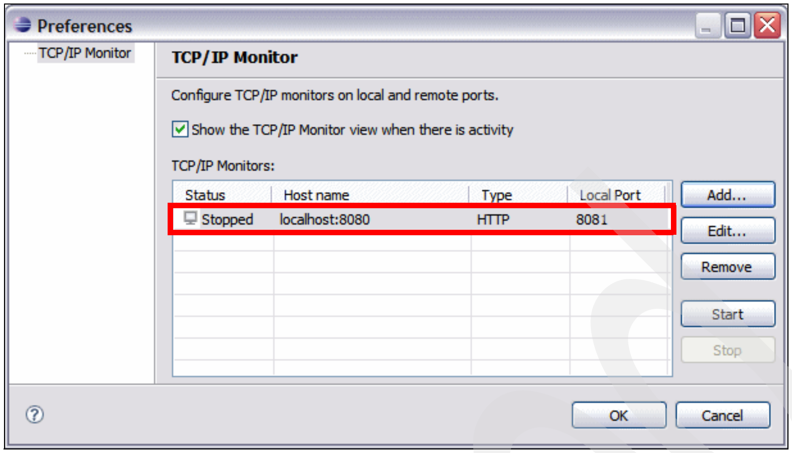
```
http://localhost:8080/DonateTopDownWeb/DonateTopDownWS?wsdl
```

11.8 Monitor the SOAP traffic using TCP/IP Monitor

In this section we describe how to use the Eclipse TCP/IP Monitor feature to monitor and inspect the SOAP messages exchanged by a Web service and a Web service client.

The TCP/IP Monitor can be configured to listen for TCP/IP packages on one port and forward them to a different host and port. Here we configure it to listen to the localhost port 8081 and forward the requests to localhost port 8080. We then use the Web Services Explorer to make requests to the DonateBean Web service using the port 8081 and we are able to monitor the messages exchanged by the server and the Web Services Explorer:

1. Select **Window** → **Show View** → **Other**. In the Show view pop-up, expand the **Debug** folder and select **TCP/IP Monitor**.
2. The TCP/IP Monitor view is shown in the bottom pane. In the view, click the **Menu** icon ▽ on the top right and select **Properties**.
3. In the Preferences pop-up, click **Add**.
4. In the New Monitor pop-up, set the fields as follows:
   - Local monitoring port: **8081**
   - Hostname: **localhost**
   - Port: **8080**
   - Type: **HTTP**
   - Time-out (in milliseconds): **0**
   - Click **OK**.



TCP/IP Monitor

Configure TCP/IP monitors on local and remote ports.

☑ Show the TCP/IP Monitor view when there is activity

TCP/IP Monitors:

Status	Host name	Type	Local Port
Stopped	localhost:8080	HTTP	8081

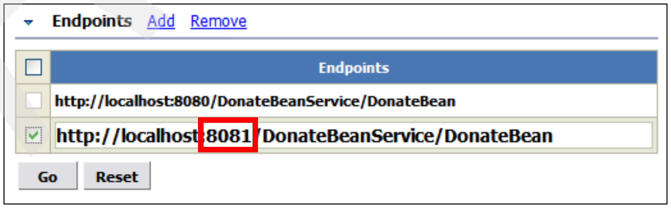
5. Select the new entry, and click **Start**.
6. Ensure that the status changes from `Stopped` to `Started`, and then click **OK**.

> **Alternative:**
>
> You can also configure the TCP/IP Monitor by selecting **Window** → **Preferences**, and in the Preferences pop-up select **Run/Debug** → **TCP/IP Monitor**.

7. Start the Web Services Explorer (select **Run** → **Launch Web Services Explorer**).
8. In the WSDL Main page, Actions pane, set the WSDL URL to:

```
http://localhost:8080/DonateBeanService/DonateBean?wsdl
```

9. Click **Go**.
10. Next to Endpoints, click **Add**. In the new endpoint, change the address to **8081**. Select the new endpoint and click **Go**.



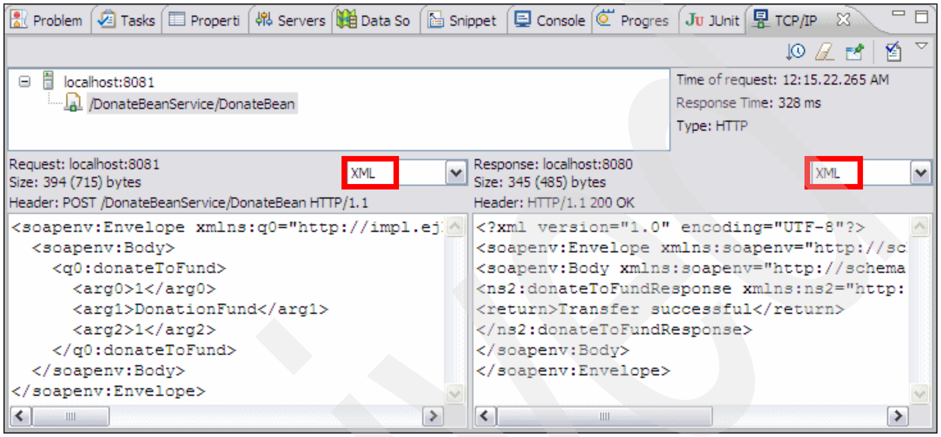
□	Endpoints
□	http://localhost:8080/DonateBeanService/DonateBean
☑	http://localhost:8081/DonateBeanService/DonateBean

11. Select the **donateToFund** operation.

12. Set the form values to **1**, **DonationFund**, **1**, and click **Go**. A Transfer successful message is displayed in the Status pane.

13. Select the TCP/IP Monitor view and note that a request is shown.

    – Change the message representation from byte to **XML** to see the formatted display of the SOAP request and response.



Time of request: 12:15.22.265 AM
Response Time: 328 ms
Type: HTTP

Request: localhost:8081
Size: 394 (715) bytes
Header: POST /DonateBeanService/DonateBean HTTP/1.1
XML

```
<soapenv:Envelope xmlns:q0="http://impl.ej
  <soapenv:Body>
    <q0:donateToFund>
      <arg0>1</arg0>
      <arg1>DonationFund</arg1>
      <arg2>1</arg2>
    </q0:donateToFund>
  </soapenv:Body>
</soapenv:Envelope>
```

Response: localhost:8080
Size: 345 (485) bytes
Header: HTTP/1.1 200 OK
XML

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://sc
<soapenv:Body xmlns:soapenv="http://schema
<ns2:donateToFundResponse xmlns:ns2="http:
<return>Transfer successful</return>
</ns2:donateToFundResponse>
</soapenv:Body>
</soapenv:Envelope>
```

14. On the left we can see the SOAP message sent to the server with the service request. In it we can see the SOAP message body having the operation requested (`donateToFund`) and the arguments values (Example 11-7).

Example 11-7 SOAP request envelope

```
<soapenv:Envelope xmlns:q0="http://impl.ejb.donate/"
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:donateToFund>
      <arg0>1</arg0>
      <arg1>DonationFund</arg1>
      <arg2>1</arg2>
    </q0:donateToFund>
  </soapenv:Body>
</soapenv:Envelope>
```

15. On the right we can see the SOAP message returned with the results of the Web service execution (Example 11-8).

Example 11-8 SOAP response envelope

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <ns2:donateToFundResponse xmlns:ns2="http://impl.ejb.donate/">
      <return>Transfer successful</return>
   </ns2:donateToFundResponse>
</soapenv:Body>
</soapenv:Envelope>
```

The TCP/IP Monitor can be used to monitor any interactions between a Web service and its client. To use the TCP/IP Monitor, just change the service endpoint URL to point to the monitored port.

Tip:

Changing the endpoint dynamically in the Web Services Explorer is easier than changing the WSDL file.

11.9 Create a Web service with complex data types

Real world Web services often use complex data types. Binding Java objects to XML is controlled and supported by JAXB, which defines how to bind an XML document to Java objects. It contains methods to marshal and unmarshal XML contents to and from Java objects.

In 11.4, “Create the bottom-up EJB Web service”, we only exposed a method with primitive data types. However, the `DonateBean` also contains a method with complex types:

```
public void donateToFund(Employee employee, Fund fund, int hours)
```

11.9.1 Bind JPA entities to an XML schema

When creating a Web service with complex data types, we have to bind the complex data types to an XML schema. In our case the data types are the JPA entities `Employee` and `Fund`:

1. In the Project Explorer, open **Employee** (in `DonateJPAEmployee/src/vacation.entities`).

2. In the `Employee` editor:
   - Add an @XmlType annotation before the class definition.

     ```
     @Entity
     @XmlType
     public class Employee ...... { ... }
     ```

   - Organize imports (**Ctrl+Shift+O**), save, and close `Employee.java`.

> **Behind the scenes:**
>
> The JAXB @XmlType annotation maps a Java class to a schema type. It makes the object serializable for being exposed by a Web service.

3. Repeat this for the **Fund** class (in `DonateJPAFund/src/donate.entities`), that is, add an @XmlType annotation before the class definition.

11.9.2 Expose a complex method in the session EJB

Next we expose a complex method in the `DonateBean` as a Web service. We want to use the `donateToFund(Employee,Fund,int)` method, but there are two problems (see **Behind the scenes** next for a detailed explanation):

- The method throws `Exception`, which cannot be converted into a Web service fault.
- The method name is the same as for `donateToFund(int,String,int)`, and WSDL operations do not distinguish by parameters.

We follow these steps:

1. In the Project Explorer, open **DonateBean** (in `DonateEJB/ejbModule/donate.ejb.impl`).
2. In the `DonateBean` editor, insert the **F72 DonateBean Complex Method** snippet from the Snippets view at the end of the class before the closing brace. (Example 11-9).
3. Organize imports (**Ctrl+Shift+O**), save, and close `DonateBean.java`.

Example 11-9 Complex method for a new Web service

```
@WebMethod
@PermitAll
public void donateToFundComplexTypes(Employee emp, Fund fund, int hours){
   try {
      donateToFund(emp, fund, hours);
   } catch (Exception e) { } //no operation
}
```

Behind the scenes:

We have created a separate method to expose the complex types. This was done because the originally defined method with the `Employee` and `Fund` types throws a `java.lang.Exception`. This exception cannot be directly mapped to a Web service fault. In 11.10, “Define a Web service fault” on page 359, we describe how to create a Web service fault.

Even if we did not want to handle Web services faults, we cannot directly expose the method:

```
public void donateToFund(Employee employee, Fund fund, int hours) ..;
```

In Java, this method name does not collide with the other `donateToFund` method, because the signatures are different (polymorphism):

```
public String donateToFund(int id, String fundName, int hours);
```

However, these methods name will have a collision when defined in the WSDL, because there only the method name is used to define the operation name.

To avoid this duplication, we could add the operation name to the exposed method in the @WebMethod annotation:

```
@WebMethod(operationName="donateToFundComplexType")
public void donateToFund(Employee employee, Fund fund, int hours)
                                                    throws Exception;
```

However, we decided not to disturb the current Web service and rather define a new method with complex arguments.

11.9.3 Deploy and test the Web service

Next we test the updated Web service using the Web Services Explorer:

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.

Behind the scenes:

When we publish the application, WASCE deploys the Web service and automatically generates the WSDL and the XSD related to it. This time the XSD is mapping the complex types, it is generated by using JAXB.

The WSDL and the XSD can be retrieved using the following URLs:

```
http://localhost:8080/DonateBeanService/DonateBean?wsdl
http://localhost:8080/DonateBeanService/DonateBean?xsd=DonateBeanServi
ce_schema1.xsd
```

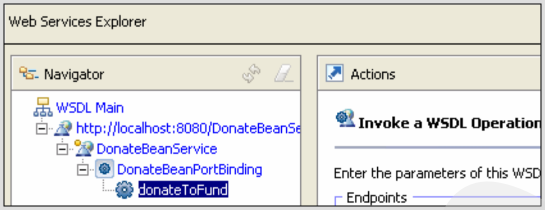
For example, the `Fund` object is mapped to the following XSD part:

```
<xs:complexType name="fund">
   <xs:sequence>
      <xs:element name="balance" type="xs:int"/>
      <xs:element minOccurs="0" name="name" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

2. In the Java EE perspective, select **Run** → **Launch the Web Services Explorer**.
3. In the Web Services Explorer browser, click the **WSDL Page** icon [ ] in the upper right.
4. In the Navigator pane, select **WSDL Main**.

Behind the scenes:

If you execute these optional steps immediately after testing the base Web service (as you did 11.5, “Test the Web service using the Web Services Explorer” on page 335), the Navigator pane will still have an entry for the initial version of the Web service with just a single method (`donateToFund`):



Adding the updated WSDL in the next step results in two service definitions in the Navigation pane: The old definition with a single method and the updated definition with two methods.

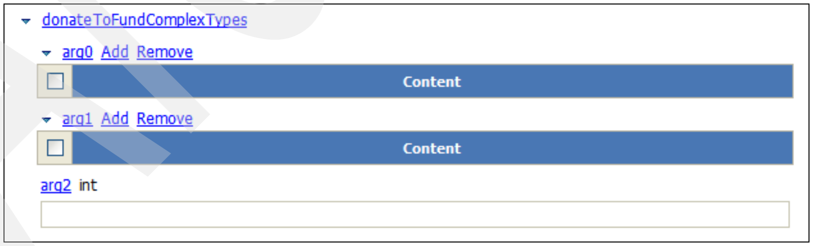
The Web Services Explorer allows you to have multiple WSDLs loaded into the Navigator and even multiple definitions of the same service. However, if you prefer, you can delete this old definition by using the **Clear** icon ( )

5. In the Actions pane, enter the following URL and click **Go**.

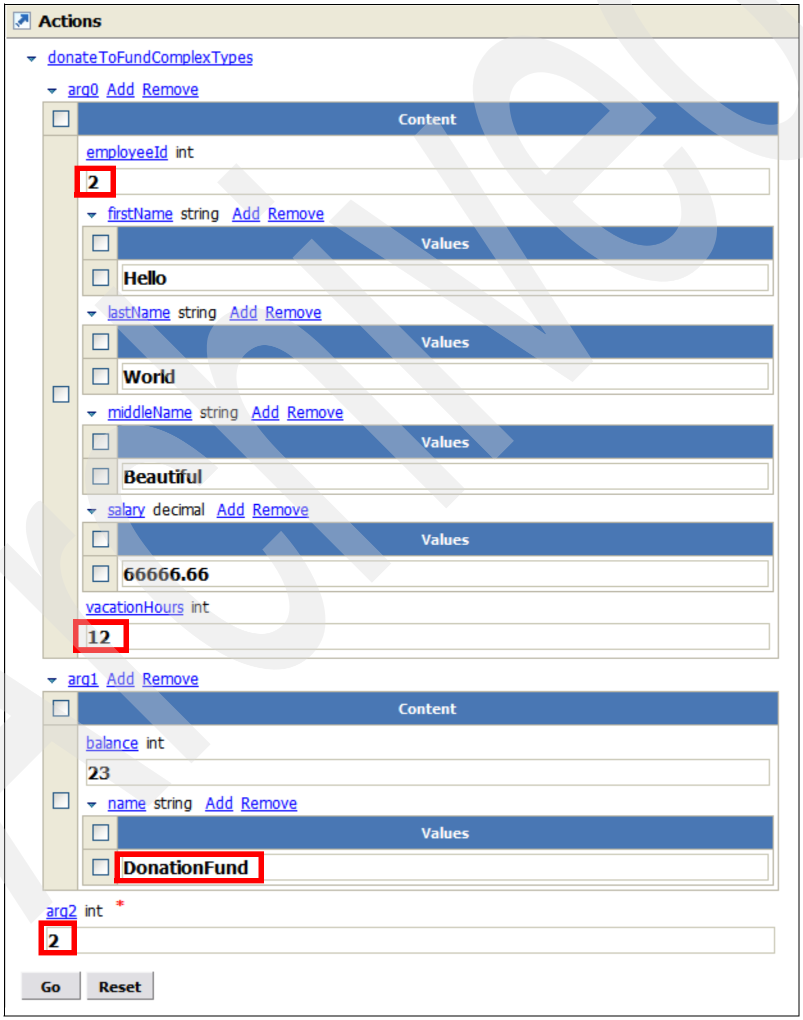
   `http://localhost:8080/DonateBeanService/DonateBean?wsdl`

6. In the Navigator pane, expand **DonateBeanPortBinding** to see both methods exposed as Web services. Select **donateToFundComplexTypes**.

7. In the Actions pane, note that arg0 (employee) and arg1 (fund) are now complex types.



8. For the first argument (`arg0`, employee), click **Add** to expand the complex argument:
   - Set `employeeId` to **2** (for the second user, `DNARMSTRONG`).
   - Type any value for `vacationHours`.
   - The other fields can be left blank, or they can be set by clicking **Add**.
9. For the second argument (`arg1`, fund), click **Add**:
   - Set `balance` to any value.
   - Click **Add** for name and type **DonationFund** as value.
10. For the third argument (`arg2`, hours to donate), type the desired hours (**2**).



employeeId int
2

firstName string Add Remove
Hello

lastName string Add Remove
World

middleName string Add Remove
Beautiful

salary decimal Add Remove
66666.66

vacationHours int
12

balance int
23

name string Add Remove
DonationFund

arg2 int *
2

Behind the scenes:

The generated schemas for the `Employee` and `Fund` element indicate that integer fields are required, and all the others are nullable (`minOccurs="0"`). Primitive types in Java (such as `int`), cannot be `null`, and required in the XSD.

Therefore, the Web Services Explorer user interface requires that you enter information for these fields, otherwise you get an immediate error:

- `employeeId` and `vacationHours` in `Employee`
- `balance` in `Fund`

From an application and schema perspective, the only required fields should be the primary keys: `employeeId` in `Employee` and `balance` in `Fund`. The `employee` complex types used by the `DonateBean` service are defined in the generated XSD:

```
<xs:complexType name="employee">
   <xs:sequence>
      <xs:element name="employeeId" type="xs:int"/>
      <xs:element minOccurs="0" name="firstName" type="xs:string"/>
      <xs:element minOccurs="0" name="lastName" type="xs:string"/>
      <xs:element minOccurs="0" name="middleName" type="xs:string"/>
      <xs:element minOccurs="0" name="salary" type="xs:decimal"/>
      <xs:element name="vacationHours" type="xs:int"/>
   </xs:sequence>
</xs:complexType>
```

Web Services Explorer generates an HTML form with fields for each element defined in the XSD complex type.

11. Click **Go** at the bottom of the pane. The result in the Status pane is empty, because the method return type is void.

12. Click **Source** on the Status pane and then study the SOAP Request Envelope and the SOAP Response Envelope. A successful response shows a `donateToFundComplexTypesResponse` element with no data (because the method return is void).

```
============================= REQUEST ENVELOPE ==================
<soapenv:Envelope .....>
  <soapenv:Body>
    <q0:donateToFundComplexTypes>
      <arg0>
        <employeeId>2</employeeId>
        <firstName>Hello</firstName>
        <lastName>World</lastName>
        <middleName>Beautiful</middleName>
        <salary>66666.66</salary>
        <vacationHours>12</vacationHours>
      </arg0>
      <arg1>
        <balance>23</balance>
        <name>DonationFund</name>
      </arg1>
      <arg2>2</arg2>
    </q0:donateToFundComplexTypes>
  </soapenv:Body>
</soapenv:Envelope>
============================= RESPONE ENVELOPE =================
<soapenv:Envelope .....>
  <soapenv:Body ......>
    <donateToFundComplexTypesResponse ......>
  </soapenv:Body>
</soapenv:Envelope>
```

Important:

In WASCE, the complex types used by a EJB Web service must all be in the same EJB project.

If the complex type classes are located in a separated project, the Web services container is not able to load the classes, even if the JAR file is imported into the EJB Web service project. And the server is not able to deploy the application.

Therefore, always keep all the EJB interfaces classes with the EJB implementations.

11.10 Define a Web service fault

Java exceptions can be converted into `wsdl:fault` elements and encode server-side error conditions to the Web service client.

In this section we modify the method `donateToFundComplexTypes` to throw a custom exception, and we add annotations to the custom exception to allow JAXB to map the exception into a `wsdl:fault`.

11.10.1 Create a custom bean and a custom exception

To create the custom exception with a data bean, we perform these steps:

1. In the Project Explorer, right-click **DonateEJB** and select **New** → **Class**. In the New Java Class/Java Class pop-up, set the Package to **donate.ejb.exception**, the Name to **DonateFaultBean**, and click **Finish**.
2. In the `DonateFaultBean` editor (Example 11-10):
   - Insert an **@XmlType** annotation above the class definition.
   - Inside the class body, definite a string:

     ```
     private String donationErrorMessage;
     ```

   - Select `donateErrorMessage` and right-click **Source** → **Generate Getters and Setters**. From the resulting pop-up, generate the getters and setters for this field.
   - Organize imports (**Ctrl+Shift+O**).
   - Save and close `DonateFaultBean.java` (see snippet **F73 Donate Fault Bean**).

Example 11-10 Web service fault bean: DonateFaultBean

```
package donate.ejb.exception;

import javax.xml.bind.annotation.XmlType;

@XmlType
public class DonateFaultBean {
   private String donationErrorMessage;

   public String getDonationErrorMessage() {
      return donationErrorMessage;
   }

   public void setDonationErrorMessage(String donationErrorMessage) {
      this.donationErrorMessage = donationErrorMessage;
   }
}
```

Behind the scenes:

Fault beans are the types that represents a Web service’s fault (exception in Java). The fault bean returns the information about the fault to the Web service client.

3. In the Project Explorer, right-click **DonateEJB** and select **New** → **Class**. Set the Package to **donate.ejb.exception**, the Name to **DonateWSException**, the Superclass to **java.lang.Exception**, and click **Finish**.

4. In the `DonateWSException` editor (Example 11-11):

   – Insert a **@WebFault** annotation above the class definition.

   – Insert the **F74 Donate Fault Exception** snippet from the Snippets view into the class body (before the closing brace.

   – Organize imports (**Ctrl+Shift+O**).

   – Optionally add the annotation **@SuppressWarnings("serial")** before the class to eliminate the warning that no `serializedVersionUID` is present (shown in red).

   – Save and close `DonateWSException.java`

Example 11-11 Web service exception class: DonateWSException

```
package donate.ejb.exception;

import javax.xml.ws.WebFault;

@WebFault
@SuppressWarnings("serial")
public class DonateWSException extends Exception {

   private DonateFaultBean fbean;

   public DonateWSException(String message, DonateFaultBean fbean) {
      this.fbean = fbean;
   }

   public DonateWSException(String message, DonateFaultBean fbean,
                            Throwable cause) {
      this.fbean = fbean;
   }

   public DonateFaultBean getFaultInfo() {
      return fbean;
   }
}
```

Behind the scenes:

The JAX-WS 2.0 specification demands that the exception annotated with @WebFault must have two constructors and one method:

```
WrapperException(String message, FaultBean faultInfo)
WrapperException(String message, FaultBean faultInfo, Throwable cause)
FaultBean getFaultInfo()
```

The *WrapperException* is replaced by the name of the exception, and *FaultBean* is replaced by the class name that implements the fault bean. The fault bean is a Java bean that contains the information of the fault and is used by the Web service client to know the cause for the fault.

The DonateWSException class implements the JAX-WS specifications.

11.10.2 Change the DonateBean to throw the custom exception

Now we change the donateToFundComlexTypes method to throw the custom exception (Example 11-12):

1. In the Project Explorer, open **DonateBean** (in DonateEJB/ejbModule /donate.ejb.impl).
2. In the DonateBean editor:
   - Change the donateToFundComplexTypes method definition to throw a DonateWSException.
   - Insert the **F75 DonateBean Complex Method Fault** snippet from the Snippets view into the catch block.
   - Organize imports (**Ctrl+Shift+O**).
   - Save and close DonateBean.java.

Example 11-12 Method with complex types with custom exception

```
public void donateToFundComplexTypes(Employee emp, Fund fund, int hours)
                                throws DonateWSException {
   try {
      donateToFund(emp, fund, hours);
   } catch (Exception e) {
      DonateFaultBean faultBean = new DonateFaultBean();
      faultBean.setDonationErrorMessage("Failed to process donation.");
      throw new DonateWSException(e.getMessage(), faultBean, e.getCause());
   }
}
```

Behind the scenes:

In the donateToFundComplexTypes method, we catch the standard Exception and convert it to DonateWSException. This is necessary because DoanteWSException follows the JAX-WS 2.0 specification.

The Exception class does not contain the methods and the constructors defined in JAX-WS 2.0. If a method throws an Exception (or any exception that does not implement the specification), the JAX-WS Web service engine generates an error when it attempts to retrieve the fault bean from the exception thrown:

```
[Axis2WebServiceContainer] Exception occurred while trying to invoke
service method doService()
org.apache.axis2.AxisFault: An error was detected during JAXWS processing
   at org.apache.geronimo.axis2.ejb.EJBInterceptor.intercept
```

11.10.3 Run the Web service with the custom exception

To test the Web service, republish the application in the server and use the Web Services Explorer as described in 11.9.3, “Deploy and test the Web service” on page 355:

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.
2. Load a new copy of the WSDL and test the updated Web service from the Web Services Explorer using the same steps described in the preceding section (11.9, “Create a Web service with complex data types” on page 352):

   ```
   http://localhost:8080/DonateBeanService/DonateBean?wsdl
   ```

3. Test with a set of values that will not generate an exception (such as employee ID of **1** and fund name of **DonationFund**. The output should be the same as before:

   – Test with a set of values that will generate an exception (such as an employee ID of **99** or a fund name of **DonationFundyy**).

   – Click **Source** on the Status pane and then scroll down to the SOAP Response Envelope. The exception response shows a DonateWSException fault:

   ```
   <soapenv:Envelope>
     <soapenv:Body>
       <soapenv:Fault ......>
         <faultcode>soapenv:Server</faultcode>
         <faultstring>donate.ejb.exception.DonateWSException
         </faultstring>
   ```

```
                <detail>
                  <ns2:DonateWSExceptionBean ......>
                    <donationErrorMessage>Failed to process donation.
                    </donationErrorMessage>
                  </ns2:DonateWSExceptionBean>
                </detail>
              </soapenv:Fault>
            </soapenv:Body>
          </soapenv:Envelope>
```

4. Close the Web Services Explorer.

Behind the scenes:

The updated WSDL and the XSD can be retrieved using the following URLs:

```
http://localhost:8080/DonateBeanService/DonateBean?wsdl
http://localhost:8080/DonateBeanService/DonateBean?xsd=DonateBeanService_
schema1.xsd
```

The `donateToFundComplexTypes` WSDL operation now contains a fault definition:

```
<operation name="donateToFundComplexTypes">
   <soap:operation soapAction=""/>
   <input><soap:body use="literal"/></input>
   <output><soap:body use="literal"/></output>
   <fault name="DonateWSException">
      <soap:fault name="DonateWSException" use="literal"/>
   </fault>
</operation>
```

Other changes were made to the WSDL message sections and to the schema (`DonateBEANService_schema1.xml`) to reflect the additional message and complex type.

In this example you have used a user defined `FaultBean` type. This was defined in the schema definition file:

```
<xs:complexType name="donateFaultBean">
   <xs:sequence>
      <xs:element minOccurs="0" name="donationErrorMessage"
                                          type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

Other than using a complex type to implement the `FaultBean`, we could have used a simple string with the error message. This could be used in simple Web services where the faults are not complex.

11.11 Explore!

JAX-WS specification defines other technologies to improve Web services performance.

11.11.1 Handlers in JAX-WS

JAX-WS applies the chain of responsibility pattern by providing interceptors to its architecture. These interceptors are called *handlers* in JAX-WS, and are used to do additional processing of inbound and outbound messages.

JAX-WS defines two types of handlers:

SOAP handlers — Used to process SOAP-specific information such as SOAP security headers.

Logical handlers — Used when there is no need to access the SOAP message. For example, logical handlers can be used for message validation or to implement REST style Web services.

Here are some references about JAX-WS handlers:

- A little bit about handlers in JAX-WS:

  `https://jax-ws.dev.java.net/articles/handlers_introduction.html`

- Writing a handler in JAX-WS:

  `http://www.java-tips.org/java-ee-tips/java-api-for-xml-web-services/writing-a-handler-in-jax-ws.html`

11.11.2 Asynchronous Web service invocation

JAX-WS supports that a Web service client can interact with a Web services in a non-blocking, asynchronous approach. This is done by adding support for both a polling 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 if the server has responded. When the server responds, the actual response is retrieved.
- 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 without waiting for a response to return, while providing for a more dynamic and efficient model to invoke Web services.

For more information about asynchronous Web service invocation, refer to the following Web sites:

- Asynchronous Web service invocation with JAX-WS 2.0

  `http://today.java.net/pub/a/today/2006/09/19/asynchronous-jax-ws-web-services.html?page=last`

- Developing asynchronous Web services with Axis2:

  `http://www.ibm.com/developerworks/library/ws-axis2/index.html`

11.11.3 Attachments performance improvement

JAX-WS supports the use of SOAP Message Transmission Optimization Mechanism (MTOM) for sending binary attachment data. By enabling MTOM, you can send and receive binary data optimally without incurring the cost of data encoding to ensure that the data is included in the XML document.

For more information, consult the MTOM Guide *Sending Binary Data with SOAP*:

`http://ws.apache.org/axis2/1_0/mtom-guide.html`

11.11.4 Generate the WSDL XML file using the jaxws-tools

Having the container automatically generate the WSDL file during deployment makes very easy to develop and publish a Web service. However, in some situations the WSDL is needed before the actual service is deployed.

The **`jaxws-tools`** provides the **`wsgen`** command. It reads the Web service endpoint class and generates the artifacts for deploying the Web service.

The command to generate the WSDL is as follows:

```
<WASCE_HOME>\bin\jaxws-tools wsgen -classpath APP_CLASSPATH -wsdl
```

Where:

`<WASCE_HOME>`	WASCE installation directory
`<APP_CLASSPATH>`	Classpath to the classes of the application that contains the Web service

The WSDL file and its attached XML schema definition files (XSD) are created.

12

Implement security for the Web service

In this chapter, we update the Web service and Web service client from the previous chapter to implement authentication (Figure 12-1).

Note that WASCE does not support the Web Services Security (WS-Security) specification. We can only use basic authentication security or use AXIS2 to use partial WS_Security for our application.

12.1 Learn!

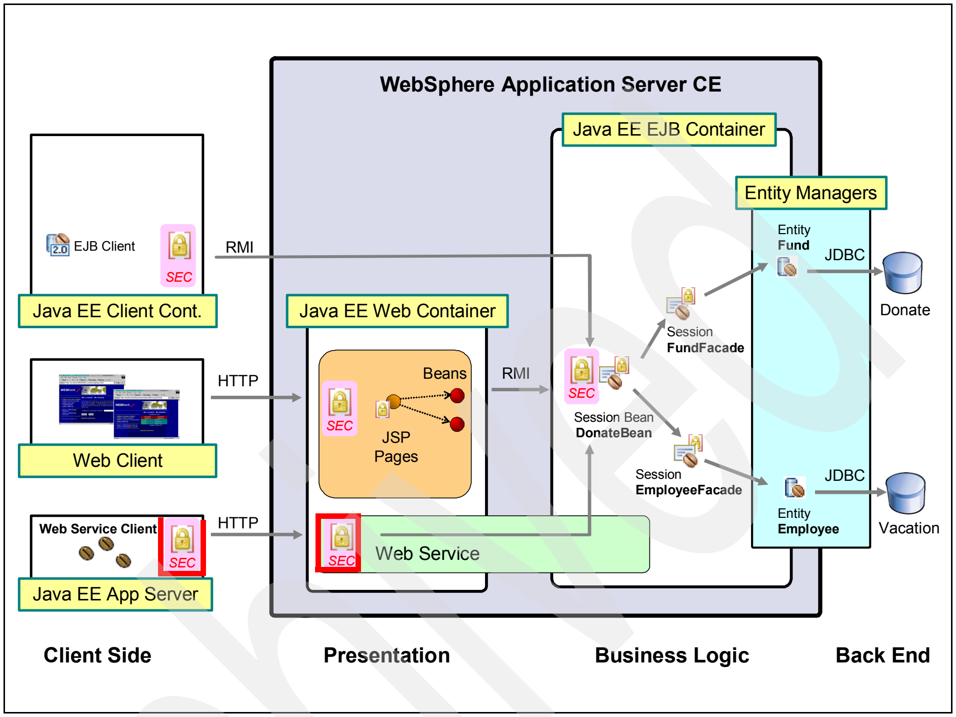


Figure 12-1 Donate application: Web service security

The JAX-WS specification requires that its implementations must support HTTP basic authentication. The specification does not specify the entire WS-* security protocols.

HTTP basic authentication (Figure 12-2) is a method to allow a client to provide credentials to a server when executing a request. It assumes that the connection between the server and the client is secure. So it is important to have an SSL/HTTPS connection when using this authentication method.

JAX-WS defines the annotations that can be used to protect the method based on roles.

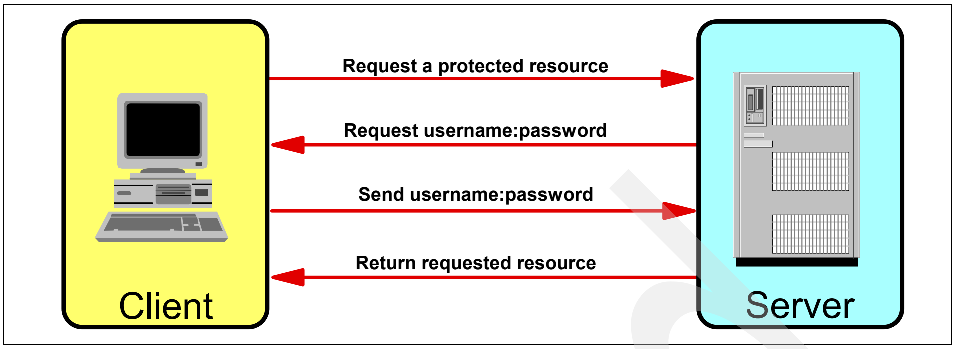

Request a protected resource
Request username:password
Send username:password
Return requested resource
Client
Server

Figure 12-2 HTTP Basic Authentication schema

12.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed all the applications of Part 2, “Core Java EE application” on page 97. In addition, you must have implemented the Web services in Chapter 11, “Create the Web service” on page 323.

12.3 Re-enable Java EE EJB declarative security

In this section we perform the set of tasks required to re-enable the Java EE EJB declarative security that we disabled in 11.3, “Disable Java EE EJB declarative security” on page 329:

1. Re-enable declarative security in the `DonateBean` session EJB:
   - In the Project Explorer, open **DonateBean** (in `DonateEJB/ejbModule/donate.ejb.impl`), and switch the security annotations for both `donateToFund` methods (remove the comment marker from @RolesAllowed and add a comment marker to @PermitAll):

```
@RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
//@PermitAll
public void donateToFund(Employee employee, Fund fund, int hours)
   ......

@RolesAllowed( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
//@PermitAll
```

```
    @WebMethod
    public String donateToFund(int employeeId, String fundName, ...)
       ......
```

- Organize the imports (**Ctrl+Shift+O)**, save, and close `DonateBean.java`.

2. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.

12.4 Test the Web service as-is

In this section we test the current Web service artifacts to verify that the `donateToFund` method fails with an authorization exception:

1. In the Java EE perspective, select **Run** → **Launch the Web Services Explorer**.
2. In the Web Services Explorer, click the **WSDL Page** icon [ ] in the upper right.
3. On the left side, in the Navigator pane, select **WSDL Main**.
4. On the right side, in the Actions pane, enter the following URL and click **Go**:

   ```
   http://localhost:8080/DonateBeanService/DonateBean?wsdl
   ```

5. In the Navigator pane, select **WSDL Main** → **http://localhost...** → **DonateBeanService** → **DonateBeanPortBinding** → **donateToFund**. You can also select the **donateToFund** operation in the Actions pane.
6. In the Actions pane, which now displays Invoke a WSDL Operation, complete the following fields:
   - arg0 (employee ID): **1**
   - Next to arg1 (fund name): Click **Add**, and type **DonationFund**.
   - arg2 (donation amount): **1**
7. Click **Go**. The Web service is invoked and the result is displayed in the Status pane.
8. Switch to the **Source** view to see the SOAP Response Envelope message indicating unauthorized access (thus confirming that EJB security was re-enabled):

   ```
   ...
   <soapenv:Fault>
      <faultcode>soapenv:Server</faultcode>
      <faultstring> javax.ejb.EJBAccessException: Unauthorized Access by
               Principal Denied: Unauthorized Access by Principal Denied
      </faultstring>
      ...
   </soapenv:Fault>
   ```

12.5 Update the Web service to support secure access

In this section we secure the EJB Web service with authentication.

12.5.1 Add authentication to an EJB-based Web service

We add HTTP basic authentication to the `DonateBean` Web service, which is based on an EJB session bean. To protect the Web service, we have to update the OpenEJB deployment plan:

1. Update the Geronimo application deployment descriptor in the EJB project:
   - In the Project Explorer, open **openejb-jar.xml** (in `DonateEJB/ejbModule/META-INF`).
   - Switch to the **Source** tab In the `openejb-jar.xml` editor and insert the **F81 Donate EJB Authentication** snippet from the Snippets views before the closing `<ejb:openejb-jar>` tag (Example 12-1).

     Example 12-1 DonateEJB OpenEJB Web service security

     ```
        <ejb:enterprise-beans>
           <ejb:session>
              <ejb:ejb-name>DonateBean</ejb:ejb-name>
              <ejb:web-service-security>
                 <ejb:security-realm-name>ExperienceJavaEERealm
                 </ejb:security-realm-name>
                 <ejb:transport-guarantee>NONE</ejb:transport-guarantee>
                 <ejb:auth-method>BASIC</ejb:auth-method>
              </ejb:web-service-security>
           </ejb:session>
        </ejb:enterprise-beans>
     </ejb:openejb-jar>
     ```

   - Save and close `openejb-jar.xml`.

Behind the scenes:

The deployment plan adds the Web service security configuration to the `DonateBean` session bean, by setting the following elements:

`security-realm-name` Links the Web service security to a specific realm name. In this case we have linked it to the `ExperienceJavaEERealm`.

`transport-guarantee` Specifies the communication that will take place between the client and the server. In this case we set the value to `NONE` so that the Web service access does not need any transport guarantees.

`auth-method` Specifies the method to authenticate the user. In this case we are using the HTTP basic authentication.

Note that these tags are similar to the tags used to protect a servlet in a Web deployment plan, which makes sense because they are protecting an HTTP accessible URL.

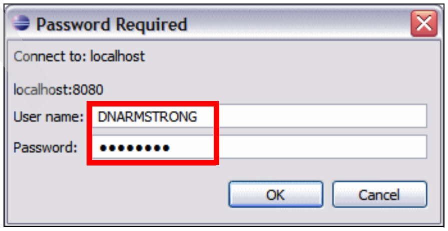
12.5.2 Test the access using the Web Services Explorer

In this section we use the Web Services Explorer to test the access:

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.
2. In the Java EE perspective, select **Run** → **Launch the Web Services Explorer**.
3. In the Web Services Explorer, click the **WSDL Page** icon [ ] in the upper right, and select **WSDL Main**.
4. In the Actions pane, enter the following URL and click **Go**:

```
http://localhost:8080/DonateBeanService/DonateBean?wsdl
```

In the Password Required pop-up, login as `DNARMSTRONG` with the proper password.



Connect to: localhost

localhost:8080

User name: DNARMSTRONG

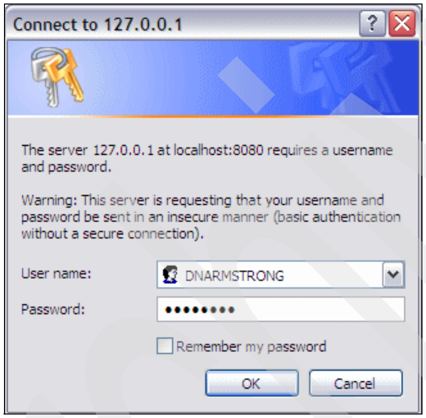
Password: ••••••••

Note:
The prompt can be hidden behind other windows, if you use an external browser.

Off course?

This prompt indicates that access to the `DonateBeanService` has been secured. Note that the prompt is displayed by Eclipse and not by the browser. This means that this authentication is for the Web Services Explorer back-end application. If you do not receive this prompt, try to switch to the Eclipse window, verify the preceding steps, and ensure that you published the updated project.

5. In the Actions pane, select the **donateToFund** operation.
6. Enter arguments **1**, **DonationFund**, **1**, and click **Go**.
7. In the Connect prompt, login as `DNARMSTRONG`.



Connect to 127.0.0.1

The server 127.0.0.1 at localhost:8080 requires a username and password.

Warning: This server is requesting that your username and password be sent in an insecure manner (basic authentication without a secure connection).

User name: DNARMSTRONG

Password: ••••••••

□ Remember my password

Behind the scenes:

The first login prompt was the result of the Web Services Explorer application accessing the WSDL and schema files from the secured DonateEAR application. The resulting user credentials were stored with the Web Services Explorer application, but not in the browser.

This second login prompt is the result of the browser window (pane within WASCE IDE) accessing the secured DonateEAR application. The resulting user credentials are stored in the browser component and not with the Web Services Explorer application.

8. The Status pane displays the `Transfer successful` message.
9. Close the Web Services Explorer.

Behind the scenes:

The HTTP basic authentication only occurs once when using a browser. After you enter the user ID and password and get successfully authenticated, the browser store the credentials and automatically include them in the follow up requests during an HTTP session. Only when a new HTTP session is created, the browser displays the authentication prompt again.

If you reload the WSDL in the Web Service Explorer, the prompt for the user ID is issued again on the invocation of the Web service.

12.6 Update the Web service client to support secure access

In the Open EJB deployment descriptor there is no way to define what type of HTTP method it protects; all of the methods are protected. The consequence is that even the WSDL URL gets protected for the HTTP GET method.

This feature makes the task of creating a Web service client more complex, because the **`wsimport`** tool does not implement the HTTP basic authentication to access a WSDL. The same issue happens when the `javax.xml.ws.Service` implementation class used by the Web service client is instantiated. The `Service` class has no means to create a URL connection using HTTP basic authentication.

If we keep the WSDL protected URL in a Web service client, application deployment fails during the Web service reference binding with this error:

```
Failed to read wsdl document
   org.apache.geronimo.common.DeploymentException: Failed to read wsdl
   document at org.apache.geronimo.jaxws.builder.EndpointInfoBuilder.build
   (EndpointInfoBuilder.java:155) at
   org.apache.geronimo.axis2.builder.Axis2ServiceRefBuilder.createService
   (Axis2ServiceRefBuilder.java:69) at
   org.apache.geronimo.jaxws.builder.JAXWSServiceRefBuilder.buildNaming
   (JAXWSServiceRefBuilder.java:161)
   ...
Caused by: javax.wsdl.WSDLException: WSDLException: faultCode=OTHER_ERROR:
   Unable to resolve imported document at
   'http://localhost:8080/DonateBeanService/DonateBean?wsdl'.:
   java.io.IOException: Server returned HTTP response code: 401 for URL:
   http://localhost:8080/DonateBeanService/DonateBean?wsdl
   at com.ibm.wsdl.xml.WSDLReaderImpl.readWSDL(Unknown Source)
```

A workaround to create a client for EJB based Web services protected by HTTP basic authentication is to import the WSDL (and any attached XSD) file and make it available from another URL.

12.6.1 Import the DonateBeanService WSDL and XSD

To resolve the problem that the **wsimport** command cannot import a WSDL from URLs protected by basic authentication, we manually import the WSDL and its XSD file:

1. From a browser, invoke the URL:

   `http://localhost:8080/DonateBeanService/DonateBean?wsdl`

   – Provide the user ID and password if prompted.

   – Save the page as `DonateBeanService.wsdl` in a temporary directory.

2. From the same browser, invoke the URL:

   `http://localhost:8080/DonateBeanService/DonateBean?xsd=DonateBeanService_schema1.xsd`

   – Save the page as `DonateBeanService_schema1.xsd` in the same directory.

> **Important!** Ensure that the files are saved with a **.wsdl** and **.xsd** extension. and not a `.wsdl.xml` and `.xsd.xml` extension.

3. Import the files into the DonateWSClientWeb project:

   – In the Project Explorer, select **WebContent** (in `DonateWSClientWeb`) and select **File** → **Import**.

   – In the Import/Select pop-up, select **General** → **File System** and click **Next**.

   – In the Import/File system pop-up:

   - Set the From directory to the temporary directory used previously.
   - On the right side select **DonateBeanService_schema1.xsd** and **DonateBeanService.wsdl**.
   - The Into folder is already set to `DonateWSClientWeb/WebContent`.
   - Click **Finish**.

> **Tip:** You can also drag the two files from a Windows Explorer window into the `DonateWSClientWeb/WebContent` folder of the Project Explorer, or select the two files and **Copy**, then right-click **WebContent** and select **Paste**.

4. Update DonateBeanService.wsdl to reference the local copy of the schema.

   – In the Project Explorer, open **DonateBeanService.wsdl** (in DonateWSClientWeb/WebContent).

   – In the DonateBeanService.wsdl editor, switch to the **Source** tab.

   – Locate the **<xsd:schema>** stanza and change the <xsd:import> to schemaLocation="DonateBeanService_schema1.xsd":

```
<xsd:import namespace="http://impl.ejb.donate/"
      schemaLocation="~~http://localhost:8080/DonateBeanService~~
                     ~~/DonateBean?xsd=~~DonateBeanService_schema1.xsd"/>
```

5. (Optional) If you implemented the optional 11.9, “Create a Web service with complex data types” on page 352, update DonateBeanService.wsdl to remove the donateToFundComplexTypes operation:

   – Locate the <**portType** name="DonateBean"> stanza, and remove the donateToFundComplexTypes operation:

```
<portType name="DonateBean">
    <operation name="donateToFund">
      ......
    </operation>
    ~~<operation name="donateToFundComplexTypes">~~
      ~~<input message="tns:donateToFundComplexTypes"></input>~~
      ~~<output message="tns:donateToFundComplexTypesResponse"></output>~~
      ~~<fault message="tns:DonateWSException" name="DonateWSException">~~
      ~~</fault>~~
    ~~</operation>~~
</portType>
```

   – Do the same under <**binding** name="DonateBeanPortBinding"...>:

```
<binding name="DonateBeanPortBinding" type="tns:DonateBean">
    <soap:binding style="document" ......./>
    <operation name="donateToFund">
      ...
    ~~<operation name="donateToFundComplexTypes">~~
      ~~<soap:operation soapAction=""/>~~
      ~~<input> ... <output> ... <fault name="DonateWSException"> ...~~
    ~~</operation>~~
</binding>
```

6. The tailored WSDL file and the XSD file are available in 7639code\ws=security\wsdl.

7. Save and close DonateBeanService.wsdl.

Behind the scenes:

After we created the DonateWSClient, we added another Web service method (`donateToFundComplexTypes`) to the `DonateBean`. However, the stub classes generated for the client do not include that method, and when the client servlet is run, we would get an error message:

```
13:27:38,921 ERROR [[ClientServlet]] Allocate exception for servlet
ClientServlet. java.lang.InstantiationException: Some objects to be
injected were not found in jndi: [javax.naming.NamingException: Could not
look up : env/donate.client.ClientServlet/mybean [Root exception is
javax.xml.ws.WebServiceException: The Endpoint description validation
failed to validate due to the following errors:
 :: Invalid Endpoint Interface ::  :: The operation names in the WSDL
portType do not match the method names in the SEI or Web service
implementation class.]]
```

12.6.2 Update and test the Web services client

In the client side, the SEI, returned by the service class, implements the `BindingProvider` interface that allows the client to access the request context:

1. In the Project Explorer, open **ClientServlet** (in `DonateWSClientWeb/Java Resources:src/donate`.client).
2. In the `ClientServlet.java` editor, replace the entire try/catch block in the `doGet` method with the **F82 Web Service Client Servlet Security** snippet:
   - Update the two password properties with the correct passwords created for `DNARMSTRONG` and `DGADAMS` (for example, `password`):

```
try {
   DonateBean bean = service.getDonateBeanPort();
   out.print("Response using service class: ");
   ((BindingProvider)bean).getRequestContext().
         put(BindingProvider.USERNAME_PROPERTY, "DNARMSTRONG");
   ((BindingProvider)bean).getRequestContext().
         put(BindingProvider.PASSWORD_PROPERTY, "password");
   out.println(bean.donateToFund(1, "DonationFund", 1));
   out.print("Response using SEI directly: ");
   ((BindingProvider)bean).getRequestContext().
         put(BindingProvider.USERNAME_PROPERTY, "DGADAMS");
   ((BindingProvider)bean).getRequestContext().
         put(BindingProvider.PASSWORD_PROPERTY, "password");
   out.println(mybean.donateToFund(1, "DonationFund", 1));
} catch (Exception e) {
   out.println("EXCEPTION: " + e.getLocalizedMessage());
}
```

- Organize imports (**Ctrl+Shift+O**).
- Save and close `ClientServlet.java`.

3. In the Project Explorer, open **DonateBeanService** (in `DonateWSClientWeb /Java Resources:src/donate.ejb.impl`).
4. In the `DonateBeanService.java` editor, locate the **@WebServiceClient** annotation before the class definition, and change the location of the WSDL:

```
From: @WebServiceClient(name = "DonateBeanService",
         targetNamespace = "http://impl.ejb.donate/", wsdlLocation =
         "http://localhost:8080/DonateBeanService/DonateBean?wsdl")
To:   @WebServiceClient(name = "DonateBeanService",
         targetNamespace = "http://impl.ejb.donate/", wsdlLocation =
         "DonateBeanService.wsdl")
```

   The WSDL file is found in the default path, and in the case of a Web project this is the `WebContent` directory.

5. Save and close `DonateBeanService.java`.
6. Publish the updated application.
7. Start a browser and open the following URL:

   `http://localhost:8080/DonateWSClientWeb/ClientServlet`

   The result of the Web service client shows a successful invocation and a failure:

```
Calling DonateBean Web service...
Response using service class: Transfer successful
Response using SEI directly: EXCEPTION: org.apache.axis2.AxisFault:
                             Transport error: 401 Error: Unauthorized
```

Behind the scenes:

We use two different user IDs to demonstrate role based security:

- `DNARMSTRONG` is granted access and the first Web service request returns `Transfer successful`.
- `DGADAMS` is denied access and the Web service request returns an `Unauthorized` error. This authorization error is generated by the Web container, we can see the HTTP 401 return code in the message.

Note that the authorization was done by the Web container, where the POJO-based Web service is defined, but the credentials were also propagated to the EJB container. This can be verified because the EJB method requires the requester to have the `DUSERS` or `DMANAGERS` role.

Off course?

If the Web service request fails, make sure that the Web service's exposed methods were not changed since you have imported the WSDL.

If they are different, the client request cannot be bound to the Web service operation, resulting in the following exception:

```
javax.xml.ws.soap.SOAPFaultException: The endpoint reference (EPR) for
the Operation not found is /DonateBeanService/DonateBean and the WSA
Action = org.apache.axis2.jaxws.marshaller.impl.alt.MethodMarshallerUtils
   .createSystemException(MethodMarshallerUtils.java:1192)
org.apache.axis2.jaxws.marshaller.impl.alt.MethodMarshallerUtils
   ......
$Proxy191.donateToFund(Unknown Source)
donate.client.ClientServlet.doGet(ClientServlet.java:58)
javax.servlet.http.HttpServlet.service(HttpServlet.java:693)
javax.servlet.http.HttpServlet.service(HttpServlet.java:806)
```

The solution to this problem is to make the Web service client match the Web service server methods. Refer to 11.7, "Create a top-down Web service" to match them.

12.7 Enable SSL for the Web service communication

HTTP basic authentication is not a secure method. It assumes that the connection between the server and the client is secure and the user ID and password can be easily intercepted. So it is important to have an SSL/HTTPS connection when using this authentication method.

12.7.1 Create an SSL certificate

We have to generate an SSL certificate to be used by WASCE. This is required because the default SSL certificated used by WASCE must be associated with the current domain. In our project, we use the domain as being `localhost`, and we use a self-signed certificate.

To create a certificate for the WASCE server, we use the **`keytool`** command provided with the IBM JDK:

1. Open a command window and switch to `C:\Program Files\IBM\Java50\bin`.
2. Generate a new keystore with a `ws` alias in it, and set the first and last name to `localhost` (`changeit` can be any password):

```
keytool -genkey -alias ws -storepass changeit -keystore mystore
```

3. When prompted for data:

   – What is your first and last name?
   [Unknown]: **localhost**
   – Leave the other fields with default value.
   – Is `CN=localhost, OU=....` correct? (type “yes” or “no”): **yes**
   – Enter key password for <ws>: (RETURN if same as keystore password):

Off course?

If the command fails with an error similar to the following one:

```
keytool error (likely untranslated): java.io.FileNotFoundException:
mystore1 (Access is denied.)
```

You do not have write access to `C:\Program Files\IBM\Java50\bin`. The workaround is to update the command to write to a directory where you do have write access:

```
keytool -genkey -alias ws -storepass changeit
         -keystore c:\temp\mystore
```

Note that you also have to update both the `-keystore` and `-file` arguments on the next command.

4. Extract the keystore certificate:

   ```
   keytool -export -keystore mystore -alias ws -file ws.cer
   ```

   – Enter keystore password: **changeit**

   ```
   Certificate stored in file <ws.cer>
   ```

5. Copy the two files **mystore** and **ws.cer** into `<WASCE_HOME>/var/security/keystores/`.

6. For the client side, we add the certificate to the Java runtime trust store:

   – Run the following command from `C:\Program Files\IBM\Java50\bin`:

   ```
   keytool -import -alias ws -file ws.cer
                                  -keystore ../jre/lib/security/cacerts
   ```

   – Enter keystore password: **changeit**

   ```
   Owner: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=.., C=..
   Issuer: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=.., C=..
   Serial number: 48566c96 (or similar)
   Valid from: xx/xx/08 xx:xx AM until: xx/xx/08 xx:xx AM
   Certificate fingerprints:
            MD5:  xx:xx:........
            SHA1: xx:xx:........
   ```

– Trust this certificate? [no]: **yes**

```
Certificate was added to keystore
```

Off course?

If the command fails with an error similar to the following one:

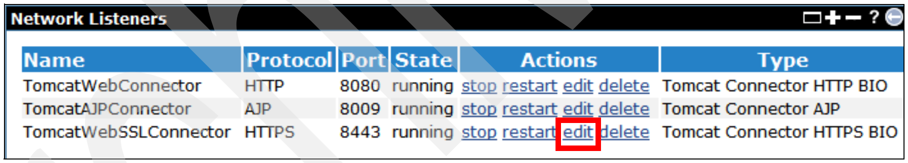
```
keytool error (likely untranslated): java.io.FileNotFoundException:
..\jre\lib\security\cacerts (Access is denied.)
```

Make sure that you have write access to `.\jre\lib\security\cacerts`.

There is no workaround to this issue: You must issue this command as a user that has write access to the `cacerts` file:

- Windows Any user that belongs to the local Administrators group
- Linux Any user with read write access to this directory

7. Open the WASCE Console (right-click the server and select **Launch WASCE Console**) and login.
8. Select **Server** → **Web Server** and click **edit** in the TomcatWebSSLConnector row.



Network Listeners

Name	Protocol	Port	State	Actions	Type
TomcatWebConnector	HTTP	8080	running	stop restart edit delete	Tomcat Connector HTTP BIO
TomcatAJPConnector	AJP	8009	running	stop restart edit delete	Tomcat Connector AJP
TomcatWebSSLConnector	HTTPS	8443	running	stop restart edit delete	Tomcat Connector HTTPS BIO

9. Set the following fields:
   – keystoreFile: **var/security/keystores/mystore**
   – keyAlias: **ws**
   – keystorePass: **changeit**
10. Click **Save**.
11. Back in the Network Listeners list, click **restart** for TomcatWebSSLConnector.

Off course?

If the command to add the certificate to the truststore fails, make sure that the certificate file is in the same directory, or change the `-file` argument to point to the absolute path of the certificate.

Behind the scenes:

The default certificate cannot be used by a client because the domain name does not match with the server domain name. Therefore, we create a keystore and a certificate associated with the localhost.

For the client, we add the certificate to Java runtime truststore. This is required because we do not have an actual certificate authority for the generated certificate. It has been self-signed.

12.7.2 Change the application to use SSL

To enable SSL, all that is required is to change the value of the `<transport-guarantee>` tag from `NONE` to `CONFIDENTIAL`. This is valid for EJB or POJO Web services:

1. In the Project Explorer, open **openejb-jar.xml** (in `DonateEJB/ejbModule/META-INF`):

   – Change the transport from `NONE` to `CONFIDENTIAL`:

   ```
   <ejb:transport-guarantee>CONFIDENTIAL</ejb:transport-guarantee>
   ```

   – Save and close `openejb-jar.xml`.

2. Publish the updated application, and run the client servlet:

   ```
   http://localhost:8080/DonateWSClientWeb/ClientServlet
   ```

   The Web service fails with the error:

   ```
   Calling DonateBean Web service...
   Response using service class: EXCEPTION: org.apache.axis2.AxisFault:
                                 Transport error: 403 Error: Forbidden
   ```

3. Open **DonateBeanService.wsdl** (in `DonateWSClientWeb/WebContent`):

   – Change the endpoint address to the SSL value (change `http` to `https`, and `8080` to `8443`).

   ```
   <soap:address
      location="https://localhost:8443/DonateBeanService/DonateBean"/>
   ```

   – Save and close `DonateBeanService.wsdl`.

4. Open **DonateBeanService** (in `DonateWSClientWeb/Java Resources: src/donate.ejb.impl`):

   – Change the URL to use the HTTPS protocol:

   ```
   url = new
         URL("https://localhost:8443/DonateBeanService/DonateBean?wsdl");
   ```

   – Save and close `DonateBeanService.java`.

5. Publish the updated application, and run the client servlet again. The Web service shows the following messages:

```
Calling DonateBean Web service...
Response using service class: Transfer successful
Response using SEI directly: EXCEPTION: org.apache.axis2.AxisFault:
                      Transport error: 401 Error: Unauthorized
```

   The first call is successful, because the user `DNARMSTRONG` is authorized. The second call fails, because the user `DGADAMS` is not authorized.

6. You can also run the Web Services Explorer with the changed endpoint, and you will be prompted to enter a user ID and password. Use this URL:

```
https://localhost:8443/DonateBeanService/DonateBean?wsdl
```

7. Reverse the changes to remove the secure connection:

   – `openejb-jar.xml` (in `DonateEJB/ejbModule/META-INF`):

```
<ejb:transport-guarantee>NONE</ejb:transport-guarantee>
```

   – `DonateBeanService.wsdl` (in `DonateWSClientWeb/WebContent`):

```
<soap:address
   location="http://localhost:8080/DonateBeanService/DonateBean"/>
```

   – `DonateBeanService` (in `DonateWSClientWeb/Java Resources:src /donate.ejb.impl`):

```
url = new
      URL("http://localhost:8080/DonateBeanService/DonateBean?wsdl");
```

12.8 Secure a Web container-based Web service

In this section we perform the tasks required to protect a POJO based Web service using basic authentication. Then we create a client to access the Web service.

We assume that the security realm is defined to authenticate the user credentials and map the users to their roles. Refer to Chapter 10, “Implement core security” on page 285 for instructions.

We use the POJO based Web service created in 11.7, “Create a top-down Web service” on page 345.

12.8.1 Configure the Web service for security

The first step is to only allow authenticated user access to the top-down Web service created in the previous chapter. To do it we change the Web deployment descriptor by adding security to the Web service URL:

1. In the Project Explorer, open **geronimo-web.xml** (in `DonateTopDownWeb/WebContent/WEB-INF`). Add the realm definition before the ending tag:

```
   <security-realm-name>ExperienceJavaEERealm</security-realm-name>
</web:web-app>
```

   – Save and close `geronimo-web.xml`.

2. In the Project Explorer, open **web.xml** (in `DonateTopDownWeb/WebContent/WEB-INF`).

   – In the `web.xml` editor, switch to the **Source** tab, and insert the **F83 Web Service Top Down XML Security** snippet from the Snippets view before the ending tag (Example 12-2).

Example 12-2 Add security to top-down Web service

```
   <login-config>
       <auth-method>BASIC</auth-method>
       <realm-name>ExperienceJavaEERealm</realm-name>
   </login-config>
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Top down WS security</web-resource-name>
         <url-pattern>/DonateTopDownWS</url-pattern>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <role-name>DMANAGERS_ROLE</role-name>
      </auth-constraint>
      <user-data-constraint>
         <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
   </security-constraint>
</web-app>
```

Note that we only add the managers role (`DMANAGERS_ROLE`), so the Web service should fail for all non-managers.

   – Save and close `web.xml`.

Behind the scenes:

Similar to the process of protecting a servlet or JSP (as we did for DonateJSFWeb), the `<security-constraint>` tag is used to secure a Web service endpoint URL.

The URL pattern added exactly matches the Web service endpoint URL. Note that only the HTTP `POST` method is being protected, SOAP 1.1 specifies that only `POST` is used to execute the service. Therefore, we allow any client to access the WSDL with a HTTP `GET` request with no authentication required.

The `<role-name>` tag specifies that only authenticated users with the `DMANAGERS_ROLE` role are allowed to access this protected resource.

The `<login-config>` tag indicates that HTTP basic authentication is used, and specifies the realm that is used to authenticate the users.

12.8.2 Test the access using the Web Services Explorer

We use the Web Services Explorer to test the access to the top-down Web service:

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.
2. In the Java EE perspective, select **Run** → **Launch the Web Services Explorer**.
3. In the Web Services Explorer browser, click the **WSDL Page** icon [ ] in the upper right, and select **WSDL Main**.
4. In the Actions pane, enter the following URL and click **Go**.

   ```
   http://localhost:8080/DonateTopDownWeb/DonateTopDownWS?wsdl
   ```

5. Select the **donateToFund** operation, enter arguments (1, **DonationFund**, 1), and click **Go**.
6. An HTTP basic authentication form is displayed. Enter the user ID **`DNARMSTRONG`** and the matching password.
7. Note the message in the Status pane, indicating a successful result from the top-down Web service.
8. Repeat the test in a new browser window, or clear the WSDL page and reload the WSDL file in the internal Web Services Explorer, and use the user ID **`DCBROWN`** (who is not a manager), and the Web service call fails with the error:

   ```
   IWAB0135E An unexpected error has occurred. 403 Forbidden
   ```

12.9 Explore!

In the Experience section, we have protected the `DonateBean` Web service using HTTP basic authentication as the JAX-WS 2.0 specifies. This authentication method is in the HTTP protocol level, and not a message based end-to-end security.

The most widely known specification for Web services end-to-end security is defined by WS-Security. Even though WS-Security is not natively supported by WASCE, it is possible to implement it using Axis2 and the `rampart` module.

> **Note:** Apache Axis2 is a core engine for Web services. Axis2 not only provides the capability to add Web services interfaces to Web applications, but can also function as a standalone server application.
>
> When using Axis2, it creates a separate container to run the Web service. **Because the Web service runs in a separate container, WASCE cannot access it to perform any injections of annotated artifacts, and Axis2 does not have access to the WASCE JNDI tree.** A workaround for this is to have Axis2 connect to the WASCE EJB container as a client as described in Chapter 4, “Prepare the legacy application” on page 77.

12.9.1 Create a secure Web service with Axis2

WASCE does not support the WS-Security standard, however, using an Axis2 container it is possible to deploy a Web service secured using WS-Security.

Set up the Axis2 runtime in Eclipse

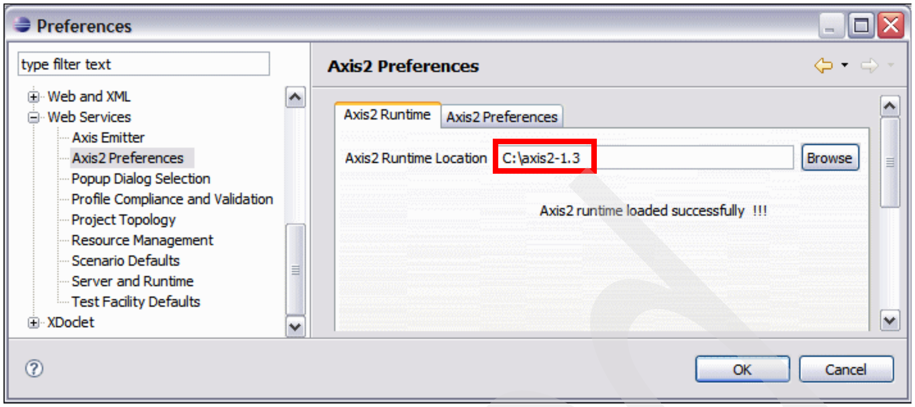
The first step is to install the Axis2 runtime to be used in the WASCE IDE and enabling the tooling for creating Web services in Axis2:

1. Download Axis2 1.3 binaries (`axis2-1.3-bin.zip`) from the following URL:

   `http://ws.apache.org/axis2/download.cgi`

   > **Note:** You must use Axis2 1.3, not Axis2 1.4.

2. Unzip the downloaded archive (to the C drive).
3. In Eclipse, select **Windows** → **Preferences**.
4. In the Preferences pop-up, select **Web services** → **Axis2 Preferences**.
   - Click **Browse** and locate the unzipped Axis2.
   - Click **OK**.



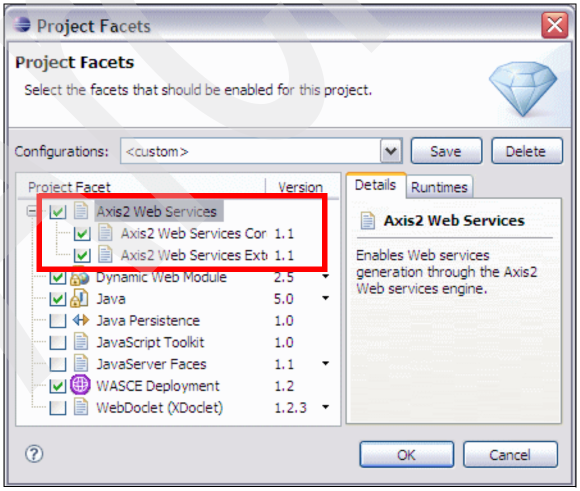
Axis2 Runtime Location C:\axis2-1.3

Axis2 runtime loaded successfully !!!

Create a Web service project with Axis2

Axis2 Web services are exposed using a Web project. To create the Web project, follow these instructions:

1. In the Project Explorer, right-click anywhere in the view and select **New** → **Dynamic Web Project**.
2. In the New Dynamic Web Project pop-up, set the Project Name to **Axis2SecureWSWeb**, and the EAR project as **DonateEAR**.
   - For Configuration, click **Modify**.
   - In the New Dynamic Web Projects Facets pop-up, besides the default selections, select **Axis2 Web Services**, and click **OK**.



Configurations: <custom>

Project Facet	Version
☑ Axis2 Web Services	
☑ Axis2 Web Services Cor	1.1
☑ Axis2 Web Services Ext	1.1
☑ Dynamic Web Module	2.5
☑ Java	5.0
□ Java Persistence	1.0
□ JavaScript Toolkit	1.0
□ JavaServer Faces	1.1
☑ WASCE Deployment	1.2
□ WebDoclet (XDoclet)	1.2.3

Axis2 Web Services

Enables Web services generation through the Axis2 Web services engine.

3. Back in the New Dynamic Web Project pop-up, click **Next**.
4. Select **Generate Deployment Descriptor**, and click **Next**.
5. Set the Group Id to **ejee**, and click **Finish**.

Behind the scenes:

By adding the Axis2 Web Services facet to the project, the IDE includes with the WAR project an Axis2 container. Inspect the project by expanding the `WebContent` and the `WEB-INF` folders:

- `axis2-web`: A Web application
- `conf`: Axis1 configuration file (`axis2.xml`)
- `lib`: Runtime JAR files
- `modules`: Axis container modules
- `services`: Deployed services
- `web.xml`: Axis servlets

The two servlets are:

`AxisServlet`	Intercepts and processes the service for exposing Web services
`AxisAdminServlet`	Administration tool for controlling the Axis2 container

6. Right-click the **Axis2SecureWSWeb** project and select **Properties**. For Java EE Module Dependencies, select **DonateEJB.jar**, and click **OK**. We want to call the session bean to perform the logic.

Configuration for WASCE

Some specific actions are required to integrate this project with WASCE:

1. Delete the `mex-1.3.mar` file from `Axis2SecureWSWeb/WebContent/WEB-INF /modules`.
2. Open **geronimo-web.xml** (in `Axis2SecureWSWeb/WebContent/WEB-INF`**)**. After the `<moduleId>` tag, add two hidden classes filters (Example 12-3).

Example 12-3 Geronimo-web.xml deployment plan

```
<web:web-app .............>
    <dep:environment>
        <dep:moduleId>
            ...
        </dep:moduleId>
        <dep:hidden-classes>
          <dep:filter>org.jdom</dep:filter>
          <dep:filter>org.apache.axis2</dep:filter>
       </dep:hidden-classes>
    </dep:environment>
    <web:context-root>/AxisSecureWSWeb</web:context-root>
</web:web-app>
```

Add the rampart security module

By default, Axis2 does not include a security module. To include and set up a security module, follow these instructions:

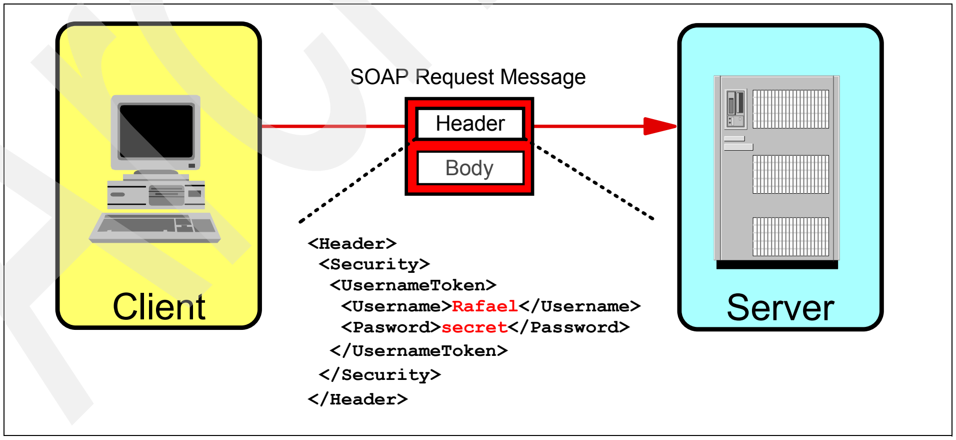
1. Download the rampart 1.3 module from the Axis2 modules URL:

   http://www.apache.org/dyn/mirrors/mirrors.cgi/ws/rampart/1_3/rampart-1.3.zip
2. Extract two MAR files from `rampart-1.3`: `rampart-1.3.mar, rahas-1.3.mar`
3. Extract all seven JAR files from `rampart-1.3\lib`.
4. Copy (or import) the `rampart-1.3.mar` and `rahas-1.3.mar` files into `Axis2SecureWSWeb/WebContent/WEB-INF/modules`.
5. Copy (or import) the seven JAR files into `Axis2SecureWSWeb/WebContent /WEB-INF/lib`.

Prepare a JavaBean for the Web service

Now we create a Web service that acts as a facade to the `DonateBean` EJB. Creating an Axis2 Web service is easily done by using the Eclipse tooling. We protect this Web service using UserNameToken Authentication, which is one of the mechanisms defined by WS-Security. It works similarly to the HTTP basic authentication, however, now the security is at the SOAP message level (Figure 12-3).

<



```
<Header>
 <Security>
  <UsernameToken>
   <Username>Rafael</Username>
   <Pasword>secret</Password>
  </UsernameToken>
 </Security>
</Header>
```

Figure 12-3 UsernameToken authentication

The password can be represented in plain text or a digest of it. Here we use the plain text representation:

1. In the AxisSecureWSWeb project, create a **SecureDonateBean** class in a new **donate.web.secure** package.
2. Add the @EJB annotation for `DonateBean` to inject the session bean:

   ```
   @EJB DonateBeanRemote donateBean;
   ```

3. Add the `lookupDonateRemote` method to locate the session bean (Example 12-4, snippet **F85 Axis Remote Lookup**). For more information about EJB 3.0 clients, refer to the Learn section in Chapter 6, “Create the entity facade session beans” on page 165.

Example 12-4 Method for remote EJB lookup

```
private DonateBeanRemote lookupDonateRemote() throws NamingException {
   InitialContext ctx;
   DonateBeanRemote donateBean;
   Hashtable props = new Hashtable();
   props.put(Context.INITIAL_CONTEXT_FACTORY,
             "org.openejb.client.RemoteInitialContextFactory");
   props.put(Context.PROVIDER_URL, "ejbd://localhost:4201");
   ctx = new InitialContext(props);
   donateBean = (DonateBeanRemote) ctx.lookup("DonateBeanRemote");
   return donateBean;
}
```

4. Add a `donateSecurely` method to perform the service (Example 12-5, snippet **F86 Axis Donate Secure**).

Example 12-5 SecureDonateBean class

```
public String donateSecurely(int employeeId, String fundName, int hours)
          throws Exception {
   DonateBeanRemote donate = lookupDonateRemote();
   return donate.donateToFund(employeeId, fundName, hours);
}
```

5. Organize imports (**Ctrl+Shift+O**).

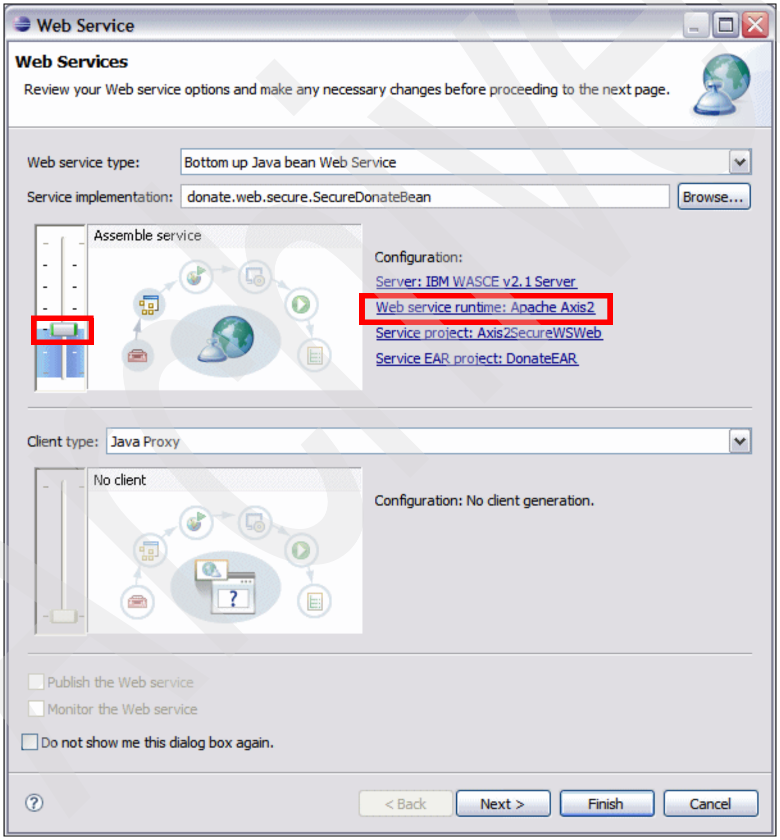
   ```
   java.util.Hashtable
   javax.naming.Context
   ```

6. Save and close `SecureDonateBean.java`.

Run the Web Service wizard

We use the Web Service wizard to create the Web service from the JavaBean:

1. Right-click **SecureDonateBean** and select **Web Services** → **Create Web Service**.
2. In the Web Service pop-up (Figure 12-4), the Web service type is selected as **Bottom up Java bean Web Service** and the `SecureDonateBean` class is set in the Service implementation field.
3. Change the level of the service generation to **Assemble service**, so we avoid automatic deployment of the service.
4. Click on the **Web service runtime** link in the Configuration section. In the Service Deployment Configuration pop-up, select **Apache Axis2**, and click **OK**.



Web Service

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

Web service type: Bottom up Java bean Web Service

Service implementation: donate.web.secure.SecureDonateBean

Assemble service

Configuration:
Server: IBM WASCE v2.1 Server
Web service runtime: Apache Axis2
Service project: Axis2SecureWSWeb
Service EAR project: DonateEAR

Client type: Java Proxy

No client

Configuration: No client generation.

□ Publish the Web service
□ Monitor the Web service
□ Do not show me this dialog box again.

Figure 12-4 Web Services wizard

5. Click **Next**.

6. In the Axis2 Web Service Java Bean Configuration, select **Generate default services.xml file**.
7. Click **Finish**.

> **Behind the scenes:**
>
> The Web Service wizard creates an Axis2 Web service by generating the Axis2 service files under `WEB-INF/services` into a folder with the service name. Inside that folder, the wizard generates the class files of this service and a `META-INF` directory that contains the `services.xml` file.

Configure the Axis2 Web service with rampart

Now we configure Axis2 to secure the Web service, which is done by configuring the `rampart` module:

1. Open the generated **services.xml** (in `Axis2SecureWSWeb/WebContent/WEB-INF/services/SecureDonateBean/META-INF`).
2. In the `services.xml` editor, add the `rampart` module and the security parameters for UsernameToken and a callback handler. Update the description and insert the **F87 Axis Rampart Services** snippet from the Snippets view before the ending `</service>` tag (Example 12-6).

Example 12-6 services.xml configuration to enable WS-Security

```
<service name="SecureDonateBean" >
   <Description>
      Secure Donation Service
   </Description>
   <messageReceivers>
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
         class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
      <messageReceiver  mep="http://www.w3.org/2004/08/wsdl/in-out"
         class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
   </messageReceivers>
   <parameter name="ServiceClass" locked="false">
                         donate.web.secure.SecureDonateBean</parameter>
   <module ref="rampart" />
   <parameter name="InflowSecurity">
      <action>
         <items>UsernameToken</items>
         <passwordCallbackClass>donate.web.secure.handler
                         .UserTokenHandlerFlowIn</passwordCallbackClass>
      </action>
   </parameter>
</service>
```

In the `services.xml` configuration, we add a callback handler to process any income message:

```
donate.web.secure.handler.UserTokenHandlerFlowIn
```

3. Create the handler class (**New** → **Class**) with a package name of **donate.web.secure.handler** and name **UserTokenHandlerFlowIn**.
4. Edit the `UserTokenHandlerFlowIn` class by replacing the class with the **F88 Axis User Token Handler** snippet (Example 12-7).

Example 12-7 UsernameToken callback handler

```
public class UserTokenHandlerFlowIn implements CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException,
                                       UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];
         if (!(pwcb.getIdentifer().equals("admin") &&
               pwcb.getPassword().equals("pwd"))) {
            throw new UnsupportedCallbackException(callbacks[i],
                                                    "Invalid credentials");
         }
      }
   }

}
```

5. Organize imports (**Ctrl+Shift+O**).

   ```
   javax.security.auth.callback.Callback
   ```

6. Save and close `UserTokenHandlerFlowIn.java`.

Test the Axis2 Web service with security enabled

Now we test the created Axis2 Web service with security enabled:

1. Ensure that the ExperienceJavaEE Server is running.
2. Deploy the DonateEAR, which now contains the Axis2SecureWSWeb project, by publishing the server.

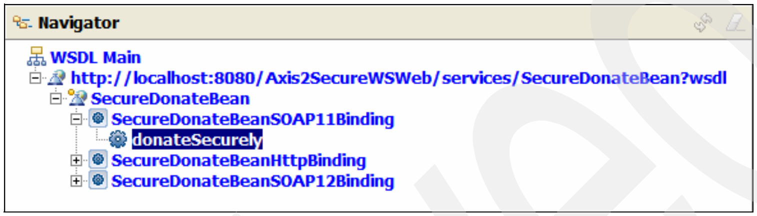
Off course?

If publishing fails, you cannot delete the DonateEAR folder from `<WASCE_HOME>/repository/ejee`. You have to stop the server, delete the folder, then restart the server.

3. Launch the Web Services Explorer as explained in Chapter 11, “Create the Web service” on page 323 (**Run → Launch the Web Services Explorer**).
4. Click the **WSDL Page** icon, and in for WSDL Main, add the SecureDonateBean WSDL URL:

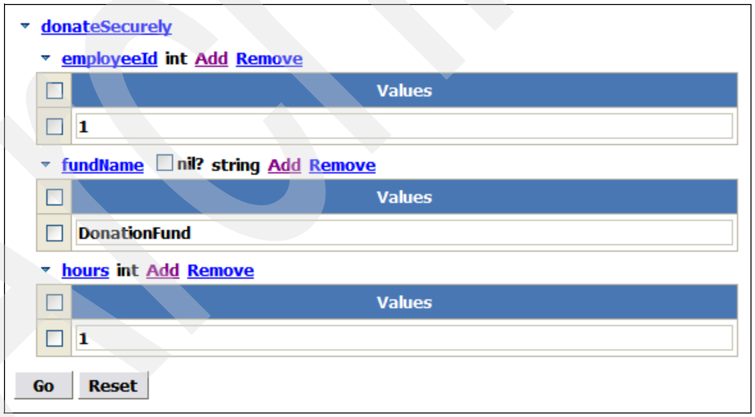
```
http://localhost:8080/Axis2SecureWSWeb/services/SecureDonateBean?wsdl
```

5. Three bindings are listed for accessing this Web service, but only SOAP 1.1 was created when using the Web Services wizard. Select **SecureDonateBeanSOAP11Binding → donateSecurely**.



```
WSDL Main
⊟ http://localhost:8080/Axis2SecureWSWeb/services/SecureDonateBean?wsdl
  ⊟ SecureDonateBean
    ⊟ SecureDonateBeanSOAP11Binding
        donateSecurely
    ⊞ SecureDonateBeanHttpBinding
    ⊞ SecureDonateBeanSOAP12Binding
```

6. In the Actions pane, which now displays Invoke a WSDL Operation, complete the three fields (by clicking **Add** for each field):
   – arg0 (employee ID): **1**
   – arg1 (fund name): **DonationFund**
   – arg2 (donation amount): **1**
   – Click **Go**.



donateSecurely

employeeId int Add Remove

□	Values
□	1

fundName □ nil? string Add Remove

□	Values
□	DonationFund

hours int Add Remove

□	Values
□	1

7. The Status pane displays that the message returned cannot be shown. Click **Source** on the top right of the Status pane.
8. The SOAP Request Envelope contains the SOAP message sent to the server (Example 12-8) and the SOAP Response Envelope contains the message received from the server (Example 12-9).

Example 12-8 Request envelop

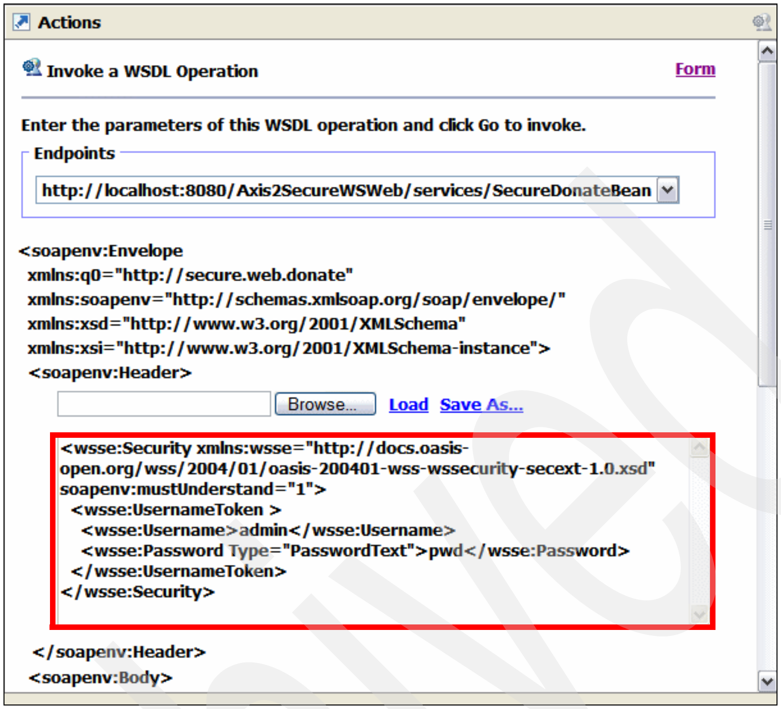
```
<soapenv:Envelope xmlns:q0=........>
<soapenv:Body>
   <q0:donateSecurely>
      <q0:employeeId>1</q0:employeeId>
      <q0:fundName>DonationFund</q0:fundName>
      <q0:hours>1</q0:hours>
   </q0:donateSecurely>
</soapenv:Body>
</soapenv:Envelope>
```

Example 12-9 Response envelop

```
<soapenv:Envelope xmlns:soapenv="....">
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Fault
            xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <faultcode>soapenv:Server</faultcode>
      <faultstring>WSDoAllReceiver: Incoming message does not contain
                                    required Security header</faultstring>
      <detail />
   </soapenv:Fault>
</soapenv:Body>
</soapenv:Envelope>
```

Note that the response fault message indicates that the service requires message authentication. To actually execute the service, we have to add the UsernameToken header to the request SOAP message.

9. In the Actions pane, click **Source** on the top right.
10. In the Actions pane, you can now set the SOAP headers in the first text field. Insert the UsernameToken security tags.



Enter the parameters of this WSDL operation and click Go to invoke.

Endpoints

http://localhost:8080/Axis2SecureWSWeb/services/SecureDonateBean

```
<soapenv:Envelope
 xmlns:q0="http://secure.web.donate"
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header>
```

```
<wsse:Security xmlns:wsse="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
soapenv:mustUnderstand="1">
  <wsse:UsernameToken >
    <wsse:Username>admin</wsse:Username>
    <wsse:Password Type="PasswordText">pwd</wsse:Password>
  </wsse:UsernameToken>
</wsse:Security>
```

```
 </soapenv:Header>
 <soapenv:Body>
```

11. Example 12-10 shows the `<wsse:Security>` header tags (copy/paste from `C:\7639code\snippets\webservicesecurity\`**`F89 Axis Security Header`**).

Example 12-10 UsernameToken header tags

```
<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
   oasis-200401-wss-wssecurity-secext-1.0.xsd" soapenv:mustUnderstand="1">
   <wsse:UsernameToken >
      <wsse:Username>admin</wsse:Username>
      <wsse:Password Type="PasswordText">pwd</wsse:Password>
   </wsse:UsernameToken>
</wsse:Security>
```

12. The second text field is already filled, with the last service request, (Example 12-11).

Example 12-11 SOAP request body

```
<q0:donateSecurely>
  <q0:employeeId>1</q0:employeeId>
  <q0:fundName>DonationFund</q0:fundName>
  <q0:hours>1</q0:hours>
</q0:donateSecurely>
```

13. Click **Go**.

14. You get this SOAP fault:

```
<faultstring>Unauthorized Access by Principal Denied</faultstring>
```

The Web service has reached the `DonateBean` session bean, but security is enabled and user credential are not valid.

The Web service should run successfully if security in the EJB is disabled.

> **Behind the scenes:**
>
> The Axis2 container is processing the Web service request, it pre-processes the message with the callback handlers defined in the `services.xml` file.

15. If you run the example with an invalid user ID or password, then you get this SOAP fault:

```
faultstring>WSDoAllReceiver: security processing failed</faultstring>
```

In this case the Web service call is rejected by the callback handler.

> **Alternative:**
>
> Using UsernameToken with a plain text password is not secure because the password can be easily retrieved from the message when it is being transmitted in the network. The first simple alternative is to encrypt the password in that message. Other authentication tokens are more reliable, for example, you can use a X.509 certificate based token or a Security Assertion Markup Language (SAML) token profile.
>
> Axis2 with `rampart` supports these other alternatives and can be useful for protecting a Web service.

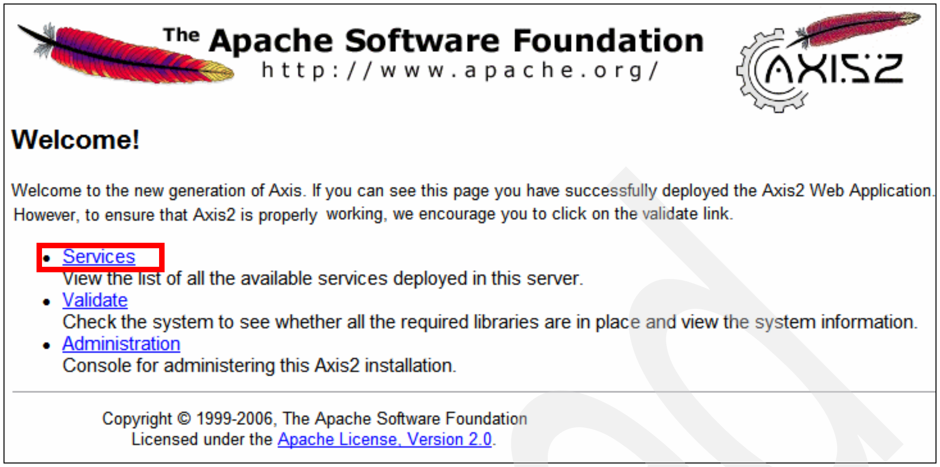
Axis2 console

The Web application in `Axis2SecureWSWeb/WebContent/axis2-web` can be used to administer Axis2:

1. Start a browser using this URL:

```
http://localhost:8080/Axis2SecureWSWeb/axis2-web/index.jsp
```

2. The Axis2 console starts; be patient (Figure 12-5).



Welcome!

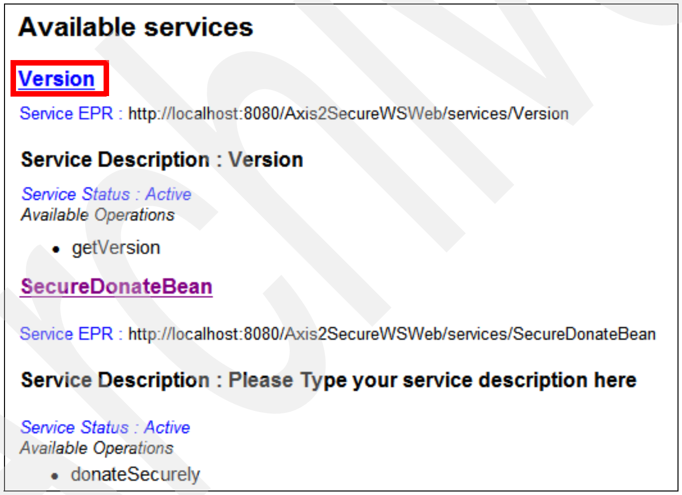
Welcome to the new generation of Axis. If you can see this page you have successfully deployed the Axis2 Web Application. However, to ensure that Axis2 is properly working, we encourage you to click on the validate link.

- Services
  View the list of all the available services deployed in this server.
- Validate
  Check the system to see whether all the required libraries are in place and view the system information.
- Administration
  Console for administering this Axis2 installation.

Copyright © 1999-2006, The Apache Software Foundation
Licensed under the Apache License, Version 2.0.

Figure 12-5 Axis2 console

3. Click **Services**, and the Web service is displayed.



Available services

Version

Service EPR : http://localhost:8080/Axis2SecureWSWeb/services/Version

Service Description : Version

Service Status : Active
Available Operations

- getVersion

SecureDonateBean

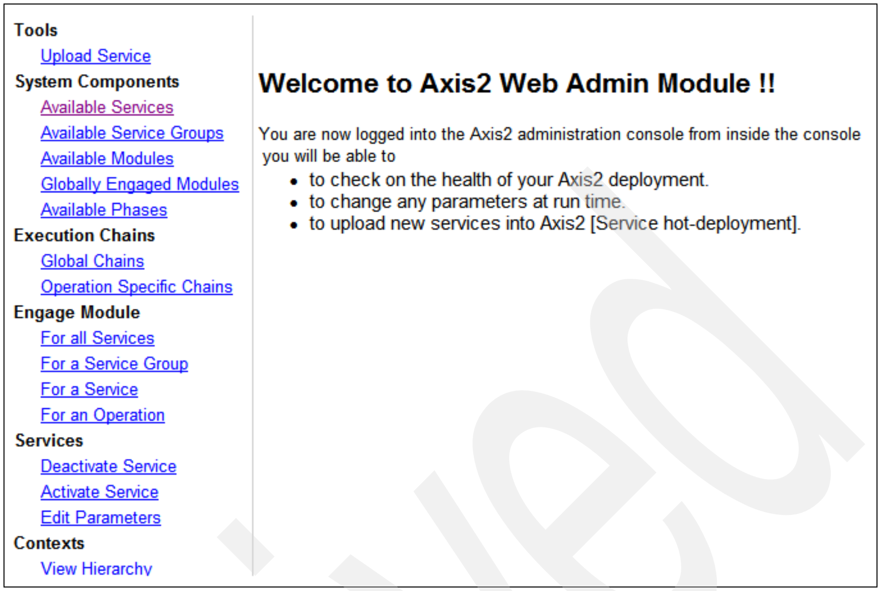
Service EPR : http://localhost:8080/Axis2SecureWSWeb/services/SecureDonateBean

Service Description : Please Type your service description here

Service Status : Active
Available Operations

- donateSecurely

4. Click **Version**, and the WSDL is displayed. Go back in the browser.
5. Click **Back Home**, then click **Validate**, and the Axis2 Happiness Page is displayed.
6. Click **Back Home**, then click **Administration**, then login as **admin** (password **axis2**), and you get to the Axis2 Web Admin Module. From here you can administer the services.



Tools

- Upload Service

System Components

- Available Services
- Available Service Groups
- Available Modules
- Globally Engaged Modules
- Available Phases

Execution Chains

- Global Chains
- Operation Specific Chains

Engage Module

- For all Services
- For a Service Group
- For a Service
- For an Operation

Services

- Deactivate Service
- Activate Service
- Edit Parameters

Contexts

- View Hierarchy

Welcome to Axis2 Web Admin Module !!

You are now logged into the Axis2 administration console from inside the console you will be able to

- to check on the health of your Axis2 deployment.
- to change any parameters at run time.
- to upload new services into Axis2 [Service hot-deployment].

7. Click **Logout** and close the Axis2 console.

Remove the application from the server

If you try to remove the application with the Axis2 Web module from the server, WASCE does not complete the task.

Workaround:

WASCE has a defect that does not allow you to undeploy an Axis2 project created in Eclipse. The only way to undeploy it is to stop the server and delete the application files directly from the WASCE `repository/ejee` folder.

To be able to continue with the examples in the book, remove the Axis2SecureWSWeb from the DonateEAR enterprise application:

1. Right-click **DonateEAR** and select **Properties**. Select **Java EE Module Dependencies** and clear **Axis2SecureWSWeb.war**. Click **OK**.
2. Stop the server.
3. Delete `DonateEAR` from `<WASCE_HOME>/repository/ejee`.
4. Restart the server.
5. Remove the DonateEAR application from the server (**Add and Remove Projects**), then select **Publish**.
6. Add the DonateEAR application to the server, and click **Publish**.

12.9.2 Additional resources

Additional Learn resources are available at the following Web sites:

- Web Services Security UsernameToken Profile 1.0:

  http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf

- Apache Rampart:

  http://ws.apache.org/rampart/

- Securing SOAP Messages with Rampart:

  http://ws.apache.org/axis2/modules/rampart/1_3/security-module.html

- UsernameToken Authentication with Rampart:

  http://wso2.org/library/240

- See "Chapter 14. Implement security for the Web service", in *Experience J2EE! Using WebSphere Application Server V6.1*, SG24-7297-00, located at:

  http://www.redbooks.ibm.com/abstracts/sg247297.html

- See "Chapter 8. Web services security", in *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257, located at:

  http://www.redbooks.ibm.com/abstracts/sg247257.html

12.9.3 Secure Web services with DataPower

What is DataPower®?

From ibm.com:

> IBM WebSphere DataPower SOA Appliances represent an important element in IBM's holistic approach to Service Oriented Architecture (SOA). IBM SOA appliances are purpose-built, easy-to-deploy network devices that simplify, help secure, and accelerate XML and Web services deployments, while extending your SOA infrastructure. These new appliances offer an innovative, pragmatic approach to harness the power of SOA while simultaneously enabling you to leverage the value of your existing application, security, and networking infrastructure investments.

There are actually three different DataPower appliances offered by IBM:

- **Integration Appliance X150** is used for integration into back-end systems, including legacy.
- **XML Accelerator XA35** is used to off-load XML, XSD, XPath, and XSLT processes from over-tasked servers to this highly efficient appliance.

- **XML Security Gateway XS40** was built by some of the world's top Web services experts to deliver secured Web services transactions.

The remainder of this section is dedicated to the XML Security Gateway XS40.

After implementing the Web services part of this book, many readers might have been puzzled by the fact that, out-of-the-box, WASCE does not support WS-Security. The IBM position on this is that our base WebSphere product offers the best software security for Web services and should be the product implementation of choice in these scenarios.

However, the hard reality is that WS-Security does not provide efficient security for Web based applications. Many will argue that the WS-Security really is only smoke screen security, and any semi-sophisticated hacker can breach this level of security. The IBM position on this is that only by the use of appliance technology, such as DataPower, it is possible to truly ensure that Web services are secure.

For more information, refer to:

http://www-306.ibm.com/fcgi-bin/common/ssi/ssialias?infotype=an&subtype=ca&appname=Demonstration&htmlfid=897/ENUS106-253#@2h@61@

This appliance provides a security-enforcement point for XML and Web services transactions, including encryption, firewall filtering, digital signatures, schema validation, WS-Security, XML access control, XPath and detailed logging, and includes:

- **XML/SOAP firewall**: The DataPower XML Security Gateway XS40 filters traffic at wire speed, based on information from layers 2 through 7 of the protocol stack; from field-level message content and SOAP envelopes to IP address, port/hostname, payload size, or other metadata. Filters can be predefined with easy point-and-click XPath filtering GUI, and automatically uploaded to change security policies based on the time of day or other triggers.
- **XML/SOAP data validation**: With its unique ability to perform XML Schema validation as well as message validation, at wire speed, the XS40 ensures incoming and outgoing XML documents are legitimate and properly structured. This protects against threats such as XML Denial of Service (XDoS) attacks, buffer overflows, or vulnerabilities created by deliberately or inadvertently malformed XML documents.

- **Field level message security**: The XS40 selectively shares information through encryption/decryption and signing/verification of entire messages or of individual XML fields. These granular and conditional security policies can be based on nearly any variable, including content, IP address, hostname or other user-defined filters.
- **XML Web services access control**: The XS40 supports a variety of access control mechanisms, including WS-Security, WS-Trust, X.509, SAML, SSL, LDAP, RADIUS and simple client/URL maps. The XS40 can control access rights by rejecting unsigned messages and verifying signatures within SAML assertions.
- **Service virtualization**: XML Web services require companies to link partners to resources without leaking information about their location or configuration. With the combined power of URL rewriting, high-performance XSL transforms, and XML/SOAP routing, the XS40 can transparently map a rich set of services to protected back-end resources with high performance.
- **Centralized policy management**: The XS40's wire speed performance enables enterprises to centralize security functions in a single drop-in device that can enhance security and help reduce ongoing maintenance costs. Simple firewall functionality can be configured via a GUI and running in minutes, and using the power of XSLT, the XS40 can also create sophisticated security and routing rules. Because the XS40 works with leading Policy Managers, it is an ideal policy execution engine for securing next generation applications. Manageable locally or remotely, the XS40 supports SNMP, script-based configuration, and remote logging to integrate seamless with leading management software.
- **Web services management/service level management**: With support for Web services Distributed Management (WSDM), Universal Description, Discovery, and Integration (UDDI), Web Services Description Language (WSDL), Dynamic Discovery, and broad support for Service Level Management configurations, the XS40 natively offers a robust Web services management framework. This framework allows the efficient management of distributed Web service endpoints and proxies in heterogeneous SOA environments, as well as SLM alerts and logging, and pull and enforce policies, which helps enable broad integration support for third-party management systems and unified dashboards, in addition to robust support and enforcement for governance frameworks and policies.

For more information, refer to the DataPower Web site:

http://www-306.ibm.com/software/integration/datapower/

Part 4

Messaging

In Part 4, we extend the core JEE application to support JMS messaging, by performing the tasks described in the following chapters:

- Chapter 13, “Create the message-driven bean” on page 405: Create a message-driven bean to receive messages and to call the Donate session EJB. Create a JMS client to send a message to the message-driven bean.
- Chapter 14, “Add publication of results” on page 433: Update the Donate session EJB to publish the donation request results to a topic representing the employee number, and create a JMS client (to subscribe to updates for an employee).
- Chapter 15, “Implement security for messaging” on page 445: Update the messaging artifacts to support security.

13

Create the message-driven bean

In this section we perform the following actions (Figure 13-1):

- Create a message-driven bean to receive messages and call the Donate session EJB.
- Create a JMS client to send a message to the message-driven bean.

13.1 Learn!

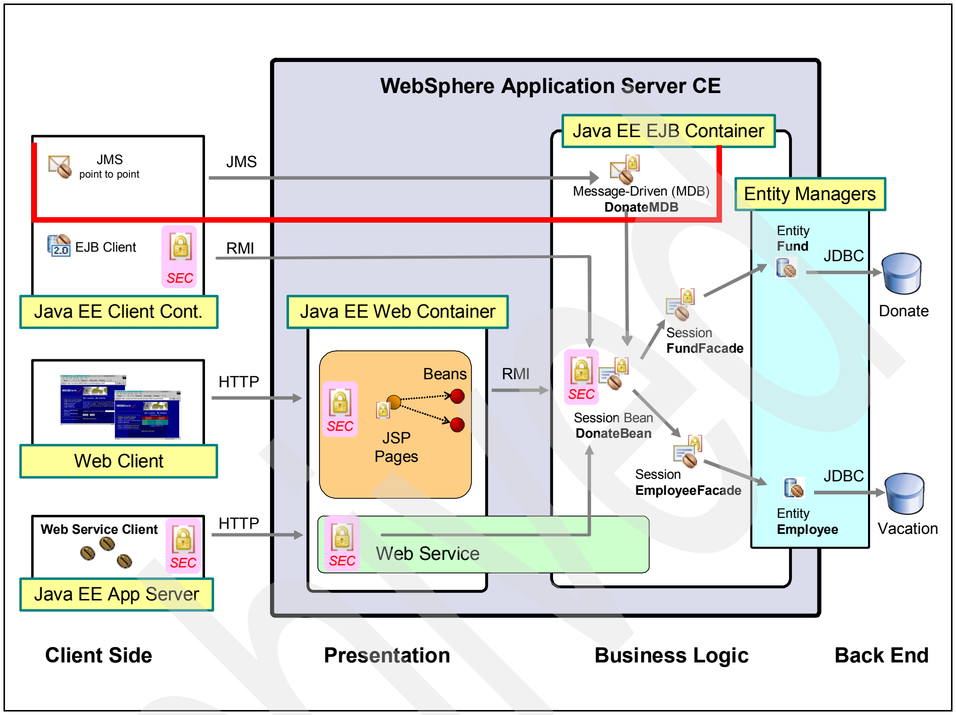


Figure 13-1 Donate Application: Message-driven bean and JMS client

Messaging provides an asynchronous communication mechanism that is an alternative to HTTP for program-to-program communication:

- HTTP is synchronous, similar in concept to an instant message application. The sender and receiver both must be active.

  This makes for a tightly coupled solution. Applications are aware of and must handle communication errors, whether short term (milliseconds) or longer term (seconds and higher). As a result, an outage in any one part—whether due to networking errors, hardware failures, application failures, or maintenance and upgrades—can cause delays across the broader system due to the cascading delays of error handling, waits, and retries.

- Messaging is asynchronous, similar in concept to an e-mail application. The sender creates the message, and gives it to a messaging subsystem, which accepts it for delivery.

The messaging subsystem takes care of sending the message to the target system—handling any networking errors or system shutdowns—and the application on the target system receives the message, either auto-started by the messaging system or when the application makes a call. The application can optionally reply, sending a similar asynchronous message back to the requester.

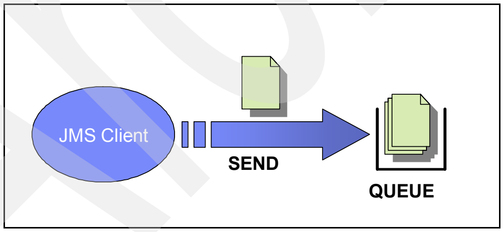
This allows for a loosely coupled solution: Applications only have to be aware that the message was “accepted” for delivery and then can proceed to other processing. An outage in any one part does not immediately block operation in the rest of the system.

Java EE contains the Java Message Service (JMS) specification, which defines an API that applications can utilize to access a messaging provider (a product implementation that provides the infrastructure that supports messaging, such as Active MQ or IBM WebSphere MQ.

JMS is the API specification only and does not describe the actual messaging product or provider. This is similar to accessing relational databases through JDBC, where JDBC defines the access mechanism but does not define the actual relational database product.

All the interaction with the Java EE server so far was over HTTP. How do you access Java EE over messaging?

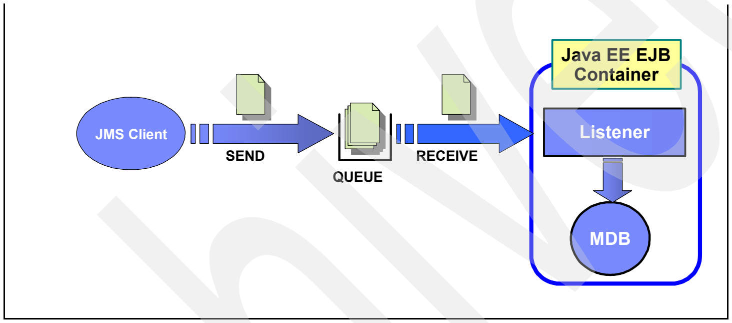
- **Client side**: A program creates a message and sends it to a JMS queue, in this case, a local queue. The message stays on the source system and is not forwarded to a remote system, and then terminates. The program does not know when (or if) the message is actually read, only that the message is persisted in the queue.



In this scenario, the program is a standalone Java program, but you could put the same basic code inside Java EE server side artifacts (such as servlets or EJBs) if you wanted to send a message from one Java EE server to another, or even to another component in the same Java EE server.

- **Java EE server side**: An EJB container is configured to listen to a queue through a definition called an activation specification. An EJB artifact called a message-driven bean (MDB) is created, with a method called `onMessage` that contains the user defined business logic, and is deployed to this EJB container, referencing the same activation specification. When the EJB container starts, it opens the activation specification and issues a receive against the queue. When a message is placed on the queue, the activation specification receives the message and calls the associated MDB `onMessage` method, which runs the user defined business logic.

  Note that MDBs have little in common with entity EJBs and session EJBs except for executing in the same Java EE container. The MDB is invoked by the overall Java EE infrastructure and does not have any external local or remote interfaces.



13.1.1 Additional resources

For complete details on setting up JMS resources through the WASCE Console, refer to the WASCE 2.0 User Guide at:

http://www.redbooks.ibm.com/redbooks/pdfs/sg247585.pdf

The following additional Learn resources are available:

- J2EE: Java Message Service API – FAQ:

  http://java.sun.com/products/jms/faq.html

- IBM developerWorks: J2EE pathfinder: Enterprise messaging with JMS:

  http://www.ibm.com/developerworks/java/library/j-pj2ee5/

- WASCE InfoCenter:

  http://publib.boulder.ibm.com/wasce/V2.1.0/en/java-messaging-services.html

13.1.2 EJB 3.0 message-driven beans

Message-driven beans are used for the processing of asynchronous JMS messages within Java EE based applications. The are invoked by the container on the arrival of a message.

In this way, message-driven beans can be thought of as another interaction mechanism for invoking EJBs, but unlike session beans, the container is responsible for invoking them when a message is received, not a client (or another bean).

To define a message driven bean in EJB 3.0 we declare a POJO with the @MessageDriven annotation:

```
@MessageDriven(activationConfig = {
         @ActivationConfigProperty(propertyName="destinationType",
                                   propertyValue="javax.jms.Queue"),
         @ActivationConfigProperty(propertyName="destination",
                                   propertyValue="queue/myQueue")
})
public class MyMessageBean implements javax.jms.MessageListener {

   public void onMessage(javax.msg.Message inMsg) {
      // implement the onMessage method
      // to handle the incoming message
      ....
   }
}
```

The main relevant features of this example are:

- In EJB 3.0, the MDB bean class is annotated with the **@MessageDriven** annotation, which specifies a set of activation configuration parameters. These parameters are unique to the particular kind of JCA 1.5 adapter that is used to drive the MDB. Some adapters have configuration parameters that let you specify the destination queue of the MDB. In the case where the adapter does not support this, the destination name must be specified using a `<message-destination>` entry in the XML binding file.
- The bean class has to implement the **MessageListener** interface, which defines only one method, **onMessage**. When a message arrives in the queue monitored by this MDB, the container calls the `onMessage` method of the bean class and passes the incoming message in as the parameter.
- Furthermore, the `activationConfig` property of the @MessageDriven annotation provides messaging system-specific configuration information.

13.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

13.3 Disable Java EE EJB declarative security

This chapter and the subsequent chapter (Chapter 14, “Add publication of results” on page 433) do not contain the application code required to support a secure session EJB. Therefore, before starting this chapter, we have to disable the Java EE EJB declarative security. We will re-enable EJB security at the start of Chapter 15, “Implement security for messaging” on page 445.

1. Disable declarative security in the `DonateBean` session EJB:
   - In the Project Explorer, open **DonateBean** (in `DonateEJB/ejbModule /donate.ejb.impl`).
   - Before the two `donateToFund` methods, comment out the `@RolesAllowed` statement and activate the `@PermitAll` statement.

     ```
     //@RolesAllowed( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
     @PermitAll
     public String donateToFund(Employee employee, ......)
        ......

     //@RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
     @PermitAll
     @WebMethod
     public String donateToFund(int employeeId, ......)
     ```
   - Save and close `DonateBean.java`.
2. Refer to 11.3, “Disable Java EE EJB declarative security” on page 329 for an explanation of the @PermitAll annotation.

13.4 Configure the test server for base JMS resources

In this section, we configure the **ExperienceJavaEE Server** test server with the required resources for the MDB: JMS resource group, connection factory, and queue.

13.4.1 Create the JMS resource group

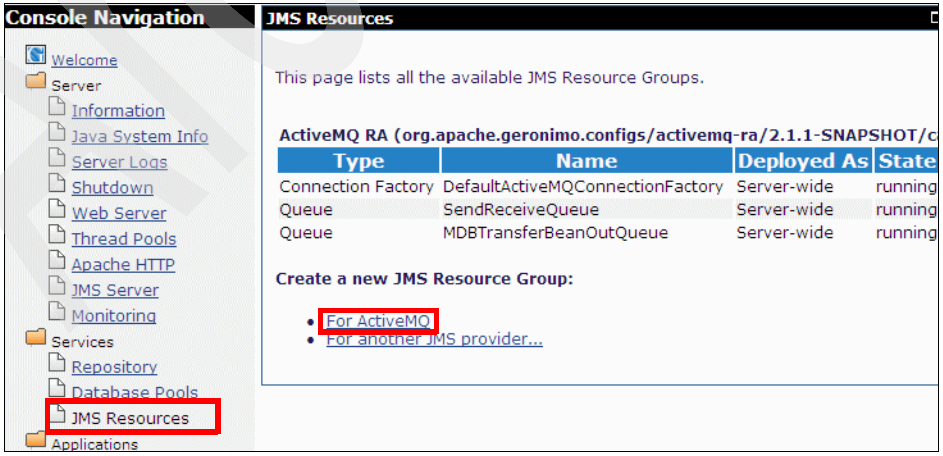
A JMS provider acts as an intermediary between message producers and consumers. You need a JMS provider for message-driven beans (MDBs) and for sending and receiving messages. A JMS provider is configured in WASCE as a resource adapter module. Any JMS provider can be integrated with WASCE by deploying the resource adapter for that JMS provider.

The J2EE Connector Architecture (JCA) is a standard way of connecting Java EE applications and enterprise information systems (EIS), such as WebSphere MQ or DB2. The architecture of WASCE is such that you can configure any JMS provider for use with WASCE, by deploying the resource adapter for that message broker.

By default, WASCE includes Apache ActiveMQ as the JMS provider. When you start a server, the ActiveMQ provider is automatically available. We can also configure a server to use other JMS providers, including WebSphere MQ.

A JMS resource group is a resource adapter module that binds together the related connection factories, queues, and topics. To create and access JMS resources, such as queues, topics, and connection factories in WASCE, we have to create a JMS resource group:

1. Start the WASCE Console:
   - – In the Servers view, right-click the **ExperienceJavaEE Server** and select **Launch WASCE Console**.
   - – Login as **system** (with password `manager`).
2. In the left hand navigation pane, select **Services** → **JMS Resources** to open the JMS Resources editor.



This page lists all the available JMS Resource Groups.

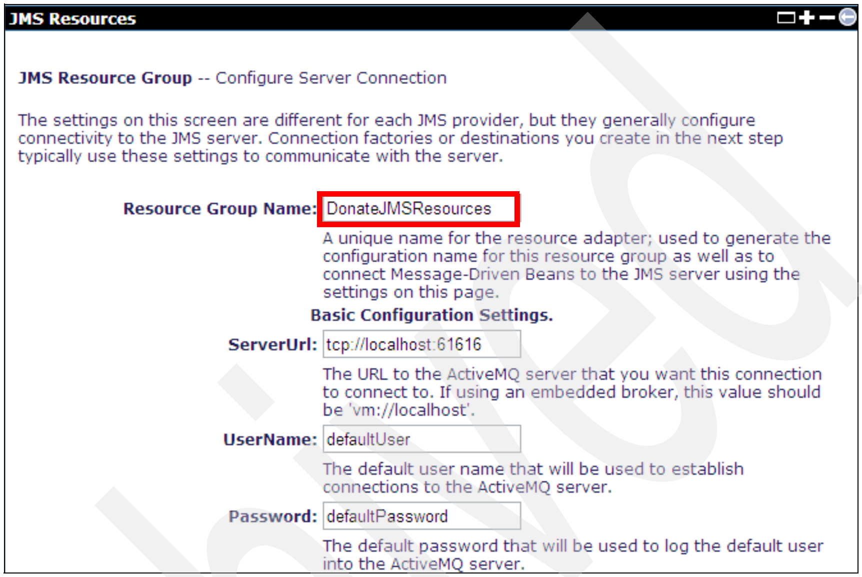
ActiveMQ RA (org.apache.geronimo.configs/activemq-ra/2.1.1-SNAPSHOT/c

Type	Name	Deployed As	State
Connection Factory	DefaultActiveMQConnectionFactory	Server-wide	running
Queue	SendReceiveQueue	Server-wide	running
Queue	MDBTransferBeanOutQueue	Server-wide	running

Create a new JMS Resource Group:

- For ActiveMQ
- For another JMS provider...

3. Under Create a new JMS Resource Group, click **For ActiveMQ**.
4. In the JMS Resources/JMS Resource Group -- Configure Server Connection page, set the Resource Group Name to **DonateJMSResources**. Accept the rest of the defaults and click **Next**.



JMS Resources

JMS Resource Group -- Configure Server Connection

The settings on this screen are different for each JMS provider, but they generally configure connectivity to the JMS server. Connection factories or destinations you create in the next step typically use these settings to communicate with the server.

Resource Group Name: DonateJMSResources
A unique name for the resource adapter; used to generate the configuration name for this resource group as well as to connect Message-Driven Beans to the JMS server using the settings on this page.

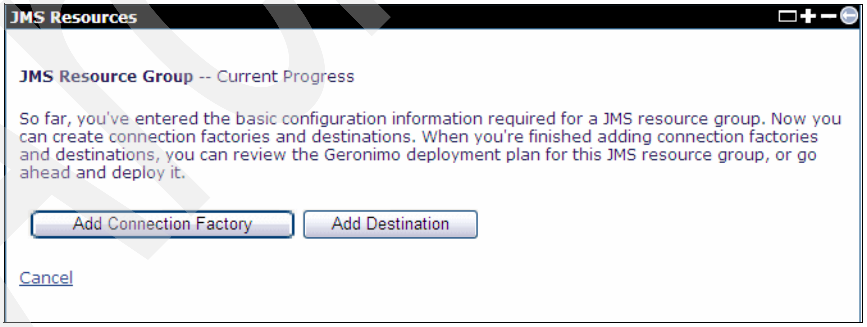
Basic Configuration Settings.

ServerUrl: tcp://localhost:61616
The URL to the ActiveMQ server that you want this connection to connect to. If using an embedded broker, this value should be 'vm://localhost'.

UserName: defaultUser
The default user name that will be used to establish connections to the ActiveMQ server.

Password: defaultPassword
The default password that will be used to log the default user into the ActiveMQ server.

5. The next page is the JMS Resources/JMS Resource Group -- Current Progress page. Proceed to the next section, where we create the JMS connection factory.



JMS Resources

JMS Resource Group -- Current Progress

So far, you've entered the basic configuration information required for a JMS resource group. Now you can create connection factories and destinations. When you're finished adding connection factories and destinations, you can review the Geronimo deployment plan for this JMS resource group, or go ahead and deploy it.

13.4.2 Create the JMS connection factory

A JMS connection factory provides a specific class of service that is used to connect to groups of queues and topics. If all the applications require the same class of service, we could use a single connection factory.

However, if the applications require different classes of service for example, one application requires that the inbound messages be processed as `UNAUTHENICATED`, while another application requires that inbound messages be processed as a specific user ID, then we need multiple connection factories.

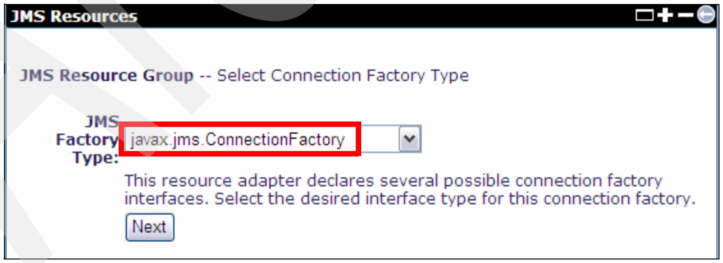
Creating separate connection factories also allows us to distribute queues between multiple messaging providers (some local, some remote), enabling us to selectively shut down messaging for one application while leaving it active for another.

We use a connection factory with the messaging client that you create in 13.6, “Create the messaging client (DonateMDBClient)” on page 426.

Note that a connection factory is not required for the MDB that we create in 13.5, “Create the message-driven bean” on page 419. The MDB accesses the class of service through the activation specification.

Thus, we could defer this configuration step until 13.6, “Create the messaging client (DonateMDBClient)” on page 426, but we do it now to keep all the JMS configuration activities in one location.

1. In the JMS Resources/JMS Resource Group -- Current Progress page, click **Add Connection Factory**.
2. In the JMS Resources/JMS Resource Group -- Select Connection Factory Type page, select **javax.jms.ConnectionFactory** from the JMS Factory Type drop-down list and click **Next**.



JMS Resources

JMS Resource Group -- Select Connection Factory Type

JMS Factory Type: javax.jms.ConnectionFactory

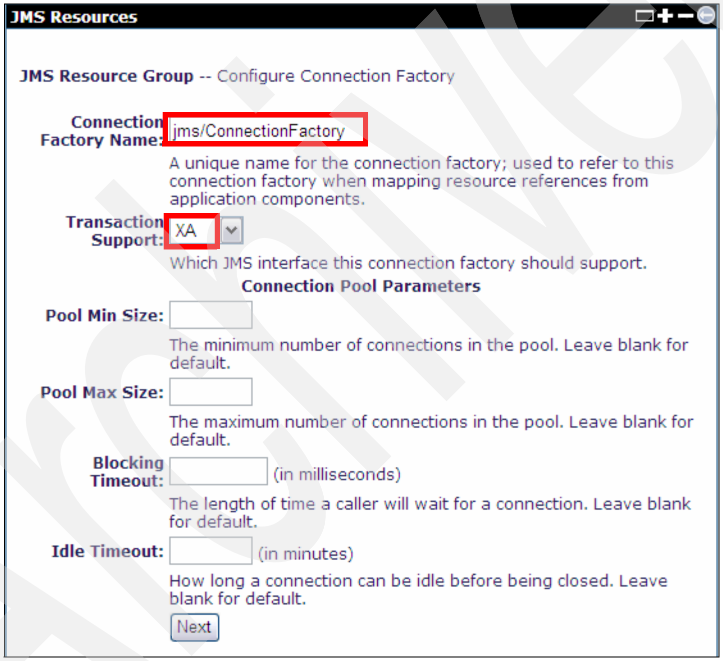
This resource adapter declares several possible connection factory interfaces. Select the desired interface type for this connection factory.

Behind the scenes:

JMS used to have separate `QueueConnectionFactory` and `TopicConnectionFactory` objects. Starting with JMS 1.1, there is a `ConnectionFactory`, which handles both queues and topics.

Best practice is to use `ConnectionFactory`. `QueueConnectionFactory` and `TopicConnectionFactory` are maintained for compatibility with older applications.

3. In the JMS Resources/JMS Resource Group -- Select Connection Factory Type page, set the Connection Factory Name to **jms/ConnectionFactory**, ensure that Transaction Support is set to **XA**, and click **Next**.



JMS Resources

JMS Resource Group -- Configure Connection Factory

Connection Factory Name: jms/ConnectionFactory
A unique name for the connection factory; used to refer to this connection factory when mapping resource references from application components.

Transaction Support: XA
Which JMS interface this connection factory should support.

Connection Pool Parameters

Pool Min Size:
The minimum number of connections in the pool. Leave blank for default.

Pool Max Size:
The maximum number of connections in the pool. Leave blank for default.

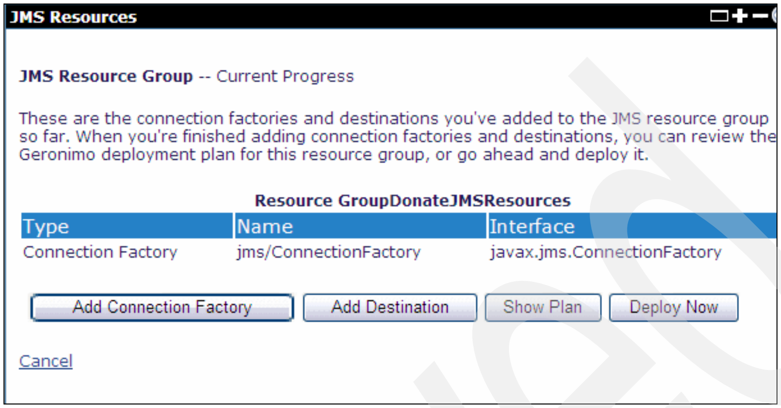
Blocking Timeout: (in milliseconds)
The length of time a caller will wait for a connection. Leave blank for default.

Idle Timeout: (in minutes)
How long a connection can be idle before being closed. Leave blank for default.

Behind the scenes:

If a single application accesses a database, and writes to a message queue, two providers are involved. Therefore, we need an XA connection.

4. The dialog returns to the JMS Resources/JMS Resource Group -- Current Progress page and should show the new connection factory. Proceed to the next section, where we create the JMS Destination.



JMS Resource Group -- Current Progress

These are the connection factories and destinations you've added to the JMS resource group so far. When you're finished adding connection factories and destinations, you can review the Geronimo deployment plan for this resource group, or go ahead and deploy it.

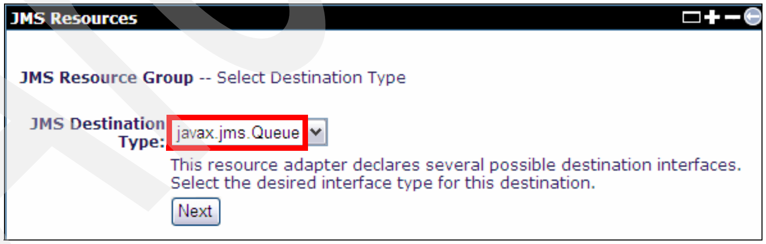
Resource GroupDonateJMSResources

Type	Name	Interface
Connection Factory	jms/ConnectionFactory	javax.jms.ConnectionFactory

13.4.3 Create the JMS queue destination

A destination queue is the physical resource that holds messages. Destinations are accessed in a persistent mode (meaning that messages are preserved across reboots) or a non-persistent mode (meaning that messages are not preserved across reboots):

1. In the JMS Resources/JMS Resource Group -- Current Progress page, click **Add Destination**.
2. In the JMS Resources/JMS Resource Group -- Select Destination Type page, select **javax.jms.Queue** as JMS Destination Type and click **Next**.

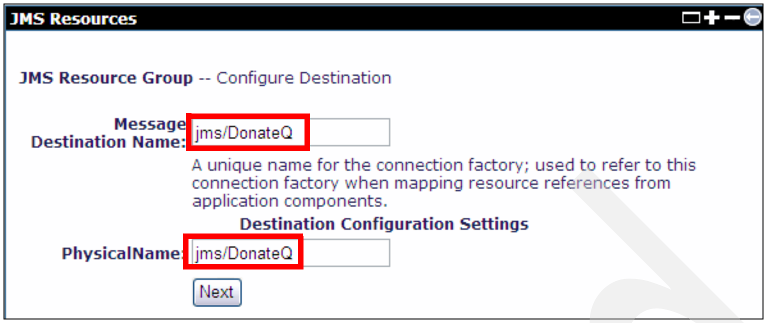


JMS Resource Group -- Select Destination Type

JMS Destination Type: javax.jms.Queue

This resource adapter declares several possible destination interfaces. Select the desired interface type for this destination.

3. In the JMS Resources/JMS Resource Group -- Configure Destination page, set the Message Destination Name to **jms/DonateQ**, set the PhysicalName to **jms/DonateQ**, and click **Next**.



JMS Resource Group -- Configure Destination

Message Destination Name: jms/DonateQ

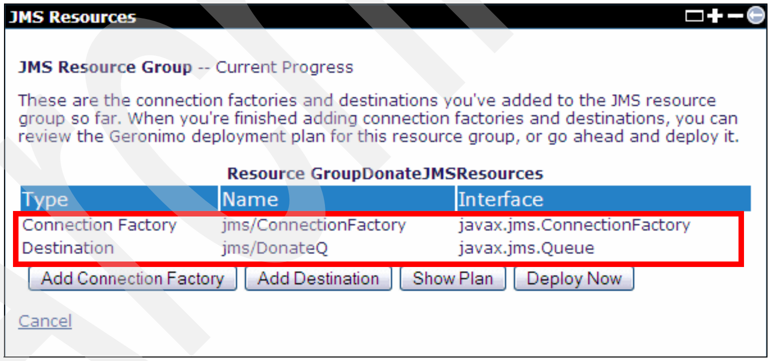
A unique name for the connection factory; used to refer to this connection factory when mapping resource references from application components.

Destination Configuration Settings

PhysicalName: jms/DonateQ

> **Behind the scenes:**
>
> The Message Destination Name is the JNDI name for lookup of the definition in WASCE V2.1
>
> The PhysicalName is the ActiveMQ queue, and in WASCE v2.1 this is a dynamic queue. Therefore, the name that someone would use to access this queue through direct ActiveMQ lookup is `dynamicQueues/jms/DonateQ`.

4. The dialog returns to the JMS Resources/JMS Resource Group -- Current Progress page and displays the new destination.



JMS Resource Group -- Current Progress

These are the connection factories and destinations you've added to the JMS resource group so far. When you're finished adding connection factories and destinations, you can review the Geronimo deployment plan for this resource group, or go ahead and deploy it.

Resource GroupDonateJMSResources

Type	Name	Interface
Connection Factory	jms/ConnectionFactory	javax.jms.ConnectionFactory
Destination	jms/DonateQ	javax.jms.Queue

5. In the JMS Resources/JMS Resource Group -- Current Progress page, click **Show Plan**.
6. In the JMS Resources/JMS Resource Group -- Show Deployment Plan page, review the plan in the Deployment Plan box and compare to the contents of Example 13-1 on page 418).

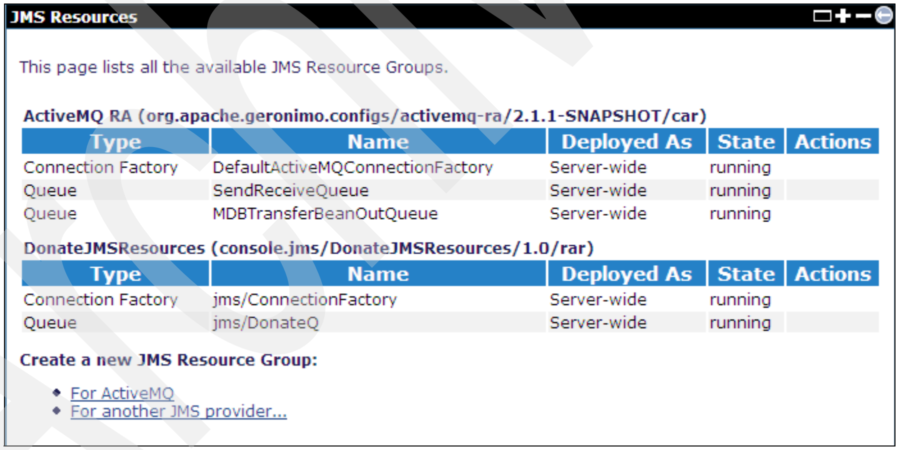
Behind the scenes:

The plan is identical to `C:\7639code\plans\JMSResources-plan.xml`. which we saved during the writing of this book. This plan can be used for automated or scripted configuration as described in A.2, “Chapter specific jump start steps” on page 559, and Appendix A.3, “Creating resources in a new server” on page 569.

To configure the necessary JMS resources in WASCE, run the following **`deploy`** command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory:

```
deploy -u system -p manager deploy
C:\7639code\plans\JmsResources-plan.xml
..\repository\org\apache\geronimo\modules\geronimo-activemq-ra\2.1.1\g
eronimo-activemq-ra-2.1.1.rar
```

7. Click **Deploy JMS Resource**.
8. In the resulting top-level JMS Resources page, verify that the `DonateJMSResources` group exists with the connection factory and destination.



JMS Resources

This page lists all the available JMS Resource Groups.

ActiveMQ RA (org.apache.geronimo.configs/activemq-ra/2.1.1-SNAPSHOT/car)

Type	Name	Deployed As	State	Actions
Connection Factory	DefaultActiveMQConnectionFactory	Server-wide	running	
Queue	SendReceiveQueue	Server-wide	running	
Queue	MDBTransferBeanOutQueue	Server-wide	running	

DonateJMSResources (console.jms/DonateJMSResources/1.0/rar)

Type	Name	Deployed As	State	Actions
Connection Factory	jms/ConnectionFactory	Server-wide	running	
Queue	jms/DonateQ	Server-wide	running	

Create a new JMS Resource Group:

- For ActiveMQ
- For another JMS provider...

Behind the scenes:

The JMS deployment plan contains a deployable version of the information entered in the dialog:

This section of the deployment plan defines all of the resource environment dependencies. You can see that this module is of group `console.jms`, has an artifact Id (or Name) of `DonateJMSResources`, is the first version, and of type resource archive repository (`rar`). As expected, it has a dependency of the `activemq-broker` to handle messaging.

Example 13-1 Deployment plan for JMS resources

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://geronimo.apache.org/xml/ns/j2ee/connector-1.2">
   <dep:environment
         xmlns:dep="http://geronimo.apache.org/xml/ns/deployment-1.2">
      <dep:moduleId>
         <dep:groupId>console.jms</dep:groupId>
         <dep:artifactId>DonateJMSResources</dep:artifactId>
         <dep:version>1.0</dep:version>
         <dep:type>rar</dep:type>
      </dep:moduleId>
       <dep:dependencies>
         <dep:dependency>
            <dep:groupId>org.apache.geronimo.configs</dep:groupId>
            <dep:artifactId>activemq-broker</dep:artifactId>
             <dep:type>car</dep:type>
         </dep:dependency>
      </dep:dependencies>
   </dep:environment>
   <resourceadapter>
      <resourceadapter-instance>
         <resourceadapter-name>DonateJMSResources</resourceadapter-name>
         <nam:workmanager
               xmlns:nam="http://geronimo.apache.org/xml/ns/naming-1.2">
            <nam:gbean-link>DefaultWorkManager</nam:gbean-link>
         </nam:workmanager>
      </resourceadapter-instance>
      <outbound-resourceadapter>
         <connection-definition>
            <connectionfactory-interface>javax.jms.ConnectionFactory
            </connectionfactory-interface>
            <connectiondefinition-instance>
               <name>jms/ConnectionFactory</name>
               <implemented-interface>javax.jms.QueueConnectionFactory
               </implemented-interface>
               <implemented-interface>javax.jms.TopicConnectionFactory
               </implemented-interface>
               <connectionmanager>
                  <xa-transaction>
                     <transaction-caching/>
                  </xa-transaction>
                  <single-pool>
                     <match-one/>
                  </single-pool>
               </connectionmanager>
            </connectiondefinition-instance>
         </connection-definition>
      </outbound-resourceadapter>
   </resourceadapter>
   <adminobject>
      <adminobject-interface>javax.jms.Queue</adminobject-interface>
      <adminobject-class>org.apache.activemq.command.ActiveMQQueue
      </adminobject-class>
      <adminobject-instance>
         <message-destination-name>jms/DonateQ</message-destination-name>
         <config-property-setting name="PhysicalName">jms/DonateQ
         </config-property-setting>
      </adminobject-instance>
   </adminobject>
   <adminobject>
      <adminobject-interface>javax.jms.Topic</adminobject-interface>
      <adminobject-class>org.apache.activemq.command.ActiveMQTopic
      </adminobject-class>
   </adminobject>
</connector>
```

Off course?

If you make an error and have to redo the JMS definitions, you first have to remove the old plan. In the WASCE Console:

- Select **Applications** → **J2EE Connectors**.
- Locate the `console.jms/DonateJMSResources` entry, click **Stop**, then click **Uninstall**.

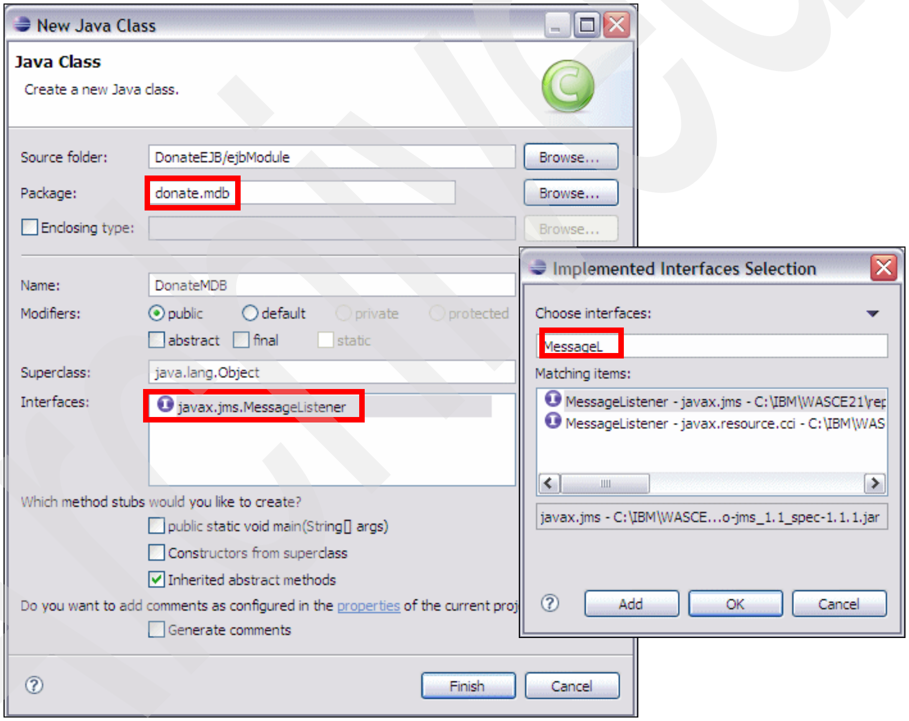
13.5 Create the message-driven bean

In this section, we create a message-driven bean (MDB), `DonateMDB`, to handle JMS messages and pass them to the session EJB.

13.5.1 Code the DonateMDB message-driven bean

With EJB 3.0, an MDB is an annotated Java class that implements the `MessageListener` interface:

1. In the Project Explorer, right-click **DonateEJB**, and select **New** → **Class**.
2. In the New Java Class/Java Class pop-up:
   - Set package to **donate.mdb**.
   - Set Name to **DonateMDB**.
   - Next to Interfaces, click **Add**, and select the `javax.jms.MessageListener` interface (start typing **MessageL**, then select the interface). Click **OK**.
   - Click **Finish**.



New Java Class

Java Class

Create a new Java class.

Source folder: DonateEJB/ejbModule

Package: donate.mdb

☐ Enclosing type:

Name: DonateMDB

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces: javax.jms.MessageListener

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments as configured in the properties of the current proj

☐ Generate comments

Implemented Interfaces Selection

Choose interfaces:

MessageL

Matching items:

MessageListener - javax.jms - C:\IBM\WASCE21\rep

MessageListener - javax.resource.cci - C:\IBM\WAS

javax.jms - C:\IBM\WASCE...o-jms_1.1_spec-1.1.1.jar

Behind the scenes:

`MessageListener` is a required interface for an MDB. It specifies that an `onMessage` method must be provided to handle incoming JMS messages.

3. In the `DonateMDB` editor, add a **@MessageDriven** annotation before the class declaration to define this Java class as an EJB 3.0 message-drive bean, and link the bean to the JMS queue created in the server:

   – Insert the **F90 MDB MessageDriven Annotation** snippet before the class definition.

     ```
     @MessageDriven(activationConfig = {
           @ActivationConfigProperty(propertyName = "destinationType",
                                     propertyValue = "javax.jms.Queue"),
           @ActivationConfigProperty(propertyName = "destination",
                                     propertyValue = "jms/DonateQ")
     })
     public class DonateMDB implements MessageListener {
        ......
     ```

4. We want to invoke the `DonateBean` session EJB to process the message. Therefore we define the `DonateBean` using an @EJB annotation:

   – In the `DonateMDB` editor, add the following line before the `onMessage` method to initialize the Donate EJB session bean interface:

     ```
     @MessageDriven(......)
     public class DonateMDB implements MessageListener {

        @EJB DonateBeanRemote donateBean;

        public void onMessage(Message arg0) {
           ......
     ```

5. In the `DonateMDB` editor, insert the `handleMessage` method before the closing brace using the **F91 MDB Handle Message Method** snippet from the Snippets view (Example 13-2).

Example 13-2 DonateMDB handleMessage method

```
private void handleMessage(Message msg) {
   try {
      String text = ((TextMessage) msg).getText();
      System.out.println("DonateMDB: Input Message = " + text);
      String[] tokens = text.split(" ");
      int employeeId = Integer.parseInt(tokens[0]);
      String fundName = tokens[1];
      int donationAmount = Integer.parseInt(tokens[2]);
      String result = donateBean.donateToFund(employeeId, fundName,
                                              donationAmount);
      System.out.println("DonateMDB: The Result Message is: " + result);
   } catch (Exception e) {
      System.out.println("DonateMDB: error in handleMessage: "
                         + e.getLocalizedMessage());
   }
}
```

Behind the scenes:

The processing of the DonateMDB handeMessage method is as follows:

- Extract the text from the input message:

```
String text = ((TextMessage)msg).getText();
```

- Split the text into the input parameters (employee number, fund name, donation amount):

```
String[] tokens = text.split(" ");
int employeeId = Integer.parseInt(tokens[0]);
String fundName = tokens[1];
int donationAmount = Integer.parseInt(tokens[2]);
```

- Invoke the donateToFund method in the DonateEJB session bean, and print the result:

```
String result = donateBean.donateToFund(employeeId, fundName,
                                         donationAmount);
System.out.println("The Result Message is: " + result);
```

6. In the DonateMDB editor, update the onMessage method to call the business logic method handleMessage, by including the statement:

```
public void onMessage(Message arg0) {
   handleMessage(arg0);
}
```

7. Organize imports (**Ctrl+Shift+O**), save, and close DonateMDB.

Alternative:

Instead of using the @MessageDriven annotation with the `activationConfig` information, we could specify the link to the JMS resources in the **ejb-jar.xml** deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar .......................... version="3.0">
<enterprise-beans>
   <message-driven>
      <ejb-name>DonateMDB</ejb-name>
      <activation-config>
         <activation-config-property>
            <activation-config-property-name>destination
            </activation-config-property-name>
            <activation-config-property-value>jms/DonateQ
            </activation-config-property-value>
         </activation-config-property>
         <activation-config-property>
            <activation-config-property-name>destinationType
            </activation-config-property-name>
            <activation-config-property-value>javax.jms.Queue
            </activation-config-property-value>
         </activation-config-property>
      </activation-config>
   </message-driven>
</enterprise-beans>
</ejb-jar>
```

The EJB 3.0 specification allows for either annotations or deployment descriptor information, or a mix. We could use a short annotation (@MessageDriven) and specify the JMS resource linkage in the deployment description.

13.5.2 Update the WASCE deployment descriptors

We have to inform WASCE about the linkage of the MDB to the `DonateJMSResources` plan with the JMS resources. This requires updates in the deployment descriptors for the EJB and the EAR projects:

1. Add the JMS dependency to the DonateEJB project:
   - In the Project Explorer, open **openejb-jar.xml** (in `DonateEJB/ejbModule/META-INF`), and switch to the **Source** tab.
   - Insert the **F92 MDB Open EJB Deployment Descriptor** snippet from the Snippets view before the ending tags (Example 13-3).
   - Save and close `openejb-jar.xml`.

Example 13-3 DonateEJB deployment descriptor for MDB

```
   <ejb:enterprise-beans>
      <ejb:session>
         ......
      </ejb:session>
      <ejb:message-driven>
         <ejb:ejb-name>DonateMDB</ejb:ejb-name>
         <ejb:resource-adapter>
            <ejb:resource-link>DonateJMSResources</ejb:resource-link>
         </ejb:resource-adapter>
      </ejb:message-driven>
   </ejb:enterprise-beans>
</ejb:openejb-jar>
```

Behind the scenes:

For an MDB we have to add a dependency in the `openejb-jar.xml` file to the JMS resource group of the JMS queue.

2. Add the JMS dependency to the DonateEAR project:

   – In the Project Explorer, open **geronimo-application.xml** (in `DonateEAR/EarContent/META-INF`).

   – In the `geronimo-application-jar.xml` editor, switch to the **Source** tab.

   – Insert the **F93 MDB EAR Deployment Dependency** snippet from the Snippets view, after the dependencies to the database pools (Example 13-4).

   – Save and close `geronimo-application.xml`

Example 13-4 DonateEAR deployment descriptor for MDB

```
<app:application ......... application-name="DonateEAR">
   <dep:environment>
      <dep:moduleId>
         .....
      </dep:moduleId>
      <dep:dependencies>
         <dep:dependency>
            <dep:groupId>console.dbpool</dep:groupId>
            <dep:artifactId>DonatePool</dep:artifactId>
         </dep:dependency>
         <dep:dependency>
            <dep:groupId>console.dbpool</dep:groupId>
            <dep:artifactId>VacationPool</dep:artifactId>
         </dep:dependency>
         <dep:dependency>
            <dep:groupId>console.jms</dep:groupId>
            <dep:artifactId>DonateJMSResources</dep:artifactId>
            <dep:version>1.0</dep:version>
            <dep:type>rar</dep:type>
         </dep:dependency>
      </dep:dependencies>
   </dep:environment>
   <security doas-current-caller="true" >
      ......
```

Behind the scenes:

To use a JMS resource group, you must add a dependency to the group in the deployment plan of the EJB of Web module. For example, if a Web application accesses a queue to send messages, you have to add a dependency in `geronimo-web.xml` to the JMS resource group of the JMS queue. In addition, the deployment plan of the enterprise application must have a reference to the deployed JMS resource plan.

13.5.3 Publish and verify the updated DonateEAR application

Next we publish the updated application to catch possible errors. We could also wait until the MDB client is completed:

1. Make sure the ExperienceJavaEE Server is running and the DonateEAR application is added to the server.
2. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.
3. In the Project Explorer, right-click **DonateBeanTester** (in `DonateEJB_Tester/src/donate.test`**)** and select **Run As** → **JUnit Test.**
4. Switch to the JUnit view and verify that the call to `donateToFund` succeeded.

Off course?

A common problem is that this fails because the tags in the EAR deployment descriptor were added incorrectly, for example, below the `</dep:environment>` tag. Publication succeeds with no failure message, but in reality it was never published. You might find that the project will not publish even after resolving the error. In that case, refer to the recovery steps described in 6.6, “Handle publishing failures” on page 202.

13.6 Create the messaging client (DonateMDBClient)

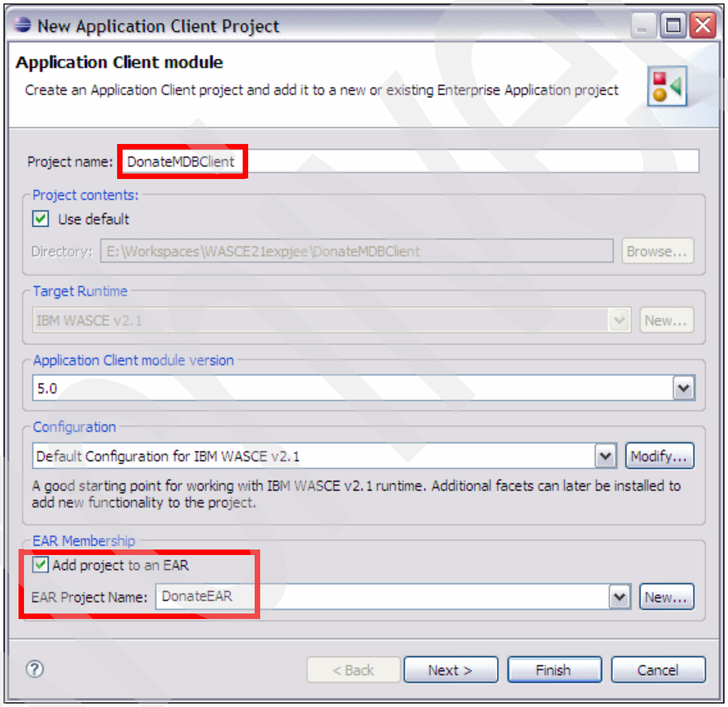
In this section, we create a Java EE application client that sends a JMS message to the Donate queue, thus allowing us to test the MDB:

1. Create the application client project:

   – In the Project Explorer, right-click **DonateEAR** and select **New** → **Application Client Project**.

   – In the Application Client module pop-up, set the name to **DonateMDBClient**, and add the project to the **DonateEAR**.

   – Click **Next**.



New Application Client Project

Application Client module

Create an Application Client project and add it to a new or existing Enterprise Application project

Project name: DonateMDBClient

Project contents:

☑ Use default

Directory: E:\Workspaces\WASCE21expjee\DonateMDBClient

Target Runtime: IBM WASCE v2.1

Application Client module version: 5.0

Configuration: Default Configuration for IBM WASCE v2.1

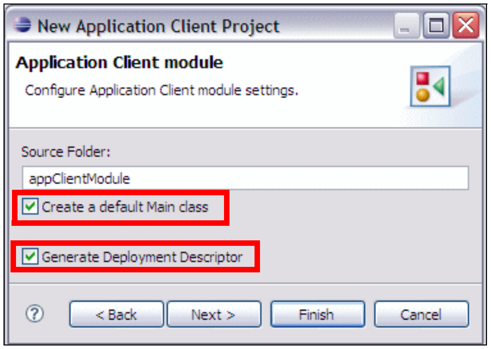
A good starting point for working with IBM WASCE v2.1 runtime. Additional facets can later be installed to add new functionality to the project.

EAR Membership

☑ Add project to an EAR

EAR Project Name: DonateEAR

   – In the New Application Client Project/Application Client module pop-up, select **Create a default Main class** and **Generate Deployment Descriptor**. Click **Next**.



New Application Client Project

Application Client module

Configure Application Client module settings.

Source Folder: appClientModule

☑ Create a default Main class

☑ Generate Deployment Descriptor

   – In the New Application Client Project/Geronimo Deployment Plan pop-up, set the Group Id to **ejee**, leave the Artifact Id empty, and click **Finish**.

2. **Workaround**: Open **geronimo-application-client.xml** (in `DonateMDBClient/appClientModule/META-INF`) to remove invalid lines (refer to 9.3.1, “Workaround: Update Geronimo deployment descriptor” on page 277 for details):

```
   ......
   </dep:server-environment>
    ~~<name:service-ref>~~
        ~~<name:service-ref-name>ServiceRefName</name:service-ref-name>~~
    ~~</name:service-ref>~~
</client:application-client>
```

   Save and close `geronimo-application-client.xml`.

3. In the Project Explorer, open **Main** (in `DonateMDBClient/appClientModule /(default package)`). Note that the class currently contains the following `main` method:

```
public static void main(String[] args) {
      // TODO Auto-generated method stub
}
```

4. In the `Main` editor, replace the // TODO line wit the **F94 MDB Client Main** snippet from the Snippets view (Example 13-5).

Example 13-5   DonateMDBClient main method

```
System.out.println("DonateMDBClient: Establishing initial context");
System.out.println(" Establishing initial context");
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
             "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
env.put(Context.PROVIDER_URL, "tcp://localhost:61616");
Context ctx = new InitialContext(env);
```

```
System.out.println("DonateMDBClient: Looking up ConnectionFactory");
ConnectionFactory cf = (ConnectionFactory) ctx.lookup("ConnectionFactory");

System.out.println("DonateMDBClient: Lookup up Queue");
Destination outDest = (Destination)
ctx.lookup("dynamicQueues/jms/DonateQ");
System.out.println("destination  = " + outDest.toString());

System.out.println("DonateMDBClient: Establishing connection");
Connection connection = cf.createConnection();

System.out.println("DonateMDBClient: Establishing session");
Session session = connection.createSession(false,
                                           Session.AUTO_ACKNOWLEDGE);
MessageProducer destSender = session.createProducer(outDest);

System.out.println("DonateMDBClient: Sending message");
// Send argument default "1 DonationFund 1"
TextMessage toutMessage = session.createTextMessage(args[0]);
destSender.send(toutMessage);

System.out.println("DonateMDBClient: Message sent!");
destSender.close();
session.close();
connection.close();
```

5. Organize imports (**Ctrl+Shift+O**) and select these imports when prompted:

   ```
   javax.jms.Destination
   javax.jms.Connection
   java.util.Hashtable
   javax.jms.ConnectionFactory
   javax.jms.Session
   javax.naming.Context
   ```

   This resolves most errors.

6. For the remaining errors, left-click the quick fix icon [💡] in the left-hand margin, and accept the recommendation to **Add throws declaration**. You do this twice before all errors are resolved. These actions add the throws clause to the method:

   ```
   public static void main(String[] args)
                                  throws NamingException, JMSException
   ```

7. To remove the remaining warnings, add this line before the `main` method:

   ```
   @SuppressWarnings("unchecked")
   public static void main(String[] args) throws ...... {
      ......
   }
   ```

8. Save and close Main.java.

Behind the scenes:

The main method performs the following logic:

- Initialize the JNDI context (the port 61616 was specified when we defined the DonateJMSResources):

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
env.put(Context.PROVIDER_URL, "tcp://localhost:61616");
Context ctx = new InitialContext(env);
```

- Locate the connection factory in the JNDI namespace:

```
ConnectionFactory cf = (ConnectionFactory) ctx.lookup
                                               ("ConnectionFactory");
```

- Locate the queue destination in the JNDI namespace:

```
Destination outDest = (Destination) ctx.lookup
                                    ("dynamicQueues/jms/DonateQ");
```

- Create and start the connection:

```
Connection connection = cf.createConnection();
```

- Create the session and the producer (sender) of messages:

```
Session session = connection.createSession(false,
                                            Session.AUTO_ACKNOWLEDGE);
MessageProducer destSender = session.createProducer(outDest);
```

- Create the message. The arguments are employee ID, fund name, amount, for example: `session.createTextMessage("1 DonationFund 1");`

```
TextMessage toutMessage = session.createTextMessage(args[0]);
```

- Send the message:

```
destSender.send(toutMessage);
```

- Free up resources:

```
destSender.close();
session.close();
connection.close();
```

Alternative:

In this example we use dynamic queues, which is not best practice for production applications, because messages are not persisted. That is, if the application was down, and a message was sent, the message is not read when the application is reactivated. This is a limitation with WASCE 2.1.

Another limitation is the ability to use annotations in a thick client. To explore annotations, run this application in a servlet using resource injection.

13.7 Test the messaging client

Testing the DonateMDBClient is similar to testing the DonateClient in 9.6, “Test the application client” on page 280:

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.
2. Run the **`client`** command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory. Ensure that you enclose the arguments in double quotes:

   ```
   client ejee/DonateMDBClient/1.0/car "1 DonationFund 1"
   ```

3. The DonateMDBClient output in the command prompt window is shown here:

   ```
   DonateMDBClient: Establishing initial context
   DonateMDBClient: Looking up ConnectionFactory
   DonateMDBClient: Lookup up Queue
   DonateMDBClient: Establishing connection
   DonateMDBClient: Establishing session
   DonateMDBClient: Sending message
   DonateMDBClient: Message sent!
   ```

4. Back in the WASCE IDE, switch to the Console view, and verify that the message was processed successfully (there might be other lines in between):

   ```
   DonateMDB: Input Message = 1 DonationFund 1
   ......
   DonateMDB: The Result Message is:  Transfer successful
   ```

Off course?

if you get the console message about unauthorized access (`DonateMDB: error in handleMessage: Unauthorized Access by Principal Denied`), then security was not disabled for the EJB (refer to 13.3, “Disable Java EE EJB declarative security” on page 410).

13.8 Test the messaging client in Eclipse

The following steps are similar to the those used in 9.6, “Test the application client” on page 280 and 10.7.3, “Test the updated application client” on page 316:

1. In the Project Explorer, right-click **Main** (in `DonateMDBClient/appClientModule/(default package)`) and select **Run As** → **Java Application**. The run fails because we are not providing an argument, but it creates a run configuration.
2. Select **Run** → **Run Configurations**.
3. In the Create, manage, and run configurations dialog, select **Java Application** → **Main**:
   - Select the **Arguments** tab, and type the following parameters (including the quotes):

     ```
     "1 DonationFund 1"
     ```

   - Select the **Classpath** tab, and select **User Entries**.
   - Click **Add External JARs**, navigate to **activemq-core-4.1.2.jar** (in `<WASCE-HOME>/repository/org/apache/activemq/activemq-core/4.1.2`), and click **Open**.
   - Click **Add External JARs**, navigate to **backport-util-concurrent-2.2.jar** (in `<WASCE-HOME>/repository/backport-util-concurrent/backport-util-concurrent/2.2`), and click **Open**.
   - Click **Apply** to save the changes, and click **Run** to execute the client.
4. In the Console view of the `Main` class (use the Console selection icon 🖥▾), verify that the messaging client completed successfully:

   ```
   DonateMDBClient: Establishing initial context
   ......
   DonateMDBClient: Message sent!
   ```

5. Switch the Console view to the ExperienceJavaEE Server, and verify that the message was processed successfully (there might be other messages between these):

   ```
   DonateMDB: Input Message = 1 DonationFund 1
   ......
   DonateMDB: The Result Message is:  Transfer successful
   ```

13.9 Explore!

The MDB example in this chapter implements messaging using the embedded JMS provider (ActiveMQ) shipped with WASCE. This implementation provides a robust implementation for WASCE to WASCE interaction, and can be used from .NET, C/C++, and Delphi[4], or from scripting languages such as Perl, Python, PHP, and Ruby.

13.9.1 WebSphere MQ

For environments where you have to connect to various back-end systems, IBM provides the WebSphere MQ product, which provides both a JMS and non-JMS messaging infrastructure for a wide variety of clients and platforms.

You can configure WASCE to use the embedded JMS provider or WebSphere MQ for MDBs. Thus, you can implement this chapter using WebSphere MQ with a few changes, to configure WebSphere MQ resources (instead of the default messaging provider resources) and to implement the EJB using the listener port artifact instead of the activation specification (the activation specification requires JCA support, and WebSphere MQ currently does not have that support).

Additional Explore resources are available at the following Web addresses:

- WebSphere MQ product page:

  http://www.ibm.com/software/integration/wmq

- Integrating WebSphere MQ with WebSphere Application Server Community Edition V2.0:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0804_pathadan/0804_pathadan.html?S_TACT=105AGX10&S_CMP=WASCE

13.9.2 Integration with WebSphere MQ

Since IBM released WASCE v1.0, several customers have come up to us about integration with WebSphere MQ. They find that their applications stretch the limits and capabilities of what currently ActiveMQ can provide them. For this reason IBM has been very proactive in producing developerWorks articles that outline in detail how to integrate WASCE with WebSphere MQ. This enables our customers to use a lightweight Java EE server for their applications, but still get the full robustness of commercial software in their messaging layer.

For detailed information on integrating WebSphere MQ with WASCE, refer to the developerWorks article *Integrating WebSphere MQ with WebSphere Application Server Community Edition V2.0* at:

http://www.ibm.com/developerworks/websphere/library/techarticles/0804_pathadan/0804_pathadan.html

14

Add publication of results

In this chapter, we upgrade the message-driven bean application by publishing the results to a JMS topic from the session bean. We also create a Java EE application client to subscribe to messages sent to this topic (Figure 14-1).

14.1 Learn!

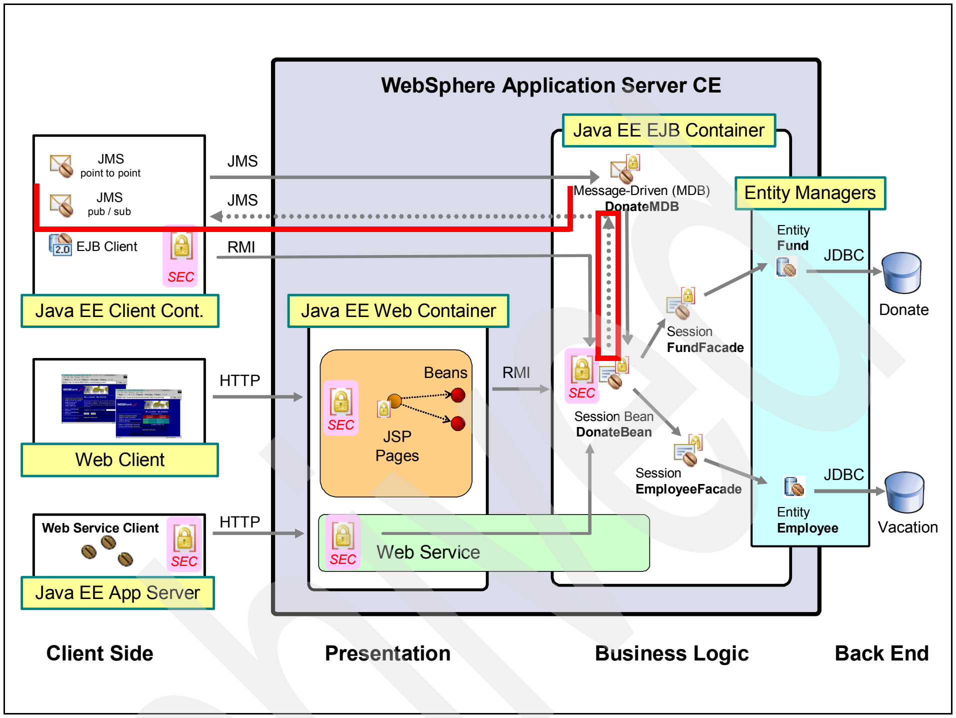


Figure 14-1 Donate application: Publication of results

The point-to-point messaging used in the previous chapter assumes that the message is sent to a single known destination. What if you do not know the intended destination? What if multiple programs might require this message?

This type of messaging need can be addressed by using a topic that represents a special type of messaging destination. It is similar to a subscription service to an electronic bulletin board, and has two important characteristics:

- A name, which can be a single-level name (such as employee numbers 1 or 2 used in this scenario) or hierarchical (`/stocks/IBM` or `/computers/IBM`). Topics can be accessed using wildcards (`/stocks/*` or even `*/IBM`).
- A subscription list, representing destinations that want to receive the messages sent to this topic.

A topic can be a pre-existing resource accessed through the JNDI name or it can be dynamically defined:

- **Publisher**: A program creates a message and sends it to a JMS topic.
- **Subscriber**: A program connects to a specific JMS topic, and then issues a receive. When the publish and subscribe engine receives a message for that topic, it places a copy of the message on the queue for each open receive (Figure 14-2).

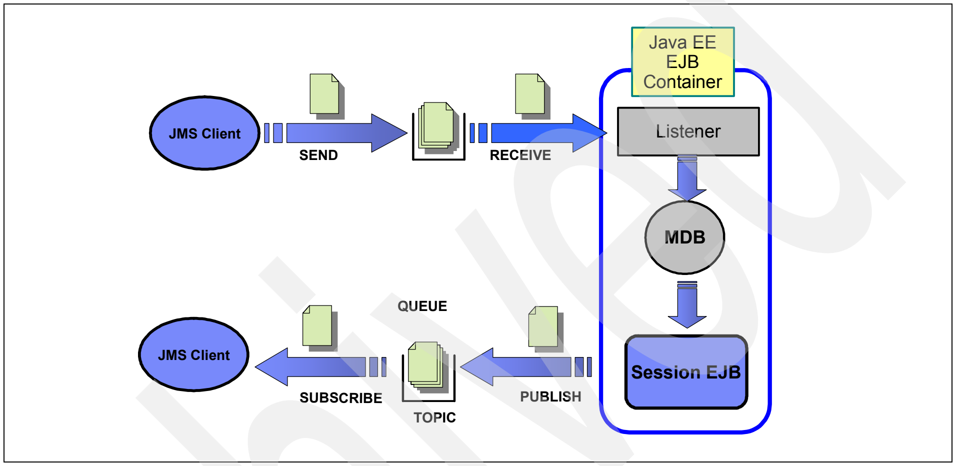


Figure 14-2 JMS with publisher and subscriber

There are no unique Learn resources for this chapter.

14.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed all the applications of Part 2, “Core Java EE application” on page 97. In addition, you must have completed Chapter 13, “Create the message-driven bean” on page 405.

14.3 Update the DonateBean session EJB

In this section, we update the `DonateBean` session EJB to publish the results of a `donateToFund` invocation to a topic:

1. In the Project Explorer, open **DonateBean** (in `DonateEJB/ejbModule/donate.ejb.impl`).
2. Add the required local variables for pub/sub processing, after the @EJB annotations, and before the `donateToFund` method:

```
//* Added for pub sub logic
@Resource(name = "jms/ConnectionFactory")
ConnectionFactory cf;

Connection connection;
Session session;
```

– Organize imports (**Ctrl+Shift+O**):

```
javax.jms.Connection
javax.jms.ConnectionFactory
javax.jms.Session
```

Behind the scenes:

What makes EJB development testing difficult are EJB code dependencies on data sources and JMS resources, as well as how EJB components are invoked by EJB clients. The EJB 3.0 specification introduces dependency injection as a mechanism to alleviate these difficulties. Instead of using JNDI lookups, an EJB can define a resource reference by injecting code. Here is an example of an EJB bean that has to call another EJB and use a data source to perform JDBC work:

```
@EJB
DonateBeanRemote donateBean;

@Resource (name="jdbc/donatedb")
private DataSource ds;
```

Dependency injection can occur in a variety of ways, for example, via a setter method or a class variable. See the specification for more details:

http://www.jcp.org/en/jsr/detail?id=220

This reference is taken from “Examining the EJB 3.0 Simplified API Specification,” available at:

http://www.ibm.com/developerworks/websphere/techjournal/0502_col_barcia/0502_col_barcia.html

3. Add a `createSession` method before the closing brace by inserting the **F95 MDB PubSub Create Session** snippet (Example 14-1).

Example 14-1 DonateBean pub/sub createSession

```
// Added for pub/sub logic
@PostConstruct
public void createSession() {
   try {
      connection = cf.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      connection.start();
   } catch (Exception e) {
      throw new EJBException("Donate: error in pub/sub createSession", e);
   }
}
```

Behind the scenes:

Before EJB 3.0 and annotations, developers had to use life cycle methods such as `ejbCreate`. Now, with annotations, it has become much easier. The EJB developer can simply use the @PostConstruct annotation before a method and this method is executed following the dependency injection. This is very useful for executing class initialization code.

4. Add a `closeSession` method before the closing brace by inserting the **F96 MDB PubSub Close Session** snippet (Example 14-2).

Example 14-2 DonateBean pub/sub closeSession

```
// Added for pub/sub logic
@PreDestroy
public void closeSession() {
   try {
      if (session != null) session.close();
   } catch (JMSException e) { e.printStackTrace(); }
   try {
      if (connection != null) connection.close();
   } catch (JMSException e) { e.printStackTrace(); }
}
```

Behind the scenes:

Similar to @PostConstruct, the EJB developer can use the @PreDestroy annotation to release the resources associated with the session and connection when the EJB is removed.

5. Add the `publishResults` method before the closing brace by inserting the **F97 MDB PubSub Publish Results** snippet (Example 14-3).

Example 14-3 DonateBean pub/sub publishResults

```
// Added for pub/sub logic
private void publishResults(int employeeId, String fundName, int hours,
String return_string) {
   try {
      Destination empTopic = session
                             .createTopic(String.valueOf(employeeId));
      MessageProducer empProducer = session.createProducer(empTopic);
      TextMessage pubMessage = session.createTextMessage("Request = "
            + employeeId + " " + fundName + " " + hours + " , result = "
            + return_string);
      empProducer.send(pubMessage);
      System.out.println("Message sent to " + empTopic + " : "
                         + pubMessage.getText());
      empProducer.close();
   } catch (Exception e) {
      System.out.println("Error in pub/sub publishResults: "
                         + e.getLocalizedMessage());
   }
}
```

t

Behind the scenes:

This code performs the following functions:

- Create a (topic) destination for the employee:

```
Destination empTopic =
      session.createTopic(String.valueOf(employeeId));
```

- Create a producer for the destination:

```
MessageProducer empProducer = session.createProducer(emp1Topic);
```

- Create the message:

```
TextMessage pubMessage = session.createTextMessage(.....);
```

- Publish the message:

```
empProducer.send(pubMessage);
```

- Free up resources:

```
empProducer.close();
```

6. Update the second donateToFund method (with simple arguments) to call the publishResults method:

```
public String donateToFund(int employeeId, String fundName, int hours) {
   String return_string = null;
   try {
      ......
   }
   publishResults(employeeId, fundName, hours, return_string);
   return return_string;
}
```

7. Organize imports (**Ctrl+Shift+O**) and select the following import:

```
javax.jms.Destination
```

8. Save and close DonateBean.java.

14.4 Create the messaging pub/sub client

In this section, we create a pub/sub client, DonatePubSubClient, which receives the published messages. These steps are similar to those used in 13.6, “Create the messaging client (DonateMDBClient)” on page 426.

1. Create the application client project:
   - In the Project Explorer, right-click **DonateEAR** and select **New** → **Application Client Project**.
   - In the New Application Client Project/Application Client module pop-up, set the Project Name to **DonatePubSubClient**, and add the project to the **DonateEAR** enterprise application. Click **Next**.
   - In the New Application Client Project/Application Client module pop-up, select **Create a default Main class**, and **Generate Deployment Descriptor**. Click **Next**.
   - In the New Application Client Project/Geronimo Deployment Plan pop-up, set the Group Id to **ejee**, leave the Artifact Id empty, and click **Finish**.
2. **Workaround**: Open **geronimo-application-client.xml** (in DonatePubSubClient/appClientModule/META-INF) to remove invalid lines (refer to 9.3.1, “Workaround: Update Geronimo deployment descriptor” on page 277 for details):

```
   ......
   </dep:server-environment>
    ~~<name:service-ref>~~
        ~~<name:service-ref-name>ServiceRefName</name:service-ref-name>~~
    ~~</name:service-ref>~~
</client:application-client>
```

3. In the Project Explorer, open **Main** (in `DonatePubSubClient/appClientModule/(default package)`).
4. In the `Main` editor, replace the entire the `main` method with the **F98 MDB PubSub Client Main** snippet from the Snippets view (Example 14-4).

Example 14-4 DonatePubSuBClient main method

```
public static void main(String[] args) throws NamingException, JMSException
{
   System.out.println("DonatePubSubClient: Establishing initial context");
   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
   env.put(Context.PROVIDER_URL, "tcp://localhost:61616");
   Context ctx = new InitialContext(env);

   System.out.println("DonatePubSubClient: Looking up ConnectionFactory");
   ConnectionFactory cf = (ConnectionFactory) ctx.lookup
                                        ("ConnectionFactory");

   System.out.println("DonatePubSubClient: Establishing connection");
   Connection connection = cf.createConnection();
   connection.start();

   System.out.println("DonatePubSubClient: Establishing session");
   Session session = connection.createSession(false,
                                              Session.AUTO_ACKNOWLEDGE);

   Destination empTopic = session.createTopic("1");
   MessageConsumer empSub = session.createConsumer(empTopic);

   System.out.println("DonatePubSubClient: Receiving messages for updates
                     to employee " + empTopic.toString());
   try {
      while (true) {
         System.out.println("Returned message = "
                        + ((TextMessage) empSub.receive()).getText());
      }
   } catch (Exception e) {
      // closing logic
      System.out.println("Exit exception block: " + e.getMessage());
      empSub.close();
      session.close();
      connection.close();
   }
}
```

5. Organize imports (**Ctrl+Shift+O**) and select the following imports when prompted:

```
javax.jms.Destination
javax.jms.Connection
java.util.Hashtable
javax.jms.ConnectionFactory
javax.jms.Session
javax.naming.Context
```

6. Add the following line before the `main` method to suppress the remaining warnings:

```
@SuppressWarnings("unchecked")
public static void main(String[] args) throws ...... {
   ......
}
```

7. Save and close `Main.java`.

Behind the scenes:

The processing of the pub/sub client is as follows:

- Create a connection factory using an initial context.
- Create and start a connection.
- Create a session, with a topic ("1"), and a consumer for the topic.
- Receive messages in a loop and print the content of the message.
- When done (cancelled), close everything nicely and exit.

14.5 Test the messaging pub/sub client

The DonatePubSubClient uses non-durable subscriptions, which means that it only receives messages published to the topic after it is started.

Therefore, in this section you first start the DonatePubSubClient, and then submit requests through the DonateMDBClient.

JMS supports the concept of durable subscriptions, which allows a program to receive messages that were sent to the topic while it was off-line. Reference 14.6, “Explore!” on page 443 for information on both durable subscriptions and a related topic of message persistence.

14.5.1 Start the DonatePubSubClient

The steps to start the pub/sub client are similar to the steps used in 13.7, “Test the messaging client” on page 430:

1. In the Servers view, publish the application.
2. Run the **client** command in an operating system prompt from the C:\IBM\WASCE21\bin directory.

   ```
   client ejee/DonatePubSubClient/1.0/car 1
   ```

3. Ensure that the following appears on the command line output:

   ```
   DonatePubSubClient: Establishing initial context
   DonatePubSubClient: Looking up ConnectionFactory
   DonatePubSubClient: Establishing connection
   DonatePubSubClient: Establishing session
   DonatePubSubClient: Receiving messages for updates to employee topic://1
   ```

14.5.2 Submit requests

Any request that invokes the DonateBean session EJB—whether started through an MDB, a Web browser, or a Web service—causes a message to be published to the topic:

1. Submit a request for employee 1 through the DonateMDBClient, by opening a command window at the C:\IBM\WASCE21\bin directory and executing the **client** command (be sure to enclose the arguments in quotes):

   ```
   client ejee/DonateMDBClient/1.0/car "1 DonationFund 1"
   ```

2. The command displays the following output:

   ```
   destination  = queue://jms/DonateQ
   DonateMDBClient: Establishing connection
   DonateMDBClient: Establishing session
   DonateMDBClient: Sending message
   DonateMDBClient: Message sent!
   ```

3. In the Eclipse Console view, verify that the DonateMDB received the message, called donateToFund, and that donateToFund published the results:

   ```
   DonateMDB: Input Message = 1 DonationFund 1
   Message sent to topic://1 : Request = 1 DonationFund 1 ,
                                   result = Transfer successful
   DonateMDB: The Result Message is: Transfer successful
   ```

4. In the command prompt window of the DonatePubSubClient, verify that the message was received:

   ```
   Returned message = Request = 1 DonationFund 1 ,
                                      result = Transfer successful
   ```

5. The DonatePubSubClient subscribes to updates for employee 1, but not for employee 2. Submit a donation for employee 2 in the DonateMDBClient command prompt:

   ```
   client ejee/DonateMDBClient/1.0/car "2 DonationFund 1"
   ```

   You get the confirmation that the message is sent, but no message is received in the DonatePubSubClient command prompt.

6. Optional: Run the JUnit test. Right-click **DonateBeanTester** (in the DonateEJB_Tester project) and select **Run As** → **JUnit test**. This test submits two donations, and you should see that the two message are received in the DonatePubSubClient:

7. Terminate the DonatePubSubClient by pressing **Ctrl+C** in the command window.

14.6 Explore!

JMS messages are maintained asynchronously from the producer and the consumer. Therefore, you have to consider how the messages should be managed, meaning how they are preserved (or not).

There are two concepts that apply:

- **Durability**, which refers to what happens to topic messages when the topic subscriber is off-line. JMS has two different API calls that determine if a durable or non-durable subscription is used (`session.createDurableSubscriber` and `session.createConsumer`).

  The default that we use in this document is non-durable subscriptions, which indicates that the subscriber only receives messages (that match that subscription) that arrive while the subscriber is active (after a `session.CreateConsumer` call is issued). If the subscriber disconnects and then reconnects later, it will not receive messages that were published while it was disconnected.

  – Advantages: A simpler programming model is possible for the program subscribing to the messages, and less overhead on the server side.

  – Disadvantages: The consumer does not receive all messages sent to the topic over time.

  Durable subscriptions receive all messages (that match the subscription) regardless of whether the client was active when the message was sent:

  – Advantages: The client receives all messages that match the topic.

  – Disadvantages: The publish and subscribe server has to maintain additional resources to store the messages and subscription information.

Durability is driven purely by the client program, which decides whether to access a topic as durable or non-durable.

- **Persistence**, which refers to how messages are preserved relative to the overall availability of the infrastructure. This applies to both queues and topics.

  Non-persistent indicates that messages are discarded when the messaging engine stops, fails, or was unable to obtain sufficient system resources. With this configuration, the messages are effectively maintained in a memory store:

  – Advantages: Messages are processed with a minimum of overhead because there is no file access.

  – Disadvantages: The messages might be lost.

  Persistent (the default value used in this book) indicates that messages are preserved. This is sometimes called *assured delivery*: The infrastructure guarantees that the message is available on the destination queue, but obviously cannot guarantee if or when a user program actually reads the message:

  – Advantages: All messages are preserved because they are written to a file store.

  – Disadvantages: Messages are processed with an additional overhead because there is file access.

  Persistence is determined by a hierarchy of settings:

  – The SIB destination definition, for example, queue, sets the “default reliability” (Best effort nonpersistent, Express nonpersistent, Reliable nonpersistent, Reliable persistent, Assured persistent) and can choose to fix this value and deny others the ability to override.

  – The JMS connection factory definition can set the *quality of service* characteristics both for persistent and non-persistent access. One of the options is *As bus destination*, which indicates that the access should default to that set in the SIB destination.

  – The JMS queue definition sets the *delivery mode* to persistent, non-persistent, or lets the client program decide.

  – The client program can set the delivery mode to persistent, non-persistent, or use the default delivery mode by using the `setDeliveryMode` method on the Message Producer object.

There are no unique Explore resources for this chapter.

15

Implement security for messaging

In this chapter, we update the MDB (DonateMDB) and the Java EE application client (DonateMDBClient) to operate with Java EE application security enabled (Figure 15-1).

15.1 Learn!

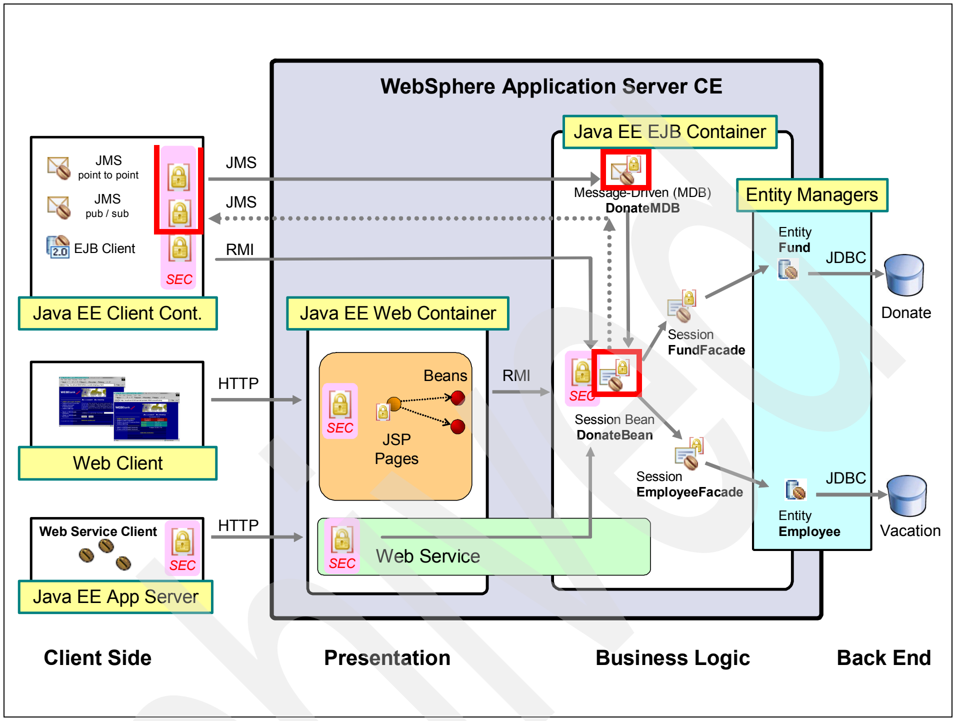


Figure 15-1 Donate application: Message security

Messaging has two levels of security:

- The first level controls interaction with the messaging provider to send and receive messages, for example, when we send JMS messages from a client, as in 13.6, “Create the messaging client (DonateMDBClient)” on page 426. We will update the client code with user ID and password information.
- The second level is setting the user context for the message received by an MDB. When an MDB receives a message through the `onMessage` method, which user ID should this message be executed under? This is important if the `onMessage` method then invokes a session EJB that is secured by either Java EE declarative security or programmatic security.

For the second level, Java EE provides the following two choices:

- Run without credentials (default), meaning that the `onMessage` method and any subsequent method or EJB in the calling chain runs as `UNAUTHENTICATED`. What happens if one of the EJBs requires a credential, for example, if role based security is applied to the EJB? The answer is that the access fails.

- Run as a specific user called a *run-as-identity*, defined through an entry in the EJB deployment descriptor, which points to a JAAS Authentication Alias (defined in the Java EE runtime environment) containing the user ID and password. The `DonateMDB` runs successfully without code changes if you add a run-as-identity.

Both of these Java EE alternatives have drawbacks:

- Running without client credentials means that you cannot access secured EJBs.
- Running with a run-as-identity means that all invocation of secured EJBs are done in the context of the same user, thus minimizing the value of implementing role-based security.

This scenario implements a third alternative: The message sender provides a user ID and password in the JMS properties header contained in the message. The `onMessage` method uses these to create a login context, and then invokes the `handleMessage` method using that login context.

The advantage of this approach (over the two Java EE provided alternatives) is that you maintain the integrity of the Java EE role-based security. The drawback is that users accessing the program must supply a user ID password in the JMS properties.

Alternatively, you can create a security realm with a custom login module, pass any information in the JMS header, extract the header, and pass information into a callback handler.

There are no unique Learn resources for this chapter. Instead, refer to the security specific sections in the Learn resources referenced in 13.1, “Learn!” on page 406.

15.2 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, “Jump start” on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed all the applications of Part 2, “Core Java EE application” on page 97. In addition, you must have finished Chapter 14, “Add publication of results” on page 433.

15.3 Re-enable Java EE EJB declarative security

In this section we perform the set of tasks required to re-enable the Java EE EJB declarative security that we disabled in 13.3, “Disable Java EE EJB declarative security” on page 410:

1. Re-enable declarative security in the `DonateBean` session EJB:
   - In the Project Explorer, open **DonateBean** (in `DonateEJB/ejbModule/donate.ejb.impl`), and switch the security annotations for both `donateToFund` methods (remove the comment marker from @RolesAllowed and add a comment marker to @PermitAll):

     ```
     @RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
     //@PermitAll
     public void donateToFund(Employee employee, Fund fund, int hours)
        ......

     @RolesAllowed( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
     //@PermitAll
     @WebMethod
     public String donateToFund(int employeeId, String fundName, ...)
        ......
     ```

   - Organize the imports (**Ctrl+Shift+O)**, save, and close `DonateBean.java`.
2. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.

15.4 Test the messaging artifacts as-is

In this section you test the current messaging artifacts to verify that the DonateMDB `onMessage` method fails to process the inbound message with Java EE application security enabled:

1. Run the **`client`** command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory. Ensure that arguments are enclosed in quotes.

   ```
   client ejee/DonateMDBClient/1.0/car "1 DonationFund 1"
   ```

2. The DonateMDBClient command output shows that a message is sent:

   ```
   DonateMDBClient: Establishing initial context
   DonateMDBClient: Looking up ConnectionFactory
   DonateMDBClient: Lookup up Queue
   DonateMDBClient: Establishing connection
   DonateMDBClient: Establishing session
   DonateMDBClient: Sending message
   DonateMDBClient: Message sent!
   ```

3. In the WASCE IDE Console view, verify that the `DonateBean` session EJB failed to process the message because the user context was `UNAUTHENTICATED`:

   ```
   DonateMDB: Input Message = 1 DonationFund 1
   DonateMDB: error in handleMessage: Unauthorized Access by Principal Denied
   ```

15.5 Update the DonateMDBClient and DonateMDB

In this section, we update the DonateMDBClient to put the user ID and password in the JMS header. We also update the DonateMDB `onMessage` method to extract these values, and then execute the donation as that user.

15.5.1 Update the DonateMDBClient code

Here we update the DonateMDBClient to put the user ID and password in the JMS header:

1. In the Project Explorer, open **Main** (in `DonateMDBClient/appClientModule /(default package)`).
2. Before sending the message using `destSender.send`, add two statements to set properties in the JMS header that specify the user ID (`DNARMSTRONG` or `DCBROWN`) and password on whose behalf the MDB should execute the `DonateBean` session EJB:

   ```
   TextMessage toutMessage = session.createTextMessage(args[0]);

   //* Added for MDB Security
   toutMessage.setStringProperty("userid", "DNARMSTRONG");
   toutMessage.setStringProperty("password","password");

   destSender.send(toutMessage);
   ```

3. Save and close `Main.java`.

15.5.2 Update the DonateMDB code

Next we update the DonateMDB `onMessage` method to extract these values, and then execute the donation as that user:

1. Add the WASCE security JAR file to the Java build path:
   – In the Project Explorer, right-click the **DonateEJB** project and select **Properties**.
   – In the resulting Properties pop-up, select **Java Build Path** and the **Libraries** tab.

- Click **Add External JARs**, and select the `geronimo-security-2.1.1.jar` (from `C:\IBM\WASCE21\repository\org\apache\geronimo\framework\geronimo-security\2.1.1`).
- Click **OK** to save and close.

Behind the scenes:

You might see this warning in the Problems view:

```
Classpath entry C:/IBM/WASCE21/repository/org/apache/geronimo
/framework/geronimo-security/2.1.1/geronimo-security-2.1.1.jar will
not be exported or published. Runtime ClassNotFoundExceptions may
result.
```

This indicates that we have a JAR file that is on the build path in Eclipse, but Eclipse is not sure if the JAR will be available in the runtime environment. This is because the WASCE library definition inside of Eclipse does not contain this JAR. The WASCE library is what Eclipse looks at when it does the validation: The JAR has to be included in the EAR or on the formal classpath.

However, in reality this JAR is in the runtime classpath of WASCE, and the Eclipse definition should really include this, but it does not.

Solution:

- In the Problems view, right-click the warning and select **Quick Fix**.
- In the Quick Fix pop-up, select **Exclude the associated raw classpath entry from the set of potential publish/export dependencies**, and click **Finish**.

2. In the Project Explorer, open **DonateMDB** (in `DonateEJB/ejbModule/donate.mdb`).
3. Locate the `onMessage` method, and replace the `handleMessage(arg0)` call with the **F99 MDB Security Message** snippet (Example 15-1).

Example 15-1 DonateMDB security onMessage method

```
public void onMessage(Message arg0) {
   //handleMessage(arg0);
   try {
      String tuserid = arg0.getStringProperty("userid");
      String tpassword = arg0.getStringProperty("password");
      LoginContext ctxLogin = ContextManager.login("ExperienceJavaEERealm",
            new UsernamePasswordCallback(tuserid,
                                             tpassword.toCharArray()));
```

```
         ContextManager.setNextCaller(ctxLogin.getSubject());
         System.out.println("DonateMDB: Subject set to "
                               + ctxLogin.getSubject().toString());
         handleMessage(arg0);
      } catch (Exception e) {
         System.out.println("DonateMDB: Exception: "
                               + e.getLocalizedMessage());
         e.printStackTrace();
      }
   }
```

4. Organize imports (**Ctrl+Shift+O**), save, and close `DonateMDB`.

Behind the scenes:

The processing of the updated `onMessage` method is as follows:

- Extract the user ID and password from the JMS properties:

```
String tuserid = msg.getStringProperty("userid");
String tpassword = msg.getStringProperty("password");
```

- Create the login context for this user and login:

```
ctxLogin = ContextManager.login("ExperienceJavaEERealm",
   new UsernamePasswordCallback(tuserid, tpassword.toCharArray()));
```

- Invoke `handleMessage` using the authenticated subject, from the login context:

```
ContextManager.setNextCaller(ctxLogin.getSubject());
handleMessage(arg0);
```

An alternative coding would set both the current and next caller:

```
Subject subject = ctxLogin.getSubject();
ContextManager.setCallers(subject, subject);
try {
   handleMessage(arg0);
} finally {
   ContextManager.setCallers(null, null);
}
```

15.6 Test the updated messaging artifacts

In this section, we test the updated messaging artifacts to verify that the MDB onMessage method successfully processes the inbound message and passes the security principal to the session bean:

1. In the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.
2. Run the **client** command in an operating system prompt from the C:\IBM\WASCE21\bin directory. Ensure that you enclose the arguments in quotes.

   ```
   client default/DonateMDBClient/1.0/car "1 DonationFund 1"
   ```

3. There is no change in the DonateMDBClient output.

   ```
   ......
   DonateMDBClient: Message sent!
   ```

4. In the WASCE IDE Console view, verify that the DonateMDB received the message, set the user context to DNARMSTRONG (or DCBROWN), and successfully invoked the DonateBean session EJB:

   ```
   DonateMDB: Subject set to Subject:
      Principal: DNARMSTRONG
      Principal: DUSERS
      Principal: ALLUSERS
      Principal: DMANAGERS
      Principal: org.apache.geronimo.security.IdentificationPrincipal[[.....]]

   DonateMDB: Input Message = 1 DonationFund 1
      ......
   EmployeeFacade: User id=DNARMSTRONG
   EmployeeFacade: isCallerInRole=true
   Message sent to topic://1 : Request = 1 DonationFund 2 ,
                               result = Transfer successful
   DonateMDB: The Result Message is: Transfer successful
   ```

15.7 Explore!

There are no unique Explore resources for this chapter.

Part 5

Advanced Java EE 5 and Web 2.0

In Part 5, we perform the advanced tasks described in the following chapters:

16

Develop a Web 2.0 client

In this chapter, we provide detailed information on how to develop a rich Web 2.0 application with HTML and JavaScript client code and servlet and Web service logic on the server.

The Web 2.0 implementation is using the **WebSphere Application Server Feature Pack for Web 2.0**, which provides Asynchronous JavaScript and XML (Ajax), JavaScript Object Notation (JSON), Representational State Transfer (REST), and the Dojo toolkit.

16.1 Learn

In this section we describe the basics of Web 2.0 technologies and how they are implemented in the WebSphere Application Server Feature Pack for Web 2.0.

16.1.1 What is a Web 2.0 client?

A Web 2.0 client is a Web application that has enhanced desktop-like responsiveness and much more intuitive and participatory user interface. The onset of mass availability of high speed Internet access combined with the powerful modern client machines with much more processor power and memory, seems to have created this new client-server paradigm where richer Web 2.0 applications are now replacing the traditional thin client (HTML only) Internet applications. These Web 2.0 applications, when run on modern Web browsers, are now capable of delivering rich user interfaces, which were before only possible with heavyweight thick graphical user interface (GUI) applications that ran on the native operating system.

Many of the popular Internet Web sites, such as Google Maps, Gmail, Yahoo Mail, Yahoo Maps, Flickr, Google Docs, and Acrobat®.com (and the list can go on forever), are using Web 2.0 technologies to provide users a more desktop-like interface, thus turning the Web browser into a pervasive operating system. The Web 2.0 technologies have been the main reason behind the success of some of these Web portals, as users have embraced them due to their superior user interface features that exceeded their expectations.

In a Web 2.0 application, the presentation handling is delegated to the client, where browsers use widgets developed using Web 2.0 technologies to render the data sent by the server. Therefore, the server is now only responsible for executing the program logic and to prepare and send the data requested by the client.

Business benefits of a Web 2.0 client

Some of the business benefits of adding Web 2.0 capabilities to Web applications include:

- A more interactive, differentiated experience that can lead to longer sessions and **increased customer loyalty**.
- Traditional Web applications have suffered from poor interactivity and responsiveness towards end users. Responsiveness, local actions which can result in **fewer abandoned transactions, higher completion rates, and higher end-user productivity**.

- Interaction in classic Web applications is based on a multi-page interface model, in which for every request the entire page is refreshed. In the Web 2.0 applications, the user interface conforms to the single page model or, in other words, does not follow the concept of multiple Web pages. The page can now instead be called a *view*, which is composed of several individual components that are updated or replaced independently, so that the entire page does not have to be reloaded on each user action. This, in turn, helps to increase the levels of interactivity, responsiveness, and **user satisfaction**.

16.1.2 Technologies for Web 2.0 client development

The Web 2.0 or rich Internet applications are not new to us. Most of the Java developers that embraced the Internet during its infancy period were particularly drawn to it because of the Java applet technology. Java applets were introduced in 1995 as part of the first version of Java by Sun Microsystems. Applets provided the ability to run dynamic data driven software applications on the browser, which was not possible before on a mainly static HTML only Internet. Applets also provided richer user interface previously available only with native applications.

Java applets were software applications or components that were written in Java and ran on the browser using the Java Virtual Machine (JVM). The applet code was hosted on the server but was transferred to the client (browser) in a bytecode when the applet was first executed. Applets ran completely on the client and interacted with the Java based applications on the server to get their data.

The new Web 2.0 or rich Internet applications follow a very similar client-server paradigm. Applets also enjoyed instant stardom, just like Web 2.0 technologies, and thousands of applet based applications were developed. These applications ranged anywhere from small utilitarian applications such as mortgage calculators to some big applications such as Yahoo's first generation finance portfolio manager and others such as bank teller and call center applications.

Applet based dynamic applications finally gave way to thin-client architecture where application servers executed most of the logic on the server side and generated HTML which was then sent to the browser. With some research you can find several reasons cited for failure of Java applets to become successful on the client, but we think some of the important reasons were that networks were much slower, the browser and JVM were in their first few generations, and the client computers were not nearly as powerful as they are today.

Today's personal computers are much more powerful than some servers used a decade ago. The browsers are now in their eighth generation and runtime environments available on the client are now also much more powerful.

Following are some of the main client side technologies that can be used to develop richer user interface for Web applications:

- Asynchronous JavaScript and XML (Ajax)
- Flex with Adobe® Flash browser plug-in
- JavaFX™ by Sun Microsystems
- Silverlight™ by Microsoft
- Rich client platform (RCP) by Eclipse Foundation
- XUL by Mozilla Foundation

Asynchronous JavaScript and XML (Ajax)

The term *Ajax* is a short form of Asynchronous JavaScript and XML. Ajax is an asynchronous programming pattern that is primarily based on JavaScript and XML technologies. Ajax is an open technique for creating rich user experiences in Web-based applications that does not require additional browser plug-ins. Ajax interfaces result in increased customer satisfaction and line of business productivity by streamlining end-user interaction with Web-based applications.

The other technologies in Ajax include XHTML, DHTML, and CSS, which provide a standards-based presentation. JavaScript is the programming language that is used to bind everything together. The Document Object Model (DOM) provides dynamic display and interaction. And XML can be used as the format for data transfer between client and server, although any text based pre-defined format can be used such as HTML, plaintext, and JavaScript Object Notation (JSON).

The core of Ajax technology is the JavaScript object `XMLHttpRequest`. With an `XMLHttpRequest` object, you can use JavaScript to make a request to the server and process the response. Examples of applications that use this technology include Google Web 2.0 application, such as Google Maps, Gmail, Google Docs, Google's Suggest dynamic lookup interface, and several other Web 2.0 applications available ont he Internet today, such as Yahoo Mail, Flickr, Facebook, and so forth. It is the JavaScript technology through the `XMLHttpRequest` object that talks to the server, which is not the normal application flow and this is where Ajax gets much of its magic.

Figure 16-1 illustrates an asynchronous request invocation with Ajax. In a traditional thin client Web application, when you fill out a form and submit it, the entire form is sent to the server. The server then processes the request (using server side technologies, such PHP, CGI, Java EE, or .NET, and sends back the markup of a new Web page. With Ajax, JavaScript technology is used to intercept the form data and send that data to a JavaScript code. JavaScript code then uses the `XMLHttpRequest` object to talk to the server side code. The server then sends data back to a callback JavaScript method that decides what to do with that data and how to render it on the client.

This object can use DOM to manipulate the structure of the existing HTML document on the browser and thus, update data elements, such as `<div>`, or update form fields on the fly, giving that desktop-like responsiveness and intuitive interaction to the application. Ajax updates the portion of the HTML page without refreshing the whole HTML page, and all without user intervention! This is the power of `XMLHttpRequest`, which can talk seemlessly back and forth with a server. The result is a dynamic, responsive, highly-interactive experience such as a desktop application, but with all the power of the Internet behind it (Figure 16-1).

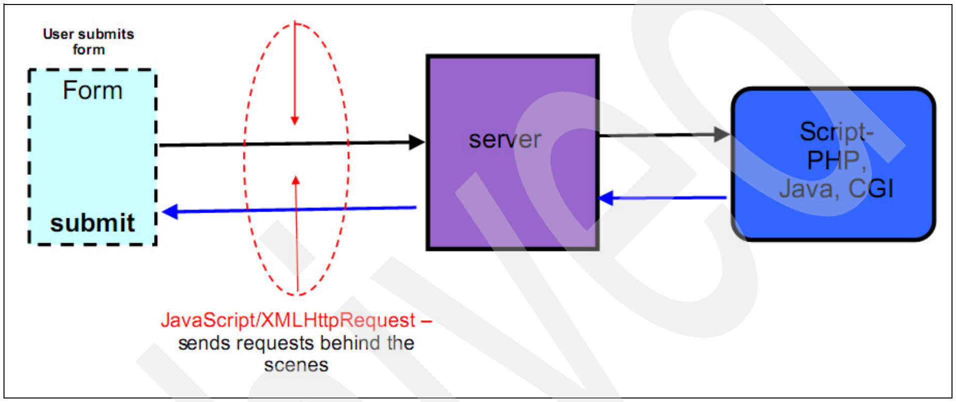

User submits form
Form
submit
server
Script-PHP, Java, CGI
JavaScript/XMLHttpRequest – sends requests behind the scenes

Figure 16-1 Client-server communication using the XMLHttpRequest object

Thus, Ajax bridges the gap between desktop applications and Web applications. Web pages are more responsive by seemlessly exchanging small amounts of data with the server. The entire Web page does not have to be reloaded each time a change is made, thus increasing Web page interactivity, speed, and usability. Ajax applications leverage standard browser features and do not require the installation of browser plug-ins.

Ajax can build clients that can front any server side application, such as Java EE, .NET, PHP, CGI, and so forth.

JavaScript Object Notation (JSON)

Java Script Object Notation (JSON) is a lightweight data interchange format. Just like XML, it is text based and easily readable by humans and can be easily parsed by machines. The JSON format was specified by Douglas Crockford in RFC-4627.

Markup languages such as XML and HTML were initially built to give semantic meaning to text with in documents. The following coding is an example of how markup can be used to provide specific meaning to a piece of a document:

```
<From>Maan Mehta</From>
<To>Ueli Wahli</To>
<Subject>This is a test message</Subject>
<Body>This is the body of the message. This is an
<bold>important</bold> message.</Body>
```

XML and especially HTML have done good job for this purpose. Over the last ten years, XML has also become a defacto standard markup language for message transfer between different systems and has played a very vital role in the evolution of Web services. On the other hand, HTML and its variants, such as XHTML and WML, have become a defacto markup languages for sending the data from the applications running on servers to browser-based clients.

JSON is a subset of JavaScript and is based on its array, string, and object literal syntax. JSON's format is language independent and its syntax is familiar to programmers of the C-family of languages, such as C, C++, Java, JavaScript, and Python. This has made it a good choice and an alternative to XML to describe data structures. Therefore, JSON can be used to serialize a data structure and transfer it over a network connection to the client (browser), where an Ajax application can deserialize the content into a JavaScript object by using a simple `eval` statement.

The following code snippet is the JSON representation of a JavaScript object that encapsulates the details of a person. The object has string attributes or fields for name, address, phone, and e-mail:

```
{
     "name": "Maan Mehta",
     "address": {
         "street": "27 Some Street",
         "city": "Toronto",
         "province": "ON",
         "postalCode": “M42 2M2”
     },
     "phone": "416-123-4567",
     "email": "maan@somewhere.ca",
}
```

If this string is contained in a JavaScript string variable called `person`, then we can create the JavaScript object by using the following `eval` statement:

```
var someOne = eval('(' + person + ')');
```

We can the access the value of the fields such as name, city, and email using the JavaScript dotted notation as follows:

```
var name = someOne.name;
var city = someOne.address.city;
var email = someOne.email;
```

On the server side, several parsers are available to serialize server side objects into the JSON format. IBM's Web 2.0 feature pack for WebSphere provides the JSON4J library to serialize the contents of the Java objects into the JSON format and vice-a-versa.

Representational State Transfer (REST)

Representational State Transfer (REST) is an architectural style for distributed systems. REST design the systems by defining its resources, these resources have a simple standard interface to be manipulated and to exchange its representations.

When applied in the Web, REST identifies resources by URIs, and it relies on the HTTP protocol for standard actions to manipulate the representations of the URIs. For example, a Web page is a resource that has a URI to be accessible. The HTTP method GET retrieves the representation of that Web page to a client.

The HTTP methods are mapped to actions on the resources as shown in Table 16-1.

Table 16-1 HTTP methods and concepts

Action	HTTP
Create	`PUT`
Read	`GET`
Update	`POST`
Delete	`DELETE`

16.1.3 WebSphere Feature Pack for Web 2.0

The IBM WebSphere Application Server Feature Pack for Web 2.0 extends the reach of service-oriented architecture (SOA) by connecting external Web services, internal SOA services, and Java EE objects into highly interactive Web application interfaces. To reduce IT costs and speed time to market, it provides a supported, open Ajax development framework that leverages existing SOA and Java EE assets to deliver rich user experiences.

Highlights and capabilities of the Feature Pack for Web 2.0 include:

- **Web 2.0 to SOA connectivity**: For enabling connectivity from Ajax clients and mash-ups to external Web services, internal SOA services, and Java EE assets. It also extends enterprise data to customers and partners through Atom Syndication Format (Atom) and Really Simple Syndication (RSS) Web feeds.

- **Ajax messaging**: For connecting Ajax clients to real-time updated data such as stock quotes or instant messaging.
- **Ajax development toolkit**: The Ajax development toolkit for WebSphere Application Server based on the Dojo toolkit with IBM extensions:
  - An enterprise-ready Ajax toolkit for building richer, more interactive Web sites (through the Dojo toolkit and IBM extensions).
  - Support for end-to-end Ajax patterns, including Java EE server integration (REST Web remoting, proxy services, Ajax messaging).
  - Lightweight data interchange in a format that is more easily consumable by Ajax clients (JSON4J library).
  - The ability to asynchronously push events through the use of the RSS and Atom.
  - Samples and tutorials for building end-to-end Ajax applications, using WebSphere Application Server products.
  - Support for tagging and feeds using RSS and Atom within Web applications.

The Feature Pack for Web 2.0 includes the following functionality:

- **Ajax client runtime**: Static content that can be added to either a WAR file or Web server.
- **Dojo toolkit**: JavaScript toolkit for creating Ajax client-side applications:

  http://dojotoolkit.org/
- IBM extensions to the Dojo toolkit that include additional widgets and client libraries:
  - **IBM Atom library**: This library provides client-side support of Atom feeds in the browser and allows for two-way communication with those feeds using the Atom Publishing Protocol (APP). Besides the base library, this also includes a reference implementation application of the dojo.data APIs and a collection of three Atom widgets for a rich viewing and editing of Atom feeds.
  - **IBM Gauge Widget library**: This library includes a pair of widgets (AnalogGauge and BarGraph) for displaying numerical data using Scalable Vector Graphics (SVG) or Vector Markup Language (VML), depending on the browser.
  - **IBM SOAP library**: This library is a Dojo toolkit extension that makes it easy for an application to connect to an existing SOAP-based Web service. Using this library you can make remote procedure calls to invoke the Web service methods directly from the client, the browser.

  - **IBM Open Search library**: This library makes it easy to invoke any Open Search-compliant service and to bind search results to widgets within the Ajax application
- **Ajax connectivity**:
  - Ajax proxy: This proxy can be used to broker client requests from multiple Domains while using Ajax. JavaScript sandboxing rules prevent the execution of network requests to servers from where the JavaScript did not originate. As an example, if the JavaScript application originated from domain A and attempts to use an `XMLHTTPRequest` to domain B, the browser will prevent the domain B request. The proxy can be used to broker the request, which provides the appearance to the client that the request came from the same server from which the JavaScript originated.
  - Remote procedure call (RPC) adapter.
  - Abdera-based library support.
- **Ajax developer's guide**
- **JSON4J library**
- **Asynchronous Web 2.0 messaging service**: The Web messaging service is a publish or subscribe implementation connecting the browser to the WebSphere Application Server service integration bus (SIB) for server-side event PUSH to the browser. The Web messaging service can enable a Web service or any other item connected to the SIB to PUSH data to the browser. You can use the Web messaging service in a new or existing application by placing a utility file library (JAR) in an application Web module, setting up a simple configuration file, and configuring servlet mappings.

JSON4J library

JSON4J library is an implementation of a set of JSON handling classes for use within Java environments. The JSON4J library provides the following functions:

- A simple Java model for constructing and manipulating data to be rendered as the JSON implementation.
- A fast transform for XML to JSON conversion, for situations where conversion from an XML reply from a Web service into a JSON structure is desired for easy use in Ajax applications. The advantage of this is that Ajax-patterned applications can handle JSON formatted data without having to rely on ActiveX® objects in Microsoft Internet Explorer XML transformations and other platform-specific XML parsers. In addition, JSON-formatted data tends to be more compact and efficient to transfer.

- A JSON string and stream parser that can generate the corresponding `JSONObject`, which represents that JSON structure in Java. You can then make changes to that `JSONObject` and serialize the changes back to the JSON implementation.

The API is distributed as an optional library and set of documentation that is included in the Feature Pack for Web 2.0. The library includes the Java class files packaged in a Java archive (JAR) format, `JSON4J.jar`, in the `lib` folder of the stand-alone package of the feature pack. This library can be used for stand-alone Java applications or Java EE Web applications that have to parse and handle JSON text. For Java EE Web applications, package the `JSON4J.jar` file in the `WEB-INF/lib` directory of the Web application archive (WAR) file.

The JSON4J library provides the following two important classes:

```
com.ibm.json.java.JSONArray
com.ibm.json.java.JSONObject
```

`JSONObject` class has a static method called `parse` that takes a parameter of type `String` and if that string is of valid JSON format, it creates and returns a deserialized Java Object. The `JSONObject` class also provides `get` and `put` methods to retrieve or set the values of properties of the Java object of type `JSONObject`. Example 16-1 illustrates the usage of these methods.

Example 16-1 Using parse and get method of the JSONObject Class

```
String jsonString = "{\"name\":\"Maan Mehta\", \"phone\":\"416-123-4567\"}";
JSONObject javaObject = JSONObject.parse(jsonString);
String fullName = javaObject.get("name");
String phoneNumber = javaObject.get("phone");
System.out.println("Name is: " + fullName + " & the Phone is: " + phoneNumber);
```

The output of code snippet in Example 16-1 is:

```
Name is: Maan Mehta and the Phone is: 416-123-4567
```

The `JSONObject` class also provides a `serialize` method that serializes and returns the Java object as a string in the JSON format. Example 16-2 illustrates the usage of the `serialize` method.

Example 16-2 Using the serialize method of the JSONObject class

```
String jsonStr = javaObject.serialize(true);
System.out.println("JSON representation of the Java Object is:\n" + jsonStr);
```

The output of code snippet in Example 16-2 is:

```
JSON representation of the Java Object is:
{
   "name": "Maan Mehta",
   "phone": “416-123-4567”
}
```

Feature Pack sample applications

The Feature Pack also includes the following sample applications that come with complete source code and ready to deploy and run on WASCE and other WebSphere Application Servers.

Feed samples

Atom Syndication Format and RSS are two standardized specifications used to deliver content feeds. Apache Abdera is the open-source project has been chosen to provide feed support within WebSphere Application Server.

Apache Abdera addresses and provides support for:

- Reading RSS content
- Atom syndication format
- Atom publishing protocol

The feed samples demonstrate these capabilities and have simple examples to read an RSS feed, create an atom feed, and read atom content.

Courier application sample

This sample Web application include simple examples that domesticate the following capabilities of RPC adapter:

- Handling of JSON response using JSON service
- Handling of XML response using XML RPC
- White and black list methods, and
- Specifying validators

The courier application is also available as an Eclipse plug-in that can be imported into an Eclipse workspace.

Web 2.0 Plants by WebSphere sample

The Feature Pack comes with a modified version of the well known Plants by WebSphere sample Java EE Web application from IBM. The Plants by WebSphere application is a fictional Web site that makes available nursery items such as vegetables, flowers, accessories, and trees. You can view an online catalog, select items, and add them to a cart. When the cart contains items, you can proceed to log in, supply credit card information, and check out.

In this sample, Dojo widgets have been used to enhance the user interface of the application. A new Dojo widget has also been created to enable drag and drop capabilities for the shopping cart. The `dojo.xhr` (GET/POST) calls have been used for the client-server communication and on the server side, an RPC adapter layer has been created to map traditional Java EE constructs, such as Enterprise Java Beans (EJBs), Web services, and POJOs, to lightweight constructs, such as JSON or XML using JSON4J and XML RPC.

Quote Streamer sample

This sample application demonstrates usage of the Web 2.0 messaging service by pushing realtime updates of stock quotes to the browser using the `cometd` transport capabilities.

16.2 Web 2.0 scenario use cases

In this section, we define the scenario use cases for the Donate sample application based on Web 2.0 technologies.

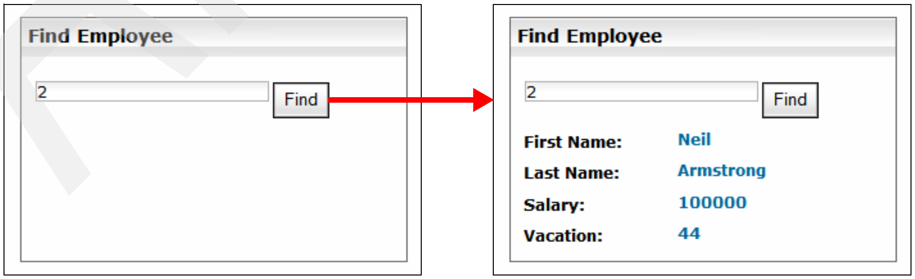
16.2.1 Use case requirements

We follow the same use cases that we have used in other chapters. The objective of the exercises in this chapter is to give you a head start in developing a Web 2.0 client for a Java EE server application using the Feature Pack for Web 2.0.

We developed two use case scenarios: Find Employee and Donate Vacation.

Find Employee

In this use case scenario (Figure 16-2), the user selects an employee from the list of Employees pre-populated in a drop-down and clicks **Find** to retrieve and view the employee details. The employee details consist of first and last name and the number of vacation hours remaining.

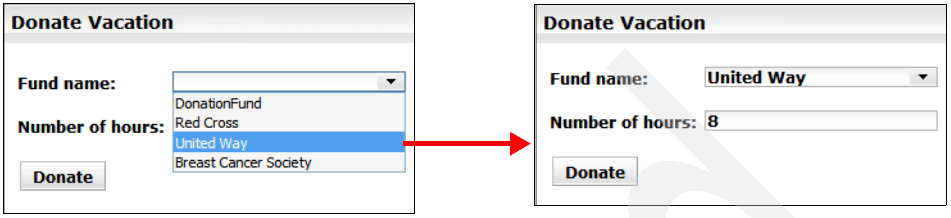


First Name: Neil
Last Name: Armstrong
Salary: 100000
Vacation: 44

Figure 16-2 Find Employee use case

Donate Vacation

In this use case scenario, the employee selects the fund to donate vacation hours, enter the number of hours to donate, and click **Donate**.



Donate Vacation

Fund name:

Number of hours:

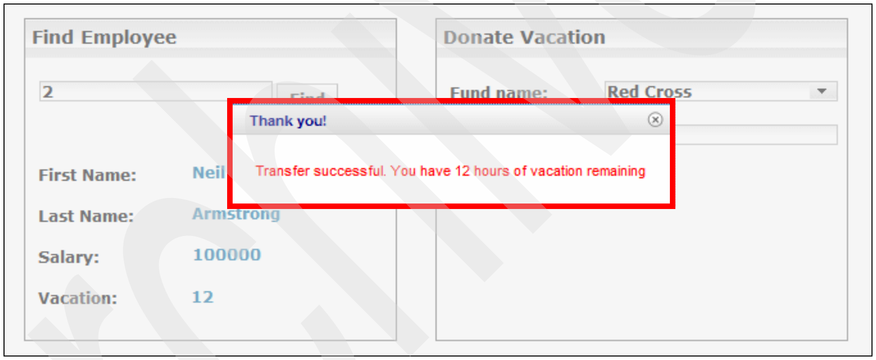
Donate Vacation

Fund name: United Way

Number of hours: 8

Figure 16-3 Donate Vacation use case

If the transaction is successful, a success status message is displayed in a modal dialog (Figure 16-4). The status message also displays the remaining vacations hours. If there is an error during the transaction, the modal dialog displays the error message instead.



Find Employee

2

First Name: Neil

Last Name: Armstrong

Salary: 100000

Vacation: 12

Donate Vacation

Fund name: Red Cross

Thank you!

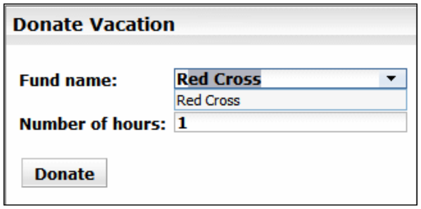
Transfer successful. You have 12 hours of vacation remaining

Figure 16-4 An Ajax modal dialog displays the status message

16.2.2 Rich user interface requirements

This section highlights the rich user interface requirements that we are going to deliver using the features provided by the Feature Pack for Web 2.0.

- Single page model: No page flicker and reloading of the whole page.
- Pre-population of drop-down fields: To eliminate errors, we pre-populate the donation fund names field (Figure 16-3).
- Type-ahead support: The input field for the donation fund name supports the type-ahead feature (Figure 16-5).



Donate Vacation

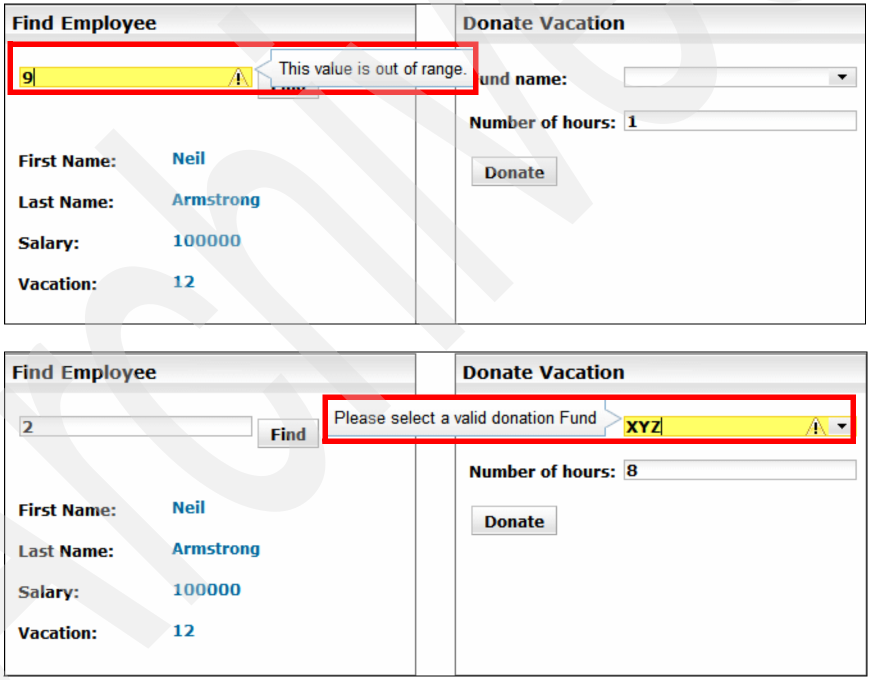
Fund name: Red Cross

Red Cross

Number of hours: 1

Figure 16-5 Type-ahead support to select the fund name

- Client-side validation: We use out-of-the-box input validation features provided by Dojo form widgets and also implement custom validation logic. For example, the employee number has a range constraint, and the fund name must be from those listed in the pre-populated drop-down field (Figure 16-6), and the amount of vacation an employee can donate should be less than their remaining vacation.



Find Employee

9

This value is out of range.

First Name: Neil

Last Name: Armstrong

Salary: 100000

Vacation: 12

Donate Vacation

und name:

Number of hours: 1

Find Employee

2

First Name: Neil

Last Name: Armstrong

Salary: 100000

Vacation: 12

Donate Vacation

Please select a valid donation Fund

XYZ

Number of hours: 8

Figure 16-6 Input field validation

- Ajax modal dialogs for status and error messages (see Figure 16-4 on page 467).

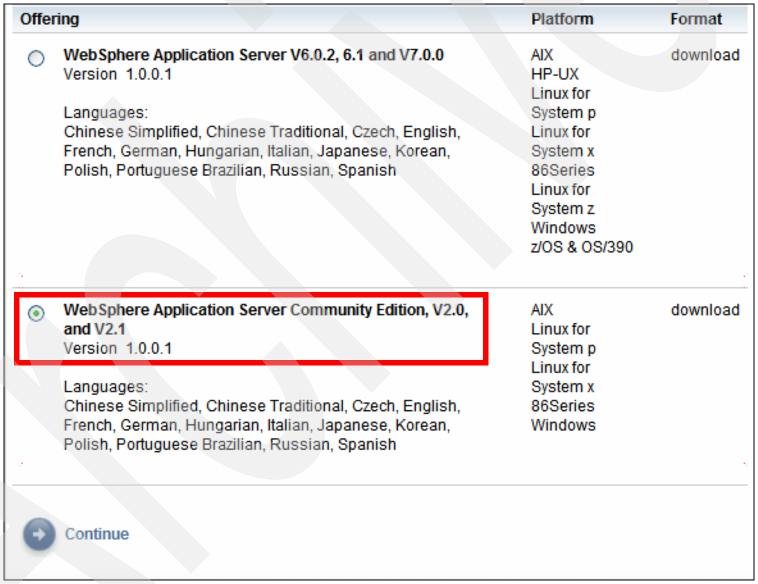
16.3 Installing the Feature Pack for Web 2.0 v1.0.0.1

The Feature Pack for Web 2.0 v1.0.0.1 is an optionally installable product extension for IBM WebSphere Application Server. The Feature Pack for Web 2.0 is also available for IBM WebSphere Application Server v6.0, v6.1, and v7.0 in addition to WebSphere Application Server Community Edition v2.0 and v2.1.

- You can download the Feature Pack by visiting the following URL for the IBM Product page for this feature pack:

  `http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/web20/`

- On this page, click **Download Feature Pack for Web 2.0 now**. You are asked to authenticate using IBM credentials. After authentication, you are taken to the download page where you are presented with options to download the feature pack relevant to your application server and its version. Select the option to download for **WebSphere Application Server Community Edition, V2.0, and V2.1** and click **Continue** (Figure 16-7).

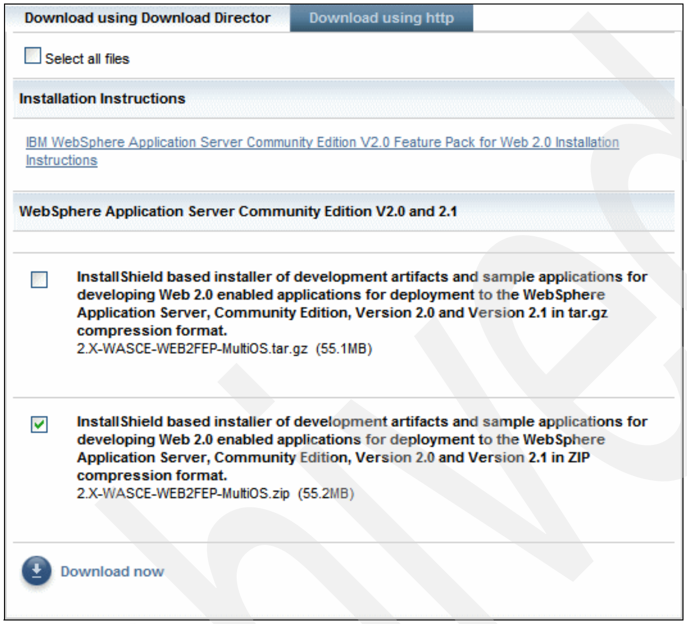


Offering	Platform	Format
○ **WebSphere Application Server V6.0.2, 6.1 and V7.0.0** Version 1.0.0.1 Languages: Chinese Simplified, Chinese Traditional, Czech, English, French, German, Hungarian, Italian, Japanese, Korean, Polish, Portuguese Brazilian, Russian, Spanish	AIX HP-UX Linux for System p Linux for System x 86Series Linux for System z Windows z/OS & OS/390	download
● **WebSphere Application Server Community Edition, V2.0, and V2.1** Version 1.0.0.1 Languages: Chinese Simplified, Chinese Traditional, Czech, English, French, German, Hungarian, Italian, Japanese, Korean, Polish, Portuguese Brazilian, Russian, Spanish	AIX Linux for System p Linux for System x 86Series Windows	download

Figure 16-7 Select to download for the appropriate application server

- On the following Web page, select to download the InstallShield based installer in the tar.gz or zip compression format (Figure 16-8):
  - Select **2.X-WASCE-WEB2FEP-MultiOS.zip** for Windows and **2.X-WASCE-WEB2FEP-MultiOS.tar.gz** for Linux.

- The size of the ZIP file is about 55 MB. The various development artifacts of the Feature Pack and sample applications have been made available in this simple zip file. About 150 MB of free disk space is required.
- Click **Download now**.



□ Select all files

Installation Instructions

IBM WebSphere Application Server Community Edition V2.0 Feature Pack for Web 2.0 Installation Instructions

WebSphere Application Server Community Edition V2.0 and 2.1

□ **InstallShield based installer of development artifacts and sample applications for developing Web 2.0 enabled applications for deployment to the WebSphere Application Server, Community Edition, Version 2.0 and Version 2.1 in tar.gz compression format.**
2.X-WASCE-WEB2FEP-MultiOS.tar.gz (55.1MB)

☑ **InstallShield based installer of development artifacts and sample applications for developing Web 2.0 enabled applications for deployment to the WebSphere Application Server, Community Edition, Version 2.0 and Version 2.1 in ZIP compression format.**
2.X-WASCE-WEB2FEP-MultiOS.zip (55.2MB)

Figure 16-8 Select the download in tar.gz or zip format

Installation of the Feature pack for Web 2.0 on Windows

The following steps describe how to install the Feature Pack for Web 2.0:

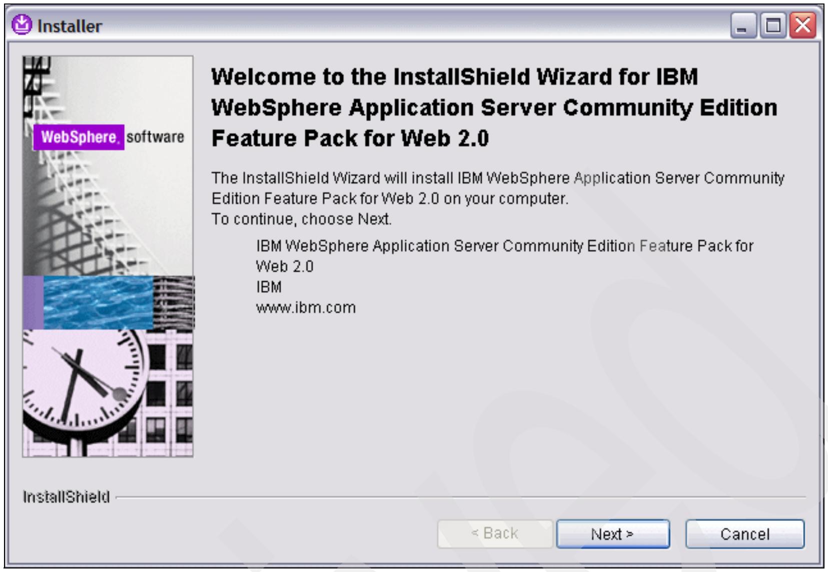
- Extract the Feature Pack to a temporary folder.
- Execute the `install.exe` from the command line for the Windows systems (or `install.sh` for AIX and Linux based systems) to invoke the Install shield based install wizard as follows:

  ```
  install.exe -is:javahome <JAVA_HOME>
  ```

  `<JAVA_HOME>` is the installation directory of the Java Virtual Machine (JVM). The installer requires a JVM to run. You can use a Java 1.4.2 or Java 5.0 compatible JVM. The recommended approach is to use the same JVM location used by WASCE. Enclose the directory in quotes if it contains spaces:

  ```
  install.exe -is:javahome "c:\Program Files\IBM\Java50"
  ```

- The InstallShield Wizard opens (Figure 16-9). Click **Next**.



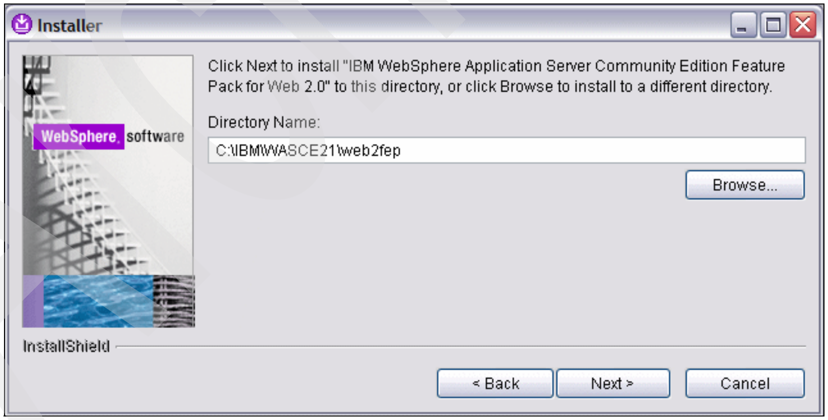
Welcome to the InstallShield Wizard for IBM WebSphere Application Server Community Edition Feature Pack for Web 2.0

The InstallShield Wizard will install IBM WebSphere Application Server Community Edition Feature Pack for Web 2.0 on your computer.
To continue, choose Next.

IBM WebSphere Application Server Community Edition Feature Pack for Web 2.0
IBM
www.ibm.com

Figure 16-9 Install wizard for Web 2.0 feature pack for WASCE

- On the next page, accept the license agreement and click **Next**.
- On the next page, you can select any directory on your file system to install the Feature Pack to. It does not have to be the installation directory of WASCE, but we recommend to install it there (Figure 16-10). Click **Next**.



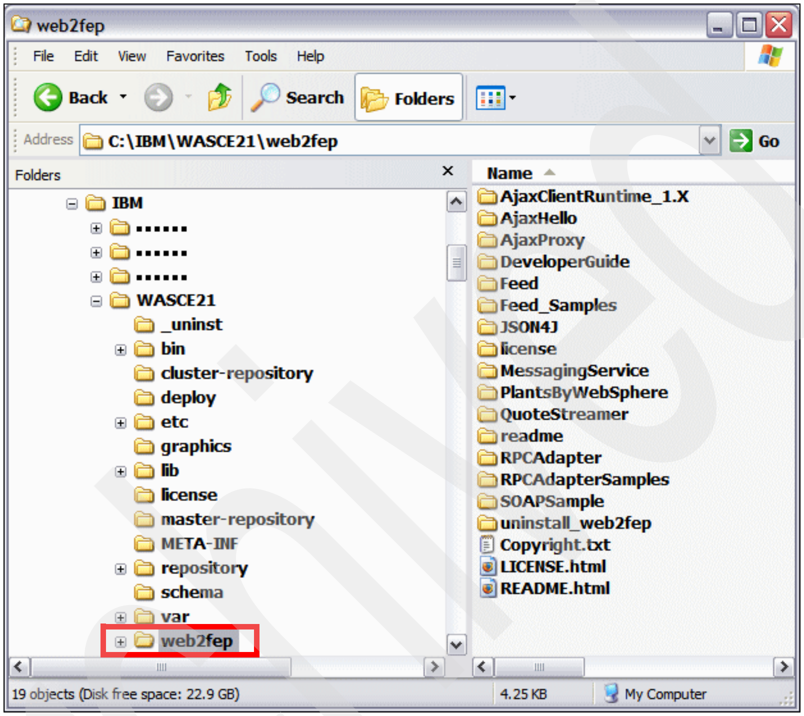
Click Next to install "IBM WebSphere Application Server Community Edition Feature Pack for Web 2.0" to this directory, or click Browse to install to a different directory.

Directory Name: C:\IBM\WASCE21\web2fep

Figure 16-10 Enter directory to install the feature pack

- Click **Next** on the next screen that summarizes the install options.
- The installation proceeds, and displays the results. Click **Finish**.

- After you have installed Feature Pack for Web 2.0, you can review the contents of the install folder (Figure 16-11). You can begin using the feature pack by working with the various included features and applications. The documentation for the individual features can be found within the individual folders.



C:\IBM\WASCE21\web2fep

```
IBM
  ......
  ......
  ......
  WASCE21
    _uninst
    bin
    cluster-repository
    deploy
    etc
    graphics
    lib
    license
    master-repository
    META-INF
    repository
    schema
    var
    web2fep
```

```
AjaxClientRuntime_1.X
AjaxHello
AjaxProxy
DeveloperGuide
Feed
Feed_Samples
JSON4J
license
MessagingService
PlantsByWebSphere
QuoteStreamer
readme
RPCAdapter
RPCAdapterSamples
SOAPSample
uninstall_web2fep
Copyright.txt
LICENSE.html
README.html
```

19 objects (Disk free space: 22.9 GB) 4.25 KB My Computer

Figure 16-11 Contents of the Install root of the feature pack

Table 16-2 describes the folders the install wizard creates under the install root of the Feature Pack.

Table 16-2 Feature Pack for Web 2.0 contents

Directory	Description
`AjaxClientRuntime_1.x`	Ajax client runtime contains a development copy of the Dojo toolkit plus IBM extensions. Copy these into your application WAR files as static resources, or place them in your Web server as static resources and reference them from your application.
`AjaxHello`	Sample application to illustrate the basics of developing applications with the Dojo toolkit.

Directory	Description
DeveloperGuide	A starting point for learning about Ajax Development, debugging, and usage patterns.
Feed	Abdera libraries for ATOM document processing.
Feed_Samples	Samples created by IBM to illustrate examples of feed usage.
JSON4J	JSON4J library, set of utility classes intended for use in a variety of Java applications that serialize and deserialize JavaScript Object Notation (JSON) formatted data.
MessagingService	Client library for using the Web messaging feature.
PlantsByWebSphere	Sample application that provides the Web 2.0 client for the WebSphere's sample application.
QuoteStreamer	Sample application that demonstrates the usage of the Web messaging service by displaying realtime updates of stock quotes using the cometd transport capabilities.
RPCAdapter	IBM RPC Adapter library for Web-remoting.
RPCAdapterSamples	Sample applications for the RPCAdapter feature, CourierApp and HelloWorld.
SoapSample	Sample application to illustrate the basics of invoking a SOAP service using the IBM SOAP extension of the Dojo toolkit.

16.4 Jump start

If you want to start with this chapter and skip some or all of the preceding chapters, follow the instructions in Appendix A, "Jump start" on page 557 to create the starting configuration for this chapter.

For this chapter to work, make sure that you have completed all the applications of Part 2, "Core Java EE application" on page 97.

You can also start with the finished applications of Part 3, "Web services" on page 321 or Part 4, "Messaging" on page 403.

16.5 Disable Java EE EJB application security

This chapter does not contain the application code required to support a secure session EJB. Therefore, before starting this chapter, we have to disable the Java EE EJB security:

1. Disable declarative security in the DonateBean session EJB:
   - In the Project Explorer, open **DonateBean** (in DonateEJB/ejbModule /donate.ejb.impl).
   - Before the two donateToFund methods, comment out the @RolesAllowed statement and add a @PermitAll statement.

```
//@RolesAllowed( { "DMANAGERS_ROLE", "DUSERS_ROLE" })
@PermitAll
public String donateToFund(Employee employee, ......)
   ......

//@RolesAllowed( { "DMANAGERS_ROLE","DUSERS_ROLE" })
@PermitAll
public String donateToFund(int employeeId, ......)
```

   - Save and close DonateBean.java.
2. Disable programmatic security in the EmployeeFacade session bean:
   - In the Project Explorer, open **EmployeeFacade** (in DonateEJB/ejbModule /vacation.ejb.impl).
   - Comment the two statements:

```
//if (!ctx.isCallerInRole("DMANAGERS_ROLE"))
// clone.setSalary(null);
```

   - Save and close EmployeeFacade.java.

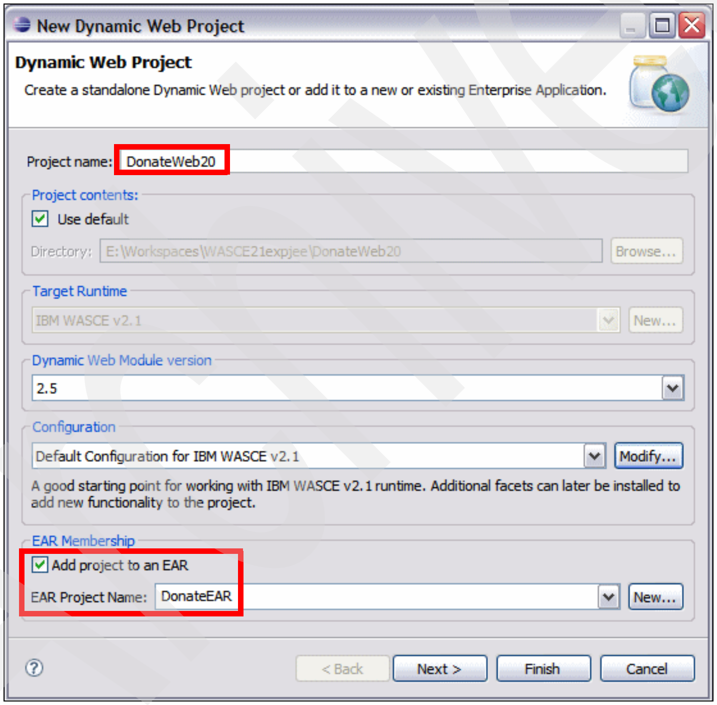
16.6 Setup the development environment

In this section we prepare a Web project for the development of the Web 2.0 sample application.

16.6.1 Create a Dynamic Web project

We use a new Dynamic Web project for the Web 2.0 sample application:

- Right-click in the Project Explorer and select **New** → **Dynamic Web Project**.
- In the New Dynamic Web Project pop-up (Figure 16-12):
  - Type **DonateWeb20** as the Project name.
  - Select **2.5** for Dynamic Web Module version.
  - Select **Default Configuration for IBM WASCE v2.1** for Configuration.
  - Select **Add project to an EAR** and select **DonateEAR** as the EAR Project Name.
  - Click **Next**.



Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: DonateWeb20

Project contents:

☑ Use default

Directory: E:\Workspaces\WASCE21expjee\DonateWeb20 Browse...

Target Runtime

IBM WASCE v2.1 New...

Dynamic Web Module version

2.5

Configuration

Default Configuration for IBM WASCE v2.1 Modify...

A good starting point for working with IBM WASCE v2.1 runtime. Additional facets can later be installed to add new functionality to the project.

EAR Membership

☑ Add project to an EAR

EAR Project Name: DonateEAR New...

Figure 16-12 Create a new Dynamic Web Project

- In the Web Module pop-up, leave the default values, select **Generate deployment descriptor**, and click **Next**.
- In the Geronimo Deployment Plan pop-up, type **ejee** for the Group Id, leave the Artifact Id empty, and click **Finish**.

16.6.2 Copy the client runtime artifacts

In this section we add the required code from the Feature Pack to the Web 2.0 project:

1. Copy the **dijit**, **dojo**, and **dojox** folders from the Feature Pack installation folder `AjaxClientRuntime_1.x` to the `WebContent` folder of the Web project:

   ```
   From: c:/IBM/WASCE21/web2fep
   To:   DonateWeb20/WebContent
   ```

   – You can select the three folder in the Windows Explorer, right-click and select **Copy**, then right-click the **WebContent** folder in the WASCE IDE and select **Paste**.

   – The three folders are development copies of the dojo runtime available as part of the Feature Pack.

   – Because we do not use the features provided by the IBM extensions, for example, Web 2.0 support to consume SOAP Web services or read Atom feeds, we do not have to copy other client runtime folders to the Web project.

   – Figure 16-13 shows the structure of the DonateWeb20 project.

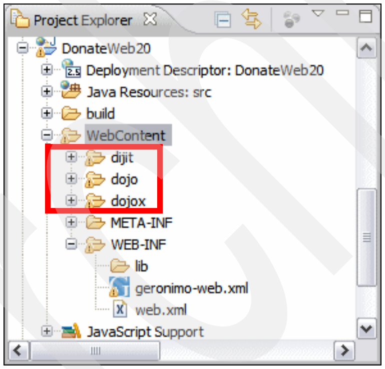


Figure 16-13 Donate Web project with dojo client runtime folders

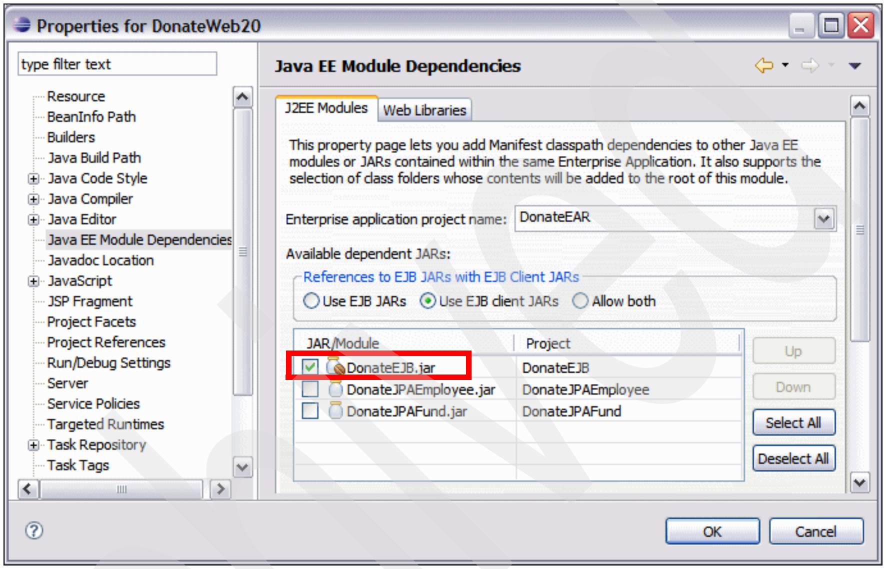
2. Copy the **JSON4J.jar** file from the Feature Pack `web2fep/JSON4J/lib` folder to the `DonateWeb/WebContent/WEB-INF/lib` folder.

16.6.3 Add the dependency to the EJB project

In this Web project, we access an EJB session bean. Therefore, we have to set up the dependency of the Web project to the EJB project.

By adding the DonateWeb20 project to the DonateEAR enterprise application, a dependency has been created in DonateEAR. Now we can set up further dependencies between projects contained in DonateEAR.

- Right-click the **DonateWeb20** project and select **Properties**.
- In the Properties dialog, select **Java EE Module Dependencies**, and select the **DonateEJB.jar** file. Click **OK** (Figure 16-14).



This property page lets you add Manifest classpath dependencies to other Java EE modules or JARs contained within the same Enterprise Application. It also supports the selection of class folders whose contents will be added to the root of this module.

Enterprise application project name: DonateEAR

Available dependent JARs:

References to EJB JARs with EJB Client JARs

○ Use EJB JARs ● Use EJB client JARs ○ Allow both

JAR/Module	Project
☑ DonateEJB.jar	DonateEJB
□ DonateJPAEmployee.jar	DonateJPAEmployee
□ DonateJPAFund.jar	DonateJPAFund

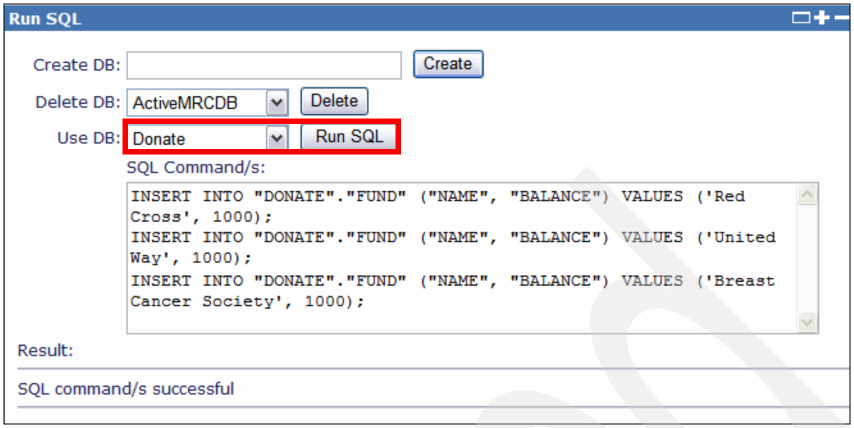
Figure 16-14 Add dependency to the DonateEJB project

16.6.4 Populate the Fund table

To demonstrate some of the features of Web 2.0 components, we have to populate the `FUND` table of the `Donate` database with additional fund names:

1. Start the WASCE Administrative Console (right-click the server in the Servers view and select **Launch WASCE Console**).
2. On the left, select **Embedded DB** → **DB Manager**, and on the right, select the **Donate** database from the drop-down list of Use DB.
3. Copy and paste the SQL statements from the **F108 Web 2.0 Populate Fund Table** snippet into the SQL Command/s text area and click **Run SQL** (Figure 16-15).

   The results message `SQL command/s successful` is displayed underneath the SQL Command/s text area.



```
INSERT INTO "DONATE"."FUND" ("NAME", "BALANCE") VALUES ('Red
Cross', 1000);
INSERT INTO "DONATE"."FUND" ("NAME", "BALANCE") VALUES ('United
Way', 1000);
INSERT INTO "DONATE"."FUND" ("NAME", "BALANCE") VALUES ('Breast
Cancer Society', 1000);
```

Result:

SQL command/s successful

Figure 16-15 Populate the FUND table with additional funds

4. Optionally add more employees to the `EMPLOYEE` table by running the **F109 Web 2.0 Populate Employee Table** snippet against the `Vacation` database.

16.7 Implement the Find Employee use case

In this section we implement the Find Employee use case using the REST technology.

16.7.1 Create a REST servlet

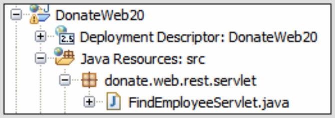
Here, we create a REST servlet that retrieves the details of an employee by interacting with the `EmployeeFacade` session EJB through simple EJB injection and access. This servlet then serializes the state of the `Employee` Java object into a JSON string and writes that string to the HTTP response output stream. The Web 2.0 client components then call this servlet to retrieve and display the details of the `Employee` object:

1. In the Project Explorer, right-click **DonateWeb20** project, and select **New** → **Servlet**.
2. In the Create Servlet pop-up, provide the following data:
   – Java package: **donate.web.rest.servlet**
   – Class name: **FindEmployeeServlet**
   – Superclass: Accept `javax.servlet.http.HttpServlet`
   – Click **Finish**.

Behind the scenes:

In creating a servlet, a servlet class and a corresponding mapping in the `web.xml` are created:

- The servlet class is created in the `donate.web.rest.servlet` package.



- The corresponding entries are inserted into the Web deployment descriptor (`DonateWeb20/WebContent/WEB-INF/web.xml`).

```
<servlet>
    <description></description>
    <display-name>FindEmployeeServlet</display-name>
    <servlet-name>FindEmployeeServlet</servlet-name>
    <servlet-class>donate.web.rest.servlet.
    FindEmployeeServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FindEmployeeServlet</servlet-name>
    <url-pattern>/FindEmployeeServlet</url-pattern>
</servlet-mapping>
```

3. Delete all the generated method skeletons (default constructor, `doGet`, and `doPost`), leaving the base class definition and the `serialVersionUID`.

```
public class FindEmployeeServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

}
```

4. Insert the **F110 Web 2.0 Find Employee REST Servlet** snippet before the closing brace (Example 16-3).
5. Organize imports (**Ctrl+Shift+O**), then save and close the class.

Example 16-3 REST servlet for find employee

```
public class FindEmployeeServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   @EJB
   EmployeeFacadeInterface employeeFacade;

   protected void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
```

```
      doPost(request, response);
   }

   protected void doPost(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {

      String empId = (String) request.getParameter("empId");

      Employee employee = null;
      try {
         employee = employeeFacade.findEmployeeById(Integer.parseInt(empId));
      } catch(Exception e) {
         e.printStackTrace();
      }

      if (employee == null) {
         response.getWriter().println("Employee not found");
      } else {
         response.getWriter().println(getJSONedEmployee(employee));
      }
   }

   public String getJSONedEmployee(Employee e) {
       JSONObject employeeObj = new JSONObject();
       employeeObj.put("firstName", e.getFirstName());
       employeeObj.put("lastName", e.getLastName());
       employeeObj.put("salary", e.getSalary());
       employeeObj.put("vacationHours", e.getVacationHours());

       JSONObject responseObj = new JSONObject();
       responseObj.put("employee", employeeObj);
       return responseObj.toString();
   }
}
```

Behind the scenes:

The key elements in this servlet are as follows:

- The @EJB annotation provides an instance of the employee facade session bean's business interface (`EmployeeFacadeInterface`). This is the injection technique of EJB 3.0, much simpler than using JNDI:

```
@EJB
EmployeeFacadeInterface employeeFacade;
```

- The application code required is the same for both a doPost and a doGet, so we redirect all doGet requests to the doPost method:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
                   response) throws ServletException, IOException {
   doPost(request, response);
}
```

- The doPost method uses the input parameter to look up the employee record from the EmployeeFacade session facade. It then passes the **employee** object to the **getJSONedEmployee** method.

```
protected void doPost(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {

   String empId = (String) request.getParameter("empId");

   Employee employee = null;
   try {
      employee=employeeFacade.findEmployeeById(Integer.parseInt(empId));
   } catch(Exception e) {
      e.printStackTrace();
   }

   if (employee == null) {
      response.getWriter().println("Employee not found");
   } else {
      response.getWriter().println(getJSONedEmployee(employee));
   }
}
```

- The **getJSONedEmployee** method serializes the contents of the **employee** object by using JSON4J library APIs.

```
public String getJSONedEmployee(Employee e) {
   JSONObject employeeObj = new JSONObject();
    employeeObj.put("firstName", e.getFirstName());
    employeeObj.put("lastName", e.getLastName());
    employeeObj.put("salary", e.getSalary());
    employeeObj.put("vacationHours", e.getVacationHours());

    JSONObject responseObj = new JSONObject();
    responseObj.put("employee", employeeObj);
    return responseObj.toString();
}
```

16.7.2 Test the FindEmployee REST servlet

To test the servlet, we deploy it to the server and run the servlet:

1. Make sure that the **ExperienceJavaEE Server** is started.
2. If the DonateEAR application is already deployed to the server, right-click the **ExperienceJavaEE Server** and select **Publish**.

   If the DonateEAR application is not deployed to the server, right-click the server, select **Add and Remove Projects**, and add the **DonateEAR** to the configured projects. Click **Finish**, and then publish the project as described previously.
3. Open a Web browser and enter the following URL:

   ```
   http://localhost:8080/DonateWeb20/FindEmployeeServlet?empId=2
   ```

   And as expected, the browser displays the serialized details of the `Employee` object in JSON format as follows:

   ```
   {"employee":{"vacationHours":44,"firstName":"Neil","salary":100000,"last
   Name":"Armstrong"}}
   ```

16.7.3 Create the Find Employee view

In this section we create the HTML and JavaScript code to invoke the `FindEmployeeServlet` and display the employee details.

Create the index.html file

First, we create an `index.html` file for the starting page:

1. Right-click **WebContent** in the DonateWeb20 Web project and select **New → HTML**.
2. In the New HTML pop-up, type **index.html** as the file name, and click **Finish**.
3. Replace the contents of the `index.html` file with the contents of the **F111 Web 2.0 Initial Home Page** snippet (Example 16-4).

Example 16-4 Home page index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Web 2.0 Client</title>
<link rel="stylesheet" type="text/css"
      href="dijit/themes/tundra/tundra.css" media="all">
<link rel="stylesheet" type="text/css"
      href="dojo/resources/dojo.css"
```

```
media="all">
<script type="text/javascript" src="dojo/dojo.js"
      djConfig="parseOnLoad: true"></script>
<script type="text/javascript" src="functions.js"></script>
</head>
<body class="tundra">
<form id="form1">
      <input type="text" id="empNumId"
            size="3"
            value="2"
            dojoType="dijit.form.NumberTextBox"
            constraints="{min:1,max:8,places:0}"
            required="true"
       />
      <button dojoType="dijit.form.Button" id="findEmpButton"
            onclick="displayEmployeeDetails">Find</button>

      <div preload="true" dojoType="dijit.layout.ContentPane"
         id="EmployeePane" href="empDetails.html" style="display:none;">
      </div>
</form>
</body>
</html>
```

Behind the scenes:

The key elements in this HTML file are as follows:

- There are two references to two JavaScript files:

  dojo.js: The main script file of dojo client runtime in the `dojo` folder
  functions.js: The JavaScript file that we will use for our client code

- There are three dojo components in the HTML form:

  `dijit.form.NumberTextBox`: Input field for employee number
  `dijit.form.Button`: Push button for find employee
  `djit.layout.ContentPane`: Pane for employee details

- We use the **NumberTextBox** component for entering the employee Id. In this example, we use the ***constraints*** attribute to specify the maximum and minimum values that are valid for employee Id. We have also specified that this field is mandatory by using the ***required*** attribute and the ***value*** attribute can be used to specify the default value.
- The **Button** component invokes the `displayEmployeeDetails` JavaScript method that we provide in a moment.

- The **ContentPane** component can be associated with the div element and in this example is bound to an HTML file, `empDetails.html`, which has markup for displaying the employee details. The ***style*** attribute used in this example with the value of `display:none` signifies that this pane is not visible when the HTML file is rendered.

For more information on dojo components, refer to the Dojo toolkit API reference and book of dojo at:

```
http://api.dojotoolkit.org/
http://dojotoolkit.org/book/dojo-book-1-0
```

Create the empDetails.html file

Next, we create an `empDetails.html` file for the employee details:

1. Right-click **WebContent** in the DonateWeb20 Web project and select **New** → **HTML**. In the New HTML pop-up, type **empDetails.html** as the file name and click **Finish**.
2. Replace the contents of the `empDetails.html` file with the contents of the **F112 Web 2.0 Employee Details Page** snippet, which has markup language for the first name, last name, salary, and vacation of an employee (Example 16-5).

Example 16-5 Employee details page

```
<div style="color: navy; font-family: Arial; font-weight: bold;
             font-size: 10pt">
<table>
   <tr>
      <td>First Name:</td><td width="5px"></td>
      <td><div id="firstName"></div></td>
   </tr>
   <tr>
      <td>Last Name:</td><td width="5px"></td>
      <td><div id="lastName"></div></td>
   </tr>
   <tr>
      <td>Salary:</td><td width="5px"></td>
      <td><div id="salary"></div></td>
   </tr>
   <tr>
      <td>Vacation:</td><td width="5px"></td>
      <td><div id="vac"></div></td>
   </tr>
</table>
</div>
```

Develop the client logic: functions.js

The retrieve of an employee is developed as a JavaScript function:

1. Right-click **WebContent** in the DonateWeb20 Web project and select **New** → **File**.
2. In the New File dialog, type **functions.js** as the file name, and click **Finish**.
3. Replace the contents of the `functions.js` file with the contents of the **F113 Web 2.0 Client JavaScript Retrieve Employee** snippet (Example 16-6).

Example 16-6 Login JavaScript function

```
dojo.require("dojo.parser");
dojo.require("dijit.layout.ContentPane");
dojo.require("dijit.form.NumberTextBox");
dojo.require("dijit.form.Button");

function displayEmployeeDetails() {
   var emp = dijit.byId('empNumId').getValue();

   var deferred = dojo.xhrGet(
      {
         url: "/DonateWeb20/FindEmployeeServlet?empId="+emp,
         handleAs: "json",
         timeout: 5000,
         preventCache: true,
         form: "form1",
         load: function(response, ioArgs) {
               showEmployeeDetails(response.employee);
         },
         error: function(response, ioArgs) {
         }
      }
   );
}

function showEmployeeDetails(employee) {
   dojo.style(dijit.byId("EmployeePane").domNode, "display", "block");
   document.getElementById("firstName").innerHTML=employee.firstName;
   document.getElementById("lastName").innerHTML=employee.lastName;
   document.getElementById("salary").innerHTML=employee.salary;
   document.getElementById("vac").innerHTML=employee.vacationHours;
}
```

> **Note:** Ignore the errors marks in the function. The Eclipse IDE does not recognize the dojo and dijit libraries.

Behind the scenes:

The key elements of the `functions.js` file are as follows:

- **dojo.require** statements, and two JavaScript functions:
  - **`displayEmployeeDetails`**: This function is executed when the `findEmpButton` button is clicked, because its name is specified as the value of the `onclick` event in the `index.html` file.
  - **`showEmployeeDetails`**: This function is executed by the `displayEmployeeDetails` function. The `employee` JSON object is passed to this function and we use the values of this `employee` object to update the **innerHTML** property of `<div>` elements in `empDetails.html`. In this method, we also make the employee details pane visible by changing the value of **display** style property to **block**.
- In the **`displayEmployeeDetails`** function, we first retrieve the value of the entered employee Id and assign it to the JavaScript variable emp:

  ```
  var emp = dijit.byId('empNumId').getValue();
  ```
- Then we use the **`dojo.xhrGet`** function, which is dojo's implementation of the `XMLHttpRequest` call for the `GET` operation. This dojo function sends an HTTP `GET` request to the server.
- In our example, we have used the following properties for the arg parameter of the `dojo.xhrGet` method:
  - **url**: This property is of type String and contains the URL to the server endpoint. In our example, we have specified the URL to the `FindEmployee` REST servlet, and we are also passing the value of the entered employee Id as a URL argument:

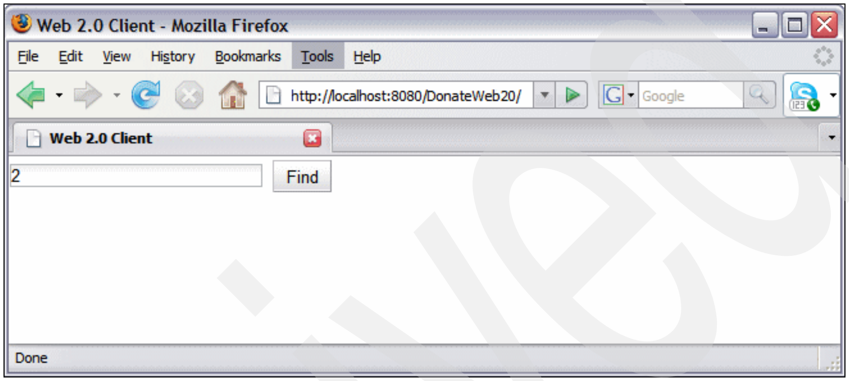
    ```
    url: "/DonateWeb20/FindEmployeeServlet?empId=" +emp
    ```
  - **load**: This property specifies the callback function that is called on a successful HTTP response code. In our example, we have specified to execute the **`showEmployeeDetails`** method.
  - **error**: This property specifies the function that is called when the xhr request fails either due to network or server, or invalid URL. It is also called if the load or handle callback function throws an exception.
  - **handleAs**: This property specifies how the response string should be handled or parsed. The acceptable values are `text` (default), `json`, `json-comment-filtered`, `javascript`, and `xml`.
  - **form**: This property specifies the DOM node for the HTML form. It is used to extract form values and send to the server.

16.7.4 Test the Find Employee Use Case

To test the index.html, we publish the project to the server and run the `index.html` file on the server:

1. Right-click the **ExperienceJavaEE Server** and select **Publish**.
2. Open a Web browser and enter the following URL (Figure 16-16):

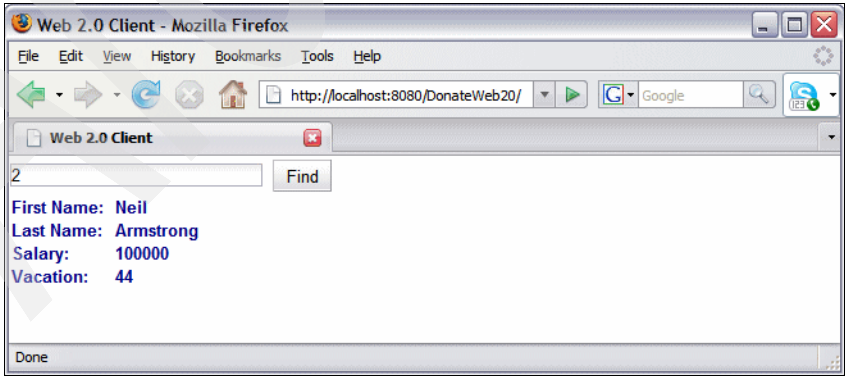
   ```
   http://localhost:8080/DonateWeb20/
   ```


Web 2.0 Client - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://localhost:8080/DonateWeb20/
Google
Web 2.0 Client
Find
Done

2

Figure 16-16 Find Employee use case: Login

3. By default, you will see that **2** is entered in the employee Id, based on the code in `index.html`. You can either click **Find** to submit the client request, or enter another value between 1 and 8.
4. Click **Find** to execute the `dojo.xhrGet` function that submits the employee Id to the server, waits for the JSON response, and then executes the show function to display the employee details content pane (Figure 16-17).

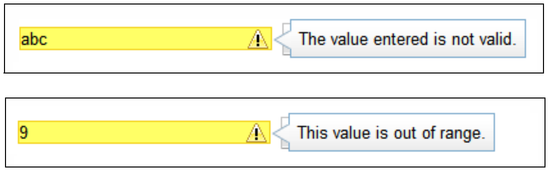

Web 2.0 Client - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://localhost:8080/DonateWeb20/
Google
Web 2.0 Client
Find
Done

2

First Name: Neil
Last Name: Armstrong
Salary: 100000
Vacation: 44

Figure 16-17 Find Employee use case: Employee details

5. Type an invalid value or a value that is not in the range, and verify that the validation is working for the employee Id field (Figure 16-18).



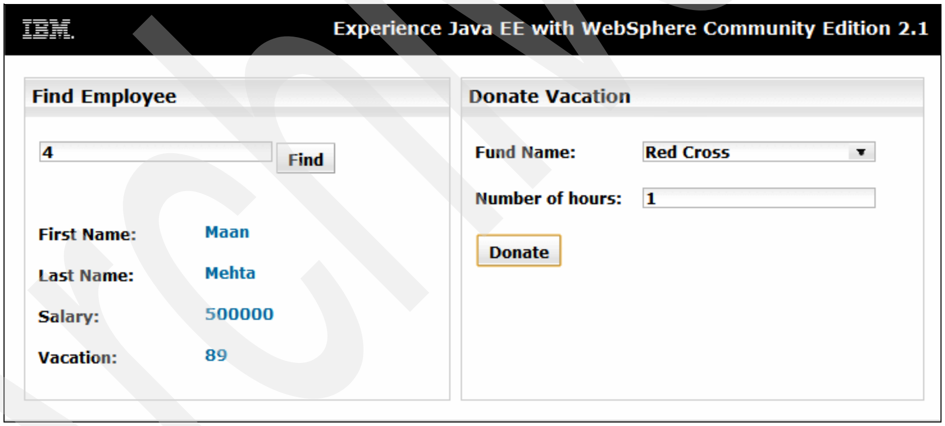
abc ⚠ The value entered is not valid.

9 ⚠ This value is out of range.

Figure 16-18 Find Employee use case: Validation

16.8 Add style to the Web 2.0 application

In this section, we add style definitions to the view. We want to provide a background color to the page and also a header with IBM logo and a banner with the book title and a skin for both the content areas of the two use cases, Find Employe and Donate Vacation. We also want to provide each content area with a title (Figure 16-19).



IBM. Experience Java EE with WebSphere Community Edition 2.1

Find Employee

4 Find

First Name: Maan

Last Name: Mehta

Salary: 500000

Vacation: 89

Donate Vacation

Fund Name: Red Cross ▾

Number of hours: 1

Donate

Figure 16-19 Adding style to the view

16.8.1 Create a cascading style sheet

Style templates are defined in a cascading style sheet that is added to the Web application:

1. Right-click **WebContent** in the DonateWeb20 Web project and select **New** → **Other** → **Web** → **CSS**, and click **Next**.
2. In the Create a new CSS file pop-up, type **index.css** as the file name and click **Finish**.

3. Replace the contents of the newly created `index.css` file with the contents of the **F114 Web 2.0 Style File** snippet (Example 16-7).

Example 16-7 Style file index.css

```
body {
   background: rgb(204, 204, 204) none repeat;
   color: rgb(0, 0, 0); padding: 10px;
}
#pageContent {
   background: rgb(255, 255, 255) none repeat;
   height: auto; width: 760px; margin: auto; margin-bottom: 20px;
   font-family: Verdana,Arial,Helvetica,sans-serif; font-weight: bold;
                                                         font-size: 12px;
}
#header {
   height: 40px; width: 760px;
   background: rgb(0, 0, 0) none repeat;
}
#ibmLogo {
   float: left; margin: 5px; padding: 0px; height: 22px; width: 60px;
   background: url('ibmLogo.gif');
}
.bannerTitle {
   float: right; margin: 10px; padding: 0px; height: 22px; width: auto;
   color: rgb(255, 255, 255); font-size: 14px;
}

/* Style for the two-column Container*/
#container #two-columns {
   height: auto; width: 760px;
   margin: auto; margin-bottom: 20px;
}
#container #two-columns #column-1 {
   float: left; padding: 5px;
   height: auto; width: 340px;
}
#container #two-columns #column-2 {
   float: left; padding: 5px;
   height: auto; width: 340px;
}

/* Clear the above float */
#container .clr { clear: both; }

/* Style for Form field Labels and Values */
.fieldLabel { padding: 0px 0px 0px 10px; float: left; width: 120px; }
.fieldValue { color: rgb(0, 102, 153); }
.fieldInp { padding: 0px 0px 0px 10px; float: left; }
```

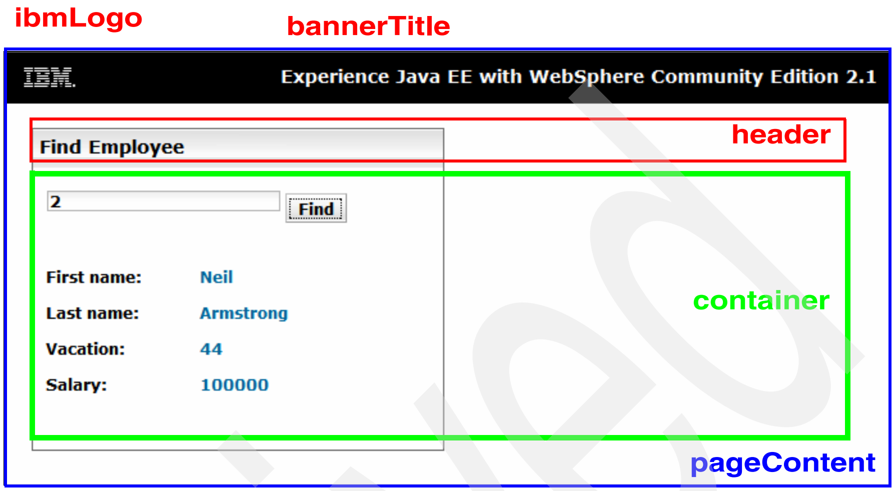
```
/* Style for Portlet Skin */
#portletTitle {
   padding: 5px 5px 0px; border-width: 0px;
   font-size: 14px; color: rgb(0, 0, 0);
   background: rgb(255, 255, 255) url('portletHeader.gif');
   height: 25px;
}
.skinOutline {
   border: 1px solid rgb(104, 104, 104); margin: 10px; padding: 0px;
   width: 320px; height: 250px;
}
```

4. Table 16-3 describes the style definitions we have introduced in the `index.css` style sheet.

Table 16-3 Description of the style sheet

Style	Description
`body`	Gray background for the page.
`pageContent`	White background for the page content area of 760 pixels. Also specifies the default font family size and weight for the content.
`header`	A header area at the top of the page with black background color and height of 40 pixels.
`ibmLogo`	Style that has ibm logo image (`ibmLogo.gif`) as its background in the header.
`bannerTitle`	Style for the title of the book in the Banner that is right-aligned in the header.
`container, column-1, column-2`	Style definitions to provide two columns in the main content area.
`fieldLabel, fieldValue, fieldInp`	Style definitions for form field labels and their values.
`portletTitle and skinOutline`	Style definition to provide portlet look and feel with a skin. The header of the portlet has a background that is an image (`portletHeader.gif`).

5. Figure 16-20 and Figure 16-21 illustrate graphically the significance of each style when applied to the Find Employee use case.



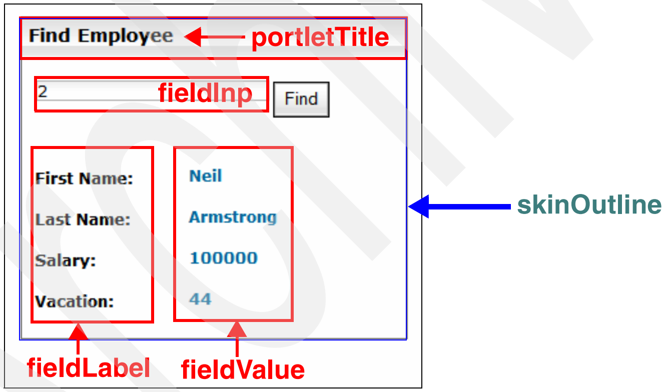
IBM. Experience Java EE with WebSphere Community Edition 2.1

Find Employee

2 Find

First name: Neil
Last name: Armstrong
Vacation: 44
Salary: 100000

Figure 16-20 Style definitions for Find Employee search



Find Employee

2 Find

First Name: Neil
Last Name: Armstrong
Salary: 100000
Vacation: 44

Figure 16-21 Style definitions for Find Employee details

16.8.2 Copy the images

Copy the two images from the `C:\7639code\web2.0` folder to the `WebContent` folder of the DonateWeb20 Web project:

- `ibmLogo.gif`: The IBM logo in the Header
- `portletHeader.gif`: Background gradient image for the Portlet title.

16.8.3 Add style definitions to the HTML files

Replace the contents of the `index.html` file with the contents of the **F115 Web 2.0 Add Style Home Page** snippet (Example 16-8).

Example 16-8 Home page with style (changes in bold)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Web 2.0 Client</title>
<link rel="stylesheet" type="text/css" href="dijit/themes/tundra/tundra.css"
      media="all">
<link rel="stylesheet" type="text/css" href="dojo/resources/dojo.css"
      media="all">
<link rel="stylesheet" type="text/css" href="index.css" media="all">

<script type="text/javascript" src="dojo/dojo.js"
            djConfig="parseOnLoad: true"></script>
<script type="text/javascript" src="functions.js"></script>
</head>
<body class="tundra">
<form id="form1">
  <div id="pageContent">
   <div id="header">
      <div id="ibmLogo"></div>
      <span class="bannerTitle">
         Experience Java EE with WebSphere Community Edition 2.1</span>
   </div>
   <div id="container">
      <div id="two-columns">
         <div id="column-1">
            <div class="skinOutline">
               <div id="portletTitle">Find Employee</div>
               <p>
               <span class="fieldInp">
                  <input type="text" id="empNumId"
                     size="3"
                     value="2"
                     dojoType="dijit.form.NumberTextBox"
                     constraints="{min:1,max:8,places:0}"
                     required="true"
                  />
               </span>
               <button dojoType="dijit.form.Button" id="findEmpButton"
                  onclick="displayEmployeeDetails">Find</button>
               </p>
               <div preload="true" dojoType="dijit.layout.ContentPane"
                  id="EmployeePane" href="empDetails.html"
                  style="display:none;">
            </div>
         </div> <!-- end of column-1 -->
         <div class="clr"></div>
      </div> <!-- end of two-columnns -->
   </div>
  </div>
</form>
</body>
</html>
```

Replace the contents of the empDetails.html file with the contents of the **F116 Web 2.0 Add Style Employee Details** snippet (Example 16-9).

Example 16-9 Employee details page with style

```
<br>
   <label class="fieldLabel">First name:</label>
   <div id="firstName" class="fieldValue"></div>
<br>
   <label class="fieldLabel">Last name:</label>
   <div id="lastName" class="fieldValue"></div>
<br>
   <label class="fieldLabel">Vacation:</label>
   <div id="vac" class="fieldValue"></div>
<br>
   <label class="fieldLabel">Salary:</label>
   <div id="salary" class="fieldValue"></div>
```

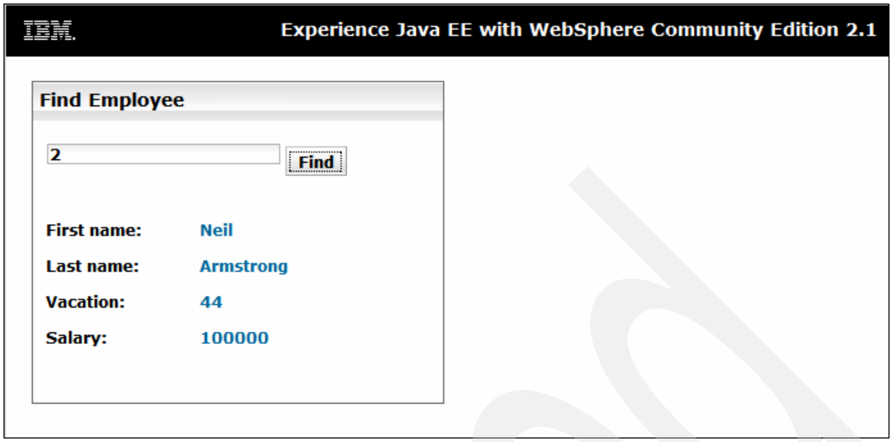
16.8.4 Test the Find Employee use case with style

To test again, we publish the project to the server and run the index.html on the server:

1. Right-click the **ExperienceJavaEE Server** and select **Publish**.
2. Right-click **index.html** and select **Run As** → **Run on Server**. This opens the home page in Eclipse’s internal Web browser
3. Alternatively, you can open an instance of your Web browser and enter the following URL:

   http://localhost:8080/DonateWeb20/

4. The home page opens. Type an employee ID (**2**) and click **Find**. The employee is retrieved and the employee details are displayed (Figure 16-22).



IBM. Experience Java EE with WebSphere Community Edition 2.1

Find Employee

2 Find

First name: Neil

Last name: Armstrong

Vacation: 44

Salary: 100000

Figure 16-22 Run of the Find Employee use case with style definitions

5. Notice that we moved the salary field to the bottom to deal better with null values (for employee 1).

16.9 Implement the Donate Vacation use case

In this section we implement the Donate Vacation use case, where an employee donates vacation hours to a donation fund.

16.9.1 Create the view for vacation donation

Let us create the view for the second use case, Donate Vacation. Insert the contents of the **F117 Web 2.0 Add Style Home Page Second Column** snippet into the `index.html` after the end of the first column (Example 16-10).

Example 16-10 HTML for the content pane for the Donate Vacation use case

```
</div> <!-- end of column-1 -->       <=== INSERT AFTER THIS LINE
<div id="column-2">
   <div class="skinOutline">
      <div id="portletTitle">Donate Vacation</div>
      <div preload="true" dojoType="dijit.layout.ContentPane"
         id="DonatePane" href="donate.html" style="display:none;">
      </div>
   </div>
</div>
<div class="clr"></div>                <=== INSERT BEFORE THIS LINE
```

Behind the scenes:

In this code snippet, we add style definitions to add a second column to index.html. In the second column we add another **dijit.layout.ContentPane** component and bind it to a new **donate.html** file.

As we did for the Find Employee use case, we add markup for the skin to surround the content pane and also provide a title (Donate Vacation). Also notice that we set the value of the style attribute to **display:none**, which means that this content pane is not visible when index.html is rendered initially. We will make it visible when the user clicks **Find** to retrieve details of the employee.

Display the Donate pane in the second column

After an employee is retrieved, the DonatePane in the second column must be activated. In the showEmployeeDetails function in function.js, add one line to show the DonatePane:

```
function showEmployeeDetails(employee) {
   dojo.style(dijit.byId("EmployeePane").domNode, "display", "block");
   dojo.style(dijit.byId("DonatePane").domNode, "display", "block");
   document.getElementById(......
   ......
```

Create the donate.html file

Next, we create the donate.html file for the Donate Vacation use case:

1. Right-click **WebContent** in the DonateWeb20 Web project and select **New** → **HTML**. In the New HTML pop-up, type **donate.html** as the file name, and click **Finish**.
2. Replace the contents of the new donate.html file with the contents of the **F118 Web 2.0 Donate Vacation Page** snippet, which has markup for selecting the fund name, the amount of vacation to donate to that fund, and the Donate button (Example 16-11).

Example 16-11 Donate Vacation use case HTML

```
<div dojoType="dojo.data.ItemFileReadStore" jsId="fundsStore"
                                                  url="funds.json"></div>
<p/>
   <label class="fieldLabel">Fund name:</label>
   <input id="fundNameField" dojoType="dijit.form.FilteringSelect"
      store="fundsStore"
      searchAttr="name"
      name="fundName"
```

```
         autocomplete="true"
         invalidMessage= "Please select a valid donation Fund"
      />
   <p/>
      <label class="fieldLabel">Number of hours:</label>
      <input type="text" id="vacField"
         name="hours" size="3"
         value="1"
         dojoType="dijit.form.NumberTextBox"
         promptMessage="Enter number of hours of vacation to donate"
         constraints="{min:1,max:90,places:0}"
         invalidMessage= "Please enter a valid number of hours"
         required="true"
      />
   <p/>
      <span class="fieldInp">
         <button dojoType="dijit.form.Button" id="donateButton"
                                                    onclick="donate">Donate</button>
      </span>
```

Behind the scenes:

In this code snippet, we add markup for two input form fields where the user can select a fund and the number of their vacation hours to donate to that fund. There is also a markup for a Donate button for the user to click to submit the donation request.

The key elements in this HTML file are as follows:

- **ItemFileReadStore**: In this code snippet, we use a data store called `ItemFileReadStore`, that comes pre-packaged with Dojo:

  ```
  <div dojoType="dojo.data.ItemFileReadStore" jsId="fundsStore"
                                              url="funds.json"></div>
  ```

  And as the name suggests, the `ItemFileReadStore` is a read-only data store that reads the contents from an HTTP endpoint. This data store expects the contents of this source to be in JSON format, which it reads and deserializes into an in-memory JavaScript object. The HTTP endpoint is specified as the value of the **`url`** attribute. This URL can point either to a static file with JSON content or to a dynamic service hosted at the server. This service can either be a servlet that writes text in JSON format to the output stream or a JAX-WS based REST Web Service.

  In this case, the contents of the `funds.json` file are read and deserialized into an in-memory JavaScript object that is specified as the value of the `jsId` attribute `fundsStore`.

- **FilteringSelect**: In this code snippet, we use a `dijit.form` component that is just like an HTML select or drop-down form field, but is populated dynamically using the contents of the in-memory `JavaScript` object that is specified as the value of the `store` attribute:

  ```
  <input id="fundNameField" dojoType="dijit.form.FilteringSelect"
        store="fundsStore"
        searchAttr="name"
        name="fundName"
        autocomplete="true"
        invalidMessage= "Please select a valid donation Fund"
  />
  ```

  In this code snippet, we use the contents of the `fundsStore` object that was created by the `ItemFileReadStore` by reading the contents of the `funds.json` file. Or in other words, we are pre-populating the contents of the fund name drop-down field with the contents of a static JSON file.

- We are using the **NumberTextBox** component for specifying the number of hours to donate. We use the `constraints` attribute to specify the maximum and minimum values that are valid. We also specify that this field is mandatory by using the `required` attribute, and the `value` attribute can be used to specify the default value.

- The **Button** component in this example results in the invocation of the `donate` JavaScript method that we will implement in a moment.

Create the funds.json file

Next, we create the static `funds.json` file that holds the names of the donation funds:

1. Right-click **WebContent** in the DonateWeb20 Web project and select **New** → **File**. In the New File pop-up, type **funds.json** as the file name and click **Finish**.

2. Replace the contents of the new `funds.json` file with the contents of the **F119 Web 2.0 Static Fund Names JSON file** snippet, which has all the fund names in JSON format (Example 16-12).

Example 16-12 Contents of the static JSON file that contains fund names

```
{identifier:"name",
items: [
   {name:"DonationFund"},
   {name:"Red Cross"},
   {name:"United Way"},
   {name:"Heart and Stroke Foundation"} <==== does not exist in database
]}
```

Behind the scenes:

- The identifier attribute instructs the data store indicating which field contains the value to be submitted, in our case the name field. In this JSON snippet for funds, the items object only has one attribute, but it could have multiple attributes.

  For example, in the following JSON snippet for US states, we can use one attribute (label) to display the name and another attribute (abbreviation) to submit the value (as set by the identifier):

  ```
  {identifier:"abbreviation",
  items: [
          {name:"Alabama",label:"Alabama",abbreviation:"AL"},
          ...
  ]}
  ```

- Notice that one fund name does not match the fund names stored in the Donate database. This allows for testing of an invalid name.

At a later stage, in 16.9.7, “Retrieve the fund names from the database” on page 507, we replace the static funds.json file with a dynamic JSON object that is retrieved from the Donate database through a servlet.

16.9.2 Implement the client logic for the Donate Vacation use case

The client logic that is executed when the **Donate** button is clicked is added to the **function.js** JavaScript file in the donate function:

1. Because we introduced two new dojo components, ItemFileReadStore and FilteringSelect, we have to add require statements to the JavaScript file. To display success and error messages in a dialog box, we also require the Dialog component.

   Add three dojo.require statement to function.js:

   ```
   dojo.require("dojo.data.ItemFileReadStore");
   dojo.require("dijit.form.FilteringSelect");
   dojo.require("dijit.Dialog");
   ```

2. Insert the contents of the **F120 Web 2.0 Implement JavaScript Client Logic** snippet at the end of the functions.js file (Example 16-13).

Example 16-13 JavaScript function for Donate Vacation use case

```
function donate() {
   dijit.byId("donateButton").disabled=true;
   var emp = dijit.byId('empNumId').getValue();
   var fund = dijit.byId('fundNameField').getValue();
```

```
   var hours = dijit.byId('vacField').getValue();

   var deferred = dojo.xhrGet(
      {
         url: "/DonateWeb20/DonateServlet?employeeId="+emp
                                 +"&fundName="+fund+"&hours="+hours,
         handleAs: "json",
         timeout: 5000,
         preventCache: true,
         form: "form1",
         load: function(response, ioArgs) {
            showMessageDialog(response);
            dijit.byId("donateButton").disabled=false;
            return response;
         },
         error: function(response, ioArgs) {
               document.getElementById('errMsg').innerHTML =
                                       "Error during xhrPost for Donate";
            dijit.byId('errDialog').show();
            return response;
         }
      }
   );
}
```

Behind the scenes:

The key elements of the `donate` function are as follows:

- Disable the Donate button to avoid double-clicking:

  ```
  dijit.byId("donateButton").disabled=true;
  ```

  We reactivate this button after the request processing is complete.

- Get the values that the user entered or selected in the employee number, fund name, and number of hours form fields:

  ```
  var emp = dijit.byId('empNumId').getValue();
  var fund = dijit.byId('fundNameField').getValue();
  var hours = dijit.byId('vacField').getValue();
  ```

- Execute a REST servlet, `DonateServlet`, by sending an HTTP GET request to the server using the `dojo.xhrGet` API and submitting the values of employee number, fund name, and the number of hours as arguments:

```
var deferred = dojo.xhrGet(
{
   url: "/DonateWeb20/DonateServlet?employeeId="+emp
                              +"&fundName="+fund+"&hours="+hours,
   handleAs: "json",
   ......
   load: function(response, ioArgs) {
      ......
   },
   error: function(response, ioArgs) {
      ......
   }
}
);
```

- In the callback function specified as the value of the **`load`** attribute of the `dojo.xhrGet` API, we pass the result string we get from the `DonateServlet` (in JSON format) to a `showMessageDialog` function that we create in a moment. We also enable the Donate button that we disabled earlier.

```
load: function(response, ioArgs) {
   showMessageDialog(response);
   dijit.byId("donateButton").disabled=false;
   return response;
},
```

- We have also specified an **`error`** call back function, which is called if there is an error executing the `dojo.xhrGet` API. In this function, we set an error message string as the value of the `innerHTML` property of the `errMsg` element. We also call the `show` method of the `errDialog` dijit component. This action displays the specified error message using dojo's message dialog component. We will specify the markup of the `errMsg` and `errDialog` elements in a moment.

```
error: function(response, ioArgs) {
   document.getElementById('errMsg').innerHTML =
                              "Error during xhrPost for Donate";
   dijit.byId('errDialog').show();
   return response;
}
```

16.9.3 Display messages in Ajax modal dialogs

We have a requirement to display the success and error messages in Ajax modal dialogs. In this section we add the markup to display the Ajax modal dialogs using the `dijit.Dialog` component:

1. Insert the contents of the **F121 Web 2.0 Markup for Ajax Modal Dialog Messages** snippet into the `index.html` after the end of second column (Example 16-14).

   Example 16-14 Using dijit.Dialog to show status messages

   ```
   </div> <!-- end of two-columnns -->       <=== INSERT AFTER THIS LINE

   <!-- Using dijit dialog to show status messages-->
   <div dojoType="dijit.Dialog" id="msgDialog" title="Thank you!"
         style="font:11px Arial; display:none; color:navy; font-weight:bold;">
      <div id="msg" style="font:11px Arial; color:red; padding:10px;"></div>
   </div>

   <div dojoType="dijit.Dialog" id="errDialog" title="Error Message"
         style="font:11px Arial; display:none; color:navy; font-weight:bold;">
      <div id="errMsg" style="font:11px Arial; color:red; padding:10px;">
      </div>
   </div>
   ```

   Behind the scenes:

   Note that the `id` attributes for displaying the error message match the ids used in the `function.js` JavaScript:

   ```
   error: function(response, ioArgs) {
      document.getElementById('errMsg').innerHTML = "......";
      dijit.byId('errDialog').show();
      return response;
   }
   ```

2. To display the success message is an Ajax dialog box, the `load` callback function invokes the `showMessage` function:

   ```
   load: function(response, ioArgs) {
      showMessageDialog(response);
      ......
   ```

3. In a first simple implementation, we only display the message returned from the server. Insert the **F122 Web 2.0 Show Message Dialog Simple** snippet at the end of the `function.js` file (Example 16-15).

Example 16-15 Display the transfer message from the servlet

```
function showMessageDialog(result){
   dijit.byId("findEmpButton").disabled=true;
   document.getElementById("msg").innerHTML = result.returnMessage;
   dijit.byId("msgDialog").show();
   dijit.byId("findEmpButton").disabled=false;
}
```

This code disables the Find button, sets the `msg` attribute from the servlet response, displays the message dialog (`msgDialog` matches the `id` in `index.html`), and enables the Find button when the dialog is closed.

16.9.4 Create a REST servlet for the Donate Vacation use case

In this section, we create a REST servlet that submits the donation request to the `DonateBean` session EJB through simple EJB injection and access. This servlet then serializes the return message in JSON format and writes that string to the HTTP response output stream. This servlet is called through the `dojo.xhrGet` API in the `donate` function specified in the `functions.js` file:

1. In the Project Explorer, right-click **DonateWeb20** project, and select **New** → **Servlet**.
2. In the Create Servlet pop-up, provide the following data:
   - Java package: **donate.web.rest.servlet**
   - Class name: **DonateServlet**
   - Click **Finish**.
3. In the `DonateServlet` editor, delete all the generated method skeletons (default constructor, `doGet`, and `doPost`), leaving the base class definition and the `serialVersionUID`.

   ```
   public class DonateServlet extends HttpServlet {
      private static final long serialVersionUID = 1L;

   }
   ```
4. Insert the **F123 Web 2.0 Servlet for Donate Vacation Use Case** snippet before the closing brace (Example 16-16).
5. Organize imports (**Ctrl+Shift+O**).
6. Save and close `DonateServlet.java`.

Example 16-16 Donate Vacation use case REST servlet

```
public class DonateServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;
```

```
   @EJB
   DonateBeanRemote donate;

   protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
         throws ServletException, IOException {
      doPost(request, response);
   }

   protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
         throws ServletException, IOException {
      int employeeId =Integer.parseInt(request.getParameter("employeeId"));
      String fundName = request.getParameter("fundName");
      int hours = Integer.parseInt(request.getParameter("hours"));
      String result = donate.donateToFund(employeeId, fundName, hours);
      JSONObject resultObj = new JSONObject();
      resultObj.put("returnMessage", result);
      response.getWriter().println(resultObj.toString());
   }
}
```

Behind the scenes:

The DonateServlet is similar to the FindEmployeeServlet. In the doPost method, the parameters are retrieved from the donate.html fields, the donateToFund method of the session EJB is invoked, and the result is returned as a JSON object with one attribute (returnMessage).

Test the DonateServlet

To test the servlet, we publish the application to the server and run the servlet:

1. Publish the updated DonateEAR application on the server.
2. Open a Web browser and enter the following URL:

   ```
   http://localhost:8080/DonateWeb20/DonateServlet?employeeId=2&fundName=Red%20Cross&hours=10
   ```

3. And as expected, the browser displays the serialized details of the success message in JSON format as follows:

   ```
   {"returnMessage":"Transfer successful"}
   ```

4. Run the servlet with an invalid fund name and the result is an error message:

   ```
   http://localhost:8080/DonateWeb20/DonateServlet?employeeId=2&fundName=BadFund&hours=2

   {"returnMessage":"Error: Fund not found with name \"BadFund\""}
   ```

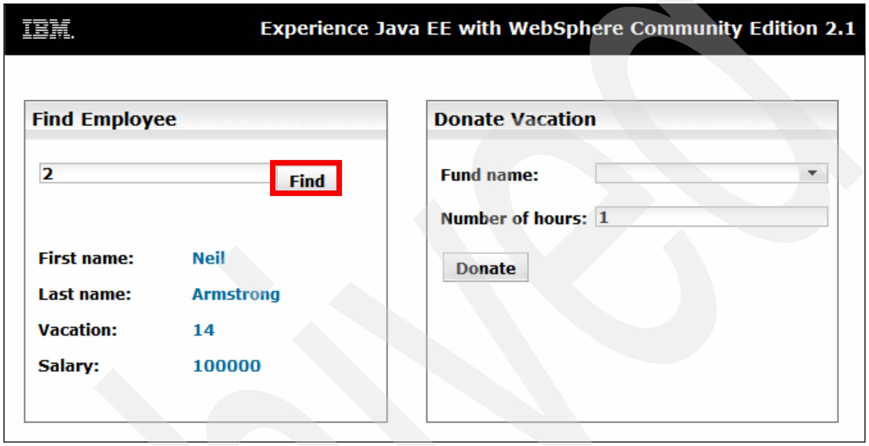
16.9.5 Test the Donate Vacation use case

Now we can test the Donate Vacation use case:

1. After the application is published, open a browser with the URL:

   `http://localhost:8080/DonateWeb20/`

2. Retrieve the employee by clicking **Find**. The employee details and the second column are displayed (Figure 16-23).



Experience Java EE with WebSphere Community Edition 2.1

Find Employee

2

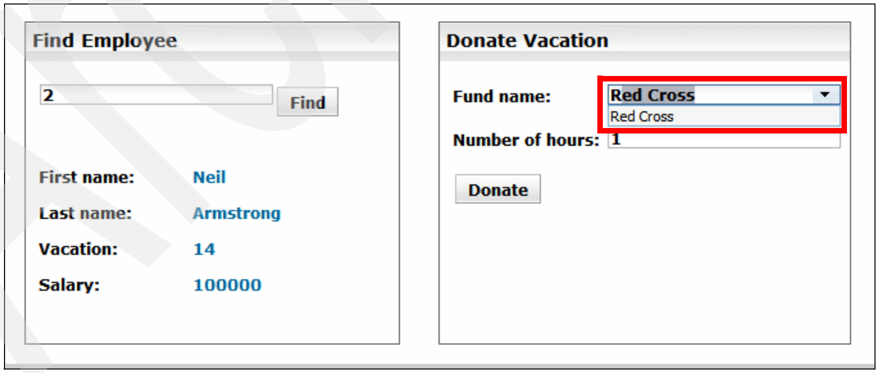
First name: Neil
Last name: Armstrong
Vacation: 14
Salary: 100000

Donate Vacation

Fund name:
Number of hours: 1

Figure 16-23 Testing the Donate Vacation use case: Find Employee

3. Experiment with the type-ahead function by typing **r** into the Fund name field. The field is filled with the correct fund name (Figure 16-24).



Find Employee

2

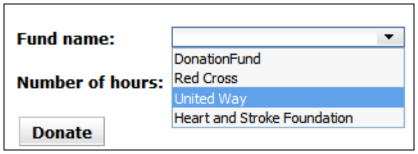
First name: Neil
Last name: Armstrong
Vacation: 14
Salary: 100000

Donate Vacation

Fund name: Red Cross
Red Cross
Number of hours: 1

Figure 16-24 Testing the Donate Vacation use case: Type-ahead

4. Alternatively use the pull-down menu to select a fund (Figure 16-25).

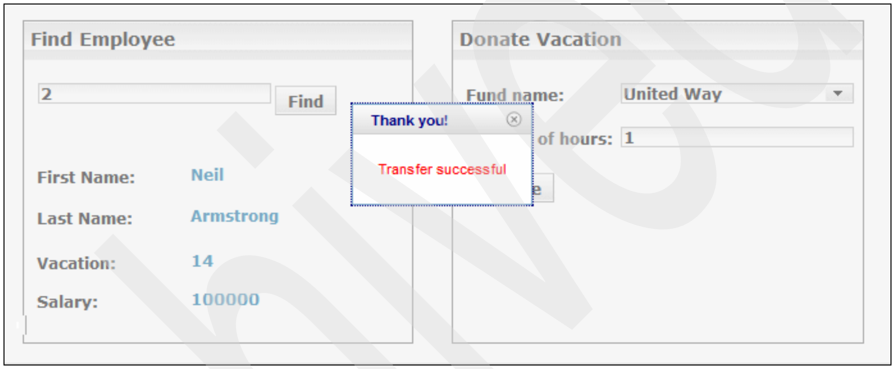


Fund name:

Number of hours:

Figure 16-25 Testing the Donate Vacation use case: Pull-down menu

5. Select a fund, type a number of hours to donate, and click **Donate** to transfer the vacation hours to a fund. The dialog box with the successful message is displayed (Figure 16-26).



Find Employee

2

First Name: Neil

Last Name: Armstrong

Vacation: 14

Salary: 100000

Donate Vacation

Fund name: United Way

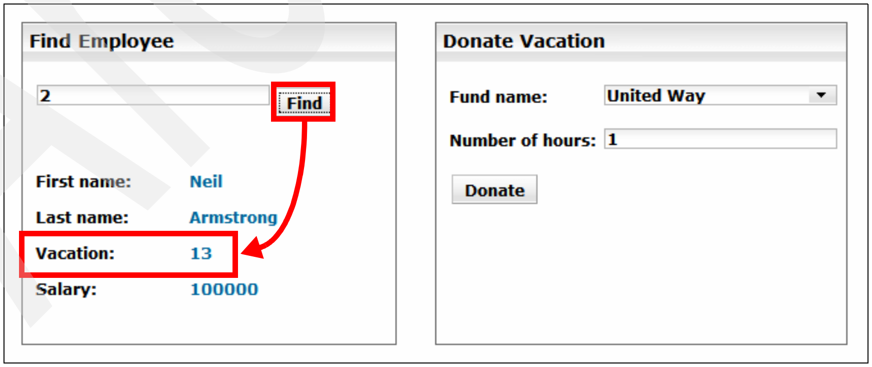
of hours: 1

Thank you!

Transfer successful

Figure 16-26 Testing the Donate Vacation use case: Transfer successful

6. Close the message dialog. Click **Find** to verify that the number of vacation hours has been decreased for the employee (Figure 16-27).



Find Employee

2

First name: Neil

Last name: Armstrong

Vacation: 13

Salary: 100000

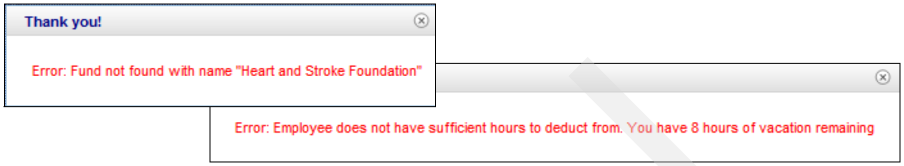
Donate Vacation

Fund name: United Way

Number of hours: 1

Figure 16-27 Testing the Donate Vacation use case: Verify vacation hours

7. If you donate hours to the Heart and Stroke Foundation fund, which does not exist in the database, or if you try to donate more hours than are available, you get an error message box (Figure 16-28).



Error: Fund not found with name "Heart and Stroke Foundation"

Error: Employee does not have sufficient hours to deduct from. You have 8 hours of vacation remaining

Figure 16-28 Donating hours to a non-existing fund or donating too many hours

16.9.6 Update the vacation hours automatically

We can improve the application by updating the vacation hours of the employee in the employee details, and to improve the transfer successful message with the remaining hours:

1. In the `function.js` file, rename the `showMessageDiaog` function to `showMessageDialog1`:

```
function showMessageDialog1(result){
   ......
```

2. Add an improved version of the `showMessage` function to `function.js` by inserting the **F124 Web 2.0 Show Message Dialog Hours** snippet at the end of the file (Example 16-17).

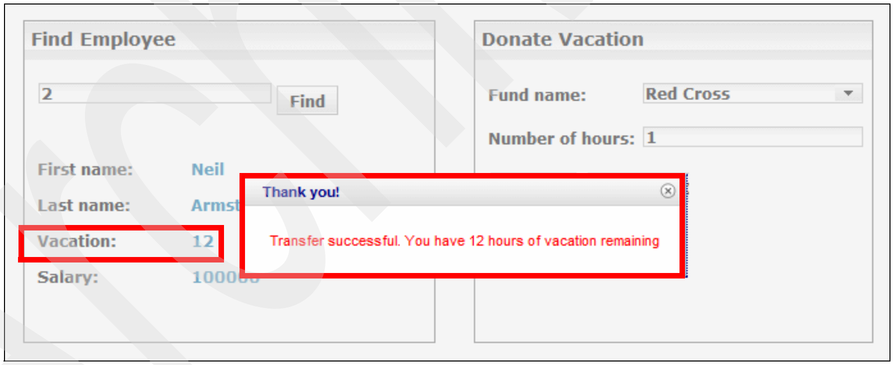
Example 16-17 Improved dialog box with remaining vacation hours

```
function showMessageDialog(result){
   dijit.byId("findEmpButton").disabled=true;
   var emp = dijit.byId('empNumId').getValue();
   var empObj;

   var deferred = dojo.xhrGet(
      {
         url: "/DonateWeb20/FindEmployeeServlet?empId="+emp,
         handleAs: "json",
         timeout: 5000,
         preventCache: true,
         form: "form1",
         load: function(response, ioArgs) {
            empObj = response.employee;
            document.getElementById("vac").innerHTML =
                                          empObj.vacationHours;
            document.getElementById("msg").innerHTML = result.returnMessage
                       + ".  You have " + empObj.vacationHours
                       + " hours of vacation remaining";
```

```
        dijit.byId("msgDialog").show();
        dijit.byId("findEmpButton").disabled=false;
      },
      error: function(response, ioArgs) {
        console.debug
          ("**** dojo.xhrGet for employee details NOT successful. ");
      }
    }
  );
}
```

> **Behind the scenes:**
>
> In the improved implementation we invoked the `FindEmployeeServlet` to retrieve the updated employee information as a JSON object (`empObj`). We store the remaining vacation hours (`empObj.vacationHours`) in the `vac` field of the employee details pane, and we add the remaining hours to the result message (`msg`).

3. Publish the application and rerun the Donate Vacation use case. The vacation hours are updated in the employee details pane and the extended message is displayed (Figure 16-29).



Find Employee

2

Find

First name: Neil

Last name: Armst

Vacation: 12

Salary: 100000

Donate Vacation

Fund name: Red Cross

Number of hours: 1

Thank you!

Transfer successful. You have 12 hours of vacation remaining

Figure 16-29 Testing the Donate Vacation use case: Remaining hours

16.9.7 Retrieve the fund names from the database

Instead of using the static file `funds.json`, we want to populate the drop-down box with the fund names from the database. We create a `GetAllFunds` servlet to retrieve the names in JSON format, and change the `donate.html` file to invoke the servlet to populate the names:

1. Open the **donate.html** file and change the first line with the ItemFileReadStore to invoke the GetAllFunds servlet:

   ```
   <div dojoType="dojo.data.ItemFileReadStore" jsId="fundsStore"
                                   url="GetAllFunds"></div>
   ```

2. Right-click the **donate.web.rest.servlet** package, and select **New** → **Servlet**.
3. Type **GetAllFunds** as Class name and click **Finish**.
4. Replace the generated methods with the **F125 Get All Funds Servlet** snippet (Example 16-18).

Example 16-18 Servlet to retrieve the fund names as a JSON object

```
public class GetAllFunds extends HttpServlet {
   private static final long serialVersionUID = 1L;

    @EJB
    FundFacadeInterface fundFacade;

   protected void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
      doPost(request, response);
   }

   protected void doPost(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
      List<Fund> funds = null;
      try {
         funds = fundFacade.getAllFunds();
      } catch(Exception e) {
         e.printStackTrace();
      }

      if (funds == null || funds.isEmpty()) {
         response.getWriter().println("Funds not found");
      } else {
         response.getWriter().println(getJSONedFunds(funds));
      }
   }

   public String getJSONedFunds(List<Fund> funds) {
      StringBuffer resp = new StringBuffer
                                 ("{identifier: \"name\", items: [");
      for (int i=0; i<funds.size(); i++) {
         Fund f = funds.get(i);
         JSONObject fundObj = new JSONObject();
         fundObj.put("name", f.getName());
         String coma="";
         if (i < funds.size()-1)
```

```
             coma=",";
         resp.append(fundObj.toString());
         resp.append(coma);
      }
      resp.append("]}");

      //System.out.println("GetAllFunds servlet: " + resp.toString() );
      return resp.toString();
   }
}
```

Behind the scenes:

The `GetAllFunds` servlet is similar to the `FindEmployeeServlet`. It uses the `FundFacadeInterface` of the session bean to retrieve the fund names, and invokes the `getJSONedFunds` method to create the JSON object to return.

5. Optionally publish the application and test the servlet using the URL:

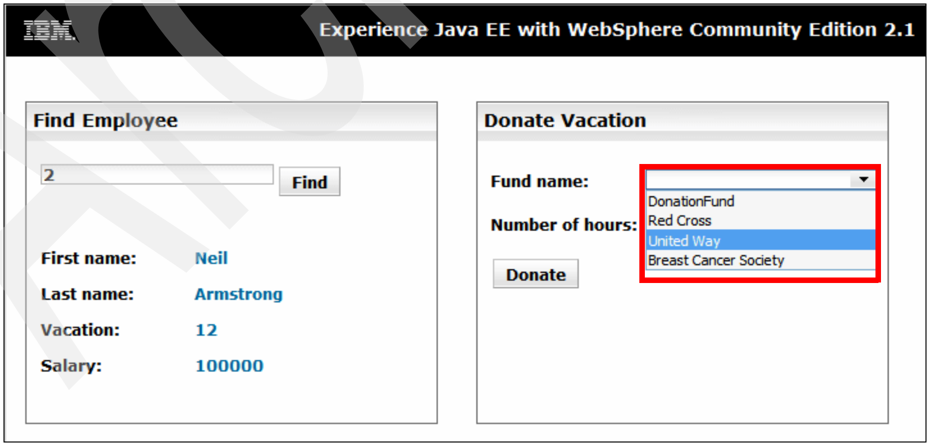
   `http://localhost:8080/DonateWeb20/GetAllFunds`

6. The servlet returns the string:

```
{identifier: "name", items: [{"name":"DonationFund"},{"name":"Red
Cross"},{"name":"United Way"},{"name":"Breast Cancer Society"}]}
```

Test the application with the database fund names

Publish the application and run it. The drop-down menu with the fund names now lists the funds that are in the database (Figure 16-30).


Experience Java EE with WebSphere Community Edition 2.1

Find Employee

2 Find

First name: Neil

Last name: Armstrong

Vacation: 12

Salary: 100000

Donate Vacation

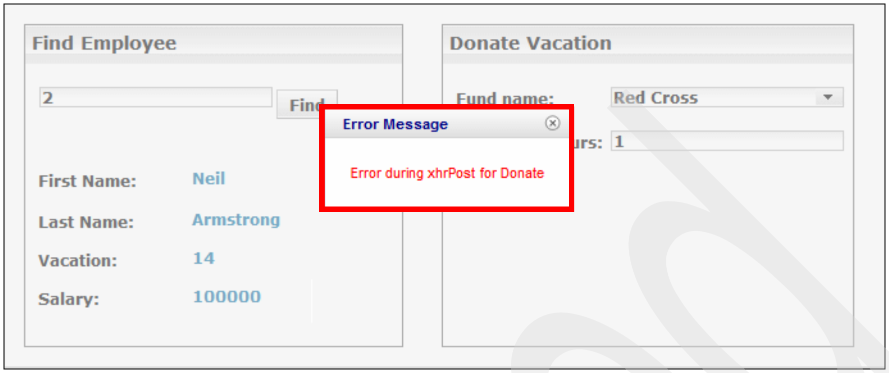
Fund name:

Number of hours:

DonationFund
Red Cross
United Way
Breast Cancer Society

Figure 16-30 Testing the application with fund names from the database

If the donation of hours to a fund fails, the error dialog is displayed (Figure 16-31).



2

First Name: Neil

Last Name: Armstrong

Vacation: 14

Salary: 100000

Fund name: Red Cross

urs: 1

Error during xhrPost for Donate

Figure 16-31 Application error dialog

16.9.8 Create a JAX-WS RESTful Web service

So far, we have used simple Java EE servlets to provide the REST service that is called by the client side code. An alternative is to create a RESTful Web service based on the JAX-WS 2.0. In this section, we show how to create such a Web service for the Donate Vacation use case. Instead of invoking the `DonateServlet`, we invoke the JAX-WS Web service to donate the specified vacation hours:

1. Create the service class for the Web service by right-clicking **DonateWeb20** and selecting **New** → **Class**.
2. In the New Java Class dialog, type **donate.web.rest.ws** as package name, **DonateRestWS** as class name, and click **Finish**.
3. Replace the generated code with the code of the **F126 Web 2.0 REST Web Service Class** snippet (Example 16-19). Press **Ctrl+Shift+O** to resolve the imports, save, and close the class.

Example 16-19 RESTful Web service class DonateRestWS

```
import javax.xml.ws.Provider
import javax.xml.transform.Source
import javax.xml.ws.handler.MessageContext
import com.ibm.json.java.JSONObject

@WebServiceProvider
@BindingType(value = HTTPBinding.HTTP_BINDING)
@ServiceMode(value = javax.xml.ws.Service.Mode.PAYLOAD)
public class DonateRestWS implements Provider<Source> {
   @Resource
   protected WebServiceContext wsCtx;

   @EJB
   DonateBeanRemote donate;

   public Source invoke(Source source) {
      MessageContext mCtx = wsCtx.getMessageContext();
      String method = (String) mCtx
                          .get(MessageContext.HTTP_REQUEST_METHOD);
      if (method.equals("POST"))
         return post(source, mCtx);
      else if (method.equals("GET")) {
         String getResponse =
                "<msg>This is the Donate RESTful Web services</msg>";
         return new StreamSource(new StringReader(getResponse));
      } else
         throw new WebServiceException("Unsupported method: " + method);
   }

   private Source post(Source src, MessageContext mCtx) {
      JSONObject response = new JSONObject();
      JSONObject result = new JSONObject();
      try {
         ServletRequest req = (ServletRequest) mCtx
                                .get(MessageContext.SERVLET_REQUEST);
         int employeeId = new Integer(req.getParameter("employeeId"));
         String fundName = req.getParameter("fundName");
         int hours = Integer.parseInt(req.getParameter("hours"));

         String donateResult = donate.donateToFund
                                       (employeeId, fundName, hours);
         result.put("returnMessage", donateResult);
      } catch (Exception e) {
         System.out.println("JAXWS exception thrown: " + e.getMessage());
         e.printStackTrace();
         result.put("returnMessage", "Error - An exception was thrown\n\n"
                                                   + e.getMessage());
      }
      response.put("result", result);
      String res = "<result><json>" + response + "</json></result>";
      StreamSource reply = new StreamSource
                                (new StringReader(res.toString()));
      return reply;
   }
}
```

Behind the scenes:

- JAX-WS defines that the `javax.xml.ws.Provider` interface must be implemented by a RESTful Web service endpoint. This interface defines the `invoke` method with the signature:

  ```
  T invoke(T request)
  ```

- The **@WebServiceProvider** annotation defines the class as a provider to the container. The implementation replaces T with a class that defines the type messages consumed or generated by the Web service. We used the `javax.xml.transform.Source` class that allows to consume and generate XML messages.
- The **@ServiceMode** annotation configures the messaging mode of the Provider. When its value is set to `MESSAGE`, the provider receives and returns the entire protocol messages. When it is set to `PAYLOAD`, the provider only has access to the messages payload, for example, the content of a SOAP body.
- The **@Resource** annotation injects the Web service context.
- The **@EJB** annotation injects the remote interface of the session bean.
- The **invoke** method verifies the HTTP method as `POST`, and invokes the `post` method.
- The **post** method retrieves the parameters (employee Id, fund name, hours to donate), and invokes the `donateToFund` method of the session bean. An XML response is returned in the format:

  ```
  <result><json> JSON-response-object </json></result>
  ```

4. Add a servlet mapping for the Web service class to the `web.xml` Web deployment descriptor before the closing `</web-app>` tag (Example 16-20).

Example 16-20 Servlet mapping for the Web service

```
  ......
  <servlet>
    <display-name>DonateRestWS</display-name>
    <servlet-name>DonateRestWS</servlet-name>
    <servlet-class>donate.web.rest.ws.DonateRestWS</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>DonateRestWS</servlet-name>
    <url-pattern>/DonateRestWS</url-pattern>
  </servlet-mapping>
</web-app>
```

Update the client logic to invoke the RESTful Web service

We have to change the client logic to invoke the Web service instead of the servlet. We could update the existing `donate` JavaScript function, but we would rather code a new **donateJAXWS** function that we invoke from a new **Donate JAXWS** button:

1. Open the **donate.html** file, and add the **Donate JAXWS** button:

```
......
<span class="fieldInp">
   <button dojoType="dijit.form.Button" id="donateButton"
                          onclick="donate">Donate</button>
   <button dojoType="dijit.form.Button" id="jaxwsButton"
                          onclick="donateJAXWS">Donate JAXWS</button>
</span>
```

2. Open the **function.js** file and add two methods at the end of the file by inserting the **F127 Web 2.0 JavaScript for REST Web Service** snippet (Example 16-21).

Example 16-21 Client JavaScript to invoke the RESTful Web service

```
function donateJAXWS() {
   console.debug("**** Entered donateJAXWS function");
   dijit.byId("jaxwsButton").disabled=true;
   var emp = dijit.byId('empNumId').getValue();
   var fund = dijit.byId('fundNameField').getValue();
   var hours = dijit.byId('vacField').getValue();

   var deferred = dojo.rawXhrPost(
      {
         url: "/DonateWeb20/DonateRestWS",
         timeout: 5000,
         handleAs : "xml",
         headers: { "Content-Type":"application/x-www-form-urlencoded" },
         form: "form1",
         postData: "employeeId="+emp+"&fundName="+fund+"&hours="+hours
      }
   );

   deferred.addCallback(
         function(response){
            var jsonObj = getJsonObjectFromXML(response).result;
            showMessageDialog(jsonObj);
            dijit.byId("jaxwsButton").disabled=false;
            return response;
         }
   );

   deferred.addErrback(
          function(response){
             document.getElementById('errMsg').innerHTML =
                             "Error during rawXhrPost for Donate";
             dijit.byId('errDialog').show();
             dijit.byId("jaxwsButton").disabled=false;
             return response;
          }
   )
}

function getJsonObjectFromXML(response) {
   var resultElement = response.getElementsByTagName("result");
   var jsonElement = resultElement[0].childNodes[0];
   var jsStr = jsonElement.childNodes[0].nodeValue;
   return dojo.fromJson(jsStr,true);
}
```

Behind the scenes:

The key elements of the **donateJAXWS** function are as follows:

- Disable the Donate JAXWS button to avoid double-clicking. We make this button active after the request processing has been completed.
- Get the values that the user entered or selected:

```
var emp = dijit.byId('empNumId').getValue();
var fund = dijit.byId('fundNameField').getValue();
var hours = dijit.byId('vacField').getValue();
```

- Execute the Web service by sending an HTTP POST request to the server using the dojo.rawXhrPost API and submitting the values of employee number, fund name, and number of hours as keyword-value pairs in the postData attribute.

```
var deferred = dojo.rawXhrPost(
   {
      url: "/DonateWeb20/DonateRestWS",
      timeout: 5000,
      handleAs : "xml",
      headers: { "Content-Type":
                       "application/x-www-form-urlencoded" },
      form: "form1",
      postData: "employeeId=" + emp + "&fundName=" + fund
                   + "&hours=" + hours
   }
);
```

- Specify the callback function that is executed upon a successful return of the request. We get the JSON representation of the result and pass that JSON object to the existing `showMessageDialog` function. We also enable the button that we disabled before.

```
deferred.addCallback(
        function(response){
            var jsonObj = getJsonObjectFromXML(response).result;
            showMessageDialog(jsonObj);
            dijit.byId("jaxwsButton").disabled=false;
            return response;
        }
);
```

- Specify the callback function that is executed when there is an error executing the request. In this function, we set the `innerHTML` value of the `errMsg` element to an error message and call the `show` method of the `errDialog` component.

```
deferred.addErrback(
        function(response){
            document.getElementById('errMsg').innerHTML =
                                "Error during rawXhrPost for Donate";
            dijit.byId('errDialog').show();
            dijit.byId("jaxwsButton").disabled=false;
            return response;
        }
)
```

- The **getJsonObjectFromXML** function retrieves the `result` element from the returned XML, then creates a JSON object from the JSON string.

Test the Donate Vacation use case with the Web service

We can now test the Web service and then invoke it from the Web 2.0 front-end:

1. Deploy the application on the server. Notice the warning in the WASCE Console when deployment is finished:

   ```
   [date] WARN  [EndpointDescriptionImpl] This implementation does not
   contain a WSDL definition and is not a SOAP 1.1 based binding. Per JAXWS
   spec. - a WSDL definition cannot be generated for this implementation.
   Name: donate.web.rest.ws.DonateRestWS
   ```

   This indicates that the Web service has been deployed.

2. Test the servlet with the following URL:

   `http://localhost:8080/DonateWeb20/DonateRestWS`

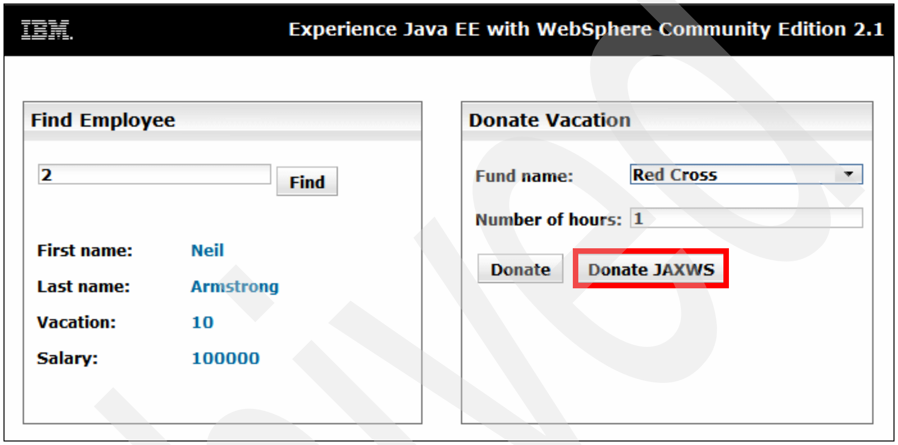
The reply to this HTTP `GET` request is:

```
<msg>This is the Donate RESTful Web services</msg>
```

3. Test the application with the URL:

   `http://localhost:8080/DonateWeb20/`

4. Retrieve an employee, then submit a donation by clicking **Donate JAXWS** (Figure 16-32).



Experience Java EE with WebSphere Community Edition 2.1

Find Employee

2

First name: Neil

Last name: Armstrong

Vacation: 10

Salary: 100000

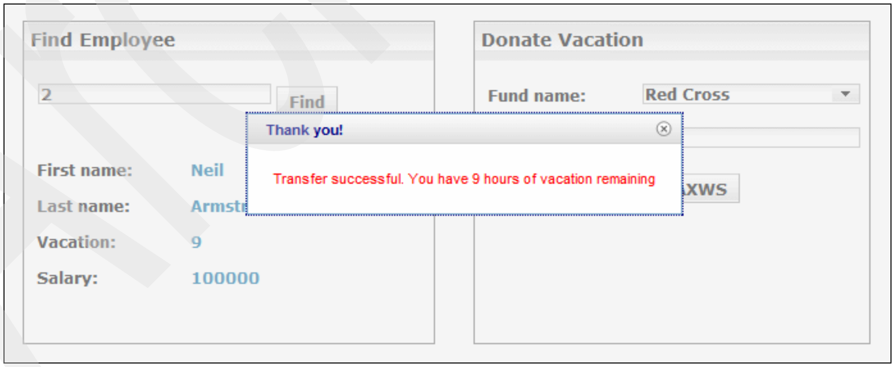
Donate Vacation

Fund name: Red Cross

Number of hours: 1

Figure 16-32 Testing Donate Vacation with the RESTful Web service

5. The result is displayed in the dialog box and the vacation hours are updated (Figure 16-33).



Find Employee

2

First name: Neil

Last name: Armstr

Vacation: 9

Salary: 100000

Donate Vacation

Fund name: Red Cross

Thank you!

Transfer successful. You have 9 hours of vacation remaining

Figure 16-33 Testing Donate Vacation with the RESTful Web service result

This concludes the implementation of the Donate Vacation use case.

Alternative:

In the REST Web service provider we used `javax.xml.ws.Source`, which generates and consumes XML files. We could have used `javax.activation.DataSource` and implement a JSON based source.

However, the Axis2 modules (used to expose the REST Web service) does not work with `DataSource` sources at the time this book was written.

Extra credit:

In the Web service class we create the returned XML as a `String` with XML tags. It is also possible to use JAXB with a defined data type to generate the XML. It would require the creation of a `jaxb.index` file or an `ObjectFactory` class. The following link provides more details on how to do this:

`http://java.sun.com/javaee/5/docs/tutorial/doc/bnbah.html`

16.10 Monitor the Web 2.0 interactions

We can use the TCP/IP Monitor to see the requests that are issued to the server when running the Web 2.0 application:

1. Define and start a TCP/IP Monitor as described in 11.8, “Monitor the SOAP traffic using TCP/IP Monitor” on page 349.
2. As an alternative, you can define a TCP/IP Monitor from the Servers view. Right-click the server and select **Monitoring** → **Properties**. In the Properties pop-up, click **Add** to define a Monitor on port 8081, then click **Start** to start the Monitor.
3. Run the Web 2.0 application using the URL:

   `http://localhost:8081/DonateWeb20/`

4. In the TCP/IP Monitor view, you can see the traffic between the Web 2.0 client and the server, including the calls to the REST servlets and the RESTful Web service.

16.11 Explore

In this section we provide additional Explore resources that could be used to enhance the Donate application.

16.11.1 Flex

Flex is an Open Source framework, originally developed by Macromedia®, which got acquired by Adobe, for building highly interactive, expressive Web applications that can run in the browser using Adobe Flash® Player runtime or on the desktop on Adobe AIR runtime. The Adobe Flash plug-in is available for every major browser and because countless Web sites uses Flash technology, the plug-in is pervasive.

16.11.2 Macromedia XML (MXML)

MXML™ is an XML-based markup language that is used to describe user interface layout and behavior. MXML code is used to declare UI components for creating rich user interfaces. These components are part of the rich library of extensible UI components that Flex provides.

For example, you use the `<mx:Button>` tag to create an instance of a button. Example 16-22 shows the complete code required to create a Flex application that displays a button control.

Example 16-22 Sample MXML file with code that renders a simple button

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Button id="someButton" label="I'm a button!" />
</mx:Application>
```

If you have coded the application using other UI markup technologies, such as Java Server Faces (JSF) or JSP tags, then the foregoing code snippet will be quite easy to understand. After you write a Flex application, you have to compile it using the Flex compiler and subsequently execute it to display the result on an HTML page (Figure 16-34).

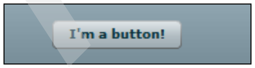


Figure 16-34 UI Rendering of a Flex button component

16.11.3 ActionScript

ActionScript is an object-oriented programming language that is used to implement client side logic. ActionScript is based on ECMAScript, the same standard that is also the basis for JavaScript. ActionScript is the programming language for the Adobe Flash Player run-time environment. It enables interactivity, data handling, and much more in Flash content and applications.

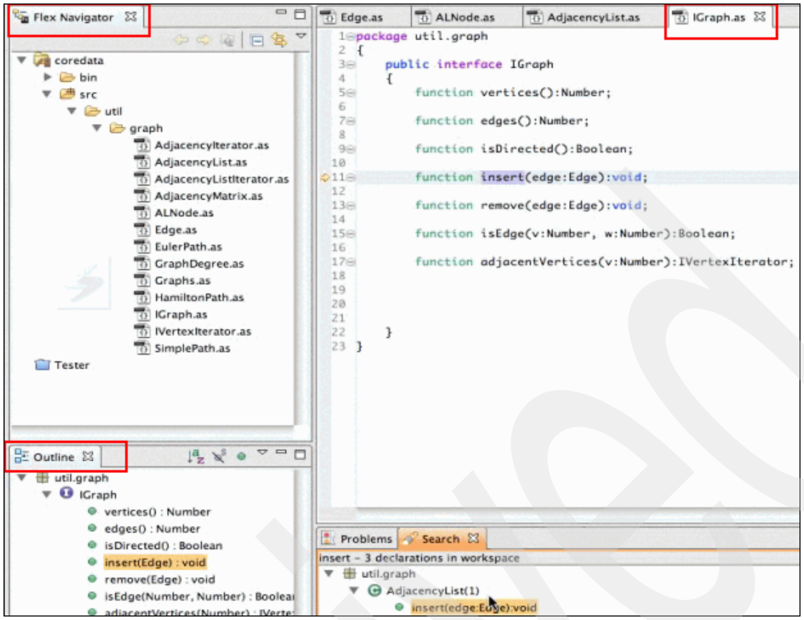
ActionScript is executed by the ActionScript Virtual Machine (AVM), which is part of Flash Player. ActionScript code is typically compiled into byte code format by a compiler. The byte code is embedded in SWF files, which are executed by the Flash Player run-time.

In Example 16-23, the `HelloWorld` class includes a single `typeHelloWorld` method, which returns a string that says Hello World!. This code snippet would be quite easy to understand for Java and JavaScript programmers.

Example 16-23 HelloWorld class written in ActionScript v3.0 syntax

```
public class HelloWorld {
   public function typeHelloWorld():String {
      var hwString:String;
      hwString = "Hello World!";
      return hwString;
   }
}
```

Adobe Flex™ Builder™ (Figure 16-35) is a commercial Eclipse based Integrated Development Environment (IDE) tool that facilitates intelligent coding, interactive step-through debugging, and visual design of the user interface layout, appearance, and behavior of rich Internet applications.



```
package util.graph
{
    public interface IGraph
    {
        function vertices():Number;

        function edges():Number;

        function isDirected():Boolean;

        function insert(edge:Edge):void;

        function remove(edge:Edge):void;

        function isEdge(v:Number, w:Number):Boolean;

        function adjacentVertices(v:Number):IVertexIterator;




    }
}
```

Figure 16-35 Adobe Flex Builder IDE

16.12 References for further learning

Here are additional references for further learning:

- RESTful Web services:

  `http://java.sun.com/developer/technicalArticles/WebServices/restful/`

- Publishing a RESTful Web service with JAX-WS:

  `http://weblogs.java.net/blog/kohlert/archive/2006/01/publishing_a_re.html`

17

Advanced deployment with WebSphere Virtual Enterprise

In this chapter, we describe how to optimize operations with WebSphere Virtual Enterprise.

17.1 Introduction

WebSphere Virtual Enterprise (WVE) is software that provides virtualization capabilities used to balance an application server infrastructure's workload. WVE provides application infrastructure virtualization capabilities that lower operational and energy costs to create, run, and manage your enterprise applications and service-oriented architecture (SOA) environment.

Increasingly, businesses are tied rigidly to the availability and speed of applications that deliver essential services to customers. Loss of availability translates into lost business, which means lost opportunity and lost revenue. To meet this need, the dynamic operations environment is a fluid and dynamic environment, enabling applications to be available continuously. It does this through the virtualization of WebSphere resources, provisioning of WebSphere applications, prioritization and scheduling of applications, and integrating with overall dynamic operations environment infrastructure management.

The dynamic operations environment consists of autonomic managers whose purpose is to maximize utilization using business goals the customer has defined. They monitor performance metrics, analyze the monitored data, offer a plan for executing actions, and have the capability to execute these actions in response to the flow of work.

Application infrastructure virtualization enables chief information officers (CIOs) and IT administrators to do more with less—literally. It allows data centers to run applications on any application server in a common resource pool. Furthermore, administrators can deploy and utilize resources quickly and seamless during peak periods and in response to the peaks and valleys of unforeseen demand in processing for various mission-critical applications. Last, but certainly not least, administrators can achieve application response times and service levels that meet service level agreements.

IBM WebSphere Virtual Enterprise can help your business extend the benefits of server virtualization using an approach that can address potential issues and limitations. WebSphere Virtual Enterprise can be thought of as a hypervisor for application servers. In addition, it enables server virtualization and application infrastructure virtualization to be combined so that you can take full advantage of the strengths of both approaches.

Originally developed for WebSphere Application Server ND, WVE has evolved to deliver a number of these capabilities to other application servers, including:

- WebSphere Application Server Community Edition
- PHP server
- Apache HTTP Server
- Apache Tomcat

- Apache Geronimo
- JBoss Application Server
- BEA WebLogic
- Custom servers

While Virtual Enterprise can provide qualities of service to all of these server offerings, the extent here will only be to address the differences between full life cycle management of WebSphere Application Server Network Deployment (WAS-ND) and WebSphere Application Server Community Edition (WASCE).

Prior to reading this section, we recommend that you briefly overview the virtual enterprise white paper:

> *Extending virtualization in the data center with application infrastructure virtualization: Introducing WebSphere Virtual Enterprise*, found at:
>
> `ftp://ftp.software.ibm.com/software/webservers/appserv/WebSphere_Virtual_Enterprise_wp.pdf`

For more information, refer to the IBM Redbooks publication, *Optimizing Operations with WebSphere Extended Deployment V6.1*, SG24-7422, and to the WebSphere Virtual Enterprise Information Center at:

> `http://publib.boulder.ibm.com/infocenter/wxdinfo/v6r1/index.jsp?topic=/com.ibm.websphere.xd.doc/info/welcome_61_xd.html`

17.2 Terminology

In this section we define the terminology of WebSphere Virtual Enterprise.

17.2.1 Load balancing

Load balancers distribute incoming client requests across servers. The technology can boost the experienced performance of applications by spreading/redirecting the load across a group of application servers. By directing requests to different servers within the group, it balances the work among them. This load balancing is transparent to users and other applications.

Traditional load balancers have limited if any intelligence of the back-end server's available capacity to handle work. They manage incoming requests in a round-robin approach. For example, if you have ten servers being load balanced, the first user request would go to server one, the second request to server two, the third request to server three, and so forth. This design can result in work requests being routed to an overloaded or hung server, or not sending a request to a server that is under-utilized. This has the associated effect of indefinable service availability of an application and poor utilization of server capacity.

WebSphere Virtual Enterprise introduces intelligence to the load balancing concept. The component of WVE that acts as an intelligent load balancer is called the *on-demand router* or ODR. All incoming requests go through the ODR. The ODR can classify an incoming request, rank its priority and queue it to a back-end server. The ODR has knowledge of the available processing capacity and health of the back-end servers and so with this information can make informed decisions on where the request should be run.

With WVE, performance goals can be defined and these then bound to specific subsets of the incoming traffic. The ODR and associated autonomic managers are able to support business goals in times of high load by making smart decisions about the work coming into the ODR. Not all work in your configuration is created equal. The ODR is able to support this concept by forwarding different flows of requests more or less quickly to achieve the best balanced results.

17.2.2 Clustering

The term clustering tends to be thrown around loosely in the software industry causing great confusion. For this reason, we will spend some time go over what clustering is.

An application server cluster is a group of servers that transparently provide enterprise services, such as servlet and JavaServer Pages (JSP) support, as if it was a single server. The servers, typically running on separate systems, exchange messages to synchronize data, allowing any individual node to process requests for a distributed application and to take over a user's session data when its node fails. Configuring multiple servers into a cluster is commonly called clustering.

Consider using a cluster of servers when you need to improve the availability of web applications or resources.

- Load balancing in a cluster can be achieved with either software, hardware, or by adding a component, like the on-demand router, that intelligently distributes requests to each node in the cluster:
  - Running multiple nodes on the same host, called vertical clustering, is used when limitations in a single node prevents it from fully utilizing the capacity of the host.
  - Running multiple nodes on different hosts, called horizontal clustering, is used to add capacity by adding an additional host. Horizontal clustering is more common than vertical clustering because it also improves availability in the event of a physical server to go down.

- Clustering improves availability by automatically synchronizing state data among all the nodes in the cluster. If any node in the cluster fails, subsequent requests can be processed by any other node in the cluster. Note that this process, commonly called failover, occurs on subsequent requests. Any requests being processed by a node when it fails will still be lost. It is extremely important to understand that this level of failover can only be achieved with software and not hardware. This is because only software can address Java objects and resources that are stored in memory or in a database system.

WVE does not provide any enhancements to the application server's default clustering capabilities, but as you will see when coupled with a highly scalable application in a clustered environment with WAS-ND, WVE can perform dynamic workload management. This feature will be covered in detail in17.3.1, “Complete life cycle management with WAS-ND” on page 527. Also, currently clustering with WASCE servers with WVE is not supported.

17.2.3 High availability

In simple terms, high availability is the degree and level in which a system is up and running. For example, if you open a browser and go to the URL `www.ibm.com`, you receive the IBM landing page and not a 404 error. Often technologists intertwine high availability with clustering—the ability for cloned systems to be able to pick up the workload of another system in the event of a failure.

17.2.4 Dynamic clusters

WVE uses the term *dynamic cluster* to distinguish the pool of servers in which it can deliver workload to. WVE can essentially turn applications on or off, or even relocate and distribute applications to each cluster member. The term cluster in this sense should not be confused with clustering for instances to share application resources as defined previously.

17.2.5 Deployment and configuration management

Deployment and configuration management is the ability to configure and manage applications and configurations across all the servers in the cluster or server farm.

For example, if application A is deployed onto 50 machines, and there is an update to application A, the ability to have the update distributed to all 50 applications and ensuring the right version is installed and running, is typically called deployment management.

Today there are three basic ways in which customers can achieve deployment management:

- Use a centralized administrative console such as WAS-ND.
- Use command line built scripts that automate through the process (usually built and maintained by the development organization).
- Manually, either use the command line or use some type of administrative console, for each server node.

17.3 Complete life cycle management with Virtual Enterprise

Middle-ware servers encompass all servers in the middle-ware tier that provide the infrastructure for applications or their data.

By configuring complete life cycle middle-ware servers, the WebSphere Virtual Enterprise dynamic operations environment can govern all operational aspects of these servers.

The following middle-ware server types are supported for a complete life cycle:

- WebSphere Application Server Network Deployment (WAS-ND) v6.1
- PHP servers
- WebSphere Application Server Community Edition v2.x (all releases)

WebSphere Virtual Enterprise environment can perform the following tasks on complete life cycle middleware servers:

- Create and remove server instances based on the needs of the dynamic operations environment
- Govern all aspects of server configuration
- Provide operational control over the server
- Deploy applications to the server
- Visualize and monitor server health and performance
- Use expression-based dynamic clustering

17.3.1 Complete life cycle management with WAS-ND

The management system of WebSphere Application Server Network Deployment (WAS-ND) provides a unified management view, called a single-system image, across a multi-process, multi-machine, heterogeneous deployment environment. It allows the system to be configured, modified, monitored, and controlled through a single point of administration from multiple administrative agents, simultaneously and dynamically.

A key feature of managing ND clusters with WVE is being able to dynamically re-distribute the workload. For example, if a set of users are on a single server, and these users start to select operations in the application that perform a heavy work load, then with most infrastructures today, these users would be stuck on this server and continue to experience a *slow* session. WVE has the ability to be able to reroute some of the users to other servers without losing any of their session data. This is because these applications are clustered and each server can pick up user session data. While other servers do perform a certain level of intelligent workload management, they might not have the high-end clustering abilities WVE needs in order to dynamically reroute requests.

17.3.2 Complete life cycle management with WASCE v2.x

WebSphere Virtual Enterprise (WVE) provides complete life cycle management for WASCE and this gives IBM advantages over many other vendors in the market place. While open source application servers offer lower cost entry points, customers usually do not consider deployment management and administration into their software purchasing decisions. Over the total life cycle of applications, it is found that initial software acquisition cost is only a small subset of the total cost of ownership. Administration is the largest cost, estimated by IBM to be around 74 percent. With WVE managing a WASCE server farm, IBM can provide an application server environment that has a lower acquisition cost and lightweight development environment with a full-blown enterprise administration and deployment management solution. No other vendor has an offering and combination such as this.

WASCE has limited clustering capabilities. When a WVE dynamic cluster of WASCE servers is set up, what is created is a set of singleton WASCE servers with matching configurations/applications. WVE can ensure that the application is highly available and response times and other quality of service metrics are ensured. WVE provides complete life cycle support for these servers but does not add the capability of clustering to the servers themselves.

WASCE only provides basic failover of Java objects stored in HTTP session known as session replication. WASCE implements HTTP session replication using memory-to-memory communication technology provided by Apache

Tomcat. This is a peer-to-peer replication where all session objects have to be serialized and copied to every instance of servers in the cluster. This algorithm is only efficient when the clusters are small. If the clusters grow too large, the overhead in storage utilization and network traffic becomes excessive.

This mechanism has obvious scalability issues, and for this reason WVE will turn this mechanism off. Our current recommendation when using CE with WVE is to either implement stateless applications, or not to use in-memory session replication.

It is important to also note that WASCE does not support EJB, JMS, or any other type of clustering/failover capabilities. This is acceptable for some applications. If session state failover is required, we recommend using either database backed persistent failover or data replication service (memory-to-memory) with our Network Deployment product. If this is not a requirement, CE can be a good alternative.

WVE does support session affinity, also known as sticky session support, with WASCE. This allows the on-demand router to route an HTTP request back to the same node that created the HTTP session associated with that request. If this node becomes unavailable, the request will be rerouted to an available node and the user will be required to re-log back into the application. However, all user session data will be lost.

Note:

The use of an HTTP session can be a complicated subject that can lead into heated I/T discussions. The overhead and complexities of EJB 2.1 and earlier releases are regarded as major contributors to its widespread use. Many I/T organizations have steered away from using EJBs and instead opted for alternative, easier to build, stateful implementations. Many organizations opted to implement conversational session state through HTTP sessions. Even though widely used, managing state through HTTP sessions is generally not recommended for building highly available scalable applications because all Java objects have to be serialized, which can cause a large overhead.

State should be implemented using frameworks and standards such as EJB. The new EJB 3.0 programming model uses annotations. The result makes state management radically simpler. Developers can now re-think implementing applications on the Java EE 5 programming model.

For more information, refer to *Best Practices for using HTTP sessions* in the IBM InfoCenter at:

```
http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp
```

Alternatively, an emerging and viable alternative for state management is to use grid or data fabric based solutions. The IBM solution in this area is called WebSphere eXtreme Scale (WXS). WXS provides application acceleration and transaction processing for objects obtained from back-end systems or stored within the WXS grid. WXS is a software layer that sits behind an application providing the application its data from WXS itself or from back-end systems. The layer can be embedded in the same Java Virtual Machine as the application or can be a distributed layer where it makes sense for the application.

17.4 Assisted life cycle management

Assisted life cycle management provides specific templates for creating representations of existing servers and applications. Servers can be controlled operationally and health and performance can be monitored visually. Currently WAS-ND/Base v5.1 and v6.0, WASCE v1.1, BEA WebLogic, Geronimo, and Tomcat can participate in an assisted life cycle management configuration.

17.5 Generic life cycle management

Generic life cycle management provides generic templates for the user to manually define servers and operational commands. Control server operations and monitor health and performance. Currently, .Net and SAP NetWeaver can participate in a generic life cycle management configuration.

18

Integration with IBM HTTP Server v7.0

In this chapter, we describe how to install the IBM HTTP Server v7.0, and how to configure it to interact with WASCE.

18.1 Introduction

The IBM HTTP Server is based on the Open Source Apache HTTP Server, a flagship and most popular Web server developed by the Apache Software Foundation. The following benefits are available when the IBM HTTP Server is configured as a remote Web server to the IBM WebSphere Application Server Community Edition (WASCE):

- Clustering of application servers and load balancing of the applications can be done with the help of inbuilt support of IBM HTTP Server.
- The IBM HTTP Server contains Secure Sockets Layer (SSL) support built-in with IBM's GSKit library, offering a way to get server affinity for SSL-encrypted messages. This helps WASCE concentrate on the business logic of the application.
- The IBM HTTP Server handles the request for static pages (including HTML, JPEG, and GIF files), while WASCE handles requests for dynamic pages (JSPs or servlets) with the help of Tomcat containers. In this way, they share the workload of handling HTTP requests.
- Using the IBM HTTP Server to front the WASCE enables the creation of a demilitarized zone or DMZ, where you can host the Web server on a server that is placed between a company's private network and the Internet. This allows to place the servers that are hosting WASCE in a more secured environment behind a private firewall and therefore, not directly connected to the Internet or some other outside public network.

The IBM HTTP Server v7.0 delivers an updated version of the Apache HTTP Server v2.2.6. According to the Apache HTTP Server home page, version 2.2.x is a major release and the start of a new stable branch, and represents the best available version of the Apache HTTP Server.

Here are some of the new features provided:

- Apache JServ Protocol (AJP) version 1.3 proxy to connect to Apache Tomcat, and proxy load balancing.
- Smart filtering
- Improved caching
- Graceful shutdown support
- Large file support
- Event multi-processing module (MPM)
- Refactored authentication and authorization modules

For more information, refer to:

http://httpd.apache.org/docs/2.2/new_features_2_2.html

18.2 Install the IBM HTTP Server v7.0

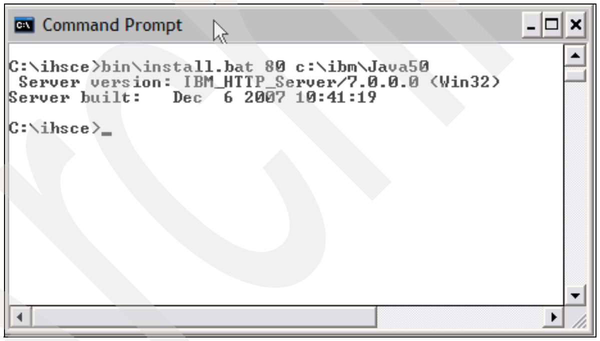
When this book was written, IBM HTTP Server v7.0 was not publicly available for download. But, it was available for those customers who were paid subscribers of IBM support for the IBM WebSphere Application Server Community Edition product. We procured the installation file for the Windows platform for the IBM HTTP Server v7.0 from the IBM support.

Following are the step by step instructions to install the IBM HTTP Server v7.0, from here on referred as HTTP Server, on the Windows platform:

1. Create a new directory where you want to install the HTTP Server. From here on this directory is referred as `<IHS_INSTALL_ROOT>`, for example: `C:\IBM\IHS`.
2. Unzip the archived installation file, `ihsce.win.ia32.zip`, into the `<IHS_INSTALL_ROOT>`.
3. Open a command window and change the directory to `<IHS_INSTALL_ROOT>`.
4. Execute the following command from the command prompt (Figure 18-1):

```
bin\install.bat <IHS_Http_Port> <IBM_JAVA5_Home>
bin\install.bat 80 "C:\Program Files\IBM\Java50"
```

   Where `<IHS_Http_Port>` is the HTTP port for the HTTP Server, and `<IBM_JAVA5_Home>` is the directory where you installed the IBM JDK V5.



```
C:\ihsce>bin\install.bat 80 c:\ibm\Java50
 Server version: IBM_HTTP_Server/7.0.0.0 (Win32)
Server built:   Dec  6 2007 10:41:19

C:\ihsce>_
```

Figure 18-1 Execute install.bat to install IBM HTTP Server v7

5. This command executes the install script that installs the HTTP Server. Upon successful installation, a status message is displayed in the command window that lists the version of the installed HTTP Server and the platform and the date of the build.
6. The install process also adds IBM HTTP Server v7.0 as a Windows service. You can verify this by opening the Services window (**Control Panel** → **Administrative Tools** → **Services**).

Verify the Installation

To verify the installation, perform these tasks:

1. In the Services window, select **IBM HTTP Server 7.0** and click the **Start** icon ▶ to start the HTTP Server.
2. When the service status changes to `Started`, open a Web browser at `http://localhost/` to view the IBM HTTP Server console (Figure 18-2).

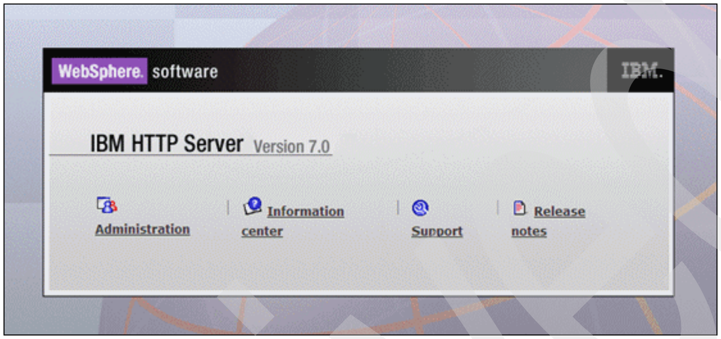


Figure 18-2 IBM HTTP Server Console

3. You can also verify if the install was successful by viewing the contents of the log file created by the install process in:

   `<IHS_INSTALL_ROOT>/logs/install.log`

4. After the HTTP Server has started, the `<IHS_INSTALL_ROOT>/logs` directory also contains the process-id file, `httpd.pid`, of the `httpd` process.

18.3 Configure the HTTP Server for WASCE

In this section we configure the HPPT Server to interact with WASCE.

18.3.1 Configure the HTTP Server as a remote Web server

Apache Geronimo comes in two flavors depending on the Web container you decide to use. You can choose between the Tomcat or Jetty distributions. WASCE is based on Geronimo, which uses Tomcat as its Web container or servlet engine.

There are two ways in which the HTTP Server can be configured to forward client requests to WASCE and thus front it as its remote Web server:

- Configuring the HTTP Server as a reverse proxy using the built-in `mod_proxy` module.

- Configuring the HTTP Server with the Tomcat Connector `mod_jk` available with the Apache Tomcat source.

The Apache Geronimo console provides a `mod_jk` configuration portlet that facilitates the configuration of the HTTP Server as a remote Web server to Geronimo. However, in the past, according to the Apache Geronimo V2.0 documentation, this portlet presented some inconsistencies recognizing the already installed Web applications.

The Apache HTTP Server v2.2.x adds native support for the Apache JServ Protocol (AJP) v1.3 used by Apache Tomcat. The old `mod_proxy` has been rewritten and has a new AJP capable protocol module (`mod_proxy_ajp`) and an integrated software load balancer (`mod_proxy_balancer`). The addition of `mod_proxy_ajp` and `mod_proxy_balancer` to the `httpd` code base reduces the number of hoops to jump through to get AJP connectivity to Tomcat. The AJP 1.3 protocol keeps an open socket and controls the communications between WASCE (actually its Tomcat component) and the HTTP Server.

Therefore, IBM WebSphere Application Server Community Edition v2.1 recommends the configuration of the IBM HTTP Server v7.0 as a remote Web server by using the `mod_proxy_ajp` module. This method is also the easiest way to re-route client requests from the HTTP Server to WASCE. This configuration process involves enabling some specific modules and adding a few lines to the HTTP Server configuration file without the need for any additional configuration on the Application Server side.

This process follows these basic steps:

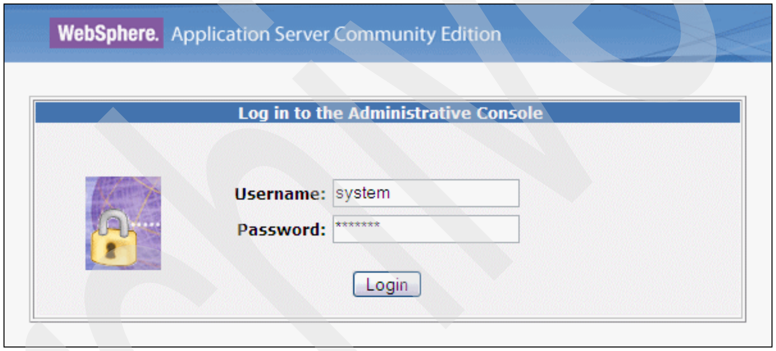
1. The `mod_proxy.so` file is available in the `<IHS7_Install_Root>/modules` folder, whereas the `mod_proxy_ajp.so` and `mod_proxy_balancer.so` files are available in the `<IHS7_Install_Root>/modules/WebSphereCE` folder.
2. It appends the following lines at the end of the HTTP Server configuration file, `<IHS7_Install_Root>/conf/httpd.conf`.

   ```
   LoadModule proxy_module modules/mod_proxy.so
   LoadModule proxy_ajp_module modules/WebSphereCE/mod_proxy_ajp.so
   LoadModule proxy_balancer_module modules/WebSphereCE/mod_proxy_balancer.so

   ProxyPreserveHost On

   # Add your URL mapping here. The following is for console app.
   ProxyPass /console ajp://<hostname>:8009/console
   ```

   – The `LoadModule` statement tells the HTTP Server to load the specified library.

- The ProxyPreserveHost statement is for keeping the original Host URL in the location header on HTTP responses, therefore, hiding the identity and location of WASCE.
- The ProxyPass statement(s) tell the Web server which URL's patterns will be re-routed. In the previous example, we route all requests with the /console URL pattern to the WASCE administrative application using the AJP protocol through the 8009 port of the server that hosts WASCE. 8009 is the non-secure port which Tomcat uses for listening to requests using the AJP protocol. Substitute <hostname> with the host name of the system, or localhost.

3. It restarts the HTTP Server.
4. It opens a browser with the URL http://<hostname>/console. You can see the WASCE Console, thus confirming that all the requests made to the HTTP Server with the /console URL pattern are being re-routed to WASCE (Figure 18-3).


WebSphere. Application Server Community Edition
Log in to the Administrative Console
Login

Username: system
Password: *******

Figure 18-3 WASCE Console reached through the HTTP Server

18.3.2 Enable SSL on the IBM HTTP Server

One of the major benefits of using the IBM HTTP Server to front WASCE as a remote Web server is the ability to encrypt the communication between the clients (browsers) and the HTTP Server by enabling SSL on the HTTP Server using digital certificates.

In this section, we take you through the process of enabling SSL on the HTTP Server.

Modify the PATH system environment variable

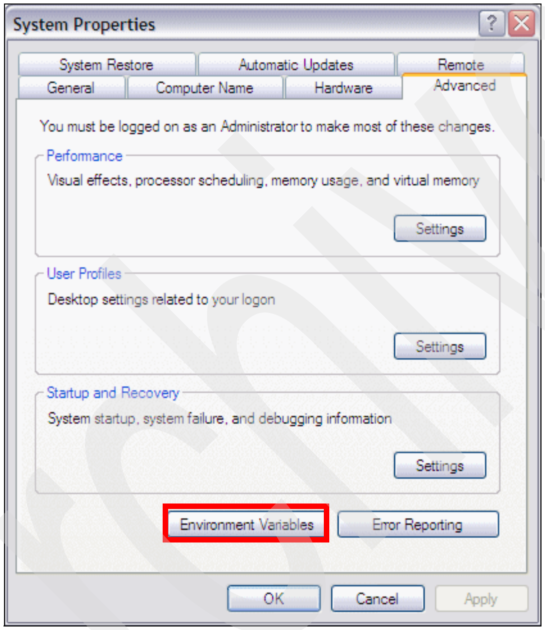
Prior to running any commands to generate the certificates, you have to add the following two paths to the PATH system variable:

```
<IBM_JAVA5_HOME>\jre\bin        ==> C:\Program Files\IBM\Java50\jre\bin
<IHS7_Install_Root>\gsk7\bin    ==> C:\IBM\IHS\gsk7\bin
```

- For Linux, use the following export command to modify the PATH variable:

```
export PATH=/opt/ihsce/_jvm/bin:/usr/local/ibm/gsk7/bin:$PATH
```

- For Windows, open the System Properties (select **Control Panel** → **System**). In the System Properties dialog, select the **Advanced** tab and click **Environment Variables** (Figure 18-4).



You must be logged on as an Administrator to make most of these changes.

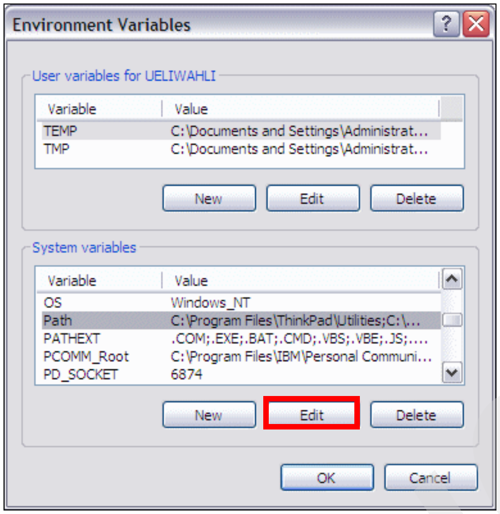
Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your logon

System startup, system failure, and debugging information

Figure 18-4 System Properties Advanced tab

- Locate the **Path** system variable in the System Variables section in the Environment Variables dialog and click **Edit** (Figure 18-5).



Environment Variables

User variables for UELIWAHLI

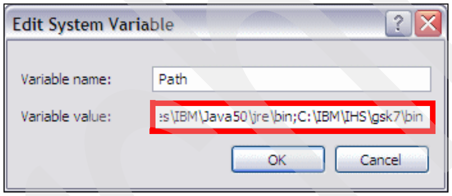
Variable	Value
TEMP	C:\Documents and Settings\Administrat...
TMP	C:\Documents and Settings\Administrat...

System variables

Variable	Value
OS	Windows_NT
Path	C:\Program Files\ThinkPad\Utilities;C:\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PCOMM_Root	C:\Program Files\IBM\Personal Communi...
PD_SOCKET	6874

Figure 18-5 Editing the Path variable

- Append the Path of the IBM Java 5 JRE and the `gsk7cmd` tool to the value of the Path System Variable (Figure 18-6).



Edit System Variable

Variable name: Path

Variable value: :s\IBM\Java50\jre\bin;C:\IBM\IHS\gsk7\bin

Figure 18-6 Append the Path system variable

Create a new key database

The digital certificate used to encrypt the data exchanged using the Secure Socket Layer (SSL) protocol is stored in an encrypted file, called a *keystore* or a key database. By convention, a new SSL certificate is always created in a new keystore.

- Create a new directory where you will store the new key database, for example, `C:\IBM\IHS\`**`ssl`**.
- Open a command window and change the directory to the newly created directory where the key database will be saved.
- Make sure that the `JAVA_HOME` environment variable is pointing to the Java 5 JDK, for example:

```
set JAVA_HOME=c:\Program Files\IBM\Java50
```

- Create a new key database using the **gsk7cmd** command:

```
gsk7cmd -keydb -create -db key.kdb -pw passw0rd -type cms -expire 365
                -stash
```

- This command creates a key database (key.kdb) with the string passw0rd as its password and sets its expiry to 365 days. The command generates the following files in the ssl directory.

```
09/10/2008  04:52 AM              80 key.crl
09/10/2008  04:52 AM         115,080 key.kdb
09/10/2008  04:52 AM              80 key.rdb
09/10/2008  04:52 AM             129 key.sth
```

- Note the size of the key database.

Create a self-signed certificate

For the purposes of this exercise, a new self-signed certificate is sufficient, because a trusted certificate authority will charge a fee to sign your certificate.

Creating a self-signed certificate generates a self-signed X509 certificate in the specified keystore. A self-signed certificate has the same issuer name as its subject name.

- In the same command window, create a self-signed certificate using the **gsk7cmd** command:

```
gsk7cmd -cert -create -db key.kdb -pw passw0rd -size 1024 -dn
"CN=<hostname>,O=ibm,C=US" -label HTTPServerCertificate -default_cert
yes -expire 365
```

- This command creates a self signed digital certificate using the key database we created in the last step and specifies the same password we used to create the key database and also sets it expiry to 365 days. The attributes of the command also assign a label to this certificate and sets the distinguished name (dn) of the entity this certificate belongs to. Notice that the size of the key database file, key.kdb, has increased.

Configure the IBM HTTP Server to enable SSL

To enable the encryption between the HTTP Server and the browser using the SSL protocol, we have to modify the HTTP Server configuration file and add lines that will load the IBM SSL Module and listen for secure HTTP on port 443 for a specified virtual host.

The following process outlines the steps to do this:

1. Open the HTTP Server configuration file, `<IHS7_Install_Root>/conf/httpd.conf`, and append the following lines to it:

   ```
   ServerName <hostname>
   LoadModule ibm_ssl_module modules/mod_ibm_ssl.so
   <IfModule mod_ibm_ssl.c>
     Listen 443
     <VirtualHost <hostname>:443>
       SSLEnable
       SSLClientAuth None
     </VirtualHost>
   </IfModule>
   SSLDisable
   KeyFile "C:\IBM\IHS\ssl\key.kdb"
   SSLV2Timeout 100
   SSLV3Timeout 1000
   ```

   These lines specify a new virtual host associated with port 443, which is enabled for SSL communication. The complete path of the key database where the self signed digital certificate is store is also specified.

2. Append the following line to `httpd.conf`:

   ```
   ProxyPass /console ajp://<hostname>:8445/console
   ```

   This `ProxyPass` statement tells the HTTP Server to route all the client requests made with the /console URL pattern to the port `8445` of the WASCE server using the AJP protocol. Port `8445` is the secure AJP port for Tomcat.

3. Restart the HTTP Server.

Verify the SSL connectivity

Follow these steps to verify connectivity:

1. Open the browser with the URL `https://localhost/console`.
2. Based on the browser you are using, you might be informed about the secure communication and will provided options to view the details of the certificate being used by the HTTP Server, and you can accept or reject it.
3. If you select to proceed, the WASCE Console application is displayed (Figure 18-3 on page 536).
4. All the communication between the browser and the HTTP Server and WASCE for this console application with be encrypted using the SSL protocol.

18.3.3 Use the HTTP server for the Donate application

The Donate Web applications use these URLs:

```
http://localhost:8080/DonateWeb/FindEmployee.jsp
http://localhost:8080/DonateJSFWeb/faces/FindEmployee.jsp
```

To forward requests from the HTTP Server to these URLs, we have to add additional statements to the `httpd.conf` file:

1. Enable HTTP forwarding by activating the `proxy_http_module`:

   ```
   LoadModule proxy_http_module modules/mod_proxy_http.so
   ```

2. Pass the URLs:

   ```
   ProxyPass /DonateWeb      http://localhost:8080/DonateWeb
   ProxyPass /DonateJSFWeb   http://localhost:8080/DonateJSFWeb
   ```

3. Now run the Donate Web applications by opening a browser with these URLs:

   ```
   http://localhost/DonateWeb/FindEmployee.jsp
   http://localhost/DonateJSFWeb/faces/FindEmployee.jsp
   ```

19

What next?

Congratulations! You have completed the Experience Java EE! scenario:
You created and tested a comprehensive Java EE application addressing real-life issues such as supporting multiple back-end databases, security, Web services, and messaging (JMS).

Our hope is that this book met or exceeded your expectations, and that by *Experiencing Java EE* you were able to move from a theoretical knowledge gained by reading introductory material to a practical knowledge gained by implementing a real-life application.

However, the knowledge level imparted by this book is a starting point. You have to continue to grow the breadth and depth of your expertise in Java EE.

The Explore resources at the end of each chapter are intended to assist you in expanding your knowledge of the current technologies discussed in this book, but they still focus on Java EE.

Java EE and the Web 2.0 capabilities used in Chapter 16, “Develop a Web 2.0 client” on page 455 provide a core foundation for a modern computing infrastructure, but they do not address all the requirements that exist. What other technologies should you consider?

19.1 Extended Web services standards and WS-I

WASCE provides the base Web services support that is required by the JAVA EE specification, and in particular by *JSR 109: Web Services for Java EE 1.2 Maintenance Release*. This specification calls for the support of the WS-I Basic Profile 1.0:

- The Web Services Interoperability Organization (WS-I) is a consortium that develops guidelines to provide enhanced interoperability between Web services implementations, both Java and non-Java. These guidelines do not replace existing specifications (like SOAP) but refine and clarify their usage to enhance interoperability.
- The WS-I Basic Profile 1.0 focused on interoperability using base standards (SOAP 1.1, WSDL 1.1, UDDI V2.0, SSL V3, and so forth).

This profile was released in 2004, and the value and impact were immediate because it provided a set of guidelines that could be used both by product providers and Web services developers. This does not imply that WS-I Basic Profile 1.0 addressed the complete spectrum of interoperability issues, but it did provide a sufficient scope to significantly reduce the problems experienced by customers and the industry.

WS-I Basic Profile 1.0 did not cover new standards that are being widely adopted. WS-I has continued to work on additional profiles to address these additional standards as well as refining and improving the existing Basic Profile based on feedback from the broader community.

The current WS-I working set of standards and profiles is available at:

`http://www.ws-i.org/deliverables/matrix.aspx`

Some of the key additional profiles and associated standards include these:

- **Basic Profile 1.1 (final 04/2006):** Maintenance update from 1.0
- **Basic Profile 1.2 (draft):** Extends the basic interoperability criteria:
  - SOAP 1.2: update of the base messaging definition.
  - WS-Addressing 1.0: Inclusion of replyTo addresses in a SOAP message, thereby enabling asynchronous Web service responses.
  - Message Transmission Optimization Mechanism 1.0 (MTOM): Interchange of binary data.

- **Basic Security Profile 1.0 (final 03/2007):** Core security considerations in Web service interactions:
  - WS-Security 1.0: Integrity (the data has not been manipulated) and confidentially (the request or response is from authorized parties) through encryption, signatures, and security tokens (for example, identity).
- **Basic Security Profile 1.1 (draft):** Update from 1.0:
  - WS-Security 1.1: Additional Kerberos and Security Assertion Markup Languages (SAML) profiles.
- **Reliable Secure Profile 1.0 (draft):** Asynchronous interactions and enhanced security:
  - WS-ReliableMessaging 1.1: Delivery of SOAP messages through alternate transports (other than TCP/IP) to de-couple the requester and responder from disruptions in the communication.
  - WS-Trust 1.3: Exchange of security tokens that can be generated and validated by a third party (trust server).
  - WS-SecureConversation 1.3: Extension of the security model to work across a sequence of Web service requests/responses, providing for a more efficient conversation level security (instead of a unique application of security for each individual request/response).

The various WS-I profiles do not cover all current and draft specification, but focus on those that the consortium members believe have the widest consensus and adoption patterns. For example:

- WS-I does not have profiles that include publish-subscribe paradigm because currently there are two competing standards (WS-Notification and WS-Eventing).
- WS-I does not have profiles that include WS-Coordination, WS-Atomic Transaction, and WS-SecurityPolicy, to name a few. These are all approved Oasis specifications that are seeing interest and consensus, but have yet to be included in WS-I activities.

WASCE supports SOAP 1.2 and MTOM (because they are required by Java EE), but does not contain any other specifications contained in the approved and draft WS-I profiles described previously.

However, developers can download and include standalone Axis2 and Axis2 modules in their applications to support these specifications, as we did in this book for WS-Security with the Apache Rampart capability in 12.9.1, “Create a secure Web service with Axis2” on page 386.

Core Axis2 and Rampart implement all the specifications included in the various WS-I profiles described previously, except for WS-ReliableMessaging. The `Sandesha2` Axis2 module implements the 1.0 version of that specification, but does not yet support the 1.1 version specified by the draft WS-I Reliable Secure Profile 1.0.

Explore resources are available at the following Web sites:

- JSR 109: Web Services for Java EE 1.2 Maintenance Release

  `http://jcp.org/en/jsr/detail?id=109`

- Web Service Interoperability Organization

  `http://www.ws-i.org`

19.2 Portals

Consider an environment where a user accesses multiple Web applications. These Web applications were all developed separately by different organizations and potentially might display related information—where the user might want to take a value from one application and use it to invoke a query in another application.

How do you allow these applications to operate in a single-user interface that uses the same security and access control model? How do you create a *user desktop* within the browser, similar to the windowed environments for normal desktops (like Windows, Macs, and Linux)?

The answer is that you implement a portal, which is a server-side application that provides a framework for presenting multiple applications within separate panes, on a single client-side browser page. Portals effectively provide *on the glass* application aggregation.

A portal environment requires two primary elements: Portlets and a portal server.

Portlets

Portlets, which are the display components for an application. In the Java EE environment, these can be based on special JSPs that utilize portlet tag libraries or Java-based classes that extend the GenericPortlet class (consider this to be similar to a servlet). Most vendors support either or both of these specifications:

- **JSR 168**: **Portlet Specification (1.0)** released in October 2003
- **JSR 286**: **Portlet Specification (2.0)** released in June, 2008

JSP-based portlets can also utilize JSF, allowing you to have a single pane that displays a sequence of related Web pages. However, the core portlet specifications do not specify the integration between portlets and JSF, so any portlet/JSF implementation is vendor specific.

- **JSR 301**: **Portlet Bridge Specification for JavaServer Faces** defines this integration, but the specification is currently in draft form.

Portlet messaging allows one portlet to be invoked based on a value from another portlet. For example, consider an automotive parts applications where the user would sequentially select year, make, and selection. The year selection could be configured as one portlet, which when set can then trigger the make selection in another portlet, which when set can then trigger the model selection in a third portlet. Other portlets could then be triggered as well, supplying additional data such as an image or a parts list.

Portal server

A portal server provides the common runtime environment for security, user interface management, common display structure, and so forth.

The portal server allows the administrator to define a series of pages, and on each page they can group one or more portlets. Access to these pages and portlets is controlled through role-based profiles, which are similar in concept to the Java EE application role-based security. As a result, each user sees a customized set of pages and portlets, providing them with a unique *desktop* for their role.

- **WebSphere Portal v6.1** is the IBM portal server, and it runs as a J2EE v1.4 (not Java EE 5) application on top of WebSphere Application Server v6.1, supporting both JSR 186 and JSR 286 portlets. WebSphere Application Server v6.1 supports the standalone execution of JSR 186 portlets, and WebSphere Application Server v7.0 supports the standalone execution of both JSR 186 and JSR 286 portlets.
- **Rational Application Developer** supports the development of both standard portlets and JSF portlets that can then be deployed to any of the environments described previously (v7.0 supports JSR 168 portlets and v7.5 supports JSR 168 and JSR 286 portlets).

WASCE does not support customer written portlets. WASCE does ship with the Pluto 1.1.6 libraries that support JSR 168 portlets, but this is for internal usage by WASCE provided artifacts within the WASCE Console.

Explore resources are available at the following Web sites:

- JSR 168: Portlet Specification (approved):

  http://www.jcp.org/en/jsr/detail?id=168
- JSR 286: Portlet Specification 2.0 (approved):

  http://www.jcp.org/en/jsr/detail?id=286
- JSR 301: Portlet Bridge Specification for JavaServer Faces (draft):

  http://jcp.org/en/jsr/detail?id=301
- IBM developerWorks: What's new in the Java Portlet Specification V2.0 (JSR 286)?

  http://www.ibm.com/developerworks/websphere/library/techarticles/0803_hepper/0803_hepper.html
- IBM developerWorks: WebSphere Portal Zone:

  http://www.ibm.com/developerworks/websphere/zones/portal

19.3 Enterprise Service Bus

Consider the example of the donate vacation function used in this document, which allows a user to donate vacation time in the increments of days. What if a related application wants to use this function, but it allows the user to submit the vacation in increments of hours? What if there are multiple applications that use increments of hours, but one is based on Web services and one is based on messaging? How do you handle these *disconnects* without writing unique custom code for each situation? The answer is an enterprise service bus (ESB).

An ESB is an infrastructure that connects dissimilar application requests, performing various operations to resolve the differences:

- **Message transformation**, taking a message format from the service requester and changing it to match the message format required by the actual service. These transformations can include field name changes, conversion between different units (such as hours to days), format changes (from text to XML for example), and field additions or field deletions.
- **Protocol conversion**, converting from one service technology, such as JMS, to another service technology, such as Web services or EJBs.
- **Endpoint routing**, delivering the service request to different instances of the back-end service based on user-defined criteria, such as the identity of the requester.
- **Quality of service conversion**, such as changing a username and password security token on the service request into a Security Assertion Markup Language (SAML) security token required by the actual service.

ESBs represent a merging of technologies that have been available for years supporting messaging (JMS) and Web service environments. These technologies collectively enable the mediation of interaction between application components that have a disconnect.

The key concept is that the disconnects are addressed within the ESB:

- Without the ESB, these disconnects are implemented uniquely in each application, leading to the problem of *interface creep*, where the interface code and path length becomes quite large because of all the permutations that must be supported. This also causes a maintenance issue, because a change in one application causes changes in many other applications.
- With the ESB, these changes are implemented once in the ESB, without changes to the service requester and the back end application. When an application changes, a single-associated change is made in the ESB, which then makes the appropriate alteration to the service request to connect to the back end application.

IBM provides several products that include ESB functionality:

- **WebSphere Enterprise Service Bus v6.1** provides an ESB capability implemented on top of WebSphere Application Server v6.1. This product is well suited for environments that already have a J2EE infrastructure or are heavily focused on Web services and messaging (JMS).
- **WebSphere Message Broker v6.1** provides a non-J2EE ESB implementation that provides a broad set of connectivity both to Web services and messaging (JMS), as well as legacy connections such as FTP, sockets, and base messaging (non-JMS).
- **WebSphere Datapower Integration Appliance XI50** provides a hardware based ESB capability.

The service integration bus (SIB) in WebSphere Application Server v6.1/v7.0 can also be described as an ESB, although typically its function is somewhat limited in comparison to the foregoing products.

IBM has multiple ESB products because ESB is an architecture: The concepts and capabilities described by the ESB architecture are applicable to a broad range of environments, and therefore more than one product implementation is appropriate. In fact, there are many scenarios where you can choose to incorporate two or more of the aforementioned ESB products. One common usage pattern includes the following two scenarios:

- The Datapower appliance in a DMZ zone provides secure *edge of network* ESB support for service requests from external zones, such as converting the user-identity token from an external format to an internal format.

- The WebSphere Enterprise Service Bus in the core J2EE infrastructure provides the manipulation of the actual message and service requests needed to map formats from several applications into the expected format required by the back end service.

Explore resources are available at the following Web sites:

- WebSphere Enterprise Service Bus product page:

  `http://www.ibm.com/software/integration/wsesb`
- *Getting Started with WebSphere Enterprise Service Bus V6*, SG24-7212:

  `http://www.redbooks.ibm.com/abstracts/sg247212.html`
- WebSphere Message Broker product page:

  `http://www.ibm.com/software/integration/wbimessagebroker`
- WebSphere DataPower Integration Appliance XI50 product page:

  `http://www.ibm.com/software/integration/datapower/xi50`

19.4 Business Process Execution Language

Portals address the need to aggregate applications *on the glass*. What if you have the requirement to aggregate application elements before reaching the end user?

Consider the example where you are able to look up a GPS coordinate anywhere in the world. What if you wanted to determine the current US State Departments travel advisories for the country represented by that coordinate? You would need to perform the following actions:

- Potentially convert the GPS coordinate from one representation to another because there are several ways of representing the GPS coordinates.
- Invoke an application that takes the GPS coordinate as an input and returns a country.
- Invoke another application that takes the country as an input and returns the US State Department travel advisories.

You could implement this integration *on the glass* using portlets and portlet messaging. But what if you wanted to make this an actual application that the user could invoke? Put in the GPS coordinates (in any valid GPS coordinates) and get as output the US State Department travel advisories? The answer is to implement a solution that implements the Web Services Business Process Execution Language (WS-BPEL) specification, which provides for the coordinated execution of service elements.

The IBM product solution that implements WS-BPEL is WebSphere Process Server v6.1, which runs on Application Server v6.1. The associated development tool is WebSphere Integration Developer v6.1, which is built on Rational Application Developer v7.0.

WebSphere Integration Developer provides a graphic development environment that allows the user to incorporate application services—defined both in WSDLs for traditional Web services and in service component architecture (SCA) files, for direct Java invocation and messaging invocation—into an organized calling sequence called a process. These processes can then be externally invoked through Web services, messaging, or direct Java.

Processes can be short running or long running:

- A short running process is similar in concept to a session EJB: All processing is done synchronously, and the process and associated application service elements all complete within a single, short-lived thread of execution. This type of process supports normal transactional coordination (commit and rollback).
- A long running process can take seconds, days, weeks, months, or years to complete. The overall process manager records the state of the process after invoking each application service element, and potentially can suspend the process if it reaches an element that is waiting for an external signal, such as a reply message from messaging, JMS, or a Web service request that contains a specific identifier called a correlation ID that indicates for which instance of the process that message is intended.

  A long-running process does not support normal transactional coordination, and instead you must use compensation (which is described next).

Note that a process needs to contain more than the calling sequence for normal execution. It also needs to control what happens when something goes wrong:

- Application services can return an exception condition, called a fault. Processes can be configured with fault handlers, allowing the developer to define alternate sequences that execute when a fault is detected, whether on an individual application-service element, on a group of application-service elements contained in a section of the process, or on the overall process.
- If an overall process fails (either due to a fault or through normal process navigation to a process element that indicates an end failing state), the process can attempt to reverse the results of the previous steps through two mechanisms:
  - It can use standard-transactional rollback if the process is short running, and if the application-service elements support transactional rollback.

- It can use a special mechanism called compensation, where the process manager invokes specific application elements to reverse the previously completed steps.

  To use compensation, each application-service element must define two different operations: One for the normal (forward) invocation and one for the compensation (reverse) invocation.

  For example, if the forward operation deletes a data record, it could return a key to the process manager, representing a record in a temporary database that contains the deleted information. If compensation needs to be invoked, the process manager could pass this key to the compensation operation, which would use this key to retrieve the record from the temporary database and recreate the record.

IBM ships its WS-BPEL implementation with a feature called the Human Task Manager. This feature, when invoked from a process, adds an item to a task list, and then suspends the process. A user, based on user-defined roles, can view a list of available tasks and then claim and complete the specific task. This causes the process to be reactivated, and to continue to the next application-service element in the process.

We believe that this is a key element in a process-oriented or workflow-oriented solution: Business processes are composed of both automated and manual (human task) steps. Any product that provides process automation must support the integration of both the automated and manual steps.

WS-BPEL is an Oasis managed specification, and the 2.0 version was released in April, 2007. The IBM implementation currently supports WS-BPEL 1.1, but it does contain many of the new WS-BPEL 2.0 features.

Explore references are available at the following Web sites:

- OASIS Web Services Business Process Execution Language Version 2.0:

  http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

- IBM developerWorks: WS-BPEL Extension for People:

  http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/

- IBM developerWorks: WebSphere Process Server and WebSphere Integration Developer:

  http://www.ibm.com/developerworks/websphere/zones/businessintegration/wps/wps.html

- *Patterns: Building Serial and Parallel Processes for IBM WebSphere Process Server V6*, SG24-7205:

  http://www.redbooks.ibm.com/abstracts/sg247205.html

19.5 Service oriented architecture

The preceding products and technologies provide a wide range of function:

- Application infrastructures, such as Java EE (and .NET).
- Application connectivity technologies such as Web services, messaging (both JMS and non-JMS), FTP, and sockets.
- Portals that provide *on the glass* integration.
- Enterprise Service Buses (ESBs) that manage the "disconnects" between application-service elements.
- Process engines based on Business Process Execution Language (WS-BPEL) that provide for the coordinated execution of application service elements.

In the previous chapters in this book we also touched on basic Web services capabilities such as UDDI registries, common data interchange formats such as XML and Service Data Objects (SDOs), and invocation frameworks such as Service Component Architecture (SCA).

How do you combine these technologies into a comprehensive architecture that addresses the needs of an enterprise computing environment, not only for the runtime environment but also for the full life cycle needs, starting from the business-level definitions and ending with the actual deployed technical implementations? The answer is to use a service oriented architecture (SOA).

It is important to view SOA as an evolutionary step in defining a distributed computing environment rather than a revolutionary step. SOA throws nothing away. Instead, SOA defines an organized architecture around the existing standards and technologies that relate to invoking application service elements or services.

SOA is related to services as the Internet or intranet are related to TCP/IP networks:

- Services and TCP/IP networks represent the discrete but unmanaged elements. Without these elements, you could not have an SOA environment, and you could not have an Internet or intranet.
- SOA and an Internet or intranet represent the managed view of these discrete elements to form an overall enterprise environment.

  For an Internet or intranet these environments do not occur automatically; instead, support organizations and standards are put into place to coordinate and manage the changes that occur over time.

The same is true for an SOA implementation: Organizations and internal standards are critical to ensure that the technologies are applied and maintained in a manner that provides a robust and reliable infrastructure.

Explore references are available at the following Web sites:

- IBM developerWorks: IBM SOA Foundation: An architectural introduction and overview:

  `http://www.ibm.com/developerworks/webservices/library/ws-soa-whitepaper`

- IBM developerWorks: SOA and Web services:

  `http://www.ibm.com/developerworks/webservices`

- *Enabling SOA Using WebSphere Messaging*, SG24-7163:

  `http://www.redbooks.ibm.com/abstracts/sg247163.html`

- *Patterns: SOA Foundation Service Creation Scenario*, SG24-7240:

  `http://www.redbooks.ibm.com/abstracts/sg247240.html`

Part 6

Appendixes

Jump start

This appendix provides the instructions needed to *jump start* individual chapters, as described in Chapter 1, “Introduction” on page 3.

Jump-start can be defined as follows: *to start or restart rapidly or forcefully*

> From Merriam-Webster on-line dictionary: http://www.merriam-webster.com

The intention is that most users will implement the Experience sections of this book in a sequential manner: Starting with the chapters in Part 1, “Preliminary activities” on page 1 and following in order.

However, readers who are already familiar with parts of Java EE might want to skip chapters or access them in a random manner. For those readers, this book provides this jump start section to allow readers to quickly configure WASCE, the WASCE IDE, and the WASCE IDE workspace each time they access a chapter out of or order or randomly.

A.1 Common jump start activities

These steps should be completed for all jump starts:

1. Complete all the activities in Chapter 2, “Install and configure software” on page 25.

   These are one-time steps and do not need to repeated if they have already been completed.

2. Complete all the activities in Chapter 3, “Configure the development environment” on page 49.

> **Reminder!**
>
> The **ExperienceJavaEE Server** test server must be running to complete the subsequent steps in this appendix.

3. Complete all the activities in Chapter 4, “Prepare the legacy application” on page 77.

   These are one-time steps and do not need to repeated if they have already been completed.

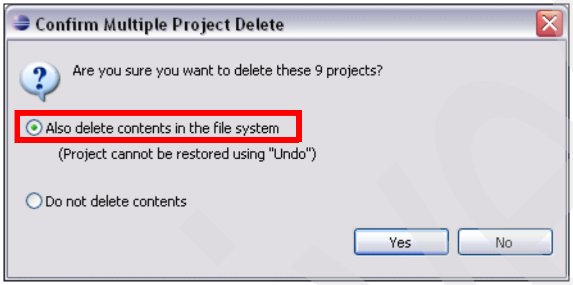
> **Off course?**
>
> If you receive an error message similar to the following one when deploying the `VacationPool` or the `DonatePool`, it means that you have already configured this resource and you can ignore this error:
>
> ```
> Error: Unable to distribute tranql-connector-derby-client-xa-1.4.rar:
> Module console.dbpool/DonatePool/1.0/rar already exists in the server.
> Try to undeploy it first or use the redeploy command.
> ```
>
> If you want to overwrite the existing configuration, repeat this command with the redeploy option instead of the deploy option:
>
> ```
> deploy -u system -p xxxxxxxx redeploy C:\7639code\plans\Vacat...
> ```

4. If you have already completed some chapters in the book, you have to clear the workspace:

   – In the Servers view, right-click **ExperienceJavaEE Server**, and select **Add and Remove Projects**.

     - In the Add and Remove Projects pop-up, select **DonateEAR** in the right pane, and click **Remove** to move it to the right pane.
     - Click **Finish**.

- Back in the Servers view, right-click **ExperienceJavaEE Server**, and select **Publish**. The deployment takes place in the background, with the status displayed in the lower right task bar. Wait for the task to complete.
- In the Project Explorer, ensure that all the projects are contracted (not expanded).
- In the Project Explorer, select all projects, right-click and select **Delete**.
- In the Confirmation Multiple Project Delete pop-up, select **Also delete contents in the file system**, and click **Yes**.



Confirm Multiple Project Delete

Are you sure you want to delete these 9 projects?

● Also delete contents in the file system
(Project cannot be restored using "Undo")

○ Do not delete contents

A.2 Chapter specific jump start steps

Important!

Complete only the following steps that apply for your situation:

- You do not have to complete a step if you have already successfully completed the referenced chapter.
- You do not have to complete a step if it is part of a chapter after the one that you are jump starting.

For example, if you are jump starting Chapter 8, “Create the Web front end” on page 221, you only have to complete steps 3, 4, 5, and 6.

Remember the **system** user ID and its password (default is **manager**) that were used to install WASCE. You have to provide the user ID and password in the commands.

1. For all chapters after Chapter 5, “Create the JPA entities” on page 99, import the starting archive for the chapter into the WASCE IDE:

   – Ensure that the WASCE IDE is started (as described in 3.3, “Create the WASCE IDE workspace” on page 53).

   – If you have any existing projects, remove them from the server and delete them from the workspace as described in A.1, “Common jump start activities” on page 558.

   – Extract the zip from the `C:\7639code\jumpstart` directory for the chapter **prior** to the one you are jump-starting into the root directory for your workspace (**`C:\Workspaces\WASCE21expjee`**). The archive file names captured at the end of each chapter are shown in Table 19-1.

Table 19-1 Archive files at the end of each chapter

Chapter	Title (abbreviated)	Archive at end of chapter
5	JPA entities	`ch5-jpa.zip`
6	Session facade	`ch6-facade.zip`
7	Session bean	`ch7-session.zip`
8	Web front-end	`ch8-web.zip`
9	Application client	`ch9-appclient.zip`
10	Core security	`ch10-security.zip`
11	Web service create	`ch11-webservice.zip`
12	Web service security	`ch12-webservice-security.zip`
13	MDB create	`ch13-mdb-create.zip`
14	MDB pub/sub	`ch14-mdb-pubsub.zip`
15	MDB security	`ch15-mdb-security.zip`
16	Web 2.0	`ch16-web20.zip`

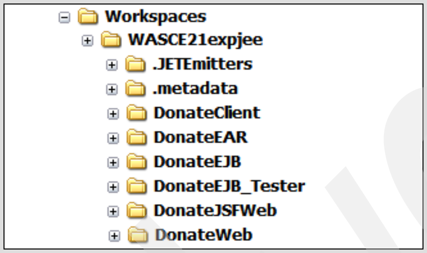
To jump start a chapter, select the archive file at the end of the previous chapter, for example, to jump start **chapter 9**, select the **ch8-web.zip** file.

For Chapter 16, “Develop a Web 2.0 client” on page 455 you can start with the `ch10-security.zip`, or any file after that.

Jump start example:

For example, the archive file to jump start Chapter 10, “Implement core security” on page 285 is `C:\7639code\jumpstart\ch9-appclient.zip`.

If you followed the instructions in 3.3, “Create the WASCE IDE workspace” on page 53, the root directory would be **`C:\Workspaces\WASCE21expjee`** and after extracting the zip file the resulting directory would look similar to the following.



```
⊟ Workspaces
  ⊞ WASCE21expjee
    ⊞ .JETEmitters
    ⊞ .metadata
    ⊞ DonateClient
    ⊞ DonateEAR
    ⊞ DonateEJB
    ⊞ DonateEJB_Tester
    ⊞ DonateJSFWeb
    ⊞ DonateWeb
```

The exact subdirectories would vary depending on which chapter you are jump starting.

- From the WASCE IDE action bar, select **File** → **Import**.
- In the Import/Select pop-up, select **General** → **Existing Projects into Workspace** and click **Next**.
- In the Import/Import Projects pop-up:
  - Next to Select root directory, click **Browse**.
  - At the resulting Browser for Folder pop-up, navigate to the root directory of the workspace (which should be **`C:\Workspaces\WASCE21expjee`**) and click **OK**.
- Back in the Import/Import Projects pop-up, next to Projects click **Select All**.
- Click **Finish**, and wait until the workspace is built.

2. For all chapters after Chapter 5, “Create the JPA entities” on page 99, configure the JPA properties for the DonateJPAEmployee and DonateJPAFund projects:
   - In the Project Explorer, right-click **DonateJPAEmployee** and select **Properties**.
     - In the Properties pop-up, select **JPA** (on the left), and set Connection to **Vacation** (on the right).
     - Click **OK**.

- In the Project Explorer, right-click **DonateJPAFund** and select **Properties**.
  - In the Properties pop-up, select **JPA**, and set Connection to **Donate**.
  - Click **OK**.

Behind the scenes:

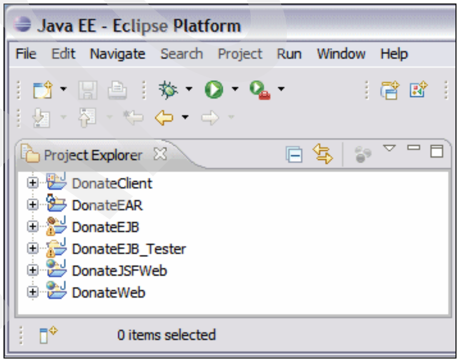
The JPA properties are stored in the workspace `.metadata` subdirectory and not under the actual project directory. For example, for DonateJPAEmployee the properties are stored in:

```
<workspace>\.metadata\.plugins\org.eclipse.core.resources\.projects
                              \DonateJPAEmployee\.indexes\properties.index
```

Therefore, they are not included as part of the standard project archive format, and must be reconfigured after importing a project archive.

Note that you get JPA warnings in the Problems view if the connection is not specified or is not active. The application does run anyway, but it is advisable to take care of the warnings.

3. If you see any errors:
   - From the WASCE IDE action bar, select **Project** → **Clean**.
   - At the Clean pop-up, select **Clean all projects**.
   - Click **OK**.
   - The projects exist now in the workspace, with possibly some errors and warnings (see **Errors and warnings** box next). The exact errors and warnings vary depending on which chapter you are jump starting.



Errors and warnings:

- If you have no connection to the two databases, you get warnings:

  ```
  No connection specified for project.  No database-specific
  validation will be performed.
  ```

  See step 2 on page 561 for a solution.

- If you have the connections defined, but you have not previously executed the steps in 5.9.4, “Run the JUnit test” on page 154 (for testing the `Fund` entity), three errors will be displayed indicating that the column and schema information cannot be fund for the `FUND` table:

  ```
  Column "balance" cannot be resolved
  Column "name" cannot be resolved
  Schema "DONATE" cannot be resolved for table "FUND"
  ```

  The errors exist at this point because the `FUND` table has not yet been created and configured. We will do this in the next step.

- Warnings of the nature `Type safety: Unchecked cast from List to List<Fund>`, can be eliminated by adding the annotation **@SuppressWarnings("unchecked")** before the method with the error.

  This annotation can be added using the Eclipse quick fix capability as used in 5.10, “Handle errors and warning” on page 156.

- Warnings of the nature `The serializable class Xxx does not declare a static final serialVersionUID field of type long`, can be eliminated by adding the annotation **@SuppressWarnings("serial")** before the class.

4. Warnings of the nature: `No grammar constraints (DTD or XML schema) detected for the document ...` can be disabled globally (for all schema files that lack grammar constraints):
   - From the action bar, select **Window** → **Preferences**.
   - In the Preferences pop-up perform the following:
     - On the left, select **XML** → **XML Files**.
     - On the right (at the resulting XML Files pane), change the Validating files/Indicate when no grammar is specified from **Warning** to **Ignore**.
     - Click **OK**.
   - The warnings should be eliminated after the next build/clean.

Off course?

Your workspace might still have errors if you are importing the project interchange on Linux or if you use an installation structure that is different than described in Chapter 2, “Install and configure software” on page 25.

- `Unbound classpath container: 'JRE System Library [Java50]' in project 'DonateClient'`

  This error results if the name of the IBM SDK/JRE is not Java50. The easiest way to resolve this problem is to rename the IBM SDK/JRE name to Java50:

  – In the action bar, select **Window** → **Preferences**.

  – In the Preferences pop-up, on the left select **Java** → **Installed JREs**.

  – In the Preferences pop-up, on the right, highlight the entry for the IBM SDK/JRE (`C:\Program Files\IBM\Java50` for Windows and `/opt/ibm/java2-i386-50` for Linux) and click **Edit**.

  – In the resulting Edit JRE pop-up:

    - Change the JRE Name to **Java50**.
    - Click **Finish**.

  – Click **OK**.

- `Project 'DonateEJB_Tester' is missing required library: 'C:/IBM/WASCE21/repository/org/apache/geronimo/framework/geronimo-security/2.1.1/geronimo-security-2.1.1.jar'`

  This error results if WASCE installation on this system is different than on the system where this project interchange was created. This problem is resolved by editing the project classpath:

  – In the Project Explorer, right-click the DonateEJB_Tester project and select **Properties**.

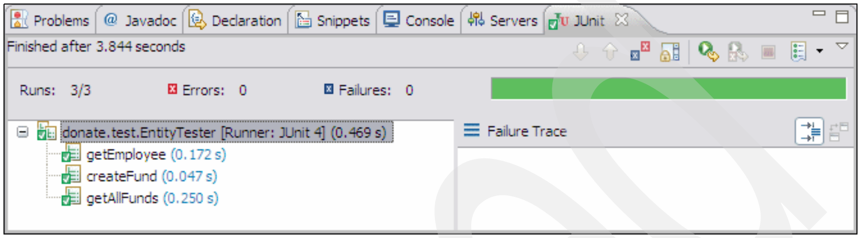
    - In the resulting pop-up, on the left select **Java Build Path**, and on the right select the **Libraries** tab.
    - On the right, highlight `geronimo-security-2.1.1.jar` and click **Edit**.
    - Navigate to the correct location for `geronimo-security-2.1.1.jar and` click **Open** (**OK** on Linux)**.**

  – Click **OK** to save and close the Properties.

5. For all chapters after Chapter 5, “Create the JPA entities” on page 99, run the `EntityTester` JUnit test to create the `Donate` database and the `FUND` table:

   – In the Project Explorer, right-click **EntityTester** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.

   – In the JUnit view, verify that the test completed successfully (0 failures and 0 errors).



```
Finished after 3.844 seconds
Runs:  3/3          Errors:  0          Failures:  0
donate.test.EntityTester [Runner: JUnit 4] (0.469 s)
    getEmployee (0.172 s)
    createFund (0.047 s)
    getAllFunds (0.250 s)
```

Behind the scenes:

The JUnit test fails with a `NullPointerException` if you did not correctly add the user libraries:

- Review 3.8, “Configure user libraries in Eclipse” on page 71 to verify the definition of the user libraries.
- Review 5.9.1, “Create the JUnit project” on page 149 to ensure that the user libraries are properly defined (with the proper JAR files) in the DonateEJP_Tester properties.

Extra credit:

The `FUND` table can now be defined in the `Donate` database, and the three errors referencing the missing columns and table should be resolved, if you refreshed the Donate connection and then validated DonateJPAFund (or cleaned the workspace).

6. For all chapters after Chapter 10, “Implement core security” on page 285, copy and edit the required user ID and password files:

   – Copy `C:\7639code\security\experiencejavaeeusers.properties` into `C:\IBM\WASCE21\var\security`.

   – Optionally edit this file and change all occurrences of **`password`** to the password value that you want to use.

- Copy C:\7639code\security\experiencejavaeegroups.properties into C:\IBM\WASCE21\var\security.
- Configure the necessary security realm in WASCE by running the following command in an operating system prompt from the C:\IBM\WASCE21\bin directory:

  ```
  deploy -u system -p manager deploy
        C:\7639code\plans\SecurityRealm-plan.xml
  ```

  The command should complete with the following message:

  ```
  Deployed console.realm/ExperienceJavaEERealm/1.0/car
  ```

Passwords:

If you change the password values in experiencejavaeeusers.properties, you have to update several code fragments:

- For all chapters after Chapter 10, “Implement core security” on page 285, update the passwords in DonateBeanTester.

  Edit **DonateBeanTester** (in DonateEJB_Tester/src/donate.test) and change the SECURITY_CREDENTIALS value in the init method to match the password used previously:

  ```
  props.put(Context.SECURITY_PRINCIPAL, "DNARMSTRONG");
  props.put(Context.SECURITY_CREDENTIALS, "password");
  props.put("openejb.authentication.realmName","ExperienceJavaEERealm");
  ```

- For all chapters after Chapter 12, “Implement security for the Web service” on page 367, update the passwords in DonateWSClientWeb.

  Edit **ClientServlet** (in DonateWSClientWeb/src/client) and change the two PASSWORD_PROPERTY values in the doGet method to match the passwords used previously.

  ```
  ((BindingProvider)bean).getRequestContext()
        .put(BindingProvider.USERNAME_PROPERTY, "DNARMSTRONG");
  ((BindingProvider)bean).getRequestContext()
        .put(BindingProvider.PASSWORD_PROPERTY, "password");
  // same for DGADAMS
  ```

- For all chapters after Chapter 15, “Implement security for messaging” on page 445, update the passwords in DonateMDBClient:

  Edit **Main** (in DonateMDBClient/appClientModule) and change the password value to match the password used previously.

  ```
  //* Added for MDB Security
  toutMessage.setStringProperty("userid", "DNARMSTRONG");
  toutMessage.setStringProperty("password","password");
  ```

7. For all chapters after Chapter 13, “Create the message-driven bean” on page 405, configure the necessary JMS resources in WASCE by running the following command in an operating system prompt from the `C:\IBM\WASCE21\bin` directory:

```
deploy -u system -p manager deploy
C:\7639code\plans\JmsResources-plan.xml
..\repository\org\apache\geronimo\modules\geronimo-activemq-ra\2.1.1\ger
onimo-activemq-ra-2.1.1.rar
```

   The command should complete with the following message:

```
Deployed console.jms/DonateJMSResources/1.0/rar
```

8. For all chapters after Chapter 5, “Create the JPA entities” on page 99, publish the DonateEAR application.

   – Ensure that the ExperienceJavaEE Server is started (as described in 3.6, “Start the ExperienceJavaEE Server test server” on page 66).

   – In the Servers view, right-click **ExperienceJavaEE Server**, and select **Add and Remove Projects**.

     - In the Add and Remove Projects pop-up, select **DonateEAR** in the left pane, and click **Add** to move it to the right pane.

     - Click **Finish**.

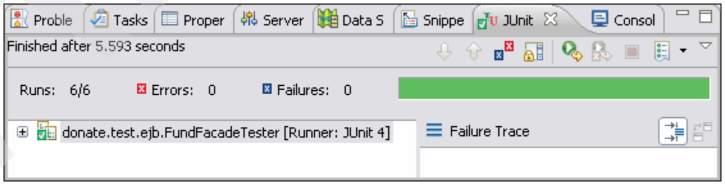
   – Back in the Servers view, right-click **ExperienceJavaEE Server** and select **Publish**.

     The deployment takes place in the background, with the status displayed in the lower right task bar. Wait for the task to complete.

9. For all chapters after Chapter 5, “Create the JPA entities” on page 99, create the `DonationFund` record in the `Donate` database.

   – In the Project Explorer, right-click **FundFacadeTester** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.

   – In the JUnit view, verify that the test completed successfully (0 failures and 0 errors).



Finished after 5.593 seconds

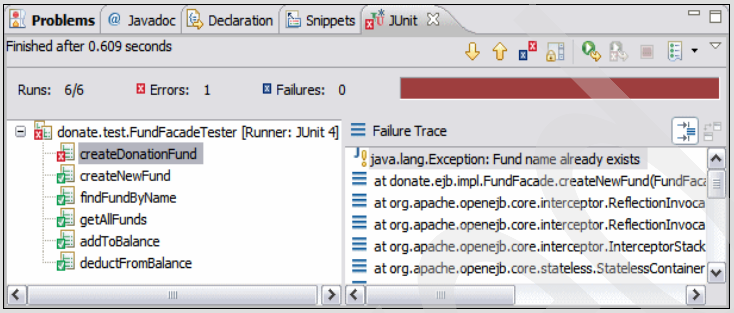
Runs: 6/6   Errors: 0   Failures: 0

donate.test.ejb.FundFacadeTester [Runner: JUnit 4]

Failure Trace

Off course?

The JUnit test fails with a fund already exists exception if the `FundFacadeTester` has been run before:



Finished after 0.609 seconds

Runs: 6/6 Errors: 1 Failures: 0

- donate.test.FundFacadeTester [Runner: JUnit 4]
  - createDonationFund
  - createNewFund
  - findFundByName
  - getAllFunds
  - addToBalance
  - deductFromBalance

```
java.lang.Exception: Fund name already exists
at donate.ejb.impl.FundFacade.createNewFund(FundFaca
at org.apache.openejb.core.interceptor.ReflectionInvoca
at org.apache.openejb.core.interceptor.ReflectionInvoca
at org.apache.openejb.core.interceptor.InterceptorStack
at org.apache.openejb.core.stateless.StatelessContainer
```

You can ignore this error because the `DonationFund` can only be created once. However, only the `createDonationFund` test should fail.

10. For all chapters after Chapter 7, “Create the Donate session bean for the business logic” on page 209, verify the Donate application.

    – In the Project Explorer, right-click **DonateBeanTester** (in `DonateEJB_Tester/src/donate.test`) and select **Run As** → **JUnit Test**.

    – In the JUnit view, verify that the test completed successfully (0 failures and 0 errors).

Behind the scenes:

Executing this JUnit test verifies the `donateToFund` method in the `DonateBean` session EJB. This is the core business logic element that is accessed in the various interfaces used in this book: Web, client, Web services, and JMS.

Congratulations! The jump start has been successfully completed.

A.3 Creating resources in a new server

At this point of time, you might have noticed the need to build new servers for various reasons. You might also have noticed that going through and defining all of the resources in the server starts to become tedious.

Here is a short list of the steps required to set up the application databases and the deployment plans in the WASCE servers.

If you just completed a jump start in preceding sections, you have executed some or all of these commands, and you do not need to execute these again for the jump start. Proceed as follows:

1. Start the WASCE server if it is not already running.
2. Create the database tables and deploy the data source according to the database provider:
   - Create the `Vacation` and `Donate` databases as described in Chapter 4, “Prepare the legacy application” on page 77.
   - Create the database pools by running these commands from the `<WASCE_HOME>\bin` directory:

```
deploy -u system -p manager deploy
   C:\7639code\plans\VacationPool-plan.xml
   ..\repository\org\tranql\tranql-connector-derby-client-xa\1.4
                     \tranql-connector-derby-client-xa-1.4.rar

deploy -u system -p manager deploy C:\7639code\plans\DonatePool-plan.xml
   ..\repository\org\tranql\tranql-connector-derby-client-xa\1.4
                     \tranql-connector-derby-client-xa-1.4.rar
```

3. Create the JMS resources by running the command:

```
deploy -u system -p manager deploy
   C:\7639code\plans\JmsResources-plan.xml
   ..\repository\org\apache\geronimo\modules\geronimo-activemq-ra
                            \2.1.1\geronimo-activemq-ra-2.1.1.rar
```

4. Create the security configuration:
   - Copy `C:\7639code\security\experiencejavaeeusers.properties` into `<WASCE_HOME>/var/security`.
   - Optionally edit this file and change the **password** values to your liking.
   - Copy `C:\7639code\security\experiencejavaeegroups.properties` into `<WASCE_HOME>/var/security`.

- Configure the necessary security realm in WASCE by running the following command in an operating system prompt from the `<WASCE_HOME>\bin` directory:

```
deploy -u system -p manager deploy
   C:\7639code\plans\SecurityRealm-plan.xml
```

Alternative for Windows:

For Windows, we provide the **`run-ExpJEE`** command file that performs these functions:

- Remove the plans if they exist already.
- Load database pools, JMS resources, and security realm plans.
- Copy users and groups to WASCE.
- Create the `Vacation` and `Donate` databases.
- Load the `Vacation` database with employees.

The command file is available as `C:\7639code\plans\run-ExpJEE.bat`.

B

Additional material

This Appendix provides additional material that you can download from the Internet as follows.

Locating the Web material

The Web material associated with this IBM Redbooks publication is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG247639`

Alternatively, you can visit the IBM Redbooks Web site at:

ibm.com/redbooks

After you access the Web site, select **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247639.

Using the Web material

The additional Web material that accompanies this publication includes the following files:

File name	*Description*
`7639code.zip`	ZIP file with sample code

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	40 GB minimum
Operating System:	Windows 2000, Windows XP, Linux
Processor:	1.5 GHz or higher
Memory:	1 GB or better

Refer to 2.2, “Hardware requirements” on page 27 and 2.3, “Software requirements” on page 28 for further information.

How to use the Web material

Download the sample code and extract the ZIP file into a suitable directory. Refer to 3.2, “Extract the sample code file” on page 52 for further instructions.

Instructions on how to use the jump start ZIP files to start at any chapter in the book are provided in Appendix A, “Jump start” on page 557.

Abbreviations and acronyms

AJAX	Asynchronous JavaScript Technology and XML
API	application programming interface
AST	Application Server Toolkit
BMP	bean managed persistence
CMP	container managed persistence
DOM	Document Object Mode
EGL	Enterprise Generation Language
EJB	Enterprise JavaBeans
ESB	Enterprise Service Bus
FAQ	frequently asked questions
FTP	File Transfer Protocol
GUI	graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IDE	integrated development environment
ITSO	International Technical Support Organization
J2EE	Java 2, Enterprise Edition
J2SE	Java 2, Standard Edition
JAAS	Java Authentication and Authorization Service
JACC	Java Authorization Contract for Containers
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML Web Services
JAXB	Java Architecture for XML Binding
JCA	Java Connector Architecture
JCP	Java Community Process
JDBC	Java Database Connectivity
JDK	Java Developer’s Kit
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JSR	Java Specification Request
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
LTPA	Lightweight Third Party Authentication
MDB	message-driven bean
MQ	message queue
MTOM	Message Transmission Optimization Mechanism
MVC	model view controller
OASIS	Organization for the Advancement of Structure
RDB	relational database
REST	Representational State Transfer
RHEL	Red Hat Enterprise Linux
RMI	Remote Method Invocation
RPC	remote procedure call
SAAJ	SOAP with Attachments API for Java
SAML	Security Assertion Markup Language

SCA	Service Component Architecture
SDO	Service Data Objects
SIB	service integration bus
SIP	Session Initiation Protocol
SOA	service oriented architecture
SOAP	Simple Object Access Protocol (also known as Service Oriented Architecture Protocol)
SQL	structured query language
SSL	Secure Sockets Layer
SVG	Salable Vector Graphics
SWAM	Simple WebSphere Authentication Mechanism
SWT	Standard Widget Toolkit
TCP/IP	Transmission Control Protocol/Internet Protocol
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
URI	uniform resource identifier
URL	uniform resource locator
UTC	Universal Test Client
VM	virtual machine
VML	Vector Markup Language
WS-BPEL	Web Services Business Process Execution Language
WS-I	Web Services Interoperability
WS-RM	Web Services Reliable Messaging
WSDL	Web Services Description Language
WSIL	Web Services Inspection Language
WTP	Web Tools Platform
WYSIWYG	what you see is what you get
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks publications” on page 577. Note that some of the documents referenced here might be available in softcopy only.

- *Experience J2EE! Using WebSphere Application Server V6.1*, SG24-7297
- *Rational Application Developer V7.5 Programming Guide*, SG24-7672
- *Rational Application Developer V7 Programming Guide*, SG24-7501
- *WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0*, SG24-7611
- *Web Services Feature Pack for WebSphere Application Server V6.1*, SG24-7618
- *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257
- *Building Dynamic Ajax Applications Using WebSphere Feature Pack for Web 2.0*, SG24-7635
- *Building SOA Solutions Using the Rational SDP*, SG24-7356
- *WebSphere Application Server V6.1: Planning and Design*, SG24-7305
- *WebSphere Application Server V6 Security Handbook*, SG24-6316
- *WebSphere Application Server V6.1: System Management and Configuration*, SG24-7304
- *Getting Started with WebSphere Enterprise Service Bus V6*, SG24-7212
- *Patterns: Extended Enterprise SOA and Web Services*, SG24-7135

Online resources

The following Web sites are a summary of those listed previously in the Learn! and Explore! sections of this publication:

- WebSphere and Rational software:

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

- WebSphere Information Center:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

- IBM Education Assistant:

  http://www.ibm.com/software/info/education/assistant

- developerWorks:

  http://www.ibm.com/developerworks

- alphaWorks:

  http://www.ibm.com/alphaworks

- Eclipse:

  http://www.eclipse.org

- Web Tools Platform:

  http://www.eclipse.org/webtools

- Eclipse plug-in central:

  http://www.eclipseplugincentral.com

- Sun Java:

  http://java.sun.com

- Java Community Process:

  http://www.jcp.org

- Apache Derby database:

  http://db.apache.org/derby

- Dojo toolkit:

  http://dojotoolkit.org/

- OASIS:

  http://www.oasis-open.org

- mozilla.org:

  http://www.mozilla.org

- Jython:

  http://www.jython.org
- Apache Tuscany:

  http://incubator.apache.org/tuscany
- Apache Rampart:

  http://ws.apache.org/rampart
- Wikipedia:

  http://en.wikipedia.org
- Web Services Interoperability Organization:

  http://www.ws-i.org

How to get IBM Redbooks publications

You can search for, view, or download Redbooks publications, Redpapers, Hints and Tips, draft publications, and Additional materials, as well as order hardcopies of Redbooks publications or CD-ROMs, at the following Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads:

ibm.com/support

IBM Global Services:

ibm.com/services

Index

IBM

Redbooks

Experience Java EE! Using WebSphere Application Server Community Edition 2.1

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Experience Java EE!
Using WebSphere Application Server Community Edition 2.1

On-ramp to Java EE development

Including EJB 3.0, Web services, JMS and Web 2.0

Detailed hands-on tutorials

This IBM Redbooks publication is a hands-on guide to developing a comprehensive Java EE application using WebSphere Application Server Community Edition v2.1 (WASCE), including core functions, security, Web services, and messaging.

Novice users are thus able to experience Java EE, and advance from theoretical knowledge gained by reading introductory material to practical knowledge gained by implementing a real-life application.

Experience is one stage in gaining and applying knowledge, but there are additional stages needed to complete the knowledge acquisition cycle. This book also helps in those stages:

- Before experiencing Java EE, you learn about the base specifications and intellectual knowledge of Java EE through brief descriptions of the theory and through links to other information sources.
- After experiencing J2EE, you explore advanced Java EE through previews of advanced topics and links to other information sources.

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

SG24-7639-00 ISBN 073843230X