IBM
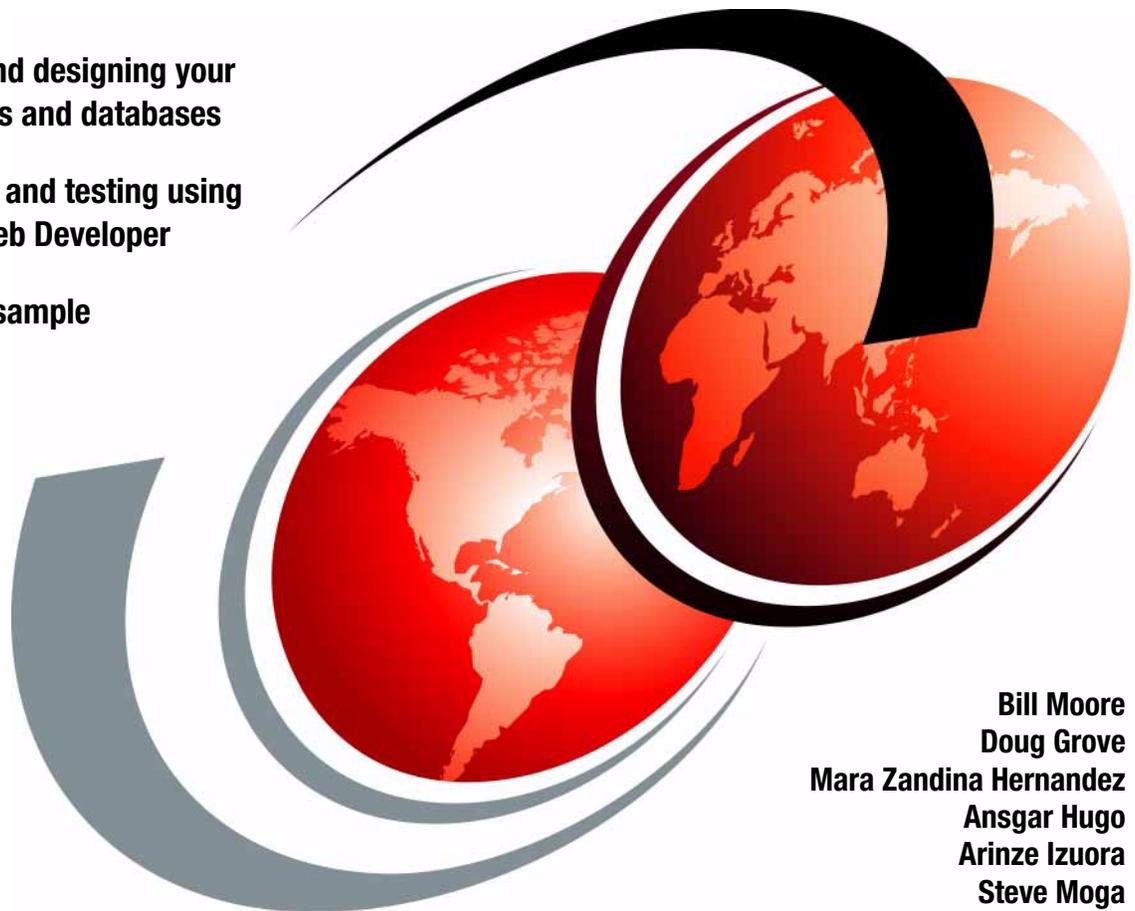
WebSphere Application Server - Express V6
Developers Guide and Development Examples

Planning and designing your applications and databases

Developing and testing using Rational Web Developer

Building a sample application

Bill Moore
Doug Grove
Mara Zandina Hernandez
Ansgar Hugo
Arinze Izuora
Steve Moga

Redbooks

IBM

International Technical Support Organization

**WebSphere Application Server - Express V6
Developers Guide and Development Examples**

October 2005

First Edition (October 2005)

This edition applies to Version 6 of WebSphere Application Server - Express.

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 illustrates 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. 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 IBM's application programming interfaces.

xiii

Trademarks

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

®	Informix®	S/390®
AIX®	iSeries™	SAA®
Balance®	Lotus®	server®
CICS®	Net.Data®	Tivoli®
ClearCase®	OS/390®	TME®
Cloudscape™	Perform™	WebSphere®
DB2 Universal Database™	pSeries®	Workplace Collaborative
DB2®	Rational Rose®	Learning™
Domino®	Rational Unified Process®	Workplace™
Eserver®	Rational®	XDE™
Eserver®	Redbooks (logo)™	z/OS®
ibm.com®	Redbooks (logo) ™	zSeries®
IBM®	Redbooks™	
IMS™	RUP®	

The following terms are trademarks of other companies:

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

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

Intel, 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, and service names may be trademarks or service marks of others.

Preface

This IBM Redbook is a practical guide for developing Web applications using the Rational Software Development Platform. We use the Rational Web Developer development environment that is provided as part of WebSphere Application Server - Express V6 to develop a sample Web application targeted to the WebSphere Application Server - Express runtime platform. We discuss a sample scenario based on realistic requirements for small and medium-sized customers, and provide a guide for the development of this scenario.

Our focus is on describing a simple process that allows nontechnical readers to understand and participate in the development of Web applications using Rational Web Developer. Our target runtime environment is the Express Application Server so we use the Rational Web Developer development environment that is part of the WebSphere Application Server - Express installation. WebSphere Application Server - Express, V6 offers a robust, easy-to-use Eclipse technology-based development environment that allows developers to create, build, and maintain dynamic Web sites, applications, and Web services. The development tools offer the same development capabilities as Rational Web Developer with the exception that they are restricted to deployment only to WebSphere Application Server - Express.

Note: Our redbook is based on the existing Redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 which was written for WebSphere Application Server - Express V5. Our example application was initially developed for this redbook. We do not redevelop the example from scratch in our new Redbook. Instead, we concentrate on examining new features of the WebSphere Application Server - Express V6 runtime and of the Rational Web Developer development environment.

During the development of this redbook, we mainly used the Rational Web Developer development environment provided with WebSphere Application Server - Express, but you can easily migrate to more advanced configurations of WebSphere Application Server and Rational Software Development Platform when you require more sophisticated capabilities. As with all applications developed with WebSphere Application Server - Express V6, our redbook examples can run without alteration in more advanced configurations of WebSphere Application Server, and can be developed in other configurations of the Rational Software Development Platform such as Rational Web Developer and Rational Application Developer.

This means that your investment in the skills acquired with WebSphere Application Server - Express is protected when migrating to these other products.

The team that wrote this redbook

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

Bill Moore is a WebSphere specialist at the International Technical Support Organization, Raleigh Center. He writes extensively and teaches classes on WebSphere and related topics. Before joining the ITSO, Bill was a Senior AIM Consultant at the IBM Transarc laboratory in Sydney, Australia. He has 19 years of application development experience on a wide range of computing platforms and using many different coding languages. He holds a Master of Arts degree in English from the University of Waikato, in Hamilton, New Zealand. His current areas of expertise include application development tools, object-oriented programming and design, and e-business application development.

Doug Grove is a Lead Developer and Principal Architect at warpFactor Inc., a Raleigh NC based company that specializes in Java™ and Web service development. Doug has designed and implemented several WebSphere

e-commerce projects. He specializes in the design and implementation of LDAP directory services. In his twenty years of professional experience, he has worked primarily for the financial and telecoms sectors. He is also typically involved with the specification, building, and deployment of hardware environments

Mara Zandina Hernandez is an IT-Specialist for IBM Global Services in Mexico. She provides technical support, problem resolution, migration services, and performance tuning for IBM customer sites.

Ansgar Hugo is an IT-Architect for IBM Global Services in Germany. He has six years of experience in application design and development focusing Java and J2EE™ applications. He holds a degree in electrical engineering from the University of Darmstadt in Germany. His areas of expertise include design and implementation of Web based solutions and Web content management systems as well as development with WebSphere and WebSphere Portal.

Arinze Izuora is a Software Engineer at the IBM Dublin Software Lab. Ireland. He has been working for IBM since 2003 and works as part of the Lotus Workplace Collaborative Learning team. Arinze has extensive experience working with relational database management systems and holds a Masters degree in Embedded Systems from the University of Manchester Institute of Science and Technology, England, UK.

Steve Moga is the Senior Technical programmer analyst at the Puget Sound Blood Center in Seattle, WA With over 16 years of development on the iSeries platform, his current areas of expertise include iSeries ILE, APIs and Web application development. He is a speaker at user groups on Net.Data and CGI programming on the iSeries platform.

The authors: Mara Zandina Hernandez, Ansgar Hugo, Steve Moga, Bill Moore, Doug Grove, Arinze Izuora

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

John Ganci
Carla Sadtler
International Technical Support Organization (ITSO), Raleigh Center, USA

Kevin Haverlock
IBM Raleigh, USA

Neil Weightman
IBM Farnborough, UK

Ueli Wahli
ITSO, San Jose Center, USA

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience

with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll 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 Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

► Use the online **Contact us** review redbook form found at:

> **ibm.com**/redbooks

► Send your comments in an email to:

> redbook@us.ibm.com

► Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 662
P.O. Box 12195
Research Triangle Park, NC 27709-2195

Part 1

The development process

This part of the book contains an overview of the features provided by WebSphere Application Server - Express, and describes a development process which can be used to develop dynamic Web applications using the Rational Software Development Platform.

1

1

Introduction

This chapter describes our objectives, our focus, and who benefits from the contents of this redbook. We introduce the material covered by the book as well as a description of the WebSphere Application Server - Express users who benefit from its contents.

This chapter covers the following topics:

► 1.1, "Our objectives" on page 4
► 1.2, "The focus of this redbook" on page 5
► 1.3, "How to use this book" on page 7

1.1 Our objectives

This book is designed to provide an overview the WebSphere Application Server - Express platform and the Rational Software Development Platform. As an overview of the material, this book will explain concepts in terms that attempt to avoid technical complexity.

The book provides a systematic approach to the Web application development process, as well as a step-by-step sample for you to see the creation or addition of commonly used areas of functionality for Web applications.

Because WebSphere Application Server - Express and the development tooling have evolved and changed from WebSphere Application Server - Express V5.0 and WebSphere Studio to WebSphere Application Server - Express V6.0 and Rational Software Development Platform, it is often difficult to keep up with the expanding features in the products. We developed the examples in this redbook to provide practical development examples of the key new technologies available in WebSphere Application Server - Express.

We have not built a new application, but rather we have expanded the development example previously built in the IBM Redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301. Reading SG24-6301 is not mandatory to understand the information contained in our redbook because our example code for this book will run on its own. However, if you want to gain a greater understanding of the development process used to originally create our example code, refer to *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301.

> **Application names:** In order to distinguish between the sample application implemented for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 and the updated sample application developed in our current redbook, we use standard names to refer to the sample applications throughout this redbook. When we refer to the sample application implemented for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301, we use the name *Sal301 application*. When we refer to the updated sample application developed in our current redbook, we use the name *Sal404 application*.

1.2 The focus of this redbook

This book is intended for technical users familiar with Web application development who are looking for information how the technical features of WebSphere Application Server - Express have changed and for new means to solve business problems. We have endeavored to give enough background information to understand the technologies involved without overwhelming the reader with too much technical reference material.

This redbook is an update of the existing redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 which covered WebSphere Application Server - Express Version 5. This redbook focuses on the development of a Web application using a simple generic set of requirements to illustrate the principles of good Web application development, rather than examine problems particular to a business domain. In *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 we used a component-based development approach, and detailed the steps to develop each component of our application, using the same techniques for each component. This enabled our readers to use the same techniques and approach as a template for developing components in their own applications. We recommend that you be familiar with the contents of *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 and that you are familiar with the sample application from the original redbook. This is because we use the sample development approach in our new redbook and all the new samples are built on top of the existing sample application from *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301.

A key difference between the *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 redbook and our current book is that we focus the new redbook on a specific subset of the technology provided by WebSphere Application Server - Express Version 6. The features and functions we discuss are chosen because they are new to version 6 of Express Application Server and we do not attempt to develop a new end to end sample application.

WebSphere Application Server - Express: A Development Example for New Developers, SG24-6301 categorized Web application developers using WebSphere Application Server - Express and WebSphere Studio into three general groups. Table 1-1 on page 6 gives an overview of these user groups.

Table 1-1   User groups

User group	User description
Group one	A business user who has a primary business role that is nontechnical
Group two	A user who has client-side and HTML Web application development as part of their primary business role
Group three	A user who has Web application development and sever-side development and maintenance as their primary business role

► Group one users

These are business users that have a primary role in a business that is nontechnical. While the users' primary role is nontechnical, they can be tasked with design, development and deployment tasks either to build a new application, update an existing set of static pages using dynamic functionality, or redesign an existing dynamic Web application to add functionality or optimize performance. A group one user generally has a working knowledge of HTML, page layout, and Web authoring tools. Users in group one are referred to as nontechnical users.

► Group two users

These users usually have Web application development as part of their primary business role. They can be tasked with creation of a Web application or maintenance and the addition of functionality to an existing Web application. Group two users are likely to be primarily focused on client-side programming. The applications they develop will be focused on end-user requirements. A group two user can also be responsible for monitoring the availability of a Web application and its general maintenance. A group two user is an HTML expert and uses DHTML to add functionality to Web pages. This user has the capability to update or make changes to server-side code (such as JavaServer Pages(JSP™) and Enterprise JavaBeans(EJB™), and Java Servlets), but they normally do not author server-side code.

► Group three users

These users have Web application development and sever-side development and maintenance as their primary business role. These users are involved with Web application development at all phases of the development process, but they are focused on server-side development, and develop code (such as JavaServer Pages Java Servlets and Enterprise JavaBeans), which is used by other developers. Users in groups two and three are referred to as technical users.

The focus of the *WebSphere Application Server - Express V5.0.2 Developer Handbook*, SG24-6555 was the user described in group one. It did not assume experience and technical background in the principles of Web application development, and associated protocols and technologies.

Group two and three users are still addressed because the book introduces the WebSphere Application Server - Express platform and WebSphere Studio and describes development in the form of development best practices for Web applications.

Our current redbook is still be useful to nontechnical users, but its target audience is intended to be technical users wanting to understand the new features of WebSphere Application Server - Express Version 6 and of the Rational Software Development Platform. Our redbook is an overview and guide rather than a detailed technical reference. This is because in version 6 WebSphere Application Server - Express is functionally equivalent to WebSphere Application Server, so detailed reference material can be found in the redbooks:

► *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446

► *WebSphere Application Server V6 System Management & Configuration Handbook*, SG24-6451

► *Rational Application Developer V6 Programming Guide*, SG24-6449

1.3 How to use this book

This redbook is organized in two parts.

Part 1, "The development process" on page 1 is a summary of the Web application development process that was used to create the original sample application in *WebSphere Application Server - Express V5.0.2 Developer Handbook*, SG24-6555. It also provides an overview of WebSphere Application Server - Express Version 6 and of Rational Software Development Platform. It contains the following chapters:

► Chapter 1, "Introduction" on page 3

  This is an introduction to the material covered in our redbook.

► Chapter 2, "Development process" on page 11

  This chapter provides a summary of the Web application development process originally described in *WebSphere Application Server - Express V5.0.2 Developer Handbook*, SG24-6555. We discuss how to plan and build

reliable and scalable Web applications and provide an overview of the architecture used by our sample application.

► Chapter 3, "Product overview" on page 33

This chapter provides an overview of WebSphere Application Server - Express and of the WebSphere platform. We also discuss the Rational Software Development Platform.

► Chapter 4, "Getting started" on page 119

This chapter describes the installation and configuration of WebSphere Application Server - Express and of the Rational Software Development Platform. We provide an overview of common tasks performed with the Rational Software Development Platform

► Chapter 5, "Requirements" on page 209

This chapter provides details of the requirements we used when creating the samples in our redbook.

Part 2, "Development examples" on page 221 describes development examples using our redbook sample solution, as a way to demonstrate new features and functions of WebSphere Application Server - Express. This part includes the following chapters:

► Chapter 6, "Web site development" on page 223

This chapter describes the Web development tools provided with the Rational Software Development Platform and we focus on the Web site navigation and Web templates

► Chapter 7, "JavaServer Faces" on page 239

This chapter describes JavaServer Faces(JSF) and we provide examples of how to use JSF in our redbook solution.

► Chapter 8, "Service Data Objects" on page 291

In this chapter we describe Service Data Objects(SDO) and examine how SDO technology could provide the data access layer for our existing redbook sample solution.

► Chapter 9, "Enterprise JavaBeans" on page 311

This chapter introduces Enterprise JavaBeans(EJB) and we discuss when EJBs could be used in our sample application.

► Chapter 10, "Java Message Service" on page 359

In this chapter we discuss the Java Message Service(JMS) and its usage in WebSphere Application Server - Express. After talking about the basic concepts of messaging and JMS we explain how we used JMS based messaging in our sample application

- ► Chapter 11, "Struts" on page 447

    The redbook sample application developed for *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 made extensive use of the Struts Web application framework. In Chapter 11, "Struts" on page 447 we provide an overview of the Struts framework so that you will better understand it's use in our sample application. We also examine the support provided in Rational Software Development Platform for building Struts applications.

- ► Chapter 12, "Web services" on page 475

    In this chapter we introduce Web services technology and provide examples of developing Web services using the Rational Software Development Platform.

- ► Chapter 13, "Database design" on page 527

    In this chapter we discuss the existing data model, database design, and SQL queries inherited from the sample application developed for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301. We discuss and document an improved database design.

- ► Chapter 14, "Code standards and quality" on page 535

    This chapter examines the need for coding standards and quality controls when implementing Web applications. We provide some example standards and discuss how you might make use of these within your development projects.

- ► Chapter 15, "Bidding component" on page 549

    In this chapter, we show you how to implement a bidding component for the SAL404Realty sample application.

2

Development process

This chapter includes development process and application architecture material from the original redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301. We have included this material in our new book so that you can understand the development of the original sample application. It is important that you understand not only the sample process and architecture, but also because it remains important to choose a good process and architecture for any new Web applications.

We discuss a development process for building small-scale Web applications using Rational Software Development Platform and WebSphere Application Server - Express. Although we borrow ideas and principles from other more formal processes, ours is only offered here only as an aid. If you find this information useful and want to follow a more formal process, see:

► The Rational Unified Process(RUP)

   http://ibm.com/software/awdtools/rup/index.html

► Extreme programming resources

   http://www.xprogramming.com/

► Extreme programming introduction

   http://www.extremeprogramming.org/

► The Agile alliance

   http://www.agilealliance.org

2.1 Development process basics

In this section we explain the basics of a development process, provide a definition of the development process, identify the reasons that make it important from both the technical and the business perspective, describe the elements necessary to realize it, and finally define the concepts behind it.

2.1.1 Definition of a development process

A development process is the set of activities needed to move from a business concept to a tangible working software system. More specifically:

► It serves as a guide for all the participants: customers, users, developers, and managers.

► It helps identifying the current state of a business in terms of infrastructure, people, and skills.

► It provides a roadmap for building the system.

► It defines who is doing what, when, and how to build a new software system or enhance an existing software system.

► It provides guidance to the order of a team's activities.

► It directs the task of individual developers and the team as a whole.

► It specifies what artifacts should be developed.

► It offers criteria for monitoring and measuring a project's products and activities.

2.1.2 Importance of a development process

It is a well known fact that a high percentage of software development projects fail. W. Wayt Gibbs ("Software's Chronic Crisis," Sept. 1994, *Scientific American*) estimated that:

► On average, large projects take 50% longer than originally estimated.
► 75% of large projects are operational failures.
► 25% of large projects are cancelled.

Most of the time, the success of a project is relative to the development process being used during its life cycle. A software development process should be lean enough to be easily adopted by the people involved in a project, and complex enough to be able to describe, define, and monitor a project from both the technical and the business perspective. When such a development process is in place, it offers the following benefits.

Business perspective benefits

Using a development process provides the following business benefits:

► The project is broken down into manageable phases, each with a clear entry and exit point.

► The current state of the business in terms of existing hardware and software, the people and their skills, is understood and documented honestly. This documentation plays the role of a tangible starting point for the project life cycle.

► The business and technical objectives for the project are documented clearly complimenting the existing state of business documentation, and hence, creating a set of documents detailing both the current business state and the desired business state.

► An initial roadmap is created outlining all the activities and roles involved for the realization of the project. This initial roadmap is used as a draft for the project plan.

► Each person involved in the project is assigned one or more roles, and is responsible for carrying out specific activities within the project life cycle.

► The initial roadmap can be extended to create a project plan that helps track the progress of a project, its tasks, the activities involved to perform a task, the resources available as well as the budget.

► A business process helps manage requirements, client expectations, and associated risks.

► A business process enables the control of changes to the software

Technical perspective benefits

Technical benefits of using a development process are:

► Facilitates the use of component-based architectures.
► Allows the development of the software to be carried out iteratively.

2.1.3 Realization of a development process

A development process is realized with the use of:

► Technologies

Technologies include computer systems, network connectivity, operating systems, development environments, and programming languages

► Organization structures

The process must fit in the company's overall organization patterns.

- ▶ Tools

  To support the process must be in place.

- ▶ People

  The expected skill set of target developers must be present.

2.1.4 Development process principles

For a graphical representation of the development process, see Table 2-1 on page 16.

Phases

From a management perspective, the life cycle of a software development process is decomposed over time into sequential phases, each concluded by a major milestone. Each phase is essentially a span of time between two major milestones. At each phase end, an assessment should be performed to determine whether the objectives of that particular phase have been met. A satisfactory assessment allows the project to move to the next phase.

Our development process is divided into the following four phases:

1. Starting a project
2. Understanding and planning a project
3. Building a solution
4. Handing over the solution

Views

Different people working on a project will view its objectives in different ways, and therefore they will have different requirements for a development process. In general the different views of a project process can be split between:

- ▶ A technical view
- ▶ A business view

Activities

An activity of a specific worker is a unit of work that an individual in that role can be asked to perform. The activity has a clear purpose, usually expressed in terms of creating or updating some artifacts, such as a model, a class, and a plan. Every activity is assigned to a specific worker. The granularity of an activity is generally a few hours to a few days, it usually involves one worker, and affects one or only a small number of artifacts. An activity should be usable as an element of planning and progress. If it is too small, it will be neglected. If it is too large, progress would have to be expressed in terms of an activity's parts.

Technical activities

The technical activities include the following:

- ► Domain modelling
- ► Requirements analysis
- ► Logical design
- ► Physical design
- ► Implementation
- ► Deployment
- ► Testing
- ► Maintenance

Business activities

Business activities include the following:

- ► Configuration management
- ► Quality management
- ► Project management
- ► Facilities management

Roles

The most central concept in a process is that of a role. A *role* defines the behavior and responsibilities of an individual, or a set of individuals working together as a team, within the context of a software engineering organization. Note that roles are not individuals; instead, roles describe the appropriate behavior and responsibilities of an individual. Individual members of the software development organization will wear different hats, or perform different roles. The mapping from individual to role, performed by the project manager when planning and staffing the project, allows different individuals to perform several different roles, and for a particular role to be played by several individuals.

Table 2-1   Development process phases, views and activities

		Starting a project	**Understanding and planning a project**	**Building a solution**	**Hand-over**
Technical view	Domain modelling				
	Requirements analysis				
	Logical design				
	Physical design				
	Implementation				
	Deployment				
	Testing				
	Maintenance				
Business view	Configuration management				
	Quality management				
	Project management				
	Facilities management				

2.2 Starting a project

The purpose of this phase of the development process is to kick-start the project. The following sections provide a brief description of some of the things you should consider in this phase.

2.2.1 Understanding your business today

You need a thorough understanding of the following topics:

► Your existing hardware and software infrastructure
► Your people
► The skills of your people

2.2.2 Where do you want to go?

You need a thorough understanding of what results you want from the process. Consider the following questions:

- ► What are the business requirements of your project?
- ► What are the key features of your project?
- ► What constraints does your project have?
- ► What are your success criteria?
- ► What is your financial forecast?
- ► What potential risks could you face during your project?

2.2.3 An initial roadmap

An initial roadmap consists of the following steps:

1. Break down your project into phases.
2. Identify the activities involved.
3. Assign roles to the people involved in the project.

2.3 Understanding and planning a project

The steps to planning a project are as follows:

1. Understand what needs to be built.
2. Describe the functional requirements.
3. Describe the nonfunctional requirements.
4. Prepare a high-level logical design of the application.

2.4 Building a solution

Here are the steps to build a solution:

1. Think of user interface considerations and create an initial navigation map.

2. Design the Web design elements that are going to be used in the application.

3. Prepare a first, nonfunctional prototype.

4. With the help of end-users, validate that:

    – The requirements were captured correctly.
    – The look and feel of the application is acceptable.

5. Prepare a low-level physical design of the application.

6. Divide the application into components and identify the dependencies between them.

7. Prepare a document that describes the functionality of each component, how it relates to the other components, and where it fits in the overall architecture of your application.

8. Give ownership of each component to a developer.

9. Update the document to reflect any changes and assigned projects.

10. Implement some functionality or extend your initial, nonfunctional prototype.

11. Enter an iterative cycle of building a component, testing, and deploying it on WebSphere Application Server - Express until all components are implemented.

2.5 Project hand-over

After the development of a project solution is complete, a good development process will help you to complete the important activities involved in the project hand-over. These activities are often overlooked or underestimated. Hand-over activities that your process should address include:

► Project deployment
► Migration from previous systems
► Data conversion and migration
► Project maintenance
► User training
► System documentation
► System administration

2.6 Real estate application architecture

In the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301, we built a sample application using the development process outlined. We refer to this application as the `Sal301 application`. The sample was not meant to solve a detailed business problem, or to suggest this was the most common business problem solved using WebSphere Studio or the Rational Software Development Platform. We chose this business problem because it is simple to describe, and does not require specialist business knowledge. In addition, the general concepts of the problem and the functions we implement in our sample solution are generic and can easily be applied to other business domains.

The sample application implements all the functionality necessary to:

► Identify a user as a visitor, registered user, agent or administrator
► Browse, view and search the catalog

- ▶ Send E-mails
- ▶ Order
- ▶ Produce and view reports
- ▶ Create, update, and delete user account information
- ▶ Create, update, and delete property information
- ▶ Create, update, and delete any other content in the system

2.6.1 Component-based architecture

The sample application is divided into components.

A *component* is a cohesive unit of software that provides a related set of functions and services. Components can be developed and delivered independently of other components, that is, they are inherently modular in nature.

There are two component types in the Sal301 application:

Business components

Business components include:

- ▶ Property catalog component
- ▶ News component
- ▶ E-mail component
- ▶ Interest list component
- ▶ Reporting component
- ▶ User component

Administrative components

Administrative components include:

- ▶ Property catalog administration
- ▶ User component

2.6.2 Layered design

In a layered design, each component is broken down into layers. Layers create separation within the software by abstracting specific types of functionality into functional layers, and providing conceptual boundaries between sets of services, for example, in the property catalog component:

Property catalog component layers consist of the following:

- ▶ Presentation layer
  - – HTML, JSP

- ► Controller layer

  - – Struts Action and Form classes

- ► Business logic layer

  - – Manager classes

- ► Domain layer

  - – JavaBeans™ used as data transfer objects(DTO)

- ► Data access layer

  - – JavaBeans with embedded SQL (data access command beans)

2.6.3 Package structure

The package structure for the Catalog component of Sal301 was similar to this:

- ► Java source:

```
com.ibm.itso.sal301r.propertycatalogcomponent.action
com.ibm.itso.sal301r.propertycatalogcomponent.form
com.ibm.itso.sal301r.propertycatalogcomponent.manager
com.ibm.itso.sal301r.propertycatalogcomponent.dto
com.ibm.itso.sal301r.propertycatalogcomponent.dao
```

- ► Web content:

```
/catalog/
```

In our current redbook we have refactored the sample application which we now refer to as the Sal404 application and renamed the packages used. For example the package structure for the Catalog component is now:

- ► Java source:

```
com.ibm.itso.sal404.propertycatalog
com.ibm.itso.sal404.propertycatalog.action
com.ibm.itso.sal404.propertycatalog.form
com.ibm.itso.sal404.propertycatalog.manager
com.ibm.itso.sal404.propertycatalog.dto
com.ibm.itso.sal404.propertycatalog.dao
```

- ► Web content:

```
/catalog/
```

2.6.4 Naming conventions

We used the following naming conventions:

- ► HTML pages and JSPs

```
DescriptionOfTheAction.jsp, for example, ViewAllProperties.jsp
```

► Struts Action classes

```
DescriptionOfTheActionAction.class, for example, AddUserAction.class
```

► Struts Form classes

```
NameOfTheFormForm.class, for example, AddPropertyForm.class
```

► Manager classes (one per component)

```
ComponentScopeManager, for example, UserManager.class
```

► Domain classes

```
ClassScope.class, for example, User.class
```

► Data Access classes:

```
DescriptionOfTheQuerySQL.class, for example, GetPropertyByIdSQL.class
```

2.7 Application architecture

Designing the architecture of an application is perhaps the most important task when developing an e-business solution using Java 2 Platform, Enterprise Edition(J2EE) and the Rational Software Development Platform. The following requirements should be considered:

► Scalability

How easy is it to scale the application to handle a higher workload?

► Extensibility

How easy is it to extend the functionality of the application? Can we easily add new functions that meet changing business requirements?

► Reliability

How reliable is the application? Does it work consistently and is it free from error?

► Efficiency

How efficient is the application? Does it perform well and use a minimum of resources to get the job done?

► Maintainability

How easy is it to maintain the application? Are changes easy and quick to make without requiring major disruption to working code?

► Portability

How portable is the application across different environments, operating systems, and so on?

2.8 Overview of the architecture

In this part of the chapter we outline the architecture choices that we made when developing the sample Sal301 application for the original redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301. We considered the principles of a well-architected solution as well as the strategies and the best practices that have been proven in real production environments. One advantage of the architecture is that the sample application can be quite easily modified and new function and technology can be added as you require. One example of this in our current redbook is that we reuse the original sample application for all our new examples.

2.8.1 Component-based development

Our first decision was to follow a *divide and conquer* approach when developing the application. We decided to conquer it by dividing the application into components.

A *component* is a cohesive independent unit of software that provides a related set of functions and services. A component adheres to the fundamental principles that underpin an object in the context of object-oriented software development.

> **Tip:** *Object-oriented software* development is a a key concept of and commonly used way of developing software. A key part of this development is the use of objects. Java is a programming language that supports object-oriented software development and allows programmers to develop objects. We assume that you already have some knowledge of this way of developing software, but a brief discussion is given below. If you need more details on object-oriented software development, see the suggested reading:
>
> ► *Object Technology: A Manager's Guide (2nd Edition),* David A. Taylor, Addison-Wesley Publishing Company, 2nd edition September 11 1997, ISBN0201309947
>
> ► *Object technology made simple,* Mory Bahar, Simple Software Publishing, First edition May 1996, ISBN0965245705
>
> ► *Making Sense of Java,* Bruce Simpson, et al, Mannings Publications, 1996, ISBN 0132632942
>
> ► *Design Patterns: Elements of Reusable Object-Oriented Software.* Erich Gamma, et al. Addison-Wesley Publishing Company, January 1995, ISBN0201633612

More specifically, a component (like an object) caters to:

- ▶ Unification of data and function

  A software object consists of state (the values of its member variables) and functions (the methods the object provides to allow manipulation of its member variables).

- ▶ Encapsulation

  The internal implementation of the methods of an object are hidden away from the object client (anyone who uses the object). This means that a dependency between a client and an object exists only in terms of the object specification and not its implementation.

- ▶ Identity

  Once created, an object can be identified uniquely regardless of its current state.

A component extends these principles by introducing the notion of a component specification which consists of one or more component interfaces.

A component specification is the definition of what a component does, as opposed to how it performs a task. This definition is exposed to the clients of the component in the form of one or more component interfaces. Each component interface outlines a list of available operations that the component can perform.

A component specification is used as part of a *realization contract*, a contract between the component designers and the developers that implement the component. In other words, the final implementation of the component has to meet the requirements set in the specification.

A component interface is used as part of a usage contract, a contract between the component interface or interfaces and the component clients. In other words, the operations exposed in the interface of a component are guaranteed to behave in the same manner under any circumstances.

The clear separation between a component specification and a component implementation is the most important characteristic of component-based development (CBD).

A component is realized with the use of objects and so it would normally contain one or more classes. Therefore, a component is normally larger grained that an object.

When the application is divided into components, we can identify a vertical separation of the functionality, as shown in Figure 2-1 on page 24.

Component 1

Component 2

Component 3

Component N
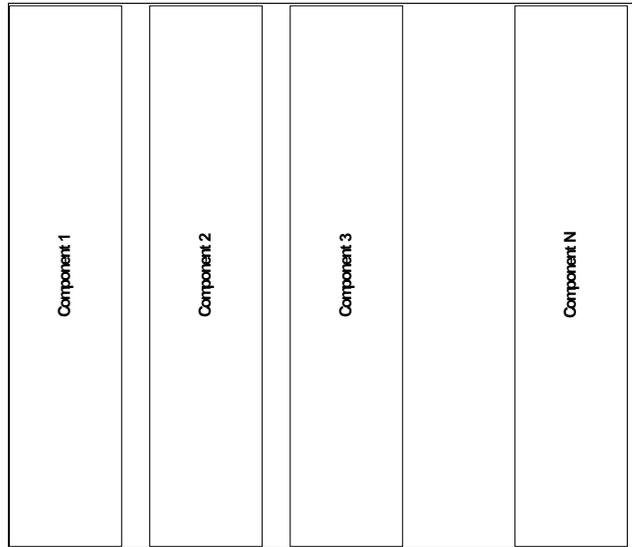
Figure 2-1   Vertical separation of the application functionality

2.8.2 Layered application design

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

A typical separation of layers in J2EE applications is:

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

When the application is divided into layers, we can identify a horizontal separation of the functionality, as shown in Figure 2-2 on page 25.

```
┌─────────────────────────────────┐
│ ┌─────────────────────────────┐ │
│ │     Presentation layer      │ │
│ └─────────────────────────────┘ │
│ ┌─────────────────────────────┐ │
│ │      Controller layer       │ │
│ └─────────────────────────────┘ │
│ ┌─────────────────────────────┐ │
│ │    Business facade layer    │ │
│ └─────────────────────────────┘ │
│ ┌─────────────────────────────┐ │
│ │        Domain layer         │ │
│ └─────────────────────────────┘ │
│ ┌─────────────────────────────┐ │
│ │      Data access layer      │ │
│ └─────────────────────────────┘ │
└─────────────────────────────────┘
```
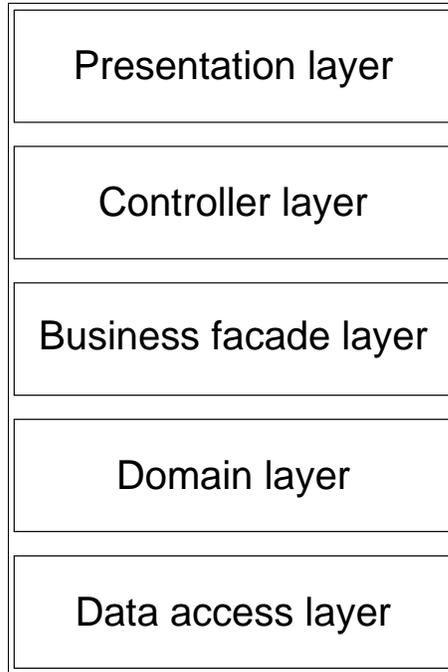
Figure 2-2   Horizontal separation of the application functionality

2.8.3 Usage of design patterns

According to *Design Patterns: Elements of Reusable Object-Oriented Software,* E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1994, ISBN: 0-201-63361-2:

> *"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use."*

During the design and implementation of the redbook sample application, we identified some design patterns that are useful.

Business facade design pattern

This design pattern is related to the session facade and message facade design patterns as described in *EJB Design Patterns,* Floyd Marinescu, John Wiley &

Sons, Inc., 2002, ISBN: 0-471-20831-0. We used this design pattern in order to minimize the dependencies between the front-end and the back-end layers of the application. Furthermore, with the use of this design pattern we can implement each use case as one business method that is executed with only one network call.

Data transfer object (DTO) design pattern

This design pattern is described in *EJB Design Patterns*, Floyd Marinescu, John Wiley & Sons, Inc., 2002, ISBN: 0-471-20831-0, and is related to the Value Object design pattern mentioned in other books about J2EE development. We used this design pattern to minimize the number of network calls between the layers of the application. Thus, when we wanted to send a collection of data as parameters to a business method we encapsulated it in a DTO and sent this object instead. Also, every time we returned results from a business method, we encapsulated the data in a DTO and returned this object instead. A DTO is, in effect, a way of transferring bulk data between layers of an application.

Data access object (DAO) design pattern

This design pattern is related to the data access command bean design pattern as described in *EJB Design Patterns*, Floyd Marinescu, John Wiley & Sons, Inc., 2002, ISBN: 0-471-20831-0. We used this design pattern in order to minimize the dependency on the underlying database and also to hide away all the database operations. Instead of obtaining a connection to a DataSource, preparing and executing a statement and getting the results directly from our manager classes, we have chosen to add another layer, the data access layer where each database operation is the responsibility of a single data access object. Therefore, if we choose to change the physical database model in the future or even the database provider, we only need to change the implementation of the data access objects, and not the implementation of our business methods.

2.9 Component architecture

Earlier in this chapter we mentioned that we decided to follow a component-based development approach while working on the sample application. This means that we had to identify areas of functionality that could be grouped together logically into a single unit of software, a component.

In this section we outline the components of the realty sample application and provide a short overview for each of them.

2.9.1 PropertyCatalog

The PropertyCatalog component provides all the methods necessary to manage the catalog of properties in the realty application.

The PropertyCatalog component implements the functionality described in the following use cases:

- ► PCC.SearchByCriteria
- ► PCC.ViewPropertyDetails
- ► PCC.ViewAgentPropertyList
- ► PCC.AddPropertyDetails
- ► PCC.ModifyPropertyDetails

The PropertyCatalog component has a dependency on the following components:

- ► User component
- ► Reporting component
- ► InterestList component

2.9.2 News

The news component provides all the methods necessary to manage news in the realty application.

The news component implements the functionality described in the following use cases:

- ► NC.ViewNewsItems
- ► NC.AddNewsItem
- ► NC.ModifyNewsItem

The news component has a dependency on the following components:

- ► User component

2.9.3 E-mail

The E-mail component provides all the methods necessary to generate and send e-mails within the realty application.

The e-mail component implements the functionality described in the following use cases:

- ► EC.SendRegistrationConfirmationMessage
- ► EC.RequestPasswordMessage

The e-mail component has a dependency on the following:

► JavaMail™ API

2.9.4 InterestList

The InterestList component provides all the methods necessary for a customer to express his interest in properties within the realty application.

The InterestList component implements the functionality described in the following use cases:

► ILC.AddProperty
► ILC.RemoveProperty
► ILC.ViewInterestList
► ILC.RequestViewing
► ILC.RequestBrochure
► ILC.ClearInterestList
► ILC.ReturnToPropertyCatalog
► ILC.CheckoutInterestList

The InterestList component has a dependency on the following components:

► User component
► PropertyCatalog component

2.9.5 Reporting

The Reporting component provides the methods necessary to generate two different types of reports within the realty sample application.

The Reporting component implements the functionality described in the following use cases:

► RC.ViewInterestForAgentReport
► RC.ViewInterestForPropertyReport

The Reporting component has a dependency on the following components:

► User component
► PropertyCatalog component
► InterestList component

2.9.6 User

The user component provides the methods necessary to manage, authorize and authenticate different types of users of the realty sample application.

The user component implements the functionality described in the following use cases:

- ► UC.Login
- ► UC.Logout
- ► UC.RegisterCustomer
- ► UC.ModifyCustomerDetails
- ► UC.DeleteCustomerDetails
- ► UC.RegisterAgent
- ► UC.ModifyAgentDetails
- ► UC.DeleteAgentDetails
- ► UC.ListUsers
- ► UC.ViewUserDetails
- ► UC.RequestPassword

The User component has a dependency on the following components

- ► E-mail component

2.9.7 Dependencies between components

The dependencies between the components of the Sal301 application are shown in Figure 2-3 on page 30.
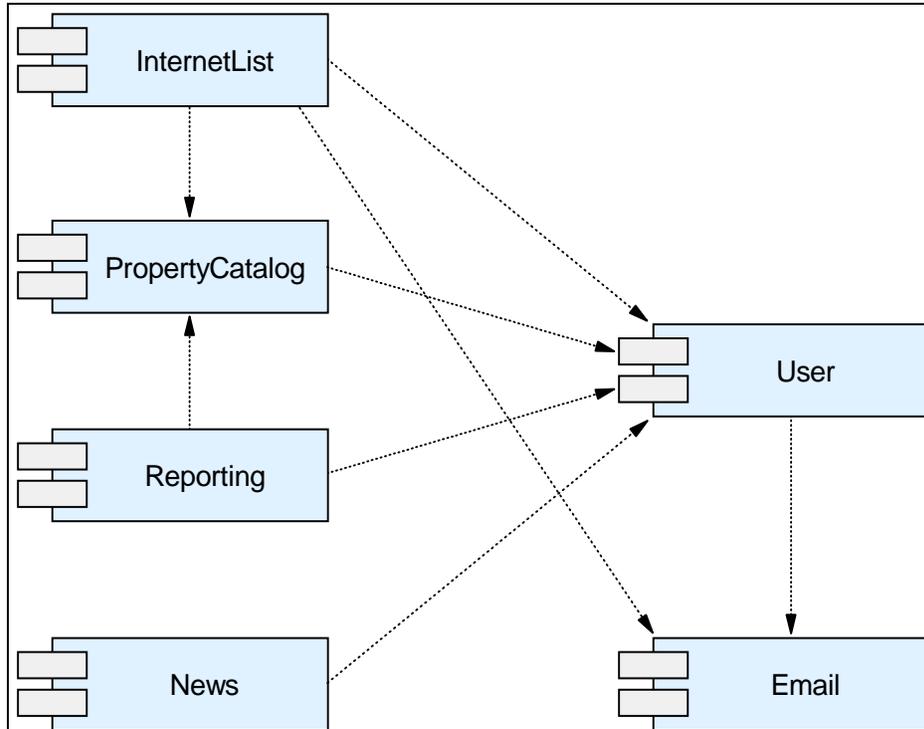
Figure 2-3   Component dependencies in the realty sample application

2.10 Layered architecture

This section describes the major layers used in our sample application.

2.10.1 Presentation layer

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

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

2.10.2 Controller layer

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

- ► Java Servlets
- ► JavaBeans
- ► Struts actions
- ► Struts forms

2.10.3 Business facade layer

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

2.10.4 Domain layer

The domain layer consists of custom-designed JavaBeans that are used to encapsulate data transferred between layers of the application. These JavaBeans are called data transfer objects (DTO), see "Data transfer object (DTO) design pattern" on page 26. To implement these DTOs we used the functionality that WebSphere Studio and the Rational Software Development Platform provide to help us generate getter and setter methods.

2.10.5 Data access layer

The data access layer consists of custom designed JavaBeans that perform database operations using JDBC™. These JavaBeans are called data access objects (DAO), see "Data access object (DAO) design pattern" on page 26. To implement these DAOs we used the various data wizards provided by WebSphere Studio and the Rational Software Development Platform.

3

Product overview

IBM WebSphere Application Server - Express V6 combines application server and development tools in one single integrated package. This package is geared towards development and operation of Web-based solutions for small and medium businesses.

Based on the latest Java technology and Web services standards, WebSphere Application Server - Express gives you an affordable, ready-to-go solution, including streamlined server functionality, quick-start application templates and robust development tools.

This chapter shows how WebSphere Application Server - Express fits in the WebSphere Application Server product family and gives an overview of the underlying WebSphere Application Server architecture and key technologies.

3.1 The WebSphere product family

WebSphere is the IBM brand of software products that are designed to work together to help deliver dynamic e-business quickly. WebSphere provides solutions for positively touching a client's business. It also provides solutions for connecting people, systems, and applications with internal and external resources. WebSphere is based on infrastructure software (middleware) designed for dynamic e-business. It delivers a proven, secure, and reliable software portfolio that can provide an excellent return on investment. Figure 3-1 shows a high-level overview of the WebSphere platform.

.
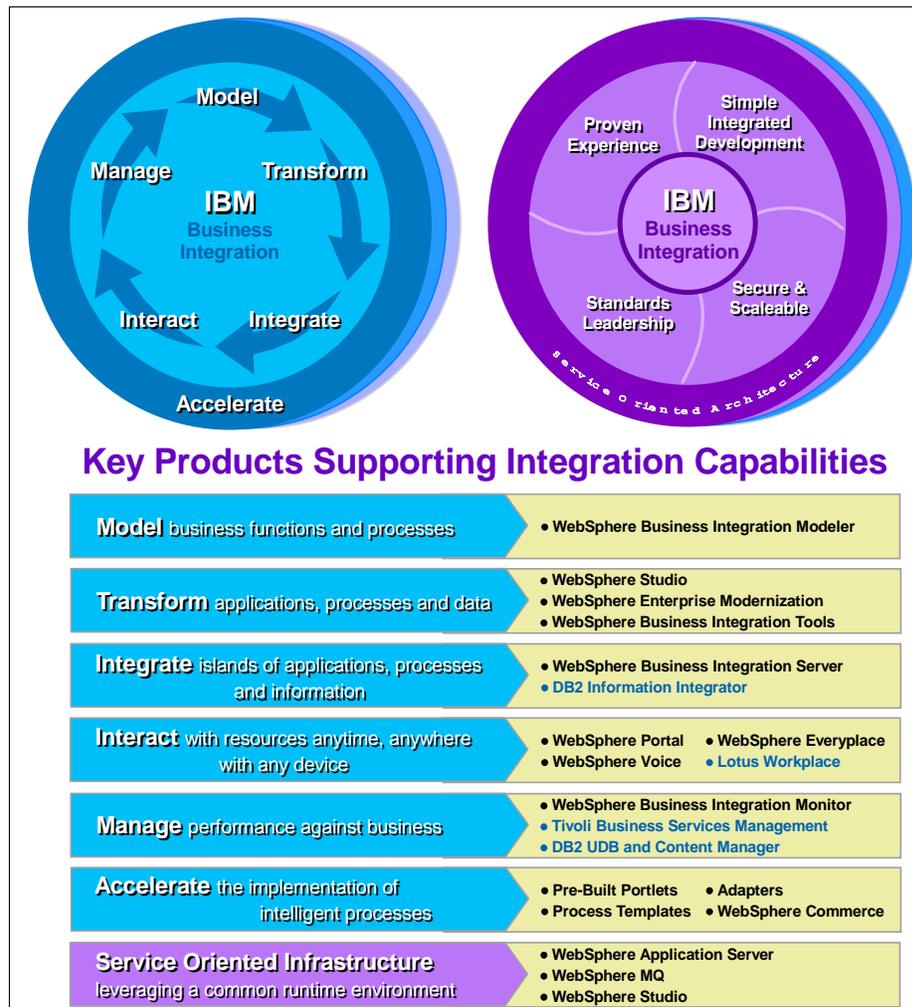
Figure 3-1 WebSphere product family

The foundation of the WebSphere brand is the application server. The application server provides the runtime environment and management tools for J2EE and Web services-based applications. Clients access these applications through standard interfaces and APIs. The applications, in turn, have access to a wide variety of external sources such as existing systems, databases, and Web services, that can be used to process the client requests.

WebSphere Application Servers are available in multiple packages to meet specific business needs. These application servers also serve as the base for other WebSphere products, such as WebSphere Commerce and WebSphere Portal, by providing the application server required for running these specialized applications.

WebSphere Application Servers are available on a wide range of platforms, including UNIX®-based platforms, Microsoft® operating systems, IBM z/OS, and iSeries. Although branded for iSeries, the WebSphere Application Server products for iSeries are functionally equivalent to those for the UNIX and Microsoft platforms.

3.1.1 The WebSphere Application Server family

WebSphere Application Server is IBM's implementation of the J2EE (Java 2 Enterprise Edition) platform, conforming to V1.4 of the specification. WebSphere Application Server is available in three unique packages that are designed to meet a wide range of client requirements. At the heart of each package is a WebSphere Application Server that provides the runtime environment for enterprise applications. This chapter will center around the runtime server component of the following packaging options of WebSphere Application Server:

► IBM WebSphere Application Server - Express V6. We will refer to this as the *Express* configuration.

► IBM WebSphere Application Server V6. We will refer to this as the *Base* configuration.

► IBM WebSphere Application Server Network Deployment V6. We will refer to this as the *Network Deployment* configuration.

3.2 WebSphere Application Server - Express V6

WebSphere Application Server - Express is geared toward those who need to get started quickly with e-business. It is specifically targeted at medium-sized businesses or departments of a large corporation, and is focused on providing ease of use and application development. Although WebSphere Application Server - Express is the entry level configuration of WebSphere Application

Server and is limited to a single server environment, it provides full support for the J2EE 1.4 programming model.

While in V5 there were restrictions in the base functionality, V6 Express has now the equivalent functionality as the base server. For example it has now an EJB container and a messaging engine for default messaging included. In that way, WebSphere Application Server - Express is easy upgradable to large scale environments with a base license or network deployment without having to change your applications.

The WebSphere Application Server - Express offering is unique from the other packages in that it is bundled with an application development tool. Although there are WebSphere Studio and Rational Developer products designed to support each WebSphere Application Server package, they are normally ordered independent of the server.

WebSphere Application Server - Express includes the Rational Web Developer application development tool. It provides a development environment geared toward Web developers and includes support for most J2EE 1.4 features with the exception of EJB and JCA development environments. However, keep in mind that WebSphere Application Server - Express does contain full support for EJB and JCA, so you can deploy applications that contain those technologies with the tools packaged with WebSphere Application Server - Express. Figure 3-2 on page 37 shows a high-level overview of WebSphere Application Server - Express.
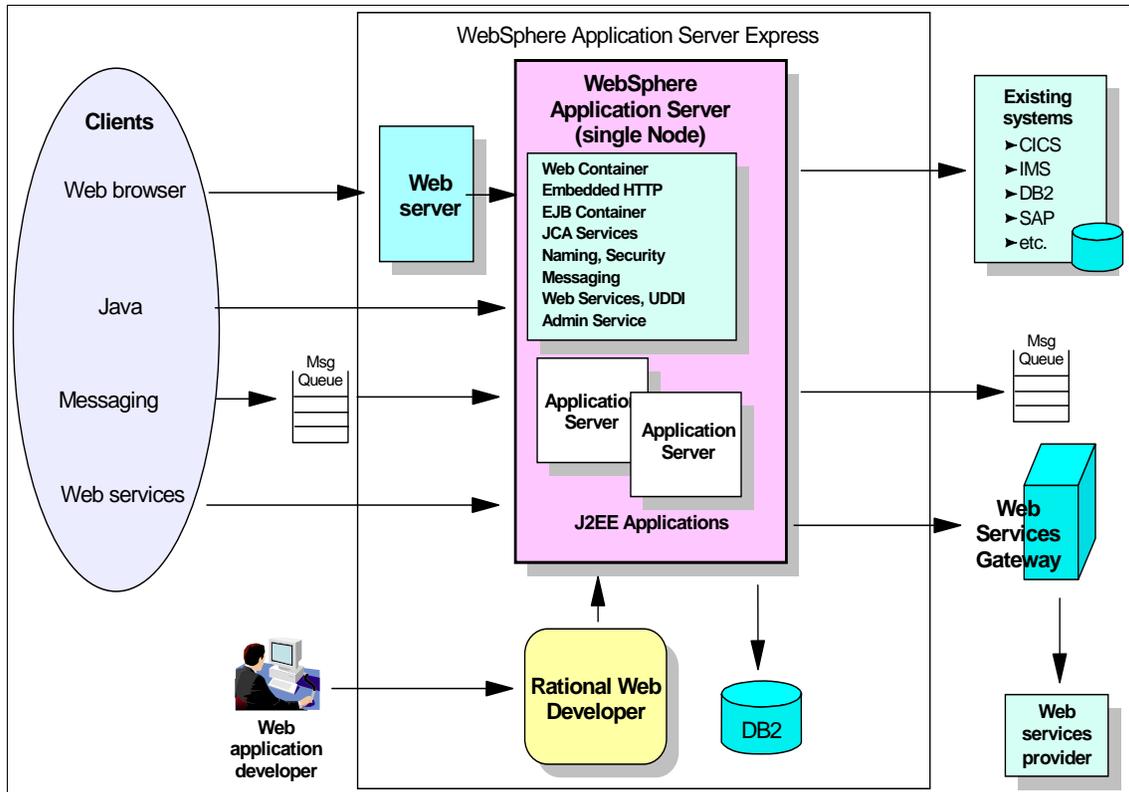
Figure 3-2   WebSphere Application Server - Express overview

Because Rational Web Developer is a subset of Rational Application Developer, you can choose to switch to Rational Application Developer at any time in your project. Alternatively, you can choose to equip only special members in your team with Rational Application Developer if you want support for the development enterprise features like EJBs.

3.2.1 The WebSphere Application Server highlights

WebSphere Application Server provides the environment to run your Web-enabled e-business applications. You can think of an application server as *Web middleware* or the central component in a three-tier J2EE e-business environment. The first tier is the HTTP server that handles requests from the browser client. The third tier might be the business database (for example, DB2 UDB for iSeries) or existing systems (for example, traditional business applications such as order processing). The middle tier is IBM WebSphere Application Server, which provides a framework for consistent, architected linkage between the HTTP requests and the business data and logic.

IBM WebSphere Application Server is intended for organizations that want to take advantage of the productivity, performance advantages, and portability that Java provides for dynamic Web sites. It includes:

► J2EE 1.4 support

► High-performance connectors to many common back-end systems to reduce the coding effort required to link dynamic Web pages to real line-of-business data

► Application services for session and state management

► Web services that enable businesses to connect applications to other business applications, to deliver business functions to a broader set of customers and partners, to interact with marketplaces more efficiently, and to create new business models dynamically

► The WebSphere Platform Messaging infrastructure to complement and extend WebSphere MQ and application server

   It is suitable for those that are currently using the WebSphere Application Server V5 embedded messaging and for those that need to provide messaging capability between WebSphere Application Server and an existing WebSphere MQ backbone.

The architecture details and technical key concepts are discussed starting from section 3.3, "WebSphere Application Server architecture" on page 58 onwards.

3.2.2 The development tool

The IBM Rational Web Developer V6 extends the capabilities of Eclipse 3.0 with with visual tools for Web, Java, and rich client applications, and full support for XML, Web services, and Enterprise Generation Language (EGL). In previous releases, this product was known as WebSphere Studio Site Developer.

For detailed information about Rational Web Developer, Rational Application Developer and the Rational Software Development Platform see the redbook *Rational Application Developer V6 Programming Guide*, SG24-6449.

Version 6 terminology

Table 3-1 on page 39 provides a basic terminology comparison between Version 6 and Version 5 for reference purposes.

Table 3-1   Terminology

Version 6	Version 5
Rational Developer or Rational Software Development Platform **Note**: used to describe family of products built on common Eclipse base.	WebSphere Studio
Rational Application Developer (referred to as Application Developer)	WebSphere Studio Application Developer
Rational Web Developer (referred to as Web Developer)	WebSphere Studio Site Developer
IBM Eclipse SDK 3.0 **Note**: IBM branded and value-added version of Eclipse SDK 3.0	WebSphere Studio Workbench (IBM supported Eclipse 2.x)
Workbench or Rational Software Development Platform	Workbench

As noted in Table 3-1, in previous releases the Rational Application Developer product was known as WebSphere Studio Application Developer, and Rational Web Developer was known as WebSphere Studio Site Developer.

Rational Software Development Platform

Figure 3-3 displays a summary of features found in the Rational Developer V6.0 products.

IBM Rational Application Developer

IBM Rational Web Developer

- Full development support
- Support for J2EE 1.2, 1.3, 1.4 Web based applications (JSF, Struts, SDO tools)
- No EJB, J2C development support

Application Server Tool (AST)

- Tool for assembly, deployment (EJB, Web Services) and debug J2EE applications
- No development support
- WebSphere Rapid Deployment
- Support for Enahanced EAR
- Server Tools – support for remote server

- Server Test Environments (WebSphere Application Server V5.0, 5.1, 6.0, Tomcat 5.0 and WebLogic 6.1, 7.1, 8.1 available separately)
- Visual Java GUI builder
- Web Diagram Editor
- Site Designer
- Page Templates
- XML tools
- Web Services tools
- Database tools
- Portal and portlet development
- Enterprise Generation Language (EGL) tools
- Debugger

- Full J2EE 1.4 support
- Portal Test Environment 5.0.2.2
- UML Visual editors
- Static and runtime analysis
- Extended debugging and profiling
- Component test automation
- ClearCase LT for team integration
- Rational Unified Process (RUP) integration

IBM Eclipse SDK 3.0

- IBM commercially supported Eclipse 3.0
- IBM Rational Software Development Platform tooling

Figure 3-3   IBM Rational Software Development Platform tools and features summary

The Rational Software Development Platform is based on the Eclipse platform and grows and expands with every release with the Eclipse project.

Eclipse project

The Eclispe project is an open source software development project devoted to creating a development platform and integrated tooling.

Figure 3-4 on page 41 depicts the high level Eclipse project architecture and shows the relationship of the following sub projects:

► Eclipse Platform
► Eclipse Java Development Tools (JDT)
► Eclipse Plug-in Development Environment (PDE).
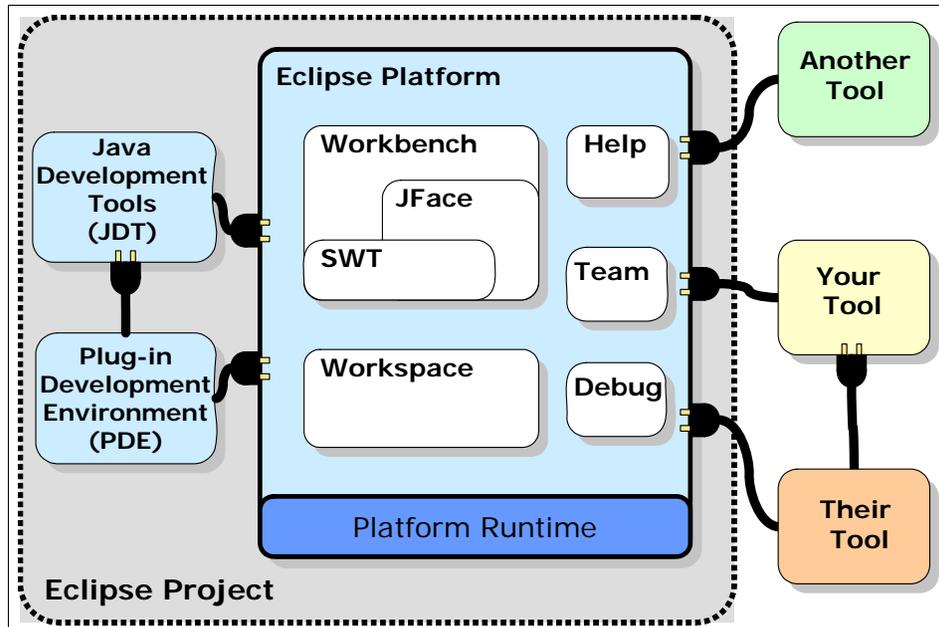
Figure 3-4   Eclipse Project overview

With a common public license that provides royalty free source code and world-wide redistribution rights, the Eclipse Platform provides tool developers with great flexibility and control over their software technology.

Industry leaders such as IBM, Borland, Merant, QNX Software Systems, RedHat, SuSE, TogetherSoft, and WebGain formed the initial eclipse.org board of directors of the Eclipse open source project.

More detailed information about Eclipse can be found at:

http://www.eclipse.org

Eclipse Platform

The Platform provides a framework and services which serve as a foundation for tools developers to integrate and extend the functionality of the Platform. The Platform includes a workbench, concept of projects, user interface libraries (JFace, SWT), built-in help engine, and support for team development and debug. The Platform can be leveraged by a variety of software development purposes including, modeling and architecture, integrated development environment (C/C++, Java, Cobol, etc.), testing, and so forth.

Eclipse Java Development Tools (JDT)

The JDT provides the plug-ins for the Platform specifically for a Java-based integrated development environment, as well as the development of plug-ins for Eclipse. The JDT add the concepts of Java projects, perspectives, views, editors, wizards, refactoring tools, to extend the Platform.

Eclipse Plug-in Development Environment (PDE)

The PDE provides the tools to facilitate the development of Eclipse plug-ins.

3.2.3 Rational Developer supported platforms and databases

This section describes the platforms and databases supported by the Rational Developer products.

Supported operating system platforms

IBM Rational Application Developer V6.0 supports the following operating systems:

► Microsoft Windows®:

- – Windows XP with Service Packs 1 and 2
- – Windows 2000 Professional with Service Packs 3 and 4
- – Windows 2000 Server with Service Packs 3 and 4
- – Windows 2000 Advanced Server with Service Packs 3 and 4
- – Windows Server 2003 Standard Edition
- – Windows Server 2003 Enterprise Edition

► Linux® on Intel®:

- – Red Hat Enterprise Linux Workstation V3 (all service packs)
- – SuSE Linux Enterprise Server (SLES) V9 (all service packs)

The IBM Agent Controller included with IBM Rational Application Developer V6.0 is supported on many platforms running WebSphere Application Server. For details, refer to the *Installation Guide, IBM Rational Application Developer V6.0* product guide (install.html) found on the IBM Rational Application Developer V6 Setup CD 1.

Supported databases

IBM Rational Application Developer V6.0 supports the following database products:

► IBM Cloudscape V5.0
► IBM Cloudscape V5.1
   (bundled with the WebSphere Application Server V6.0 Test Environment)
► IBM DB2 Universal Database V8.1

- ► IBM DB2 Universal Database V8.2
- ► IBM DB2 Universal Database Express V8.1
- ► IBM DB2 Universal Database Express V8.2
- ► IBM DB2 Universal Database for iSeries V4R5
- ► IBM DB2 Universal Database for iSeries V5R1
- ► IBM DB2 Universal Database for iSeries V5R2
- ► IBM DB2 Universal Database for iSeries V5R3
- ► IBM DB2 Universal Database for z/OS and OS/390 V7
- ► IBM DB2 Universal Database for z/OS V8
- ► Informix Dynamic Server V7.3
- ► Informix Dynamic Server V9.2
- ► Informix Dynamic Server V9.3
- ► Informix Dynamic Server V9.4
- ► Microsoft SQL Server V7.0
- ► Microsoft SQL Server 2000
- ► Oracle8i V8.1.7
- ► Oracle9i
- ► Oracle10g
- ► Sybase Adaptive Server Enterprise V12
- ► Sybase Adaptive Server Enterprise V12.5

3.2.4 Rational Web Developer V6 product packaging

Table 3-2 lists the software CDs included with IBM Rational Web Developer V6.0.

Table 3-2   IBM Rational Web Developer V6.0 product packaging

CD title	Windows	Linux
IBM Rational Application Developer V6.0 core installation files	X	X
IBM WebSphere Application Server V6.0 Integrated Test Environment	X	X
IBM WebSphere Application Server V5.0 Integrated Test Environment	X	X
Language Pack	X	X
IBM Rational Agent Controller **Note**: Support for many additional platforms	X	X

3.2.5 Rational Web Developer tools

This section provides a brief description for the following tooling including with the Rational Application Developer:

- ► Web development tools
- ► Relational database tools

- XML tools
- Java development tools (JDT)
- Web services development tools
- Enterprise Generation Language (EGL) tools
- Team collaboration
- Debugging tools
- Performance profiling tools
- Server tools for testing and deployment
- Plug-in development tools

Web development tools

The professional Web development environment provides the necessary tools to develop sophisticated Web applications consisting of static HTML pages, JSPs, servlets, XML deployment descriptors, and other resources.

Wizards are available to generate ready to run Web applications based on SQL queries and JavaBeans. Links between Web pages can be automatically updated when content is moved or renamed.

The Web development environment brings all aspects of Web application development into one common interface. Everyone on your Web development team, including content authors, graphic artists, programmers, and Web masters, can work on the same projects and access the files they need.

Such an integrated Web development environment makes it easy to collaboratively create, assemble, publish, deploy, and maintain dynamic, interactive Web applications.

The Web development tools provide the following features:

- Support for latest Web technology with an intuitive user interface
- Advanced scripting support to create client-side dynamic applications with VBScript or JavaScript™
- Web Art Designer to create graphic titles, logos, buttons, and photo frames with professional-looking touches
- Animated GIF Designer to create life-like animation from still pictures, graphics, and animated banners
- Over 2,000 images and sounds in the built-in library
- Integrated, easy-to-use visual layout tool for JSP and HTML file creation and editing
- Web project creation, using the J2EE-defined hierarchy
- Creation and visual editing of the Web application deployment descriptor file (web.xml)

- ▶ Automatic update of links as resources are moved or renamed
- ▶ A wizard for creating servlets
- ▶ Generation of Web applications from database queries and JavaBeans
- ▶ J2EE WAR/EAR deployment support
- ▶ Integration with the WebSphere unit test environment

Relational database tools

The database tools provided with the WebSphere family products allow you to create and manipulate the data design for your project in terms of relational database schemas.

You can explore, import, design, and query databases working with a local copy of an already existing design. You can also create an entirely new data design from scratch to meet your requirements.

The database tools provide a metadata model used by all other tools that need relational database information, including database connection information. In that way, tools, although unaware of each other, are able to share connections.

The SQL Statement Wizard and SQL Query Builder provide a GUI-based interface for creating and executing SQL statements. When you are satisfied with your statement, you can use the SQL to XML Wizard to create an XML document, as well as XSL, DTD, XSD, HTML, and other, related artifacts.

The relational database tools support connecting to, and importing from, several database types, including DB2, Oracle, SQL Server, Sybase, and Informix.

XML tools

The comprehensive XML toolset provided by the WebSphere Studio family of products includes components for building DTDs, XML Schema Definition(XSD) and XML files. With the XML tools you can perform all of the following tasks:

- ▶ Create, view, and validate DTDs, XML schemas, and XML files.
- ▶ Create XML documents from a DTD, from an XML schema, or from scratch.
- ▶ Generate JavaBeans from a DTD or XML schema.
- ▶ Define mappings between XML documents and generate XSLT scripts that transform documents.
- ▶ Create an HTML or XML document by applying an XSL style sheet to an XML document.
- ▶ Map XML files to create an XSL transformation script and to visually step through the XSL file.

- ► Define mappings between relational tables and DTD files, or between SQL statements and DTD files, to generate a document access definition (DAD) script, used by IBM DB2 XML Extender. This can be used either to compose XML documents from existing DB2 data or to decompose XML documents into DB2 data.
- ► Generate DADX, XML, and related artifacts from SQL statements and use these files to implement your query in other applications.

Java development tools (JDT)

All WebSphere Studio family of products provide a professional-grade Java development environment with the following capabilities:

- ► Application Developer Version 6 ships with the IBM JRE V1.4.2
- ► Incremental compilation
- ► Ability to run code with errors in methods
- ► Crash protection and auto-recovery
- ► Error reporting and correction
- ► Java text editor with full syntax highlighting and complete content assist
- ► Refactoring tools for reorganizing Java applications
- ► Intelligent search, compare, and merge tools for Java source files
- ► Scrapbook for evaluating code snippets
- ► Pluggable run-time support for JRE switching and targeting of multiple run-time environments from IBM and other vendors

Web services development tools

Web services represent the next level of function and efficiency in e-business. Web services are modular, standards-based e-business applications that businesses can mix and match dynamically in order to perform complex transactions with minimal programming.

The WebSphere Studio family of products that include the Web services feature help you to build and deploy Web services-enabled applications across the broadest range of software and hardware platforms used by today's businesses. These tools are based on open, cross-platform standards such as SOAP, Web Services Definition Language (WSDL), and Universal Description, Discovery and Integration (UDDI).

Enterprise Generation Language (EGL) tools

IBM Enterprise Generation Language (EGL) is a business application-centric procedural programming language and environment used to develop batch, text user interface (TUI), and Web applications. When developing an EGL Web application, the developer creates the EGL source files using wizards and the source editor. Java/J2EE source is generated from the EGL source files so that the application can then be deployed to WebSphere Application Server.

Team collaboration

Team developers do all of their work in their individual Workbenches, and then periodically release changes to the team code. This model allows individual developers to work on a team project, share their work with others as changes are made, and access the work of other developers as the project evolves. At any time, developers can update their Workbenches by retrieving the changes that have been made to the team code.

All products of the WebSphere Studio family support the Concurrent Versions System (CVS) and the Rational ClearCase LT products.

Other software configuration management (SCM) repositories can be integrated through the Eclipse Workbench SCM adapters. SCM adapters for commercial products are provided by the vendors of those products.

Debugging tools

The Rational Software Development Platform tools include a debugger that enables you to detect and diagnose errors in your programs running either locally or remotely. The debugger allows you to control the execution of your program by setting breakpoints, suspending launches, stepping through your code, and examining the contents of variables.

You can debug live server-side code as well as programs running locally on your workstation.

The debugger includes a debug view that shows threads and stack frames, a process view that shows all currently running and recently terminated processes, and a console view that allows developers to interact with running processes. There are also views that display breakpoints and allow you to inspect variables.

Performance profiling tools

The Rational Software Development Platform products provide tools that enable you to test the performance of your application. This allows you to make architectural and implementation changes early in your development cycle, and significantly reduces the risk of finding serious problems in the final performance tests.

The profiling tools collect data related to a Java program's run-time behavior, and present this data in graphical and non-graphical views. This assists you in visualizing program execution and exploring different patterns within the program.

These tools are useful for performance analysis and for gaining a deeper understanding of your Java programs. You can view object creation and garbage collection, execution sequences, thread interaction, and object references. The

tools also show you which operations take the most time, and help you find and plug memory leaks. You can easily identify repetitive execution behavior and eliminate redundancy, while focusing on the highlights of the execution.

Server tools for testing and deployment

The server tools provide a unit test environment where you can test JSPs, servlets and HTML files, (EJB testing is supported in Application Developer). You also have the capability to configure other local or remote servers for integrated testing and debugging of J2EE applications.

The following features are included:

► A copy of the complete WebSphere Application Server run-time environment
► Standalone unit testing
► Ability to debug live server-side code using the integrated debugger
► Support for configuring multiple servers

Plug-in development tools

The Rational Software Development Platform includes the Plug-in Development Environment (PDE) that is designed to help you develop platform plug-ins while working inside the platform Workbench and it provides a set of platform extension contributions (views, editors, perspectives, and so on) that collectively streamline the process of developing plug-ins inside the Workbench. The PDE is not a separate tool, but it is a perspective.

The following project types are supported:

► Plug-in project

   Application Developer is based on the concept of plug-ins that have a clearly defined structure and specification. This project supports the ability to create, test, and deploy a plug-in in the PDE.

► Fragment project

   A plug-in fragment is used to provide additional plug-in functionality to an existing plug-in after it has been installed. Fragments are ideal for shipping features such as language or maintenance packs that typically trail the initial products by a few months.

► Plug-in component

   PDE attaches a special component nature to plug-in and fragment projects to differentiate them from other project types. The project must have a specific folder structure and a component manifest. The project must be set up with references to all of the plug-in and fragment projects that will be packaged into the component.

3.2.6 WebSphere Application Server licensing and platforms

Table 3-3 shows the features included with each WebSphere Application Server packaging option. The WebSphere Application Server - Express package is highlighted in grey.

Table 3-3   WebSphere Application Server packaging and license terms

	Express V6	**Base V6**	**ND V6**
Licensing terms	Limited to a max of 2 CPUS	Unlimited CPUs	Unlimited CPUs
WebSphere Application Server	Yes	Yes	Yes
Network Deployment	No	No	Yes
IBM HTTP Server V6 Web server plug-ins	Yes	Yes	Yes
IBM HTTP Server	Yes	Yes	Yes
Application Client (not on zLinux)	Yes	Yes	Yes
Application Server Toolkit	Yes	Yes	Yes
DataDirect Technologies JDBC Drivers for WebSphere Application Server	Yes	Yes	Yes
Development tools	Rational Web Developer (single use license)	Rational Application Developer Trial	Rational Application Developer Trial
Database	IBM DB2 Universal Database Express V8.2 (limited production use only[a])	IBM DB2 Universal Database Express V8.2 (development use only)	IBM DB2 UDB Enterprise Server Edition V8.2 for WebSphere Application Server Network Deployment
Production ready applications	IBM Business Solutions	No	No
Note Not all features are available on all platforms. See the System Requirements Web page for each WebSphere Application Server package for more information.			

	Express V6	Base V6	ND V6
Tivoli Directory Server for WebSphere Application Server (LDAP server)	No	No	Yes
Tivoli Access Manager Servers for WebSphere Application Server	No	No	Yes
Edge Components	No	No	Yes
Note Not all features are available on all platforms. See the System Requirements Web page for each WebSphere Application Server package for more information.			

a. The DB2 license only allows the database to be used for storing technical data such as session persistence, UDDI Registry data, messaging engine data and data for the Scheduler and EJB Timer components. Note: It is explicitly not permitted to use it to build a custom data repository, meaning productive business data.

WebSphere Application Server - Express is supported on the following operating system platforms:

► AIX
► H-UX
► Linux (Intel)
► Linux on iSeries
► Linux on pSeries
► Solaris™
► Windows

WebSphere Application Server - Express supports the following LDAP servers: IBM Directory Server

► z/OS Security Server
► Lotus Domino Enterprise Server
► Sun™ ONE™ Directory Server
► Windows Active Directory 2000 and 2003
► NDS eDirectory

WebSphere Application Server - Express supports the following database servers:

► IBM DB2
► Cloudscape
► Oracle
► Sybase
► Microsoft SQL Server

► Informix

WebSphere Application Server - Express supports the following Web servers:

► IBM HTTP Server
► Apache Server
► Microsoft Internet Information Services
► Lotus Domino Enterprise Server
► Sun ONE Web Server, Enterprise Edition
► Sun Java™ System Web Server
► Covalent Enterprise Ready Server

For the exact operating system levels and requirements, see the following Web site:

http://ibm.com/software/webservers/appserv/express/requirements

Supported operating system platforms

IBM Rational Application Developer V6.0 supports the following operating systems:

► Microsoft Windows:

  – Windows XP with Service Packs 1 and 2
  – Windows 2000 Professional with Service Packs 3 and 4
  – Windows 2000 Server with Service Packs 3 and 4
  – Windows 2000 Advanced Server with Service Packs 3 and 4
  – Windows Server 2003 Standard Edition
  – Windows Server 2003 Enterprise Edition

► Linux on Intel:

  – Red Hat Enterprise Linux Workstation V3 (all service packs)
  – SuSE Linux Enterprise Server (SLES) V9 (all service packs)

3.2.7 New in WebSphere Application Server - Express V6

WebSphere Application Server - Express V6 continues with the tradition of providing support for the current J2EE specifications. In addition, it focuses on features that provide ease-of-use, simplification of application development and deployment, high availability, and flexibility. The following sections give you the highlights of the new features and functionality provided with WebSphere Application Server - Express V6.

Programming support

The following are highlights of the new application programming features for WebSphere Application Server - Express V6:

J2EE 1.4 support

WebSphere Application Server - Express V6 provides full support for J2EE 1.4. The J2EE specification requires a certain set of specifications to be supported. Among these are EJB 2.1, JMS 1.1, JCA 1.5, Java Servlets 2.4, and JSP 2.0.

WebSphere Application Server - Express V6 also provides support for J2EE 1.2 and 1.3 to ease migration.

WebSphere Application Server - Express V6 is shipped with JDK™ 1.4.2, which includes the new Java Web Start feature. Java Web Start is an application-deployment technology that includes the portability of applets, the maintainability of servlets and JavaServer™ Pages™ (JSP) file technology, and the simplicity of mark-up languages such as XML and HTML. It is a Java application that allows full-featured Java 2 client applications to be launched, deployed and updated from a standard Web server.

Web services

Web services support has been updated to include the latest in technology options, including:

- ► *Java API for XML-based RPC (JAX-RPC) 1.1:* This specification enables you to develop SOAP-based interoperable and portable Web services and Web service clients. The JAX-RPC programming model is defined by the Web services standard JSR 101.

- ► *Web services for Java 2 Platform, Enterprise Edition:* This specification defines the programming model and run-time architecture for implementing Web services based on the Java language.JSR 109 - WSEE

- ► *SOAP with Attachments API for Java (SAAJ) 1.2:* This specification is used for SOAP messaging that works behind the scenes in the JAX-RPC implementation.

- ► *Web Services Security (WS-Security):* This specification proposes a standard set of SOAP extensions that you can use to build secure Web services.

- ► *Web Services-Interoperability (WS-I) Basic Profile 1.1:* This is a set of non-proprietary Web services specifications that promote interoperability. The *Web Services-Interoperability (WS-I) Attachments Profile* compliments the WS-I Basic Profile 1.1 to add support for interoperable SOAP messages with attachments-based Web services.

- ► *Java API for XML Registries (JAXR) 1.0:* This specification defines a Java client API for accessing both UDDI (Version 2 only) and ebXML registries.

- ► *Universal Description, Discovery and Integration (UDDI) V3:* The UDDI specification defines a way to publish and discover information about Web services.

In addition, WebSphere Application Server - Express V6 adds the following value-add to the standards:

► Custom bindings to supplement JAX-RPC features, allowing you to create your own custom bindings needed to map Java to XML and XML to Java conversions

► Support for generic SOAP elements

► Multi-protocol support for using a stateless session EJB as the Web service provider providing enhanced performance without changes to JAX-RPC clients

The private UDDI Registry previously shipped with the V5 Network Deployment package, is now available in all packages and implements V3.0 of the UDDI specification.

Service Data Objects (SDO)

SDO, formerly WebSphere Data Objects, provides unified data access and representation across heterogeneous data stores. With SDO, data mediators perform the actual work of accessing the data stores. Clients query a data mediator service and get a data graph in response. The data graph consists of structured data objects representing the data store. Clients update the data graph and send it back to the mediator service to have the updates applied.

SDO is not intended to replace other data access technologies, but rather to provide an alternate choice. It has the advantage of simplifying the application programming tasks required to access data stores.

SDO support is included in WebSphere Studio Application Developer 5.1.1 and in Rational Application Developer 6.0. This support includes:

► Wizards and views for working with data objects
► Relational data lists
► Relational data objects

WebSphere Application Server 6.0 support for SDO includes:

► Support for SDO naming and packaging
► Externalization of the following APIs
  – SDO Core APIs
  – JDBC Data Mediator for relational databases supported by WebSphere Application Server
  – EJB Mediator for entity EJBs

SDO is defined by JSR 235. For more information, see:

```
ftp://www6.software.ibm.com/software/developer/library/j-commonj-sdowmt
/Commonj-SDO-Specification-v1.0.doc
```

JavaServer Faces (JSF) v1.0

JavaServer Faces (JSF) is a user interface framework or API that eases the development of Java-based Web applications. JSF makes J2EE more approachable to non-Java application developers with HTML, scripting, and page layout skills.

WebSphere Application Server - Express V6 supports JavaServer Faces 1.0 at a runtime level.

Programming Model Extensions (PME)

Programming Model Extensions (PME) formerly found in more advanced WebSphere Application Server packaging options are now available in the Express, Base, and Network Deployment packages.

In Base and Express the following PMEs are available:

► Last Participant Support
► Internationalization Service
► WorkArea Service
► ActivitySession Service
► Extended JTA Support
► Startup Beans
► Asynchronous Beans (now called WorkManager)
► Scheduler Service (now called Timer Service)
► Object Pools
► Dynamic Query
► Web Services Gateway Filter Programming Model (with
► migration support)
► Distributed Map
► Application Profiling

In Network Deployment the following PMEs are available :

► Back-up Cluster Support
► Dynamic WLM

System management

WebSphere Application Server - Express V6 has enhanced the usability of the WebSphere administration tools. There is also a focus on enhanced application deployment features.

The following are highlights of the new system management features for WebSphere Application Server - Express V6:

Improved system management model

Several improvements have been made to the basic system management features of WebSphere Application Server V6. These improvements are:

► Configuration archiving allows you to create a complete or partial archive of an existing WebSphere Application Server configuration. This archive is portable and can be used to create new configurations based on the archive.

► A WebSphere Application Server instance is now defined by a profile. After installing the product, you create the runtime environment by building profiles. Using profiles allows you to easily configure multiple runtimes with one set of install libraries.

► You can now define a generic server as an application server instance in the administration tools and associate it with a non-WebSphere server or process that is needed to support the application server environment.

► You can also define external Web servers. As a managed server, you can start and stop the Web server and automatically push the plug-in configuration to it. This requires a node agent to be installed on the machine and is typically used when the Web server is behind a firewall.

  You can also define a Web server as an unmanaged server for placement outside the firewall. This allows you to create custom plug-ins for the Web server but you must manually move the plug-in configuration to the Web server machine.

  As a special case, you can define the IBM HTTP Server as an unmanaged server, but treat it as a managed server. This does not require a node agent since the commands are sent directly to the IBM HTTP Server administration process.

Administrative console updates

The WebSphere Administrative Console has been updated with ease of use in mind. New panels have been added to facilitate the new V6 features such as service integration, the integrated UDDI Registry, and the new Web server options. The navigation has been reworked to reduce the number of clicks required to reach most configuration settings.

Application management

Improvements in application management techniques have been added to facilitate rapid application deployment and efficient update procedures.

► Enhanced EAR files can now be built using Rational Application Developer or the Application Server Toolkit. The enhanced EAR contains bindings and server configuration settings previously done at deployment time. This allows

developers to predefine known run-time settings and can speed up deployment.

> **Note:** Rational Application Developer V6 is not included in the WebSphere Application Server - Express V6 package. Instead, the Rational Web Developer is included. With this tool you can deploy EAR files. It provides support for editing deployment descriptors for Web applications. What is missing is the support for creating and editing of EJB projects. If you need that support in the development tool be reminded that you can separately purchase the Rational Application Developer, that seamlessly integrates with WebSphere Application Server - Express. Rational Application Developer is a super-set of Rational Web Developer. You can see the relationship in Figure 3-3 on page 40.

► Fine grain application update capabilities allow you to make small delta changes to applications without doing a full application update and restart.

► WebSphere Rapid Deployment provides the ability for developers to use annotation based programming. This is a step forward in the automation of application development and deployment.

Service integration
The service integration functionality within WebSphere Application Server supports both message-oriented and service-oriented applications. The primary component of this functionality is the service integration bus, which provides the support for messaging and Web services applications. One or more application servers or clusters join a bus to become bus members. The service integration bus becomes a component of the Enterprise Service Bus (ESB).

The service integration functionality provides:

► A fully compliant J2EE 1.4 JMS messaging provider, integrated within the application server

This messaging provider is the default messaging provider for the application server. It can support multi-server configurations and can be linked to WebSphere MQ, appearing as a queue manager. This new JMS provider replaces the embedded JMS provider available in WebSphere Application Server V5.

► An integrated Web services infrastructure and support for the Web Services Gateway provide you with a single point of control, access and validation of Web service requests. With these features, you can control which Web services are available to different groups of Web service users.

The service integration bus is fully integrated with the administration tools, WebSphere security, installation processes, and provides support for clustering enablement.

Security enhancements

Updates to security features in WebSphere Application Server V6 include:

► Java Authorization Contract for Containers (JACC) 1.0 support details the contract requirements for J2EE containers and authorization providers. With this detail, authorization providers can perform the access decisions for resources in J2EE 1.4 application servers such as WebSphere Application Server. This support facilitates the plug-in of third-party authorization servers.

► WebSphere Application Server Version 6.0 provides an embedded IBM Tivoli Access Manager client that is JACC compliant and can be used to access a Tivoli Access Manager server for authentication and authorization.

> **Note:** Tivoli Access Manager server is bundled in the Network Deployment package and not included in WebSphere Application Server - Express.

3.2.8 Physical Packaging

WebSphere Application Server - Express contains 18 CDs including seven Primary CDs and 11 Supplemental CDs. The primary CDs include the base application server with a separate CD for each supported platform for a total of seven CDs.

Each CD contains:

► Standalone Node

This is an application server instance, configured by default to operate as a standalone server

► IBM HTTP Server

Although, this is no longer installed automatically with the WebSphere Application Server Install, IBM HTTP Server can be installed separately with its own installation program.

► Web server plug-ins for all of the Web servers supported by WebSphere Application Server

Although these are no longer integrated with the WebSphere Application Server installation; they can be installed separately with their own installation program.

► Application Client supporting the J2EE client container and JNLP (Java Network Launch Protocol)

► Data Direct JDBC Driver1 supporting SQL Server

The supplemental CDs include:

► Six CDs containing the Rational Web Developer (previously called WebSphere Studio Site Developer) product for single developer use

► CDs containing DB2 Express

► Two CDs containing a set of production-ready applications, such as the IBM Telephone Directory, for use on Windows or Linux on Intel, referred to as IBM Business Solutions

In addition, the Application Server Toolkit is available for download from WebSphere Developer Domain.

3.3 WebSphere Application Server architecture

Even though this book is dealing with the WebSphere Application Server - Express product package and neither the Base configuration nor the Network Deployment configuration are included in this package, we discuss all three configurations in the following sections.

One reason to discuss the three packages here is to complete the overview of the WebSphere Application Server family and to give you some idea of possibilities of upgrade paths when you want to think about extending your environment. Another reason is that it is important to understand the basic concepts of the WebSphere Application Server architecture, because they are the underlying fundamentals of all three packaging options.

The following sections discuss the architecture of WebSphere Application Server and will mention what is included and the differences between the different packaging options.

3.4 Application server configurations

At the heart of each member of the WebSphere Application Server family you will find an application server with essentially the same architectural structure.

While the application server structure is identical for Base and Express, there are differences in licensing terms, the development tool provided, and platform support. With Base and Express, you are limited to standalone application servers. Each standalone application server provides a fully functional J2EE 1.4 environment.

Network Deployment adds additional elements that allow for more advanced topologies, providing workload management, scalability, high availability, and central management of multiple application servers.

3.4.1 Standalone server configuration

Express, Base, and Network Deployment all support a single standalone server environment, though with Express and Base standalone is your only option. In this configuration, each application server acts as a unique entity. An application server runs one or more J2EE applications and provides the services required to run those applications.

Multiple standalone application servers can exist on a machine, either through independent installations of the WebSphere Application Server code, or through the creation of multiple configuration profiles within one installation. However, there is no common management or administration provided for multiple application servers. Standalone application servers do not provide workload management or failover capabilities.

Figure 3-5 on page 60 shows an architectural overview of a standalone application server.
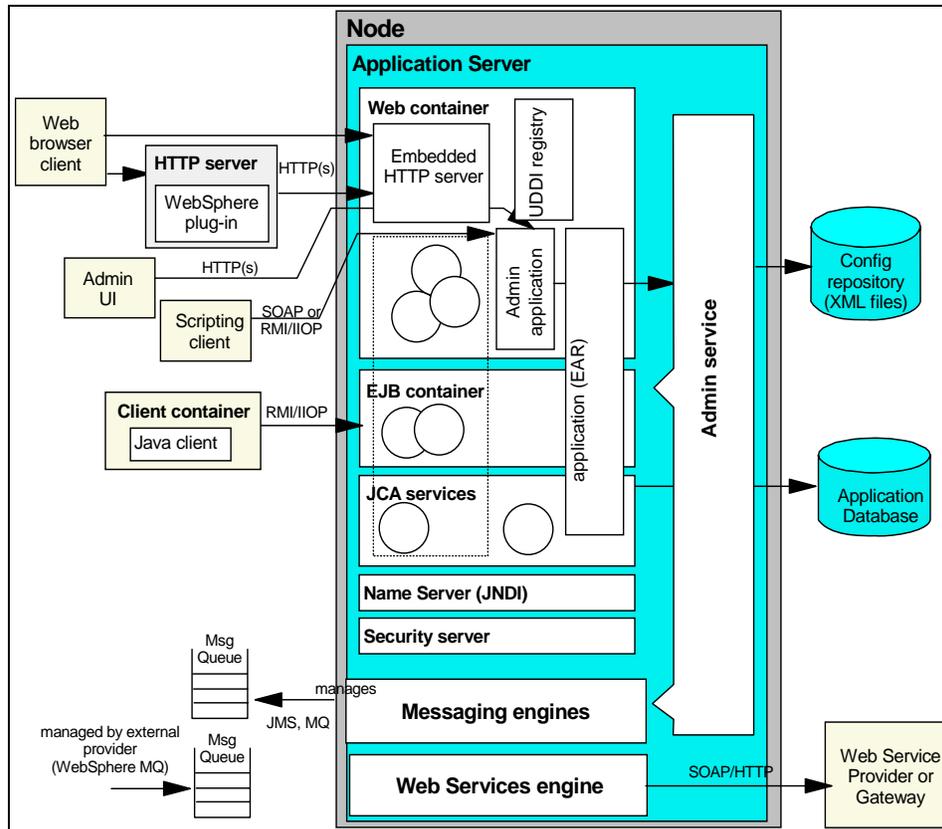
Figure 3-5   Standalone server

3.4.2 Distributed server configuration

With Network Deployment, you can build a distributed server configuration, enabling central administration, workload management, and failover. One or more application servers are federated (integrated) into a cell and managed by a deployment manager. The application servers can reside on the same machine as the deployment manager or on multiple separate machines. Administration and management is done centrally from the administration interfaces using the deployment manager.

With this configuration, multiple application servers can be created to run unique sets of applications and managed from a central location. But more importantly, application servers can be clustered for workload management and failover capabilities. Applications installed to the cluster are replicated across the application servers. When one server fails, another server in the cluster can

continue processing. Workload is distributed among Web containers and EJB containers in a cluster using a weighted round-robin scheme.

Figure 3-6 illustrates the basic components of an application server in a distributed server environment.
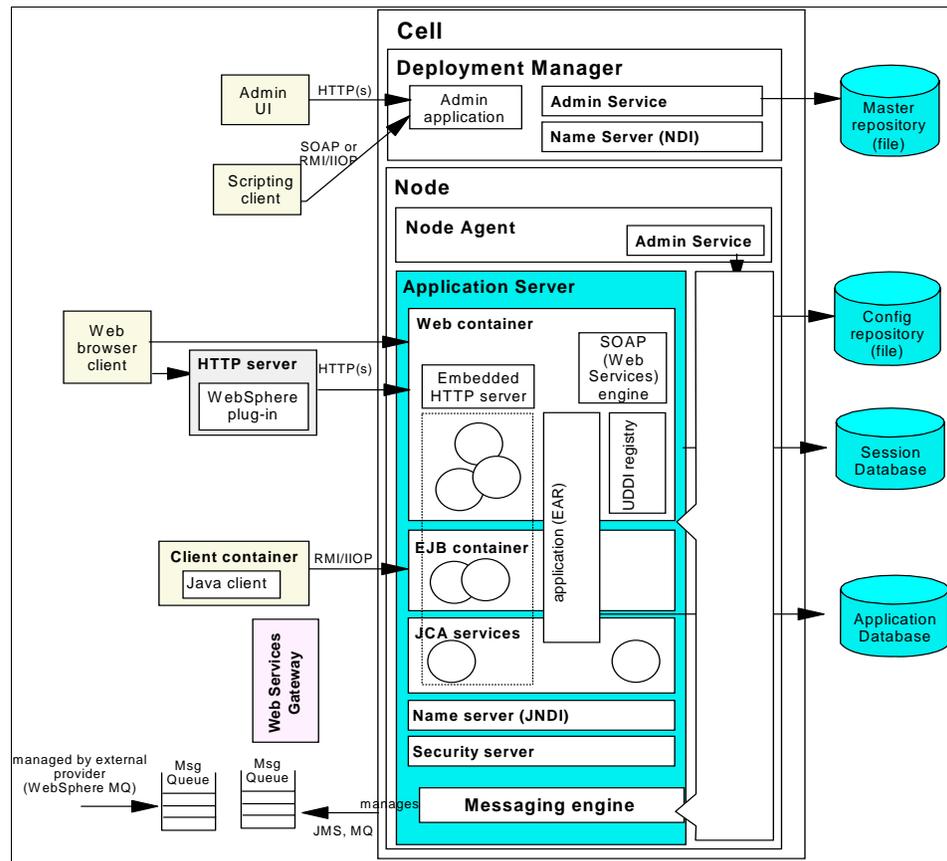
Figure 3-6   Distributed server environment

3.5 Cells, nodes and servers

Regardless of the configuration, the WebSphere Application Server is organized based on the concepts of cells, nodes and servers. While all these elements are present in each configuration, cells and nodes do not play an important role until you start taking advantage of the features provided with Network Deployment.

Application servers

The *application server* is the primary run-time component in all configurations. This is where the application actually executes. All WebSphere Application Server configurations can have one or more application servers. In the Express and Base configurations, each application server functions as a separate entity. There is no workload distribution or common administration among application servers. With Network Deployment you can build a distributed server environment consisting of multiple application servers maintained from a central administration point. In a distributed server environment, application servers can be clustered for workload distribution.

Nodes, node groups, and node agents

A *node* is a logical grouping of WebSphere-managed server processes that share common configuration and operational control. A node is associated with one physical installation of WebSphere Application Server. In a standalone application server configuration, there is only one node.

With Network Deployment, you can configure multiple nodes to be managed from one common administration server. In these centralized management configurations, each node has a node agent that works with a deployment manager to manage administration processes.

Node group is a new concept with V6. A *node group* is a grouping of nodes within a cell that have similar capabilities. Its purpose is to validate that the node is capable of performing certain functions before allowing them. For example, a cluster cannot contain both z/OS and nonz/OS nodes. In this case, multiple node groups would be defined, one for the z/OS nodes and one for nonz/OS. A DefaultNodeGroup is automatically created based on the deployment manager platform. This node group contains the deployment manager and any new nodes with the same platform type.

Cells

A *cell* is a grouping of nodes into a single administrative domain. In the Base and Express configurations, a cell contains one node. That can may have multiple servers, but the configuration files for each server are stored and maintained individually.

In a distributed server configuration, a cell can consist of multiple nodes, all administered from a single point. The configuration and application files for all nodes in the cell are centralized into a cell master configuration repository. This centralized repository is managed by the deployment manager process and synchronized out to local copies held on each of the nodes.

3.6 Servers

WebSphere Application Server supplies application servers to provide the functions required to host applications. It also provides the ability of defining external servers to the administration process. Table 3-4 shows the types of servers that can be defined to the WebSphere Application Server administration tools.

Table 3-4   WebSphere Application Server server support

WebSphere Application Server types	Express & Base	ND
Application server	Yes	Yes
Application server clustering	No	Yes
External Web server	Yes	Yes
External generic server	No	Yes
WebSphere V5 JMS servers	No	Yes

3.6.1 Application server

Application servers provide the run-time environment for application code. They provide containers and services that specialize in enabling the execution of specific Java application components. Each application server runs in its own Java virtual machine (JVM™).

3.6.2 Clusters

Network Deployment offers the option of using application server clustering to provide enhanced workload distribution. A *cluster* is a logical collection of application server processes, with the purpose of providing workload balancing and high availability.

Application servers that belong to a cluster are members of that cluster and must all have identical application components deployed on them. Other than the applications configured to run on them, cluster members do not have to share any other configuration data.

For example, one cluster member might be running on a large multi-processor server while another member of that same cluster might be running on a small mobile computer. The server configuration settings for each of these two cluster members are very different, except in the area of application components assigned to them. In that area of configuration, they are identical.

The members of a cluster can be located on a single node (*vertical* cluster), across multiple nodes (*horizontal* cluster) or on a combination of the two.

When you install, update, or delete an application, the updates are automatically distributed to all members in the cluster. In V5, updating an application on a cluster required stopping the application on every server in the cluster, installing the update, then restarting the server. In V6, you have a rollout update option that will update and restart the application servers on each node, one node at a time. This provides continuous availability of the application.

3.6.3 JMS servers (V5)

In V5, JMS servers provided the default messaging support for WebSphere Application Server. For migration purposes, Network Deployment in V6 supports cells that contain both V5 and V6 nodes (the deployment manager must be at V6) and by extension, supports existing JMS servers in V5 application servers in the cell.

3.6.4 External servers

You have the ability to define servers other than WebSphere application servers to the administrative process.

► Generic servers

A *generic server* is a server that is managed in the WebSphere administrative domain, although it is not a server supplied by the WebSphere Application Server product.The generic server can be any server or process that is necessary to support the application server environment, including a Java server, a C or C++ server or process, a CORBA server, or a Remote Method Invocation (RMI) server.

► Web servers

Web servers can be defined to the administration process as a Web server node. This allows applications to be associated with one or more defined Web servers. Web server nodes can be managed or unmanaged.

Managed nodes have a node agent on the Web server machine, allowing the deployment manager to administer the Web server. The Web server can be started or stopped from the deployment manager. The Web server plug-in for the node can be generated and automatically pushed to the Web server. You would normally have managed Web server nodes behind the firewall with the WebSphere Application Server installations.

Unmanaged Web server nodes are not managed by WebSphere. You normally find these outside the firewall or in the DMZ. Web server plug-in files

must be manually copied or FTPed to the Web server. However, defining the Web server as a node allows you to generate custom plug-in files for it.

As a special case, if the unmanaged Web server is an IBM HTTP Server, you can administer the Web server from the WebSphere administrative console and automatically push the plug-in configuration file to the Web server. This is done by the deployment manager using HTTP commands to the IBM HTTP Server administration process and does not require a node agent.

3.7 Containers

The J2EE 1.4 specification defines the concept of containers to provide runtime support for applications. There are two containers in the application server implementation:

► Web container - processes HTTP requests, servlets and JSPs
► EJB container - processes enterprise beans (EJB)

In addition, there is an application client container that can run on the client machine.

Table 3-5   WebSphere Application Server container support

Container type	Express & Base	ND
Web container	Yes	Yes
EJB container	Yes	Yes
Application client container	Yes	Yes

3.7.1 Web container

The Web container processes servlets, JSP files and other types of server-side includes. Each application server run time has one logical Web container, which can be modified, but not created or removed. Each Web container provides the following:

► Servlet processing

When handling servlets, the Web container creates a request object and a response object, then invokes the servlet service method. The Web container invokes the servlet's destroy method when appropriate and unloads the servlet, after which the JVM performs garbage collection.

- ► Embedded HTTP server

  The Web container runs an embedded HTTP server for handling HTTP(S) requests from external Web server plug-ins or Web browsers. The embedded Web server is based on the IBM HTTP Server product.

  Directing client requests to the embedded Web server is useful for testing or development purposes. In the simplest of configurations, this may even be appropriate for production use. However, in most cases, the use of an external Web server and Web server plug-in as a front-end to the Web container is more appropriate for a production environment.

- ► Session management

  Support is provided for the javax.servlet.http.HttpSession interface described in the Servlet API specification.

- ► Web services engine

  Web services are provided as a set of APIs in cooperation with the J2EE applications. Web services engines are provided to support SOAP.

Web server plug-ins

Although the Web container has an embedded HTTP server, a more likely scenario is that an external Web server will be used to receive client requests. The Web server can serve requests that do not require any dynamic content, for example, HTML pages. However, when a request requires dynamic content (JSP/servlet processing), it must be forwarded to WebSphere Application Server for handling.

The mechanism to accomplish this is provided in the form of a Web server plug-in. The plug-in is included with the WebSphere Application Server packages for installation on a Web server. An XML configuration file, configured on the WebSphere Application Server, is copied to the Web server plug-in directory. The plug-in uses the configuration file to determine whether a request should be handled by the Web server or an application server. When a request for an application server is received, it is forwarded to the appropriate Web container in the application server. The plug-in can use HTTP or HTTPs to transmit the request.

3.7.2 EJB container

The EJB container provides all the runtime services needed to deploy and manage enterprise beans. It is a server process that handles requests for both session and entity beans.

The enterprise beans (packaged in EJB modules) installed in an application server do not communicate directly with the server. Instead, the EJB container provides an interface between the EJBs and the server. Together, the container and the server provide the bean run-time environment.

The container provides many low-level services, including threading and transaction support. From an administrative viewpoint, the container manages data storage and retrieval for the contained beans. A single container can host more than one EJB JAR file.

3.7.3 Client application container

The client application container is a separately installed component on the client's machine. It allows the client to run applications in an EJB-compatible J2EE environment.

There is a command-line executable (`launchClient`) that is used to launch the client application along with its client container runtime.

3.8 Application server services

The application server provides other services besides the containers. For a list of these see Table 3-6.

Table 3-6   WebSphere Application Server services

Service	Express & Base	ND
JCA services	Yes	Yes
Transaction service	Yes	Yes
Dynamic cache service	Yes	Yes
Message listener service	Yes	Yes
ORB service	Yes	Yes
Administration service (JMX™)	Yes	Yes
Diagnostic trace service	Yes	Yes
Debugging service	Yes	Yes
Name service (JNDI)	Yes	Yes
a These services are Programing Model Extensions.		

Service	Express & Base	ND
Performance Monitoring Infrastructure (PMI) service	Yes	Yes
Security service (JAAS and Java 2 security)	Yes	Yes
Service integration bus (SIB) service	Yes	Yes
Application profiling service [a]	Yes	Yes
Compensation service [a]	Yes	Yes
Internationalization service [a]	Yes	Yes
Object pool service [a]	Yes	Yes
Startup beans service [a]	Yes	Yes
Activity session service [a]	Yes	Yes
Work area partition service [a]	Yes	Yes
Work area service [a]	Yes	Yes
[a] These services are Programing Model Extensions.		

3.8.1 JCA services

Connection management for access to Enterprise Information Systems (EIS) in WebSphere Application Server is based on the J2EE Connector Architecture (JCA) specification, also sometimes referred to as J2C. The connection between the enterprise application and the EIS is done through the use of EIS-provided resource adapters, which are plugged into the application server. The architecture specifies the connection management, transaction management, and security contracts that exist between the application server and EIS.

The Connection Manager in the application server pools and manages connections. It is capable of managing connections obtained through both resource adapters defined by the JCA specification and data sources defined by the JDBC 2.0 Extensions Specification.

3.8.2 Transaction service

WebSphere applications can use transactions to coordinate multiple updates to resources as one unit of work such that all or none of the updates are made permanent. Transactions are started and ended by applications or the container in which the applications are deployed.

WebSphere Application Server is a transaction manager that supports the coordination of resource managers through their XAResource interface and participates in distributed global transactions with transaction managers that support the CORBA Object Transaction Service (OTS) protocol (for example, application servers) or Web Service Atomic Transaction (WS-AtomicTransaction) protocol.

WebSphere Application Server also participates in transactions imported through J2EE Connector 1.5 resource adapters. WebSphere applications can also be configured interact with (or to direct the WebSphere transaction service to interact with) databases, JMS queues, and JCA connectors through their local transaction support when distributed transaction coordination is not required.

The way that applications use transactions depends on the type of application component, as follows:

► A session bean can either use container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (where the bean manages transactions itself).

► Entity beans use container-managed transactions.

► Web components (servlets) use bean-managed transactions.

In WebSphere Application Server, transactions are handled by three main components:

► A transaction manager that supports the enlistment of recoverable XAResources and ensures that each such resource is driven to a consistent outcome either at the end of a transaction or after a failure and restart of the application server.

► A container in which the J2EE application runs. The container manages the enlistment of XAResources on behalf of the application when the application performs updates to transactional resource managers (for example, databases). Optionally, the container can control the demarcation of transactions for enterprise beans configured for container-managed transactions.

► An application programming interface (UserTransaction) that is available to bean-managed enterprise beans and servlets. This allows such application components to control the demarcation of their own transactions.

3.8.3 Dynamic cache service

The dynamic cache service improves performance by caching the output of servlets, commands, Web services, and JSP files. The dynamic cache works within an application server, intercepting calls to cacheable objects, for example,

through a servlet's service() method or a command's execute() method, and either stores the object's output to or serves the object's content from the dynamic cache.

Because J2EE applications have high read-write ratios and can tolerate small degrees of latency in the currency of their data, the dynamic cache can create an opportunity for significant gains in server response time, throughput, and scalability.

The following caching features are available in WebSphere Application Server.

► Cache replication

   Cache replication among cluster members takes place using the WebSphere data replication service. Data is generated one time and copied or replicated to other servers in the cluster, thus saving execution time and resources.

► Cache disk offload

   By default, when the number of cache entries reaches the configured limit for a given WebSphere server, eviction of cache entries occurs, allowing new entries to enter the cache service. The dynamic cache includes an alternative feature named disk offload, which copies the evicted cache entries to disk for potential future access.

► Edge Side Include caching

   The Web server plug-in contains a built-in ESI processor. The ESI processor has the ability to cache whole pages, as well as fragments, providing a higher cache hit ratio. The cache implemented by the ESI processor is an in-memory cache, not a disk cache, therefore, the cache entries are not saved when the Web server is restarted.

► External caching

   The dynamic cache has the ability to control caches outside of the application server, such as that provided by the Edge components, a nonz/OS IBM HTTP Server's FRCA cache, and a nonz/OS WebSphere HTTP Server plug-in ESI Fragment Processor. When external cache groups are defined, the dynamic cache matches externally cacheable cache entries with those groups, and pushes cache entries and invalidations out to those groups. This allows WebSphere to manage dynamic content beyond the application server. The content can then be served from the external cache, instead of the application server, improving savings in performance.

3.8.4 Message listener service

With EJB 2.1, an ActivitionSpec is used to connect message-driven beans to destinations. However, existing EJB 2.0 message-driven beans can be deployed

against a listener port as in WebSphere V5. For those MDBs, the message listener service provides a listener manager that controls and monitors one or more JMS listeners. Each listener monitors a JMS destination on behalf of a deployed message-driven bean.

3.8.5 Object Request Broker service

An Object Request Broker (ORB) manages the interaction between clients and servers, using IIOP. It enables clients to make requests and receive responses from servers in a network-distributed environment.

The ORB provides a framework for clients to locate objects in the network and call operations on those objects as though the remote objects were located in the same running process as the client, providing location transparency. The client calls an operation on a local object, known as a stub. Then the stub forwards the request to the desired remote object, where the operation is run and the results are returned to the client.

The client-side ORB is responsible for creating an IIOP request that contains the operation and any required parameters, and for sending the request in the network. The server-side ORB receives the IIOP request, locates the target object, invokes the requested operation, and returns the results to the client. The client-side ORB demarshals the returned results and passes the result to the stub, which, in turn, returns to the client application, as though the operation had been run locally.

WebSphere Application Server uses an ORB to manage communication between client applications and server applications as well as communication among product components.

3.8.6 Admin service

The admin service runs within each server JVM. In Base and Express, the admin service runs in the application server. In the Network Deployment configuration, each of the following hosts an admin service:

► Deployment manager
► Node agent
► Application server

The admin service provides the necessary functions to manipulate configuration data for the server and its components. The configuration is stored in a repository in the server's file system.

The admin service has a security control and filtering functionality, providing different levels of administration to certain users or groups using the following admin roles:

- ► Administrator
- ► Monitor
- ► Configurator
- ► Operator

3.8.7 Name service

Each application server hosts a name service that provides a Java Naming and Directory Interface™ (JNDI) name space. The service is used to register resources hosted by the application server. The JNDI implementation in WebSphere Application Server is built on top of a Common Object Request Broker Architecture (CORBA) naming service (CosNaming).

JNDI provides the client-side access to naming and presents the programming model used by application developers. CosNaming provides the server-side implementation and is where the name space is actually stored. JNDI essentially provides a client-side wrapper of the name space stored in CosNaming, and interacts with the CosNaming server on behalf of the client.

The naming architecture is used by clients of WebSphere applications to obtain references to objects related to those applications. These objects are bound into a mostly hierarchical structure, referred to as a *name space*. The name space structure consists of a set of name bindings, each consisting of a name relative to a specific context and the object bound with that name. The name space can be accessed and manipulated through a name server.

The following are features of a WebSphere Application Server name space:

- ► Distributed name space

  For additional scalability, the name space for a cell is distributed among the various servers. The deployment manager, node agent and application server processes all host a name server.

  The default initial context for a server is its server root. System artifacts, such as EJB homes and resources, are bound to the server root of the server with which they are associated.

- ► Transient and persistent partitions

  The name space is partitioned into transient areas and persistent areas. Server roots are transient. System-bound artifacts such as EJB homes and resources are bound under server roots. There is a cell persistent root, which

can be used for cell-scoped persistent bindings, and a node persistent root, which can be used to bind objects with a node scope.

► Federated name space structure

A name space is a collection of all names bound to a particular name server. A name space can contain naming context bindings to contexts located in other servers. If this is the case, the name space is said to be a *federated name space*, because it is a collection of name spaces from multiple servers. The name spaces link together to cooperatively form a single logical name space.

In a federated name space, the real location of each context is transparent to client applications. Clients have no knowledge that multiple name servers are handling resolution requests for a particular requested object.

In the Network Deployment distributed server configuration, the name space for the cell is federated among the deployment manager, node agents, and application servers of the cell. Each such server hosts a name server. All name servers provide the same logical view of the cell name space, with the various server roots and persistent partitions of the name space being interconnected by means of the single logical name space.

► Configured bindings

The configuration graphical interface and script interfaces can be used to configure bindings in various root contexts within the name space. These bindings are read-only and are bound by the system at server startup.

► Support for CORBA Interoperable Naming Service (INS) object URLs

WebSphere Application Server contains support for CORBA object URLs (corbaloc and corbaname) as JNDI provider URLs and lookup names.

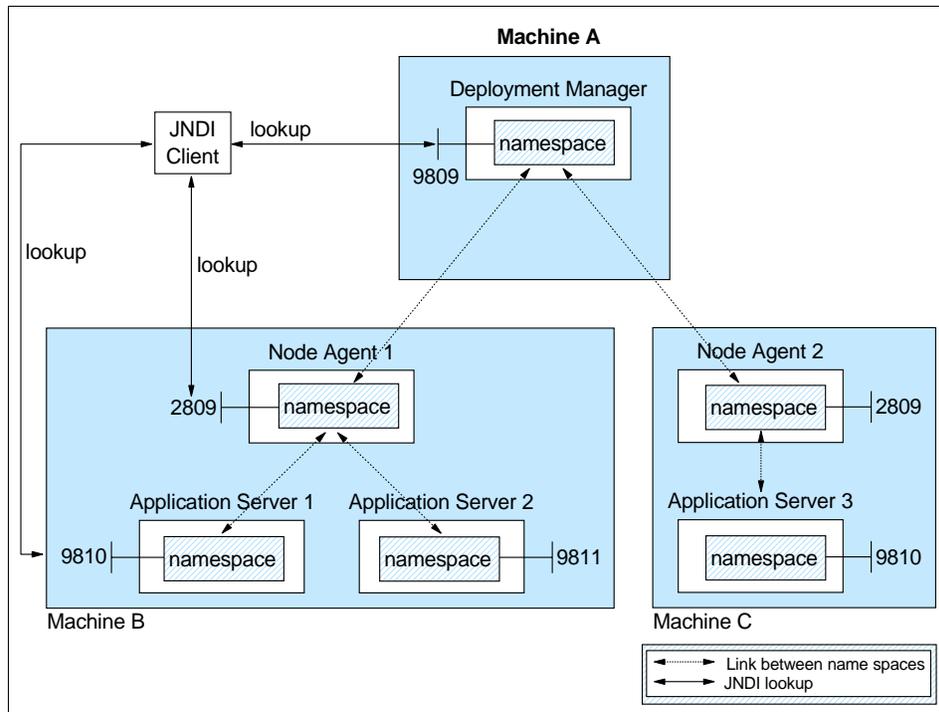Figure 3-7 on page 74 summarizes the naming architecture and its components.

Figure 3-7   Naming topology

3.8.8 PMI service

WebSphere Application Server collects data on runtime and applications through the Performance Monitoring Infrastructure (PMI). This infrastructure is compatible with and extends the JSR-077 specification.

PMI uses a client-server architecture. The server collects performance data from various WebSphere Application Server components and stores it in memory. This data consists of counters such as servlet response time and data connection pool usage. The data can then be retrieved using a Web client, Java client or JMX client. WebSphere Application Server contains Tivoli Performance Viewer, now integrated in the WebSphere administrative console, which displays and monitors performance data.

WebSphere Application Server also collects data by timing requests as they travel through the product components. PMI request metrics log time spent in major components, such as Web containers, EJB containers, and databases. These data points are recorded in logs and can be written to Application Response Measurement (ARM) agents used by Tivoli monitoring tools.

3.8.9 Security service

Each application server JVM hosts a security service that uses the security settings held in the configuration repository to provide authentication and authorization functionality.

3.9 Data Replication Service (DRS)

The Data Replication Service (DRS) is responsible for replicating in-memory data among WebSphere processes.

DRS can be used for:

- ► HTTP session persistence and failover.
- ► Stateful session EJB persistence and failover (new in V6.0)
- ► Dynamic cache replication

Replication is done through the use of replication domains consisting of server or cluster members that have a need to share internal data. Multiple domains can be used, each for a specific task among a set of servers or clusters. While HTTP session replication and EJB state replication can (and should) share a domain, a separate domain is needed for dynamic cache replication.

A domain can be defined so that each domain member has a single replicator that sends data to another member of the domain, or each member has multiple replicators that send data to multiple members of the domain.

Two topology options are offered:

- ► Peer-to-peer

    Each application server stores sessions in its own memory. It also stores sessions to and retrieves sessions from other application servers. In other words, each application server acts as a client by retrieving sessions from other application servers, and each application server acts as a server by providing sessions to other application servers. This mode, working in conjuction with the workload manager, provides hot failover capabilities.

- ► Client/server

    Application servers act as either a replication client or a server. Those that act as replication servers store sessions in their own memory and provide session information to clients. They are dedicated replication servers that just store sessions but do not respond to the users' requests. Client application servers send session information to the replication servers and retrieve sessions from the servers. They respond to user requests and store only the sessions of the users with whom they interact.

3.10 Virtual hosts

A *virtual host* is a configuration enabling a single host machine to resemble multiple host machines. It allows a single physical machine to support several independently configured and administered applications. It is not associated with a particular node. It is a configuration, rather than a live object, which is why it can be created, but not started or stopped.

Each virtual host has a logical name and a list of one or more DNS aliases by which it is known. A *DNS alias* is the TCP/IP host name and port number used to request the servlet, for example yourHostName:80. When a servlet request is made, the server name and port number entered into the browser are compared to a list of all known aliases in an effort to locate the correct virtual host and serve the servlet. If no match is found, an HTTP 404 error is returned to the browser.

IBM WebSphere Application Server provides two default virtual hosts:

► default_host

   This virtual hose is used for accessing most applications. The default settings for default_host map to all requests for any alias on ports 80, 9443, and 9080.

   For example:

   ```
   http://localhost:80/servlet/snoop
   http://localhost:9080/servlet/snoop
   ```

► admin_host

   The admin>_host is specifically configured for accessing the WebSphere Application Server administrative console. Other applications are not accessible through this virtual host. The default settings for admin_host map to requests on ports 9060 and 9043.

   For example:

   ```
   http://localhost:9060/admin
   ```

3.11 Session management

In many Web applications, users dynamically collect data as they move through the site based on a series of selections on pages they visit. Where the user goes next, and what the application displays as the user's next page (or next choice) can depend on what the user has chosen previously from the site. In order to maintain this data, the application stores it in a session.

WebSphere supports three approaches to track sessions:

- SSL session identifiers

  SSL session information is used to track the HTTP session ID.

- Cookies

  The application server session support generates a unique session ID for each user, and returns this ID to the user's browser using a cookie. The default name for the session management cookie is JSESSIONID. Using cookies is the most common method of session management.

- URL rewriting

Session data can be kept in local memory cache, stored externally on a database, or kept in memory and replicated among application servers.

Table 3-7 shows the session support for each WebSphere Application Server configuration.

Table 3-7   WebSphere Application Server session management support

Session type	Express and Base	ND
Cookies	Yes	Yes
URL rewriting	Yes	Yes
SSL session identifiers	Yes	Yes
In memory cache	Yes	Yes
Session persistence using a database	Yes	Yes
Memory-to-memory session persistence	No	Yes

The Servlet 2.4 specification defines session scope at the Web application level, meaning that session information can only be accessed by a single Web application. However, there can be times when there is a logical reason for multiple Web applications to share information, for example a user name. WebSphere Application Server provides an IBM extension to the specification allowing session information to be shared among Web applications within an enterprise application. This option is offered as an extension to the application deployment descriptor. No code change is necessary to enable this option. This option is specified during application assembling.

3.11.1 HTTP Session persistence

Many Web applications use the simplest form of session management, the in-memory local session cache. The local session cache keeps session

information in memory and local to the machine and WebSphere Application Server where the session information was first created. Local session management does not share user session information with other clustered machines. Users only obtain their session information if they return to the machine and WebSphere Application Server holds their session about subsequent accesses to the Web site.

Most importantly, local session management lacks a persistent store for the sessions it manages. A server failure takes down not only the WebSphere instances running on the server, but also destroys any sessions managed by those instances.

By default, WebSphere Application Server places session objects in memory. However, the administrator has the option of enabling persistent session management. This instructs WebSphere to place session objects in a persistent store. Using a persistent store allows the user session data to be recovered by an application server on restart or another cluster member after a cluster member in a cluster fails or is shut down. Two options for HTTP session persistence are available:

► Database

Session information can be stored in a central session database for session persistence.

In a single-server environment, the session can be persisted when the user's session data must be maintained across a server restart or when the user's session data is too valuable to lose through an unexpected server failure.

In a multi-server environment, the multiple application servers hosting a particular application need to share this database information in order to maintain session states for the stateful components.

► Memory-to-memory using data replication services

In a Network Deployment distributed server environment, WebSphere internal replication enables sessions to be shared among application servers without using a database. Using this method, sessions are stored in the memory of an application server, providing the same functionality as a database for session persistence.

3.11.2 Stateful session EJB persistence

With V6 you now have failover capability of stateful session EJBs. This function uses data replication services and interacts with the workload manager component during a failover situation.

3.12 Web services

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. WebSphere Application Server can act as both a Web service provider and as a requester. As a provider, it hosts Web services that are published for use by clients. As a requester, it hosts applications that invoke Web services from other locations.

WebSphere Application Server supports SOAP-based Web service hosting and invocation.

Table 3-8   WebSphere Application Server server support

Support type	Express & Base	ND
Web services support	Yes	Yes
Private UDDI v3 Registry	Yes	Yes
Web Services Gateway	No	Yes
Enterprise Web services	Yes	Yes

Web services support includes the following:

► Web Services Definition Language (WSDL), an XML-based description language that provides a way to catalog and describe services. It describes the interface of Web services (parameters and result), the binding (SOAP, EJB), and the implementation location.

► Universal Description, Discovery and Integration (UDDI), a global, platform-independent, open framework to enable businesses to discover each other, define their interaction, and share information in a global registry.

UDDI support in WebSphere Application Server V6 includes UDDI V3 APIs, some UDDI V1 and V2 APIs, UDDI V3 client for Java, and UDDI4J for compatiblity with UDDI V2 registries. It also provides a UDDI V3 Registry, that is integrated in WebSphere Application Server.

► SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment.

► eXtensible Markup Language (XML) provides a common language for exchanging information.

► JAX-RPC (JSR 101) provides the core programming model and bindings for developing and deploying Web services on the Java platform. It is a Java API for XML-based RPC and supports JavaBeans and enterprise beans as Web service providers.

- ► Enterprise Web services (JSR 109): this adds EJBs and XML deployment descriptors to JSR 101.

- ► WS-Security is a specification that covers a standard set of SOAP extensions to use when building secure Web services to provide integrity and confidentiality. It is designed to be open to other security models including PKI, Kerberos, and SSL. WS-Security provides support for multiple security tokens, multiple signature formats, multiple trust domains, and multiple encryption technologies. It includes security token propagation, message integrity, and message confidentiality. The specification is proposed by IBM, Microsoft, and VeriSign for review and evaluation. In the future, it will replace existing Web services security specifications from IBM and Microsoft including SOAP Security Extensions (SOAP-SEC), Microsoft's WS-Security and WS-License, as well as IBM's security token and encryption documents.

- ► JAXR is an API that standardizes access to Web services registries from within Java. The current JAXR version, 1.0, defines access to ebXML and UDDI V2 registries. WebSphere Application Server provides JAXR level 0 support, meaning it supports UDDI registries.

  JAXR does not map precisely to UDDI. For a precise API mapping to UDDI V2, IBM provides UDDI4J and IBM Java Client for UDDI v3.

- ► SAAJ stands for SOAP with Attachments API for Java and defines a standard for sending XML documents over the Internet from the Java platform.

> **IBM value add:** In addition to the requirements of the specifications, IBM includes the following value add features to its Web services support:
>
> ► Custom bindings
>
>   JAX-RPC does not support all XML schema types. This feature allows developers to create their own custom bindings needed to map Java to XML and XML to Java conversions.
>
> ► Support for generic SOAP elements
>
>   In cases, where you do not want mapping to XML schema or custom binding, but want to keep it generic, this will allow you to eliminate binding and instead use the generic SOAPElement type.
>
> ► Multi-protocol support for a stateless session EJB as the Web service provider providing enhanced performance without any changes to JAX-RPC client.
>
> ► Client caching
>
>   In WebSphere Application Server V5, there was support for server-side Web service caching for Web services providers running within the application server. In addition to this server side caching, V6 introduces caching for Web services clients running within a V6 application server, including the Web Services Gateway.

3.12.1 Enterprise services (JCA Web services)

Enterprise services offer access over the Internet to applications in a platform-neutral and language-neutral fashion. They offer access to Enterprise Information Systems (EIS) and message queues and can be used in a client/server configuration without the Internet. Enterprise services can access applications and data on a variety of platforms and in a variety of formats.

An enterprise service wraps a software component in a common services interface. The software component is typically a Java class, EJB, or JCA resource adapter for an EIS. In services terminology, this software component is known as the implementation. Enterprise services primarily use WSDL and WSIF to expose an implementation as a service.

Using the Integrated Edition of WebSphere Studio, you can turn CICS and IMS transactions using J2EE Connector Architecture (JCA) into Web services.

3.12.2 Web service client

Applications that invoke Web services are known as Web service clients or Web service requestors. An application that acts as a Web service client is deployed to WebSphere Application Server like any other enterprise application. No additional configuration or software is needed for the Web services client to function. Web services clients can also be stand-alone applications.

A Web service client will bind to a Web service server to invoke the Web service. The binding is done using a service proxy (or *stub*), which is generated based on the WSDL document for the Web service. This service proxy contains all the needed information to invoke the Web service and is used locally by the clients to access the business service. The binding can also be done dynamically using Web Services Invocation Framework (WSIF).

3.12.3 Web service provider

An application that acts as a Web service is deployed to WebSphere Application Server in the same way as any other enterprise application. The Web services are contained in Web modules or EJB modules.

Publishing the Web service to a UDDI registry makes it available to anyone searching for it. Web services can be published to a UDDI registry using the Web Services Explorer provided with Rational Application Developer.

When using Rational Application Developer to package the application for deployment, no additional configuration or software is needed for the Web services client to function. The SOAP servlets are automatically added and a SOAP admin tool is included in a Web module.

If not, you can use the endptEnabler command-line tool found in the WebSphere bin directory to enable the SOAP services within the EAR file and add the SOAP admin tool.

3.12.4 Enterprise Web Services

The Enterprise Web Services, based on the JSR 109 specification request, provides the use of JAX-RPC in a J2EE environment defining the runtime architecture, as well as the implementation and deployment of Web services in a generic J2EE server. The specification defines the programming model and architecture for implementing Web services in Java based on JSRs 67, 93, 101, and future JSRs related to Web services standards. You can find the list of JSRs at this Web site:

```
http://www.jcp.org/en/jsr/all
```

3.12.5 IBM WebSphere UDDI Registry

WebSphere Application Server V6.0 provides a private UDDI registry that implements V3.0 of the UDDI specification. This enables the enterprise to run its own Web services broker within the company or provide brokering services to the outside world. The UDDI registry installation and management is now integrated in with WebSphere Application Server.

There are three interfaces to the registry for inquiry and publish:

▶ Through the UDDI SOAP API.

▶ Through the UDDI EJB client interface.

▶ Through the UDDI user console. This Web-based GUI interface can be used to publish and to inquire about UDDI entities but only provides a subset of the UDDI API functions.

Security for the UDDI registry is handled using WebSphere security. To support the use of secure access with the IBM WebSphere UDDI Registry, you need to configure WebSphere to use HTTPS and SSL.

A relational database is used to store registry data.

3.12.6 Web Services Gateway

The Web Services Gateway bridges the gap between Internet and intranet environments during Web service invocations. The gateway builds upon the Web services Definition Language (WSDL) and the WSIF for deployment and invocation.

With V6, the Web Services Gateway is fully integrated into the integration service technologies which provides the runtime. The administration is done directly from the WebSphere administrative console.

The primary function of the Web Services Gateway is to map an existing WSDL-defined Web service (target service) to a new service (gateway service) that is offered by the gateway to others. The gateway thus acts as a proxy. Each target service, whether internal or external is available at a service integration bus destination.

The role formerly played by filters in the V5 Web Services Gateway is now provided by JAX-RPC handlers. The use of JAX-RPC handlers provides a standard approach for intercepting and filtering service messages. JAX-RPC handlers interact with messages as they pass into and out from the service integration bus. Handlers monitor messages at ports, and take appropriate action depending upon the sender and content of each message.

Exposing internal Web services to the outside world

Web services hosted internally and made available through the service integration bus are called *inbound services*. Inbound services are associated with a service destination. Service requests and responses are passed to the service through an endpoint listener and associated inbound port.

From the gateway's point of view, the inbound service is the target service. To expose the target service for outside consumption, the gateway takes the WSDL file for the inbound service and generates a new WSDL file that can be shared with outside requestors. The interface described in the WSDL is exactly the same, but the service endpoint is changed to the gateway, which is now the official endpoint for the service client.
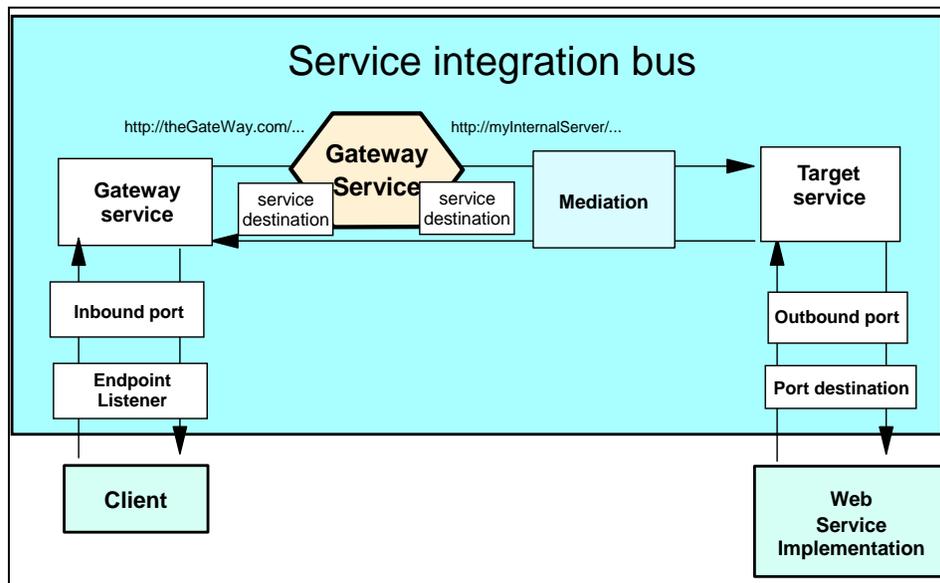
Figure 3-8   Exposing Web services through the Gateway

Externally hosted Web services

A Web service that is hosted externally and made available through the service integration bus is called an *outbound service*. To configure an externally-hosted service for a bus, you first associate it with a service destination, then you configure one or more port destinations (one for each type of binding, for example SOAP over HTTP or SOAP over JMS) through which service requests and responses are passed to the external service.

From the gateway's point of view, the outbound service is the target service. Mapping a gateway service to the target service will allow internal service requestors invoke the service as though it were running on the gateway. Again, a

new WSDL is generated by the gateway showing the same interface but naming the gateway as service provider rather than the real internal server. All requests to the gateway service are rerouted to the actual implementation specified in the original WSDL.

Of course, every client could access external Web services by traditional means, but if you add the gateway as an additional layer in between, clients do not have to change anything if the service implementor changes. This scenario is very similar to the illustration in Figure 3-8 on page 84, with the difference that the Web service implementation is located at a site on the Internet.

UDDI publication and lookup

The gateway facilitates working with UDDI registries. As you map a service for external consumption using the gateway, you can publish the exported WSDL in the UDDI registry. When the services in the gateway are modified, the UDDI registry is updated with the latest changes.

3.13 Service integration bus

The service integration bus provides the communication infrastructure for messaging and service-oriented applications, thus unifying this support into a common component. The service integration bus provides a JMS 1.1 compliant JMS provider for reliable message transport and has the capability of intermediary logic to intelligently adapt message flow in the network. It also supports the attachment of Web services requestors and providers.

The service integration bus capabilities have been fully integrated within WebSphere Application Server, enabling it to take advantage of WebSphere security, administration, performance monitoring, trace capabilities, and problem determination tools.

The service integration bus is often referred to as just a *bus*. When used to host JMS applications, it is also often referred to as a *messaging bus*.

Figure 3-9 on page 86 illustrates the service integration bus and how it fits into the larger picture of an Enterprise Service Bus (ESB).
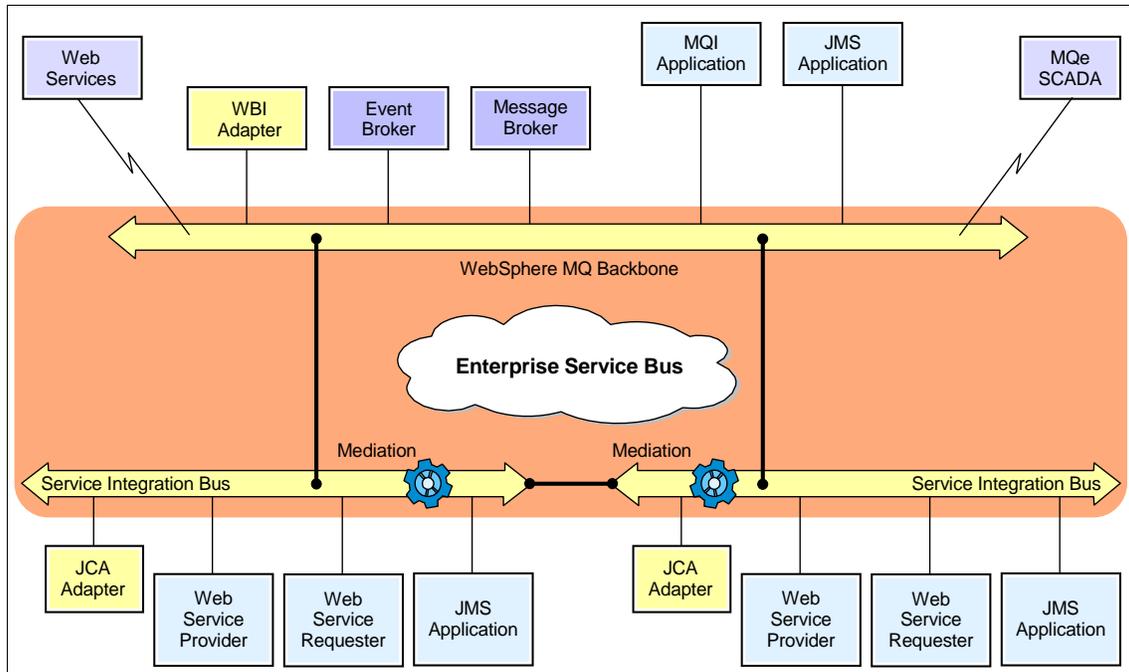
Figure 3-9   The Enterprise Service Bus

A service integration bus consists of the following:

► *Bus members* are application servers or clusters that have been added to the bus.

► A *Messaging engine* is the application server or cluster component that manages bus resources. When a bus member is defined, a messaging engine is automatically created on the application server or cluster. The messaging engine provides a connection point for clients to produce or consume messages from.

An application server will have one messaging engine per bus it is a member of. A cluster will have at least one messaging engine per bus and can have more. In this case the cluster owns the messaging engine(s) and determines on which application server(s) they will run.

► *Destinations* are the places within the bus that applications attach to exchange messages. Destinations can represent Web service endpoints, messaging point-to-point queues, or messaging pub/sub topics. Destinations are created on a bus and hosted on a messaging engine.

► The *message store* is represented by each messaging engine using a set of tables in a data store (JDBC database) to hold information such as messages, subscription information, and transaction states. Messaging

engines can share a database, each using its own set of tables. The message store can be backed by any JDBC database supported by WebSphere Application Server.

3.13.1 Application support

The service integration bus supports the following application attachments:

► Web services

– Requestors using the JAX-RPC API

– Providers running in WebSphere Application Server as stateless session beans and servlets (JSR-109)

– Requestors or providers attaching with SOAP/HTTP or SOAP/JMS

► Messaging applications

– Inbound messaging using JFAP-TCP/IP (or wrapped in SSL for secure messaging)

JFAP is a proprietary format and protocol used for service integration bus messaging providers.

– MQ application in an MQ network using MQ channel protocol

– JMS applications in WebSphere Application Server V5 using MQ client protocol

– JMS applications in WebSphere Application Server V6

3.13.2 Service integration bus and messaging
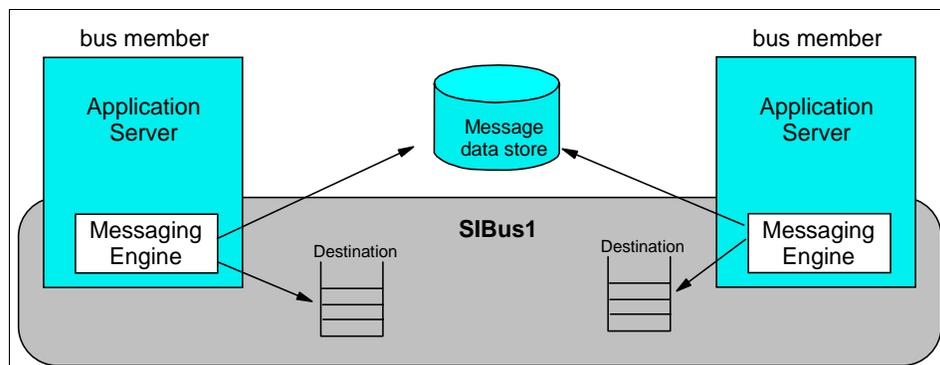
Figure 3-10   Service Integration Bus

With Express or Base configuration, you typically have one stand-alone server with one messaging engine on one service integration bus. With Network Deployment you have more flexibility in your options.

The following are valid topologies:

► One bus and one messaging engine (application server or cluster)
► One bus with multiple messaging engines
► Multiple buses within a cell, which might or might not be connected to each other
► Buses connected between cells
► One application server that is a member of multiple buses and having one messaging engine per bus.
► A connection between a bus and a WebSphere MQ queue manager

When using this type of topology, consider the following:

– WebSphere MQ can coexist on the same machine as the WebSphere default messaging provider. In V5, the embedded JMS server and WebSphere MQ could not coexist on the same machine.
– A messaging engine cannot participate in a WebSphere MQ cluster.
– The messaging engine can be configured to look like another queue manager to WebSphere MQ.
– WebSphere applications can send messages to WebSphere MQ queues direct, or though the service integration bus.
– You can have multiple connections to WebSphere MQ, but each must be to a different queue manager.
– WebSphere Application Server V5 JMS client can connect to V6 destinations. Also, a V6 JMS application can connect to an embedded JMS provider in a V5 server if so configured. However, you cannot connect a V5 embedded JMS server to a V6 bus.

Mediation

Mediation is the ability to manipulate a message as it traverses the messaging bus (destination). For example:

► Transform the message.
► Reroute the message.
► Copy and route to additional destinations.
► Interact with non-messaging resource managers (databases, for example).

Mediation is controlled using a mediation handler list. The list is a collection of mediation handlers (Java programs that perform the function of a mediation) that are invoked in sequence.

Clustering

In a distributed server environment, you can use clustering for high availability and scalability. As you have seen, a cluster can be added as a bus member.

► High availability

One messaging engine is active in the cluster. In the event the messaging engine or server fails, the messaging engine on a standby server is activated.

► Scalability

A single messaging destination can be partitioned across multiple active messaging engines in the cluster (messaging order is not preserved).

QOS

Quality of service (QOS) can be defined on a destination basis to determine how messages are (or aren't) persisted. QOS can also be specified within the application.

Message Driven Beans

With EJB 2.1, Message Driven Bean (MDB) in the application server that listen to queues and topics are linked to the appropriate destinations on the service integration bus using JCA connectors (ActivationSpec objects). Support is also included for EJB 2.0 MDBs to be deployed against a listener port.

3.13.3 Web services and the integration bus

Through the service integration bus Web services (SIBWS) enablement, you can achieve the following goals:

► You can take an internal service that is available at a service destination, and make it available as a Web service.

► You can take an external Web service, and make it available at a service destination.

► You can use the Web services gateway to map an existing service, either an internal service or an external Web service, to a new Web service that appears to be provided by the gateway.

3.14 Security

Table 3-9 shows the security features supported by the WebSphere Application Server configurations.

Table 3-9   WebSphere Application Server security support

Security feature	Express & Base	ND
Java 2 security	Yes	Yes
J2EE security (role mapping)	Yes	Yes
JAAS	Yes	Yes
CSIv2	Yes	Yes
JACC	Yes	Yes
Security authentication	LTPA, SWAM	LTPA
User registry	Local OS, LDAP, Custom Registry	Local OS, LDAP, Custom Registry

Figure 3-11 presents a general view of the logical layered security architecture model of WebSphere Application Server. The flexibility of that architecture model lies in pluggable modules that can be configured according to the requirements and existing IT resources.
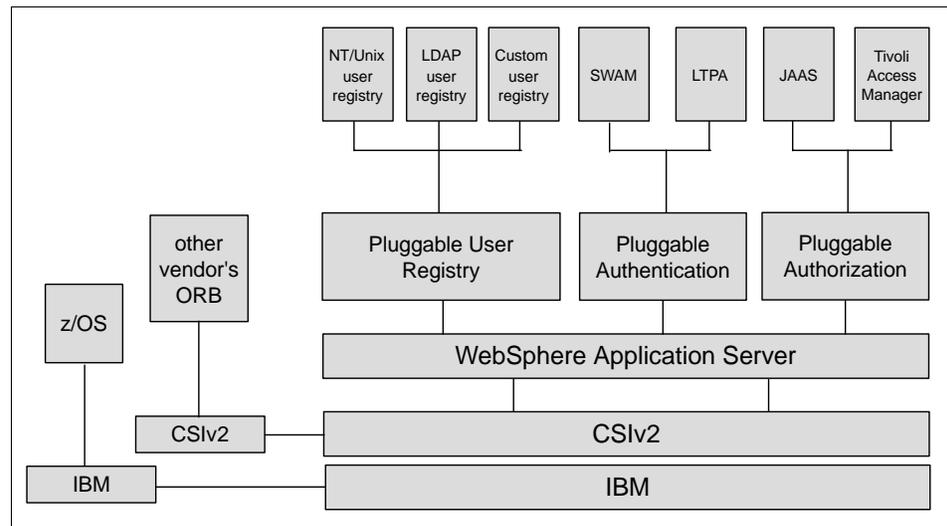
Figure 3-11   WebSphere Application Server security architecture

IBM WebSphere Application Server security sits on top of the operating system security and the security features provided by other components, including the Java language.

► Operating system security protects sensitive WebSphere configuration files and authenticates users when the operating system user registry is used for authentication.

► Standard Java security is provided through the Java Virtual Machine (JVM) used by WebSphere and the Java security classes.

► The Java 2 Security API provides a way to enforce access control, based on the location of the code and who signed it. Java 2 security guards access to system resources such as file I/O, sockets, and properties. WebSphere global security settings allow you to enable or disable Java 2 security and provide a default set of policies. Java 2 security can be activated or inactivated independently from WebSphere global security.

The current principal of the thread of execution is not considered in the Java 2 security authorization. There are instances where it is useful for the authorization to be based on the principal, rather the code base and the signer.

► The Java Authentication and Authorization Services (JAAS) is a standard Java API that allows the Java 2 security authorization to be extended to the code base on the principal as well as the code base and signers. The JAAS programming model allows the developer to design application authentication in a pluggable fashion, which makes the application independent from the underlying authentication technology. JAAS does not require Java 2 security to be enabled.

► The Common Secure Interoperability protocol adds additional security features that enable interoperable authentication, delegation and privileges in a CORBA environment. It supports interoperability with the EJB 2.1 specification and can be used with SSL.

► J2EE security uses the security collaborator to enforce J2EE-based security policies and support J2EE security APIs. APIs are accessed from WebSphere applications in order to access security mechanisms and implement security policies. J2EE security guards access to Web resources such as servlets/JSPs and EJB methods based on roles defined by the application developer. Users and groups are assigned to these roles during application deployment.

► Java Authorization Contract for Containers (JACC) support allows the use of third party authorization providers to be used for access decisions. The default JACC provider for WebSphere Application Server is the Tivoli Access Manager (bundled with Network Deployment). The Tivoli Access Manager client functions are integrated in WebSphere Application Server.

► IBM Java Secure Socket Extension (JSEE) is the Secure Sockets Layer (SSL) implementation used by WebSphere Application Server. It is a set of Java packages that enable secure Internet communications. It implements a Java version of SSLand Transport Layer Security (TLS) protocols and includes functionality for data encryption, server authentication, message integrity, and client authentication.

WebSphere Application Server security relies on and enhances all the above mentioned layers. It implements security policy in a unified manner for both Web and EJB resources. WebSphere global security options are defined at the cell level, although individual servers can override a subset of the security configuration. When using mixed z/OS and distributed nodes, the security domain features will be merged.

3.14.1 User registry

The pluggable user registry allows you to configure different databases to store user IDs and passwords that are used for authentication and authorization. There are three options:

► Local operating system user registry

When configured, WebSphere uses the operating system's users and groups for authentication.

► LDAP user registry

In many solutions, LDAP user registry is recommended as the best solution for large scale Web implementations. Most of the LDAP servers available on the market are well equipped with security mechanisms that can be used to securely communicate with WebSphere Application Server. The flexibility of search parameters that an administrator can set up to adapt WebSphere to different LDAP schemas is considerable.

► Custom user registry

This leaves an open door for any custom implementation of a user registry database. WebSphere API provides the UserRegistry Java interface that you should use to write the custom registry. This interface can be used to access virtually any relational database, flat files and so on.

Only one single registry can be active at a time.

3.14.2 Authentication

Authentication is the process of establishing whether a client is valid in a particular context. A client can be either an end user, a machine, or an application. The pluggable authentication module allows you to choose whether

WebSphere will authenticate the user or will accept the credentials from external authentication mechanisms.

An authentication mechanism in WebSphere typically collaborates closely with a user registry when performing authentication. The authentication mechanism is responsible for creating a credential which is a WebSphere internal representation of a successfully authenticated client user. Not all credentials are created equal. The abilities of the credential are determined by the configured authentication mechanism.

Although WebSphere provides several authentication mechanisms, only a single active authentication mechanism can be configured at once. The active authentication mechanism is selected when configuring WebSphere global security.

WebSphere provides two authentication mechanisms: Simple WebSphere Authentication Mechanism (SWAM) and Lightweight Third Party Authentication (LTPA). These two authentication mechanisms differ primarily in the distributed security features each supports.

► Simple WebSphere Authentication Mechanism(SWAM)

The SWAM authentication mechanism is intended for simple, nondistributed, single application server type run-time environments. The single application server restriction is due to the fact that SWAM does not support forwardable credentials. This means that if a servlet or EJB in application server process 1 invokes a remote method on an EJB living in another application server process 2, the identity of the caller identity in process 1 is not transmitted to server process 2. What is transmitted is an unauthenticated credential, which, depending on the security permissions configured on the EJB methods, may cause authorization failures.

Since SWAM is intended for a single application server process, single signon (SSO) is not supported.

The SWAM authentication mechanism is suitable for simple environments, software development environments, or other environments that do not require a distributed security solution.

SWAM relies on the session ID. It is not as secure as LTPA, therefore using SSL with SWAM is strongly recommended.

► Light Weight Third Party Authentication (LTPA)

Lightweight Third Party Authentication (LTPA) is intended for distributed, multiple-application servers and machine environments. It supports forwardable credentials and SSO. LTPA is able to support security in a distributed environment through the use of cryptography. This allows LTPA to

encrypt, digitally sign and securely transmit authentication related data and later decrypt and verify the signature.

LTPA requires that the configured user registry be a central shared repository such as LDAP or a Windows domain type registry.

3.14.3 Authorization

WebSphere Application Server as the following standard authorization features:

► Java 2 security architecture uses a security policy to specify who is allowed to execute code in the application. Code characteristics such as a code signature, signer ID, or source server, determine whether or not the code will be granted access to be executed.

► JAAS, which extends this approach with role-based access control. Permission to execute a code is granted not only based on the code characteristics but also on the user running it. JAAS programming models allow the developer to design application authentication in a pluggable fashion, which makes the application independent from the underlying authentication technology.

For each authenticated user, a Subject class is created and a set of Principals is included in the subject in order to identify that user. Security policies are granted based on possessed principals.

WebSphere Application Server uses an internal authorization mechanism by default. As an alternative, you can define external JACC providers to handle authorization decisions. During application installation, security policy information is stored in the JACC provider server using standard, JACC-defined interfaces. Subsequent authorization decisions are made using this policy information. As an exception to this, all administrative security authorization decisions are made by the WebSphere Application Server default authorization engine.

3.14.4 Security components

Figure 3-12 on page 95 shows an overview of the security components that come into play in WebSphere Application Security.
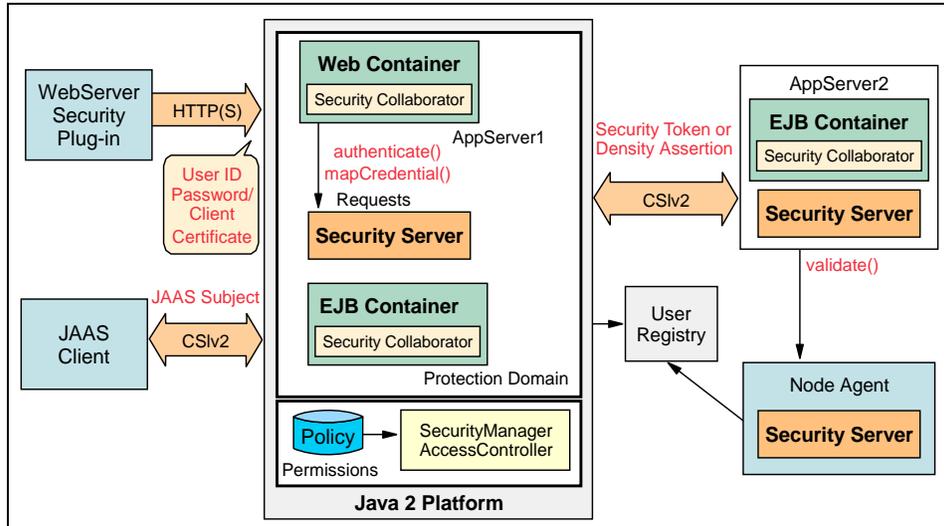
Figure 3-12   WebSphere Application Security components

Security server

The security server is a component of WebSphere Application Server that runs in each application server process. If multiple application server instances are executed on a single node, then multiple security servers exist on that node.

The security server component is responsible for managing authentication and for collaborating with the authorization engine and the user registry.

Security collaborators

Security collaborators are application server processes responsible for enforcing security constraints specified in deployment descriptors. They communicate with the security server every time that authentication and authorization actions are required. The following security collaborators are identified.

► Web security collaborator

The Web security collaborator resides in the Web container and provides the following services to the application:

– Checks authentication.

– Performs authorization according to the constraint specified in the deployment descriptor.

– Logs security tracing information.

► EJB security collaborator

The EJB security collaborator resides in the EJB container. It uses CSIv2 and SAS to authenticate Java client requests to enterprise beans. It works with the security server to perform the following functions:

– Check authorizations according to the specified security constraint.

– Support communication with local user registry.

– Log security tracing information.

– Communicate external ORB using CSIv2 when a request for a remote bean is issued.

3.14.5 Security flows

The following sections outline the general security flow.

Web browser communication

The following steps describe the interaction of the components from a security point of view when a Web browser sends a request to a WebSphere application.

1. The Web user requests a Web resource protected by WebSphere Application Server.

2. The Web server receives the request, recognizes that the requested resource is on the application server, and, using the Web server plug-in, redirects the request.

3. The Web server plug-in passes the user credentials to the Web security collaborator, which performs user authentication.

4. After successful authentication, the Web request reaches the Web container. The Web security collaborator passes the user's credentials and the security information contained in the deployment descriptor to the security server for authorization.

5. Upon subsequent requests, authorization checks are performed either by the Web collaborator or the EJB collaborator, depending on what the user is requesting. User credentials are extracted from the established security context.

Administrative tasks

Administrative tasks are issued using either the Web-based administrative console or the wsadmin scripting tool. The following steps illustrate how the administration tasks are executed.

1. The administration client generates a request that reaches the server side ORB and JMX MBeans. The JMX MBeans represent managed resources.

2. The JMX MBeans contact the security server for authentication purposes. JMX beans have dedicated roles assigned and do not use user registry for authentication and authorization.

Java client communication

These steps describe how a Java client interacts with a WebSphere application.

1. A Java client generates a request that reaches the server side ORB.

2. The CSIv2 or IBM SAS interceptor performs authentication on the server side on behalf of the ORB, and sets the security context.

3. The server side ORB passes the request to the EJB container.

4. After submitting a request to the access protected EJB method, the EJB container passes the request to the EJB collaborator.

5. The EJB collaborator reads the deployment descriptor from the .ear file and user credential from the security context.

6. Credentials and security information is passed to the security server which validates user access rights and passes this information back to the collaborator.

7. After receiving a response from the security server, the EJB collaborator authorizes or denies access to the user to the requested resource.

3.15 Resource providers

Resource providers define resources needed by running J2EE applications. Table 3-10 shows the resource provider support of the WebSphere Application Server configuration.

Table 3-10   WebSphere Application Server resource provider support

Resource provider	Express & Base	ND
JDBC provider	Yes	Yes
Mail providers (JavaMail)	Yes	Yes
JMS providers	Yes	Yes
Resource environment providers	Yes	Yes
URL providers	Yes	Yes
Resource adapters	Yes	Yes

3.15.1 JDBC resources

A data source represents a real-world data source, such as a relational database. When a data source object has been registered with a JNDI naming service, an application can retrieve it from the naming service and use it to make a connection to the data source it represents.

Information about the data source and how to locate it, such as its name, the server on which it resides, its port number, and so on, is stored in the form of properties on the DataSource object. This makes an application more portable because it does not need to hard code a driver name, which often includes the name of a particular vendor. It also makes maintaining the code easier because if, the data source is moved to a different server for example, all that needs to be done is to update the relevant property in the data source. None of the code using that data source needs to be touched.

Once a data source has been registered with an application server's JNDI name space, application programmers can use it to make a connection to the data source it represents.

The connection will usually be a pooled connection. That is, once the application closes the connection, the connection is returned to a connection pool, rather than being destroyed.

Data source classes and JDBC drivers are implemented by the data source vendor. By configuring a JDBC provider, we are providing information about the set of classes used to implement the data source and the database driver, that is, it provides the environment settings for the DataSource object.

Data sources

In WebSphere Application Server, connection pooling is provided by two parts, a JCA Connection Manager and a relational resource adapter.
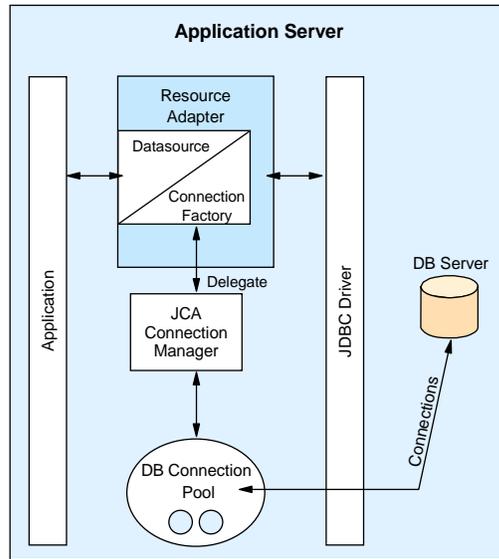
Figure 3-13   Resource adapter in J2EE connector architecture

The JCA Connection Manager provides the connection pooling, local transaction, and security supports. The relational resource adapter provides both JDBC wrappers and JCA CCI implementation that allows BMP, JDBC applications and CMP beans to access the database.

Version 4 data sources: WebSphere Version 4.0 provided its own JDBC connection manager to handle connection pooling and JDBC access. This support is included with WebSphere Application Server V5 and V6 to provide support for J2EE 1.2 applications. If an application chooses to use a Version 4 data source, the application will have the same connection behavior as in WebSphere Version 4.

3.15.2 Mail providers

The JavaMail APIs provide a platform and protocol-independent framework for building Java-based mail client applications. The JavaMail APIs require service providers, known in WebSphere as protocol providers, to interact with mail servers that run the pertaining protocols.

A Mail provider encapsulates a collection of protocol providers. WebSphere Application Server has a Built-in Mail Provider that encompasses three protocol providers: SMTP, IMAP and POP3. These protocol providers are installed as the default and should be sufficient for most applications.

- Simple Mail Transfer Protocol (SMTP) is a popular transport protocol for sending mail. JavaMail applications can connect to an SMTP server and send mail through it by using this SMTP protocol provider.
- Post Office Protocol (POP) is the standard protocol for receiving mail.
- Internet Message Access Protocol (IMAP) is an alternative protocol to POP3 for receiving mail.

To use other protocols, you must install the appropriate service provider for those protocols.

In addition to service providers, JavaMail requires the JavaBeans Activation Framework (JAF) as the underlying framework to deal with complex data types that are not plain text, like Multipurpose Internet Mail Extensions (MIME), Uniform Resource Locator (URL) pages, and file attachments.

The JavaMail APIs, the JAF, the service providers and the protocols are shipped as part of WebSphere Application Server using the following Sun licensed packages:

- mail.jar: contains the JavaMail APIs, and the SMTP, IMAP, and POP3 service providers.
- activation.jar: contains the JavaBeans Activation Framework.

3.15.3 JCA resource adapters

The J2EE Connector Architecture (JCA) defines a standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EIS), for example ERP, mainframe transaction processing, database systems, and existing applications not written in the Java programming language.

The JCA resource adapter is a system-level software driver supplied by EIS vendors or other third-party vendors. It provides the connectivity between J2EE components (an application server or an application client) and an EIS.

To use a resource adapter, you need to install the resource adapter code and create connection factories that use the adapter.

One resource adapter, the WebSphere Relational Resource Adapter, is predefined for handling data access to relational databases. This resource adapter provides data access through JDBC calls to access databases dynamically. It provides connection pooling, local transaction, and security support. The WebSphere persistence manager uses this adapter to access data for container-managed persistence (CMP) beans.

3.15.4 URL providers

URL providers implement the functionality for a particular URL protocol, such as HTTP, by extending the java.net.URLStreamHandler and java.net.URLConnection classes. It enables communication between the application and a URL resource that is served by that particular protocol.

A URL provider named Default URL Provider is included in the initial WebSphere configuration. This provider utilizes the URL support provided by the IBM JDK. Any URL resource with protocols based on the Java 2 Standard Edition 1.3.1, such as HTTP, FTP or File, can use the default URL provider.

Customers can also plug in their own URL providers that implement other protocols not supported by the JDK.

3.15.5 JMS providers

The JMS functionality provided by WebSphere includes support for three types of JMS provider:

► Default messaging provider (service integration bus)
► WebSphere MQ provider
► Generic JMS providers
► V5 default messaging provider (for migration)

There can be more than one JMS provider per node. That is, a node can be configured to concurrently make use of any combination (or all) of the default messaging providers, WebSphere MQ JMS providers and a generic JMS providers. In addition, WebSphere MQ and the default messaging provider can coexist on the same machine.

The support provided by WebSphere administration tools for configuration of JMS providers differs depending upon the provider. Table 3-11 provides a summary of the support.

Table 3-11   WebSphere administration support for JMS provider configuration

Configurable objects	Default messaging provider	WebSphere MQ JMS provider	Generic JMS provider	V5 default messaging - WebSphere JMS provider
Messaging system objects (queues/topics)	Y	N	N	Y
JMS administered objects (JMS connection factory and JMS destination)	Y	Y	N	Y

Default messaging provider

The default messaging provider for WebSphere Application Server is the service integration bus, providing both point-to-point and publish/subscribe functions. Within this provider, you define JMS connection factories and JMS destinations corresponding to service integration bus destinations.

WebSphere MQ JMS provider

WebSphere Application Server supports the use of full WebSphere MQ as the JMS provider. The product is tightly integrated with the WebSphere installation, with WebSphere providing the JMS client classes and administration interface, while WebSphere MQ provides the queue-based messaging system.

Generic JMS providers

WebSphere Application Server supports the use of generic JMS providers, as long as they implement the ASF component of the JMS 1.0.2 specification. JMS resources for generic JMS providers are not configurable using WebSphere administration.

V5 default JMS provider

For backwards compatibility with earlier releases, WebSphere Application Server also includes support for the V5 default messaging provider which enables you to configure resources for use with V5 embedded messaging. The V5 default messaging provider can also be used with a service integration bus.

3.15.6 Resource environment providers

The java:comp/env environment provides a single mechanism by which both JNDI name space objects and local application environment objects can be looked up. WebSphere Application Server provides a number of local environment entries by default.

The J2EE specification also provides a mechanism for defining custom (nondefault) environment entries using <resource-env-ref> entries defined in an application's standard deployment descriptors. The J2EE specification separates the definition of the resource environment entry from the application by:

1. Requiring the application server to provide a mechanism for defining separate administrative objects that encapsulate a resource environment entry. The administrative objects are to be accessible with JNDI in the application server's local name space (java:comp/env).

2. Specifying the administrative object's JNDI lookup name and the expected returned object type in <resource-env-ref>.

WebSphere Application Server supports the <resource-env-ref> mechanism by providing administration objects for the following:

► Resource environment provider

This provider defines an administrative object that groups together the referenceable, resource environment entry administrative objects and any required custom properties.

► Referenceable

The Referenceable provider defines the class name of the factory class that returns object instances implementing a Java interface.

► Resource environment entry

This entry defines the binding target (JNDI name), factory class and return object type (with the link to the referenceable) of the resource environment entry.

3.16 Workload management

Clustering application servers that host Web containers automatically enables plug-in workload management for the application servers and the servlets they host. Routing of servlet requests occurs between the Web server plug-in and the clustered application servers using HTTP or HTTPS.
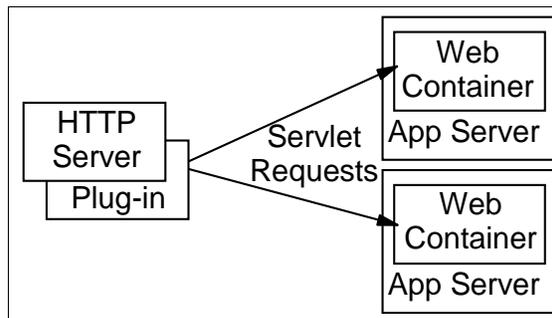
Figure 3-14   Plug-in (Web container) workload management

This routing is based on weights associated with the cluster members. If all cluster members have identical weights, the plug-in sends equal requests to all members of the cluster assuming no strong affinity configurations. If the weights are scaled in the range from zero to twenty, the plug-in routes requests to those cluster members with the higher weight value more often. A rule of thumb formula for determining routing preference would be:

```
% routed to Server1 = weight1 / (weight1+weight2+...+weightn)
```

There are n cluster members in the cluster.

The Web server plug-in temporarily routes around unavailable cluster members.

Workload management for EJB containers can be performed by configuring the Web container and EJB containers on separate application servers. Multiple application servers with the EJB containers can be clustered, enabling the distribution of EJB requests between the EJB containers.
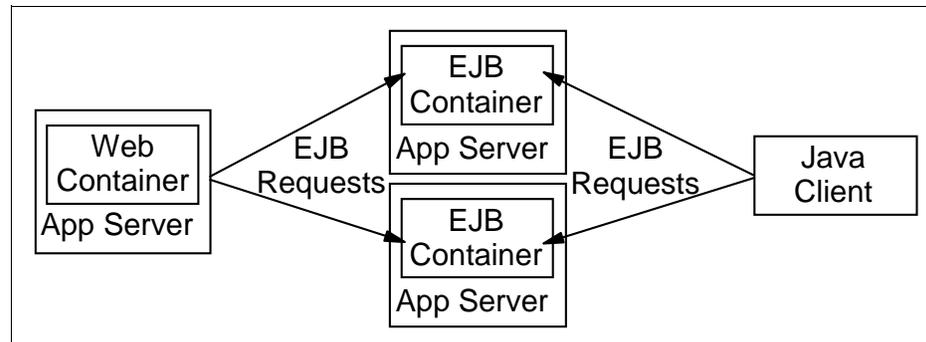
Figure 3-15   EJB workload management

In this configuration, EJB client requests are routed to available EJB containers in a round robin fashion based on assigned server weights. The EJB clients can be servlets operating within a Web container, stand-alone Java programs using RMI/IIOP, or other EJBs.

The server-weighted, round-robin routing policy ensures a distribution based on the set of server weights that have been assigned to the members of a cluster. For example, if all servers in the cluster have the same weight, the expected distribution for the cluster would be that all servers receive the same number of requests. If the weights for the servers are not equal, the distribution mechanism sends more requests to the higher weight value servers than the lower weight value servers. The policy ensures the desired distribution, based on the weights assigned to the cluster members. In V6, the balancing mechanism for weighted round-robin had been enhanced to ensure more balanced routing distribution among servers.

You can also choose to have requests sent to the node on which the client resides as the preferred routing. In this case, only cluster members on that node will be chosen (using the round-robin weight method). Cluster members on remote nodes will only be chosen if a local server is not available.

3.17 High availability

With WebSphere Network Deployment V6, the high availability features have been significantly improved. The following is a quick overview of the failover capabilities:

► HTTP server failover

Combining multiple HTTP servers, along with a load balancing product such as provided with the Edge components can be used to provide HTTP Server failover.

► Web container failover

The HTTP server plug-in in the Web server is aware of the configuration of all Web containers and can route around a failed Web container in a cluster. Sessions can be persisted to a database or in-memory using data replication services.

► EJB container failover

Client code and the ORB plug-in can route to the next EJB container in the cluster.

► Deployment manager and node agent

The need for failover in these two components has been reduced, thus no built-in failover capability has been provided. The loss of the deployment manager only affects configuration. It is recommended that you use a process nanny to restart the Node Agent if it fails.

► Critical services failover

Hot standby and peer failover for critical services (WLM routing, PMI aggregation, JMS messaging, transaction manager, and so on) is provided through the use of HA domains.

An HA domain defines a set of WebSphere processes (a core group) that participate in providing high availability function to each other. Processes in the Core Group can be the deployment manager, Node Agents, application servers or cluster members.

One or more members of the core group can act as an HA coordinator, managing the HA activities within the core group processes. If an HA coordinator server fails, another server in the core group takes over the HA coordinator duties. HA policies define how the failover occurs.

Workload management information is shared between core members and failover of critical services is done among them in peer-to-peer fashion. Little configuration is necessary and, in many cases, this function works with the defaults that are created automatically as you create the processes.

► Transaction log hot standby

With WebSphere Application Server Network Deployment V6, transaction logs can be maintained on NAS (Network Attached Storage). When a cluster member fails, another cluster member recovers the transaction log, thus enabling the failover 2PC transactions. The time required for the failover is dramatically reduced from what it took in V5.

► JMS messaging failover

The messaging engine keeps messages in a remote database. When a server in a cluster fails, WebSphere selects an online server to run the Messaging Engine and the workload manager routes JMS connections to that server.

3.18 Administration

WebSphere Application Server's administration model is based on the JMX framework. JMX allows you to wrap hardware and software resources in Java and expose them in a distributed environment. JMX also provides a mapping framework for integrating existing management protocols, such as SNMP, into JMX's own management structures.

Each application server has an administration service that provides the necessary functions to manipulate configuration data for the server and its components. The configuration is stored in a repository. The repository is actually a set of XML files stored in the server's file system.

3.18.1 Administration tools

Table 3-12 shows the administration tool support for WebSphere Application Server by configuration.

Table 3-12   WebSphere Application Server administration tool support

Tool	Express & Base	ND
Administrative console	Yes	Yes
Commands	Yes	Yes
wsadmin	Yes	Yes

Administrative console
The WebSphere Administrative Console is a Web-browser based interface that provides configuration and operation capability. The administrator connects to

the application using a Web browser client. Users assigned to different administration roles can manage the application server and certain components and services using this interface.

The administrative console is a system application, crucial to the operation of WebSphere and as such, is not exposed as an enterprise application on the console. In standalone application servers, the administrative console runs in the application server. In the Network Deployment distributed server environment, the administrative console application runs on the deployment manager. When a node is added to a cell, the administrative console application is deleted from the node and the configuration files are integrated into the master cell repository to be maintained by the deployment manager.

Commands

WebSphere Application Server provides a set of commands in the *<server_install>*/bin directory with which you can perform a subset of administrative functions.

For example, the `startServer` command is provided to start an application server.

wsadmin scripting client

The wsadmin scripting client provides extra flexibility over the Web-based administration application, allowing administration using the command-line interface. Using the scripting client not only makes administration quicker, but helps automate the administration of multiple application servers and nodes using scripts.

The scripting client uses the Bean Scripting Framework (BSF), which allows a variety of scripting languages to be used for configuration and control. Only two languages have been tested and are supported in V6: jacl and jython (or jpython).

The wsadmin scripting interface is included in all WebSphere Application Server configurations but is targeted toward advanced users. The use of wsadmin requires indepth familiarity with application server architecture and a scripting language.

3.18.2 Configuration repository

The configuration repository holds copies of the individual component configuration documents stored in XML files. The application server's admin service takes care of the configuration and makes sure it is consistent during the runtime.

The configuration of unfederated nodes can be archived for export and import, making them portable among different WebSphere Application Server instances.

3.18.3 Centralized administration

The Network Deployment package allows multiple servers and nodes to be administered from a central location. This is facilitated through the use of a central deployment manager that handles the administration process and distributes the updated configuration to the node agent for each node. The node agent, in turn, is responsible for maintaining the configuration for the servers in the node.

Table 3-13   WebSphere Application Server distributed administration support

Administration tool	Express & Base	ND
Deployment manager	No	Yes
Node agent	No	Yes
Application servers	Standalone	Standalone or distributed server, clustering

Managed processes

All operating system processes that are components of the WebSphere product are called managed servers or managed processes. JMX support is embedded in all managed processes and these processes are available to receive administration commands and to output administration information about the state of the managed resources within the processes.

WebSphere provides the following managed servers/processes:

► Deployment manager

  The deployment manager provides a single point to access all configuration information and control for a cell. The deployment manager aggregates and communicates with all of the node agent processes on each node in the system.

► Node agent

  The node agent aggregates and controls all of the WebSphere managed processes on its node. There is one node agent for each node.

► Application server

This is the managed server that hosts J2EE applications.

Deployment manager

The deployment manager process provides a single, central point of administrative control for all elements in the cell. It hosts the Web-based administrative console application. Administrative tools that need to access any managed resource in a cell usually connect to the deployment manager as the central point of control.

The deployment manager is responsible for the content of the repositories (configuration and application binaries) on each of the nodes. It manages this through communication with the node agent process resident on each node of the cell.

Using the deployment manager, horizontal scaling, vertical scaling and distributed applications are all easy to administer and manage, because application servers are node-managed, and one or more nodes is cell-managed.

Node agent

The node agent is an administrative process and is not involved in application serving functions. It hosts important administrative functions such as:

► File transfer services
► Configuration synchronization
► Performance monitoring

The node agent aggregates and controls all the managed processes on its node by communicating with:

► The deployment manager to coordinate configuration synchronization and to perform management operations on behalf of the deployment manager

► Application servers and managed Web servers to manage (start and stop) each server and to update its configuration and application binaries as required

Only one node agent is defined and run on each node. In a standalone server environment, there is no node agent.

Master configuration repository

In a distributed server environment, a master configuration repository that contains all of the cell's configuration data is maintained by the deployment manager. The configuration repository at each node is a synchronized subset of the master repository. The node repositories are read-only for application server

access because only the deployment manager can initiate their update, pushing configuration changes out from the cell master configuration repository.

3.19 Application flow

Figure 3-16 shows the typical application flow for Web browser clients using either JDBC (from a servlet) or EJB to access application databases.
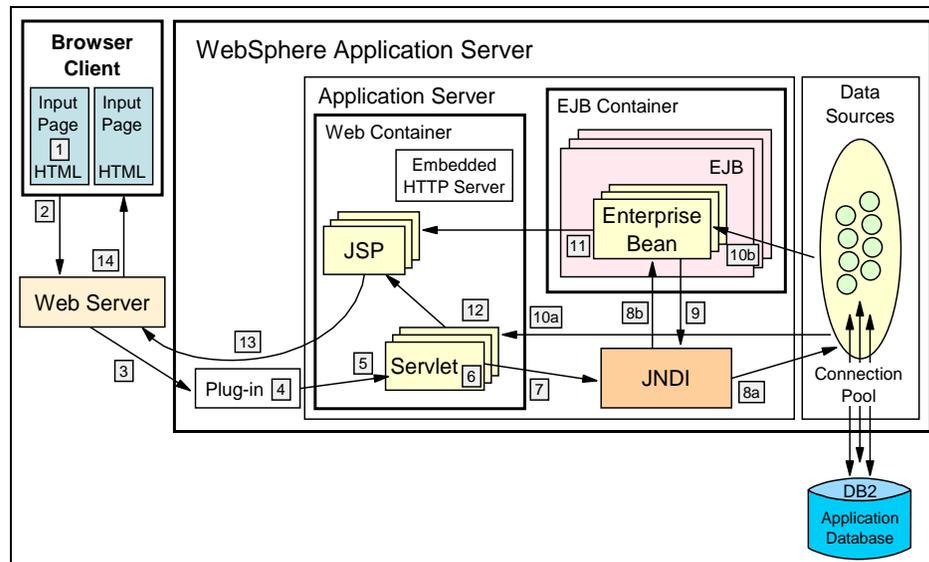
Figure 3-16   Application flow

1. A Web client requests a URL in the browser (input page).

2. The request is routed to the Web server over the Internet.

3. The Web server immediately passes the request to the Web server plug-in. All requests go to the WebSphere plug-in first.

4. The Web server plug-in examines the URL, verifies the list of hostname aliases from which it will accept traffic based on the virtual host information, and chooses a server to handle the request.

5. A stream is created. A stream is a connection to the Web container. It is possible to maintain a connection (stream) over a number of requests. The Web container receives the request and, based on the URL, dispatches it to the proper servlet.

6. If the servlet class is not loaded, the dynamic class loader loads the servlet (servlet *init( )*, then *doGet( )* or *doPost( )*).

7. JNDI is now used for lookup of either datasources or EJBs required by the servlet.

8. Depending upon whether a datasource is specified or an EJB is requested, the JNDI will direct the servlet:

    a. To the corresponding database, and get a connection from its connection pool in the case of a data source

    b. To the corresponding EJB container, which then instantiates the EJB when an EJB is requested

9. If the EJB requested involves an SQL transaction, it goes back to the JNDI to look up the datasource.

10. The SQL statement is executed and the data retrieved is sent back:

    a. To the servlet
    b. To the EJB

11. Data beans are created and handed off to JSPs in the case of EJBs.

12. The servlet sends data to JSPs.

13. The JSP generates the HTML that is sent back through the WebSphere plug-in to the Web server.

14. The Web server sends the output page (output HTML) to the browser.

3.20 Developing and deploying applications

Figure 3-17 on page 112 shows a high level view of the stages of application development and deployment.
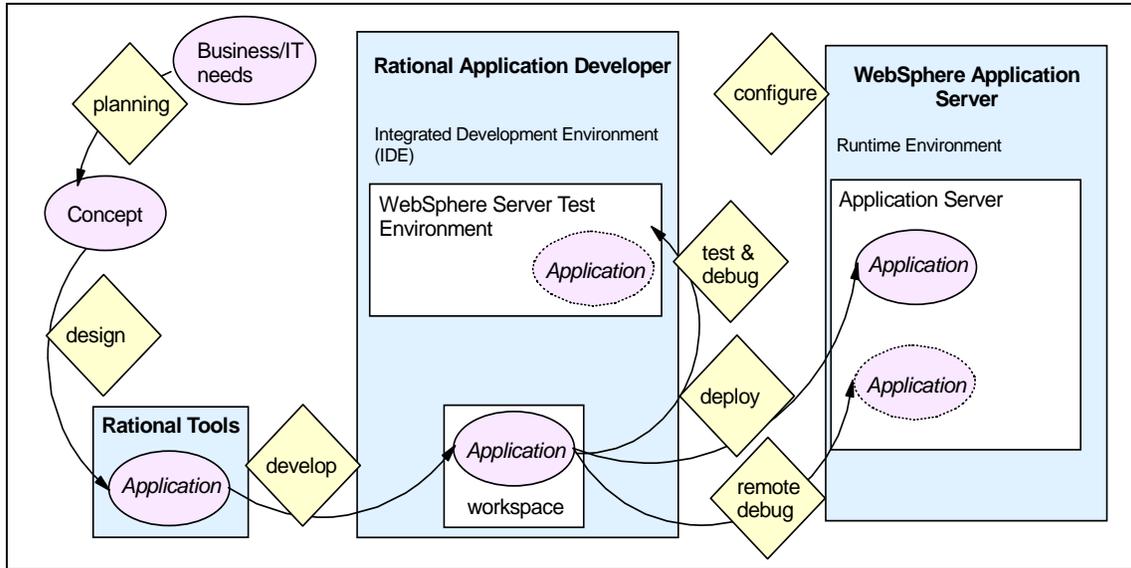
Figure 3-17   Develop and deploy

3.20.1 Application design

Design tools like Rational Rose or Rational XDE can be used to model the application using the Unified Modeling Language (UML). The output of the modeling will generally consist of use case scenarios, class diagrams, and starter code generated based on the model.

3.20.2 Application development

Application development is done using Rational Application Developer (or a comparable IDE) to create the enterprise application.

You can start by importing pregenerated code such as from Rational Rose, a sample application, an existing production application, or you can start from scratch.

Rational Application Developer provides many tools and aids to get you started quickly. It also supports team development using CVS or Rational ClearCase, allowing multiple developers to share a single master source copy of the code.

During the development phase, component testing can be done using the built-in WebSphere Application Server test environment. Rational Application Developer provides server tools capable of creating and managing servers both in the test environment and on remote server installations. The application is automatically

packaged into an EAR file for deployment when you run the application on a server using Rational Application Developer.

3.20.3 Application packaging

J2EE applications are packaged into Enterprise Application Archive (EAR) files to be deployed to one or more application servers. A J2EE application contains any or all of the modules shown in Table 3-14.

Table 3-14   J2EE 1.3 application modules

Module	Filename	Contents
Web module	<module>.war	Servlets, JSP files, and related code artifacts.
EJB module	<module>.jar	Enterprise beans and related code artifacts.
Application client module	<module>.jar	Application client code.
Resource adapter module	<module>.rar	Library implementation code that your application uses to connect to Enterprise Information Systems (EIS).

This packaging is done automatically in Rational Application Developer when you export an application for deployment. If you are using another IDE, WebSphere Application Server (with the exception of the Express configuration) provides the Application Server Toolkit for packaging applications.

Enhanced EAR files

The enhanced EAR, introduced in WebSphere Application Server V6.0, is a regular J2EE EAR file with additional configuration information for resources usually required by J2EE applications. While adding this extra configuration information at packaging time is not mandatory, it can simplify deployment of J2EE applications to WebSphere.

When an enhanced EAR is deployed to a WebSphere Application Server V6.0 server, WebSphere can automatically configure the resources specified in the enhanced EAR. This reduces the number of configuration steps required to set up the WebSphere environment to host the application.

3.20.4 Application deployment

Applications are installed on application servers using the WebSphere Administrative Console or the wsadmin scripting interface. An application can be deployed to a single server or a cluster. In the case of a cluster, it is installed on each application server in the cluster.

Installing an application involves the following:

- ► Defining JNDI names for EJB home objects
- ► Specifying data source entries for entity beans
- ► Binding EJB references to the actual EJB JNDI names
- ► Mapping Web modules to virtual hosts
- ► Specifying listener ports for message-driven beans
- ► Mapping application modules to application servers
- ► Mapping security roles to users or groups
- ► Binding resource references (created during packaging) to actual resources

    For example, a data source would need to be bound to a real database.

The use of an enhanced EAR file simplifies this installation process.

After a new application is deployed, the Web server plug-in configuration file needs to be regenerated and copied to the Web server.

Application update
In previous releases, deploying an update to an application required a complete EAR file to be deployed and the application to be restarted. WebSphere Application Server V6 now allows partial updates to applications and makes it possible to restart only parts of an application.

Updates to an application can consist of individual application files, application modules, zip files containing application artifacts, or the complete application. All module types can be started (though only Web modules can be stopped).

In V6, you have a rollout start option for installing applications on a cluster that will stop, update, and start each cluster member in turn, ensuring availability.

3.20.5 WebSphere Rapid Deployment

WebSphere Rapid Deployment is designed to simplify the development and deployment of WebSphere applications. It is a collection of Eclipse plug-ins that can be integrated within development tools or run in a headless mode from a user file system. WebSphere Rapid Deployment is currently integrated in Rational Web Developer, Rational Application Developer, and the Application Server Toolkit. Initially, there are features that are only supported in headless mode.

During development, annotation-based programming is used. The developer adds metadata tags into the application source code that are used to generate artifacts needed by the code, thus reducing the number of artifacts the developer needs to create.

These applications are packaged into an enhanced EAR file containing the J2EE EAR file along with deployment information, application resources, and properties (environment variables, JAAS authentication entries, shared libraries, classloader settings, and JDBC resources). During installation, this information is used to create the necessary resources. Moving an application from one server to another also moves the resources.

WebSphere Rapid Deployment automates installation of applications and modules onto a running application server by monitoring the workspace for changes and then driving the deployment process.

3.21 Technology support summary

The following tables break down the highlights of support provided by each WebSphere Application Server packaging option.

Table 3-15   WebSphere Application Server features and technology support

Support type	Base and Express V6	ND V6
Client and server support for the Software Development Kit for Java Technology Edition 1.4 (SDK 1.4.2)	Yes	Yes
J2EE 1.2, 1.3 programming support	Yes	Yes
J2EE 14. programming support[1] ► EJB 2.1 ► Servlet 2.4 ► JSP 2.0 ► JMS 1.1 ► JTA 1.0 ► JavaMail 1.3 ► JAF 1.0 ► JAXP 1.2 ► Connector 1.5 ► Web Services 1.1 ► JAX-RPC 1.1 ► SAAJ 1.2 ► JAXR 1.0 ► J2EE Management 1.0 ► JMX 1.2 ► JACC 1.0 ► JDBC 3.0	Yes	Yes
WebSphere Rapid Deployment	Yes	Yes

Support type	Base and Express V6	ND V6
Service Data Objects (SDO)	Yes	Yes
Messaging support ▶ Integrated JMS 1.1 messaging provider ▶ Support for WebSphere MQ and generic JMS providers ▶ Message driven beans	Yes	Yes
Web services runtime support	Yes	Yes
Security support ▶ Java 2 ▶ J2EE ▶ JACC 1.0 ▶ JAAS 1.0 ▶ CSIv2 and SAS authentication protocols ▶ LDAP or local operating system user registry[3] ▶ Simple WebSphere Authentication Mechanism (SWAM) ▶ LTPA authentication mechanism ▶ Kerberos (Technology Preview	Yes	Yes
Multi-node management and Edge components		
Workload management and failover	No	Yes
Deployment manager	No	Yes
Central administration of multiple nodes	No	Yes
Load Balancer		Yes
Caching Proxy		X
Dynamic caching	Yes	Yes
Performance and analysis tools		
Performance Monitoring Infrastructure (PMI)	Yes	Yes
Log Analyzer	Yes	Yes
Tivoli Performance Viewer (integrated in the administration console)	Yes	Yes
Administration and tools		

Support type	Base and Express V6	ND V6
Administration and tools ► Web-based administration console ► Integrated IBM HTTP Server and Application Server Administration Console ► Administrative scripting ► Java Management Extension (JMX) 1.2 ► J2EE Management (JSR-077) ► J2EE Deployment (JSR-088) ► Application Server Toolkit	Yes	Yes
Web services ► JAX-RPC v1.0 for J2EE 1.3, v1.1 for J2EE 1.4 ► JSR 109 (Web services for J2EE) ► WS-I Basic Profile 1.1.2 support ► WS-I Simple SOAP Binding Profile 1.0.3 ► WS-I Attachments Profile 1.0 ► SAAJ 1.2 ► UDDI V2 and V3 ► JAXR ► WS-TX (transactions) ► SOAP 1.1 ► WSDL 1.1 for Web services ► WSIL 1.0 for Web services ► OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) ► OASIS Web Services Security: UsernameToken Profile 1.0 ► OASIS Web Services Security X.509 Certificate Token Profile	Yes	Yes
Web Services Gateway	No	Yes
Private UDDI v3 Registry	Yes	Yes
Programming Model Extensions[2]		

Support type	Base and Express V6	ND V6
► Last Participant Support ► Internationalization Service ► WorkArea Service ► ActivitySession Service ► Extended JTA Support ► Startup Beans ► Asynchronous Beans (now called WorkManager) ► Scheduler Service (now called Timer Service) ► Object Pools ► Dynamic Query ► Web Services Gateway Filter Programming Model (with ► migration support) ► Distributed Map ► Application Profiling	Yes	Yes
► Back-up Cluster Support ► Dynamic WLM	No	Yes
1. The APIs required for J2EE 1.4 can be seen in the Application Programming Interface section of the J2EE 1.4 specifications at http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf 2. Business process choreography and business rule beans remain in WebSphere Business Integration Server Foundation.		

4

Getting started

This chapter provides an overview of the installation, configuration and administration of WebSphere Application Server - Express. We discuss the setup used in our redbook environment. Remember that we are not attempting to describe the steps necessary to setup and configure production sites.

For details about planning and administering production sites, see:

► *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446

► *WebSphere Application Server V6 System Management & Configuration Handbook*, SG24-6451

119

4.1 Product packaging

WebSphere Application Server - Express is a member of the WebSphere Application Server family of products. Express is aimed at small to medium-sized businesses or departments of large organization that are interested in developing and deploying dynamic Web sites. It provides these organizations with development and administration tools that enable them build and manage dynamic Web sites with ease.

WebSphere Application Server - Express is bundled with an application development tool (Rational Web Developer) that enable users to quickly and easily build Web applications that include support for J2EE 1.4 features although users can choose other development tools as they wish. This is an outline of what is included with WebSphere Application Server - Express V6:

► WebSphere Application Server
► IBM HTTP Server V6 Web server plug-ins
► IBM HTTP Server
► Application Client
► Application Server Toolkit
► JDBC Driver for WebSphere Application Server
► Development Tool - Rational Web Developer
► IBM DB2 Universal Database Express V8.2

> **Note:** Not all features are available on all platforms.

The following items are the supported platforms:

► Unix family - AIX, H-UX, Linux(intel, iSeries, pSeries, zSeries), Solaris
► Windows

4.2 Rational Web Developer

Rational Web Developer is the new development tool for creating Web applications that will run on the WebSphere Application Server family of products. It supports applications developed for version 4, 5 and 6. It is intended for Web developers who develop and manage complex Web sites. Designed according to the J2SE™ and J2EE specifications, Rational Web Developer supports JSPs, Java Servlets, HTML, Javascript, and DHTML. Rational Web Developer further includes tools for developing images and animated GIFs.

The following items are included in the Rational Web Developer package:

► Page Designer

This is an advanced HTML and JSP editor and helps developers build complex Web pages. It also has dynamic element support, which enables the addition of other technologies.

► Web Site Designer

This is used to manage the overall appearance and configuration of a Web site. It has a new attribute view support where page properties can be easily and quickly modified.

► Page templates

Similar to style sheets which define attributes for HTML tags, templates can be used to provide a common look and feel. A template can be created from any JSP or HTML page. When a template is modified or changed, pages that make use of it are not automatically changed, but must be rebuilt for the template changes to apply.

► Struts v1.1

Struts v1.1 is supported and wizards for developing the following are included:

– Action wizards create a new action wizard, can also have action mapping.
– Action form wizard creates a new form bean.
– Exception wizard creates a new struts exception.
– Module wizard creates a new struts module.
– Web diagram creates a new web diagram.

4.3 Installing WebSphere Application Server - Express

In this section, we look at installing WebSphere Application Server - Express.

4.3.1 Hardware requirements

Before we start the installation process, we describe the system hardware requirements for installation on Windows. Because requirements can change from time to time, please be sure to check the overview of product requirements at this Web site:

http://ibm.com/software/webservers/appserv/express/requirements/

The same URL also lists the system requirements for all the platforms supported by WebSphere Application Server - Express. The detailed list of all hardware and software requirements for all platform supported by WebSphere Application Server - Express is at this Web site:

http://ibm.com/software/webservers/appserv/doc/latest/prereq.html

When we wrote this redbook, the supported platforms were:

- ► AIX
- ► Linux for x86
- ► Linux for pSeries
- ► Linux for iSeries
- ► HP-UX
- ► Solaris
- ► Windows 2000
- ► Windows 2003
- ► Windows XP Professional

The following hardware pieces are the minimum system requirements for installation on Windows:

- ► Intel Pentium® processor (or equivalent) at 500 MHz or faster, Intel EM64T, or AMD 64-bit, Opteron (32-bit Operating System support only)

- ► Minimum 990 MB free disk space for installation

- ► Minimum 512 MB physical memory, 1 GB recommended

- ► CD-ROM drive

4.3.2 Installing using the launchpad

If you are installing from a download, create an installation folder and unzip the downloaded files to that folder.

If you are installing from a CD, you are not required to unzip any files. You need to have administrator rights on the machine on which you are installing the software. If you do not have administrator rights, the correct components will not install properly.

The easiest way to install WebSphere Application Server - Express and its companion software is to use the launchpad. The steps are:

1. Either open the installation folder where you unzipped the download files or insert the product CD.

2. Run the launchpad.bat file, to start the installation launchpad. You can browse the launchpad to explore the features of WebSphere Application Server - Express and learn more about the components you can install.

   Figure 4-1 on page 123 shows the welcome page of the installation launchpad.
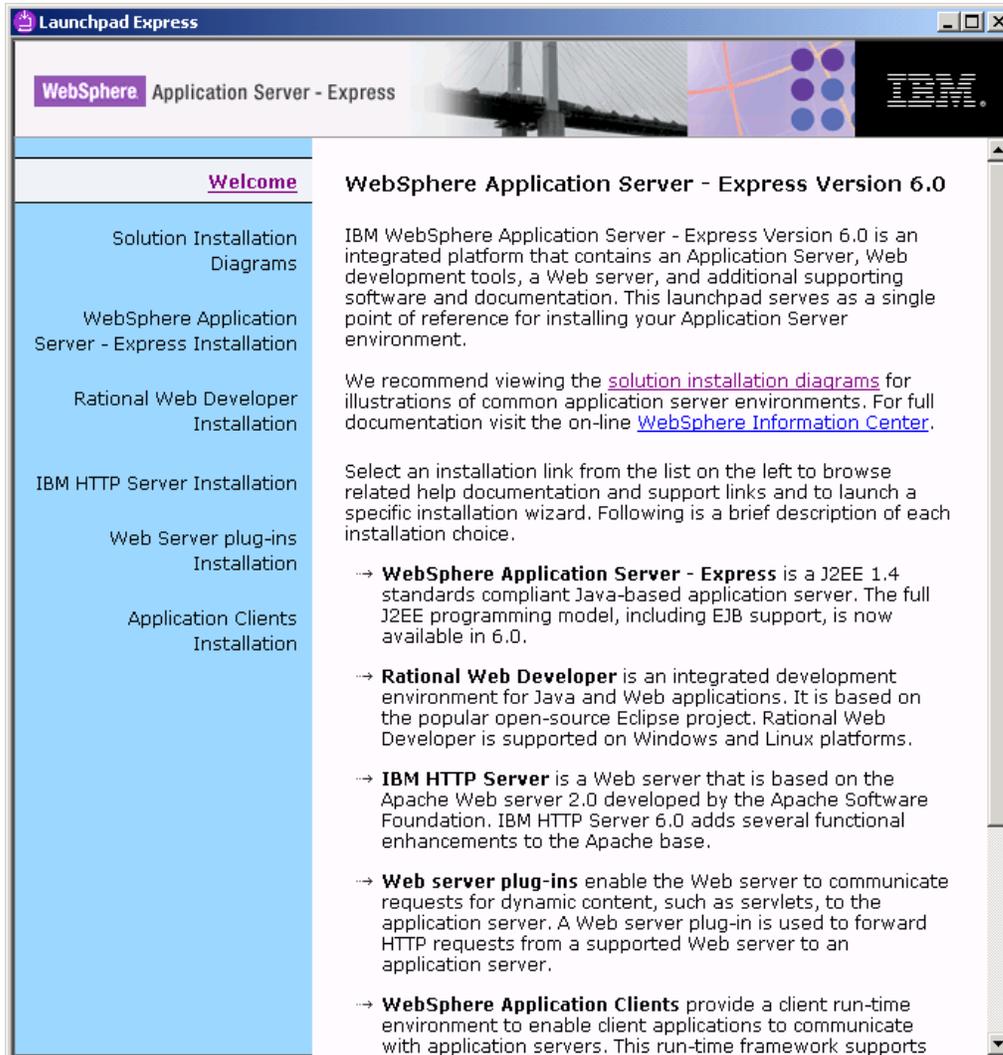
Figure 4-1   Installation launchpad

3. Click the **Solution Installation Diagrams** link to review the solution install diagrams and associated descriptions. We are setting up a simple development environment so we will do a single machine installation of Rational Web Developer and a single machine installation of WebSphere Application Server - Express. If you plan a production site, we recommend that you consider some of the alternative solution installations. You might also want to read the redbook *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446.

See Figure 4-2 for an example of the solution installation diagrams page of the launchpad.
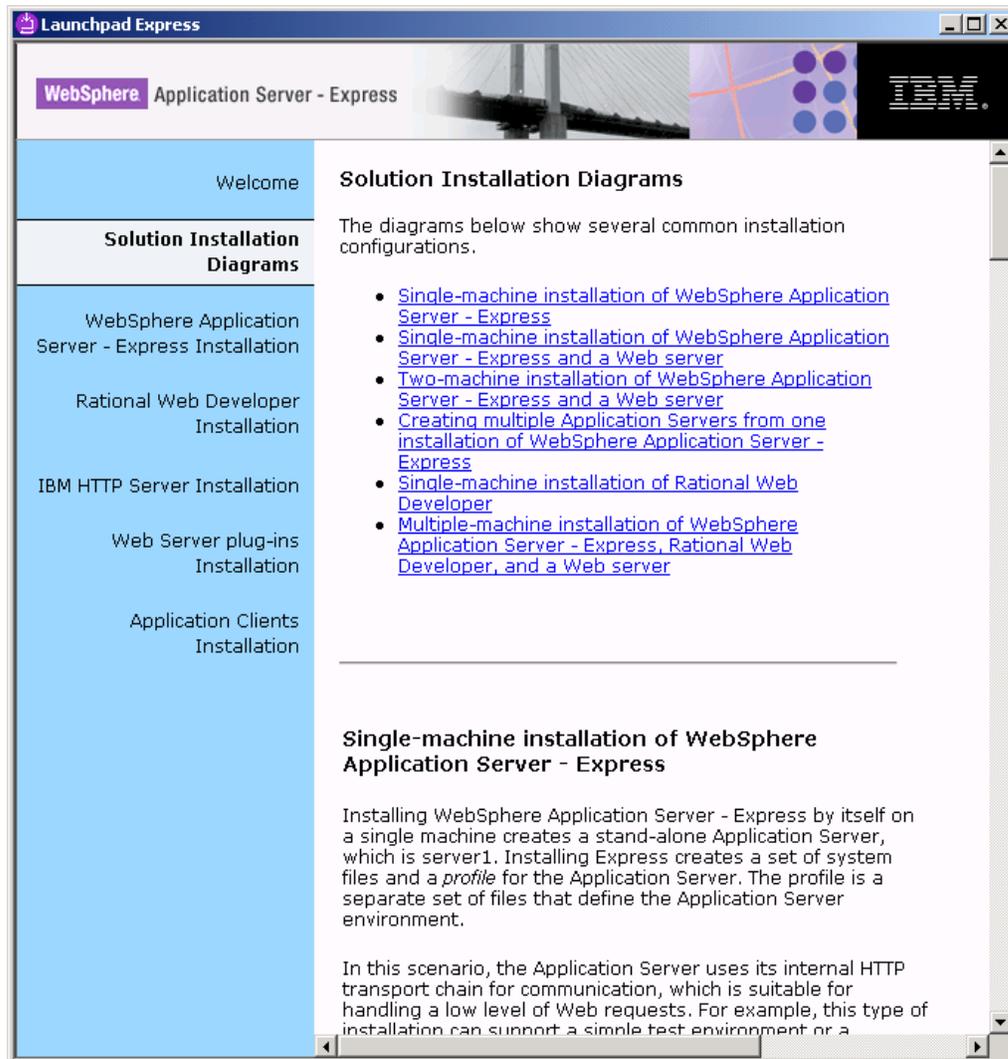
Figure 4-2   Solution installation diagrams

4.3.3 Install WebSphere Application Server - Express

The steps to install WebSphere Application Server - Express are:

1. Click the **WebSphere Application Server - Express Installation** link on the launchpad welcome page.

2. Click the L**aunch installation wizard for the WebSphere Application Server - Express** link on the WebSphere Application Server - Express installation page as shown in Figure 4-3.
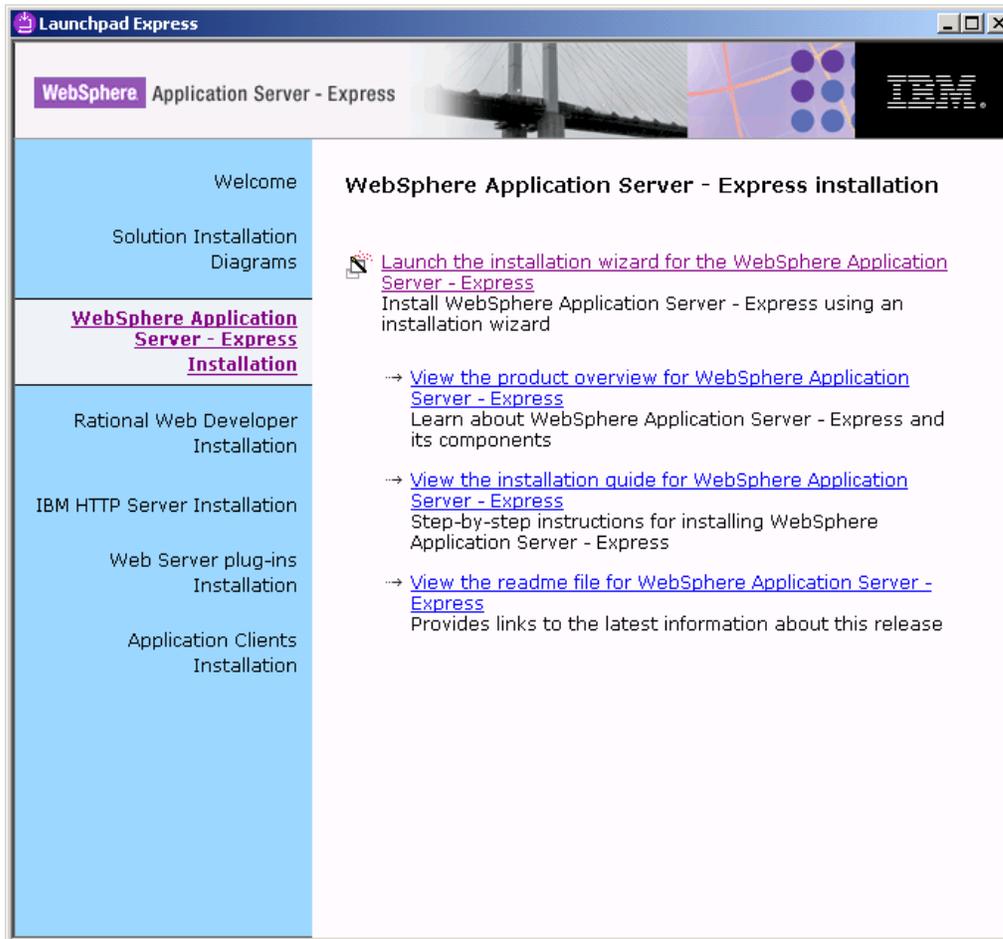
Figure 4-3   WebSphere Application Server - Express installation page

3. When the installation wizard appears, click **Next**. See Figure 4-4 on page 126.
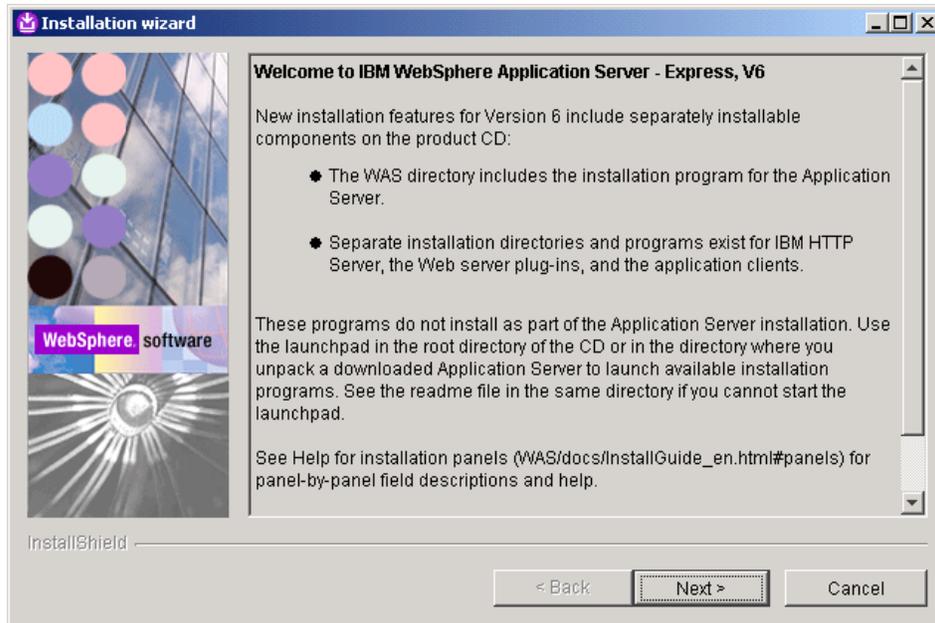
Figure 4-4   Installation wizard

4. Read and accept the terms and conditions. Click **Next** to continue. See Figure 4-5 on page 127.
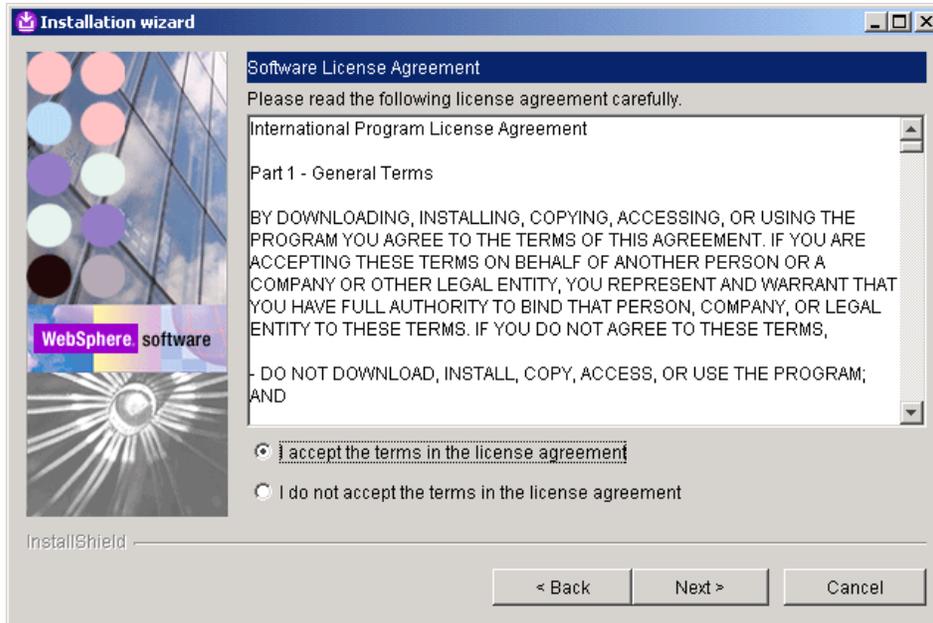
Figure 4-5   Terms and conditions for WebSphere Application Server - Express

5. The wizard checks your system to see if it meets the installation prerequisites. If the check is passed as shown in Figure 4-6 on page 128 then click **Next**.
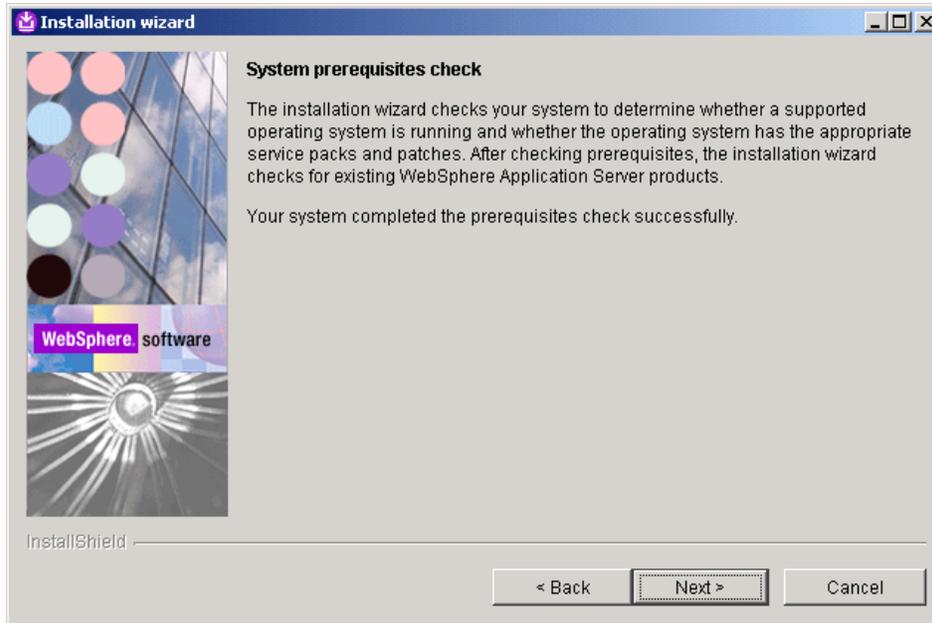
Figure 4-6   Prerequisite check

6. Choose an installation directory and click **Next**. Figure 4-7 on page 129 shows the default install location on Windows. We changed this to `<drive>\was`.
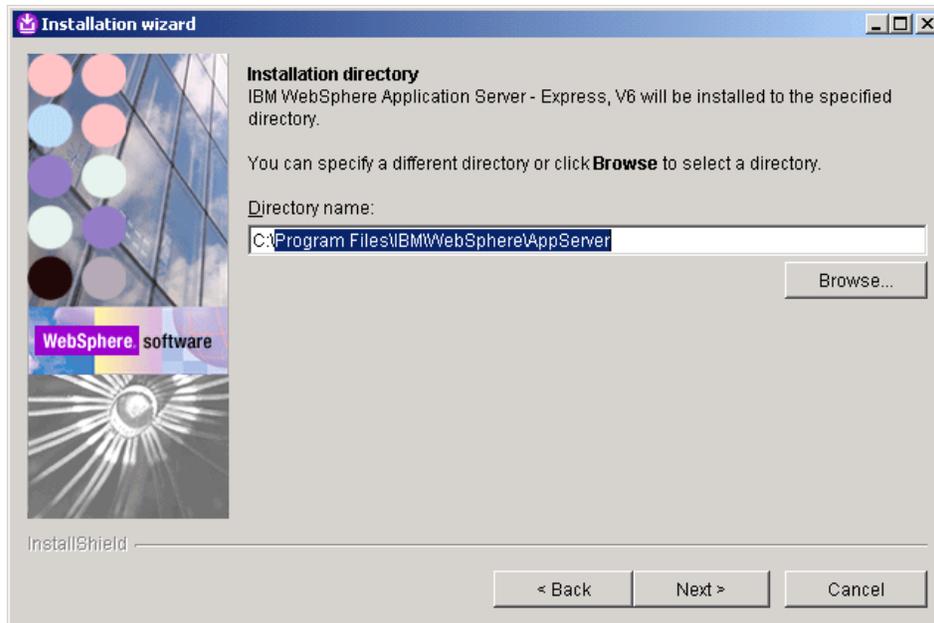
Figure 4-7   Installation directory

7. Choose **Custom installation** and click **Next**. See Figure 4-8 on page 130.
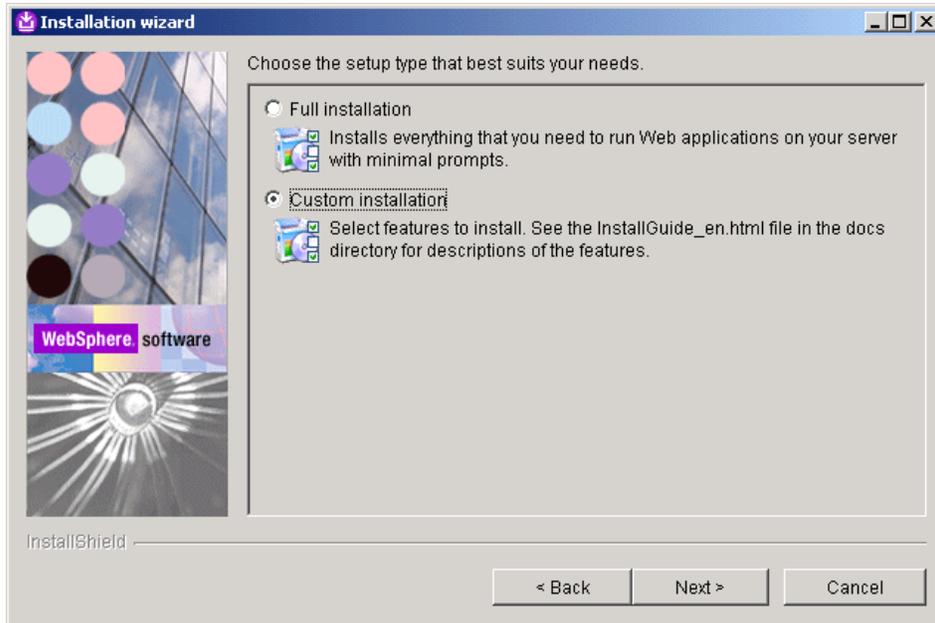
Figure 4-8   Select installation type

> 8. Select features to install and click **Next**. Figure 4-9 on page 131 shows the available features. We selected to install all available features.
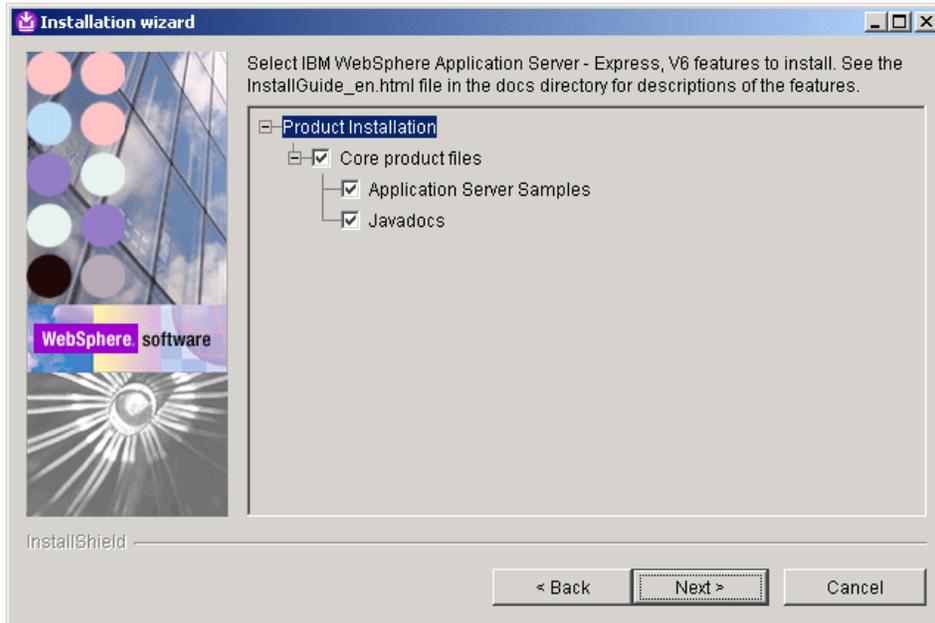
Figure 4-9   Features to install

9. The next page, as shown in Figure 4-10 on page 132 allows you to change the ports used by the application server. We used the default port values. Click **Next**.
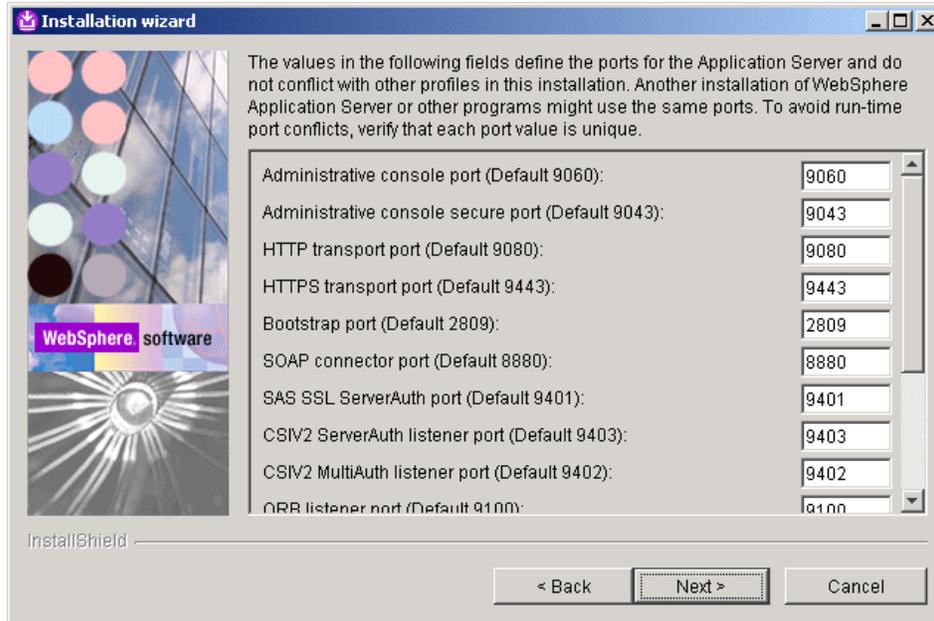
Figure 4-10   Ports used

10. Node and host names are chosen based on the hostname of the machine you are installing on. Figure 4-11 on page 133 shows an example of the names generated when we installed. Accept the default names and click **Next**.
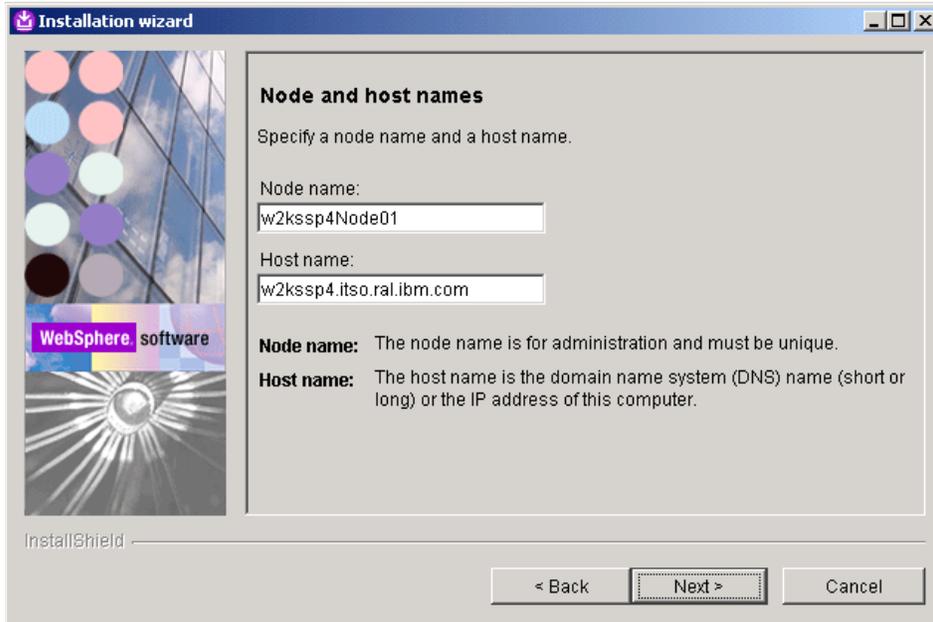
Figure 4-11   Node and host names

11. The next page allows you to choose how to start WebSphere Application Server. It can be run as a Windows service or by using a Windows user account. See Figure 4-12 on page 134 for an example. Click **Next**.
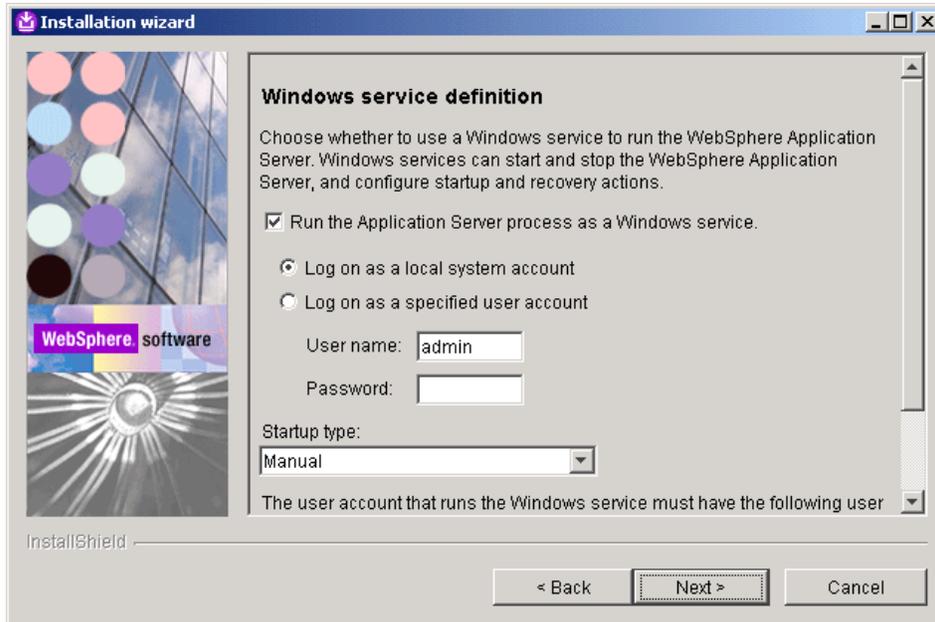
Figure 4-12   Startup options

12.The wizard shows you a summary of what is being installed. See Figure 4-13 on page 135 for an example. Click **Next**.
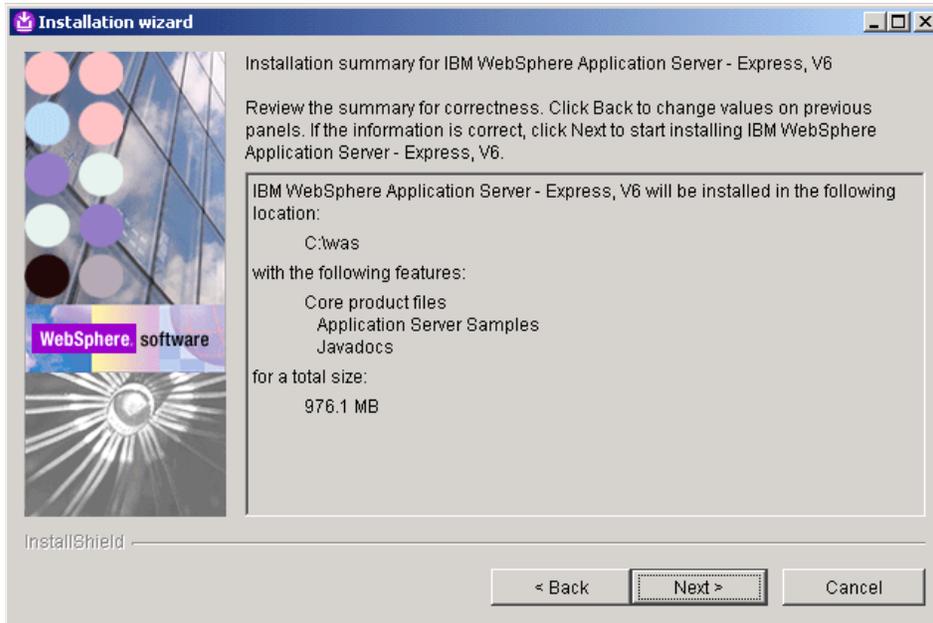
Figure 4-13   Installation summary

13. Figure 4-14 on page 136 shows the installation finish page after a successful installation of WebSphere Application Server - Express.
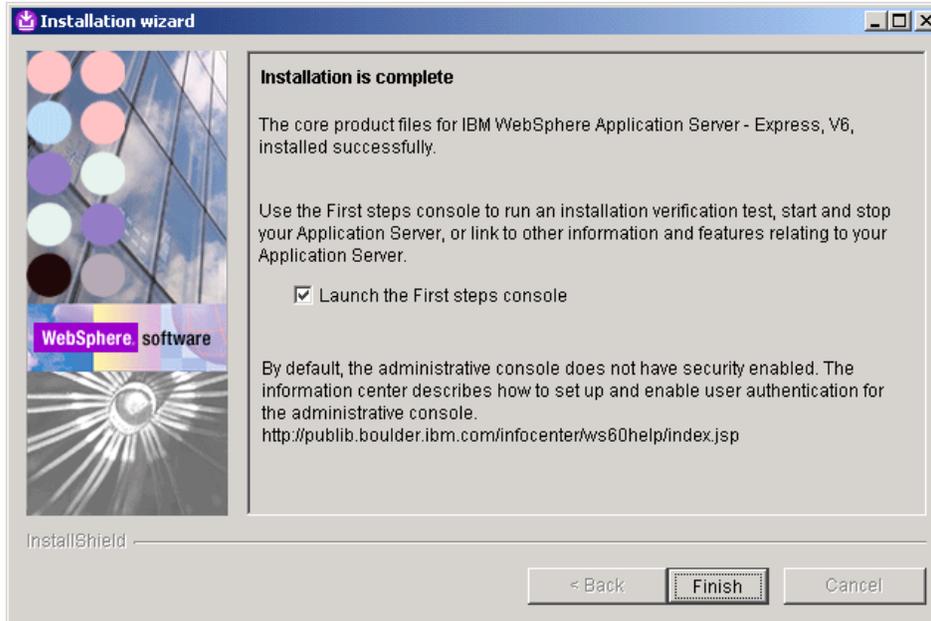
Figure 4-14   Installation complete

14. Click **Finish** to exit the installation wizard.

4.3.4 Using the first steps console

After installation of WebSphere Application Server - Express you can use the first steps console to verify that the server is correctly installed.

1. First steps can be launched from the last page of the installation wizard by selecting **Launch the First steps console** and clicking **Finish** as shown in Figure 4-14.

   You can also launch first steps by choosing **Start** → **Programs** → **IBM WebSphere** → **Application Server - Express v6** → **Profiles** → **default** → **First steps**. See Figure 4-15 on page 137 for an example of the first steps console.
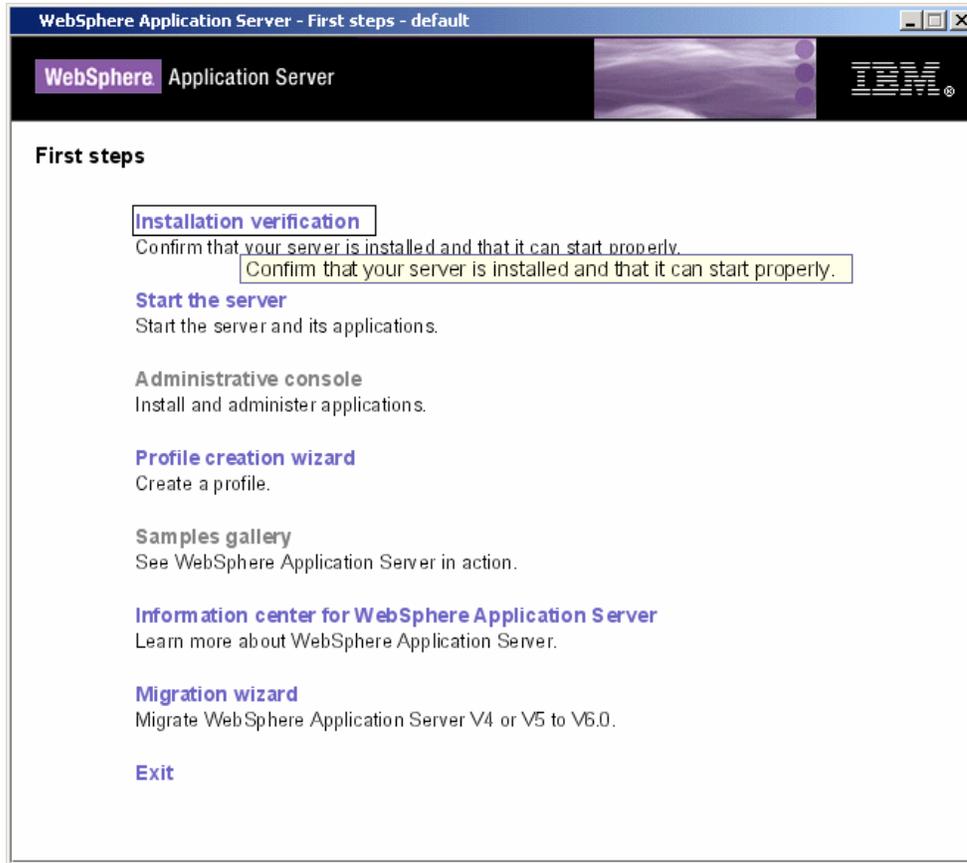
Figure 4-15   First steps console

2. Select **Installation verification** to start the server, and verify that the WebSphere Application Server - Express installation was successful. Figure 4-16 on page 138 shows the output from a successful installation verification.

```
First steps output - Installation verification                                    _ □ ×

Server name is:server1
Profile name is:default
Profile home is:C:\was\profiles\default
Profile type is:default
Cell name is:w2kssp4Node01Cell
Node name is:w2kssp4Node01
Current encoding is:Cp1252
Server port number is:9080
IVTL0020I: Could not connect to Application Server, waiting for server to start
IVTL0010I: Connecting to the WebSphere Application Server w2kssp4.itso.ral.ibm.com on port: 9080
IVTL0020I: Could not connect to Application Server, waiting for server to start
Start running the following command:cmd.exe /c "C:\was\profiles\default\bin\startServer.bat" server1 -profileName default
>ADMU7701I: Because server1 is registered to run as a Windows Service, the
>         request to start this server will be completed by starting the
>         associated Windows Service.
>ADMU0116I: Tool information is being logged in file
>
>         C:\was\profiles\default\logs\server1\startServer.log
>
>ADMU0128I: Starting tool with the default profile
>
>ADMU3100I: Reading configuration for server: server1
>
>ADMU3200I: Server launched. Waiting for initialization status.
>
>ADMU3000I: Server server1 open for e-business; process id is 1888
>
IVTL0015I: WebSphere Application Server w2kssp4.itso.ral.ibm.com is running on port 9080 for profile default
Testing server using the following URL:http://w2kssp4.itso.ral.ibm.com:9080/ivt/ivtserver?parm2=ivtservlet
IVTL0050I: Servlet Engine Verification Status - Passed
Testing server using the following URL:http://w2kssp4.itso.ral.ibm.com:9080/ivt/ivtserver?parm2=ivtAddition.jsp
IVTL0055I: JSP Verification Status - Passed
Testing server using the following URL:http://w2kssp4.itso.ral.ibm.com:9080/ivt/ivtserver?parm2=ivtejb
IVTL0060I: EJB Verification Status - Passed
IVTL0035I: Scanning the file C:\was\profiles\default\logs\server1\SystemOut.log for errors and warnings
IVTL0040I: 0 errors/warnings were detected in the file C:\was\profiles\default\logs\server1\SystemOut.log
IVTL0070I: IVT Verification Succeeded
IVTL0080I: Installation Verification is complete

◄ ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬                                        ►
```

Figure 4-16   Installation verification

3. You can use the first steps console to start the WebSphere Administrative
   Console. Click **Administrative Console** as shown in Figure 4-17 on
   page 139.

WebSphere Application Server – First steps – default

WebSphere Application Server   IBM.

First steps

Installation verification
Confirm that your server is installed and that it can start properly.

Stop the server
Stop the server and its applications.

Administrative console
Install and administer applications.

Install and administer applications.

Profile creation wizard
Create a profile.

Samples gallery
See WebSphere Application Server in action.

Information center for WebSphere Application Server
Learn more about WebSphere Application Server.

Migration wizard
Migrate WebSphere Application Server V4 or V5 to V6.0.

Exit

Figure 4-17   Using First steps to start the WebSphere Administrative Console

4. Figure 4-18 on page 140 shows the logon page of the WebSphere Administrative Console. As well as using first steps to start the WebSphere Administrative Console you can also use the following URLs in a Web browser:

– `http://localhost:9060/admin`

– `http://localhost:9060/ibm/console`
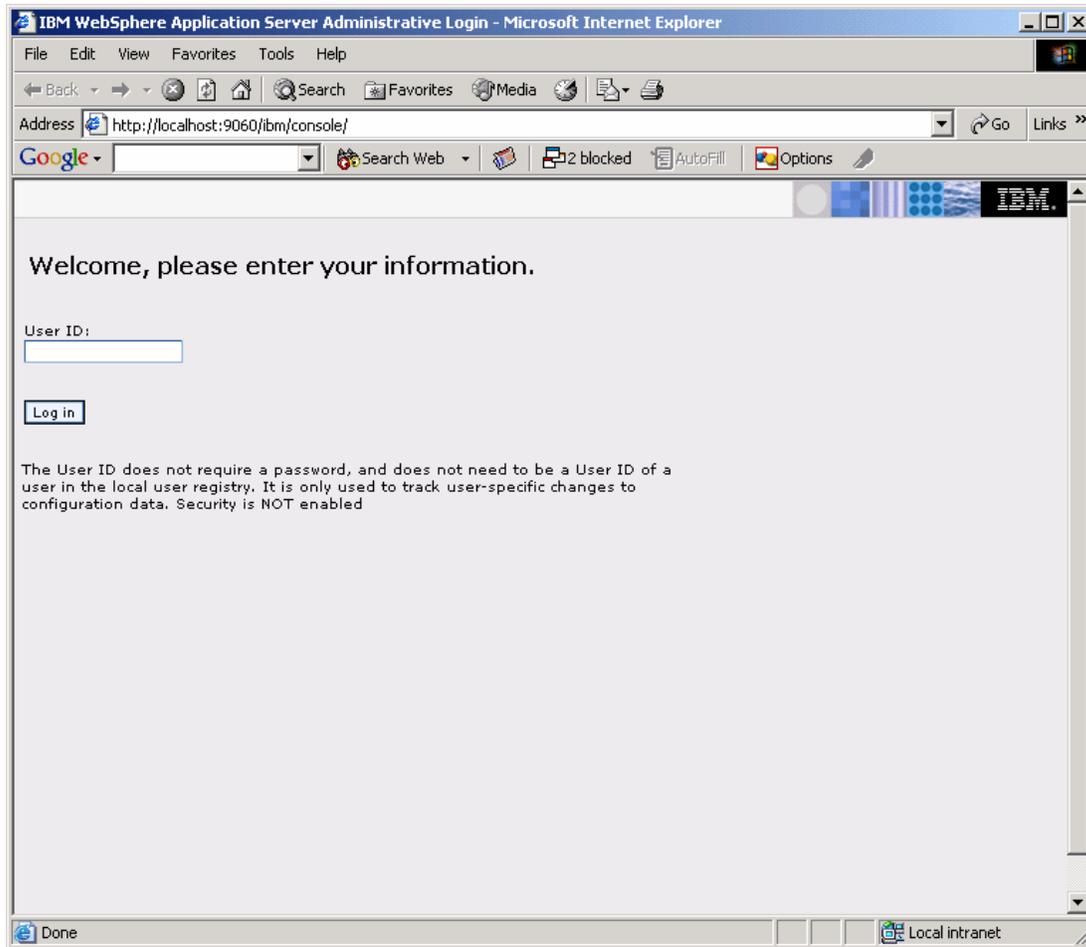
Figure 4-18   Logon page for the WebSphere Administrative Console

5. Security is not enabled by the WebSphere Application Server - Express installation so you can enter any ID for the WebSphere Administrative Console logon. Figure 4-19 on page 141 shows the main page of the WebSphere Administrative Console that is displayed after you logon.
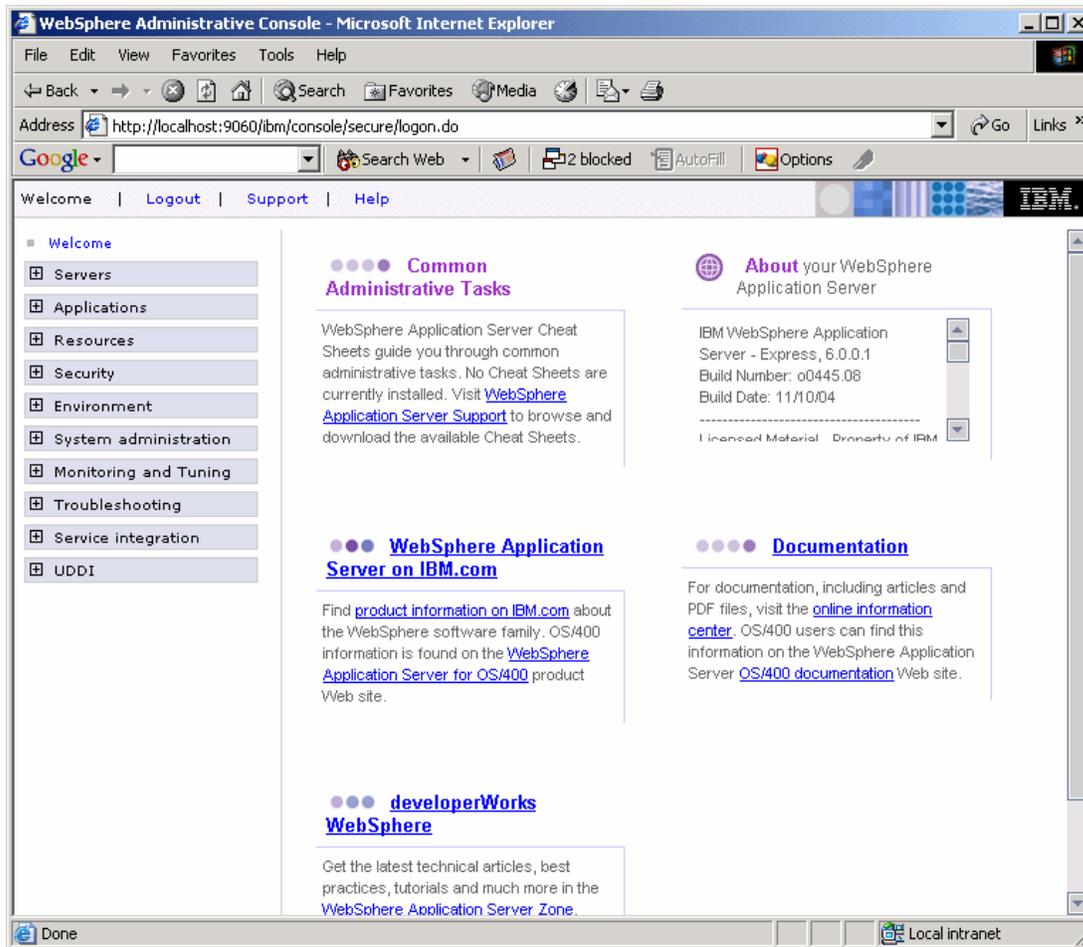
Figure 4-19   Main page of the WebSphere Administrative Console

6. After completing these verification steps, you can stop the Express Application Server by choosing **Stop the Server** from the first steps console. See Figure 4-20 on page 142 for an example of the first steps output from stopping the server.
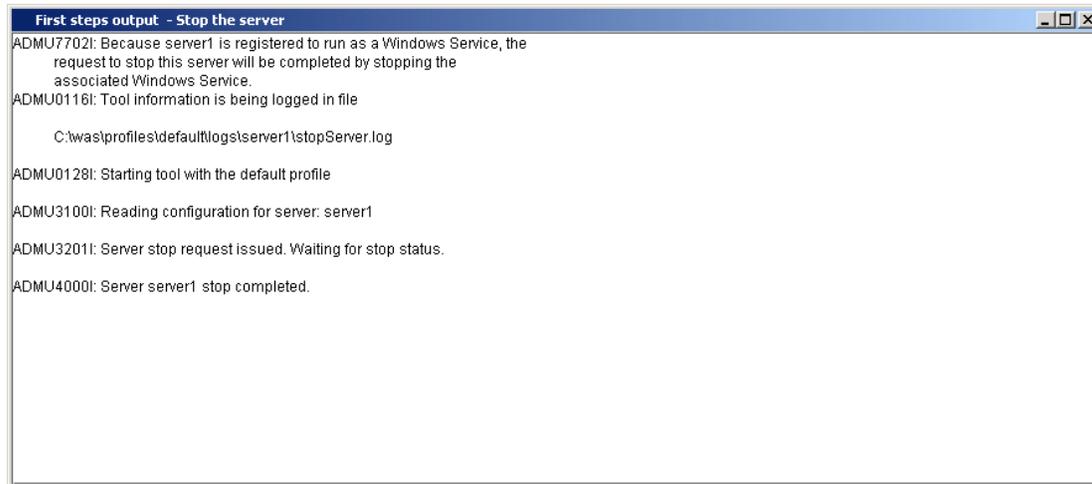
Figure 4-20   Using first steps to stop the server

7. Close the first steps output page and click **Exit** on the first steps console to close first steps.

4.4 Administration basics

In this section we look at some of the basic administration tasks for a WebSphere Application Server - Express installation on the Windows platform. Detailed administration guidance is not provided by our redbook. We suggest that you read the *WebSphere Application Server V6 System Management & Configuration Handbook*, SG24-6451 for detailed about administering WebSphere Application Server.

4.4.1 Starting and stopping the server

To start and stop the server, perform these tasks:

1. To start the server choose **Start** → **Programs** → **IBM WebSphere** → **Application Server - Express v6** → **Profiles** → **default** → **Start the server.** A command prompt will appear with to show messages about the server start.

   You can also start the server using the command line. Open a command prompt, change to the bin directory where you installed WebSphere Application Server - Express and enter:

   ```
   startServer server1
   ```

2. To stop the server, choose **Start** → **Programs** → **IBM WebSphere** → **Application Server - Express v6** → **Profiles** → **default** → **Stop the server.** A command prompt will appear with to show messages about the server stop.

You can also stop the server using the command line. Open a command prompt, change to the bin directory where you installed WebSphere Application Server - Express and enter:

```
stopServer server1
```

4.4.2 Starting the WebSphere Administrative Console

The WebSphere Administrative Console provides an integrated browser-based user interface where all aspects of the Express Application Server can be managed. The console is actually a Web application that runs on the Express Application Server , so the server must be running in order to access the WebSphere Administrative Console.

1. To start the WebSphere Administrative Console choose **Start** → **Programs** → **IBM WebSphere** → **Application Server - Express v6** → **Profiles** → **default** → **Administrative console.** When the server is initially set up security is not enabled by default, but the WebSphere Administrative Console requires a user ID to track configuration changes.

2. Enter a user name and click **OK**.

If you enter a user ID that is already in use, that is in the active session, you will receive a message indicating that another user is currently logged in with the same user ID and you will be prompted to take one of the following actions:

– Force the existing user ID out of the session.
– Wait for the existing user to log out or time out of the session.
– Specify a different user.

WebSphere Administrative Console basics

All aspects of the server can be managed from the menu on the left hand side of the WebSphere Administrative Console. In this section we look at some of the common administration tasks.

► Manage the server

The servers link allows you to manage your application server. Because we are running WebSphere Application Server - Express edition, we only have access to one server. The server option also allows you to manage any Web server that you have running in conjunction with your application server.

a. To display the servers to manage, choose **Servers** → **Application servers**. See Figure 4-21 on page 144 for an example.
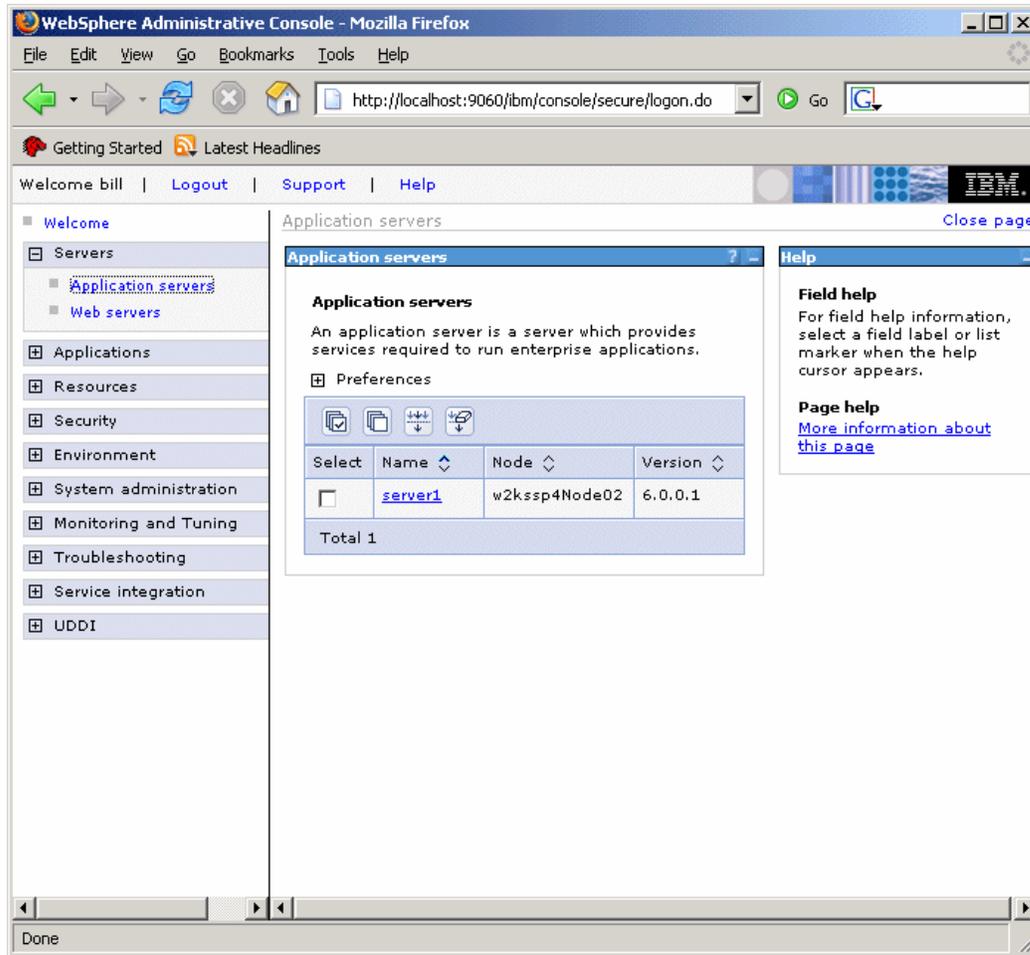
Figure 4-21   Administering servers

> b. Choose **server1** to see a properties page for the server which allows you to modify the server properties. See Figure 4-22 on page 145 for an example.
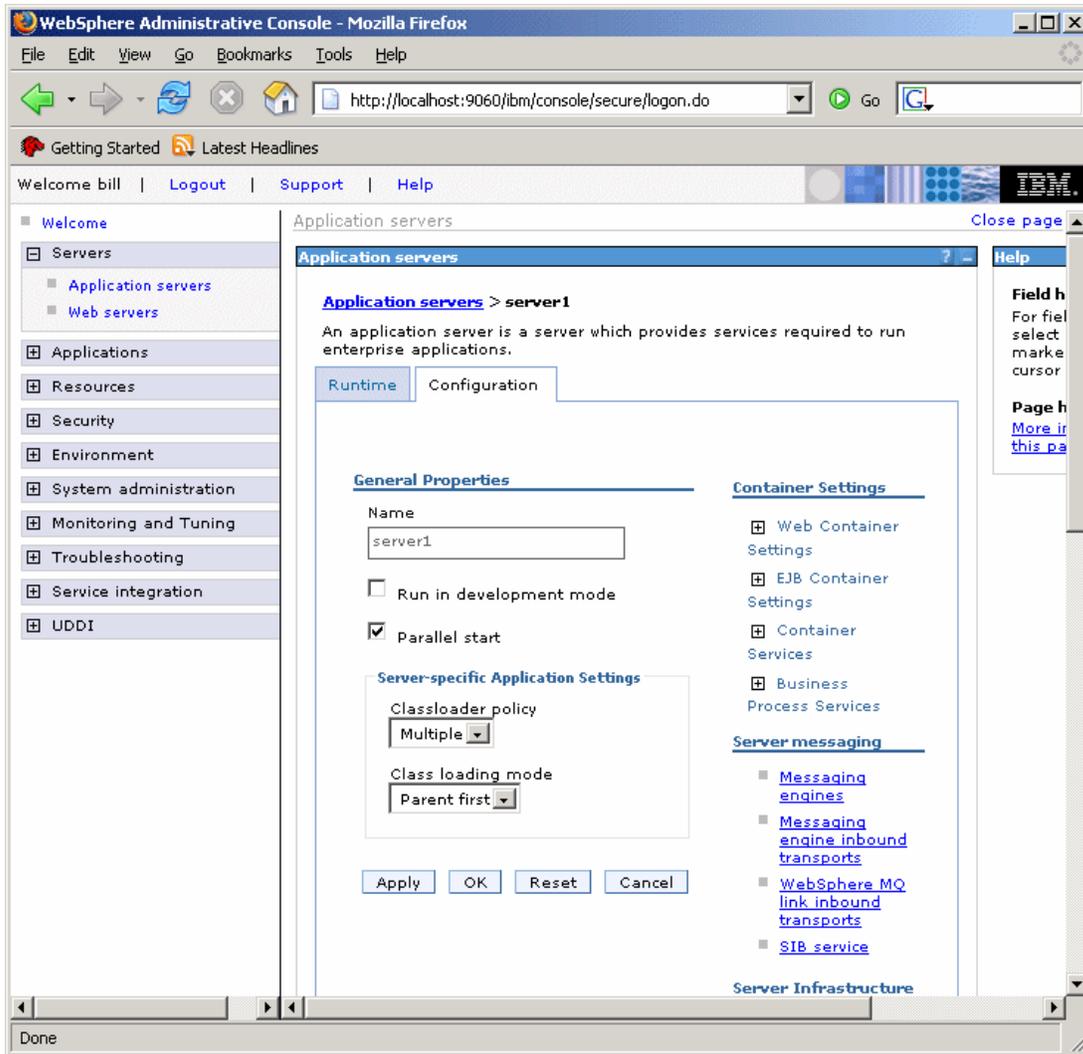
Figure 4-22   Server properties

Tip: Help is available in the WebSphere Administrative Console either for a specific field or for the page.Context sensitive hover help is also available. Figure 4-23 on page 146 shows these help alternatives.
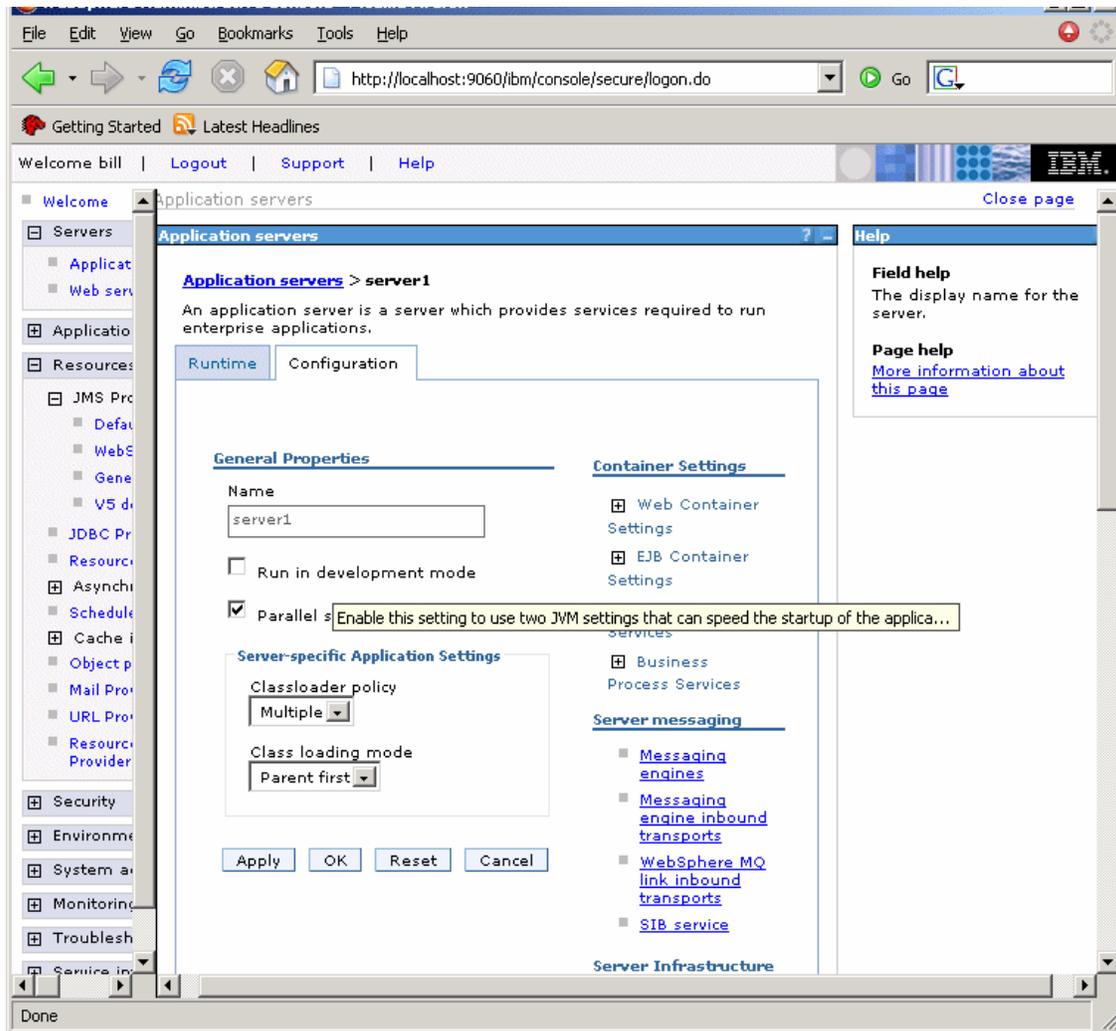
Figure 4-23   Getting help in the WebSphere Administrative Console

► Applications

This choice allows you to start and stop installed applications, You can also install new applications. To use these options choose **Applications** → **Enterprise Applications** to manage applications. This shows all the installed applications and their status, running or stopped.

With **Applications** → **Install New Application**, you can install new applications.

We see how to install a new application when we install the Sal404 application as described in 4.7, "Deploying the sample application" on page 174.

Figure 4-24 shows an example list of installed applications.
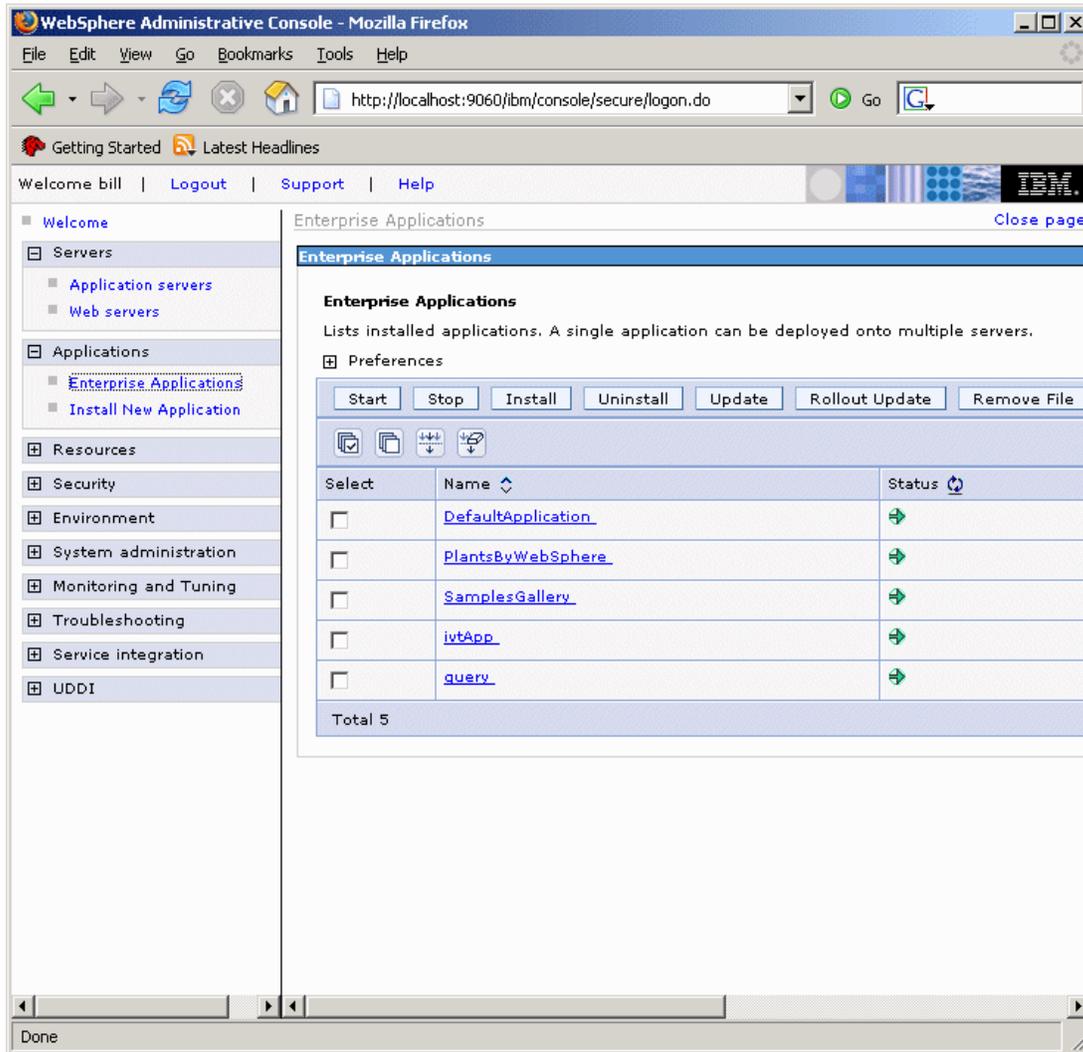
Figure 4-24   Installed applications

Click any of the applications to see the application properties. See Figure 4-25 on page 148.
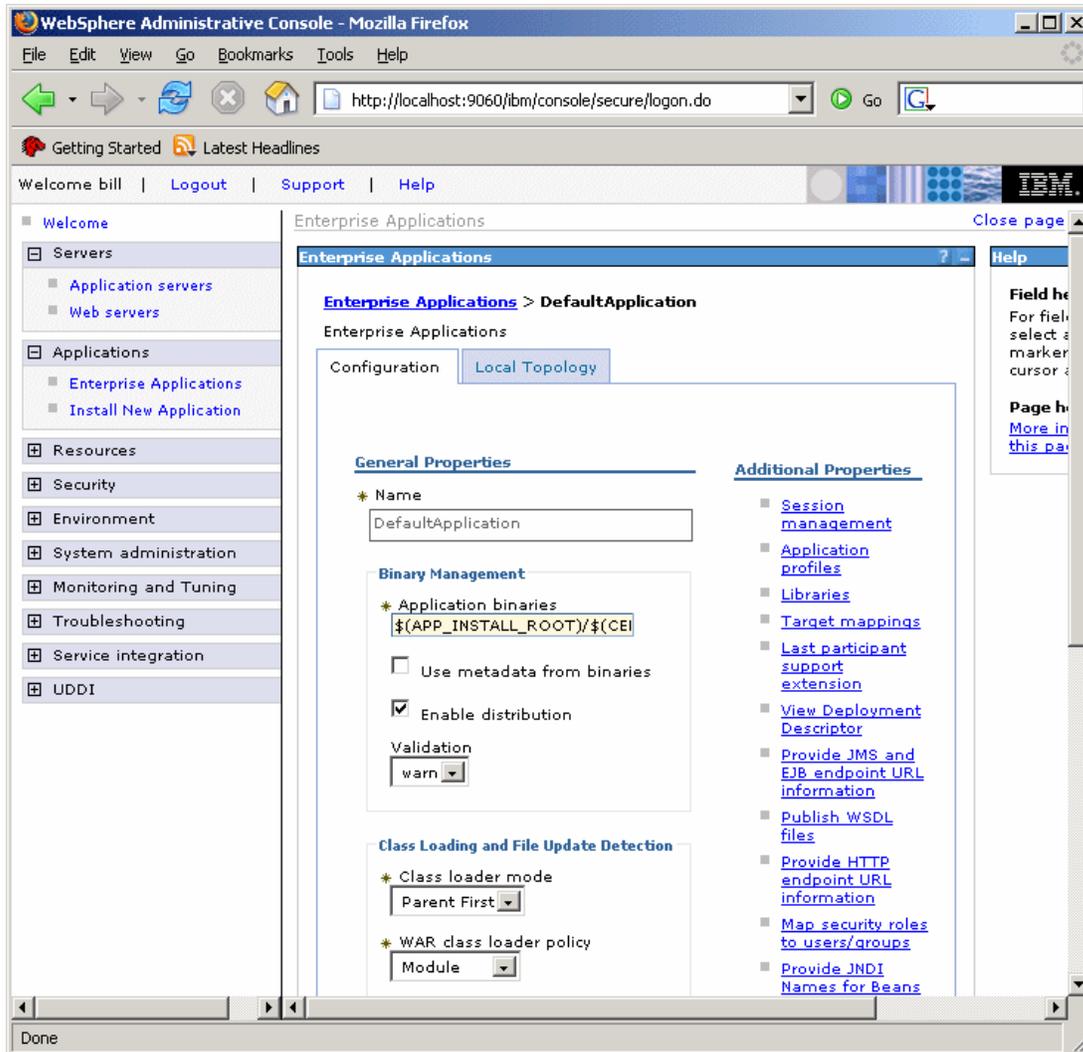
Figure 4-25   Application properties

► Resources

The Resources options as shown in Figure 4-26 on page 149 allow you create and manage resources for your applications. Such resources include JDBC providers, Mail providers, Schedulers., and so on. We show how to create a JDBC resource for the sample application in 4.7.2, "Creating the JDBC resources" on page 176. For JMS resource creation and management, see 10.6, "Setup JMS the environment" on page 408.
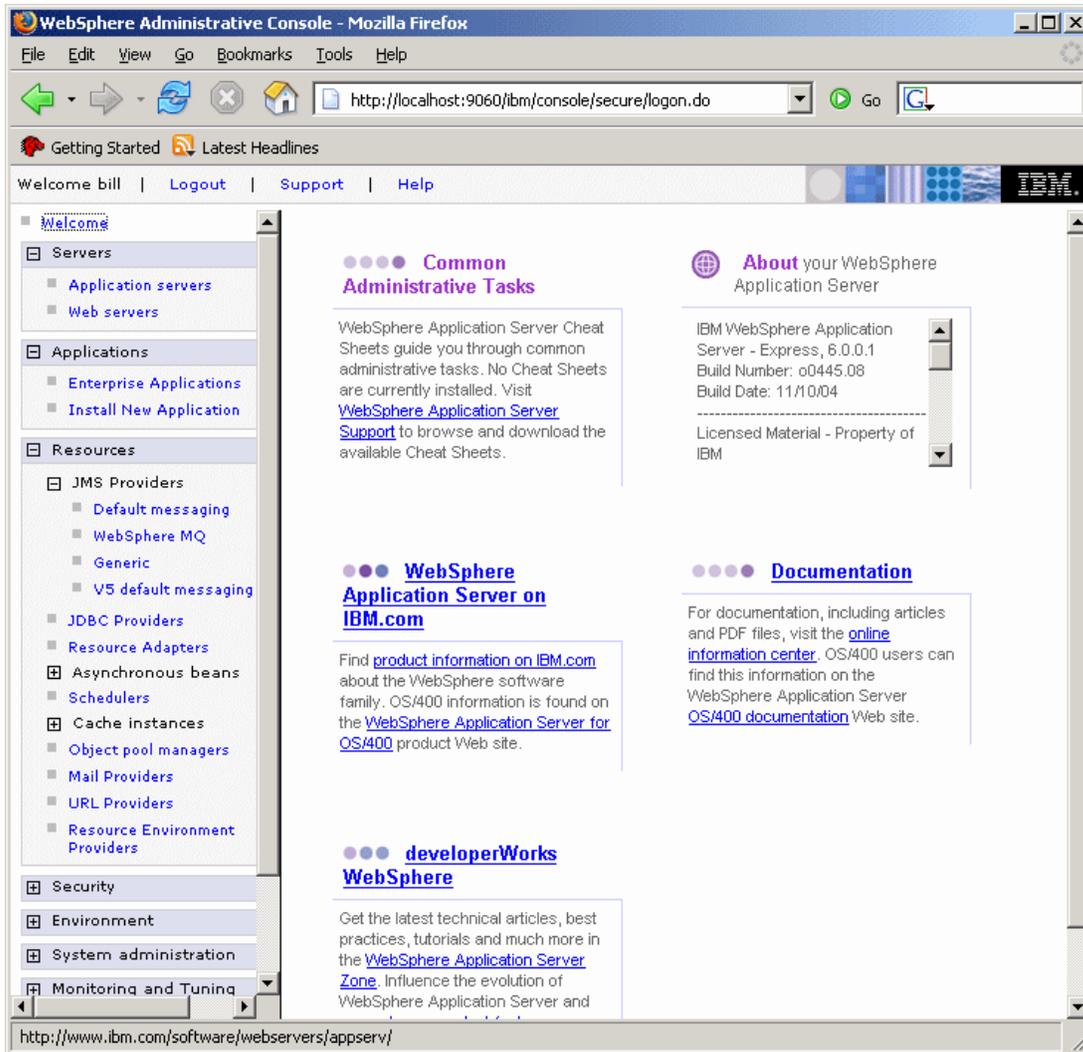
Figure 4-26   Managing resources

4.5 Installing Rational Web Developer

Rational Web Developer comes with WebSphere Application Server - Express and can be installed from the WebSphere Application Server - Express installation launch pad. If you followed the steps for installing WebSphere Application Server - Express the launch pad should still be active, otherwise run `launchpad.bat` from the installation folder.

The mains steps for installing Rational Web Developer are:

1. Select **Rational Web Developer Installation**.

2. We suggest that you review the product readme file and the installation guide(both are accessible from the Rational Web Developer installation page) before installing. In particular, check that you meet the documented system requirements for your installation platform. For Windows systems the requirements include:

   – Intel Pentium III 800MHZ processor minimum or higher
   – 1024 x 768 display minimum
   – Minimum 768 MB available RAM, 1 GB recommended
   – 3.0 GB minimum hard drive space for installing Rational Web Developer

   The official product requirements are available on the Web at:

   `http://ibm.com/software/awdtools/developer/web/sysreq/index.html`

3. Click **Launch the installation wizard for Rational Web Developer** as shown in Figure 4-27 on page 151.
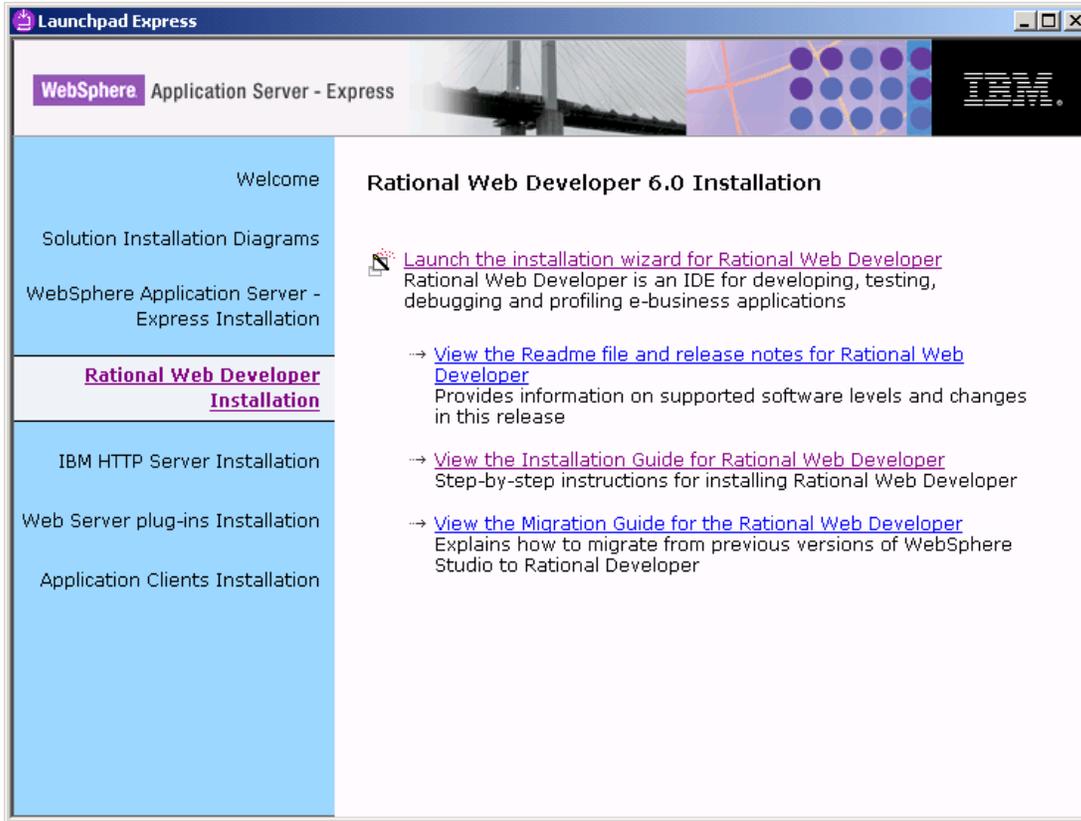
Figure 4-27   Launch Rational Web Developer install

4. You are prompted to enter the location of the first installation disk for Rational Web Developer. Because we were installing from downloaded software, we entered the location of the `disk1` folder in our install directory. See Figure 4-28 for an example. Click **OK**.
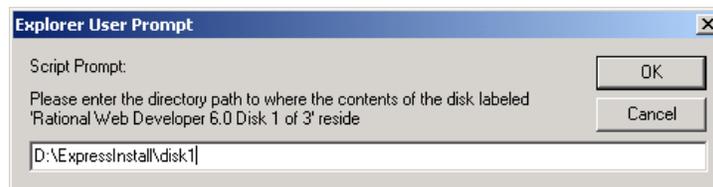
Figure 4-28   Enter location for Rational Web Developer install disk

5. When the installation wizard appears, click **Next**. See Figure 4-29.
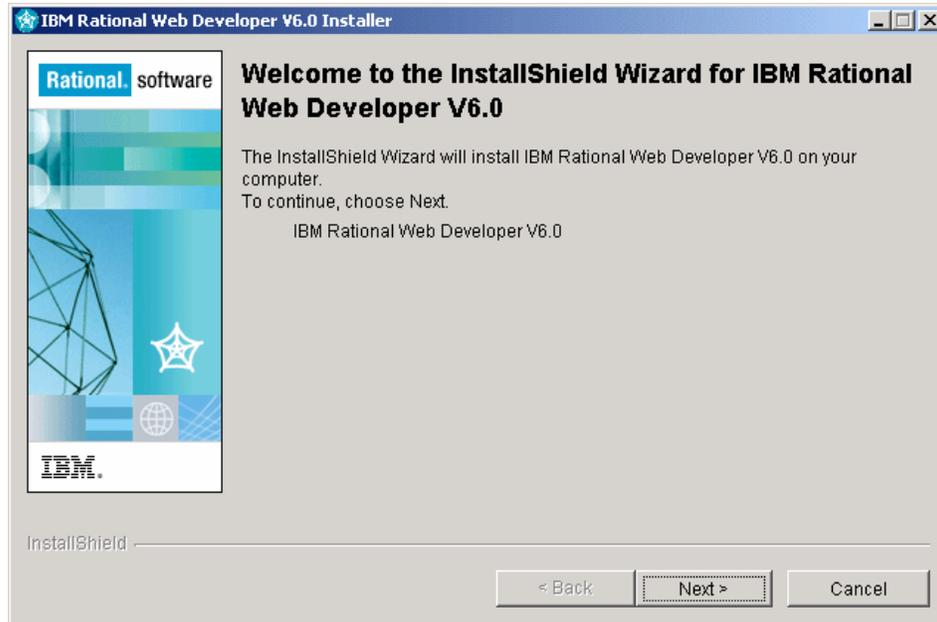
Figure 4-29   Welcome page for Rational Web Developer installation

6. Read and accept the license terms and conditions, check the **I accept the terms in the license agreement,** and click **Next**. See Figure 4-30 on page 153.
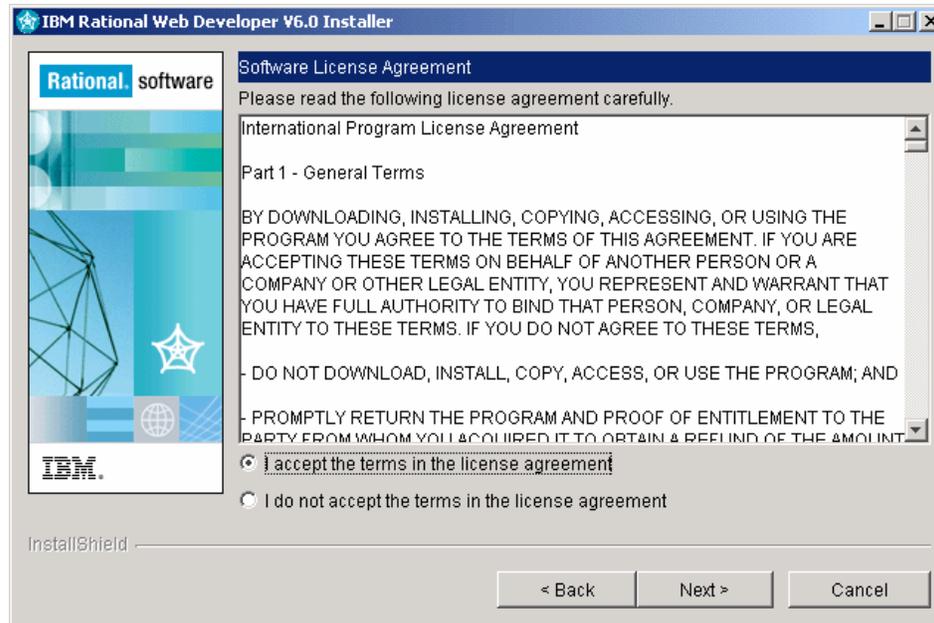
Figure 4-30   Rational Web Developer terms and conditions

7. Enter a directory name for the installation of Rational Web Developer. The default is `<drive>\Program Files\IBM\Rational\SDP\6.0`. We changed this to `<drive>\rsdp` as shown in Figure 4-31 on page 154. Click **Next**.
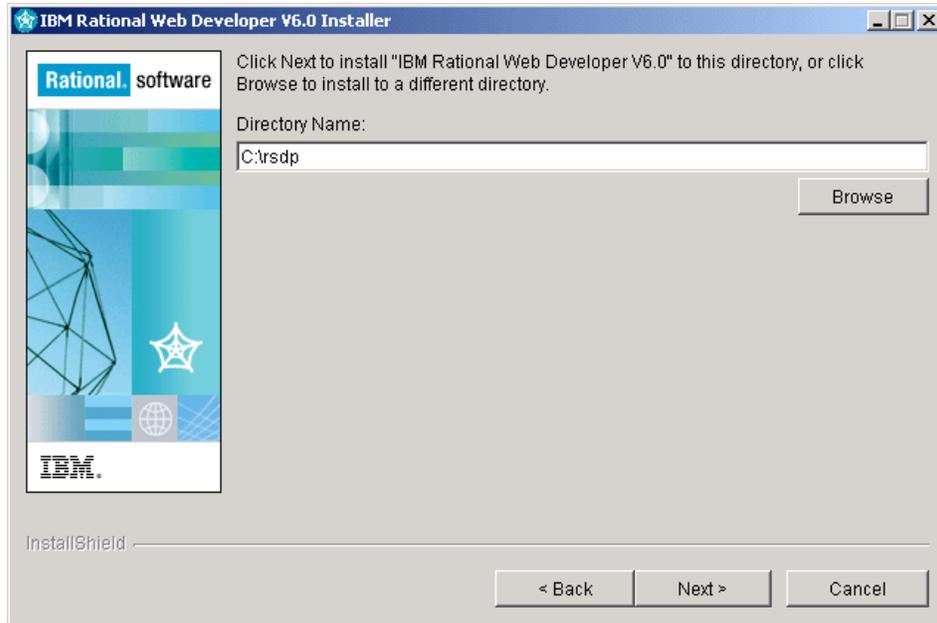
Figure 4-31   Installation directory for Rational Web Developer

8. The next page of the installation wizard allows you to chose what features to install. We did not need any of the additional features, so chose not to select them. Figure 4-32 on page 155 shows that we did choose to install the IBM WebSphere Application Server V6.0 Integrated Test Environment. This option is selected by default and assumes that you have not already installed Express Application Server on the same machine. Click **Next** to continue.

> **Note:** The WebSphere Application Server - Express installation documentation says that *IBM does not recommend installing the Express server on the same machine with Rational Web Developer. Rational Web Developer installs its own copy of the Express server for a totally functional test environment.* We found that if the Express Application Server was already installed before Rational Web Developer, then any attempt to install the IBM WebSphere Application Server V6.0 Integrated Test Environment failed.
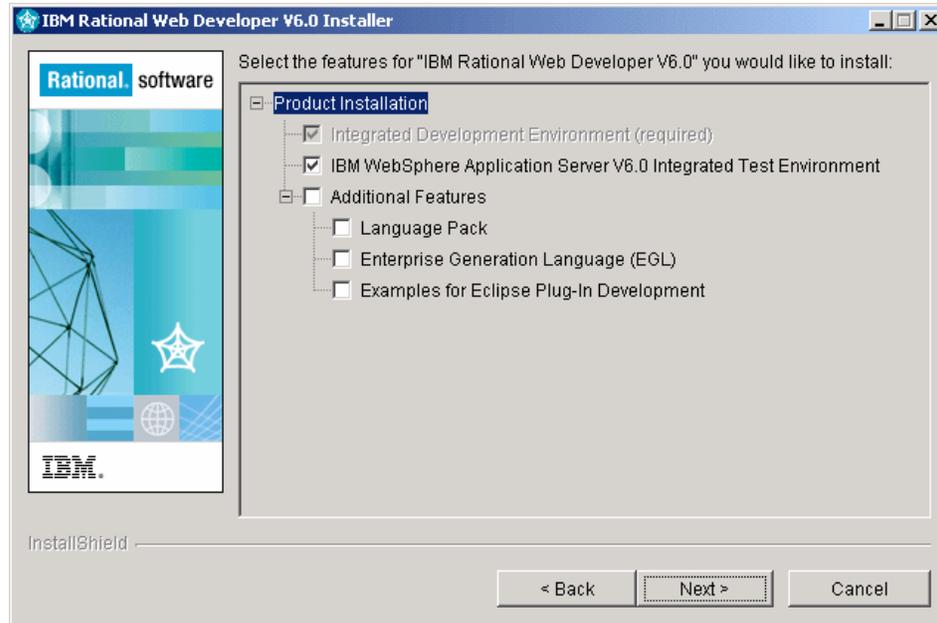
Figure 4-32   Selecting product features to install

9. Read the installation summary and click **Next**. See Figure 4-33 on page 156.
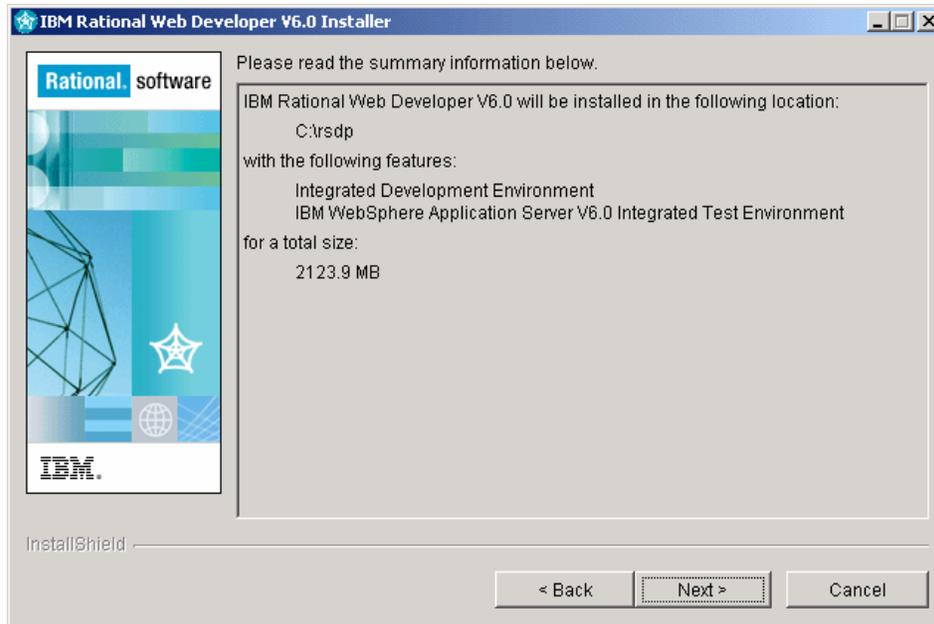
Figure 4-33   Rational Web Developer installation summary

10.Figure 4-34 on page 157 shows the results of a successful installation Click
   Next.
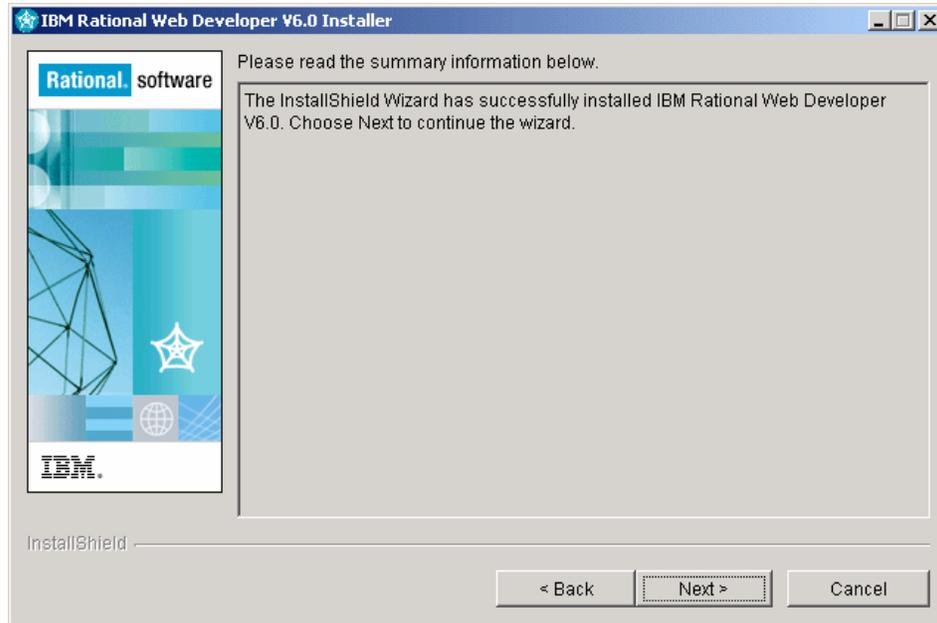
Figure 4-34   Successful installation of Rational Web Developer

11. The last page of the installation wizard is shown in Figure 4-35 on page 158. This provides information about how to update Rational Web Developer and about how to start the IDE. Click **Finish**.
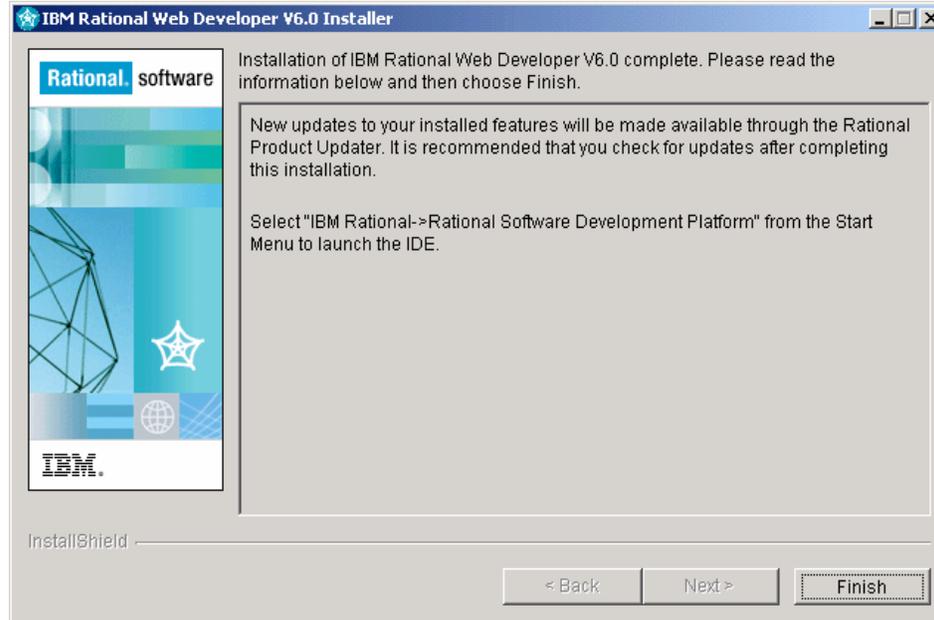
Figure 4-35   Installation complete for Rational Web Developer

4.5.1 Express Application Server and Rational Web Developer

As stated in the note about step 8 on page 154, the IBM product documentation suggests that WebSphere Application Server - Express and Rational Web Developer are not installed on the same machine, but this can be done if you need to have this capability. We found that if Rational Web Developer was installed before the Express Application Server , we were able to have both products working on one machine. The steps to do this are:

1. Install Rational Web Developer and the WebSphere Application Server V6.0 Integrated Test Environment as described in 4.5, "Installing Rational Web Developer" on page 149.

2. Launch the WebSphere Application Server - Express installation as described in 4.3, "Installing WebSphere Application Server - Express" on page 121.

3. The installation wizard detects the installed WebSphere Application Server V6.0 Integrated Test Environment as though it is another installation of WebSphere Application Server. The dialog box shown in Figure 4-36 on page 159 is displayed and you can choose to install another copy of the Express Application Server or to add features to the current install.
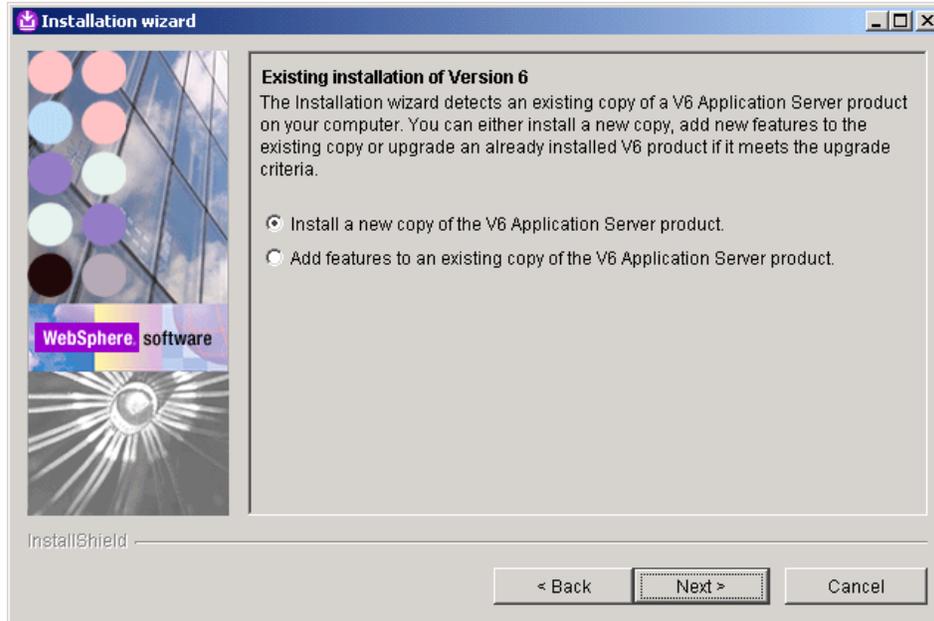
Figure 4-36   Duplicate installation detected

4. Select **Install a new copy of the V6 Application Server product** and click **Next**.

5. Read the information about the existing installation as shown in Figure 4-37 on page 160 and then click **Next**.
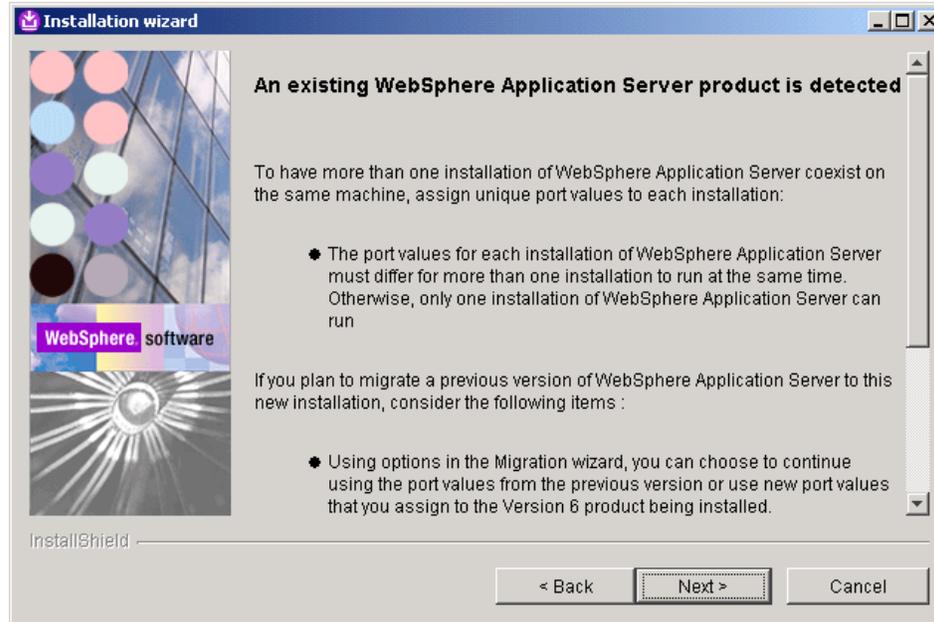
Figure 4-37   Existing Express Application Server detected

6. The next page of the installation wizard shows you the ports that will be used by the Express Application Server . See Figure 4-38 on page 161. The wizard will default all the port values to use different numbers than those used by the already installed Integrated Test Environment.
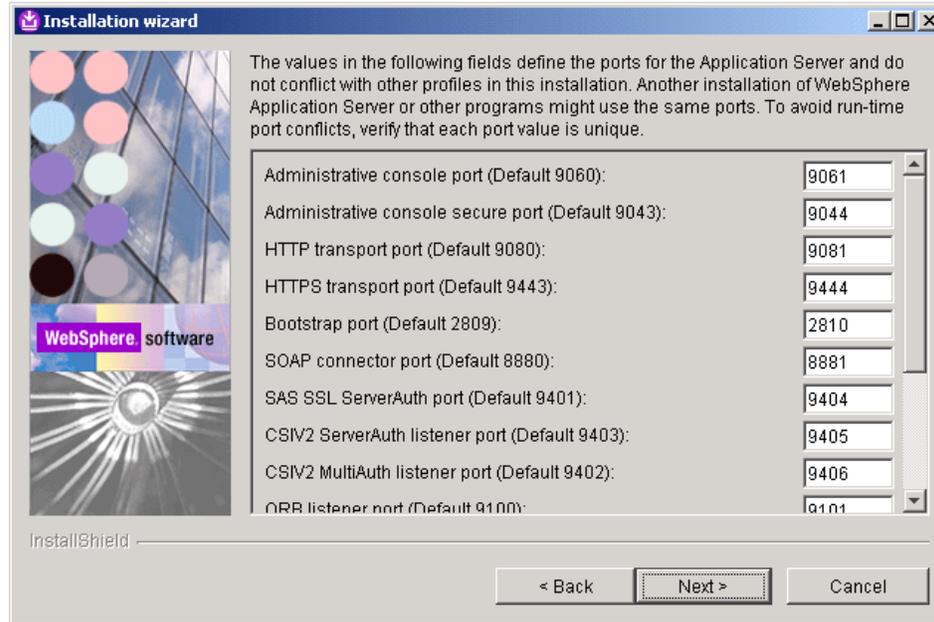
Figure 4-38   Port assignments for the Express Application Server

7. We do not need to run the Integrated Test Environment at the same time as the Express Application Server , so we changed the suggested port assignments back to their default values.

8. The rest of the installation steps are unchanged from those described in 4.3, "Installing WebSphere Application Server - Express" on page 121.

4.6 Installing DB2

Our redbook sample application requires a working DB2 database. WebSphere Application Server - Express V6 ships with IBM DB2 Universal Database Express Edition V8.2, so we installed this database on our redbook machines. We show the steps we used for our installation. Remember that we are not setting up a production DB2 system, so we do not describe the planning and installation necessary for a robust production environment.

The steps to follow for our simple development and test setup are:

1. DB2 setup is not started from the WebSphere Application Server - Express launchpad, but by running the separate DB2 launchpad. Figure 4-39 on page 162 shows the DB2 launchpad.
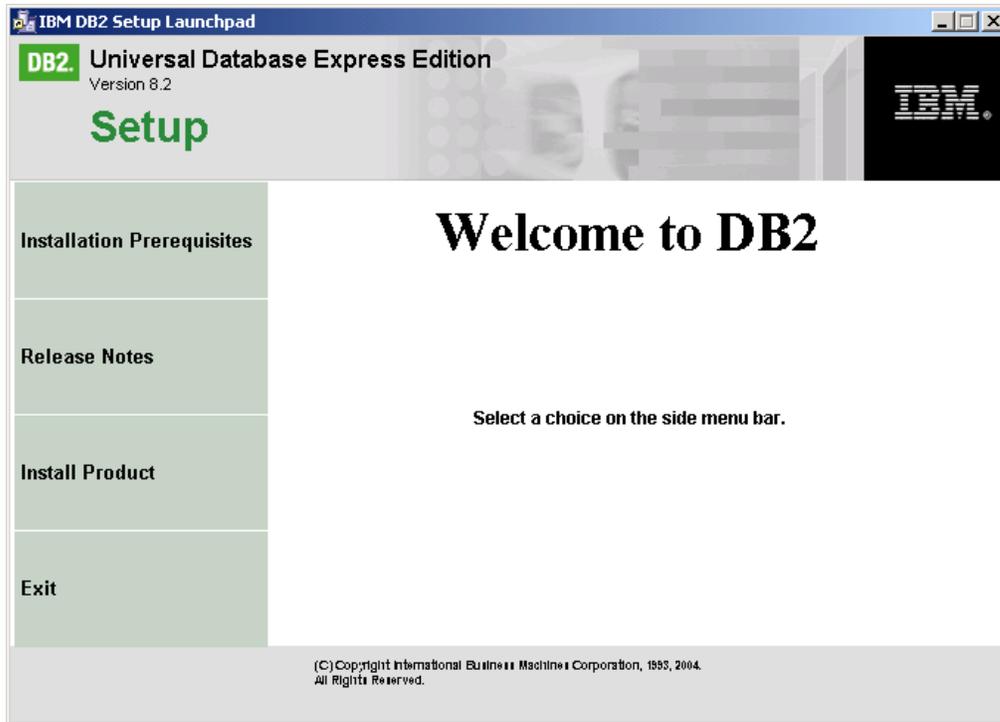
Figure 4-39   DB2 setup launchpad

    2. Check that you meet the installation prerequisites. You can do this by selecting **Installation Prerequisites** on the DB2 launchpad as shown in Figure 4-40 on page 163.
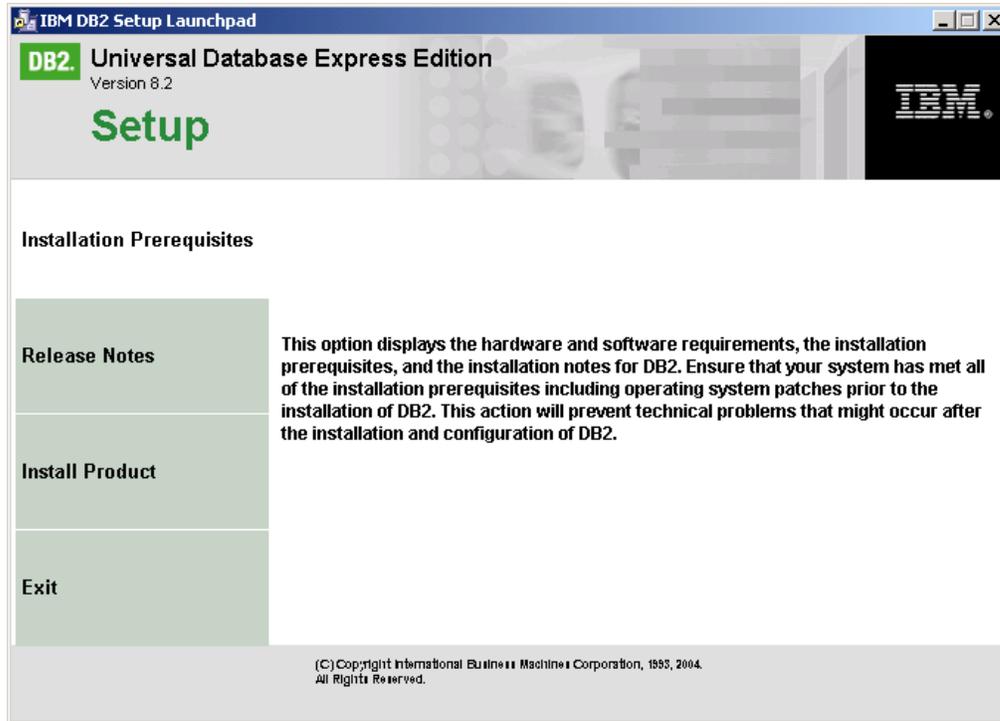
Figure 4-40   DB2 install prerequisite checks

3. Select **Install Product** from the DB2 launchpad and chose to install DB2
    UDB Express as shown in Figure 4-41 on page 164. Click **Next**.
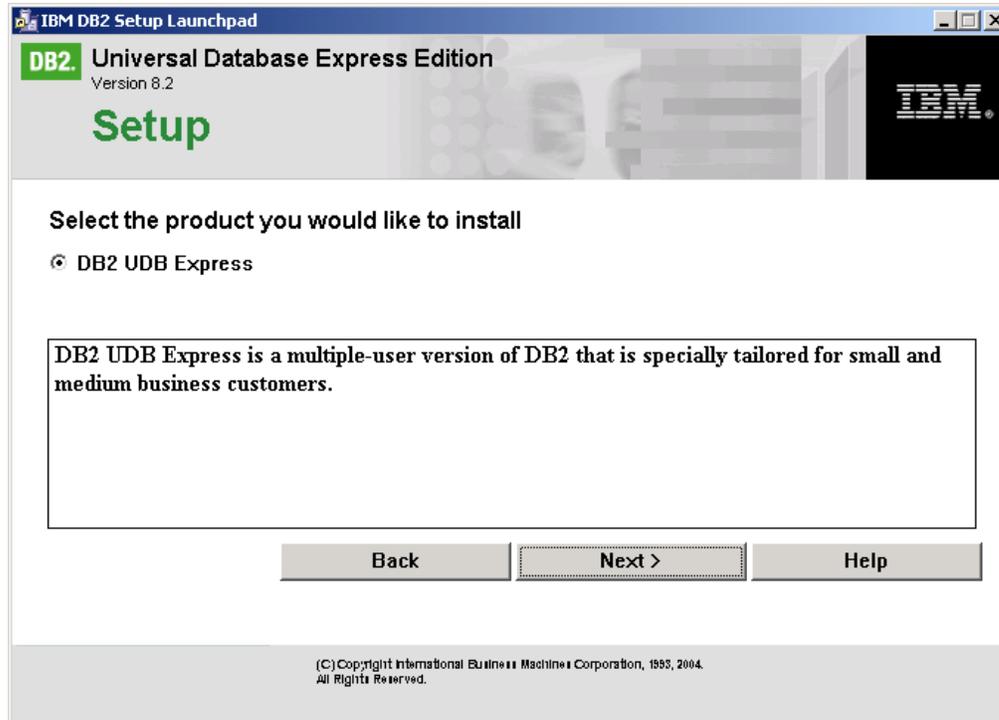
Figure 4-41   Install UDB

4. On the welcome page of the setup wizard click **Next**.

5. Accept the licence agreement and click **Next**.

6. Select the installation type. We chose **Custom** as shown in Figure 4-42 on page 165. Click **Next**.
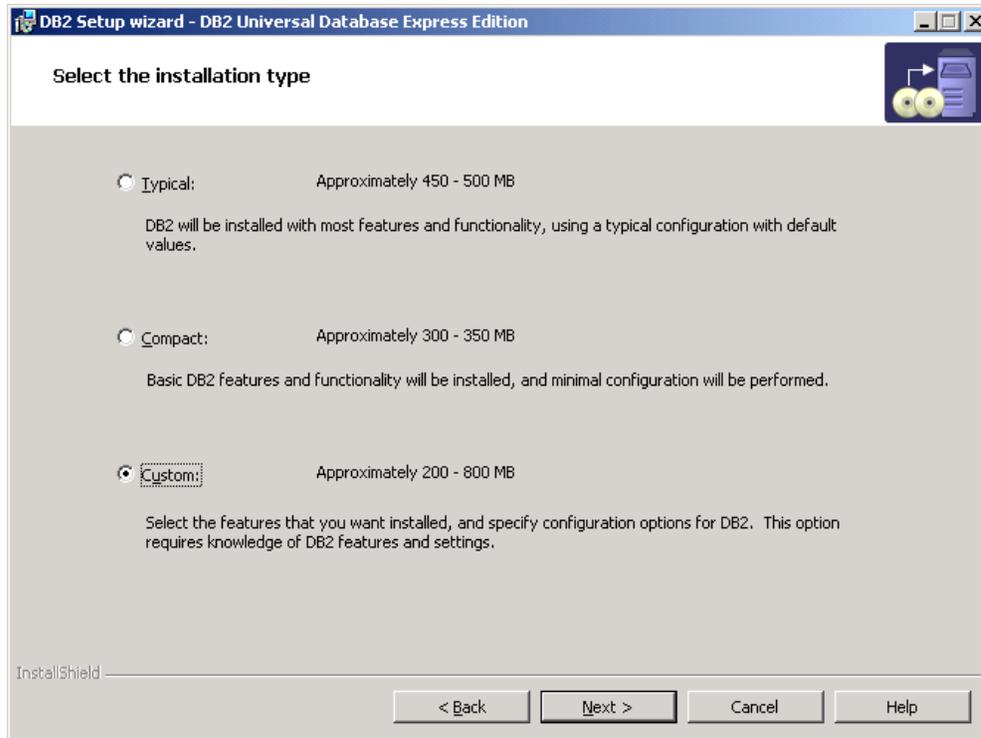
Figure 4-42   Select DB2 installation type

7. The next page shown in Figure 4-43 on page 166 allows you to select features to be installed and to change the installation drive and directory.

   We changed the default installation directory from the default of C:\Program Files\IBM\SQLLIB to `C\SQLLIB`. It is not necessary to change any of the features installed, but we chose to remove features that supported protocols other than TCP/IP and also removed some features we did not use such as support and samples for warehouses and business intelligence. Click **Next**.
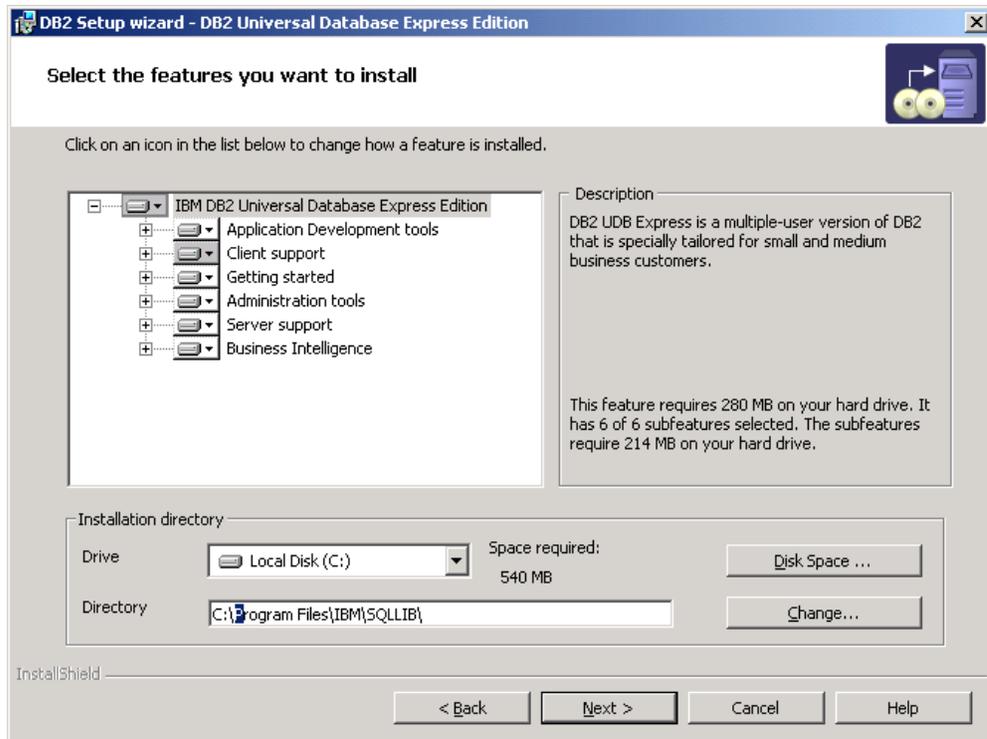
Figure 4-43   Chose DB2 features

8. We did not need to install any extra language support so we click **Next**. See Figure 4-44 on page 167.
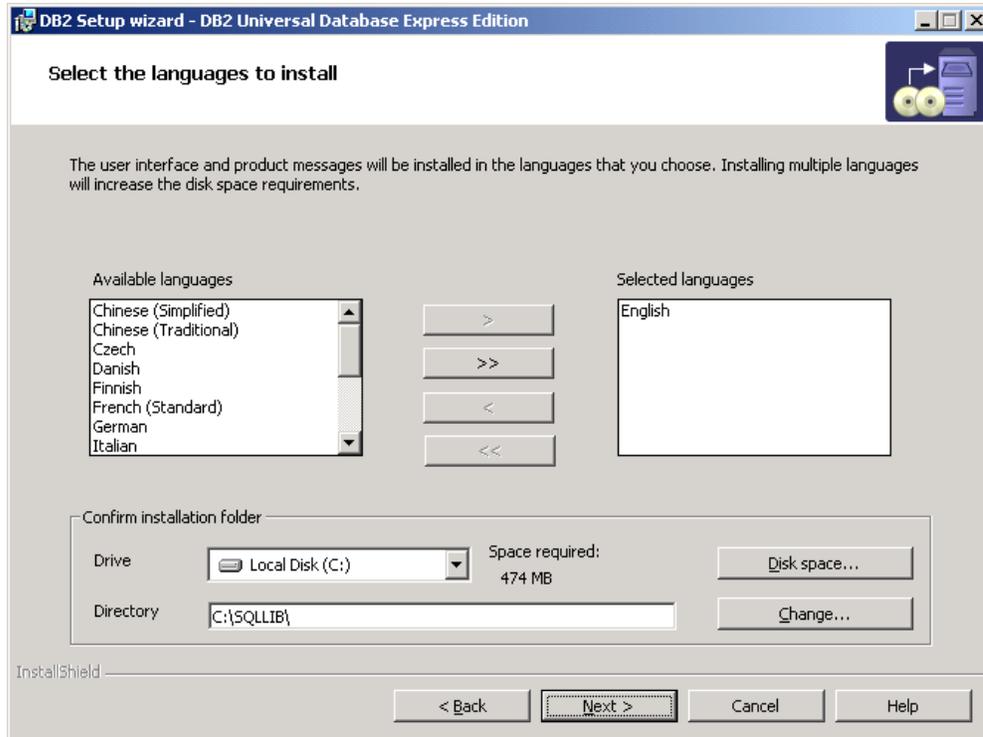
Figure 4-44   DB2 languages supported

9. We chose not to install the DB2 Information Center on our machine, so you can select to access the DB2 Information Center On the IBM Web site and click **Next**. See Figure 4-44 on page 167.
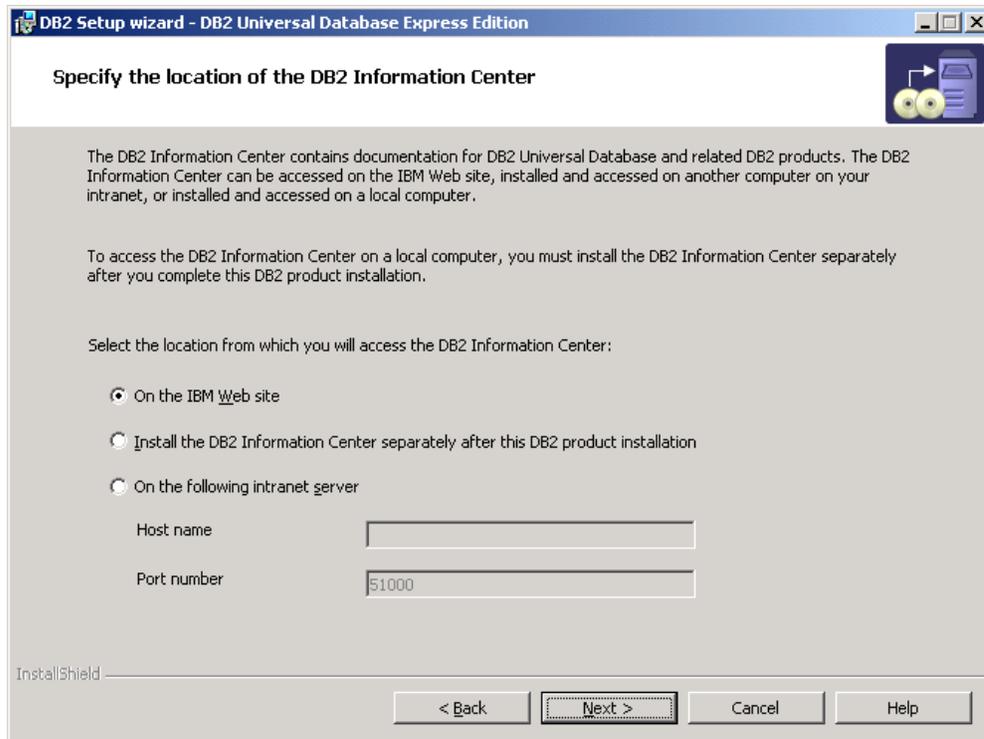
Figure 4-45   Location for the DB2 Information Center

10. Chose the same user ID for all DB2 services and click **Next**. We used
    `db2admin` for our user ID as shown in Figure 4-46 on page 169.
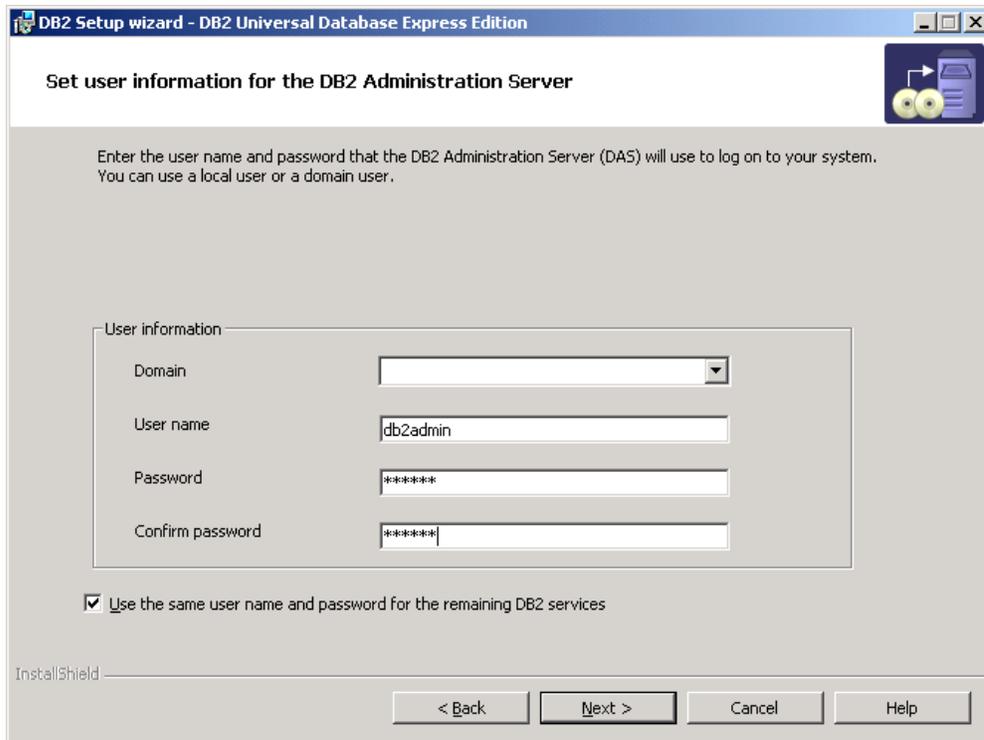
Figure 4-46   DB2 user IDs

11.Chose a local contact list and no notification as shown in Figure 4-47 on page 170. Click **Next**.
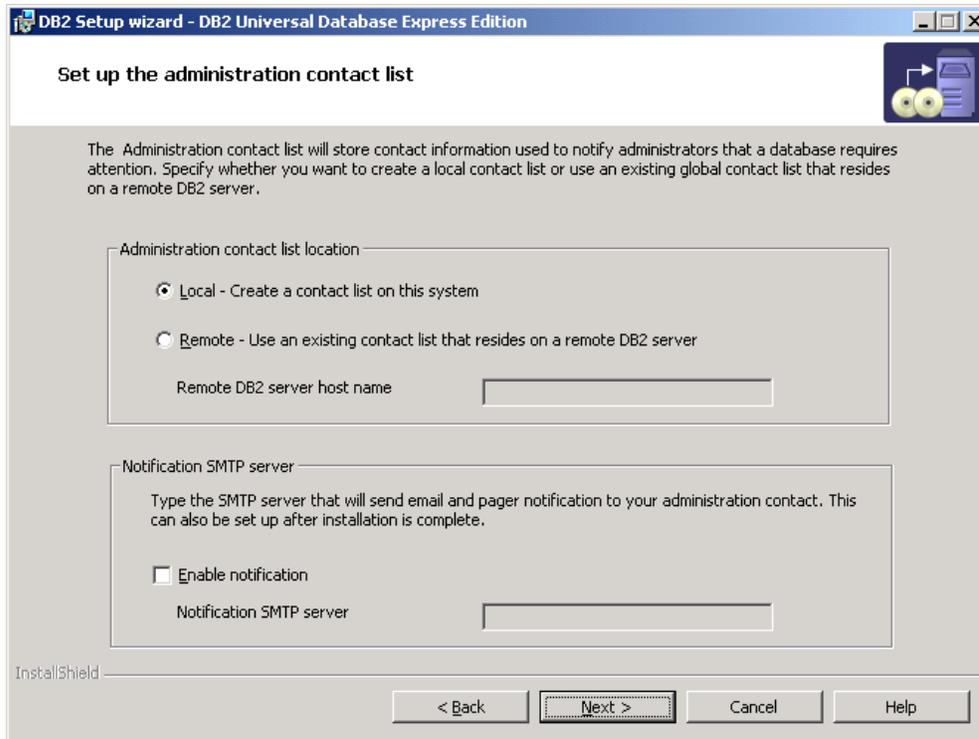
Figure 4-47   DB2administration contact lists and notification

12. A warning dialog box is displayed advising a notification SMTP server has not been specified. Click **OK** to continue. See Figure 4-48 on page 171.
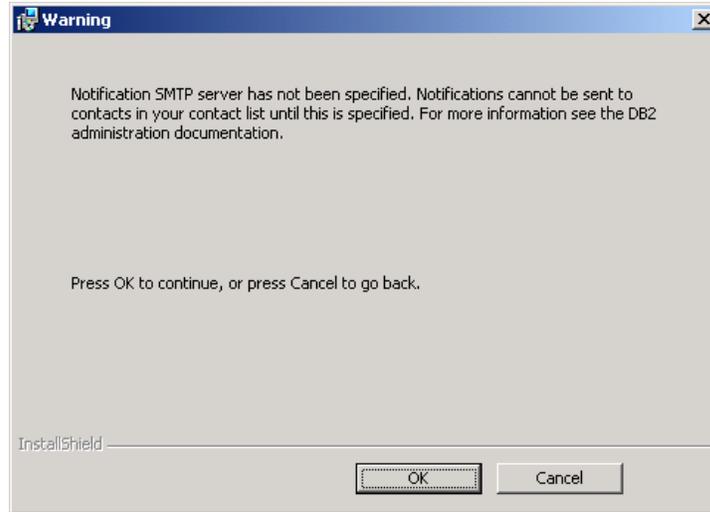
Figure 4-48   Notification SMTP warning

13. Allow the installation to configure a default DB2 instance and click **Next**. See Figure 4-49 on page 172.
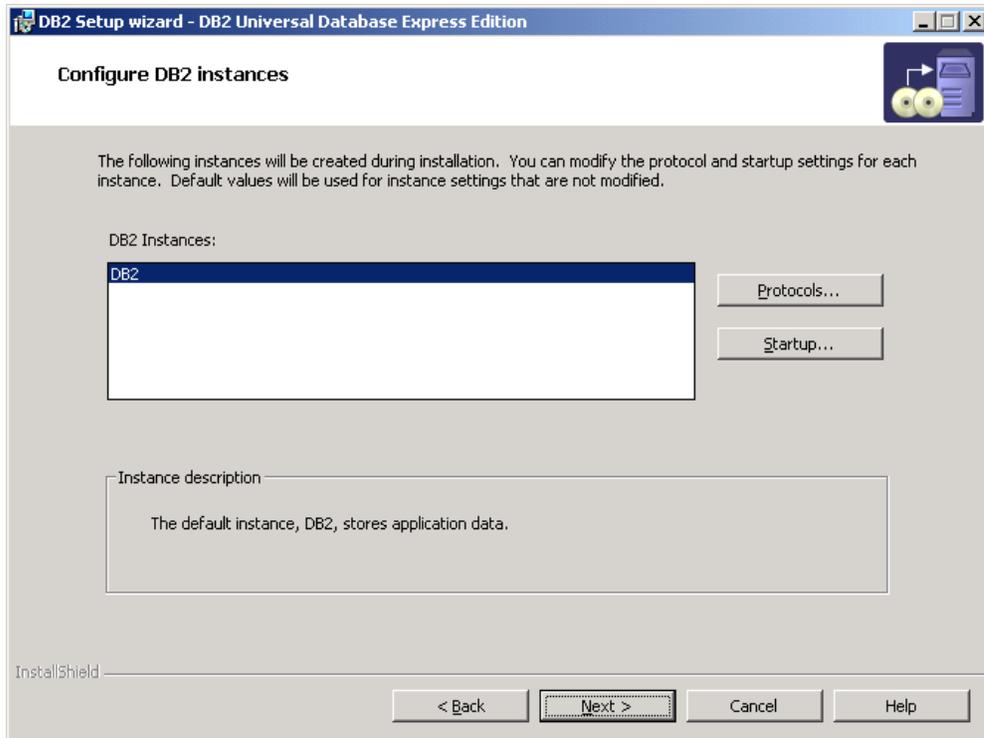
Figure 4-49   Configure DB2 instance

14. Select **Do not prepare the DB2 tools catalog on this computer** and click **Next**.

15. When asked whether to set up an administration contact select **Defer the task until after installation is complete** and click **Next**.

16. It is not necessary to enable operating system security fro DB2 objects. Click **Next** as shown in Figure 4-50 on page 173.
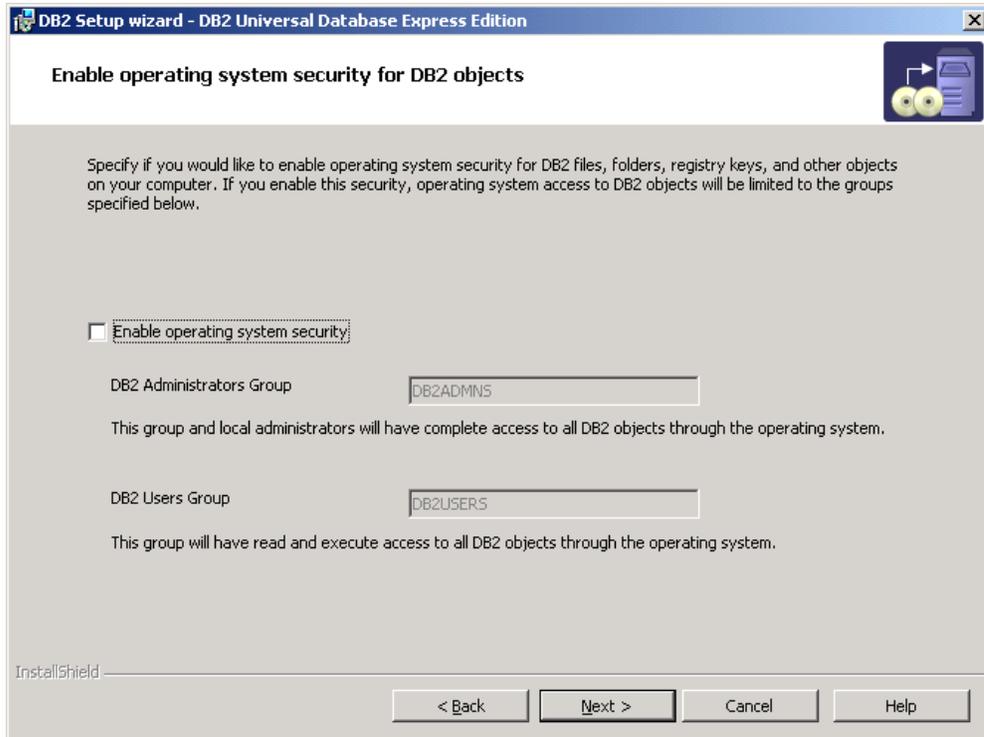
Figure 4-50   Security for DB2 objects

17. Review the installation settings and click **Install** to begin copying files. See Figure 4-51 on page 174.
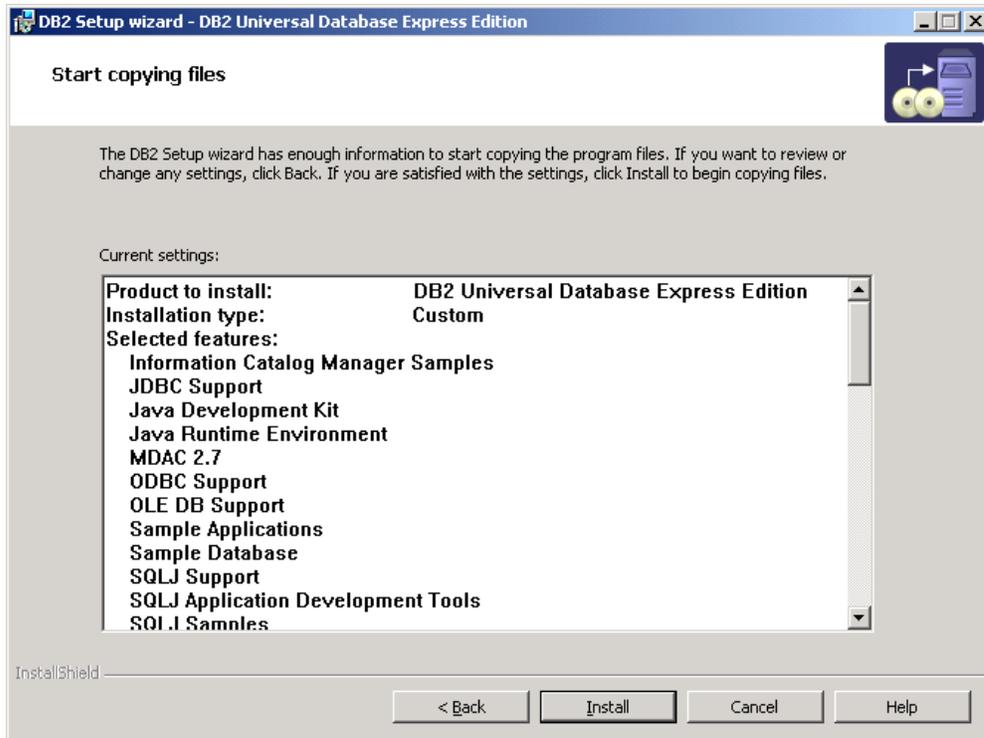
Figure 4-51   DB2 setup summary

> 18.When setup is complete click **Finish**.

4.7 Deploying the sample application

This section describes the steps needed to install the sample application. We assume here that you have obtained our redbook additional material, including includes the sample EAR files and support files.

For details about obtaining the redbook sample material see Appendix A, "Additional material" on page 617.

4.7.1 Running the sample database script

The database script creates the sample application database. This script is specifically for DB2 on Windows although with some modification it would work on other database platforms. The steps to create the sample database on the Windows platform are:

1. Open a DB2 command prompt. Select **Start** → **Programs** → **IBM DB2** → **Command Line Tools** → **Command Window.**

2. Change the current directory to where you unzipped the database creation script.

3. Enter the command **db2 -tvf DBSCRIPT.SQL**

   This command runs the database script and you are prompted to enter the password for your database administrator password.

   If you want to use a database creation log, change the command you enter to **db2 -tvf DBSCRIPT.SQL > log.txt.** This runs the database script as well as creating a log file that you can look through to see if any SQL statement failed. You will first have to modify the database script by adding your database administrators password to the command file. To do this, open the script and locate the line that reads:

   ```
   CONNECT TO SAL404R USER DB2ADMIN;
   ```

   Add the your password the end of the line. For example enter:

   ```
   CONNECT TO SAL404R USER DB2ADMIN USING <password>;
   ```

   Where <password> is the password for the user db2admin.

4. In order to verify that the SAL404R database has been successfully created, enter the following command in the DB2 command window:

   ```
   db2 list db directory
   ```

   This lists all DB2 databases that have been created and reside on your machine. The result should be similar to the output shown in Example 4-1.

Example 4-1   Verifying the SAL404R database creation

```
S:\material\database>db2 list db directory

 System Database Directory

 Number of entries in the directory = 1

Database 1 entry:

 Database alias                       = SAL404R
 Database name                        = SAL404R
 Database drive                       = C:\DB2
 Database release level               = a.00
 Comment                              = SAL404R Information System
 Directory entry type                 = Indirect
 Catalog database partition number    = 0
 Alternate server hostname            =
 Alternate server port number         =
```

4.7.2 Creating the JDBC resources

The JDBC resources represent the application database for our Sal404 application, including the relevant login details. In this section we create the required database resources needed for our sample application.

Create an authentication alias

When we connect to our sample database from Express Application Server , we want to provide appropriate user logon credentials. To do this we need to create an authentication alias as follows:

1. Using the WebSphere Administrative Console choose **Security** → **Global security.**

2. Scroll to the right of the security page and under the **Authentication** heading choose **JAAS Configuration** and then **J2C Authentication data**. See Figure 4-52 on page 177.
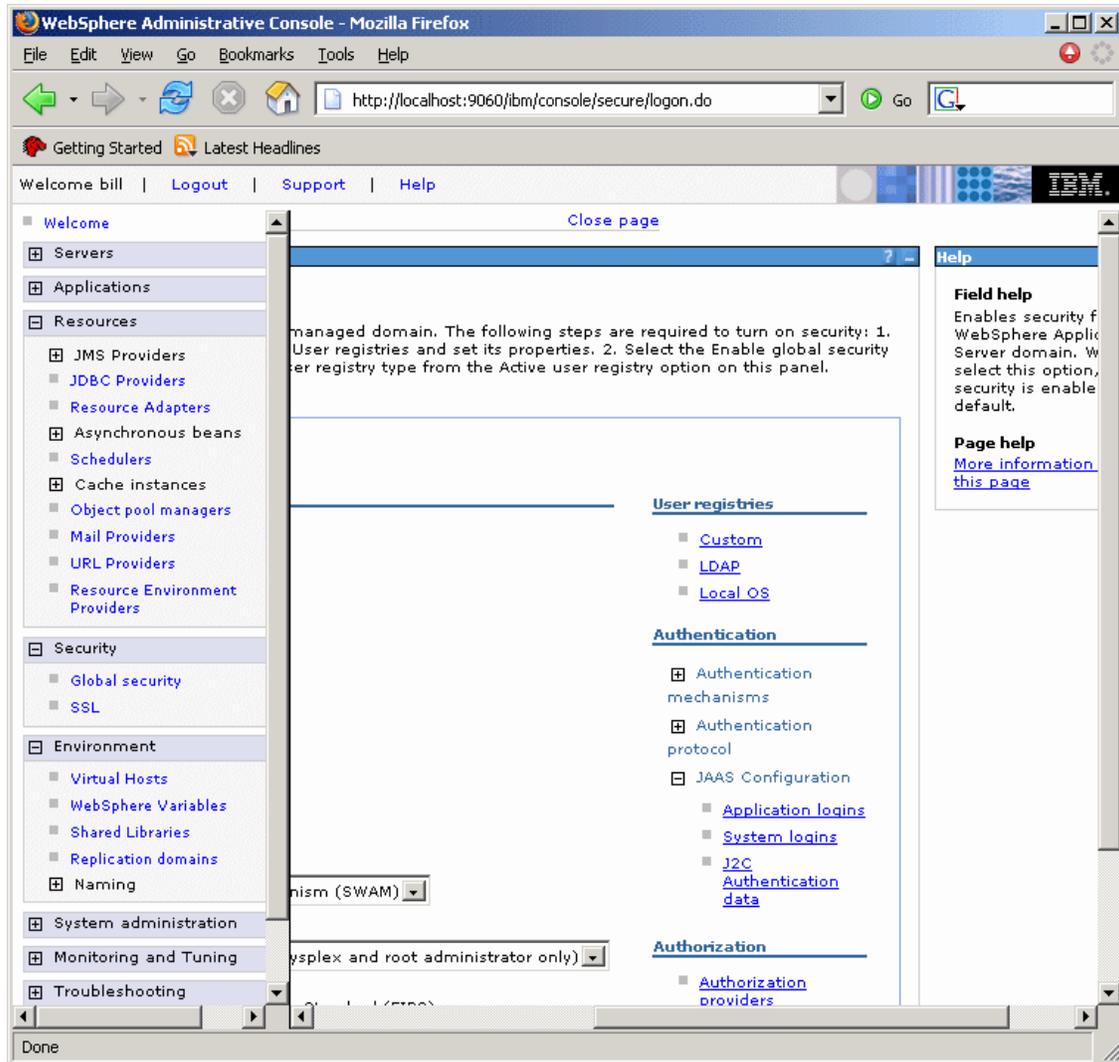
Figure 4-52   J2C Authentication data

3. Click **New** and enter the following values as shown in Figure 4-53 on page 178.

   – Alias: `db2adminAlias`
   – User ID: `db2admin`
   – Password: your db2admin password
   – Description: Any optional value you choose
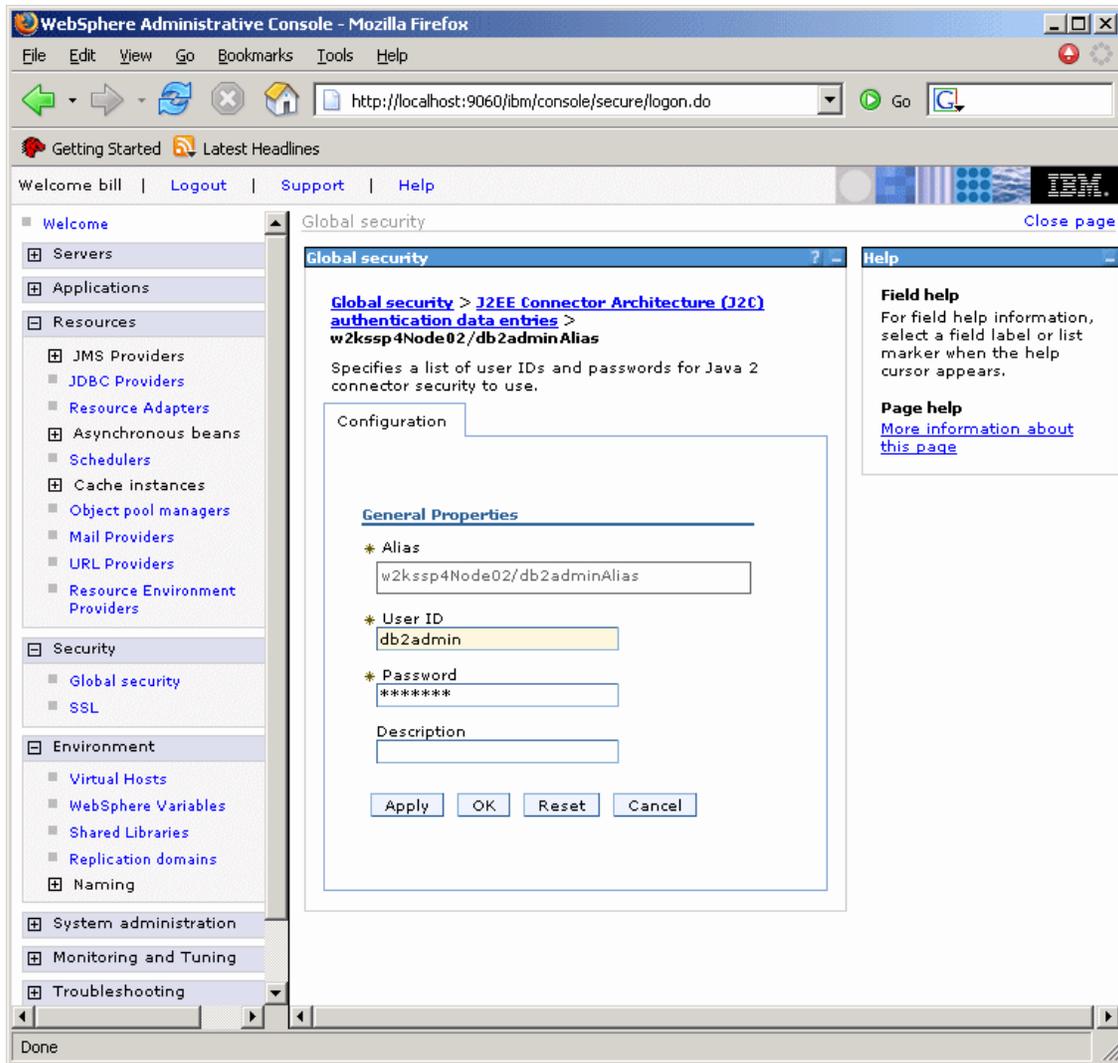
Figure 4-53   Create alias for db2admin

4. Click **OK** to create the new alias.

5. Save the administration changes to the master configuration.

Create a JDBC provider

To connect to our database manager we need to define a JDBC provider to Express Application Server . The steps are:

1. Using the WebSphere Administrative Console, choose **Resources** → **JDBC Providers.**

2. We want the resource to be defined at a server level and not using node or cell scope. Select **Server : server1** and click **Apply** as shown in Figure 4-54.
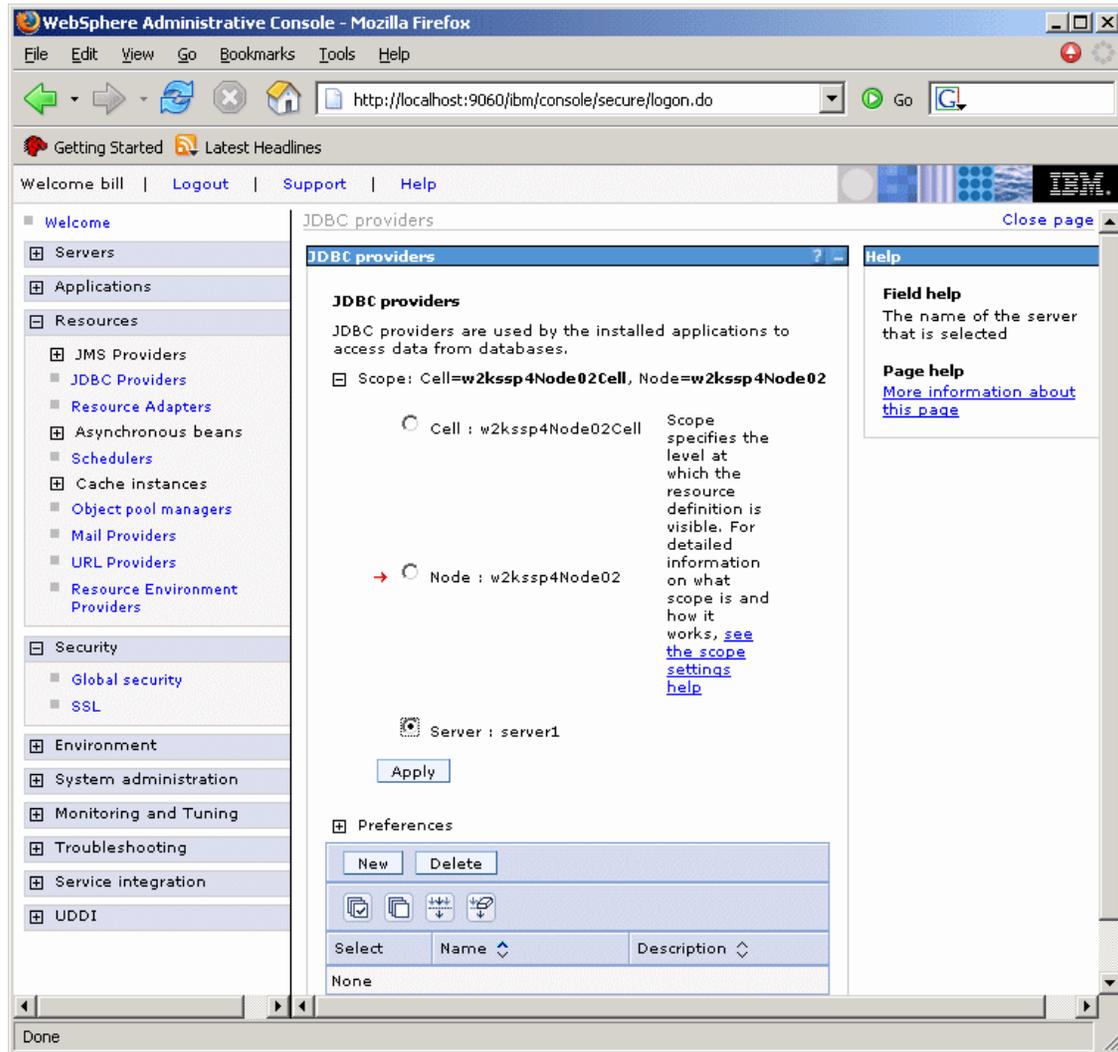
Figure 4-54   Setting server scope for resource definition

3. Click **New** to create a new JDBC provider

4. Choose the following values for the General Properties of the provider and click **Next**. See Figure 4-55 on page 180.

– Step 1: **DB2**
– Step 2: **DB2 Legacy CLI-based Type 2 JDBC Driver**
– Step 3: **Connection pool data source**
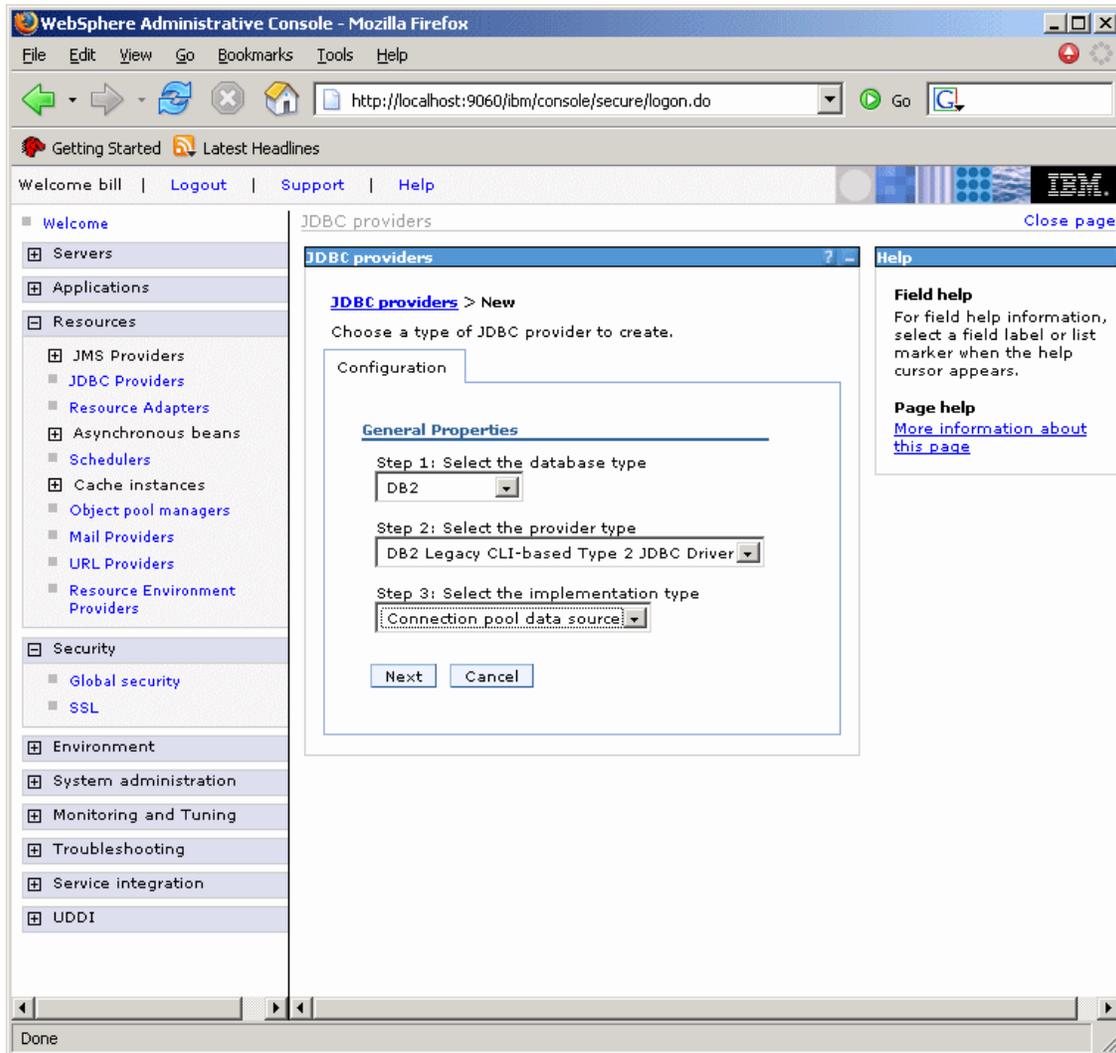
Figure 4-55   General properties for the new JDBC provider

5. Click **OK** and save the changes.

Create a JDBC DataSource

Using our new JDBC provider, we now need to create a JDBC DataSource that allows us to connect to the SAL404R database. The steps are:

1. Using the WebSphere Administrative Console choose **Resources** → **JDBC Providers** and select the new JDBC provider **D.B2 Legacy CLI-based Type 2 JDBC Driver**.

2. Scroll to the right of the page and under the **Additional Properties** heading choose **Data sources** as shown in Figure 4-56.
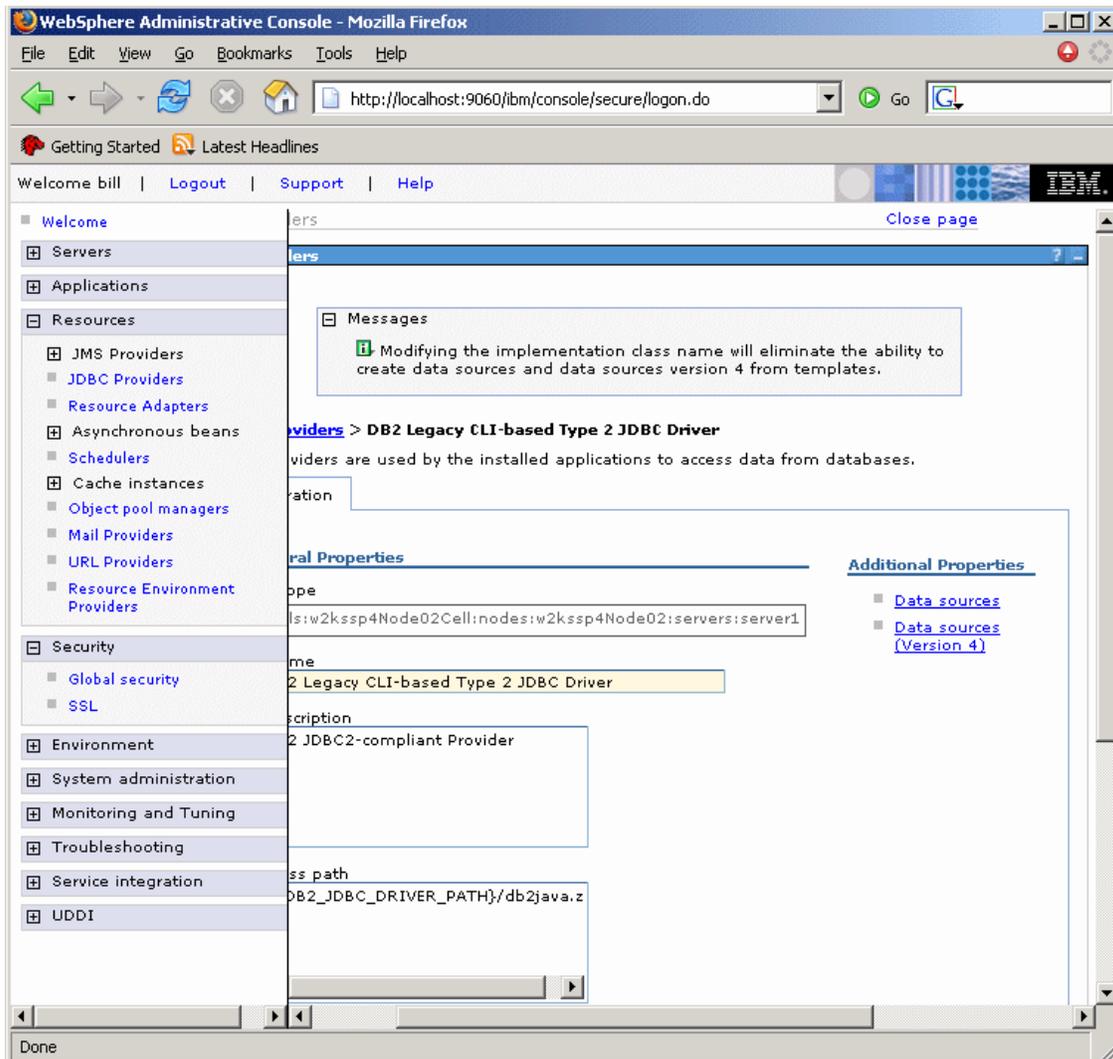
Figure 4-56   Create a DataSource

3. Click **New** and provide the following values for our new DataSource:

   – Name: `sal404`

– JNDI name: `jdbc\sal404`
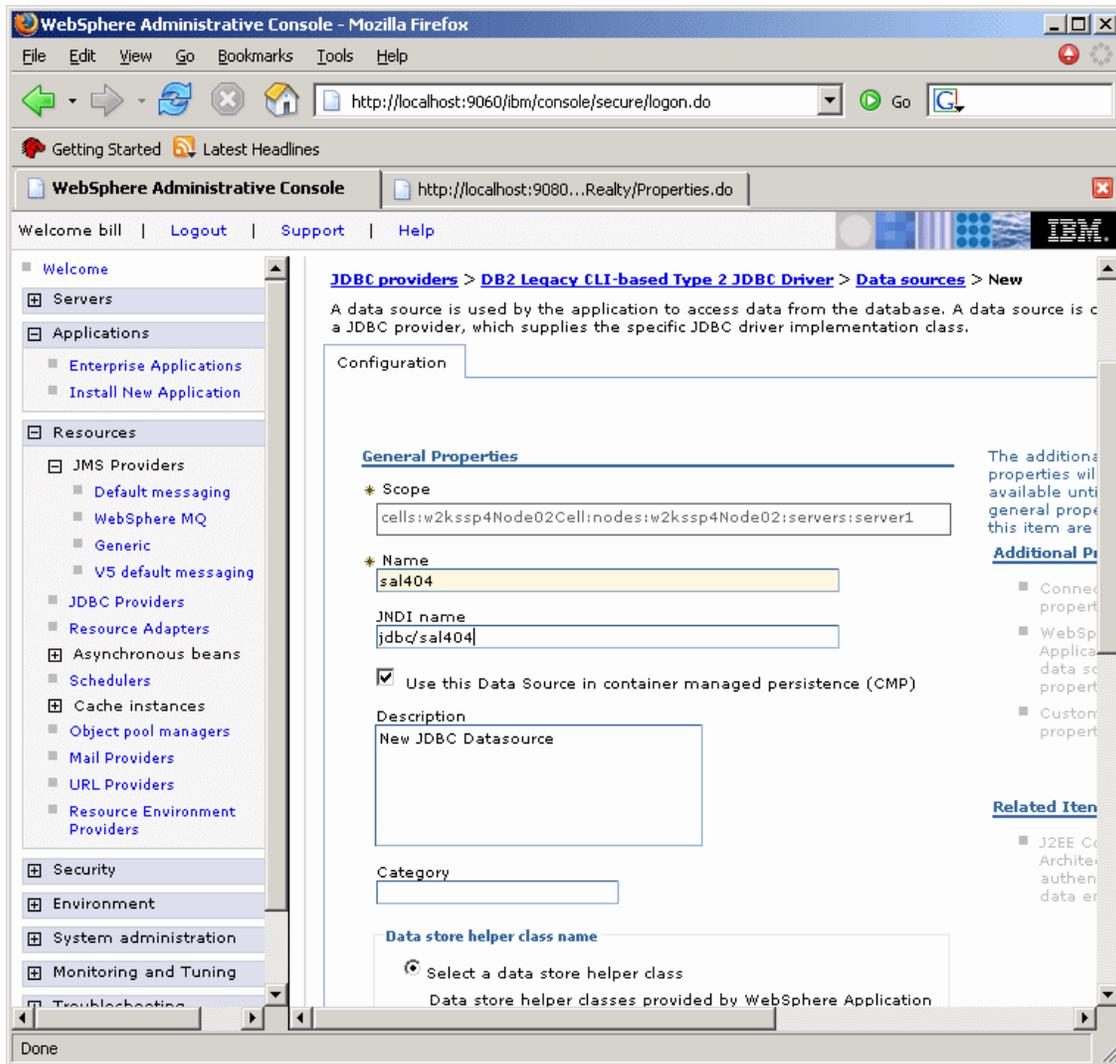
See Figure 4-57.

Figure 4-57   DataSource properties: Part 1

– Component-managed authentication alias: **db2adminAlias**
– Database name: `sal404r`

See Figure 4-58 on page 183.
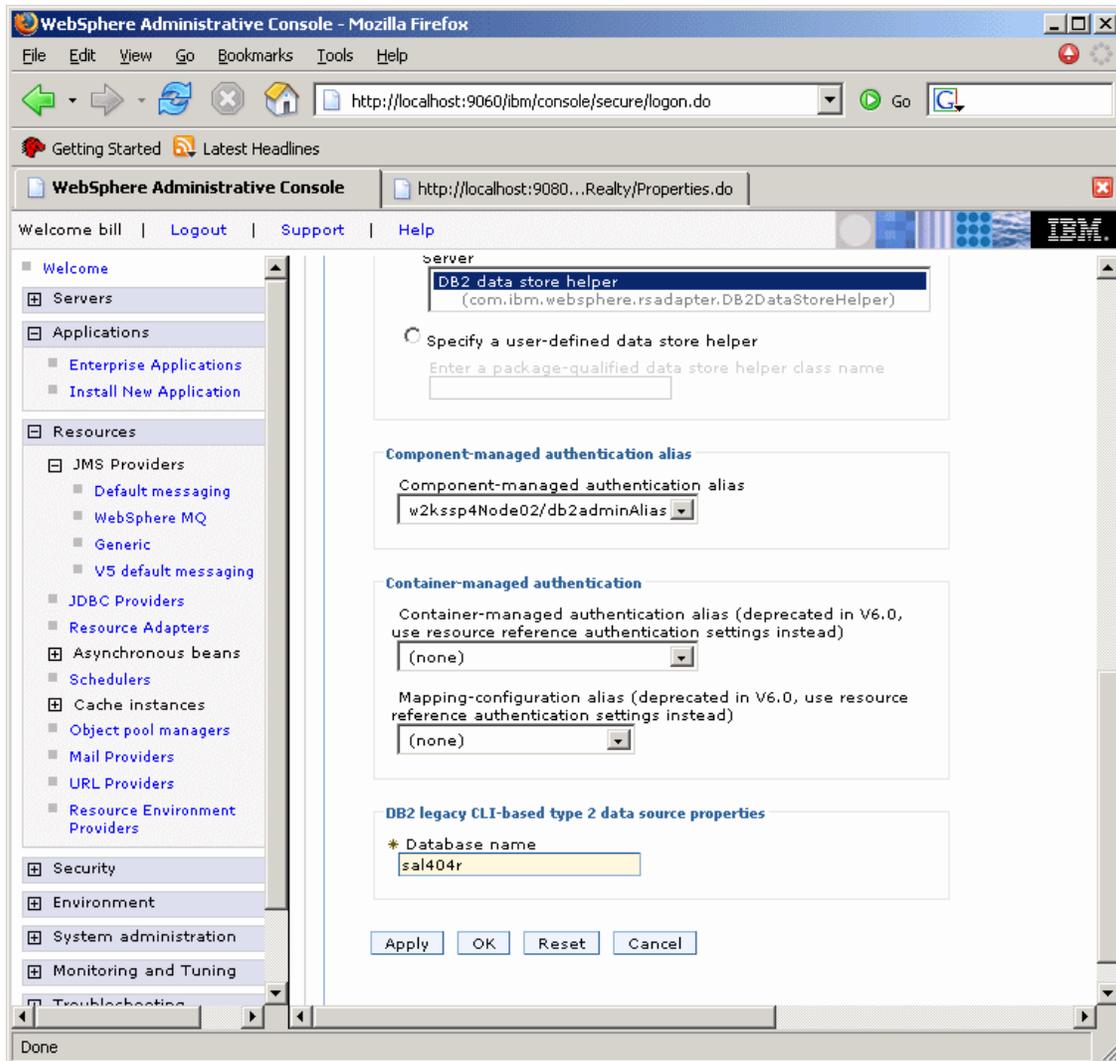
Figure 4-58   DataSource properties: Part 2

   4. Click **OK** and save the configuration.

Set up WebSphere variables

When our JDBC provider was created, it located the required Java driver classes by using a classpath setting of `${DB2_JDBC_DRIVER_PATH}/db2java.zip` where `${DB2_JDBC_DRIVER_PATH}` is a reference to a WebSphere variable that must be set to the correct location where our DB2 install places `db2java.zip`.

To set the variable, use the following steps:

1. Using the WebSphere Administrative Console choose **Environment** → **WebSphere Variables** and select **DB2_JDBC_DRIVER_PATH**. See Figure 4-59.
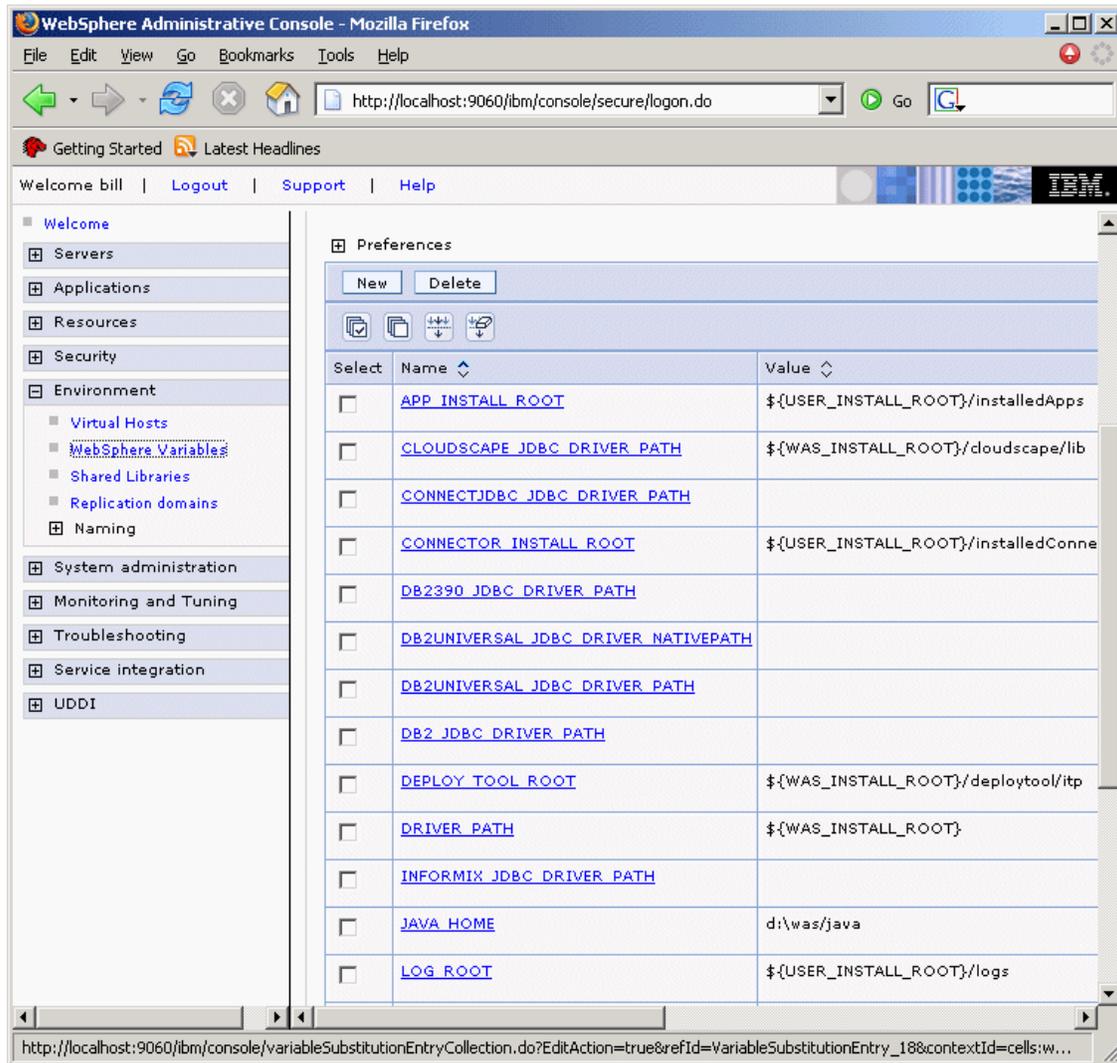
Figure 4-59   WebSphere variables

2. Enter `c:\sqllib\java` as the value for the `DB2_JDBC_DRIVER_PATH` variable as shown in Figure 4-60 on page 185.
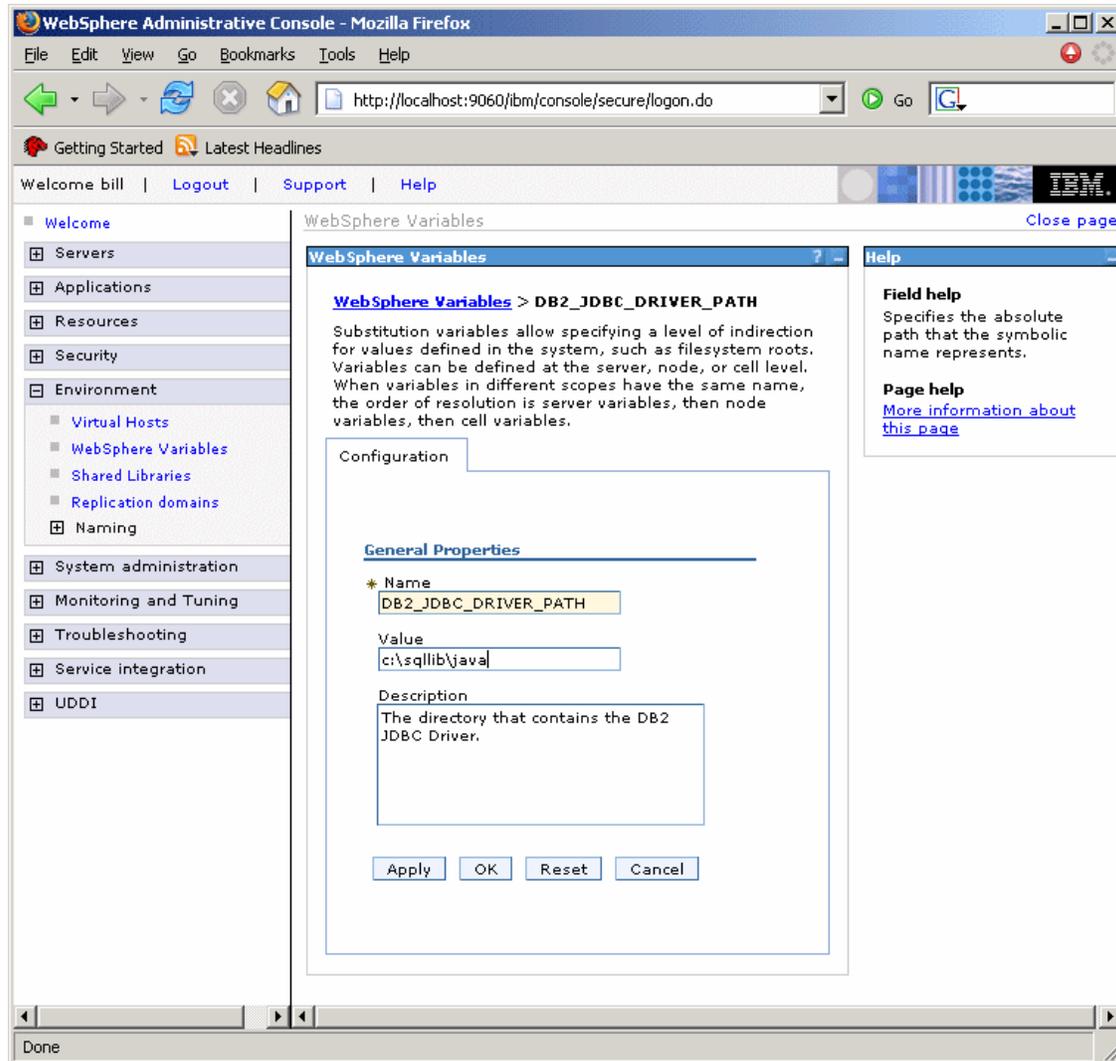
Figure 4-60 Set value for DB2_JDBC_DRIVER_PATH variable

3. Click **OK** and save the configuration.

Test the DataSource connection

To confirm that the JDBC resources are correctly configured you can test the connection to the SAL404R database using the DataSource you created. The steps are:

1. Using the WebSphere Administrative Console choose **Resources** → **JDBC Providers** and select the new JDBC provider **D.B2 Legacy CLI-based Type 2 JDBC Driver**.

2. Scroll to the right of the page and under the **Additional Properties** heading choose **Data sources.**

3. Check the **sal404** DataSource and click **Test connection**. See Figure 4-61 for an example of a successful test of the DataSource connection.
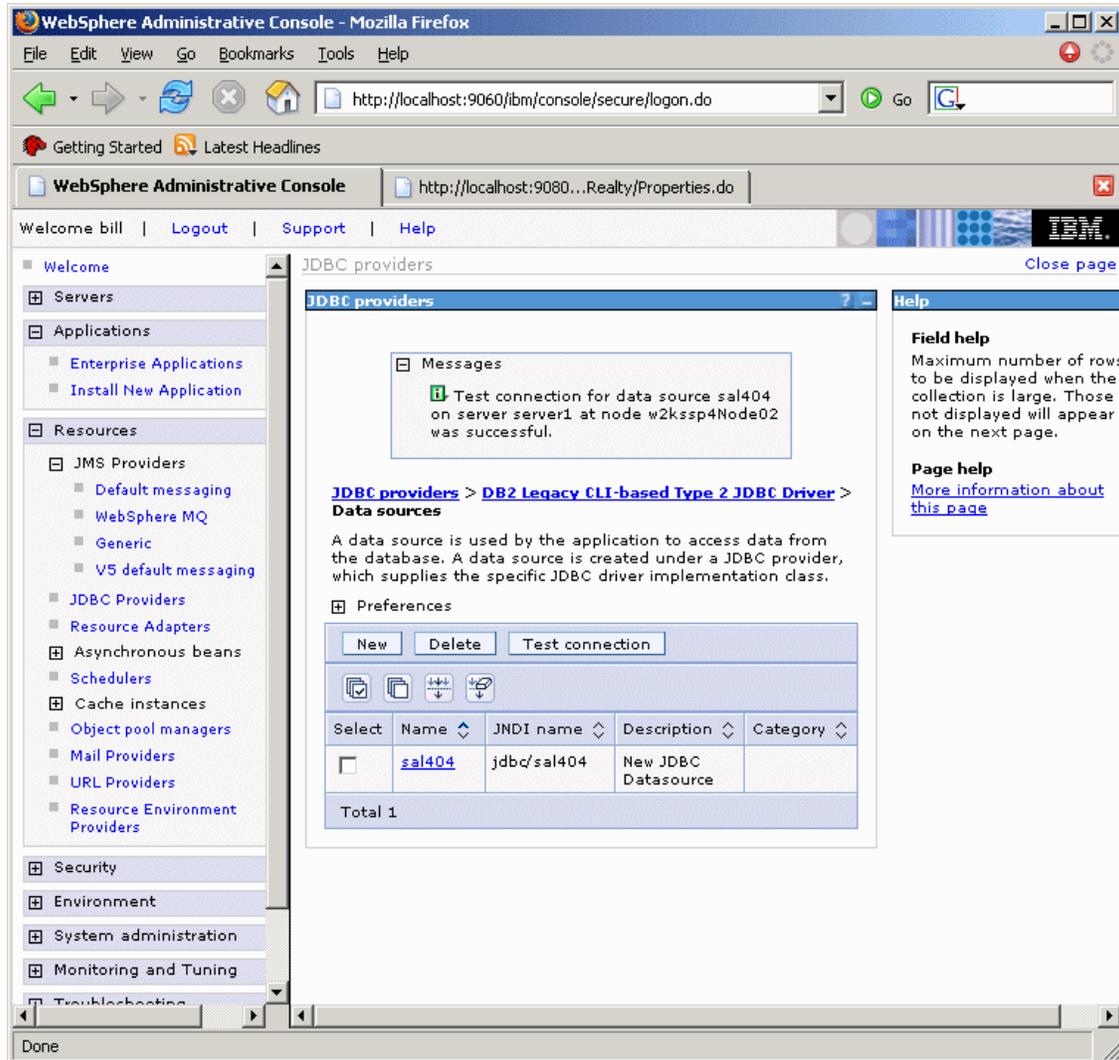
Figure 4-61   Testing the DataSource connection

4.7.3 Configuring JMS

Our Sal404 sample application uses JMS resources as described in Chapter 10, "Java Message Service" on page 359. The application will not run until the required JMS resources are created and configures. The steps we need to take are:

1. Create a new service integration bus.

   See "Create a new service integration bus" on page 409 for details.

2. Specify the server running our application to be a member of that bus.

   See "Add the current server as a bus member" on page 411 for details.

3. Setup a message queue in the bus.

   See "Create a queue as a destination" on page 412, "Create a queue for outgoing messages" on page 413, and "Verify your queues" on page 413 for details.

4. Restart the server.

5. Setup a queue connection factory.

   See "Set up a queue connection factory" on page 415 for details.

6. Setup a queue for incoming messages.

   See Figure 10-25 on page 416 for details.

7. Setup an activation specification.

   See "Set up an activation specification" on page 418 for details.

8. Setup a queue for outgoing messages

   See "Set up a queue for outgoing messages" on page 419 for details.

4.7.4 Configuring LOG4J

The sample application makes use of LOG4J for logging. The LOG4J JAR file can be downloaded from:

   http://logging.apache.org/

In this section we look at setting up LOG4J for the sample application as part of the application installation.

1. Using the WebSphere Administrative Console choose **Environment** → **Shared Libraries** → **New**. See Figure 4-62 on page 188.
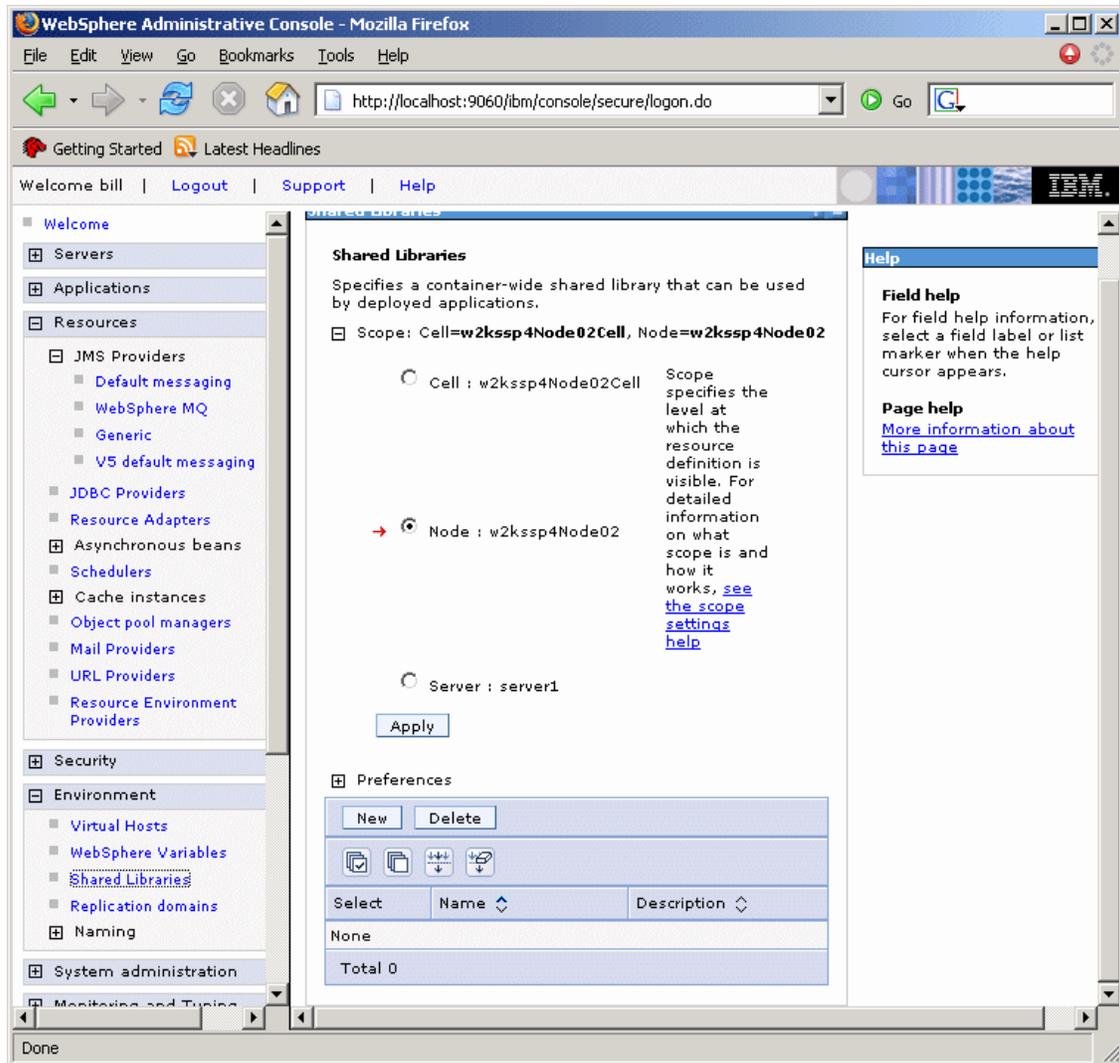
Figure 4-62   New shared library

2. Enter `LOG4J` in the name field. In the classpath field, enter the full path of where the LOG4J JAR file is located. It is good practice to create a dedicated folder for the JAR file. We decided to place the JAR in the `c:\sal404` directory, which is the folder we created for the LOG4J log files. See Figure 4-63 on page 189.
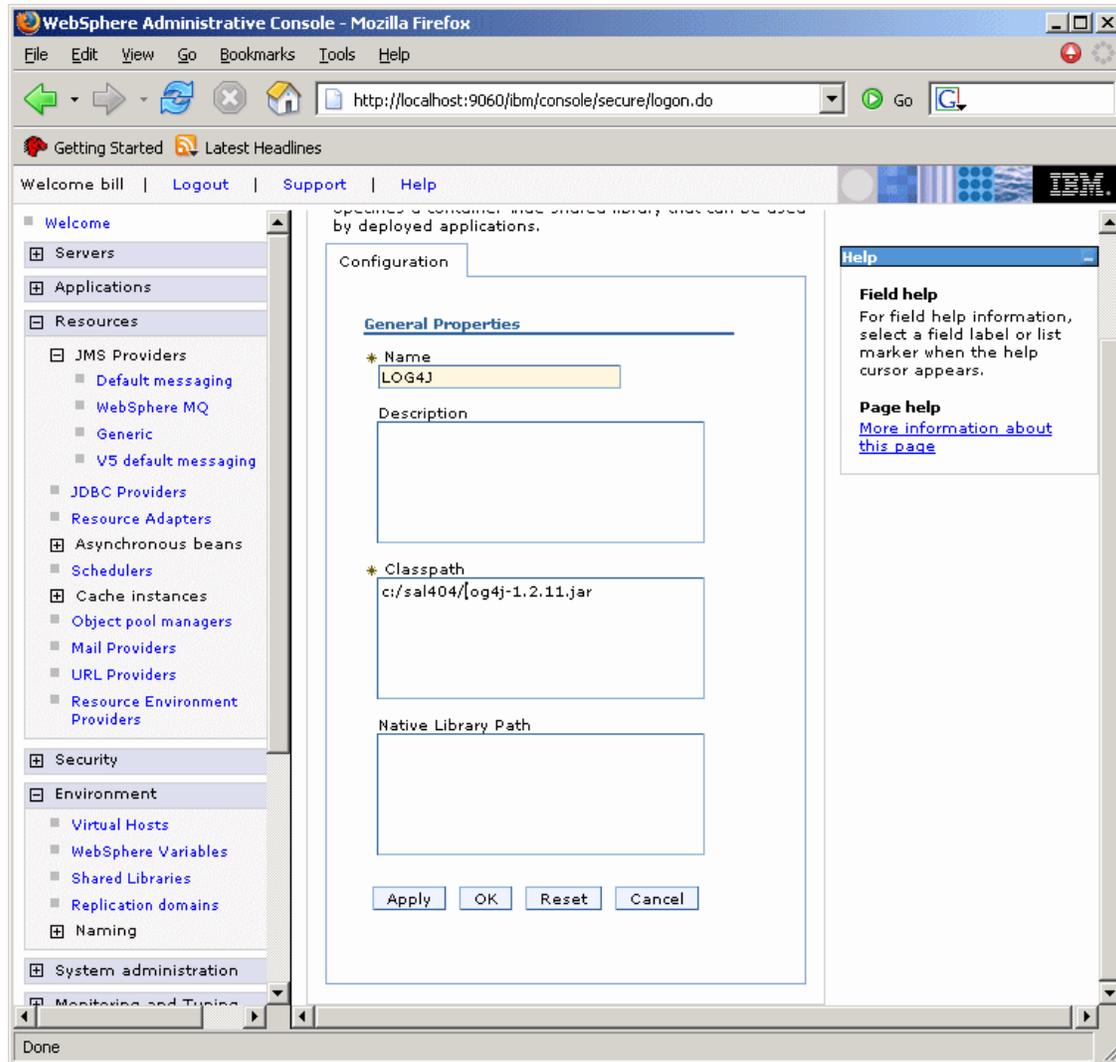
Figure 4-63   LOG4J shared library

3. Click **OK** and save the configuration.

4. You also need to make sure that the LOG4J properties are set correctly before testing our sample application. A default log4j.properties file is packaged with our sample application EAR. The key property to set is the path to the log file in use. In our sample we specify `log4j.appender.ROOT.File=c:/SAL404/sal404.log`.

Note: You must create the folders and file specified for use by LOG4J before starting our sample application. If you need to change the default settings we provide, you will have to extract the properties files from the EAR, make the required changes, and then repackage the EAR.

4.7.5 Installing the Sal404 application EAR

In this section we look at installing the Sal404 sample application EAR file. The EAR file is provided as part of our redbook additional material. See "How to use the Web material" on page 618 for details of where the EAR file can be found in our additional material. To install the EAR file, the steps are:

1. Using the WebSphere Administrative Console choose **Applications** → **Install New Application.**

2. Click **Browse** to locate the sample application EAR file as shown in Figure 4-64 on page 191.
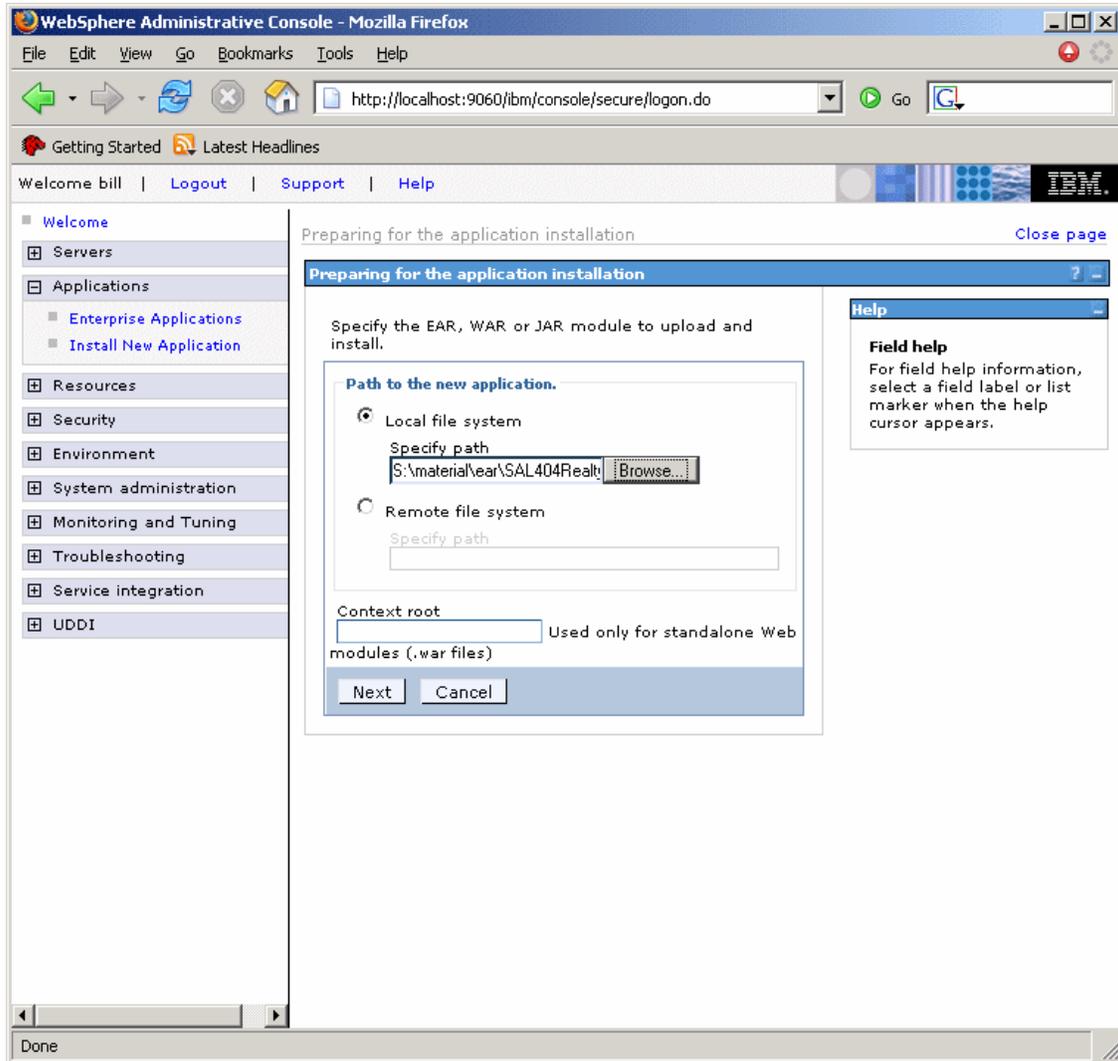
Figure 4-64   Locate EAR file for installation

3. Click **Next**.

4. On the next page, Preparing for the application installation, we can accept all the default settings and click **Next**. See Figure 4-65 on page 192.
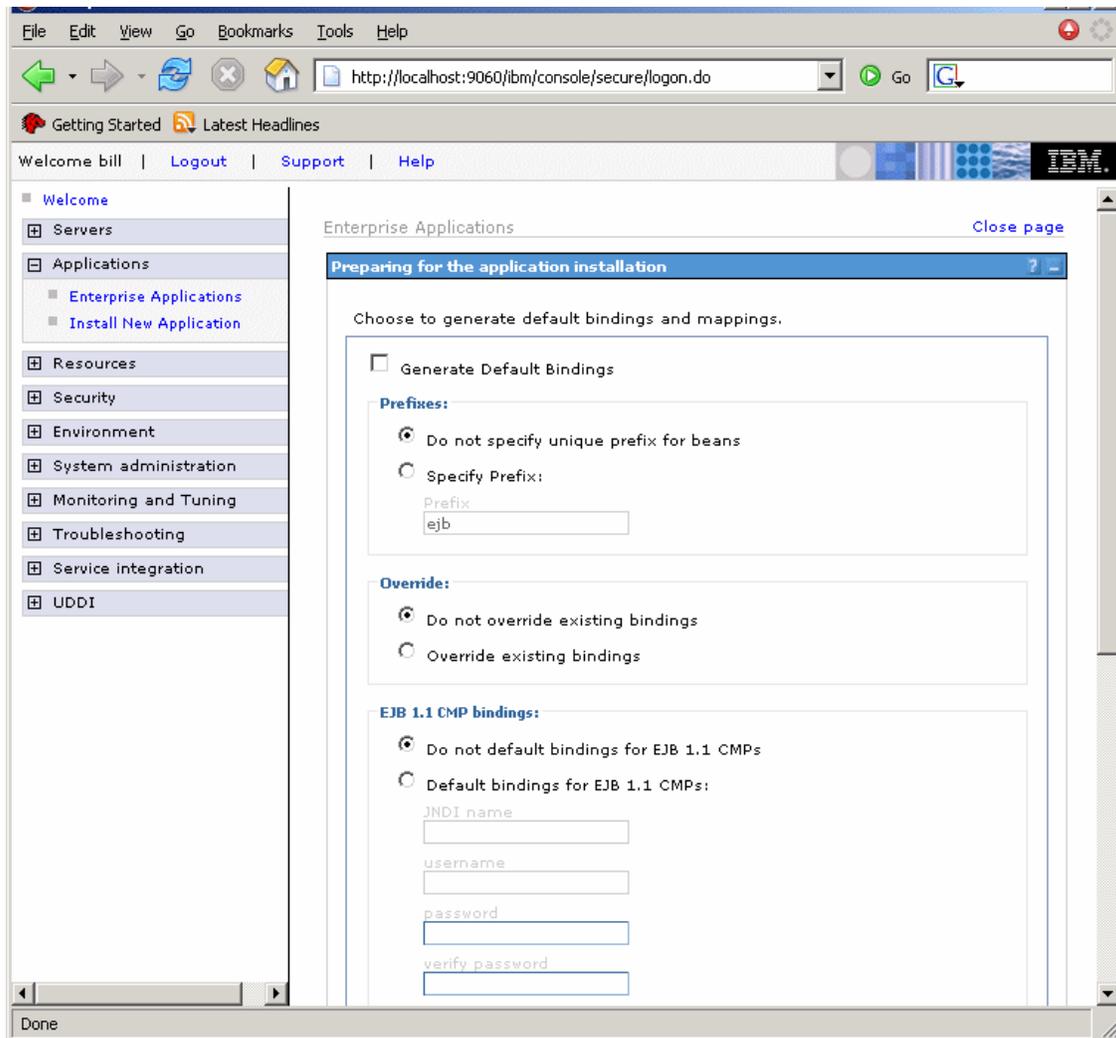
Figure 4-65   Preparing for the installation

5. On the Select installation options page as shown in Figure 4-66 on page 193, we can also accept the default settings and click **Next**. In fact we can take default settings for each of the steps in the EAR installation process. Instead of clicking **Next** on the Select installation options page we could select the link for the summary step and proceed directly to the installation summary page where we could click **Finish** to complete our install.

However for this exercise, we advise you to review each of the installation steps to get a better idea of the components of the application and how it is

deployed. You can continue to click **Next** to proceed through all the steps in the installation or select a link for a particular step to go directly to that page.
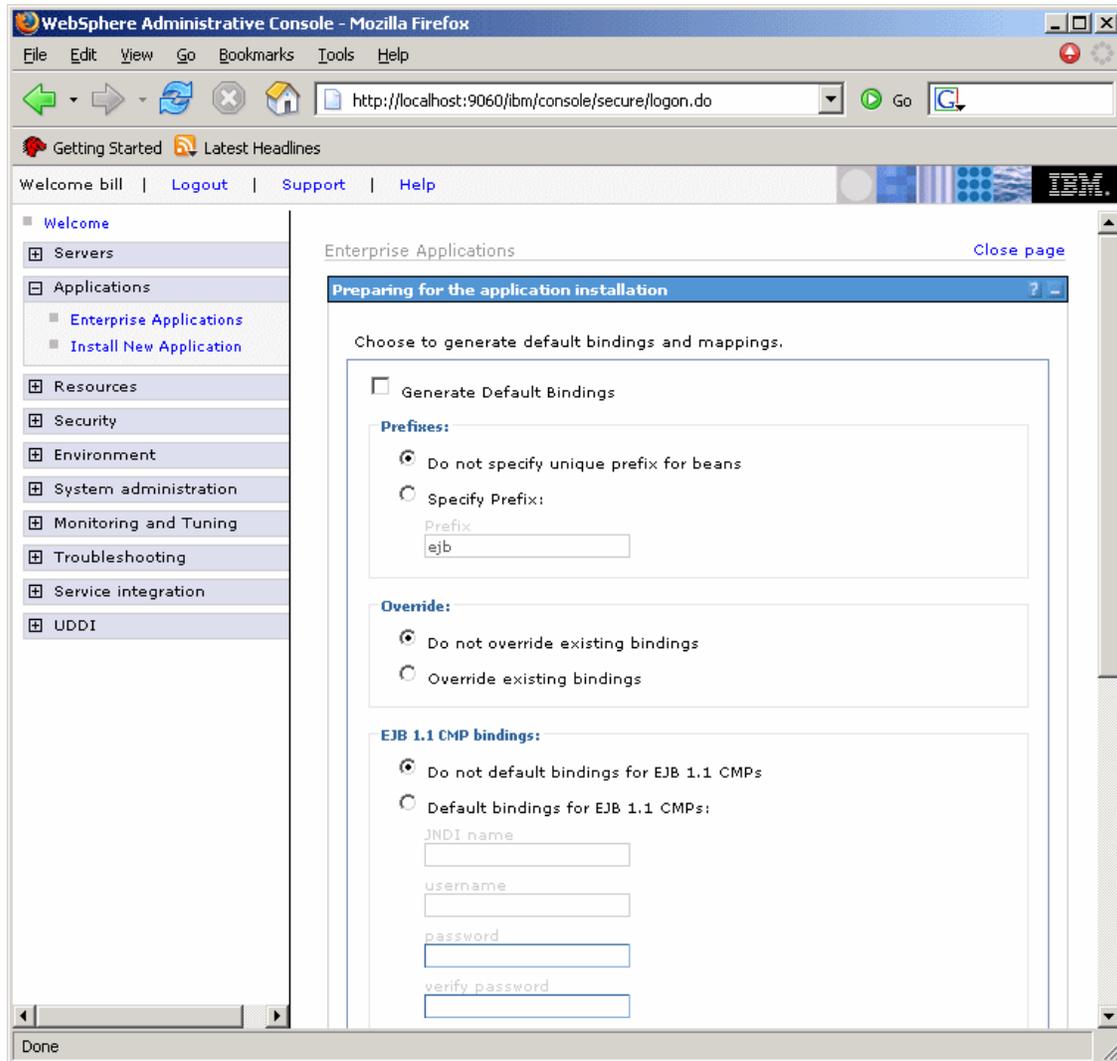
Figure 4-66   Installation options

6. As you can see from Figure 4-66, there are 12 steps we can view as part of the EAR installation. These are:

   – Step 1: Select installation options

   – Step 2: Map modules to servers

   – Step 3:Select current backend ID

- Step 4: Provide listener bindings For message-driven beans

- Step 5: Provide JNDI Names for Beans

- Step 6: Provide default data source mapping for modules containing 2.x entity beans

- Step 7: Map data sources for all 2.x CMP beans

- Step 8: Map EJB references to beans

- Step 9: Map resource references to resources

- Step 10: Map virtual hosts for Web modules

- Step 11: Ensure all unprotected 2.x methods have the correct level of protection

- Step 12: Summary

> **Note:** The number of steps required to install an EAR file is dynamic and changes depending of the components packaged in the EAR.

7. Figure 4-67 on page 195 shows the summary page for the installation of our sample EAR. Click **Finish** to complete the installation.
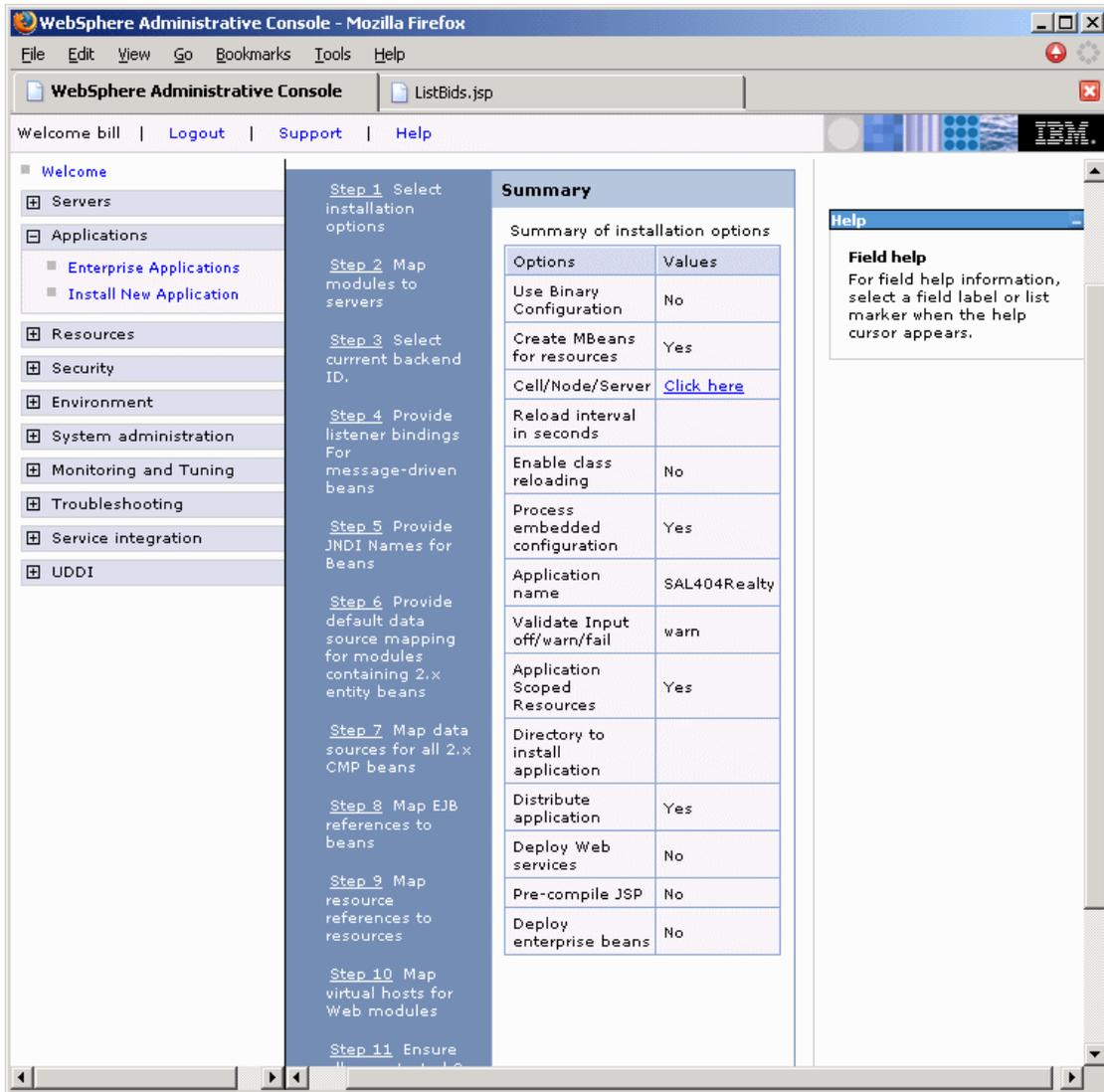
Figure 4-67   Installation summary

8. Figure 4-68 on page 196 shows the messages from a successful installation of the Sal404 sample application EAR file.
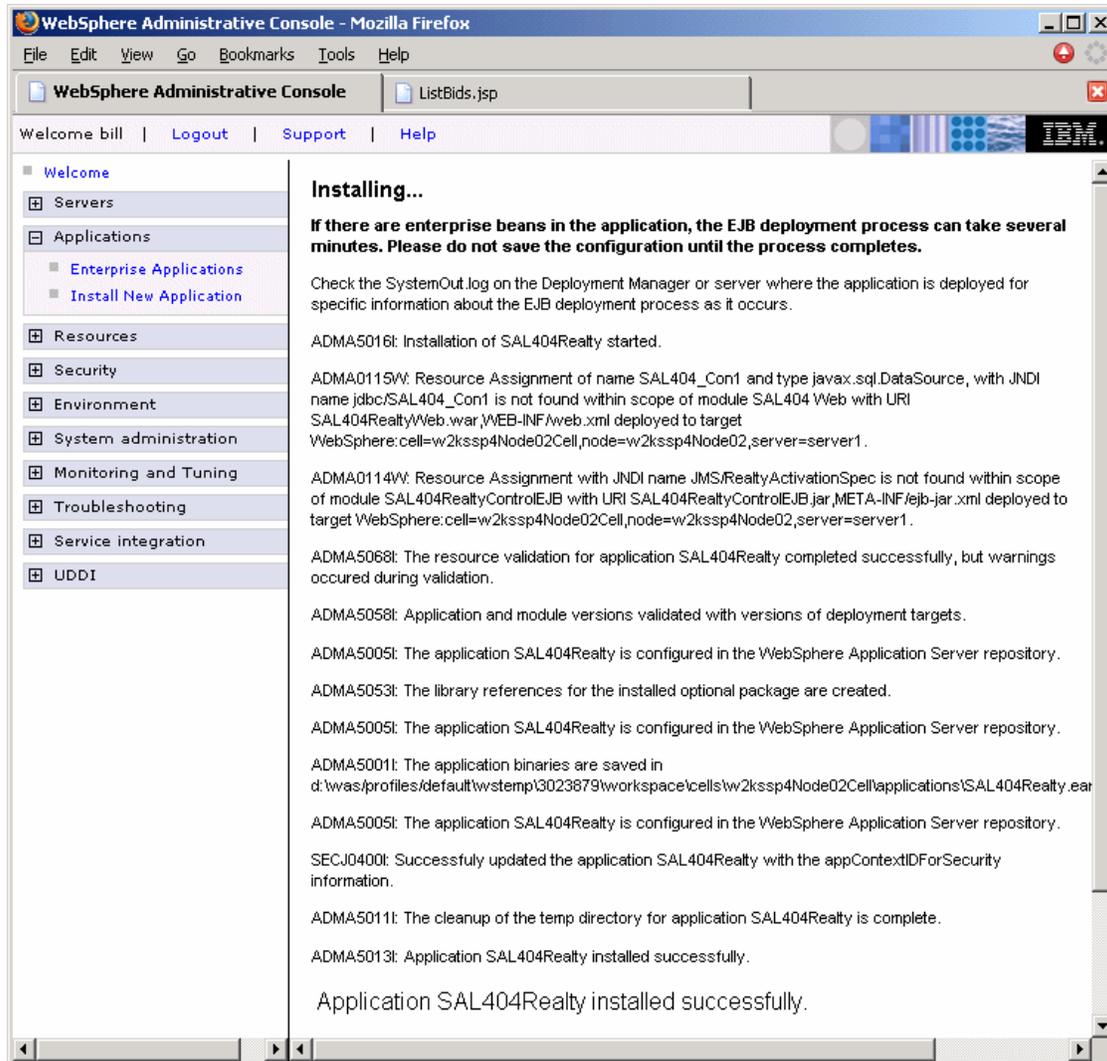
Figure 4-68   EAR file installation successful

9. Save the configuration changes.

4.8 Testing the Sal404 sample application

The purpose of testing the sample application is to verify the application is successfully running and extracting data from the data source. Complete the following steps to test the sample application:

1. In the WebSphere Administrative Console, expand **Applications** →
   Enterprise Applications from the menu.
2. In the Enterprise Applications window, check the box to the left of
   SAL404Realty application and click **Start**. This starts the application and is
   shown in Figure 4-69.
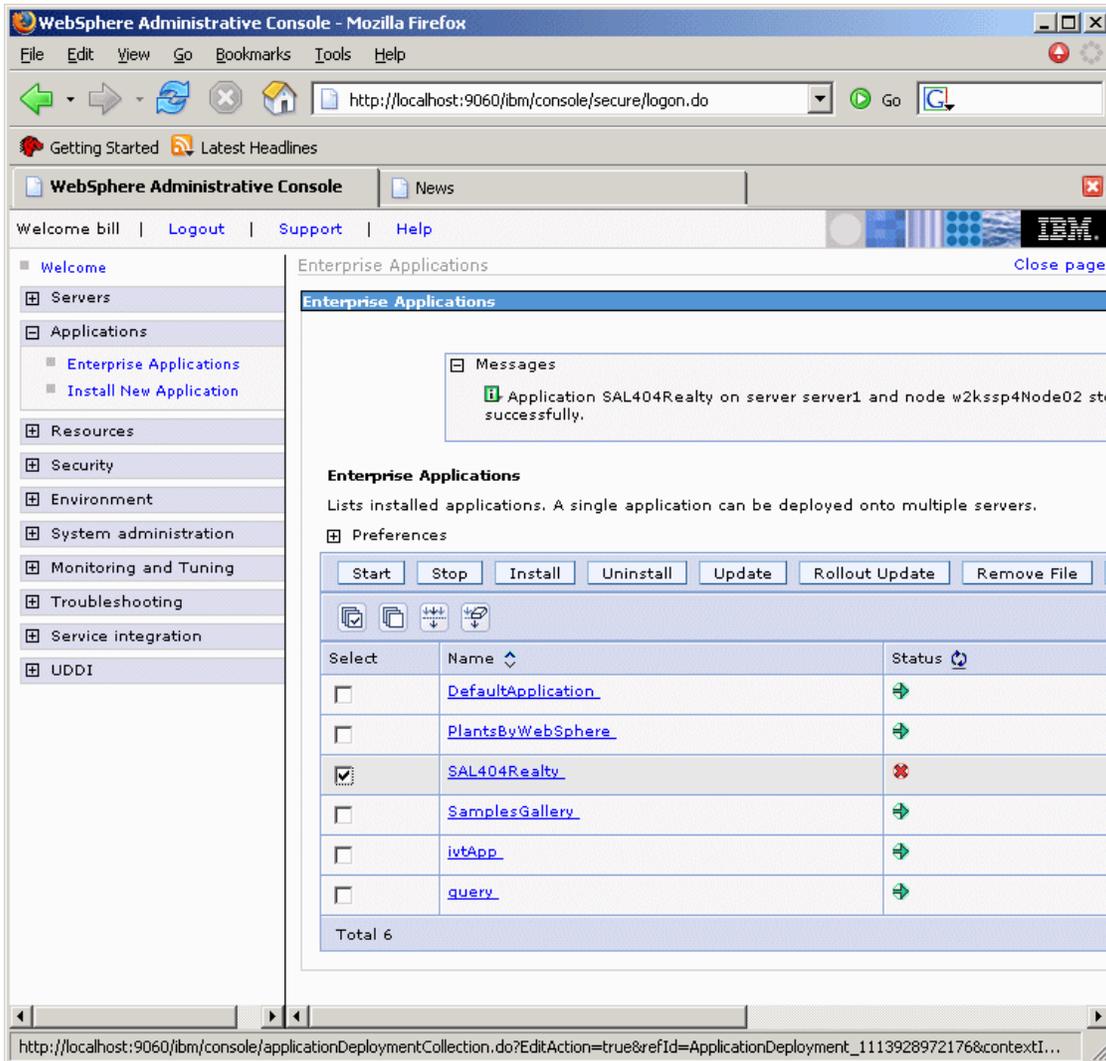
Figure 4-69   Start the application

3. The status image turns green after the application has started as shown in
   Figure 4-70 on page 198.
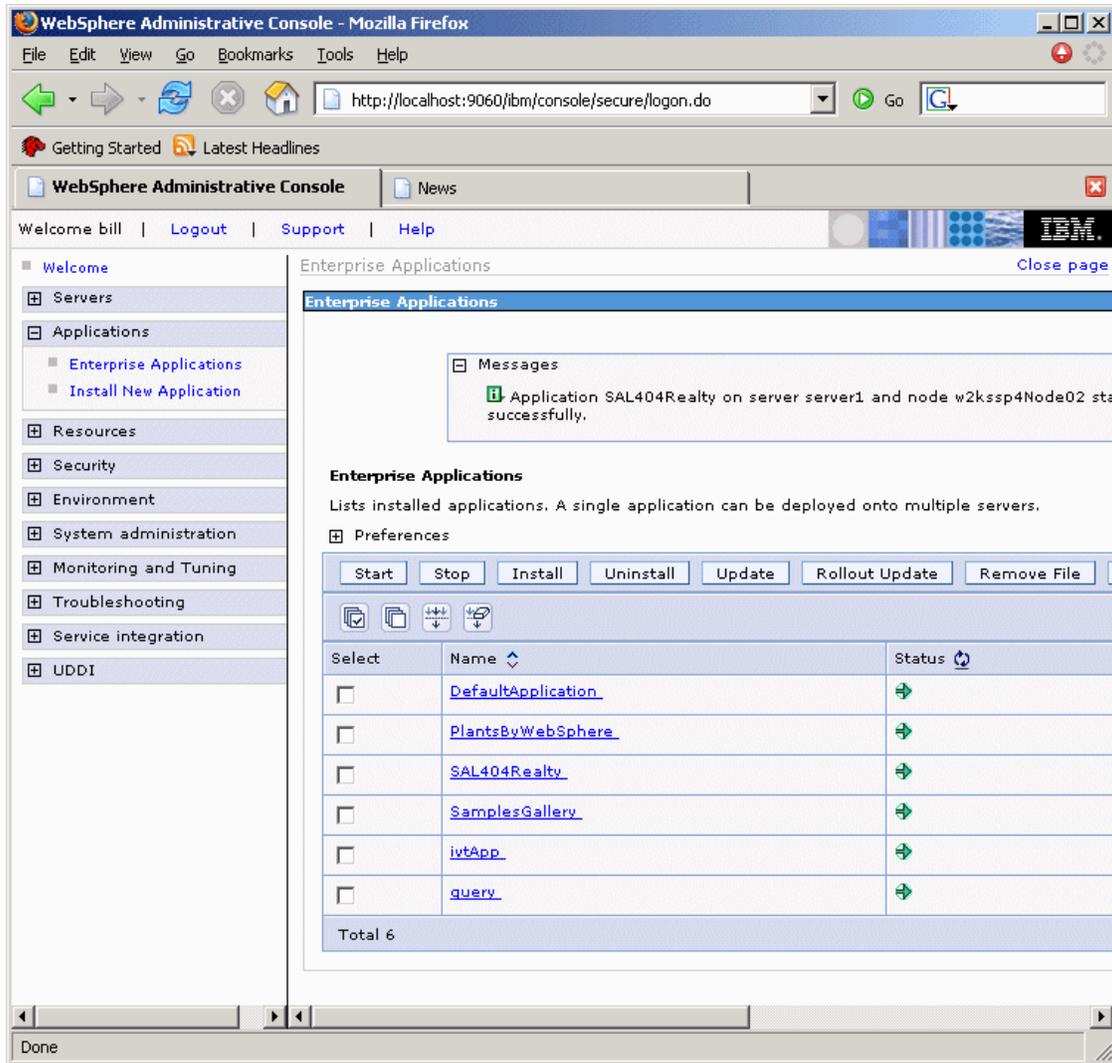
Figure 4-70   Application started

4. Open a Web browser and enter the address:

```
http://localhost:9080/SAL404Realty/
```

The SAL404 Realty Home Page will appear in the Web browser as shown in Figure 4-71 on page 199.
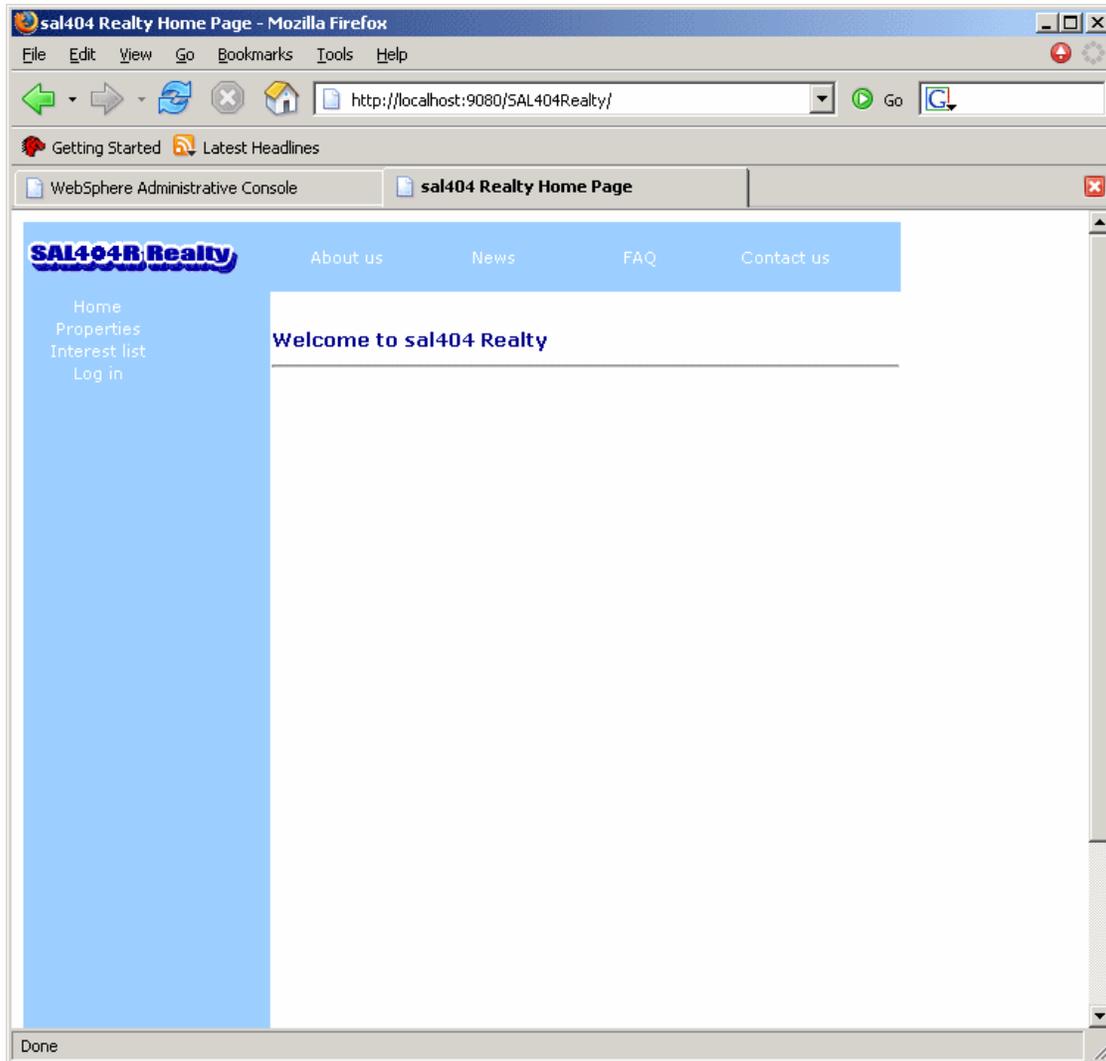
Figure 4-71   Sal404 home page

5. Follow these steps to verify that the user information is properly configured.

   a. Click **Log in**.

   b. At the User Login page enter `bill` in the User ID field and `password` in the Password field. See Figure 4-72 on page 200.
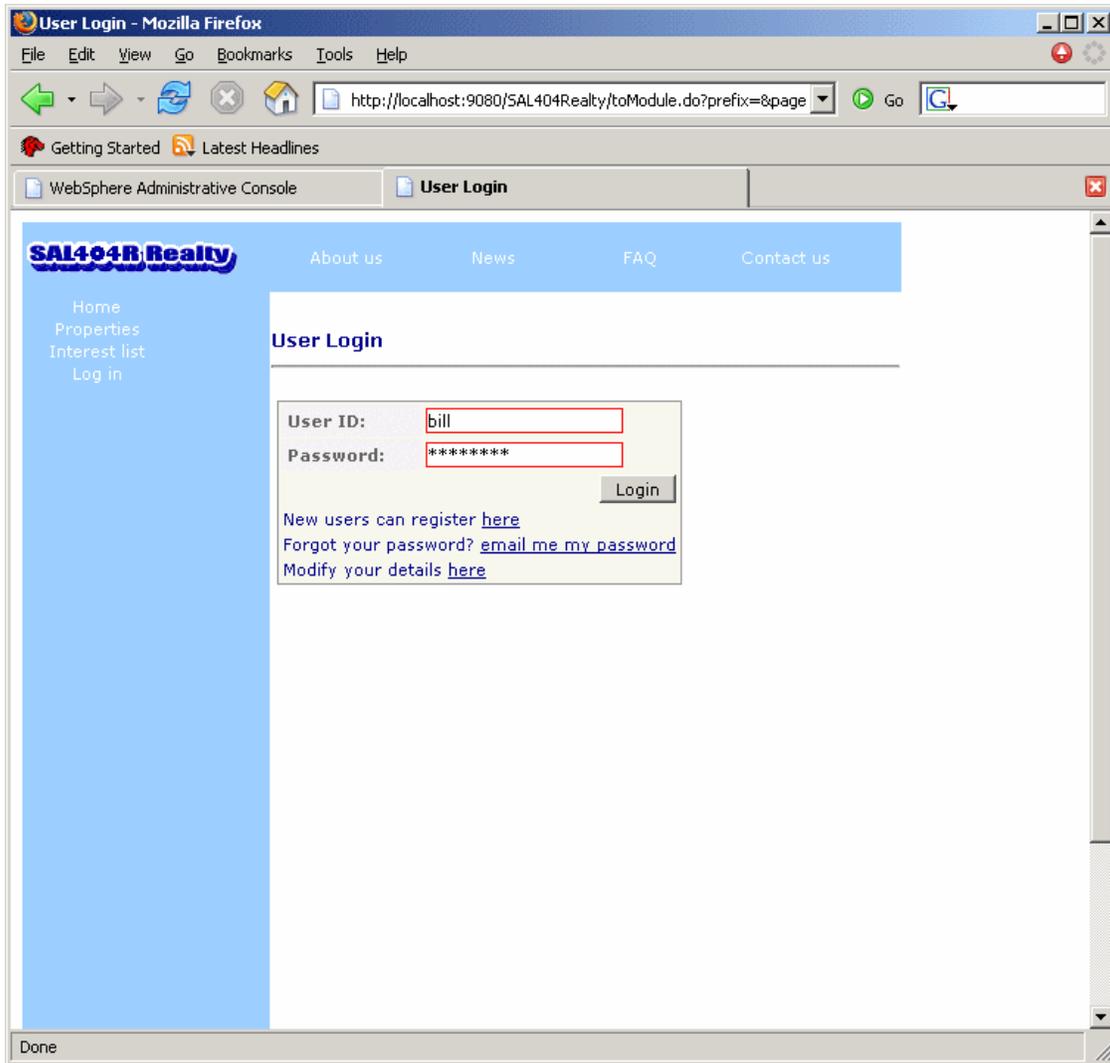
Figure 4-72   Login to Sal404 application

> c. You should be logged in as user bill and the page shown in Figure 4-73 on page 201 is displayed.
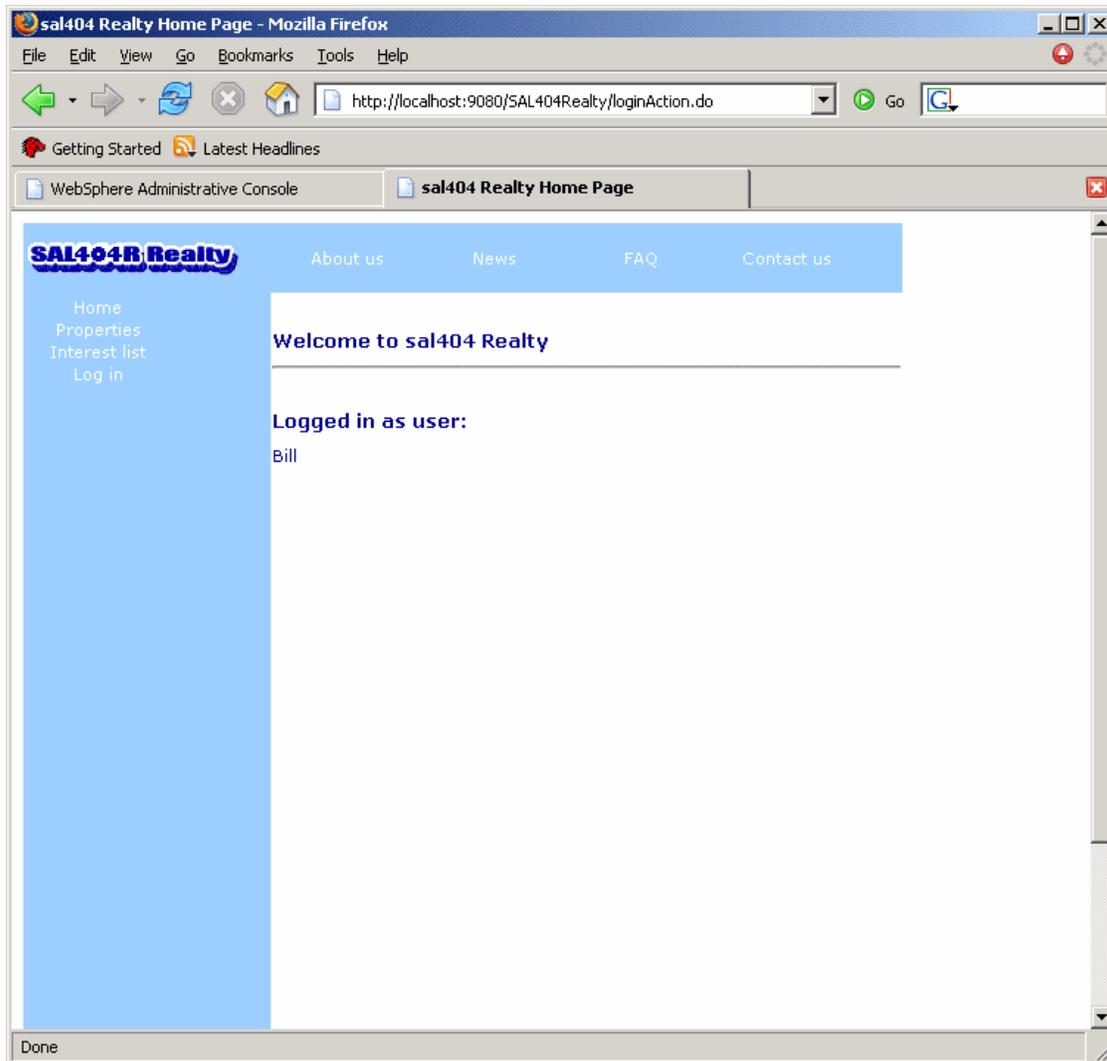
Figure 4-73   Logged into the Sal404 application

> **Note:** The tests described in this section are sufficient to show that the Sal404 application is installed successfully. However we recommend that you explore the application further both to become more familiar with its features and also to double check that all areas are correctly configured and working properly.

If you want to test the JMS features of the Sal404 application as described in "Test the Message Driven Bean" on page 443 you will first have to use the

WebSphere Administrative Console to install the test application EAR SAL404JmsClientEAR.ear from our redbook additional material. Similarly, if you want to test the Web services features as described in Chapter 12, "Web services" on page 475 you will first have to use the WebSphere Administrative Console to install the test application EAR SAL404TestServices.ear from our redbook additional material.

4.9 Installing Sal404 code in Rational Web Developer

We provide the source of our sample code so that you can import it to a Rational Software Development Platform workspace. In this section we show how to add our sample code to the Rational Web Developer environment that is shipped with WebSphere Application Server - Express.

If you want to develop or test code that uses Enterprise JavaBeans, then you will need to install and configure our sample code in Rational Application Developer. We do not provide separate instructions for installing the code in Rational Application Developer because the same method as we document for Rational Web Developer also works with Rational Application Developer.

4.9.1 Importing project interchange files

Our redbook additional material includes a project interchange file called Sal404Interchange.zip that contains our Sal404 sample code in a format that can be easily imported into Rational Web Developer. To import the code:

1. Start Rational Web Developer and choose **File** → **Import**.
2. Select **Project Interchange** as the import format and click **Next** as shown in Figure 4-74 on page 203.
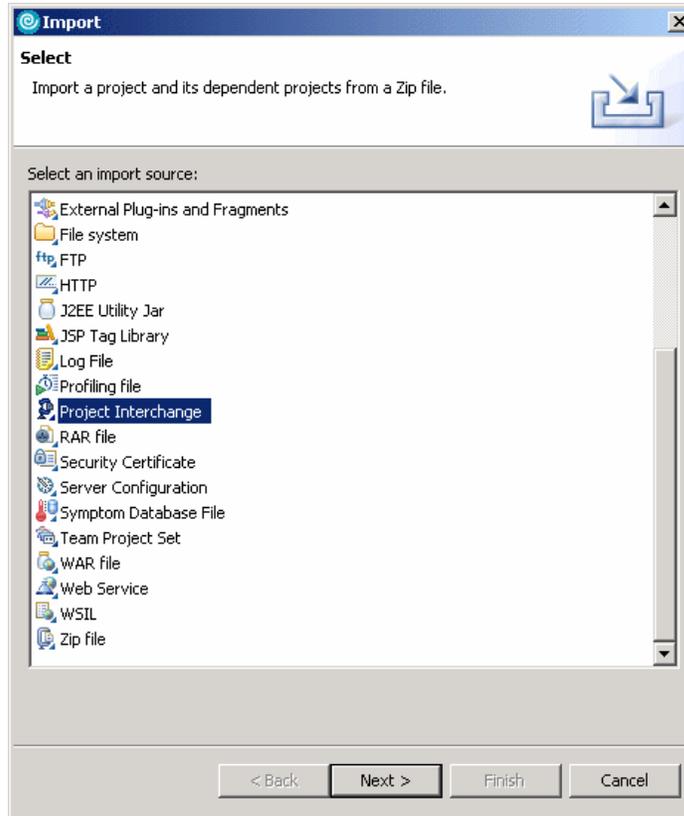
Figure 4-74 Import from project interchange

3. Click **Browse** to locate the downloaded Sal404Interchange.zip file.

4. Click **Select All** to import all of the Sal404 sample code and then click **Finish**. See Figure 4-75 for an example.
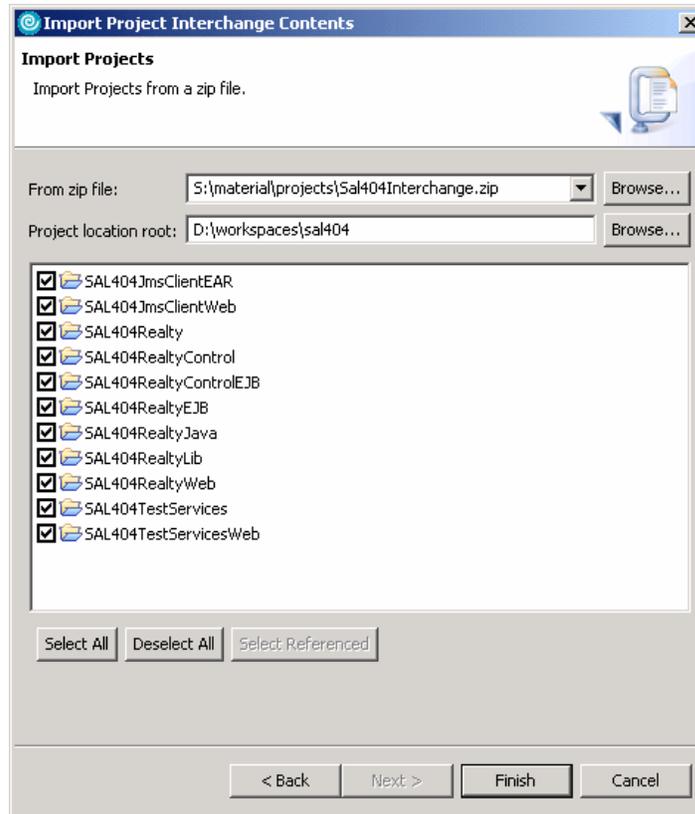
Figure 4-75   Import Sal404 applications

4.9.2 Test Sal404 with Rational Software Development Platform

The Sal404 sample application can be run and tested using the WebSphere Application Server test server that is provided as part of Rational Web Developer or Rational Application Developer. To deploy the Sal404 sample application using the default test server that is created when Rational Web Developer is installed, the steps are:

1. Start Rational Web Developer and switch to the servers view.

2. Start the default test server by right clicking the **WebSphere Application Server v6.0** server and choosing **Start**.

3. Once the server starts, you can use the WebSphere Administrative Console to administer the server. Start the WebSphere Administrative Console by right-clicking the **WebSphere Application Server v6.0** server and choosing

Run administrative console. This will start the WebSphere Administrative Console in a Web browser embedded within Rational Web Developer.

> **Tip:** You can still start the WebSphere Administrative Console externally to Rational Web Developer by entering the following URLs in a Web browser:
> - `http://localhost:9060/admin`
> - `http://localhost:9060/ibm/console`

4. To deploy the Sal404 sample application, right-click the **WebSphere Application Server v6.0** server and choose **Add and remove projects**.

5. Select **SAL404Realty** from the list of available projects and click **Add** > to move it to the list of configured projects, then click **Finish**. See Figure 4-76.
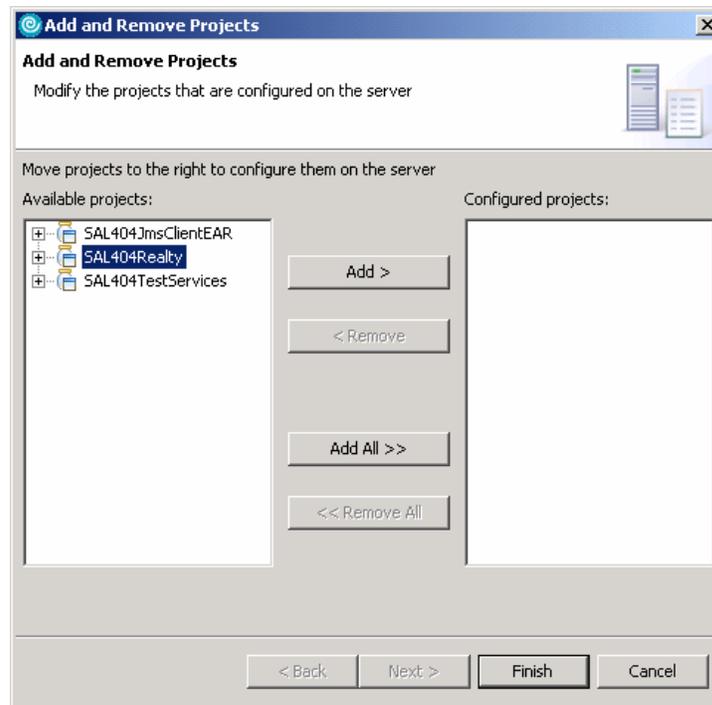
Figure 4-76   Deploy Sal404 to test server

> **Note:** If you want to test the JMS features of the Sal404 application as
> described in "Test the Message Driven Bean" on page 391 you will first
> have to add the SAL404JmsClientEAR project to the Rational Web
> Developer test server. Similarly if you want to test the Web services
> features as described in Chapter 12, "Web services" on page 423 you will
> first have to add the SAL404TestServices project to the test server.

6. When the SAL404Realty project is first added to the test server, publishing
   will fail with an error dialog box similar to that shown in Figure 4-77. This is
   because you have not yet configured the resources necessary for the Sal404
   application to run in the test server.

Figure 4-77   Publish errors

7. To complete the configuration of the Sal404 application so that it will run in the
   test server, perform these tasks:

   – Create the required JDBC resources using the same steps described in
     4.7.2, "Creating the JDBC resources" on page 176.

   – Create the required JMS resources using the same steps described in
     4.7.3, "Configuring JMS" on page 187.

8. To test the JSF features of the Sal404 application in the test server you might
   need to alter the password of the authentication alias used to connect to the
   SAL404R database. The JSF implementation of the News component does
   not use the Sal404 DataSource created in, "Create a JDBC DataSource" on

page 180 because it uses a connection that is defined as a deployment setting in the application EAR file. To alter this setting, use these steps:

a. Navigate to the SAL404Realty project, right-click **Deployment Descriptor: SAL404Realty**, and click **Open**.

b. Choose the **Deployment** tab and scroll down to the Authentication section. Choose **wdo_SAL404_Con1** from the JAAS authentication list, and click **Edit**. See Figure 4-78 on page 207.
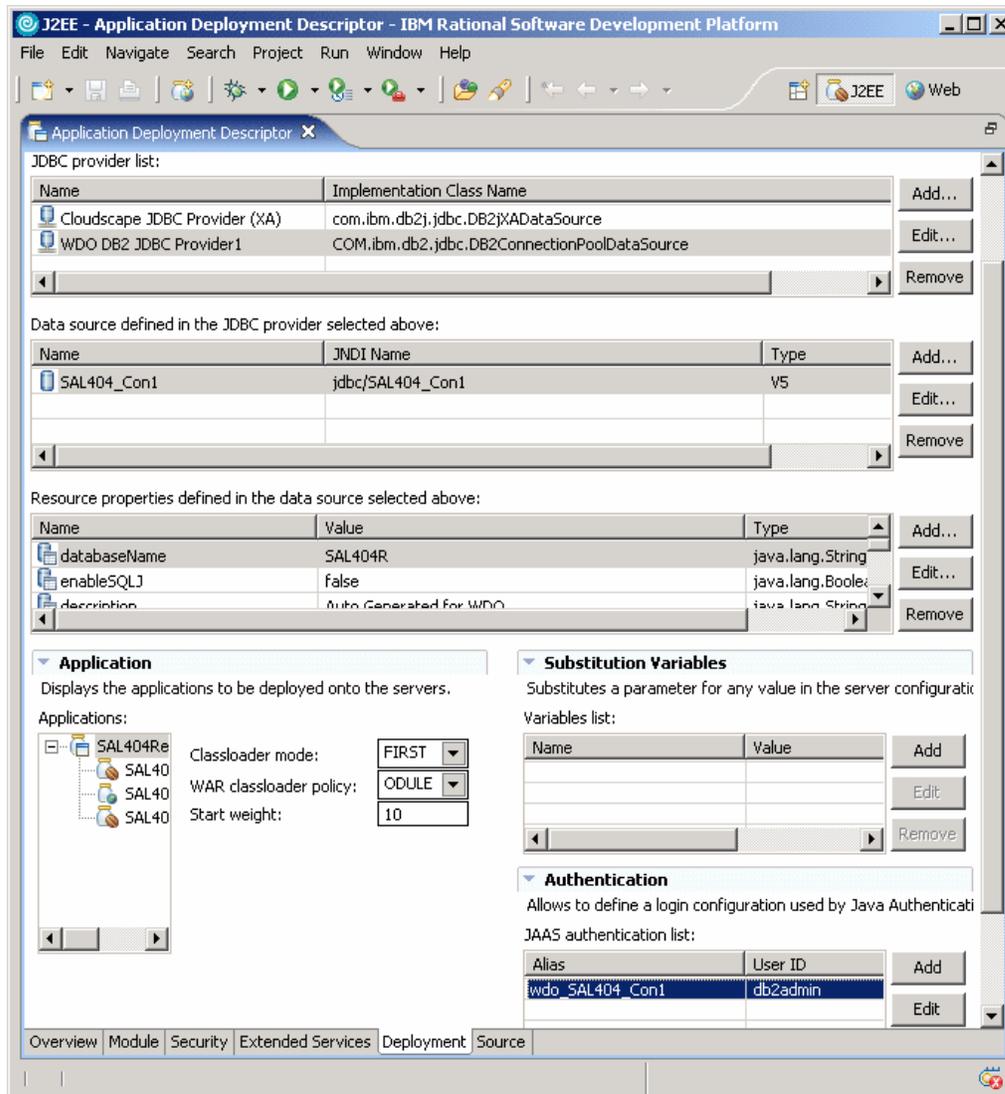

Figure 4-78   Edit deployment settings for the SDO database connection

c. Enter the correct password for the db2admin database user as shown in Figure 4-79 and click **OK** to save the changed authentication entry.
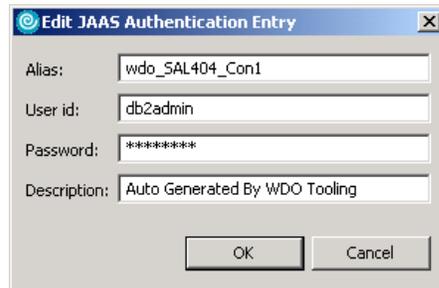
Figure 4-79   Change the authentication entry

d. Save and close the application deployment descriptor.

e. Restart the Sal404 sample application by right-clicking the **WebSphere Application Server v6.0** server and choosing **Restart Project** → **SAL404Realty**.

5

Requirements

This chapter provides an overview of the new requirements for the sample application developed for our redbook. The Sal404 example application is based on the SAL301RRealty sample application developed for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301. We describe requirements, design, and specification of our sample application. When describing the requirements, we outline both the new functionality of our sample and also describe the enhancements made to the existing application.

5.1 Application overview

Our sample application is a standard layered application and it is still mainly implemented as a Struts application, so it is also very model-view-controller (MVC) oriented. Many texts and articles are available for detailed discussions on n-tier applications and MVC. Your basic understanding of these concepts is assumed. The goal of this chapter is to present a practical overview what we developed and why it was required, rather than to discuss the theoretical aspects of our design choices.

The sample application presented is fairly straightforward. The application manages:

- Users
- Properties
- News items
- Interest in properties
- Reference data

The application also provides for e-mail alerts.

The term *manages* simply means that the application provides for Create/Read/Update/Delete (CRUD) functionality for a given component.

The application also has some miscellaneous functionality. This includes HTTP session management, logging, and so on. The HTTP session management is of particular importance. As in most Web applications, session management is used both for security aspects of the application and also as a place to store the application state as well as any user and session-specific information.

The model-view-controller (MVC) design pattern separates the parts of an application. MVC is not unique to Web applications or to the Struts implementation; it was around well before Web applications.

Web applications that use model-view-controller are divided into three functional areas:

Model The model contains the core of the application function, and captures the state of the application. It does not include knowledge of the view or controller. The model is the business logic, which in most cases involves access of data stores like relational databases. The development team that handles the model may be expert at writing DB2 COBOL programs, or EJB entity beans, or some other technology appropriate for storing and manipulating enterprise data.

View	The view is the look of the application. The view presents, gathers, and submits information, but it does not include knowledge of the model or controller. The view is the code that presents images and data on Web pages. The code comprises JavaServer Pages and the JavaBeans that store data for use by the JavaServer Pages.
Controller	The controller manages the execution flow of the application, passing appropriate state information between the model and the view.

This model-view-controller division (sometimes called *Model 2* - see 11.3, "Model-view-controller (MVC) pattern with Struts" on page 450 for more details) includes parts that are independent of each other, so that changing how one part is implemented does not require changes to the other parts. For example, the view of a Web application can change many times due to usability testing. However, the business logic (the model) acting on the input does not need to change (assuming the inputs to the business logic stay the same).

Figure 5-1 shows the traditional 3 layer application architecture followed by our sample application. As illustrated in Figure 5-1, the Web server at runtime contains both the view and controller components of a Model 2 Web application, while a third tier (which is usually outside of the Web server) contains the model. The diagram also references Struts-specific components.
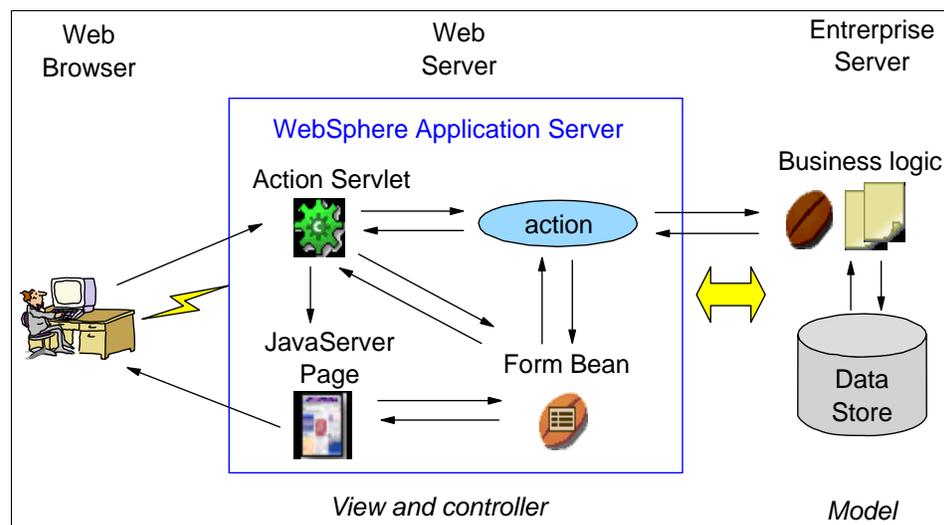
Figure 5-1   Three tiered application and MVC roles

5.2 Requirements

Several new requirements have been introduced for this version of the application. These include:

- ► A bidding system
- ► Search the property catalog with a Web services
- ► Notification of the addition or modification of a news item with a Web service
- ► User maintenance with a Message Driven Bean (MDB)
- ► Implementation of the news system using JavaServer Faces(JSF)
- ► New functionality will be created to manage reference data

The bidding system is the only new component. The other new requirements expose existing functionality using different technologies.

5.2.1 Bidding system

The bidding system will allow users to bid on properties. The bidding system will work this way:

- ► To keep our requirement simple, we will allow only one outstanding, or open, bid on a property. A property may have more than one bid, but all bids except one will be in the rejected state. The last remaining bid will be in either the open state or the accepted state. Potentially, all bids can be in the rejected state. A user cannot place a bid on a property if that property has an open bid or an accepted bid. A result of this simplification is that only the latest chronological bid can be open or accepted.
- ► Sellers will be allowed to see the bids on the property and accept or reject the open bid. All of the bids that have been placed on a property are referred to as the *bid history*. The agent selling the property and administrators will be allowed to see the bid history but will not be allowed to accept or reject a bid. When a bid is displayed to any user, both the bid amount and the bid state are displayed.
- ► A bidder will be allowed to only view their bids. The bidder can have placed multiple bids on a property that have been rejected. Thus, a bidder is shown his or her own filtered view of the bid history.
- ► Nonlogged-in users cannot place bids or see any bidding history.
- ► Only properties that are in the active status can be bid upon. When a bid is accepted, the property status is updated to sold.
- ► The system will restrict the seller, or the agent selling the property, from bidding on the property.
- ► Nonrequirements are as follows:

- It is not a requirement that an accepted bid be of a greater amount than any rejected bid.
- Notifications to any bidder, seller, or agent when a bid is placed, accepted or rejected are also not required.

5.2.2 Catalog search and news feed Web services

A Web service interface will be provided to allow users to obtain a list of properties that match a input search criteria.

A Web service interface will be provided to allow users to add and list news items.

5.2.3 User maintenance with Java Message Service

A Java Message Service(JMS) interface will be provided to the User management component. This will allow users to be created or modified upon receipt of a JMS message.

Whenever a user is added or modified by the Web application, an output JMS message will be created.

5.2.4 Use JavaServer Faces for the news component

This requirement allows us to replace the Struts view and controller functions in the News component with a JavaServer Faces implementation. The purpose is to investigate JSF technology and become familiar with the JSF tools provided by Rational Web Developer.

We also compare a JSF and SDO implementation of the news component with a JSF implementation that reuses our existing data access layer.

5.2.5 Reference data component

Nearly all applications have the concept of reference data. Reference data can be lists of user titles (Mr., Mrs., Miss) or country codes. Statuses are also typically reference data as are items such as user roles (Administrator, Customer, Agent, and so on).

The reference data component will be implemented just as any other component. Even though it is stateless, the manager will still be wrapped by a session data class. This follows the application architecture and design principles.

The motivation for the reference data component is to centralize the placement of reference data. Also, as will be seen in the design section for the reference data component, the implementation will minimize resource usage for the application.

The reference session data class will always be available in the HTTP session. There is no need to check for its existence.

The reference session data class will expose the lists for property types and statuses and the country code list.

5.3 Specification

More detailed specifications for the requirements are presented in this section. Where the requirement is for significant new function, we have provided a detailed specification, but for requirements where we are enhancing existing functions we do not attempt to do a detailed specification in this chapter. This is because many of our enhancement are designed to explore the use of new technology provided by WebSphere Application Server - Express V6 rather than to meet new application requirements. Refer to the specific chapters in Part 2, "Development examples" on page 221 for more details about these technical investigations.

5.3.1 Bidding system

The bidding system will be implemented in accordance with the application design principles. It will be a Struts-based component using JSPs for the presentation layer. Struts actions will be used to control application flow.

The business logic and persistence layers will be implemented via the delegate and manager patterns. A new database table will be needed to store the billing history. In short, all application layers will need to be implemented.

Presentation layer

The viewProperties.jsp displays the results of a property search. An additional column will need to be added to the search results table. This column, called Bids, will take the user to the bid history page for the property. This page will show the filtered bid history.

To follow the requirements, we will implement the following:

► If the user is not logged in, no Bids column is shown.

- ► If the user is the seller, an administrator, or the agent selling the property, the bids link is shown. This is regardless of the state of the property. This allows those in these roles to see bid history for sold properties.
- ► If the user is a buyer and has bid on a property, then the bids link is shown.

A new JSP will be required that displays the bid history. This page is inherently linked to a property and some property details will be displayed.

The customer will place a bid on a new Add Bid page. If a customer places a bid on a property, he or she will be sent back to the property search page. The page will have been updated by repeating the previous search.

The presentation of bid fields on the bid page should be determined from the currently selected property DTO.

The property search page will be enhanced to show two new buttons. These will allow a seller to search for all of their listed properties (in any state), and will allow a bidder to search for all of the properties on which they have bid.

A Struts action that manages bidding will need to be created. It is desirable to place the new Struts forms and actions in their own Struts module.

Business logic layer

A session data class for the property component does not exist and will need to be created.

The PropertyCatalogManager will need to be enhanced so that on a search, the biddable flag is returned correctly. This will also require the addition of the biddable (the getter will be isBiddable) to the property DTO. The manager will also need to be enhanced to include the bidding history. The bidding history should also have an attribute on it indicating the user role with respect to the bidding history.

The major addition to the business logic layer will be the methods and logic that forms the biddable flag and the filtered bid list.

The bid manager should not cache any bid history. If a user refreshes a page or performs another search, the bid statuses and history should be updated accordingly. This will prevent the user from having to log out and log back in to see updated bidding information.

The property search criteria will need to include the capability to search for properties for sale by a user. This can be done by consulting only the property tables. The search criteria will also need to support finding a list of properties that

the user has bid upon. This will be done with a query to the bidding system to return a list of property IDs that the user has bid upon.

A new manager will need to be created to manage bids. This will use a session data class facade even though the bidding system is stateless.

Create bid
When creating a bid, an error can be returned indicating that the property is not available for bidding. This can occur when several customers attempt to bid on the same property simultaneously.

This method will need to be passed to the property ID, the customer ID, and the bid amount.

Change bid status
This method will allow the bid status to be modified. It will require the bid ID and the new status. The only status transition allowed by the requirements is from the open state to either the rejected or accepted state.

Persistence layer
A new table will need to be created for the bidding history. This table will contain, at a minimum, the bid amount and bid status and the user ID that placed the bid and property ID that has been bid upon.

5.3.2 Reference data component

Reference data is used extensively in an application. The data can also be quite large. The largest reference data element in the current application is the country code list. The Sal301 application, being a Web application only, stored this reference data in the HTTP session. For small, not too busy, Web sites this approach works well. If, however, the reference data is very large or there are many concurrent sessions, storing this information in the HTTP session results in an excessive resource consumption in the Web container.

In order to address the resource consumption issue, the reference data component is implemented with stateless session beans. The J2EE container will maintain a pool of reference data beans and allocate one to the Web container when the Web container needs to build a JSP that contains reference data. The initial reference data component has been implemented by removing the property type and status lists and the country code lists methods from the property catalog manager. These have been refactored into a new class, the ReferenceDataHelper.

The ReferenceDataHelper has been exposed as an EJB. exposed by a manager, and then placed into the HTTP session using the ReferenceSessionData class.

5.3.3 Session management

The session management provided by our application is relatively simple. There are two main components. These two components initialize the session when a new session is created and prevent the user from viewing certain pages when they have not logged in.

Session initialization is performed by a session listener. The session listener is registered in the web.xml file as shown in Figure 5-2.
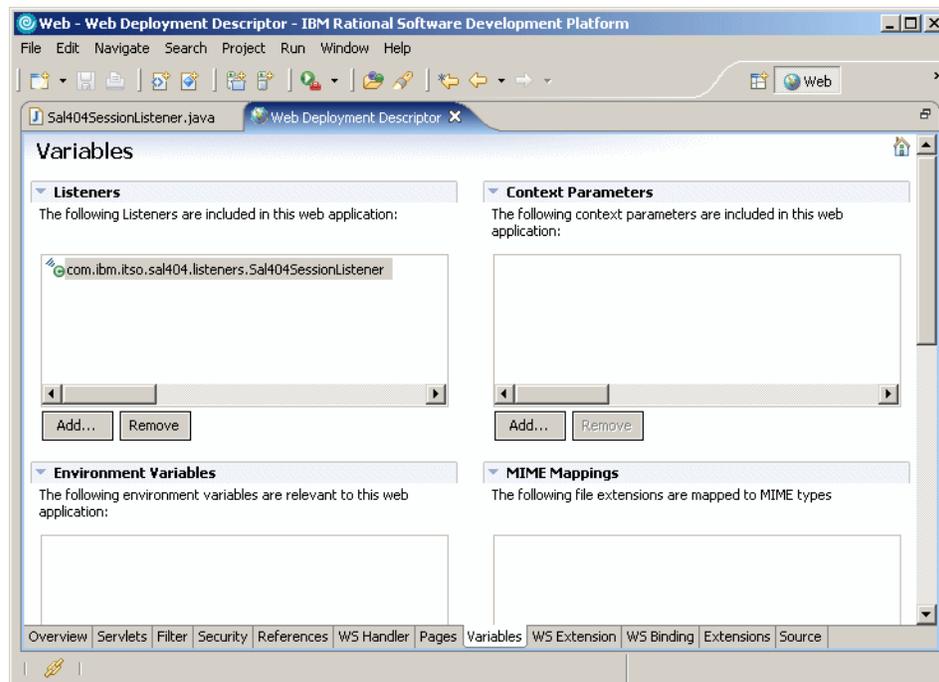
Figure 5-2   Session listener registration

The code for the session listener is shown in Example 5-1:

Example 5-1   Sal404SessionListener

```
package com.ibm.itso.sal404.listeners;

import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
```

```
import org.apache.log4j.Logger;

import com.ibm.itso.sal404.referencedatacomponent.session.ReferenceSessionData;

public class Sal404SessionListener implements HttpSessionListener {
    // Configure Log4J Logger
    private static Logger logger =
Logger.getLogger(Sal404SessionListener.class);

     /* (non-Java-doc)
      * @see java.lang.Object#Object()
      */
    public Sal404SessionListener() {
        super();
    }

    /* (non-Java-doc)
     * @see sessionCreated(HttpSessionEvent arg0)
     */
    public void sessionCreated(HttpSessionEvent arg0) {
        logger.info("ENTRY: sessionCreated");
        arg0.getSession().setAttribute("rdd", new ReferenceSessionData());
        logger.info("EXIT:  sessionCreated");
    }

    /* (non-Java-doc)
     * @see sessionDestroyed(HttpSessionEvent arg0)
     */
    public void sessionDestroyed(HttpSessionEvent arg0) {
    }

}
```

The implementation shown in Example 5-1 on page 217 places the
ReferenceSessionData class into the session whenever a session is created. As
a result, the session will always exist, and the ReferenceSessionData object will
always be available.

In the earlier version of our sample written for *WebSphere Application Server -
Express: A Development Example for New Developers*, SG24-6301, objects
were placed into the session in the Struts actions. One of the goals of our new
redbook application is to centralize session management so that it is consistently
done and can be maintained in one place.

5.3.4 Session data

The redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 used the concept of managers to provide services for the Sal301 application. These managers are very functional and provide no object-oriented behavior. They expose CRUD functionality and little more.

The Sal301 application is a simple example for entry level developers. The application is quite simple and the development model presented is also quite simple. The JSPs in the Sal301 application read Java classes from the HTTP session and populate their various fields with data from these classes. The Struts Action classes process the input forms, call the appropriate managers and data returned from the managers is then placed in session scope. This is sufficient for a simple application, but the session does tend to get cluttered up with various beans. In this redbook we create session data objects to begin addressing these issues.

When we look at the at the managers (UserManager, PropertyCatalogManager, Newsmanager, and InterestListManager) in the Sal301 application, we see that they all do share some common functions and behaviors. They all manage lists of objects, there is the concept of the currently selected object, and so forth. Similarly you can see that our Sal301 views also need to be aware of concepts such as the currently selected object and also to handle list of objects. In the Sal301 application such data was commonly stored in the HTTP session in the form of DTOs and vectors of DTOs.

As a result we decided to implement session data classes in the Sal404 to more effectively manage the data stored in our session. The session data classes are created by the session listener as shown in Example 5-1 on page 217. This means that we can be sure they always exist in our session and that they are consistently named. By centralizing the session data creation we can also be sure that multiple copies of the same data are not being stored.

We created a session data class for many of the major components of our application. The classes we created are in the SAL404RealtyControl project are:

► UserSessionData
► ReferenceSessionData
► NewsSessionData
► BidSessionData

The session data classes typically share common attributes and behavior. They usually have:

► An attribute to hold an instance of the manager for their component

For example the BidSessionData class has a bidManager attribute while the NewsSessionData class has a newsManager attribute.

► An attribute called current to hold a DTO for the current selected data

► A list attribute to hold a vector of DTOs

For example the BiddSessionData class has a bidList attribute that holds a vector of BidDTO objects, while the NewsSessionData class has a newsList attribute that holds a vector of NewsDTO objects.

► Methods to create, update, and delete data for their component

These methods are wrappers around a call to the appropriate methods on the manager for the component.

Part 2

Development examples

In this part we describe our redbook sample solution, and detail the steps to design and develop examples that illustrate new features and function in WebSphere Application Server - Express Version 6 and in the Rational Software Development Platform.

6

Web site development

In this chapter we provide a brief overview of the Web development tools provided with Rational Web Developer. Because our sample application was already developed, we were not required to build many new Web pages during this redbook project. As a consequence this chapter does not provide a detailed guide to all the Web development features of Rational Web Developer. Our main focus is to provide a guide to some of the more important tools and to direct you to where you can find more information.

We do provide some details on how to use templates with our sample application because this was a major enhancement we made.

Note that our overview of Rational Web Developer tools for Web development is a summary of material that can be found in greater detail in the redbook *Rational Application Developer V6 Programming Guide*, SG24-6449.

223

6.1 Introduction to Web applications

There are many Web development technologies as well as tools included in Rational Web Developer. In this chapter we do not discuss all the technologies available to us. Our redbook concentrates only on the tools and technologies we need in our sample application. For an overview of our use of Struts see Chapter 11, "Struts" on page 447. For an examination of how to use JSF with our existing sample see Chapter 7, "JavaServer Faces" on page 239. Our focus in this chapter is on an overview of the tools for developing dynamic Web applications using JavaServer Pages (JSP) and Java Servlets technology, and for static Web sites using HTML.

6.1.1 Concepts and technologies

This section provides an overview of the concepts and technology used by J2EE Web applications.

WebSphere Application Server - Express is based on a number of key technologies that are discussed in this section. We also assume that you have a basic understanding of Internet technologies and Web applications. It is useful to have a working knowledge of:

► Web (HTTP) servers

  Some details to get you started on this topic can be found here:

  – http://www.serverwatch.com
  – http://httpd.apache.org/

► HTML

  To get started see: http://www.w3.org/MarkUp/

► HTTP

  To get started see http://www.w3.org/Protocols/

► Java Servlets

  To get started see:

  – http://java.sun.com/products/servlet/
  – http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html

► JavaServer Pages

  To get started see:

  – http://java.sun.com/products/jsp/product.html
  – http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPIntro.html

▶ JavaBeans

  To get started see:

  – http://java.sun.com/products/javabeans/
  – http://java.sun.com/docs/books/tutorial/javabeans/index.html

Java 2 Platform, Enterprise Edition

WebSphere Application Server - Express fully supports the Java 2 Platform, Enterprise Edition (J2EE) specification. For more details on Java 2 Platform, Enterprise Edition see:

▶ http://java.sun.com/j2ee/
▶ http://java.sun.com/j2ee/tutorial/

Java Servlets and JavaServer Pages

JavaServer Pages have quickly become the presentation mechanism's most popular technology because it enables developers of varied skills to create dynamic and rich content very easily. The flexibility of the JSP to combine both a markup language and an enterprise programming language makes it easier for developers with different skill sets to collaborate on projects.

Java Database Connectivity

Accessing the database has always been an extremely important part of enterprise application services. The Java Database Connectivity (JDBC) interface makes it very easy to access disparate databases from multiple vendors. In addition, JDBC enabling wizards inside of Rational Software Development Platform alleviate the need for developers to think about the infrastructure-level code of accessing the database.

XML

Internally, XML is the glue that unites the WebSphere architecture and enables the technology to work together. Indeed, its simple design and extensible nature for describing just about anything make it an extremely powerful language. It is an essential ingredient to dynamic e-business applications. Unfortunately, the language can sometimes be difficult to manage. Rational Software Development Platform shields developers from the sometimes difficult nature of the language with a beautiful and friendly facade, allowing them to leverage every bit of its power.

Static and dynamic Web application technologies

Web technologies can be described as either static or dynamic, depending on whether the page displayed to an individual user is constant or generated at the time it is served.

Static Web applications

A *static* Web site is one in which the user's Web browser accesses content determined only by the contents of the file system on the Web server machine. Because the user's experience is determined only by the content of these files and not by any action of the user or any business logic running on the server machine, the site is described as static.

In most cases, the communication protocol used for interacting with static Web sites is the Hypertext Transfer Protocol (HTTP).

HTTP follows a request/response model. A client sends an HTTP request to the server providing information about the request method being used, the requested Uniform Resource Identifier (URI), the protocol version being used, various other header information and often other details, such as details from a form completed on the Web browser. The server responds by returning an HTTP response consisting of a status line, including a success or error code, and other header information followed by a the HyperText Markup Language (HTML) code.

Dynamic Web applications

Dynamic Web applications are applications that are accessed using HTTP (Hypertext Transfer Protocol), usually with a Web browser as the client-side user interface to the application. The flow of control logic, business logic, and generation of the Web pages for the Web browser are all handled by software running on a server machine. Many different technologies exist for developing this type of application, but our redbook sample uses the Java technologies provided by our Java 2 Platform Enterprise Edition application server which is WebSphere Application Server - Express.

Enterprise application

An enterprise application project contains the hierarchy of resources that are required to deploy a Java 2 Platform, Enterprise Edition (J2EE) application. It can contain a combination of Web modules, EJB modules, JAR files, and application client modules. It includes a deployment descriptor and an IBM extension document, as well as files that are common to all J2EE modules that are defined in the deployment descriptor. It can contain a complete application that might be a combination of multiple modules. Enterprise applications make it easier to deploy and maintain code at the level of a complete application instead of as individual pieces.

There are a couple methods of creating an Enterprise Application using Rational Web Developer:

- ▶ Create New Enterprise Application wizard

  This wizard can by started by selecting **File** → **New** → **Project. Select J2EE** → **Enterprise Application**.

- ▶ Create the enterprise application as part of creating a new Web Project.

- ▶ Create enterprise application using Import wizard.

  If you are importing a Web project, you can create an enterprise application with the import wizard.

Enterprise Application projects are exported as enterprise archive (EAR) files that include all files defined in the enterprise application project as well as the appropriate module archive file for each J2EE module project defined in the deployment descriptor, such as Web archive (WAR) files and EJB JAR files.

An enterprise application can contain JAR files to be used by the contained modules. This allows sharing of code at the application level by multiple Web or EJB modules.

The enterprise application deployment descriptor contains information about the components that make up the enterprise application. This deployment descriptor is called application.xml and is located under the META-INF directory.

Web application

The Java Servlets specification 2.4 and the J2EE specification contain the concept of a Web application. A Web application contains JavaServer Pages, Java Servlets, applets, Java classes, HTML files, graphics, and descriptive meta information that connects all these elements. The format is standardized and compatible between multiple vendors.

The specification also defines a hierarchical structure for the contents of a Web application that can be used for deployment and packaging purposes. Many servlet containers, including the one provided by Express Application Server, support this structure.

Any Web resource can be included in a Web application, including the following:

- ▶ Servlets and JavaServer Pages
- ▶ Utility classes

  Standard Java classes may be packaged in a Java archive (JAR) file. JAR is a standard platform-independent file format for aggregating files (mainly Java classes files).

- ► Static documents

  HTML files, images, sounds, videos, and so forth come under this category. This term includes all the documents a Web server is able to handle and to provide to client requests.

- ► Client side applets, beans, and classes
- ► Descriptive meta information tying all of the above elements together
- ► Custom tag libraries
- ► Struts
- ► XML files
- ► Web services

The directory structure for a Web application requires the existence of a WEB-INF directory. This directory contains Java and support classes that contain application logic. Access to these resources is controlled through the servlet container, within the application server.

Figure 6-1 shows an example of a typical directory structure under the WEB-INF directory.
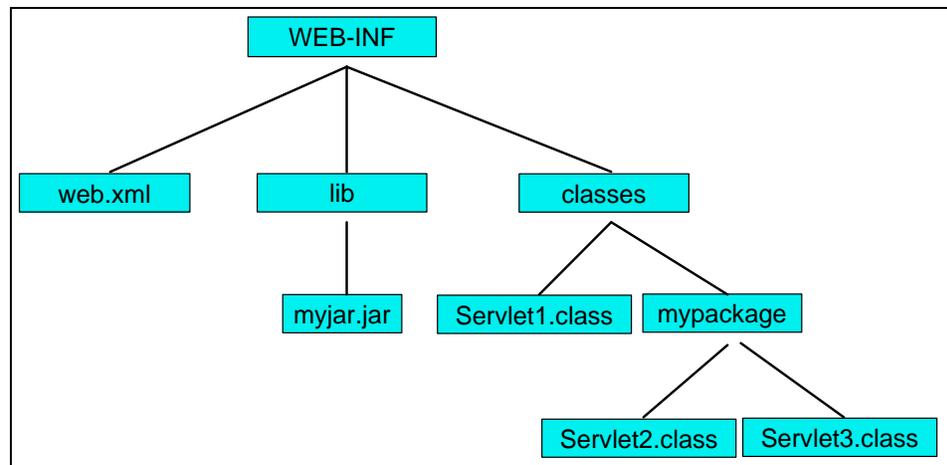
Figure 6-1   The WEB-INF directory: A sample structure

The required elements are:

- ► web.xml: This file is required and is the deployment descriptor for the Web application.

► lib: This directory is required and is used to store all the Java classes used in the application. This directory will typically contain JAR files, including tag libraries.

► classes: This directory is also required. Typically, servlet classes and utility classes for the servlets compose this directory. It is possible to keep a package structure in this directory and to put class files under several subdirectories of the classes directory (as it is done for the Servlet2.class file in the subdirectory mypackage in Figure 6-1 on page 228).

Although there are no other requirements for the directory structure of a Web application, we recommend that you organize the resources in separate logical directories for easy management, an images folder to contain all graphics, for example.

As with the enterprise application, a deployment descriptor exists for the Web application. The Web deployment descriptor, web.xml, contains elements that describe how to deploy the Web application and its contents to the servlet container within the Web server. Note that JSPs execute as servlets and are treated as such in the Web deployment descriptor.

The deployment descriptor file enables the application configuration to be specified independently from the server. It clearly simplifies the deployment process because the same application can be deployed into different servers without having to review its content.

6.1.2 Web development tooling

Rational Application Developer includes many Web development tools for building static and dynamic Web applications. In this section, we highlight the following tools and features:

► Web perspective and views
► Web projects
► Web Site Designer
► Page Designer
► Page templates
► CSS Designer
► Javascript Editor
► WebArt Designer
► Animated GIF Designer
► File creation wizards

These tools as well as other tools are further illustrated in the examples found throughout the chapter.

6.1.3 Web perspective and views

Web developers can use the Web perspective and supporting views within Rational Application Developer to build and edit Web resources, such as servlets, JSPs, HTML pages, style sheets and images, as well as the deployment descriptor files.

The Web perspective can be opened by selecting **Window** → **Open Perspective** → **Web** from the Workbench. Figure 6-2 displays the default layout of the Web perspective with a simple home.html open.
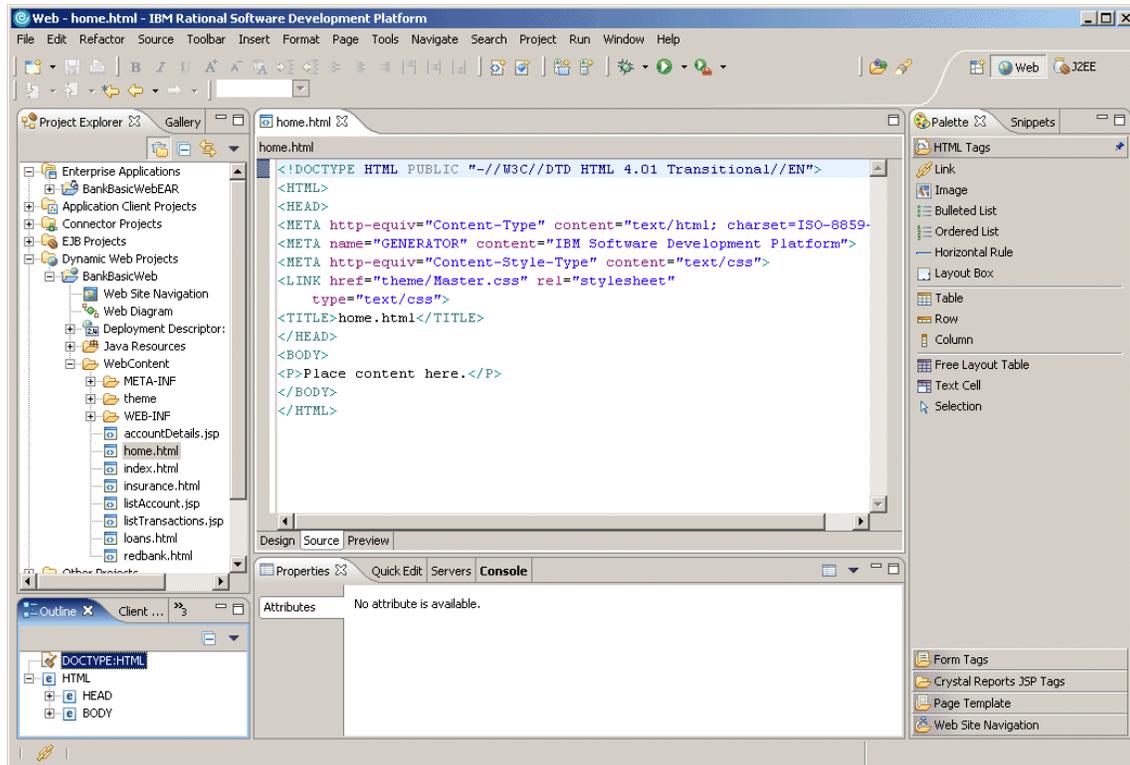
Figure 6-2   Web perspective

6.1.4 Web projects

In Rational Web Developer, you create and maintain Web resources in Web projects. They provide an environment that enables you to perform activities such as link-checking, building, testing, and publishing. Within a Web project, Web resources can be treated as a portable, cohesive unit.

Web projects can be static or dynamic. Static Web projects are comprised solely of static resources, which can be served by a traditional HTTP server (HTML files, images, and so on), and are useful for when you do not have to program any business logic. J2EE Web projects, on the other hand, can deliver dynamic content as well, which gives them the ability to define Web applications.

A Web application contains components that work together to realize some business requirements. It might be self-contained, or access external data and functions, as is usually the case. It is comprised of one or more related servlets, JavaServer Pages and regular static content, and is managed as a unit.

6.1.5 Web Site Designer

The Web Site Designer is provided to simplify and speed up the creation of the Web site navigation and creation of HTML and JSP pages. You can view the Web site in a Navigation view to add new pages, delete pages and move pages in the site. The Web Site Designer is especially suited for building pages that use a page template.

The Web Site Designer is used to create the structure for your application in much the same way you would create a book outline to serve as the basis for writing a book. You use the Web Site Designer to visually lay out the flow of the application, rearranging the elements (JSPs, HTML pages) until it fits your needs. You continue by creating pages based on this design.

As you build your Web site design, the information is stored in the website-config.xml file so that navigation links and site maps can be generated automatically. This means that when the structure of a site changes, for example when a new page is added, the navigation links are automatically regenerated to reflect the new Web site structure.

The Web Site Designer can be used with existing or new projects. To create a Web site configuration for a Web project that does not currently have one, there is an option on the context menu of the Web project, **Convert to Web Site** in the Web perspective.

To launch the Web Site Designer, double-click **Web Site Navigation** found in the root of your Web project folder. Figure 6-3 on page 232 displays a sample Web site navigation and pages in Web Site Designer.
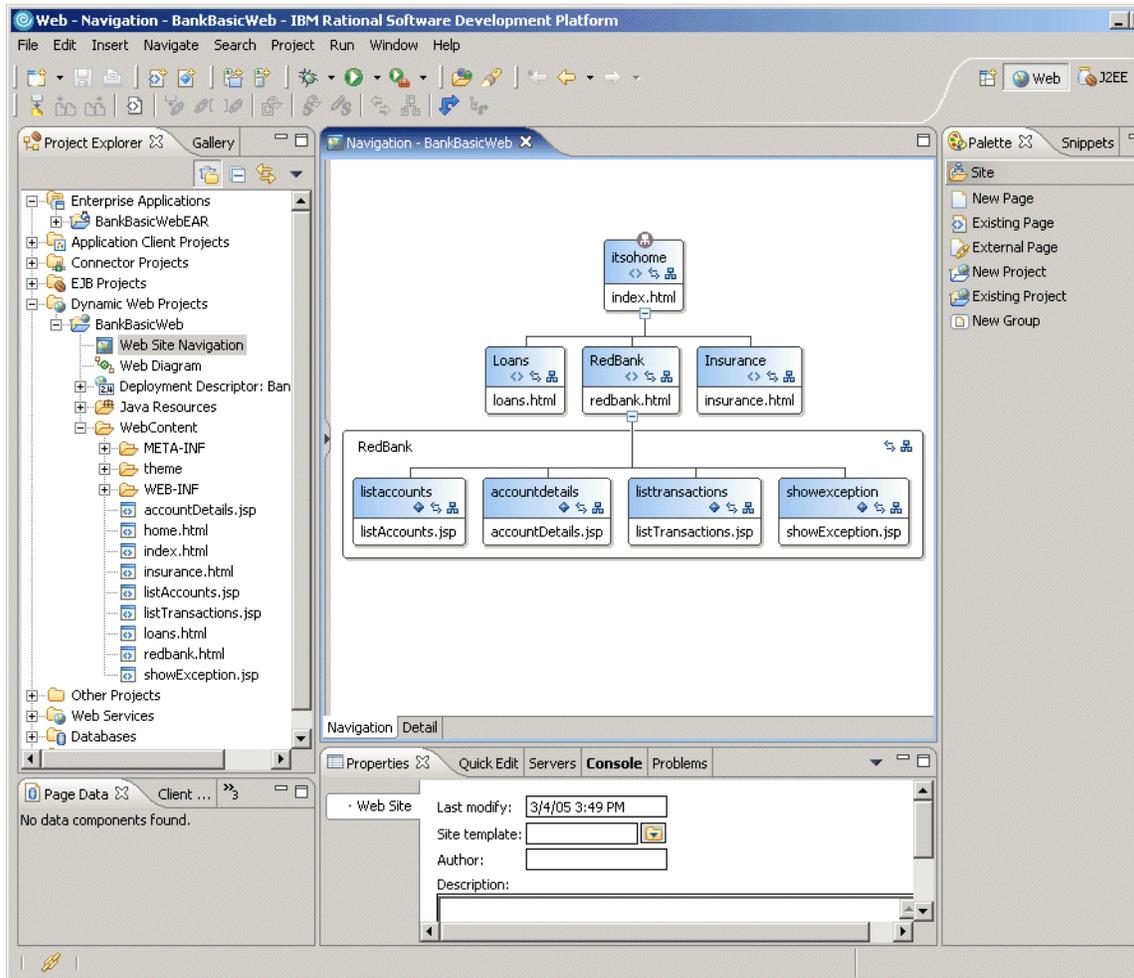
Figure 6-3   Web Site Designer

6.1.6 Page Designer

Page Designer is primary editor for developing HTML, XHTML, JSPs, and Faces JSP source code. It has three representations of the page, including Design, Source, and Preview. The Design tab provides a WYSIWYG environment to visual design the contents of the page. As its name implies, the Source tab provides access to the page source code. The Preview tab shows what the page would like if displayed in a Web browser.

6.1.7 Page templates

A *page template* contains common areas that you want to appear on all pages, and content areas that will be unique on the page. They are used to provide a common look and feel for a Web project.

Use the Page Template File creation wizard to create the file. Once you create it, you can modify the file in the Page Designer. The page templates are stored as *.htpl for HTML pages and *.jtpl files JSP pages. Changes to the page template will be reflected in pages that use that template. Templates can be applied to individual pages, groups of pages, or applied to an entire Web project. You can mark areas as read-only, thus Page Designer will not allow the user to modify those areas.

6.1.8 CSS Designer

Style sheets can be created when you create the Web project, or they can be added later. It is a good idea to decide on the overall theme (color, fonts, and so forth) for your Web application in the beginning and create the style sheet at the start of the development effort. Then as you create the HTML and JSP files, you can select that style sheet to ensure that the look of the Web pages will be consistent. Style sheets are commonly kept in the WebContent/theme folder.

The CSS Designer is used to modify cascading style sheet *.css files. The changes are immediately applied to the Design page in Page Designer, if the HTML file is linked to the CSS file.

6.1.9 Javascript Editor

The Javascript Editor provides a Source page and Preview page to enable you to work with source files and view them as though in a Web browser. The Snippets palette includes Javascript code that you can drag and drop into your Web pages.

6.1.10 WebArt Designer

Use the WebArt Designer program to create and edit image files. Using WebArt Designer, you can create shape objects, draw a simple map, as well as create logos and buttons often seen on Web pages. The Page Designer also enables you to edit GIF and JPEG images, but WebArt Designer offers a much richer set of functionality for editing images.

6.1.11 Animated GIF Designer

An animated GIF is a series of image files in GIF format, displayed sequentially to give the appearance of an animation. You can insert an animated GIF into a page in the same way as a regular GIF image file. Animated GIFs can be viewed on a regular Web browser without any special plug-ins.

The AnimatedGif Designer is a program for creating animated GIF files, and comes with a gallery of predefined animation files. With the AnimatedGif Designer, you can:

► Combine several images to create an animation.
► Apply an animation effect on a single image to create an animation.
► Apply an animation effect on text to create an animated banner.

Launch the AnimatedGif Designer by clicking **Tools** → **AnimatedGif Designer** from the menu bar.

6.1.12 File creation wizards

Rational Application Developer provides many Web development file creation wizards by selecting **File** → **New** and then selecting the wizards as seen in Figure 6-4.
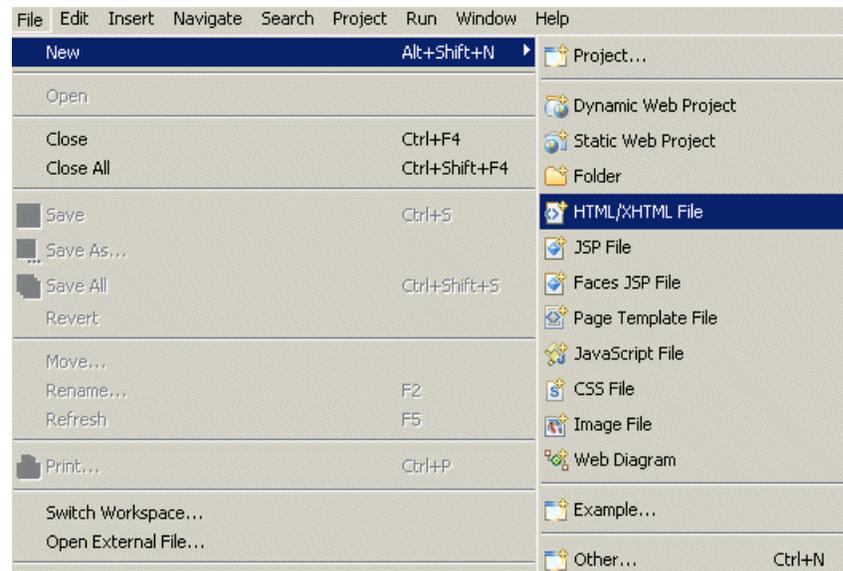
Figure 6-4   File creation wizard selection

HTML file wizard

The HTML File wizard creates an HTML file in a specified folder, with the option to create from a page template. In addition, the markup language included can be defined as type, HTML, HTML Frameset, Compact HTML, XHTML, XHTML Frameset, and WML 1.3.

JSP file wizard

The JSP File wizard creates a JSP file in a specified folder, with the option to create from a page template or create as a JSP Fragment, and define the markup language (HTML, HTML Frameset, Compact HTML, XHTML, XHTML Frameset, and WML 1.3). In addition, you optionally have the ability to selecting a model (none, JSP, Struts Portlet JSP, Struts JSP).

Faces JSP file wizard

This wizard is similar to the JSP file wizard in capability. It has a different set of models. This wizard is covered in more detail in Chapter 7, "JavaServer Faces" on page 239.

Page Template file wizard

The Page Template file wizard creates new page template files in a specified folder, with the option to create from a page template or create as a JSP Fragment, and define the markup language (HTML, HTML Frameset, Compact HTML, XHTML, XHTML Frameset, and WML 1.3). Optionally, you can create a new page template from an existing page template. In addition, you can select from one of the following models; Template contains Faces Components, Template containing only HTML, Template containing JSP.

Javascript file wizard

The Javascript file wizard creates a new Javascript file in a specified folder.

CSS file wizard

The CSS file wizard creates a new cascading style sheet (CSS) in a specified folder.

Image file wizard

The Image file wizard creates new image files (bmp, mif, gif, png, jpg) in a specified folder.

6.1.13 Our sample Web site project

With Rational Web Developer we can create different project types, including:

- ► Enterprise Application project
- ► J2EE Application Client project
- ► Dynamic Web project
- ► Static Web project
- ► EJB project
- ► Connector project
- ► Java project
- ► Simple project
- ► Server project
- ► Component Test project
- ► EMF Project
- ► Checkout Projects from CVS
- ► Feature patch
- ► Feature project
- ► Fragment project
- ► Plug-in project
- ► Update Site project

Our redbook sample application incudes a Dynamic Web project called SAL404RealtyWeb. The main enhancement we have made to our Web application for this redbook is to use Web page templates. This technology was not available to us when the previous redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 was written.

> **Note:** For additional information about project types see the redbook *Rational Application Developer V6 Programming Guide*, SG24-6449.

Why we use templates

The advantages of using templates in our Web project include:

- ► You have the ability to update a group of Web pages simultaneously and automatically, simply by updating a page template and saving it.

- ► You get decreased time for development and maintenance (speed).

- ► You can change the corporate image for a entire site with minimal modifications.

- ► You can provide a new presentation for the site without affecting site functionality.

- ► You can provide standardized presentation for the final user.

- ► You manage common contents of multiple Web pages (such as header, footer, and menu sections) using a separate file called a page template file.

- ► You can manage layouts of multiple Web pages.

- Templates help designers keep in mind both aesthetic and technical points of view. Templates are functional and easy-to-edit, but at the same time, they support originality and artistic characteristics consistent with the designer's style.

- Templates provide a shell around which you can quickly build a professional Web site.

- They help ensure that your site's pages are consistent in appearance, navigation, and overall design.

- They are available with custom graphics and logos already in place.

- They help novice Web masters create an attractive, functional site that visitors will enjoy using.

Templates in Rational Web Developer

With Rational Web Developer you can create a new HTML, XHTML, JSP, or Faces JSP page using Page Designer. When creating these pages you can using a template. Sample templates are supplied with Rational Web Developer and you can create your own custom templates.

For our redbook sample applications, we created page templates to use with our Struts application. These templates replaced the JSP include approach used in the SAL301R application developed for the *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 redbook.

Details of how we created and used templates in our Struts example application can be found in Chapter 11, "Struts" on page 447.

We also used templates to provided a common look and feel to our JSF examples. For more details see Chapter 7, "JavaServer Faces" on page 239.

7

JavaServer Faces

This chapter describes and presents the use of JavaServer Faces (JSF) and Service Data Objects (SDO) technologies for a subset of the realty application.

We briefly describe JSF and compare it with Struts, then we implement a home page, the about us page, and the news application using JSF and SDO.

JSF is part of J2EE 1.4. Therefore, the runtime for JSF is included in the WebSphere Application Server - Express V6.0.

7.1 Introduction to JSF

JavaServer Faces (JSF) is a framework for developing Java Web applications. The JSF framework aims to unify techniques for solving a number of common problems in Web application design and development, such as:

► User interface development

JSF has the capability of directly binding user interface (UI) components to model data. JFS abstracts request processing into an event-driven model. Developers can use extensive libraries of prebuilt UI components that provide both basic and advanced Web functionality. UI components are Java objects residing on the server.

► Navigation

JSF introduces a layer of separation between business logic and the resulting UI pages; stand-alone flexible rules drive the flow of pages.

► Session and object management

JSF manages designated model data objects by handling their initialization, persistence over the request cycle, and cleanup.

► Validation and error feedback

JSF has the capability of direct binding of reusable validators to UI components. The framework also provides a queue mechanism to simplify error and message feedback to the application user. These messages can be associated with specific UI components.

► Internationalization

JSF provides tools for internationalizing Web applications, supporting number, currency, time, and date formatting, and externalizing of UI strings.

JSF is 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.

7.1.1 Model-view-controller architecture

Applications built with JavaServer Faces are intended to follow the model-view-controller (MVC) architectural pattern (Figure 7-1 on page 241). According to the MVC pattern, a software component should separate its business logic along the following lines:

Model   Encapsulates the state and behavior of the application.
View   Renders the model.
Controller   Processes user events and drives model and view updates.
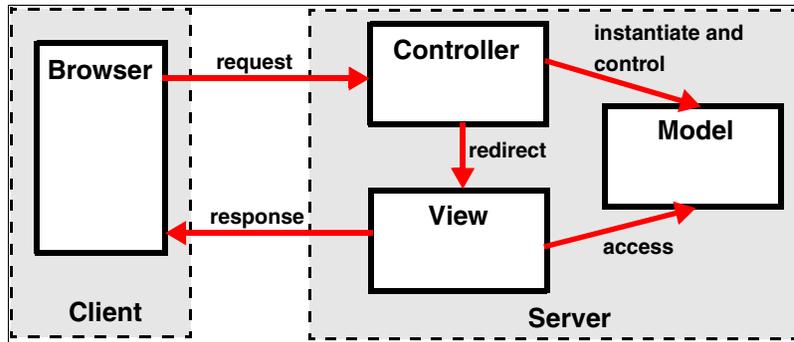
Figure 7-1   MVC Web application architecture

We can group the important components of JSF as belonging to these categories, as in Figure 7-2:

► Model

    Managed beans make up the model of a JSF application. These Java beans typically interface with reusable business logic components or external systems, such as a mainframe or database.

► View

    JSPs make up the view of a JSF Web application. These JSPs are created by combining model data with predefined and custom-made UI components.

► Controller

    The FacesServlet, which drives navigation and object management, makes up most of a JSF application's controller. Event listeners also contribute to the controller logic.
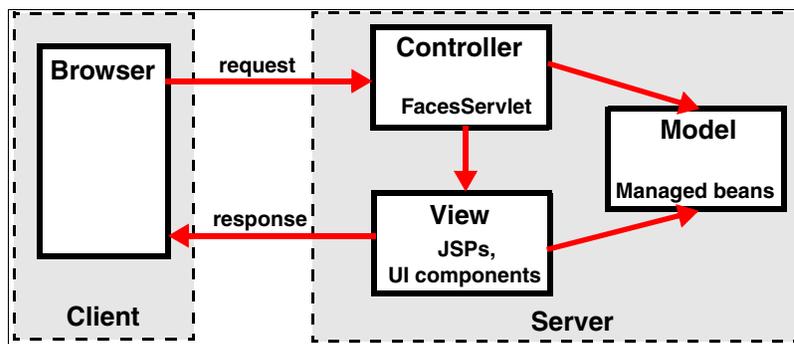
Figure 7-2   JSF's MVC breakdown

7.1.2 JSF Web application structure

The structure of a JSF Web application is shown in Figure 7-3.
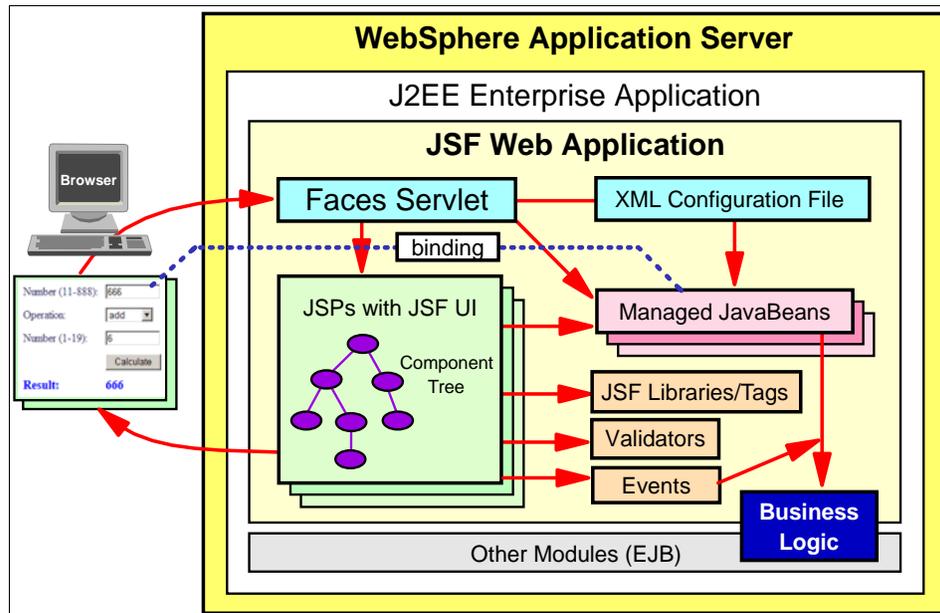
Figure 7-3   JSF Web application structure

A JSF Web application consists of:

► Faces servlet, a servlet provided by JSF

► An XML configuration file (faces-config.xml) holding information about supported languages, managed beans, navigation rules, validators, and converters

► JSPs built from JSF components

Each JSP is a tree of components and each components is a server-side Java object.

► JSF tag libraries, which support the JSP components

► Validators used to validate user input data

► Managed JavaBeans holding hold the application data

Properties of managed beans are bound to user interface components (for example, input and output fields). JSF moves and converts the data between UI components and managed beans.

- ► Events invoking logic in managed beans

  Typical events are command buttons and hyperlinks. In addition changes to values in UI components can trigger value-changed events.

- ► Business logic invoked by event processing

  Action methods, invoked, for example, by command buttons typically passes managed beans as data transfer objects to business logic classes.

7.1.3 JSF support in Rational Web Developer

JSF is supported by extensive tooling in Rational Web Developer. Here is a short description of the JSF support:

- ► A Web project is automatically configured for JSF by including the Faces servlet, the tag libraries, and a style sheet to tailor JSF components.

- ► Page Designer has been extended to support JSF pages (JSPs with JSF components) through a Palette view, the Properties view, the Page Data view, and the Quick Edit view.

- ► The Palette view is used to drag JSF components into a JSF page to layout the page. IBM provides a number of custom JSF components in addition to the standard JSF components.

- ► The Properties view is used to provide the details for each component dropped into a JSF page. This includes, for example, output text, field length, data types, and standard validation.

- ► The Page Data view is used to define managed beans. Properties of managed beans can be dragged into a JSF page and a layout of input or output fields is created using JSF components. Using drag and drop, the binding of UI components to managed bean properties is accomplished. Managed beans are registered in the XML configuration file.

- ► Navigation rules specify the sequence of JSF pages. These rules are defined for command buttons and hyperlinks and they use symbolic names for JSF pages. Navigation rules are stored in the XML configuration file.

- ► The Quick Edit view is used to enter the action logic associated with command buttons and hyperlinks. Action logic must return a string result that is the symbolic name of the next JSF page. An empty string or null result redisplays the current JSF page.

- ► A server-side Java class, called *page code*, is created for each JSF page. This Java class is registered as a managed bean for action logic. Code entered in the Quick Edit view becomes a method in the page code class. The page code class provides a number of utility methods and access to all UI components and managed beans through simple get methods.

► A Web diagram shows the structure of a JSF Web application with nodes for JSF pages and lines for actions between pages.

Figure 7-4 shows the Web application with supporting tooling in Rational Web Developer.
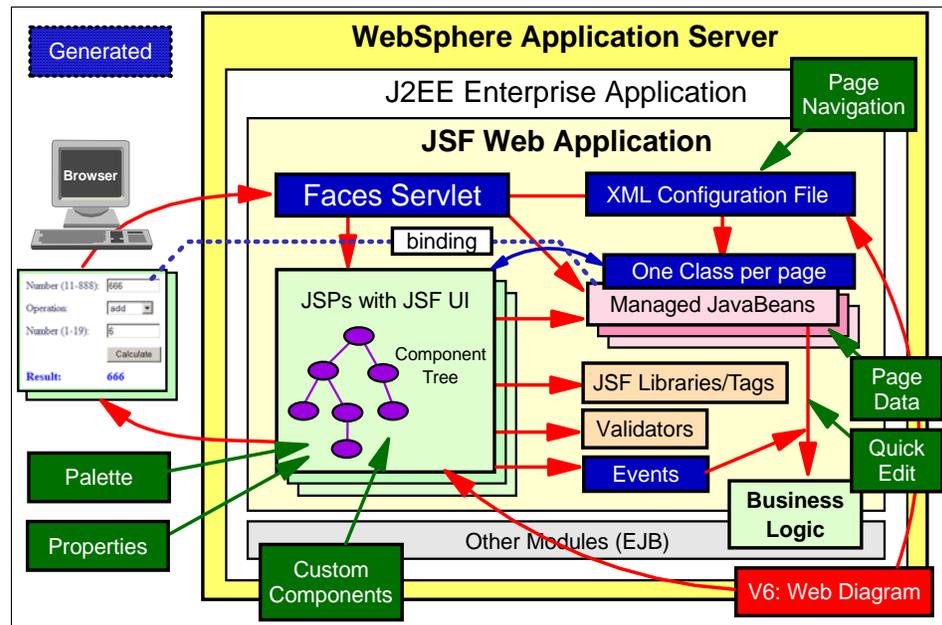
Figure 7-4   JSF Web application support by Rational Web Developer

7.2 Comparing JSF and Struts

Struts and JSF contain many of the same features, and both have a very similar overall structure. However, depending on the requirements of the application, it is useful to examine the various discrepancies between the two frameworks (Table 7-1).

Table 7-1   JJavaServer Faces and Struts feature comparison

	JavaServer Faces	**Struts**
Components	► Rich data bound UI components with events provided ► Custom components	► HTML components and a few Struts components ► Struts-specific tag library

	JavaServer Faces	Struts
Data beans	► Managed JavaBeans with properties bound to UI components ► Managed beans are independent of JSPs	► Form beans associated with input fields of JSPs
Device independence	► Reader kits that provide device independence	► None
Error-handling and validation	► Validation framework ► Predefined validators	► Validation in form beans or driven by an XML descriptor (`validation.xml`)
Scripting (logic)	► Scripts can be attached to events ► All components accessible from scripts	► Scripts written in Java action classes ► Form data, but not components, accessible
Page flow	► Simple navigation file (`faces-config.xml`) ► XML-based	► Sophisticated, flexible framework ► XML-based
Session and object management	► Automatic	► Manual

Apart from the features of the frameworks, it is important to consider the strength of their relative tools, maturity, and future directions (Table 7-2).

Table 7-2   JavaServer Faces and Struts considerations

	JavaServer Faces	Struts
Tooling	IBM tooling is relatively new, many tools soon to follow	Extensive, mature IDE support
Maturity	Relatively young, though at least one version of the specification has been ratified	Quite mature, stable, not subject to significant changes
Future plans	Lots of development vitality, future implementations planned, more extensive component libraries to be developed, and new contexts to be supported	Relatively static
J2EE support	part of J2EE 1.4	Open Source

7.2.1 Validation

The tooling is not aware of Struts validation. It is supported by the product and the user registration process in the Struts sample provides an example for using the validation framework. Even though there is no tooling support for the validation framework, using it does provide for the benefits of externalizing the validation rules and reducing the amount of code in form beans in the validate() method. Having an external definition for an account number, for example, allows the developer to state that a field must be a valid account number. The definition is in just one place and can be referenced from many forms.

> **Important:** When the tooling is said to be unaware of the validation framework, this means that not only is there no tooling support for the validation framework, but changes to the validation configuration files are not picked up automatically. The project must be republished for validation rules changes to take effect.

JSF validation is fully supported by the Rational Web Developer tooling. Validation rules can be applied in the Properties view.

Both the Struts Validation framework and the JSF validation support optional JavaScript validation in the browser.

7.2.2 XML configuration management

Struts is configured by default through the struts-config.xml, while JSF is configured through the faces-config.xml file. Managing concurrent access to these files by a development team can be very challenging.

The Struts tooling supports multiple Struts modules. Using multiple Struts modules allows for the various functional groups (user, news, property) to have separate modules. This eases the problem of concurrent access and lessons the need to merge changes.

Similarly, Struts validation also supports multiple configuration files. The default configuration file, validation.xml, can also be supplanted with additional configurations files. It would be possible to have a separate validation file that corresponded to each Struts module. Once again, this would ease concurrent access to this resource.

With Struts, a developer should be aware when a given action will modify the configuration file. Creating a new Struts action, new Struts form bean, or adding a new forward will change the Struts configuration files. The tooling in Rational Web Developer also provides for explicit modification of the Struts configuration

file. By expanding the Struts node under the Web project and selecting **new,** a developer can create a Struts artifact in the specified Struts module.

Because the tooling is unaware of the validation framework, these XML configuration files must be edited explicitly. The developer must be aware that she is modifying the file.

Even though there is multiple module support, best practice still is to make sure that the current version has been fetched from the repository. This should be modified and checked back in as soon as possible, decreasing the need for merging.

JSF manages all of its configurations through the one configuration file. These include managed beans, navigation rules, and validators. Unlike the Struts tooling, there is no way to know explicitly that the configuration file will be modified. A developer in a team environment must be aware that actions as simple as changing a validation rule or changing a navigation rule will change the `faces-config.xml`. Managing concurrent access to the `faces-config.xml` file can be quite difficult and the need to merge conflicts can be common.

Note also that there are several graphical representations for the Struts components. The Struts configuration XML editor provides an easy to use interface for browsing or editing the Struts configuration files. There is no equivalent support for JSF.

7.2.3 Templating

Templating in Struts is typically provided by Tiles. JSF does not have any explicit templating mechanism. Rational Web Developer has its own templating system that supports both Struts and JSF. Both a Struts and a JSF template are provided with the sample application.

The template tooling has many of the features of Tiles and is quite easy to use. One of the benefits of the template tooling is that it allows both the design and the preview views to work correctly. As can be seen in the Struts sample code that uses JSP includes, the tooling does not follow JSP includes. This produces render errors in both the design and preview views.

7.3 Introduction to Service Data Objects

Service Data Objects (SDO) is a data programming architecture and API for the Java platform that unifies data programming across data source types, provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data.

The Java specification request is JSR-235 and can be found here:

http://www.jcp.org/en/jsr/detail?id=235

SDO are designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and Enterprise Information Systems. The SDO architecture consists of three major components: the data object, the data graph, and the data mediator (Figure 7-5).
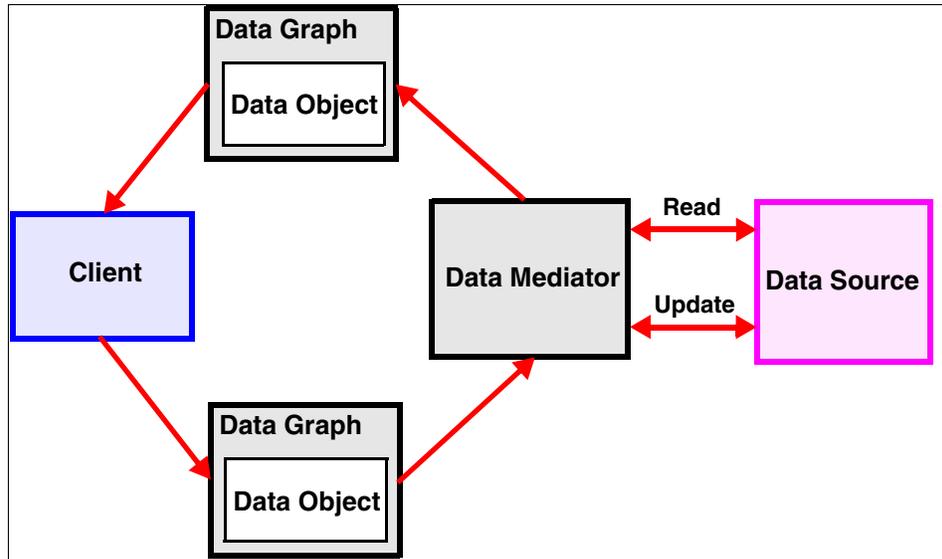
Figure 7-5   SDO architecture

Data object

The *data object* is designed to be an easy way for a Java programmer to access, traverse, and update structured data. Data objects have a rich variety of strongly and loosely-typed interfaces for querying and updating properties. This enables a simple programming model without sacrificing the dynamic model required by tools and frameworks. A data object can also be a composite of other data objects.

Data graph

SDO is based on the concept of disconnected data graphs. A *data graph* is a collection of tree-structured or graph-structured data objects. Under the disconnected data graphs architecture, a client retrieves a data graph from a data source, mutates the data graph, and then applies the data graph changes to the data source. The data graph also contains some metadata about the data

object, including change summary and metadata information. The metadata API enables applications, tools, and frameworks to introspect the data model for a data graph. Applications then handle data from heterogeneous sources uniformly.

Data mediator

The task of connecting applications to data sources is performed by a *data mediator*. Client applications query a data mediator and get a data graph in response. Client applications send an updated data graph to a data mediator to have the updates applied to the original data source. This architecture allows applications to deal principally with data graphs and data objects, providing a layer of abstraction between the business data and the data source.

7.3.1 Rational Web Developer support for SDO

Rational Web Developer does not support the complete specification of SDO. The implementation is based on WebSphere Data Objects and only supplies a data mediator for JDBC database access and for EJB access through EJB query language.

WebSphere Data Objects represents a precursor to the more broadly accepted SDO architecture. These architectures, however, are largely the same so we use the term *SDO* in this book.

In this chapter we only deal with the implementation for relational database access using JDBC. Rational Web Developer provides two data graphs for relational database access:

- ▶ Relational record, starting with one row of a table
- ▶ Relational record list, starting with multiple rows of a table

Both relational record and relational record list can extend the data of the first table through relationships (*joins*) to other tables.

Relational record

Rational Web Developer provides a special structure, called a *relational record,* to access the content of a single record in a relational database. This structure allows you to display, create, and update single records from JSF pages. A relational record can be a join record for multiple tables.

When you create a relational record, Rational Web Developer guides you through creating a database query. The tools then automatically generate a JDBC mediator that performs that query. The structure of the data graph supplied to that mediator is also stored so that you can treat the data object as a

strongly-typed, managed bean. You can bind UI components in JSF pages to the properties of this bean, thereby using the data object in a Web application.

Relational record list

In Rational Web Developer, you can define a structure to access the content of several records of a relational database. This structure is called a *relational record list*. A relational record list can be a join for multiple tables.

When you create a relational list, you define a query that returns multiple database records. The tools then act in much the same way as defining a relational record. The query and record structure are stored so that you can treat the results as a managed bean. You can bind the properties of this bean to the UI components of the JSF pages in a Web application.

Note that you can easily convert a relational record list into a relational record by using a query that only returns one record of the table.

7.4 Design of the JSF SDO sample

In this section we implement a subset of the realty application using JSF and SDO. Figure 7-6 on page 251 shows the structure of the JSF application using a Web diagram.
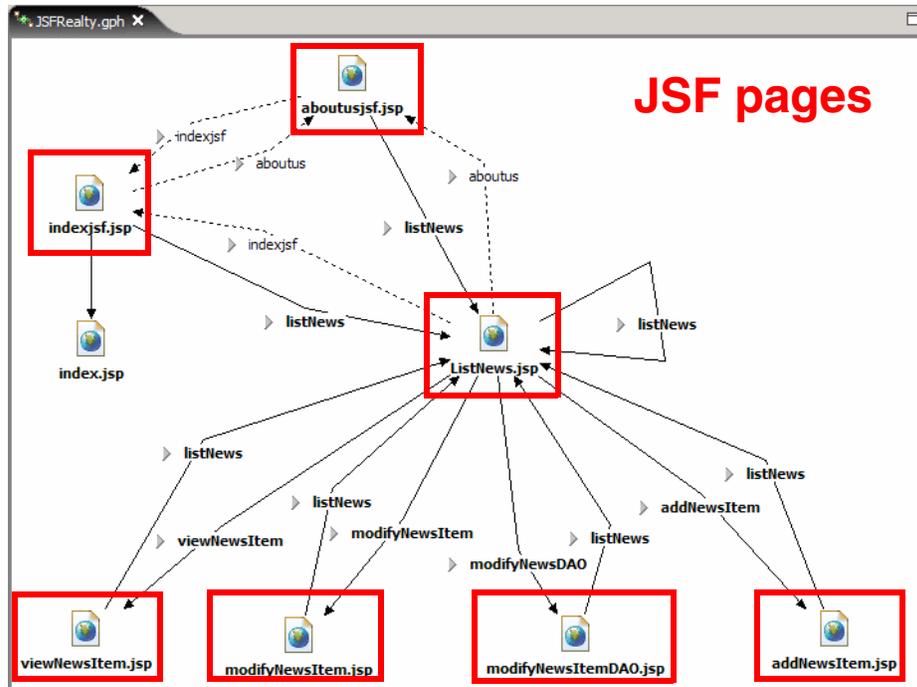
Figure 7-6   JSF application structure

The JSF application consists of these JSF pages:

- ► `indexjsf.jsp` Home page of the JSF application
- ► `index.jsp` Original home page (non-JSF) can be accessed as well
- ► `generalJSF/aboutusjsf.jsp` About Us page
- ► `newsJSF/ListNews.jsp` List of all news items using SDO
- ► `newsJSF/viewNewsItem.jsp` Details of one news item using SDO
- ► `newsJSF/addNewsItem.jsp` Form to enter a new news item using SDO
- ► `newsJSF/modifyNewsItem.jsp` Form to update a news item using SDO
- ► `newsJSF/modifyNewsItemDAO` Form to update a news item using the existing data access objects

Note that the diagram only shows the main links between the JSF pages. Through the use of a template, the home pages, the About Us page, and the ListNews page can be accessed from each page.

We use SDO to access the NEWS_ITM table. To illustrate the invocation of business logic from JSF action logic, we also implement the update page using the existing data access object (DAO).

7.4.1 JSF template

We use the template facility of Rational Web Developer to create a standard layout with links for the JSF application (Figure 7-7).

Figure 7-7   Design of the JSF template

The previous redbook, *WebSphere Application Server-Express: A Development Example for New Developers*, SG24-6301, used JSP includes to provide the template for the site layout. This has several disadvantages. Changes to the template might require updates to all existing pages that were created from a previous template version, for example. A serious drawback to this method is that the tooling does not follow JSP includes. This means that the design and preview views for a JSP will not render correctly, making these view nearly unusable.

Struts Tiles would have been a good an alternative method for page templating, but the tooling support for Tiles is limited. It also has the page preview problem. A Struts based template has been generated and can be found in the theme folder (`sal404.jtpl`).

The template for JSF uses the tooling provided in Rational Web Developer and uses JSP fragments for page links and a JSF page for the body. We can develop the fragments independently, assemble them into a template, and then apply the template to the JSF pages. The JSF page developer is not burdened with template code and can concentrate on the JSF page design.

Layout template can be created using templates supplied with the product. The template used in this application will be generated from scratch.

7.5 Implementing the JSF application

We implement the JSF application using these steps:

1. Develop the three JSP fragments
2. Develop the template using the fragments
3. Develop the home page (`indexjsf.jsp`) and apply the template
4. Develop the About Us page (`aboutusjsf.jsp`) and apply the template
5. Develop the news application (four pages) using JSF and SDO
6. Develop the news application page using JSF and DAO
7. Apply the template to the news application pages
8. Finishing the application so that only administrators can update news items

7.5.1 Creating the JSP fragments

In this section, we implement the JSP fragments similar to the Struts fragments. We use a new folder, generalJSF, for the fragments.

Head fragment

The head fragment contains the links to the About Us page and the news application (ListNews page). To create the fragment follow these steps:

1. Select the **generalJSF** folder and **New → JSP File**.

2. Enter `nav_head1` as name, select **Create as JSP Fragment**, and select **None** for the model. Click **Finish**.

3. Replace the generated code with the lines containing the links to the two JSF pages, as in Example 7-1:

Example 7-1   Creating the head fragment

```
<TABLE border="0" cellpadding="6">
   <TBODY>
      <TR>
         <td align="center" width="75">
            <A href="/SAL404Realty/faces/generalJSF/aboutusjsf.jsp">
               About Us</A></td>
         <td align="center" width="75">
            <A href="/SAL404Realty/faces/newsJSF/ListNews.jsp">
               News</A></td>
      </TR>
   </TBODY>
</TABLE>
```

4. Save and close the nav_head1.jspf file.

Side fragment

In the same way, create the nav_side1.jspf fragment containing the links to the two home pages, as in Example 7-2:

Example 7-2 Creating the nav_side1.jspf fragment

```
<TABLE width=120 cellpadding="6">
    <TBODY>
        <TR>
            <td align="center"><A href="/SAL404Realty/index.jsp">
                                    Home (non-JSF)</A></td>
        </TR>
        <TR>
            <td align="center"><A href="/SAL404Realty/faces/indexjsf.jsp">
                                    Home (JSF)</A></td>
        </TR>
    </TBODY>
</TABLE>
```

Bottom fragment

In the same way create the nav_foot1.jspf fragment containing the code to list the login information of the user, as in Example 7-3:

Example 7-3 Creating the nav_foot1.jspf

```
<TABLE class="" cellspacing="0" cellpadding="4">
    <TBODY>
        <tr>
            <TD width="120">SAL404 Realty</TD>
            <td><c:out
                value="${sessionScope.userSessionData.logInUser.userName}" />
            </td>
            <td><c:out
                value="${sessionScope.userSessionData.logInUser.role}" />
            </td>
        </tr>
        <tr>
            <td><c:out value="${confirmationMessage}" /></td>
        </tr>
    </TBODY>
</TABLE>
```

Note that this code uses the JSTL standard library to display information stored in the session data object userSessionData and in the confirmationMessage object creates by the news application.

7.5.2 Creating the template

We create the template in the `theme` folder and use the fragments for the layout. To create the JSF template follow these steps:

1. Select the `theme` folder and **New** → **Page Template File**.

2. Enter `sal404jsf` as name, select **Template containing JSP**, and select **Configure advanced options** (Figure 7-8).
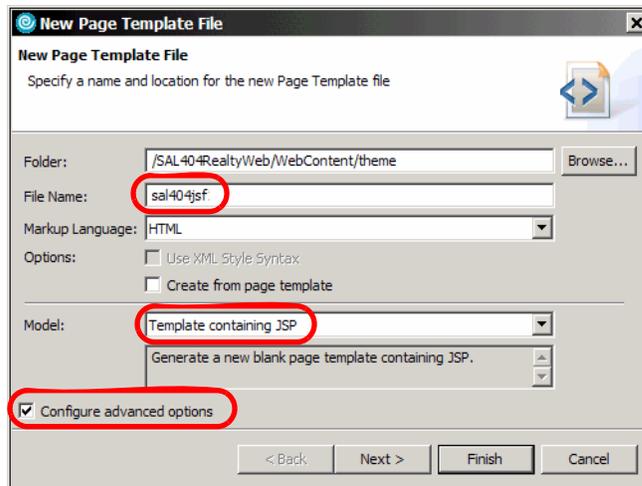
Figure 7-8   Creating the JSF template: name and mode

3. On the next page we add the required JSP tag library for JSTL. Click **Add**, locate the tag library and add it to the page (Figure 7-9).
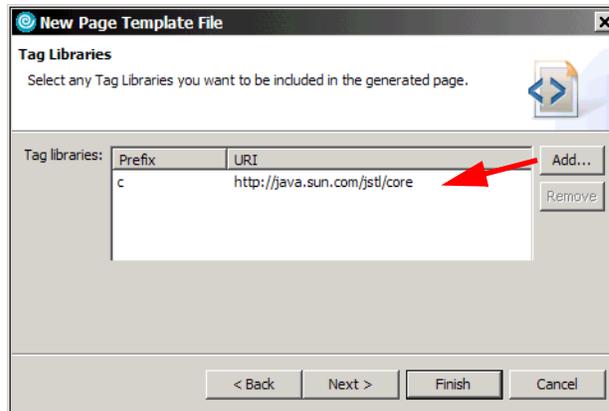
Figure 7-9   Creating the JSF template: tag library

4. Click **Finish**. A message box alerts you that you require at least one content area.

5. Delete the default text.

6. Expand the HTML Tags section in the Palette. Select **Table** and drop it into the template. Enter **3** for rows, **2** for columns, and remove the 0 from border.

7. Expand **Web Content/static/images** and drag the **realestateblue.gif** into the first table cell (Figure 7-10).
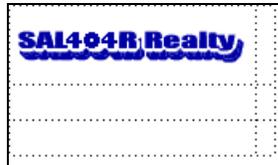
Figure 7-10   Creating the template: table and image

8. Select the table by clicking the border. and in the Properties view select **Aqua** for the color (click the button on the right).

9. Expand the **Page Template** section in the Palette, Select **Page Fragment** and drop it into the top-right cell. When prompted select the **nav_head1.jspf** fragment under **Web Content/generalJSF**.

10. Repeat this process and drop the **nav_side1.jspf** fragment into the middle-left cell and the **nav_foot1.jspf** fragment into the bottom-right cell.

11. Select **Content Area** in the Palette and drop it into the middle-right cell. When prompted, accept **bodyarea** as the name.

12. Select the cell of the body area and in the Properties view select **White** for the color. The finished design is shown in Figure 7-11.

Figure 7-11   Creating the template: fragments and body

13. Switch to the **Source** tab in Page Designer. Move the JSTL tag library in front of the `<HEAD>` tag:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<HEAD>
```

Without this change, we got errors in the footer section when applying the template to JSF pages.

14. Save and close the template.

7.5.3 Creating the home page

With the simple home page, we illustrate how to create a JSF page and apply the template to the page. Follow these steps to create the home page:

1. Select the **Web Content** folder and **New** → **Faces JSP File**.

2. Enter `indexjsf` as the name. Deselect **Create from page template** (it is easier to apply the template later) and deselect **Create as JSP Fragment**. Select **Basic** for model and click **Finish**.

3. Replace the default text with a heading 1, Welcome to SAL404 Realty:

   ```
   <H1>Welcome to SAL404 Realty</H1>
   ```

4. Expand the **Page Template** section in the Palette. Select **Apply/Replace Template**. and click into the JSF page. This only works in the Design tab. When prompted, select the **sal404jsf/jtpl** template, click **Next** and **Finish**.

Testing the home page

Make sure the SAL404Realty enterprise application is added to the server. Select the **indexjsf.jsp** and **Run** → **Run on Server**. The home page is displayed.

You can test the Home (non-JSF) link. In the old home page, click **Login**. Login as an administrator, then redisplay the JSF home page (Figure 7-12).

Figure 7-12   JSF home page without and with user login

7.5.4 Creating the About Us page

The About Us page is as simple as the home page. Create a new Faces JSP file named `aboutusjsf` in the generalJSF folder.

1. Replace the default text with:

Example 7-4   Creating aboutusjsf

```
<H1>All about SAL404 Realty</H1>
    <HR>
    <H2>For all your real estate needs</H2>
    <P>Founded in 2003 by a dynamic group of redbook residents
    sal404 Realty is a privately held company specializing in selling
    hot housing property.
    </P>
```

2. Apply the template to finish the About Us page.

7.5.5 Creating the news list page

We develop the complete news application using JSF and SDO and apply the template later. We start with the ListNews page, then add view, insert, and update actions.

To create the ListNews page, follow these steps:

1. Create a folder named `newsJSF` for the news application.
2. Select the **newsJSF** folder and **New → Faces JSP File**.
3. Enter `ListNews` as name, deselect the other options, and click **Finish**.
4. Replace the default text with a heading 1, List News as in step 3 on page 257.

Creating the relational record list for news items

To display the news items, we create a relational record list (SDO data graph).

1. In the Page Data view select **New → Relational Record List**.

2. Enter `listNews` as name. Select **Fill record with existing data from the database** and click **Next**.

3. For the connection click **New**. This is required at least once for a Web application. Enter the following information:

   – Enter `SAL404_Con1` as name and click **Create New DB Connection**.
   – Select **Choose a DB2 alias**.
   – Select the **IBM DB2 Application** JDBC driver.
   – Enter `SAL404R` for connection alias.
   – Verify the JDBC driver class location.
   – Enter `db2admin` as User ID and the correct password.

- Click **Test Connection** to make sure it works.
- Click **Finish** (Figure 7-13).
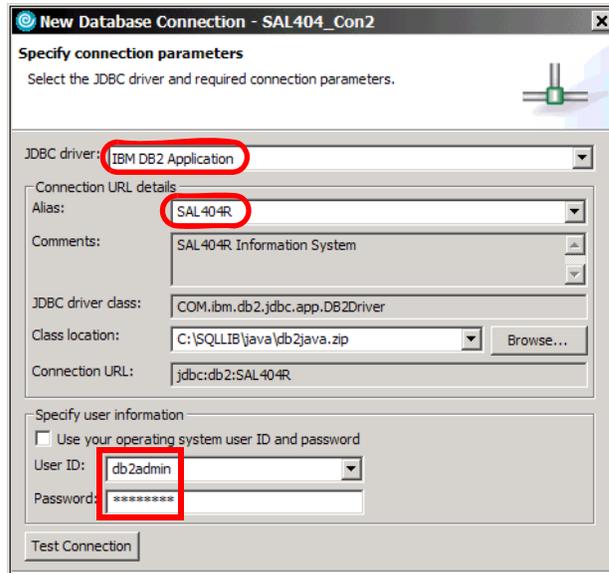
Figure 7-13   Creating a database connection

4. The tables of the SAL404R database are retrieved. Select the **NEWS_ITM** table and click **Next**, Figure 7-14 on page 260.
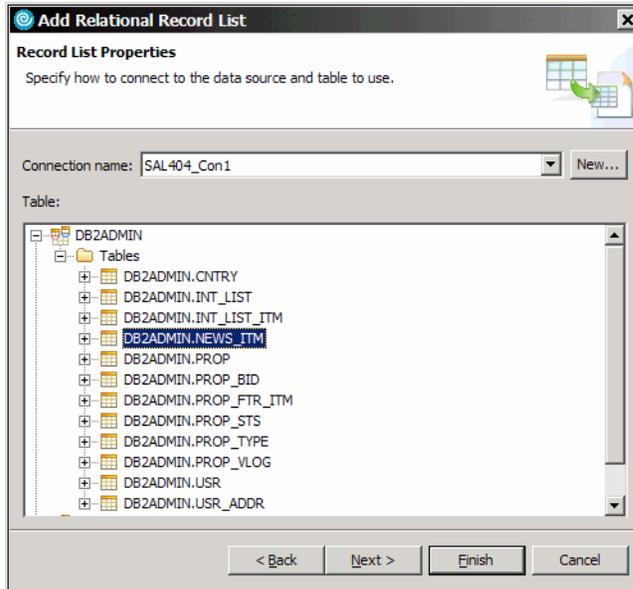
Figure 7-14   Creating a relational record list: table selection

5. Deselect the **NEWS_ITM_BODY** column. It is not displayed in the news list.

6. Click **Order results**. Select the **NEWS_ITM_DATE** column and click **>**. Select **Descending**, then click **Close** (Figure 7-15).

7. Click **Finish**.
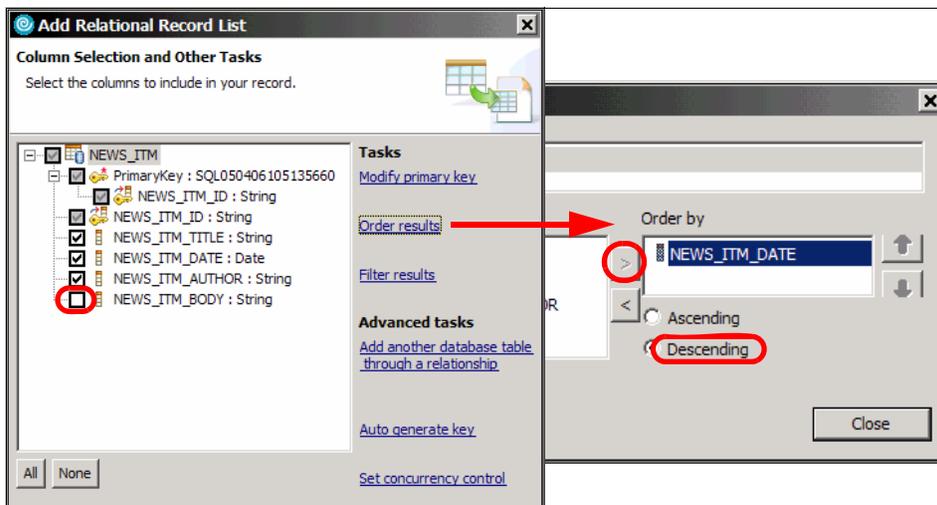
Figure 7-15   Creating the relational record list: columns and sorting

The relational record list appears in the Page Data view, together with two action methods (Figure 7-16).
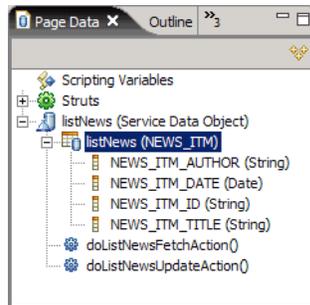
Figure 7-16   Creating the relational record list: page data

Displaying news items

To display the news items, we drop the relational record list into the JSF page:

1. Select the **listNews** bean in the Page Data view then drag and drop it into the JSF page.

2. In the Insert Record List dialog box (Figure 7-17) deselect the **NEWS_ITM_ID** (we do not list the ID), and change the labels to `Title`, `Date`, and `Author`.
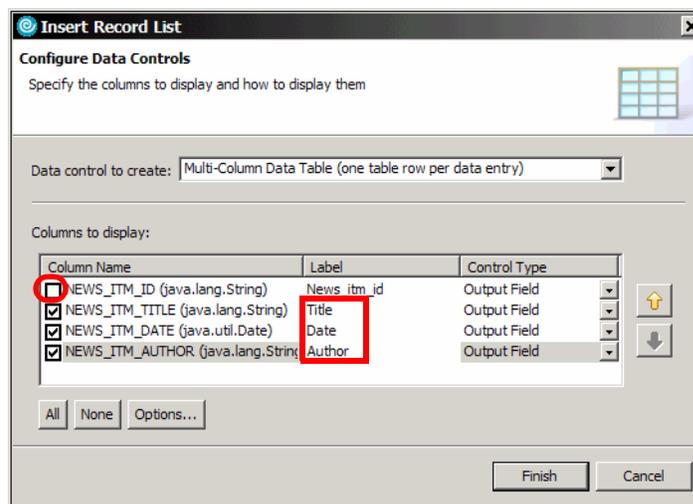
Figure 7-17   Displaying the relational record list

3. Click **Finish** and a JSF data table is created in the page.

Tailoring the news list

The data table can be tailored in many ways. Here we use a few of the options:

1. Select the data table and in the Properties view set the border to **1**. To select the data table, select any column, and in the Properties view select the line **h:dataTable**.

2. Select **Display options** (under `h:dataTable`) and click **Add a deluxe pager**. Enter **8** for rows per page.

3. Save the page (Figure 7-18).

Figure 7-18   ListNews page: initial design

Testing the news list

At this point we can test the page.

1. Select the **ListNews.jsp** and **Run** → **Run on Server**. The list of news items is displayed (Figure 7-19).

Figure 7-19   ListNews page: initial run

7.5.6 Preparing the news list page for selection and updates

We want to be able to perform a number of options after displaying the news items:

► Select an item to display the details
► Select an item to update the details
► Select multiple items for delete
► Add a new item

For this purpose, we use a number of JSF options:

► A hyperlink to select an item to display the details
► A column with check boxes for selection (update and delete)
► Command buttons to invoke Add, Modify, and Delete functions

Here are the instructions to enhance the ListNews page:

1. Select the data table and in the Properties view select **Row actions** and then click **Add selection column to the table**. This action adds a column with check boxes in the front of the table.

2. In the Palette, select the **Command - Hyperlink** component and drop it onto the {NEWS_ITM_TITLE} field in the data table. This action adds a link icon 🖼 in front of the field name.

3. We want to pass the news item ID to the next JSF page to display the details. Select the **link icon**, and in the Properties view select **Parameter**. Click **Add Parameter**, and a line is added to the list. Overtype the name with `NEWS_ITM_ID`. Select the value field and click the **icon** on the right. When prompted, expand the **listNews** bean and select the **NEWS_ITM_ID** column. The value is displayed as #{varlistNews.NEWS_ITM_ID}.

4. Select the **Command - Button** component in the Palette. Drop a button under the data table. Repeat this three times.

5. Select each button and in the Properties view set the Id field to **addButton**, **modifyButtonSDO**, **modifyButtonDAO**, and **deleteButton**. Select **Display options** in the Properties view and set the button labels to **New, Modify SDO. Modify DAO,** and **Delete**.

6. Select the **Display Errors** component in the Palette and drop it under the table.

7. Open the stylesheet.css file. In the `theme` folder for the .message and .messages component set the color to red, as in Example 7-5:

Example 7-5   Setting the theme color

```
.message {
    color: red
}

.messages {
    color: red
}
```

8. Add a separator (`<hx:outputSeparator></hx:outputSeparator>`) under the heading. This component is hidden in the Palette, but can be inserted using **JSP** → **Insert Custom** and selecting the **outputSeparator** from the hx tag library.

Figure 7-20 shows the final design of the page.

Figure 7-20   List news page design

9. Select the **New** button and in the properties view click **Add Rule**. Enter `newsJSF/addNewsItem.jsp` as the target page. Select **The outcome named** and enter `addNewsItem`. Select **Any action** and **This page only**. Click **OK**.

   Repeat this procedure in Figure 7-21 for three more rules.
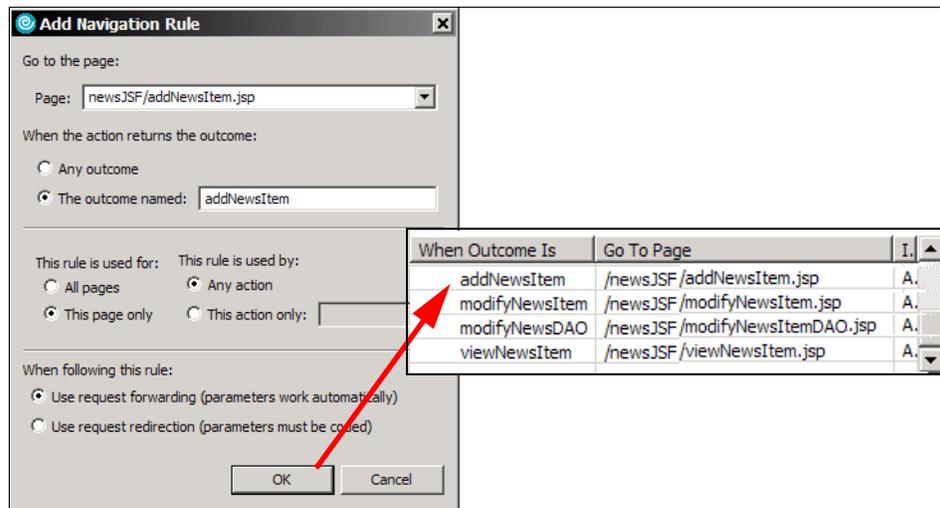
Figure 7-21   Navigation rules

   The outcome values are the values that must be returned by the action logic.

10. Select the **New** button and in the Quick Edit view select **Command** in the left frame and click inside the right frame. In the sample code that appears, change the return statement to `return "addNewsItem";` so that the addNewsItem page will be displayed.

11.Select the hyperlink **NEWS_ITM_TITLE** and in the Quick Edit view enter this code:

```
log("View news item: "+ getRequestParam().get("NEWS_ITM_ID"));
getRequestScope().put("newsId", getRequestParam().get("NEWS_ITM_ID") );
return "viewNewsItem";
```

We retrieve the parameter that was setup in the hyperlink and store it in the request block for the next JSF page (viewNewsItem).

The logic for update and delete is more complex and can wait for later.

Create skeleton pages for news action testing

To test the ListNews page, we create the four skeleton pages for view, add, and modify:

1. In the newsJSF folder, create four JSF pages named viewNewsItem, addNewsItem, modifyNewsItem, and modifyNewsItemDAO.

2. Change the default text to a text identifying the new page.

3. Run the ListNews page and test the hyperlink and New actions. The target pages should be displayed.

7.5.7 Creating the news item details page

The viewNewsItem.jsp displays the selected news item when you click the hyperlink. The NEWS_ITM_ID is passed as parameter and stored in the request block as newsId. We use a relational record to retrieve the details of the selected item.

Here are the short instructions to create the page:

1. Open the **viewNewsItem.jsp**.

2. Change the initial text to `<H1>News Item Details</H1>`.

3. In the Page Data view select **New** → **Scripting Variable** → **Request Scope Variable**. Enter `newsId` as name and `java.lang.String` as type.

4. In the Page Data view select **New** → **Relational Record**. Enter `newsItem` as name and select **Fill record with existing data from the database**.

5. Select the **NEWS_ITM** table.

6. Leave all columns selected.

7. Click **Filter results**. A default condition is generated.

8. Select the condition and click the **Edit** icon ✐. For the value click the … button and locate the request scope variable **newsId**. Close the filter dialog and click **Finish**.

9. Drop the **newsItem** bean into the JSF page. In the dialog select **Displaying an existing record**. Deselect the **NEWS_ITM_ID**. Change the labels to shorter names. Click **Finish** as in Figure 7-22.
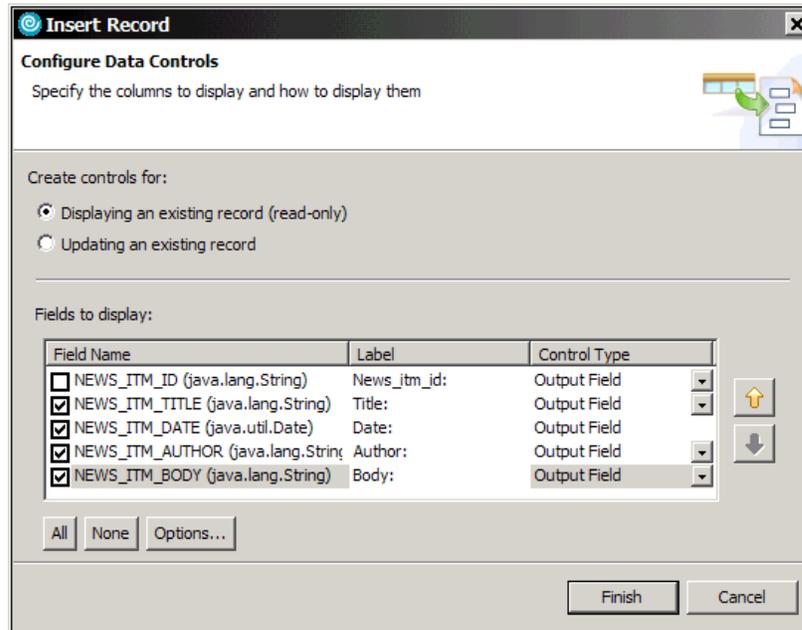
Figure 7-22   Displaying a news item

10. Add a **Cancel** button (with ID `cancelButton`) under the table.

11. Add a separator (`<hx:outputSeparator>`) under the heading (Figure 7-23).

Figure 7-23   Design of the news item details page

12. Select the **Cancel** button and add a navigation rule in the Properties view:

- Target page: `newsJSF/listNews.jsp`
- Outcome: `listNews`
- Select **All pages** (global rule)

13.In the Quick Edit view, enter the code `return "listNews";`

Testing the news item details

Rerun the ListNews.jsp, then select news items by clicking the title. The details are displayed in Figure 7-24.
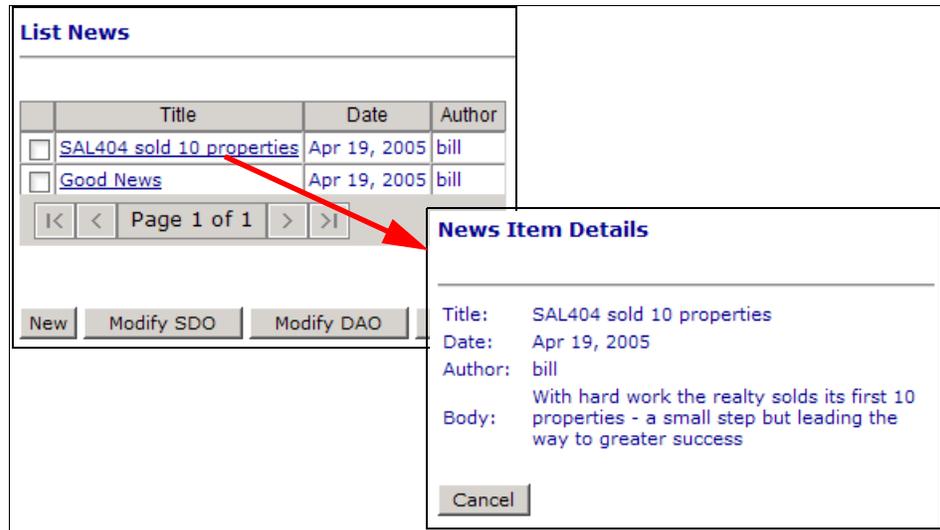
Figure 7-24   Displaying the news details

Note the format of the date. This can be tailored by selecting the date output field and in the Properties view and changing the format type and style.

7.5.8 Creating the news item add page

The addNewsItem.jsp is used to enter a new news item. We use a relational record to enter the details of a new item.

To create the page, follow these short instructions:

1. Open the **addNewsItem.jsp**.

2. Change the initial text to `<H1>Add News</H1>`.

3. In the Page Data view select **New** → **Relational Record**. Enter `addNewsItem` as name and select **Create an empty record in order to create a new row in the database**.

4. Select the **NEWS_ITM** table.

5. Leave all columns selected.

6. Click **Finish**.

7. Drop the **addNewsItem** bean into the JSF page. In the dialog box, *deselect* the **NEWS_ITM_ID;** we have to generate a key using Java code. Change the labels to shorter names. For the NEWS_ITM_BODY, select **Input Text Area** as type.

8. Click **Options**. Select **Submit button** and set the label to **Add**.

9. Click **Finish**. The table for data entry is inserted.

10. Select the **Add** button and change the ID to `addButton`.

11. Add a Command - Button next to the Add button. In the Properties view select **Reset** as type and set the Id to `resetButton`.

12. Add the separator `<hx:outputSeparator>` under the heading (Figure 7-25).

Figure 7-25   Design of the add news item page (initial)

Adding validation of input fields

To validate the input fields we use the predefined validators and display errors next to each field:

1. Change the IDs of the input fields to more convenient names (`newsTitle` instead of `text1`).

2. Select the **NEWS_ITM_TITLE** input field. In the Properties view select **Validation** (Figure 7-26 on page 269). Select **Value is required**, enter **6** for minimum length and 100 for maximum length (database limit). Select **Display validation error messages in an error message control.** This creates a message field next to the input field.

3. For the NEWS_ITM_DATE field, select **Value is required** and create the error message field.

4. For the NEWS_ITM_AUTHOR field select **Value is required,** enter **4** and **100** as length limits, and create the error message field.

5. For the NEWS_ITM_BODY field, select **Value is required,** enter **10** as minimum length, and create the error message field. In the Properties view for size enter **40** for width and **4** for height to make the text area bigger.
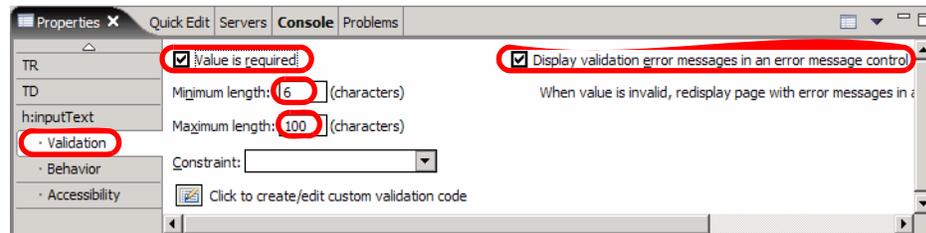
Figure 7-26   Input data validation

6. The design of the add news item page is shown in Figure 7-27.

Figure 7-27   Final design of the add news item page

Adding the logic to add a news item

For a new news item, we have to generate a proper key for the database. We also need a confirmation message from the resource properties file:

1. The resource properties file is named `ApplicationResources.properties` in the `sal404realtyweb.resources` package.

2. The news section should have these entries in Example 7-6:

Example 7-6   Logic entries for news items

```
# News
error.news.search=<li>Database error finding news items
error.news.invalidDate=<li>The date input is invalid
error.title.required=<li>Title is required
error.date.required=<li>A date is required
```

```
news.error.addNewsItem=<li>Failure adding news item
news.error.fetchNewsItem=<li>An error occured while fetching a new item
news.error.no.delete.selection=<li>Please select at least one news item to
delete
news.error.no.modify.selection=<li>Please select a news item to modify
news.modify.ok=News modified successfully
news.add.ok=News added successfully
news.delete.ok=News items deleted successfully
news.error.delete=Failure deleting news items
news.error.modify=Failure modifying news item
```

3. The logic for the Add button (Quick Edit view) is shown in Example 7-7. The lines in bold must be added to the logic that is already generated for the Add button.

Example 7-7   Logic to add a news item

```
ResourceBundle r = ResourceBundle
            .getBundle("sal404realtyweb.resources.ApplicationResources");
try {
   KeyGenerator generator = new KeyGenerator();
   String newsId = generator.getKeyString();
   getAddNewsItem().setString("NEWS_ITM_ID",newsId);
   getAddNewsItemMediator().applyChanges(
                getRootDataObject(getAddNewsItem()));
   getRequestScope().put("confirmationMessage", r.getString("news.add.ok"));
} catch (Throwable e) {
   logException(e);
   getRequestScope().put("confirmationMessage",
                        r.getString("news.error.addNewsItem"));
} finally {
   SDOConnectionWrapper = null;
}
return "listNews";      // return to the news list
```

4. The logic for the Cancel button is one line: `return "listNews";`

Testing news item add

Rerun the `ListNews.jsp`, then click **New** for a new item. Enter the details and click **Add**. The new item is displayed in the list, sorted by date. Note that you have to enter the date in the format `Mmm dd, yyyy`. See Figure 7-28 on page 271.

Figure 7-28   Adding a news item to the list

7.5.9 Implementing news item selection

For both modification and deletion of news items we select news items using the check box column. We have to prepare a managed bean to receive the selected items. The best way to work with check boxes is to bind an integer array (Integer[]) to the check box field. This array contains the indexes of the selected rows.

We provide the com.ibm.itso.sal404.news.selection.NewsSelection class for this purpose. This class contains a property with getter and setter methods:

```
private Integer[] selectedNews = new Integer[0];
```

Open the ListNews.jsp and follow these steps:

1. In the Page Data view select **New** → **JavaBean**. Enter `newsSelection` as name, browse to the NewsSelection class, select **Make this JavaBean reusable** and select **request** as scope. Click **Finish,** as in Figure 7-29 on page 272.
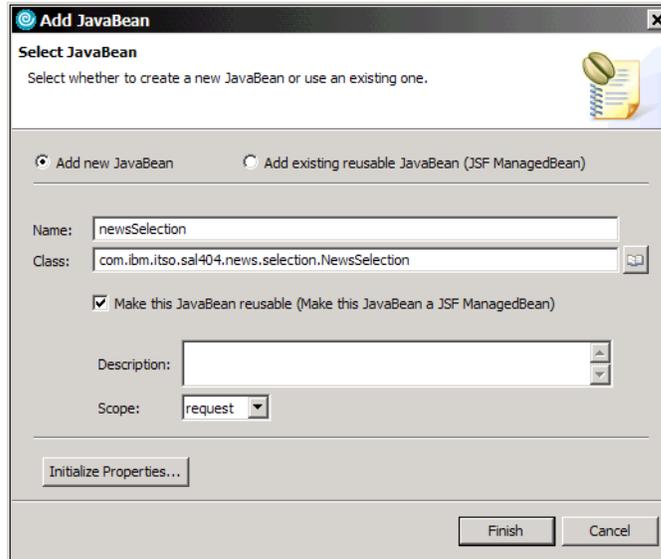
Figure 7-29   Defining a managed bean

2. The newsSelection bean appears in the Page Data view. A getNewsSelection method is generated into the page code class.

3. Expand the **newsSelection** bean, then drag and drop the **selectedNews** property onto the check box field. This property will contain the indexes of the selected rows.

4. Select a **Display Errors** component in the Palette and drop it under the table. We use this to display error messages for incorrect selection.

7.5.10 Implementing news item delete

To delete news items, we remove the selected rows from the data graph and commit the changes.

1. In the ListNews.jsp, select the **Delete** button

2. In the Quick Edit view enter the code in Example 7-8.

Example 7-8   Logic to delete news items

```
ResourceBundle r = ResourceBundle
            .getBundle("sal404realtyweb.resources.ApplicationResources");
Integer[] selected = getNewsSelection().getSelectedNews();
if (selected.length == 0) {
   getFacesContext().addMessage("",
      new FacesMessage(FacesMessage.SEVERITY_ERROR,
```

```
                        r.getString("news.error.no.delete.selection"),
                        r.getString("news.error.no.delete.selection")) );
    return "";
}
for (int i=0; i<selected.length; i++) {
    int row = selected[i].intValue();
    DataObject newsItem = (DataObject)getListNews().get(row);
    String newsId = newsItem.getString("NEWS_ITM_ID");
    log("Delete news item: " + row + " id: " + newsId);
    newsItem.delete();
}
try {
    getListNewsMediator().applyChanges( getRootDataObject(getListNews()) );
    getRequestScope().put("confirmationMessage",
                        r.getString("news.delete.ok") );
} catch(Exception e) {
    getFacesContext().addMessage("",
        new FacesMessage(FacesMessage.SEVERITY_ERROR,
            r.getString("news.error.delete"),r.getString("news.error.delete")) );
} finally {
    SDOConnectionWrapper = null;
}
getNewsSelection().setSelectedNews( new Integer[0] );
return "";
```

The processing is as follows:

3. Retrieve the resource bundle for error messages.

4. Retrieve the selected rows array.

5. Verify that at least one item is selected. Return with an error message if no rows were selected.

6. Scan the array for selected items.

7. For each item, retrieve the data object of the news item and delete it.

8. Call the mediator to commit the changes.

9. Set the confirmation message for successful delete.

10. In case of errors, return with an error message.

11. Clear the selection array so that no rows are selected after returning.

Testing news item delete

To test deleting a news item, perform the following tasks:

1. Rerun the ListNews.jsp,
2. Select the news items added and click **Delete**.

7.5.11 Implementing news item update using SDO

To update a news item, we retrieve the item and display the data in input fields for updating. First we have to pass the selected news item id to the update page. In the ListNews.jsp, select the **Modify SDO** button and in the Quick Edit view enter the code in Example 7-9.

Example 7-9 Logic to pass a news item ID to the update page

```
ResourceBundle r = ResourceBundle
        .getBundle("sal404realtyweb.resources.ApplicationResources");
Integer[] selected = getNewsSelection().getSelectedNews();
if (selected.length != 1) {
   getFacesContext().addMessage("",
        new FacesMessage(FacesMessage.SEVERITY_ERROR,
                        r.getString("news.error.no.modify.selection"),
                        r.getString("news.error.no.modify.selection") ));
   return "";
}
int row = selected[0].intValue();
String newsId = ((DataObject)getListNews().get(row)).getString("NEWS_ITM_ID");
log("Modify new item: " + row + " id: " + newsId);
getRequestScope().put("newsId", newsId);
return "modifyNewsItem";
```

The processing is as follows:

1. Retrieve the resource bundle for error messages.
2. Verify that exactly one item is selected.
3. Retrieve the ID of the selected item from the data object.
4. Pass the ID in request scope to the update page.

Implementing the news item update page (SDO)

To implement update using SDO, open the **modifyNewsItem.jsp** and proceed as follows:

1. In the Page Data view select **New** → **Scripting Variable** → **Request Scope Variable**. Enter `newsId` as name and `java.lang.String` as type. Click **OK**.

2. In the Page Data view select **New** → **Relational Record**.

3. Enter `newsItem` as the name. Select **Reuse metadata definition from an existing record or record list** and select the **newsItem.xml** file. Select **Fill record with existing data from the database**. Click **Next** then **Finish**.

4. Drag and drop the **newsItem** bean into the JSF page. Select **Updating an existing record**. *Deselect* the **NEWS_ITM_ID**. Change the labels to short names, For the NEWS_ITM_BODY, select **Input Text Area** as type.

Click **Options** and select **Submit button** with the label `Modify`. Deselect **Delete button**. Click **OK**, then click **Finish**.

5. Add a **Command - Button** with the label **Cancel**.

6. Add a separator (`<hx:outputSeparator>`) under the heading.

7. Implement the same validation as for the add news item page (see "Adding validation of input fields" on page 268). The design of the modify news item page is shown in Figure 7-30.

Figure 7-30   Design of the news item update page (SDO)

8. Enter the Cancel button logic in the Quick Edit view: `return "listNews";`

9. The logic for the **Modify** button is generated into the page code class. Update the logic as shown in Example 7-10. The only additions are to display confirmation or error messages.

Example 7-10   Logic to update a news item (SDO)

```
ResourceBundle r = ResourceBundle
        .getBundle("sal404realtyweb.resources.ApplicationResources");
try {
   getNewsItemMediator()
        .applyChanges(getRootDataObject(getNewsItem()));
   getRequestScope().put("confirmationMessage",
                         r.getString("news.modify.ok"));
} catch (Throwable e) {
   logException(e);
   getRequestScope().put("confirmationMessage",
                         r.getString("news.error.modify"));
} finally {
   SDOConnectionWrapper = null;
}
return "listNews";
```

Testing news item update (SDO)

To test the new itme update, follow these steps:

1. Rerun the `ListNews.jsp`, then select the news item to be updated.

2. Click **Modify SDO**. Change the data and click **Modify**. The updated values are displayed in the news list.

7.5.12 Implementing news item update using DAO

To update a news item, we retrieve the news item (NewsItemDTO) using DAO and pass it in session scope to the update page.

1. In the ListNews.jsp, select the **Modify DAO** button and in the Quick Edit view enter the code in Example 7-11.

Example 7-11   Logic to retrieve the news item using DAO

```
ResourceBundle r = ResourceBundle
            .getBundle("sal404realtyweb.resources.ApplicationResources");
Integer[] selected = getNewsSelection().getSelectedNews();
if (selected.length != 1) {
   getFacesContext().addMessage("",
      new FacesMessage(FacesMessage.SEVERITY_ERROR,
                          r.getString("news.error.no.modify.selection"),
                          r.getString("news.error.no.modify.selection")) );
   return "";
}
int row = selected[0].intValue();
String newsId = ((DataObject)getListNews().get(row)).getString("NEWS_ITM_ID");
log("Modify new item: " + row + " id: " + newsId);
NewsManager newsManager = new NewsManager();
try {
   NewsItemDTO newsItem = newsManager.viewNewsItemDetails(newsId);
   getSessionScope().put("newsItem", newsItem);
   return "modifyNewsDAO";
} catch (ApplicationException e) {
   getFacesContext().addMessage("",
      new FacesMessage(FacesMessage.SEVERITY_ERROR,
                          r.getString("news.error.fetchNewsItem"),
                          r.getString("news.error.fetchNewsItem") ));
   return "";
}
```

The processing is similar to SDO, but the news item is retrieved using the existing NewsManager and placed into session scope for the next page.

Implementing the news item update page (DAO)

To implement update using DAO, open the modifyNewsItemDAO.jsp and proceed as follows:

1. In the Page Data view select **New** → **JavaBean**. Enter `newsItem` as name and locate the `NewsItemDTO` class. Select **Make this JavaBean reusable** and select **session** scope. Click **Finish**. The DTO bean is shown in the Page Data view (Figure 7-31).
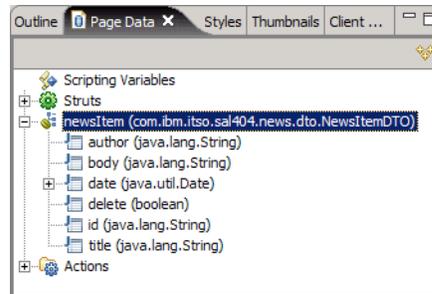
Figure 7-31   Session scope variable in the Page Data view

2. Drag and drop the **newsItem** bean into the JSF page. Select **Inputting data**. *Deselect* the **id** and **delete** fields. Rearrange the fields in the sequence `title`, `date`, `author`, and `body`. For the `body` select **Input Text Area** as type.

3. Click **Options** and select **Submit button** with the label `Update`. Click **OK**, then click **Finish**.

4. Add a **Command - Button** with the label `Cancel`.

5. Add a separator (`<hx:outputSeparator>`) under the heading.

6. Implement the same validation as for the add news item page (see "Adding validation of input fields" on page 268).

7. The design of the page is the same as Figure 7-30 on page 275, except that we used an Update button.

8. Enter the logic for the Cancel button in the Quick Edit view:

```
getSessionScope().remove("newsItem");
return "listNews";
```

9. Enter the logic for the **Update** button as shown in Example 7-12. The news item is updated by passing the DTO to the `NewsManager` for update.

Example 7-12   Logic to update a news item (DAO)

```
ResourceBundle r = ResourceBundle
            .getBundle("sal404realtyweb.resources.ApplicationResources");
NewsItemDTO newsItem = getNewsItem();
```

```
String newsId = newsItem.getId();
log("Updating News item: " + newsId + " " + newsItem.getTitle());
NewsManager newsManager = new NewsManager();
try {
    newsManager.modifyNewsItem(newsItem);
    getRequestScope().put("confirmationMessage",
                          r.getString("news.modify.ok"));
} catch (ApplicationException e) {
    getRequestScope().put("confirmationMessage",
                          r.getString("news.error.modify"));
}
getSessionScope().remove("newsItem");
return "listNews";
```

Testing news item update (DAO)

To test the update, follow these steps:

1. Rerun the ListNews.jsp, then select the news item to be updated.

2. Click **Modify DAO**. Change the data and click **Update**. The updated values are displayed in the news list (Figure 7-32).
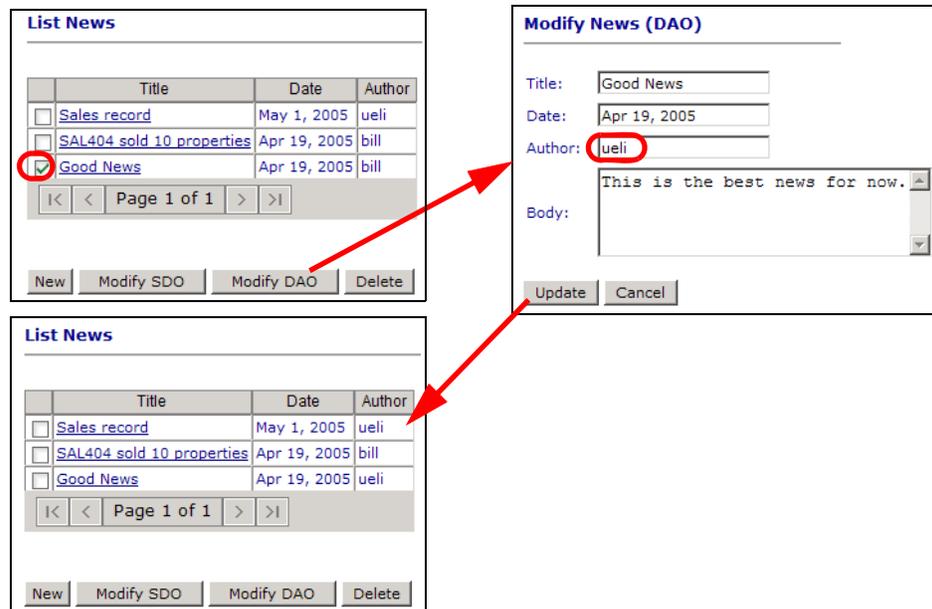
Figure 7-32   Updating a news item

7.5.13 Applying the template to the news application

We have now tested the functionality of the news application. Now we can apply the template to the pages.

There are two ways to add the template:

▶ We can create a site diagram and apply the template to multiple pages.

▶ We can open each page and then apply the template from the Palette as we did for the home page.

Creating a site diagram

To create a site diagram of the JSF application, open the **Web Site Diagram** under the Web application SAL40RealtyWeb. Then follow these steps:

1. Drag the **indexjsf.jsp** into the empty diagram.

2. Select the home page and **Add existing pages**.

3. Click **Add Other Pages** and locate the index.jsp, aboutusjsf.jsp, and the ListNews.jsp. Click **OK**.

4. Select the **ListNews.jsp** and click **Add Other Pages**. Select all other the pages in the newsjsf folder.

   The structure is displayed in Figure 7-33.
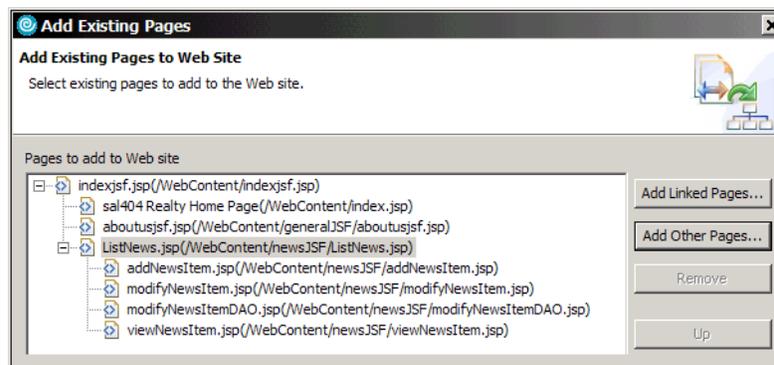
Figure 7-33   Building the site diagram

5. Click **Finish** and the diagram is displayed in Figure 7-34 on page 280.

   – Note the color of the pages with the template.

   – You can invoke the Page Designer for a page from the diagram by double-click or **Open**.
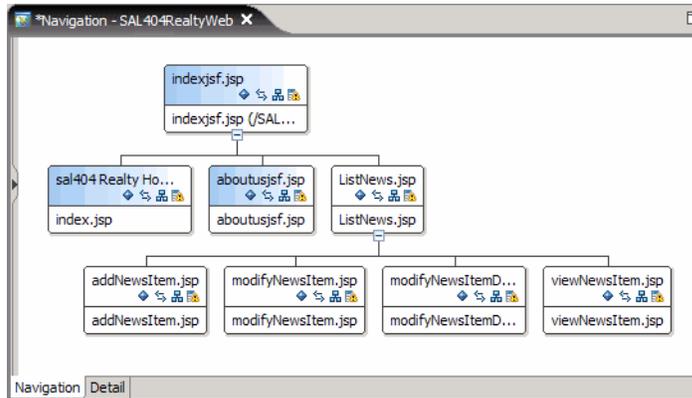
Figure 7-34   Site diagram of JSF application

Applying the template

To apply the template to multiple pages, follow these steps:

1. Select all the news pages and **Page Template** → **Apply Template**.

2. Select **User-defined page template** and then select the **sal404jsf.jtpl** template.

3. Progress through the dialog box and click **Finish**.

4. Select **Appearance** → **Show Page Template** from the context menu and the name of the template is also displayed in each box, as in Figure 7-35.
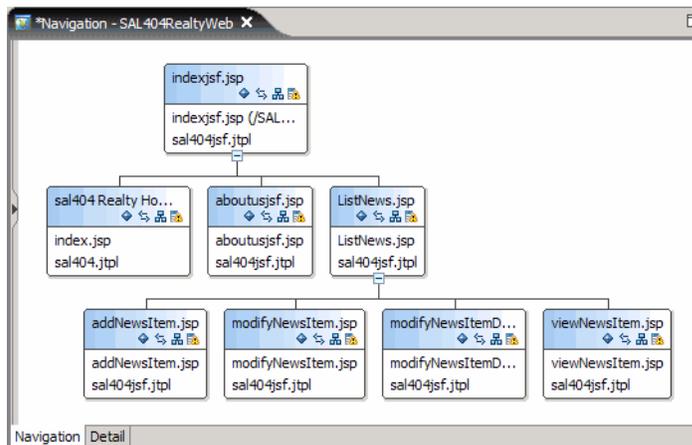
Figure 7-35   Site diagram after applying the template

7.5.14 Running the JSF application

Run the JSF application again and now the news pages are displayed in the template. Also notice that the confirmation messages of updates and deletes are displayed in the footer area of Figure 7-36.

Figure 7-36   JSF application run

7.5.15 Securing news update for administrators

The tasks of adding, deleting, and updating news items must be reserved for administrators. We can use the JSTL standard library to hide the selection check boxes and the command buttons in the news list page by placing them into a conditional block:

```
<c:if test='${userSessionData.logInUser.role==\"Administrators\"}'>
    .....
</c:if>
```

Open the ListNews.jsp and in the **Source** tab, add the code shown in Example 7-13.

Example 7-13   Securing the news update facility

```
<h:form styleClass="form" id="form1">
    ......
    <h:dataTable id="table1" ....................>
        <h:column id="column4">
            <c:if test='${userSessionData.logInUser.role==\"Administrators\"}'>
            <hx:inputRowSelect styleClass="inputRowSelect" id="rowSelect1"
```

```
                       value="#{pc_ListNews.newsSelection.selectedNews}">
                       <f:convertNumber /></hx:inputRowSelect>
             <f:facet name="header"></f:facet>
             </c:if>
       </h:column>
       <h:column id="column1">
       ......
       ......
   </h:dataTable>
   <h:messages styleClass="messages" id="messages1"></h:messages>
   <BR>
   <BR>
   <c:if test='${userSessionData.logInUser.role==\"Administrators\"}'>
   <hx:commandExButton type="submit" value="New" ......></hx:commandExButton>
   <hx:commandExButton type="submit" value="Modify SDO" ......></hx:command..>
   <hx:commandExButton type="submit" value="Modify DAO" ......></hx:command..>
   <hx:commandExButton type="submit" value="Delete" ......></hx:command..>
   </c:if>
</h:form>
```

Rerun the application. The update facility is only visible after login as an administrator. Note that you have to use the non-JSF home page to access login.

7.6 JSF and SDO control files

JSF and SDO maintain information in control files in the Web project structure. These files are maintained by the system and should not be touched.

Some of the file names start with a period and are not visible by default in the Project Explorer view. To view .xxxxx files, select the arrow pull-down ▼ in Project Explorer and select **Filters**. Deselect **.*** at the bottom of the list and click **OK**.

7.6.1 JSF control files

The main control file of JSF is the XML configuration file (faces-config.xml) in the WEB-INF folder. This file is used by the Faces servlet to manage the JSF environment.

The faces-config.xml file contains this information:

► *Managed beans* are application data beans defined in the Page Data view. For example, the newsItem bean for news item update using DAO is defined in the configuration file. See Example 7-14 on page 283.

Example 7-14   Managed bean configuration

```
<managed-bean>
    <managed-bean-name>newsItem</managed-bean-name>
    <managed-bean-class>com.ibm.itso.sal404.news.dto.NewsItemDTO<//..>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

▶ *Page code classes* are the Java classes created for each JSF page are also defined as managed beans, for example:

Example 7-15   Page code classes

```
<managed-bean>
    <managed-bean-name>pc_ListNews</managed-bean-name>
    <managed-bean-class>pagecode.newsJSF.ListNews</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

▶ *Navigation rules* are defined for each JSF page. Example 7-16 shows the rules for the ListNews page:

Example 7-16   Navigation rules for ListNews.jsf

```
<navigation-rule>
    <from-view-id>/newsJSF/ListNews.jsp</from-view-id>
    <navigation-case>
        <from-outcome>modifyNewsItem</from-outcome>
        <to-view-id>/newsJSF/modifyNewsItem.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>addNewsItem</from-outcome>
        <to-view-id>/newsJSF/addNewsItem.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>viewNewsItem</from-outcome>
        <to-view-id>/newsJSF/viewNewsItem.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>modifyNewsDAO</from-outcome>
        <to-view-id>/newsJSF/modifyNewsItemDAO.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

▶ Additional tags are used for life cycle phase listeners and application property resolvers.

A control file, .jspPersistence, in the Web project holds information about the scripting variables (request and session scope) that were defined in the Page Data view for JSF pages.

7.6.2 SDO control files

SDO keeps information about the connection and the relational records in its control files.

Connection information

SDO keeps the connection information in the .wdo_connections file in the project. This file keeps the specification of the development connection (used during design of the relational records) and the runtime connection (used for testing). In our case the abbreviated information is in Example 7-17.

Example 7-17   SDO connection information

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.websphere.wdo.connections:connections ......>
  <connection id="SAL404_Con1">
    <development xsi:type="..." id="SAL404_Con1_dev" name="SAL404_Con1"/>
    <runtime xsi:type="..." auto-deploy="true"
       class-location="C:\SQLLIB\java\db2java.zip" id="SAL404_Con1_runtime"
       classname="COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource"
       database-name="SAL404R" jndi-name="jdbc/SAL404_Con1"
       password="{xor}NissMGsmMCo=" resource-reference-name="SAL404_Con1"
       server-name="localhost" sql-vendor-type="30" userid="db2admin"/>
  </connection>
</com.ibm.websphere.wdo.connections:connections>
```

Resource reference

The runtime connection through a data source must be configured in the server. For this purpose a resource reference (SAL404_Con1) is created in the Web deployment descriptor, visible on the References page. The reference points to the data source with a JNDI name of jdbc/SAL404_Con1.

Extended deployment descriptor

Rational Web Developer deploys this data source to the server automatically by using the extended deployment descriptor of the enterprise application.

Open the deployment descriptor of the SAL404Realty enterprise application. Select the **Deployment** page (Figure 7-37 on page 285) and you find a JDBC provider (WDO DB2 JDBC Provider1). Select the JDBC provider and you find the SAL404_Con1 data source with the jdbc/SAL404_Con1 JNDI name.
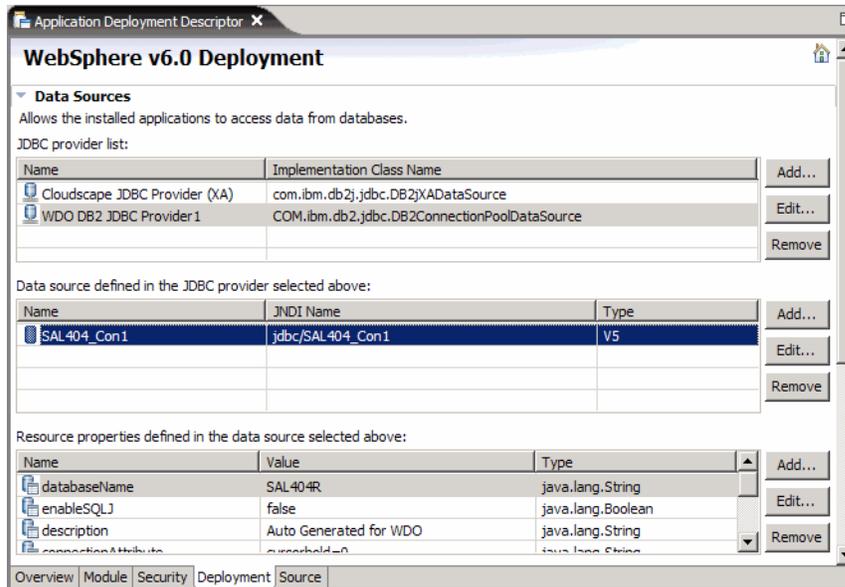
Figure 7-37   Enterprise application deployment descriptor with data source

This data source is automatically deployed to the test server when the enterprise application is deployed. For deployment to a production environment, the data source should be defined by an administrator, although automatic deployment to a WebSphere server works as well.

Relational records and relational record lists

The definition of the relational records is kept in XML files in the WEB-INF/wdo folder. For our application we defined one record list (`listNews`) and two records (`newsItem` and `addNewsItem`):

▶ The `listNews.xml` file is shown in Example 7-18.

Example 7-18   ListNews.xml relational definition

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.websphere.sdo.mediator.jdbc.metadata:Metadata xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:com.ibm.websphere.sdo.mediator.jdbc.metadata="..."
    rootTable="//@tables.0">
  <tables schemaName="DB2ADMIN" name="NEWS_ITM">
    <primaryKey columns="//@tables.0/@columns.0"/>
    <columns name="NEWS_ITM_ID" type="4"/>
    <columns name="NEWS_ITM_TITLE" type="4" nullable="true"/>
    <columns name="NEWS_ITM_DATE" type="10" nullable="true"/>
    <columns name="NEWS_ITM_AUTHOR" type="4" nullable="true"/>
```

```
      </tables>
      <orderBys column="//@tables.0/@columns.2"/>
</com.ibm.websphere.sdo.mediator.jdbc.metadata:Metadata>
```

▶ The `newsItem.xml` file is almost the same, but includes a filter condition to
  retrieve one record only, and no sorting (`orderBys`):

```
  <filter predicate="( NEWS_ITM_ID = ? )">
     <filterArguments name="requestScopenewsId" type="4"/>
   </filter>
```

▶ The `addNewsItem.xml` file includes the `NEWS_ITM_BODY` column.

The SDO mediator uses these XML files to generate the JDBC SQL calls to
retrieve and update the data in the database.

7.7 SDO API

In this section we take a brief look at the API of SDO.

7.7.1 SDO calls generated into the page code class

When using relational records in a JSF page, the SDO API calls are generated
into the page code class.

We can see most of the API calls in the ModifyNewsItem class that was
generated for the modifyNewsItem.jsp:

▶ Example 7-19 shows the declaration of SDO objects:

Example 7-19   Declaration of SDO objects

```
      protected DataObject newsItemParameters;
      protected JDBCMediator newsItemMediator;
      private static final String SDOConnection_name = "SAL404_Con1";
      private ConnectionWrapper SDOConnectionWrapper;
      private static final String newsItem_metadataFileName =
                                "/WEB-INF/wdo/newsItem.xml";
      protected DataObject newsItem;
```

► Example 7-20 shows the method to create the connection wrapper (getSDOConnectionWrapper):

Example 7-20   Method for getSDOConnectionWrapper

```
Connection con = ConnectionManager
                    .createJDBCConnection(SDOConnection_name);
SDOConnectionWrapper = ConnectionWrapperFactoryImpl.soleInstance
                .createConnectionWrapper(con);
```

► Example 7-21 shows the method to create the mediator using the XML file and the connection (getNewsItemMediator):

Example 7-21   Method for getNewItemMediator

```
newsItemMediator = JDBCMediatorFactoryImpl.soleInstance
                    .createMediator(
                        getResourceInputStream(newsItem_metadataFileName),
                        getSDOConnectionWrapper());
```

► The method to setup the parameters for retrieve (getNewsItemParameters) is:

```
newsItemParameters = getNewsItemMediator().getParameterDataObject();
```

► Example 7-22 shows the method to retrieve the news item (doNewsItemFetchAction):

Example 7-22   Retrieve doNewItemFetchAction method

```
resolveParams(getNewsItemParameters(), newsItemArgNames,
                            newsItemArgValues, "newsItem_params_cache");
DataObject graph = getNewsItemMediator().getGraph(
                                        getNewsItemParameters());
newsItem = (DataObject) graph.getList(0).get(0);
```

► Example 7-23 shows the method to access the news item (getNewsItem):

Example 7-23   Access getNewsItem method

```
if (newsItem == null) {
        doNewsItemFetchAction();
}
return newsItem;
```

► Method to update the news item (doNewsItemUpdateAction):

```
getNewsItemMediator().applyChanges(getRootDataObject(getNewsItem()));
```

getRootDataObject is a helper method in the parent class, PageCodeBase.

► Example 7-24 shows the method to delete the news item (`doNewsItemDeleteAction`):

Example 7-24   Delete news item (doNewItemDeleteAction)

```
DataObject root = getNewsItem().getDataGraph().getRootObject();
getNewsItem().delete();
getNewsItemMediator().applyChanges(root)
```

These methods are invoked automatically when an SDO object (relational record) is used in a JSF page:

► The binding of a JSF component to a record item invokes the get method (`getNewsItem`). This triggers a call to the fetch method, requiring in succeeding order: the parameters, the mediator, and the connection.

► The logic of the Update button invokes the update method (`doNewsItemUpdateAction`) that uses the mediator to commit the changes to the database.

7.7.2 SDO API of the data object

The SDO API of the data object enables you to write programs using SDO without a GUI front-end, or to perform more complex logic than updating fields from a panel.

Once a data object has been retrieved, the properties can be accessed using the API. For example, to retrieve and manipulate a news item (`DataObject newsItem`) we can use the coding in the following examples.

► Retrieve a property in Example 7-25:

Example 7-25   Retrieve a property

```
(String)newsItem.get("NEWS_ITM_AUTHOR");       // get retrieves an Object
newsItem.getString("NEWS_ITM_AUTHOR");
newsItem.getDate("NEWS_ITM_DATE");
```

► Update a property in Example 7-26:

Example 7-26   Update a property

```
newsItem.setString("NEWS_ITM_AUTHOR", "john");
newsItem.setDate("NEWS_ITM_DATE", new Date());
newsItem.set("NEWS_ITM_TITLE", "Excellent News");
```

► Retrieve data objects from a relational record list in Example 7-27:

Example 7-27   Retrieve objects from a relational record list

```
for (int i=0; i < getListNews().size(); i++) {
    DataObject newsItem = (DataObject)getListNews().get(index);
    // process item: newsItem.get, newsItem.set
    // delete item:  newsItem.delete();
}
```

► Update the database with changes (relational record):

```
getNewsItemMediator().applyChanges(getRootDataObject(getNewsItem()));
```

► Update the database with changes (relational record list):

```
getListNewsMediator().applyChanges( getRootDataObject(getListNews()) );
```

► Complex data graphs that consist of data from multiple tables (relationships) require XPath expressions to access nested tables. For example, to access employees in a department would require code such as in Example 7-28:

Example 7-28   Retrieveing employee entries by department

```
DataObject dept = getDepartment();
nrEmployees = ( (List)dept.get("DEPARTMENT_EMPLOYEE") ).size();
for (int i=0; i < nrEmployees; i++) {
    String firstname = dept.getSring("DEPARTMENT_EMPLOYEE"+i+"/FIRSTNAME");
    DataObject employee = dept.getDataObject("DEPARTMENT_EMPLOYEE"+i);
    String lastname  = employee.getString("LASTNAME");
}
```

The `firstname` example shows how to access data within a contained object using an XPath expression without retrieving the contained object.

8

Service Data Objects

In this chapter we discuss Service Data Objects(SDO), as well as provide an overview of the technology and what it offers. We describe the SDO architecture and the support provided by Rational Web Developer for building SDO applications. We provide an example of incorporating SDO functionality into our redbook sample application.

291

8.1 SDO technology

A common model for developing e-business solutions employs an n-tier distributed environment or architecture, where the different tiers communicate with each other to provide the overall system functionality. Accessing data in the n-tier architecture is very dependent on the type of data source because a specific technology or API is used. Consider a JDBC data source, an application developer will have to learn, the JDBC APIs necessary to seemlessly retrieve, manipulate and update database records. If we consider an XML data source, a different set of APIs will have to be used. This means that developers spend a lot of time learning specific technologies for accessing and using different data sources.

Service Data Objects (SDO) aims at providing a data programming architecture and APIs for the Java platform that unifies data programming across data source types. It also provides a robust support for common application patterns and enables applications, tools and frameworks to more easily query, view, bind, update and introspect data. Using SDO implies that developer only need to learn one API irrespective of the type of data source been used or accessed. Some of the data sources supported by SDO include the following:

► Relational databases
► Entry EJB components
► XML pages
► Web services
► Java Connector Architecture

The SDO framework incorporates a good number of J2EE patterns and best practices, making it easy to incorporate proven architecture and designs into applications. For example, most Web applications that access databases cannot be connected to these databases all of the time, SDO supports the disconnected programming model where applications are only connect when needed. Applications developed today are often inherently complex, consisting of numerous layers, and this presents numerous problems that have to be solved, including:

► How will data be stored?
► How will data be passed from component to component?
► How will the data be presented to the end user in a GUI?

The SDO programming model uses patterns which allow a clear and clean separation of each of the these concerns.

8.2 SDO architecture

The SDO architecture consists of three major components, the data object, the data graph and the data mediator. Figure 7.1 outlines the SDO data architecture.
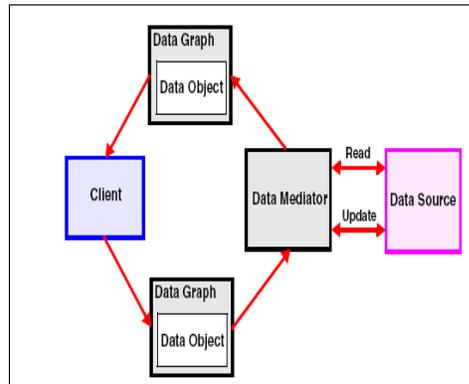
Figure 8-1   SDO architecture

In this section, we take a closer look at the different components in the SDO architecture.

8.2.1 Data mediator services

The data mediator performs the task of connecting applications to data sources. Clients query the data mediator and get a data graph as a response. The data mediator is also responsible for applying changes made to a data graph on behalf of the application. With this architecture in place, applications can deal primarily with data graphs and data objects, providing a layered abstraction between business data and the underlying data source. The data mediator you choose will be dependent of the type that is for a JDBC data source. A JDBC data mediator service will have to be developed.

8.2.2 Data object

Data objects provide an easy way for accessing, traversing and updating structured data.They are SDO representations of structured data. Data objects have a variety of strongly and loosely typed interfaces for querying and updating properties. A data object can also contain other data objects. Data objects are generic and provide a common view of structured data built by the *data mediator service*(DMS). If we consider a JDBC DMS, it usually needs to know the underlying persistence technology, the relational database and how to access and configure it. The SDO clients need not know anything about it.

8.2.3 Data graph

Data graphs provide a container for a tree of data objects. It is a collection of tree-structured or graph-structured data objects. SDO is based on the concept of disconnected data graphs. A client retrieves a data graph from a data source, mutates the graph, and applies the data graph changes made to the data source. A data graph can also include objects representing data from different data sources.

Data graphs are serialized as XML when in transit between application components. The data graph also contains some metadata about the data object, including change summary. The metadata API provide applications, and tools. The data graph also frameworks the ability to introspect the data model for the data graph, enabling applications to handle data from heterogeneous data sources in a uniform way.

8.2.4 Change summary

Change summary is used to represent changes made to a data graph returned by a data mediator service. They are initially empty when returned to a client and are populated as the data graph is modified. Change summaries are used when changes are applied to the backend data store; they allow the data mediator service to update data stores efficiently and incrementally by providing lists of changed properties along with their old values and the created and deleted data objects in the data graph.

8.2.5 Properties, types and sequences

Data objects hold their contents in a series of properties. Each property has an attribute type: a primitive type (int for example); a commonly used data type such as DATE; or if a reference, the type of another object. Each data object provides read and write methods also called getter and setters for its properties. Several overloaded versions of these methods are provided, allowing the properties to be accessed by passing the property name as a string, number as an int or property object itself.

8.3 SDO requirements

The SDO specification includes the following requirements for common elements:

► Dynamic data API

Data objects commonly have typed Java interfaces. It is not always desirable to create interfaces for data objects, for example when transferring data obtained from an SQL output. In this situation, it is necessary to use a dynamic store and associated API. SDO has the ability to represent data objects through a standard dynamic data API.

► Support for static data API

In cases where metatdata is known in advance, such as when the SQL relational schema is known, SDO supports code-generating interfaces for data objects. When the static API is used, the dynamic API is still available for use by the developer. The sample application makes better use of the static API as the data schema is known.

► Complex data objects

Complex or *compound* data objects are data objects that contain other data objects. For example, in the sample application the property table can be considered as a compound data object because it contains a user object and an address object. When dealing with compound objects, change history is harder to implement because inserts, deletes, and reordering have to be tracked as well as simple changes. SDO supports arbitrary graphs of data objects with full change summaries.

► Change history

A client receives a data object from another tier, makes updates to the data object, then passes the data object to the same or another tier. It is necessary for the receiving tier to know what changes were made, so SDO supports full change history.

► Navigation through data graphs

SDO provides navigation capabilities on the dynamic data API.

► Metadata

Many applications are built with the knowledge of the shape of data being returned. This is to say, applications know what data to expect. For frameworks there is a need to introspect on data object metadata to acquire information about a particular data object. SDO provides APIs for metadata because Java reflection does not return sufficient information.

► Relationship integrity

Constraints enable the ability to define relationships between data objects and ways to enforce integrity of those constraints including cardinality, ownership semantics, and so forth. In the sample application, a bid has a relationship with the user table, and, inversely, the bid table contains a list of users. If the user details change in the user table this should be reflected in

the bid table. Data object relationships use regular Java objects as opposed to primary and foreign key relationships.

8.4 SDO versus other technologies

SDO is not the only technology that proposes to solve the problem of data integration in distributed applications. In this section we look at other technologies and frameworks that aim to serve the same purpose. We briefly look at how they fair with SDO. Some of the frameworks mentioned include JDO, JAXB, EMF.

8.4.1 SDO and WebSphere Data Objects

WebSphere Data Objects is the name of an early release of SDO shipped with WebSphere Application Server 5.1. The phrase *WebSphere Data Objects* does not exist anymore because the technology is now referred to as *SDO*.

8.4.2 SDO and JDO

Java Data Object (JDO) has been standardized through the Java community process with release 1.0 and a maintenance release 1.0.1 in May 2004, although a Java specification request for JDO2.0 was filed on 04/20/2004. The specification request has been approved by the Java community process and is now forming an expert group. JDO looks at data programming in the Java environment and provides a common API for accessing different data stores such as databases, file systems, or transaction processing systems. JDO preserves relationships between Java objects and, at the same time, permits concurrent access to the data. JDO is very similar to SDO in that it simplifies and unifies data programming for the Java environment, freeing developers to focus on business logic instead of the underlying technology. The main difference however is that JDO looks at the persistence issue only (J2EE data tier or enterprise information system), while SDO is more general and represents data that can flow between any J2EE tier, for example between the presentation and business tier. For more about JDO, consult:

http://access1.sun.com/jdo/

8.4.3 SDO and EMF

Eclipse modelling framework (EMF) is based on a data model defined using Java interfaces, XML schema, or UML class diagrams. EMF generates a unifying metamodel called Ecore which, in conjunction with the framework, can be used to create high quality implementations of the model. It provides persistence, an

efficient reflective generic object manipulation API, and a change notification framework. It also includes generic reusable classes for building EMF model editor. EMF and SDO both deal with data presentation For more information, consult the reference material included in this book.

8.4.4 SDO and JAXB

JAXB, which stands for Java API for XML Data Binding, was released by the JCP in January 2003. JAXB seeks to represent XML data as Java objects in memory. As an XML binding framework for the Java language, it saves developers the need to parse or create XML documents. It performs:

▶ The marshalling and serializing of Java to XML
▶ The unmarshalling and deserializing of XML to Java

SDO defines a Java binding framework of its own while JAXB is only focused on a Java to XML binding. As outlined earlier in this chapter, SDO offers both static and dynamic binding APIs. However, JAXB only offers static binding APIs.

8.5 SDO example

In this section we describe how to use SDO with our Sal404 sample application. In Chapter 7, "JavaServer Faces" on page 239 we created an implementation of the Sal404 news component that used JSF and we used wizards provided by Rational Web Developer to generate SDO access to our database. We reuse some of this generated SDO code and examine how to replace the existing database access of our Sal404 application with SDO-based database access.

8.5.1 Examining the generated SDO code

Before we write SDO code for use in our sample application, it is important that we understand the SDO API. To do this we can refer to the code generated by the JSF SDO wizards. See 7.7, "SDO API" on page 286 for details of the generated SDO API code.

The main SDO classes are:

▶ DataObject

  – Holds the data graph
  – Uses getList(0) to get a record list
  – Holds parameters
  – Provides get/set methods to access the data by type (getString, getBigDecimal)

- List: simple user class (interface) for record list
    - ECoreEList is an internal EMF class that holds the list
    - Use getEObject to access the data graph (DataObject)
- JDBCMediator
    - Created with connection and XML file
    - Key methods include:
        - getGraph(parameterObject) to retrieve data
        - getEmptyGraph -to prepare for an insert
        - getParameterDataObject to get parameter object to insert real values
        - get/setConnectionWrapper to get the associated JDBC connection
- SDOConnectionWrapper
    - Wraps a JDBC connection
- DataGraph
    - Access to root DataObject and change summary

The following is a typical sequence of code as generated by the SDO wizards:

1. Create SDOConnectionWrapper using the connection name.
2. Create mediator using SDOConnectionWrapper and XML file.
3. Prepare parameters for retrieve.
4. Retrieve data.
5. Retrieve and update values in the data.
6. Commit changes.

8.5.2 Implementing SDO-based data access

In this section we alter the news manager class in our Sal404 sample application to use SDO based data access objects.

Listing news using SDO

We start by using SDO to provide a list of news items. The steps to take are:

1. Copy the XML database definition files.

    a. Expand the **SAL404RealtyWeb** project to folder WebContent\WEB-INF\wdo

    b. Select **listNews.xml**, right-click and choose **Copy**.

    c. Select the **SAL404RealtyJava** project, right-click and choose **Paste**.

2. Change the listNews.xml file so that NEWS_ITM_BODY column is also retrieved. Example 8-1 on page 299 shows the modified XML file.

Example 8-1   listNews.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.websphere.sdo.mediator.jdbc.metadata:Metadata xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.websphere.sdo.mediator.jdbc.metadata="http:///com/ibm/websphere/s
do/mediator/jdbc/metadata.ecore" rootTable="//@tables.0">
  <tables schemaName="DB2ADMIN" name="NEWS_ITM">
    <primaryKey columns="//@tables.0/@columns.0"/>
    <columns name="NEWS_ITM_ID" type="4"/>
    <columns name="NEWS_ITM_TITLE" type="4" nullable="true"/>
    <columns name="NEWS_ITM_DATE" type="10" nullable="true"/>
    <columns name="NEWS_ITM_AUTHOR" type="4" nullable="true"/>
   <columns name="NEWS_ITM_BODY" type="4" nullable="true"/>
  </tables>
  <orderBys column="//@tables.0/@columns.2"/>
</com.ibm.websphere.sdo.mediator.jdbc.metadata:Metadata>
```

3. Modify the app.properties file in the SAL404RealtyJava project to set a flag to determine whether the News database access will use the existing JDBC data access objects, or our new SDO-based data access objects. See Example 8-2 for the definition of the newsItemBackend flag.

Example 8-2   Modify app.properties to provide a newsItemBackend flag

```
# Flag to determine whether to use JDBC or SDO database access for news items
# Valid values are JDBC and SDO
# newsItemBackend=JDBC
newsItemBackend=SDO
```

4. Create a new Java class called SDOListNewsDAO in the com.ibm.itso.sal404.news.dao package of the SAL404RealtyJava package. See Figure 8-2 on page 300.
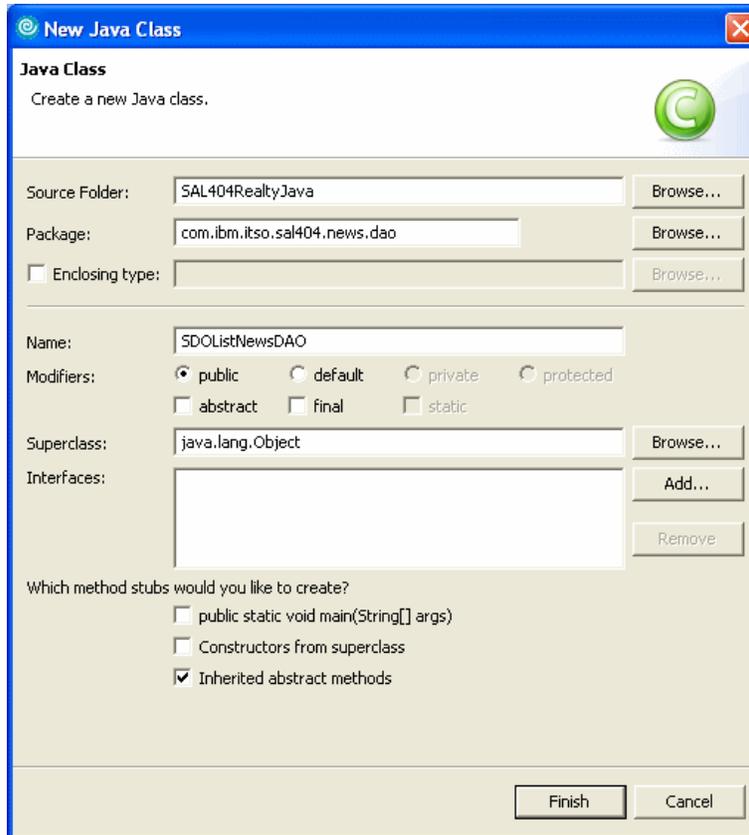
Figure 8-2   Create class SDOListNewsDAO

> **Tip:** When we implement our new SDO-based DAO to list news, we can copy much of the code from the generated ListNews.java from the pagecode.newsJSF package in the SAL404RealtyWeb project.

5. Set up variables for SDO objects as shown in Example 8-3. These can be copied from ListNews.java to SDOListNewsDAO.java.

Example 8-3   SDO variables

```
protected DataObject listNewsParameters;
protected List listNews;
protected JDBCMediator listNewsMediator;
private ConnectionWrapper SDOConnectionWrapper;
```

6. Set up a variable for the XML file as shown in Example 8-4. This can be copied from ListNews.java to SDOListNewsDAO.java and the file path changed to /listNews.xml.

Example 8-4   XML file variable

```
private static final String listNews_metadataFileName = "/listNews.xml";
```

7. Write a method called `getSDOConnectionWrapper()` which will return a SDOConnectionWrapper. We decided not to copy this code from the method in ListNews.java, because the generated code uses a utility class called com.ibm.websphere.sdo.access.connections.ConnectionManager which is not in the classpath for SDOListNewsDAO.java. The original method also uses a hard-coded connection name where we prefer to use connection details from our app.properties file. Our Sal404 example application already implements a utility class called DatabaseConnectionFactory that can make a database connection using parameters from app.properties. Example 8-5 shows our completed `getSDOConnectionWrapper()` method.

Example 8-5   getSDOConnectionWrapper() method

```
protected ConnectionWrapper getSDOConnectionWrapper()
          throws MediatorException, ApplicationException
  {
      logger.info("Entry: getSDOConnectionWrapper");
      if (SDOConnectionWrapper == null)
      {
          logger.info("getSDOConnectionWrapper getting new connection");
          DatabaseConnectionFactory factory = new
           DatabaseConnectionFactory();
          Connection con;
          con = factory.getConnection();
          logger.info("getSDOConnectionWrapper new connection obtained");
          SDOConnectionWrapper = ConnectionWrapperFactoryImpl.soleInstance
                  .createConnectionWrapper(con);
          logger.info("getSDOConnectionWrapper Wrapper obtained");
      }
      return SDOConnectionWrapper;
  }
```

8. Create a mediator of type JDBCMediator by implementing a getListNewsMediator(). The code we used is very similar to that generated for ListNews.java, except that we access the listNews.xml file using getResourceInputStream. See Example 8-6 on page 302 for the complete `getListNewsMediator()` method.

Example 8-6   getListNewsMediator() method

```
public JDBCMediator getListNewsMediator() throws InvalidMetadataException,
            FileNotFoundException, IOException, MediatorException,
            ApplicationException
  {
      logger.info("Entry: getListNewsMediator");
      if (listNewsMediator == null)
      {

          logger.info("Creating new ListNewsMediator");
          listNewsMediator = JDBCMediatorFactoryImpl.soleInstance
                  .createMediator(
                          getResourceInputStream(listNews_metadataFileName),
                          getSDOConnectionWrapper());
          logger.info("Created new ListNewsMediator");

      }
      return listNewsMediator;
  }
```

9. Next we need to retrieve the data graph from the mediator:

    ```
    graph = getListNewsMediator().getGraph();
    ```

    Next, get the list from the graph:

    ```
    listNews = graph.getList(0);
    ```

10. The list contains DataObjects, but our sample application expects to use
    NewsItemDTOs so we next write code to iterate through the list and create a
    NewsItemDTO from each DataObject. We combined the code required for
    steps 9 and 10 into a single method called `buildNewsList()`. Example 8-7
    shows the completed `buildNewsList()` method.

Example 8-7   buildNewsList() method

```
private Vector buildNewsList() throws InvalidMetadataException,
            FileNotFoundException, MediatorException, IOException,
            ApplicationException
  {
      logger.info("Entry: buildNewsList");
      Vector theList = new Vector();
      DataObject graph;

      graph = getListNewsMediator().getGraph();
      listNews = graph.getList(0);
      for (Iterator iter = listNews.iterator(); iter.hasNext();)
```

```
        {
            DataObject element = (DataObject) iter.next();
            NewsItemDTO item = new NewsItemDTO();
            item.setAuthor(element.getString("NEWS_ITM_AUTHOR"));
            item.setBody(element.getString("NEWS_ITM_BODY"));
            item.setDate(element.getDate("NEWS_ITM_DATE"));
            item.setTitle(element.getString("NEWS_ITM_TITLE"));
            item.setId(element.getString("NEWS_ITM_ID"));
            theList.addElement(item);
        }

        return theList;

    }
```

11. The `buildNewsList()` is called from the public method `listNewsItems()` in our SDOListNewsDAO. See Example 8-8 for the code in the `listNewsItems()` method.

Example 8-8   istNewsItems() method

```
public Vector listNewsItems() throws ApplicationException
    {
        Vector allNews = new Vector();
        try
        {
            allNews = this.buildNewsList();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            logger.error(getClass().getName() + " listNewsItems: " +
e.getMessage());

            ApplicationException ae = new ApplicationException();
            ae.setStrutsMessage("news.error.listFailed");
            throw ae;
        }

        return allNews;
    }
```

12. After competing the SDOListNewsDAO class, we altered the NewsManager class to call the SDO list news DAO when the newsItemBackend=SDO property is set in our application.properties file. To do this, we altered the

viewNewsItem() method in NewsManager to test whether to use JDBC or SDO for the data access. See Example 8-9.

> **Note:** We have put this test as a method in the NewsManager class because it allows us to progressively move from JDBC to SDO and to mix both types of access. We do not suggest that you follow this approach in production systems where it is likely that you will do all your news data access one way or the other.

Example 8-9   viewNewsItem() method

```
public Vector viewNewsItem() throws ApplicationException
    {
        logger.info("ENTRY: viewNewsItem");
        Vector vectorOfNewsItem = new Vector();
        if (useJDBC)
            vectorOfNewsItem = listNewsItemsJDBC();

        if (useSDO)
            vectorOfNewsItem = listNewsItemsSDO();

        return vectorOfNewsItem;
    }
```

13.We then defined a listNewsItemsSDO() method in NewsManager that calls the method listNewsItems() in SDOListNewsDAO. See Example 8-10.

Example 8-10   listNewsItemsSDO() method

```
private Vector listNewsItemsSDO() throws ApplicationException
    {
        logger.info("Entry: listNewsItemsSDO");
        //      Vector for all news items
        Vector vectorOfNewsItem = new Vector();

        //Create the DAO object
        SDOListNewsDAO newsItemDAO = new SDOListNewsDAO();
        vectorOfNewsItem = newsItemDAO.listNewsItems();
        logger.info("Exit: listNewsItemsSDO");
        return vectorOfNewsItem;

    }
```

14. This completes the code needed to list news items using SDO. You can test this by making sure that the newsItemBackend=SDO property is set in your application.properties file and then using the news functions of the Sal404 sample application.

Modifying news using SDO

In this section, we use SDO to modify a news item. The code we used is very similar to that used for listing news items and we can also copy code from the generated ModifyNewsItem.java from the pagecode.newsJSF package in the SAL404RealtyWeb project. The steps to take are:

1. Copy the XML database definition files newsItem.xml from the SAL404RealtyWeb project folder WebContent\WEB-INF\wdo to the SAL404RealtyJava project.

2. Change the newsItem.xml file so that the filter argument does not use a parameter from the request scope.

   Change:

   ```
   <filter predicate="( NEWS_ITM_ID = ? )">
           <filterArguments name="requestScopenewsId" type="4"/>
   </filter>
   ```

   To:

   ```
   <filter predicate="( NEWS_ITM_ID = ? )">
           <filterArguments name="newsId" type="4"/>
       </filter>
   ```

3. . Example 8-11 shows the modified XML file.

Example 8-11   newsItem.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.websphere.sdo.mediator.jdbc.metadata:Metadata xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.websphere.sdo.mediator.jdbc.metadata="http:///com/ibm/websphere/s
do/mediator/jdbc/metadata.ecore" rootTable="//@tables.0">
  <tables schemaName="DB2ADMIN" name="NEWS_ITM">
    <primaryKey columns="//@tables.0/@columns.0"/>
     <columns name="NEWS_ITM_ID" type="4"/>
     <columns name="NEWS_ITM_TITLE" type="4" nullable="true"/>
     <columns name="NEWS_ITM_DATE" type="10" nullable="true"/>
     <columns name="NEWS_ITM_AUTHOR" type="4" nullable="true"/>
     <columns name="NEWS_ITM_BODY" type="4" nullable="true"/>
```

```
    <filter predicate="( NEWS_ITM_ID = ? )">
      <filterArguments name="newsId" type="4"/>
    </filter>
  </tables>
</com.ibm.websphere.sdo.mediator.jdbc.metadata:Metadata>
```

4. Create a new Java class called `SDOModifyNewsDAO` in the com.ibm.itso.sal404.news.dao package of the SAL404RealtyJava package.

> **Tip:** When we implement our new SDO based DAO to modify news we can copy much of the code from the generated ModifyNewsItem.java from the pagecode.newsJSF package in the SAL404RealtyWeb project. Many of the changes we need to make are also very similar to this we did when creating the SDOListNewsDAO.

5. Set up variables for SDO objects as shown in Example 8-12.

Example 8-12   SDO variables for SDOModifyNewsDAO

```
final String parameter = "newsId";

 private DataObject newsItem;

 protected DataObject newsParameters;

 protected JDBCMediator newsMediator;

 private ConnectionWrapper SDOConnectionWrapper;

 private static final String modifyNews_metadataFileName = "/newsItem.xml";
```

6. Write a method called `getSDOConnectionWrapper()` which will return a SDOConnectionWrapper. This can be copied from SDOListNewsDAO.

7. Create a mediator of type JDBCMediator by implementing a getNewsMediator(). The code we used is very similar to that we wrote for SDOListNewsDAO.

8. Create a method called FetchNewsItem(String newsId) which does the following:

   a. Creates a parameter data object that is used to select the correct news item we want to modify.

   ```
   newsParameters = getNewsMediator().getParameterDataObject();
   newsParameters.set(parameter, newsId);
   ```

b. Uses the data mediator and the parameter object to build a data graph of news items.

```
graph = getNewsMediator().getGraph(newsParameters);
```

c. Extracts the first data object from the data graph because this will be the news item we want to modify.

```
setNewsItem((DataObject) graph.getList(0).get(0));
```

9. Write a method called modifyNewsItem() that will apply any changes to our data object. Example 8-13 shows the completed method.

Example 8-13   modifyNewsItem() method

```
private void modifyNewsItem() throws InvalidMetadataException,
FileNotFoundException, MediatorException, IOException, ApplicationException,
SQLException
    {
        getNewsMediator().applyChanges(
                getNewsItem().getDataGraph().getRootObject());
}
```

10.Write a method that takes data from the input NewsItemDTO and populates the news item data object. See Example 8-14.

Example 8-14   modifyNews(NewsItemDTO tempNewsItemDTO) method

```
/**
     * @param tempNewsItemDTO
     */
    public void modifyNews(NewsItemDTO tempNewsItemDTO)
    {
        try
        {
            fetchNewsItem(tempNewsItemDTO.getId());
            DataObject news = getNewsItem();
            news.setString("NEWS_ITM_TITLE",tempNewsItemDTO.getTitle());
            news.setString("NEWS_ITM_AUTHOR",tempNewsItemDTO.getAuthor());
            news.setString("NEWS_ITM_BODY",tempNewsItemDTO.getBody());
            news.setDate("NEWS_ITM_DATE",tempNewsItemDTO.getDate());
            setNewsItem(news);
            modifyNewsItem();


        } catch (InvalidMetadataException e)
        {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        } catch (FileNotFoundException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MediatorException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ApplicationException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

11. After competing the SDOModifyNewsDAO class, alter the NewsManager class to call the SDO modify news DAO when the newsItemBackend=SDO property is set in our application.properties file.

   To do this we altered the modifyNewsItem(NewsItemDTO tempNewsItemDTO) method in NewsManager to test whether to use JDBC or SDO for the data access. See Example 8-15.

Example 8-15   modifyNewsItem(NewsItemDTO tempNewsItemDTO) method

```
public void modifyNewsItem(NewsItemDTO tempNewsItemDTO)
            throws ApplicationException
    {
        logger.info("ENTRY: modifyNewsItem");
        if (useJDBC)
            modifyNewsItemJDBC(tempNewsItemDTO);

        if (useSDO)
            modifyNewsItemSDO(tempNewsItemDTO);
        return;
    }
```

12.We then define a modifyNewsItemSDO(NewsItemDTO tempNewsItemDTO) method in NewsManager that calls the method modifyNews(tempNewsItemDTO) in SDOModifyNewsDAO. See Example 8-16.

Example 8-16   modifyNewsItemsSDO(NewsItemDTO tempNewsItemDTO) method

```
private void modifyNewsItemSDO(NewsItemDTO tempNewsItemDTO)
    {
        logger.info("ENTRY: modifyNewsItemSDO");
        SDOModifyNews dao = new SDOModifyNews();
        dao.modifyNews(tempNewsItemDTO);

    }
```

13.This completes the code needed to modify news items using SDO. You can test this modification by making sure that the newsItemBackend=SDO property is set in your application.properties file and then using the news functions of the Sal404 sample application.

9

Enterprise JavaBeans

This chapter describes the reasons to consider using Enterprise JavaBeans (EJB) and then works through the process of creating and referencing an EJB.

The use of EJBs in applications has been described and debated extensively in print and on the Internet. While references to this documentation are provided in this chapter, the goal is to provide a clear, presentation and understandable motive for using EJBs.

9.1 Why use Enterprise JavaBeans?

The Sal301 sample application presented in *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 is quite simple and demonstrates quite well what can be accomplished with a Web container. A Web container processes JavaServer Pages(JSP), manages users HTTP sessions, and provides a variety of mechanisms for interacting with back-end systems. Basically, a Web container in and of itself provides all of the infrastructure required for a three-tier architecture: presentation, business logic and persistence layers.

What more do you need?

The limitations of the Web container become apparent in large and complex systems. The HTTP session tends to grow very large, becoming a resource consumption problem when a large number of users access the system. There also tends to be substantial duplication of data in the HTTP sessions.

The benefits of using EJBs include better scalability and layering of applications. The Web container becomes associated with only the presentation layer, and the J2EE container contains the business logic and persistence layers.

> **Important:** When generating EJBs, it is important to first create a new source folder in the EJB project. If you do not create the new source folder first, then your only option available for creating the source folder in the wizards is ejbModule. This is the default EJB deployment source folder. This folder is not shared with source code repositories and is meant to contain only deployment artifacts.

9.2 The EJB architecture

This section provides a brief overview of the key features of the Enterprise JavaBean (EJB) architecture. You can read a more detailed summary in the redbook *Rational Application Developer V6 Programming Guide*, SG24-6449.

EJB is an architecture for server-centric, component-based, distributed object-oriented business applications written in the Java programming language.

The EJB architecture depicted in Figure 9-1 on page 313 reduces the complexity of developing business components by providing automatic, or nonprogrammatic, support for system level services, thus freeing developers to concentrate on the development of business logic.
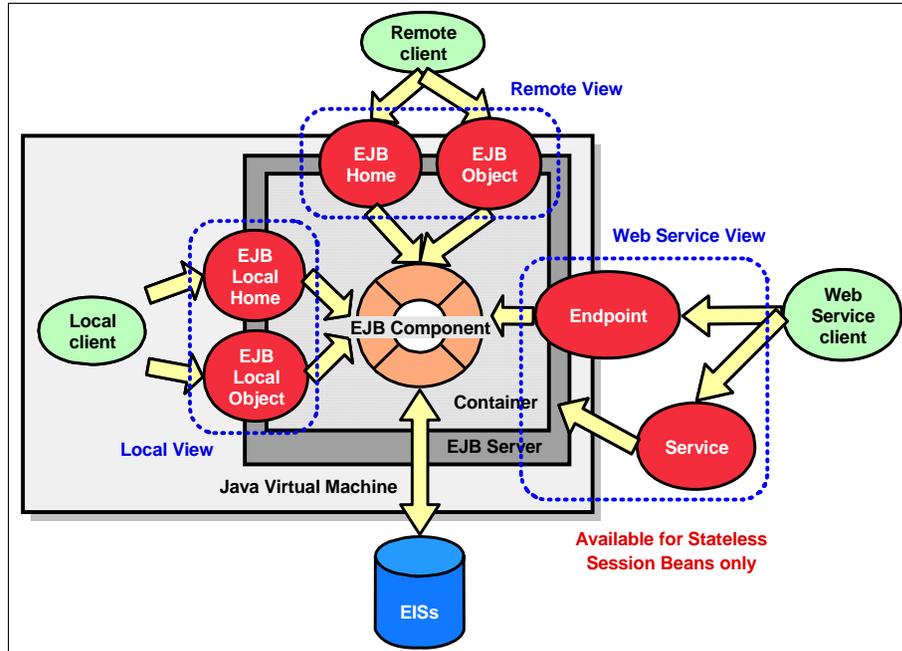
Figure 9-1   EJB architecture overview

In the following sections we briefly explain each of the EJB architecture elements depicted in:

► EJB server
► EJB container
► EJB components

9.2.1 EJB server

An *EJB server* is the part of an application server that hosts EJB containers. It is sometimes referred to as an *Enterprise Java Server* (EJS).

The EJB server provides the implementation for the common services available to all EJBs. The EJB server's responsibility is to hide the complexities of these services from the component requiring them. The EJB specification outlines eight services that must be provided by an EJB server:

► Naming
► Transaction
► Security
► Persistence
► Concurrency

- ► Life cycle
- ► Messaging
- ► Timer

9.2.2 EJB container

The *EJB container* functions as a runtime environment for enterprise beans by managing and applying the primary services that are needed for bean management at runtime. In addition to being an intermediary to the services provided by the EJB server, the EJB container also provides for EJB instance life cycle management and EJB instance identification. EJB containers create bean instances, manage pools of instances, and destroy them.

Containers are transparent to the client in that there is no client API to manipulate the container, and there is no way for a client to tell in which container an enterprise bean is deployed.

One of the container's primary responsibilities is to provide the means for remote clients to access components inside them. Remote accessibility enables remote invocation of a native component by converting it into a network component. EJB containers use the Java RMI interfaces to specify remote accessibility to clients of the EJBs.

The responsibilities that an EJB container must satisfy can be defined in terms of the primary services. Specific EJB container responsibilities are as follows:

- ► Naming
- ► Transaction
- ► Security
- ► Persistence
- ► Concurrency
- ► Life cycle
- ► Messaging
- ► Timer

9.2.3 EJB components

EJB components run inside an EJB container, their runtime environment. The container offers life-cycle services to these components, and provides them with an interface to the EJB server. It also manages the connections to the Enterprise Information Systems (EIS), including databases and existing systems.

Client views

For client objects to send messages to an EJB component, the component must provide a view. A view is a client interface to the bean, and can be local or remote:

- ► A local view can be used only by local clients to access the EJB. *Clients* in this case means that they reside in the same JVM as the server component.

- ► With a remote view, any client, possibly distributed, can access the component.

The motivation for local interfaces is that remote calls are more expensive than local calls. Which one to use is influenced by how the bean itself is to be used by its clients because local and remote depict the clients' view of the bean. An EJB client might be a remote client, such as a servlet running on another process, or might be a local client, such as another EJB in the same container.

EJB types

There are three main types of EJBs: entity beans, session beans, and message-driven beans (see Figure 9-2).
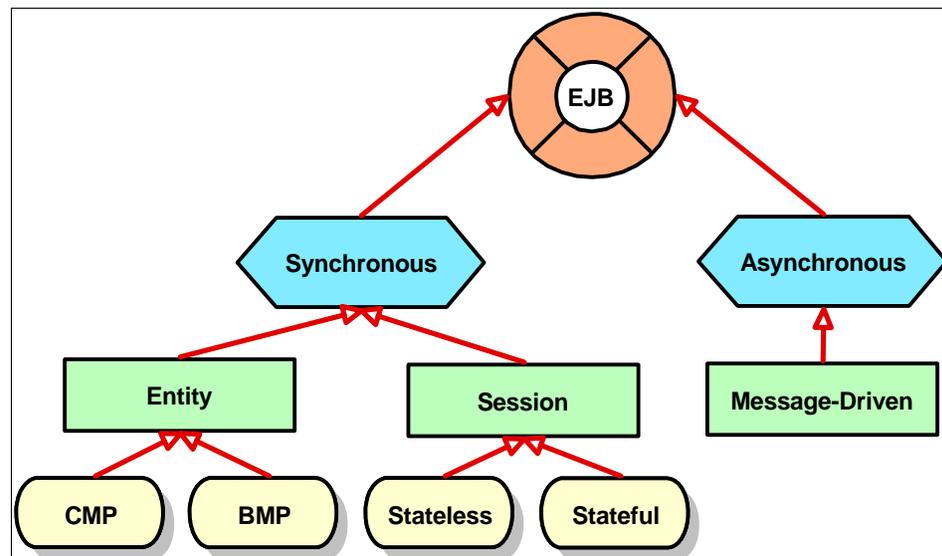
Figure 9-2   EJB types

- ► *Entity beans* are modeled to represent business or domain-specific concepts, and are typically the nouns of your system, such as *customer* and *account*. Entity beans are persistent. They maintain their internal state (attribute values) between invocations and across server restarts. Because of their persistent nature, entity beans usually represent data stored in a database.

- *Session beans* are modeled to represent a task or workflow of a system, and to provide coordination of those activities. A session bean is commonly used to implement the facade of EJB modules. Although some session beans might maintain state data, this data is not persistent; it is just conversational.

  Session beans can either be *stateless* or *stateful*. Stateless session beans are beans which maintain no conversational state, and are pooled by the container to be reused. Stateful session beans are beans that keep track of the conversational state with a specific client. Thus, they cannot be shared among clients.

- Like session beans, *message-driven beans* (MDB) can also be modeled to represent tasks. However, they are invoked by the receipt of asynchronous messages, instead of synchronous ones. The bean either listens for or subscribes to messages that it is to receive.

Entity and session beans are accessed synchronously through a remote or local EJB interface method invocation. This is referred to as *synchronous invocation*. When a client makes an invocation request, it will be blocked, waiting for the return. Clients of EJBs invoke methods on session and entity beans. An EJB client can be remote (servlet), or local (another EJB within the same JVM).

Message-driven beans are not accessible through remote or a local interfaces. The only way for an EJB client to communicate with a message-driven bean is by sending a JMS message. This is an example of asynchronous communication. The client does not invoke the method on the bean directly, but rather uses JMS constructs to send a message. The container delegates the message to a suitable message-driven bean instance to handle the invocation. EJBs of any type can also be accessed asynchronously by means of a timer event, started by the EJB Timer Service.

9.2.4 Using stateless session EJBs

Our Sal404 sample application relies heavily on reference data. This includes:

- Property types
- Property statuses
- Country codes
- User roles

WebSphere Application Server - Express V5 did not provide an EJB container. Therefore, all reference data for the Sal301 sample application was stored in the HTTP session. This leads to large session sizes when a large amount of reference data is used. A common solution to this problem is to use a singleton to manage reference data. A *singleton* is a class that can be instantiated only once and cannot be an interface. The use of singletons however, generally results in a new set of problems to be solved. These include access concurrency

control and management. It would certainly be undesirable to have to restart the Web container in order to refresh reference data.

An EJB container provides for pooling, caching, access control and life cycle management for EJBs. With the pooling and caching provided by the container, a small pool of stateless session EJBs can service a large number of clients.

The Rational Application Developer tooling make the creation of EJBs very simple. In order to create the reference data EJBs, we simply extend an existing class and expose the reference data functionality with an EJB.

The ReferenceDataHelper is a plain old Java object (POJO) that uses our application DAOs application to query the database.

Creating the stateless session EJB

To create an EJB that will handle our application reference data, follow these steps:

1. Choose **File** → **New** → **Other,** and select **EJB** → **Enterprise Bean** as shown in Figure 9-3. Click **Next**.
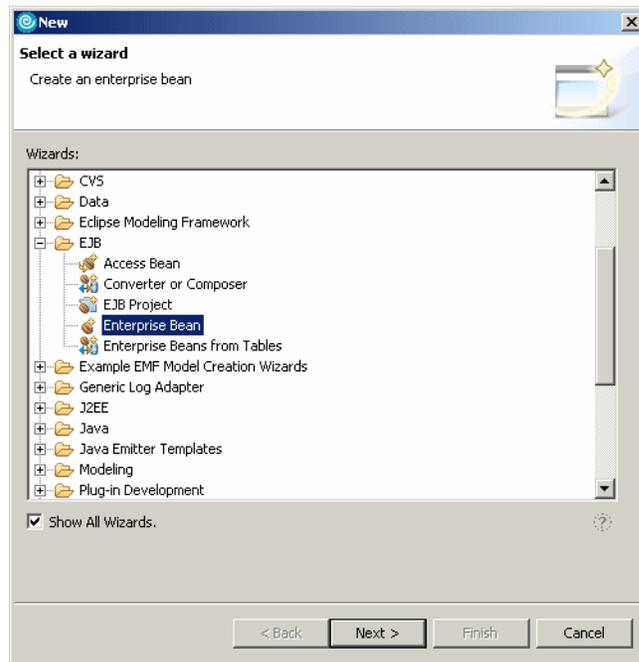
Figure 9-3   Getting started

2. On the next page of the wizard make sure that **Session bean** is selected (the default), that the EJB project name is correct, and enter a name for the new EJB. See Figure 9-4 for an example.
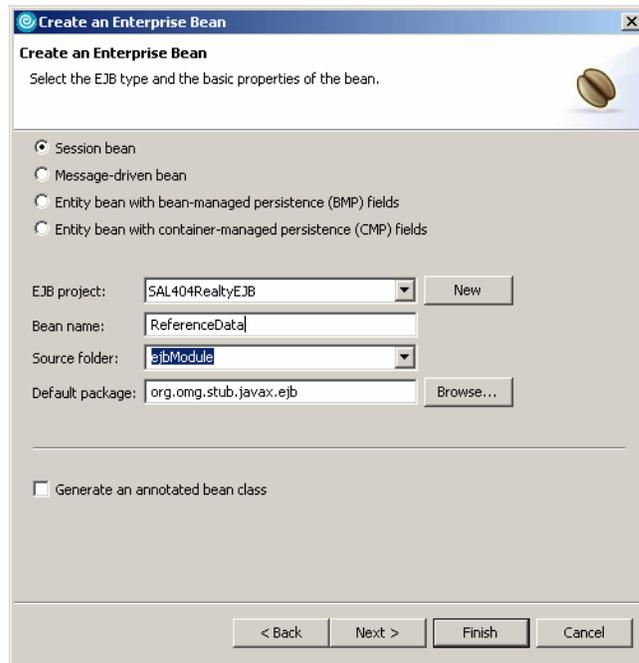
Figure 9-4   Enter bean type

3. Choose the client view for your new EJB. See Figure 9-5 on page 319. Select **Bean** as the transaction type. The default transaction type allows the J2EE container to manage transactions, but the DAOs in our application explicitly set auto commit on the JDBC database connection. This means that the EJB container cannot manage the transaction. The bean must do it. Here, this is of little concern because the bean only performs database reads.

   The default behavior for the wizard is to create only the remote client view. We want to also create the local client view. Both client views will be created, although the application will only use the local client view. Using the local client view, the application can use pass-by-reference for data transfer. A remote client view would need to serialize and deserialize the reference data lists.

   Because the application is not intended to be deployed to cluster environments, use of the local client view is preferred. Even in the case of clustered environments, each node in the cluster would have a reference data system. There is no reason to transport reference data over a network linking the nodes in the cluster.
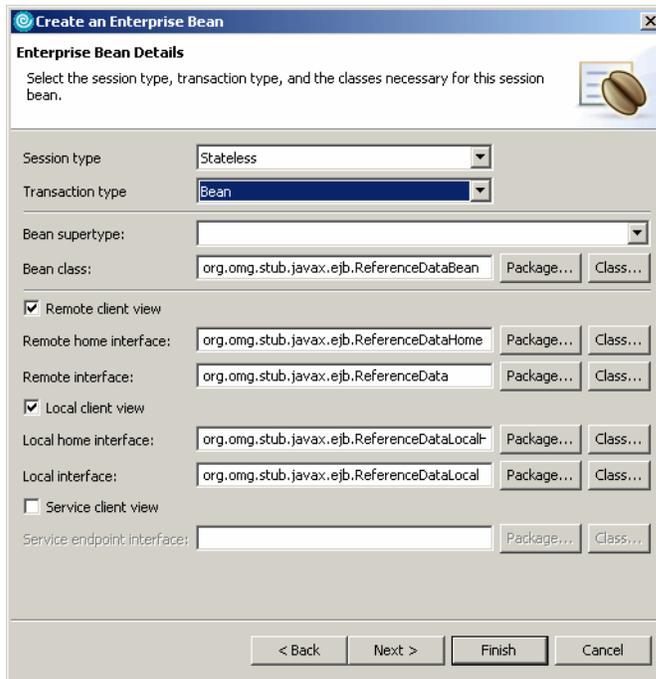
Figure 9-5   Chose client views

4. Select the bean superclass as shown in Figure 9-6 on page 320. We are extending the ReferenceDataHelper class.
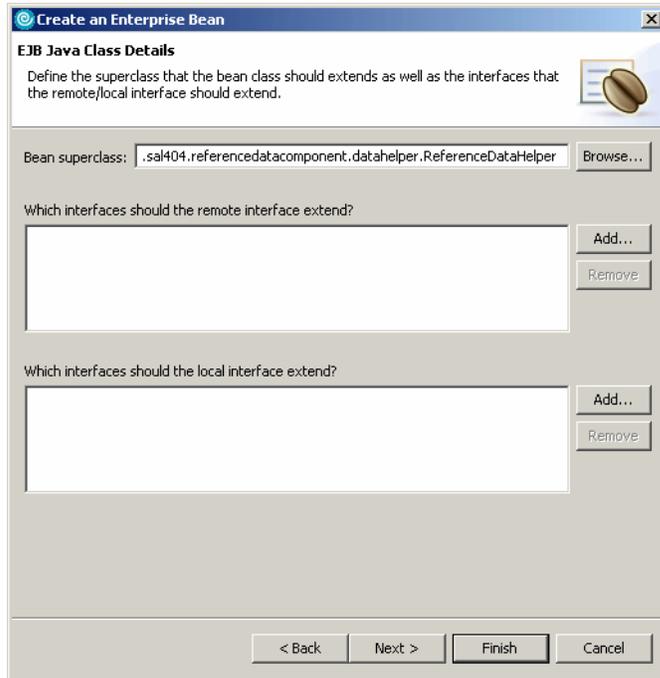
Figure 9-6   Select bean superclass

5. Place the visualization diagram in the default.dnx and click **Finish**. See Figure 9-7 on page 321.
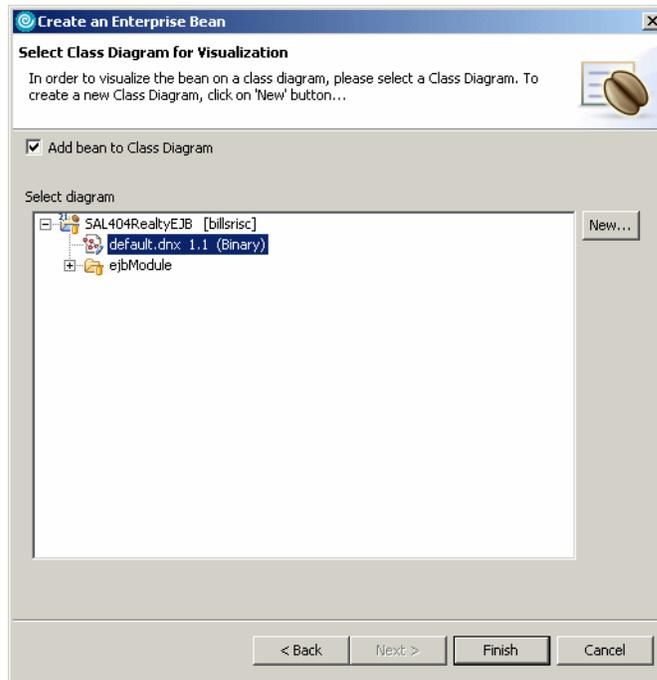
Figure 9-7   Place the visualization in the default.dnx

6. Navigate to the newly created ReferenceDataBean in the SAL404RealtyEJB
   project and open the ReferenceDataBean.java file. See Figure 9-8 on
   page 322. Right-click in the Java editor and choose **Source** →
   Override/Implement Methods. The newly generated code will be inserted at
   the cursor, so place the cursor at the end of the class.
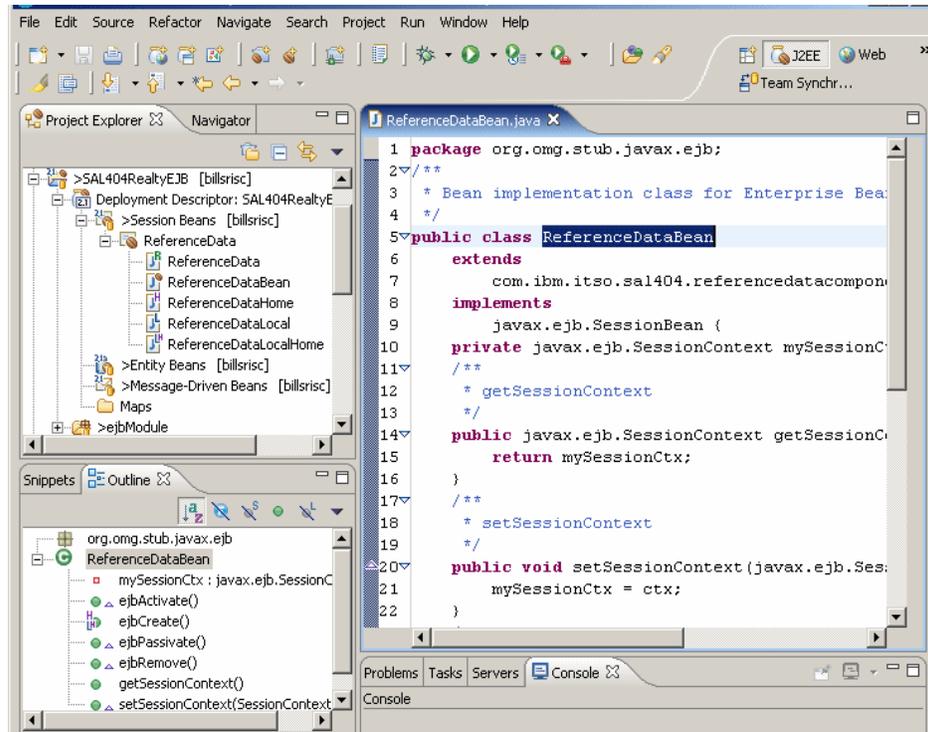
Figure 9-8   ReferenceDataBean

7. Select all three methods to implement and click **OK**, as in Figure 9-9 on page 323.
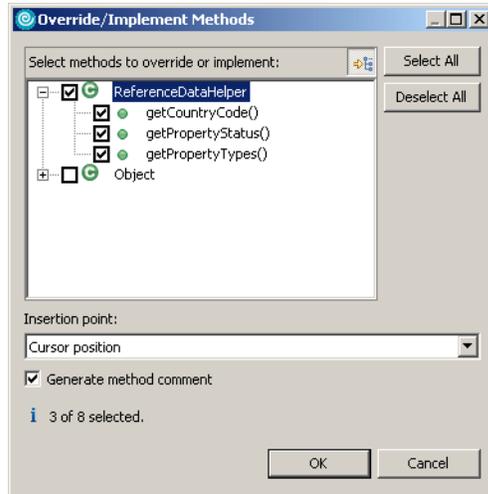
Figure 9-9   Override methods

8. Promote the new methods to the local and remote interfaces as shown in
    Figure 9-10 on page 324.
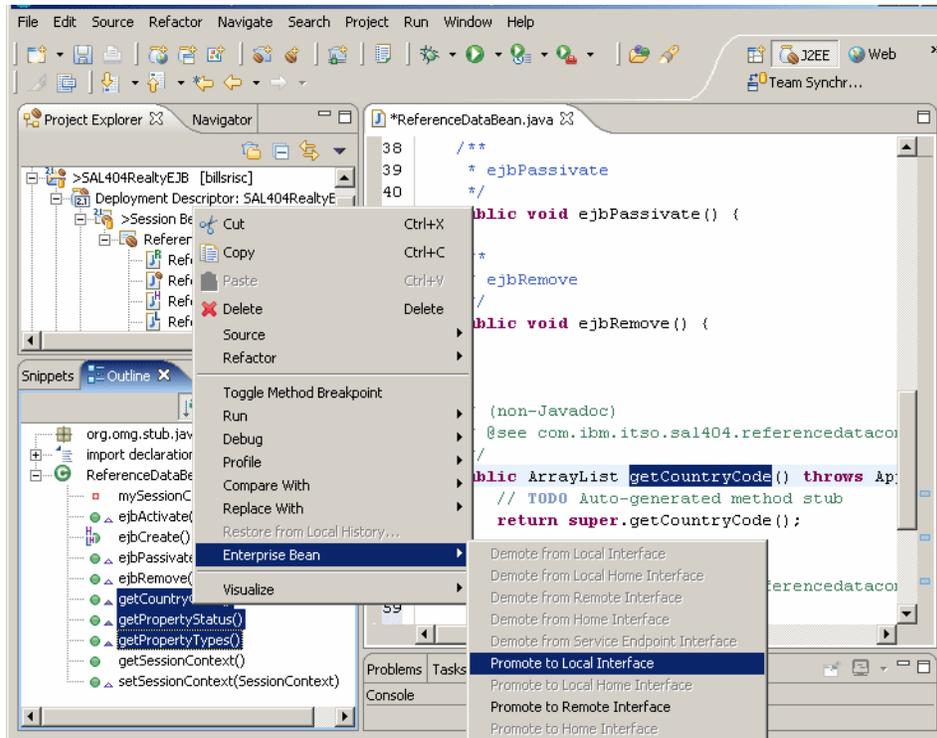
Figure 9-10   Promote the new methods to the local and remote interfaces

This completes the creation of the reference data EJB.

As methods are added to the ReferenceDataHelper (for new reference data types), repeat the process of overriding the methods in the ReferenceDataBean and promoting them to the EJB local and remote interfaces. If there are errors from the process, these can be cleared by right-clicking the EJB deployment descriptor and selecting **Deploy**.

As well as being relatively simple, this process has several advantages. In this case, the ReferenceDataHelper is a plain old Java object (POJO). It can be tested as a simple Java class. Any new methods can simply be added to the EJB interface. Note also that everything in the EJB project is generated code. The Java class that has the functionality is in a Java project. This makes project organization simpler and saves developers from having to look at all of the generated code if they are not interested in it.

One other consequence is that the deployment code will have warnings about unused imports. It is preferable to have no warnings in the code.

9. To hide the warnings in the EJB project, launch the project properties dialog box and set the Unused imports property to **Ignore**. See Figure 9-11.
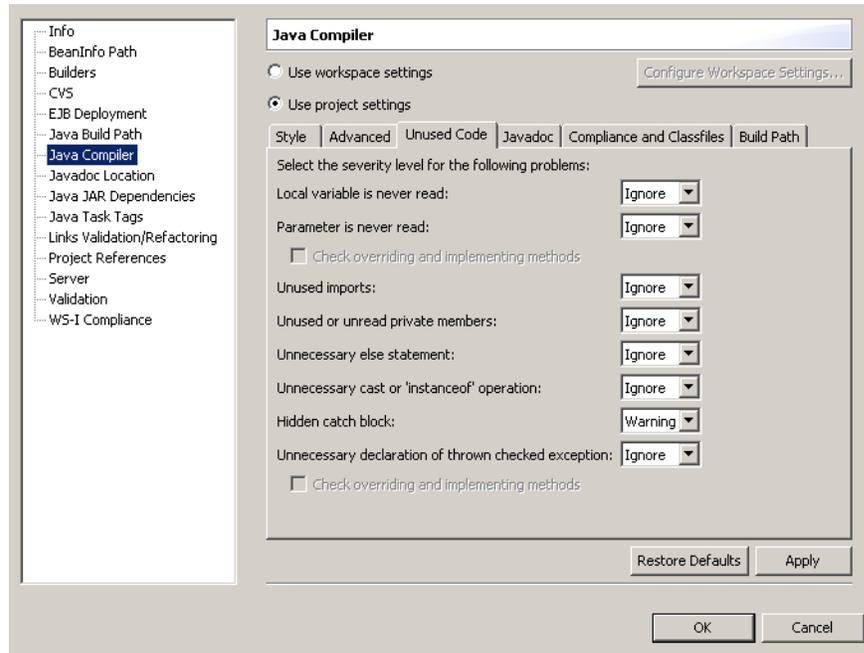
Figure 9-11   Hide warnings in the EJB project

The Java compiler settings are on a project-by-project basis. Because the POJO is in a Java project, the warnings for unused imports can be left on in that project.

Exposing the EJB to the sample application

In order to expose the reference data EJB functionality to the Sal404 application, we will use our standard layered design to create a new manager. The steps are:

1. Create a new ReferenceDataManager class in the com.ibm.itso.sal404.referencedata package of the SAL404RealtyControl Java project.

2. Open the new class in the Java editor and from the snippets view, select the **EJB** folder and then choose **call a Session bean service method**. See Figure 9-12 on page 326.
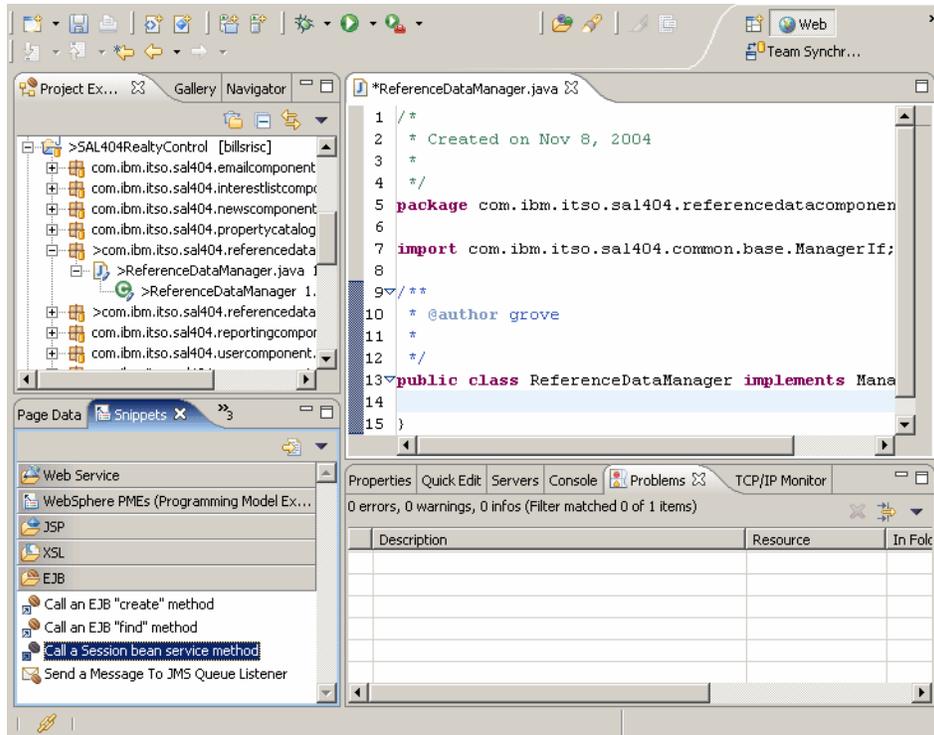
Figure 9-12 Select the EJB snippets

3. Select the **ReferenceData** EJB as shown in Figure 9-13 and click **Next**.
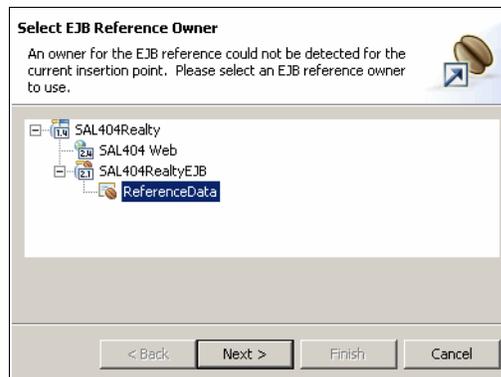
Figure 9-13 Select the EJB

4. Select the EJB reference **ejb/ReferenceData** as shown in Figure 9-14 on page 327 and click **Next**.
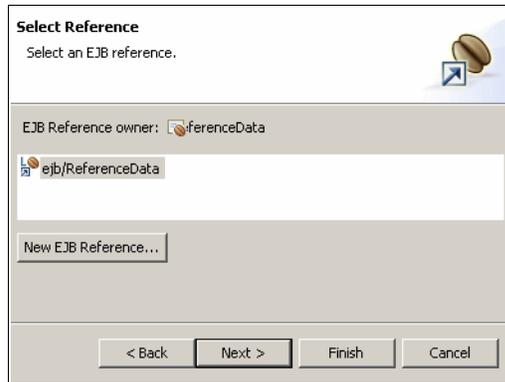
Figure 9-14   Select the EJB reference

5. Select the **getPropertyTypes()** method and click **Finish,** as shown in
    Figure 9-15.
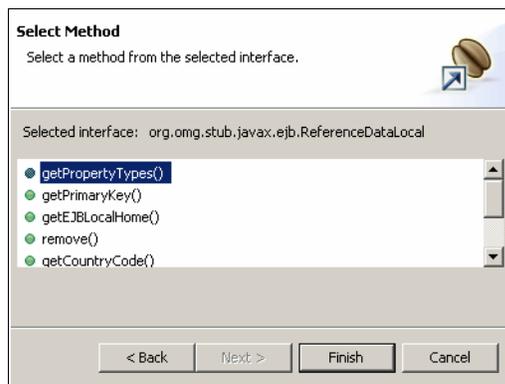
Figure 9-15   Select the method

6. The snippet that is placed in the code needs to be placed into a method.
    Repeat the snippet wizard for the remaining methods in the reference data
    EJB. The completed code is shown in Example 9-1.

Example 9-1   EJB references in the ReferenceDataManager

```
public ArrayList getPropertyStatus() {
        ReferenceDataLocal aReferenceDataLocal = createReferenceDataLocal();
        ArrayList anArrayList = null;
        try {
            anArrayList = aReferenceDataLocal.getPropertyStatus();
        } catch (ApplicationException ex) {
            ex.printStackTrace();
        }
```

```
            return anArrayList;
        }

        public ArrayList getPropertyTypes() {
            ReferenceDataLocal aReferenceDataLocal = createReferenceDataLocal();
            ArrayList anArrayList = null;
            try {
                anArrayList = aReferenceDataLocal.getPropertyTypes();
            } catch (ApplicationException ex) {
                ex.printStackTrace();
            }
        return anArrayList;
        }

        public ArrayList getCountryCode() {
            ReferenceDataLocal aReferenceDataLocal = createReferenceDataLocal();
            ArrayList anArrayList = null;
            try {
                anArrayList = aReferenceDataLocal.getCountryCode();
            } catch (ApplicationException ex) {
                ex.printStackTrace();
            }
        return anArrayList;
        }

        public ArrayList getUserRoles() {
            ReferenceDataLocal aReferenceDataLocal = createReferenceDataLocal();
            ArrayList anArrayList = null;
            try {
                anArrayList = aReferenceDataLocal.getUserRoles();
            } catch (ApplicationException ex) {
                ex.printStackTrace();
            }
        return anArrayList;
        }
```

7. The serviceLocatorMgr.jar needed for the createReferenceDataLocal() method is automatically added to the EAR as a utility jar. The manifest classpath for the EJB project is also automatically updated.

8. The final step is to tell the Web project that the EJB exists and what its JNDI name is by adding a reference in the Web projects deployment descriptor. See Figure 9-16 on page 329.
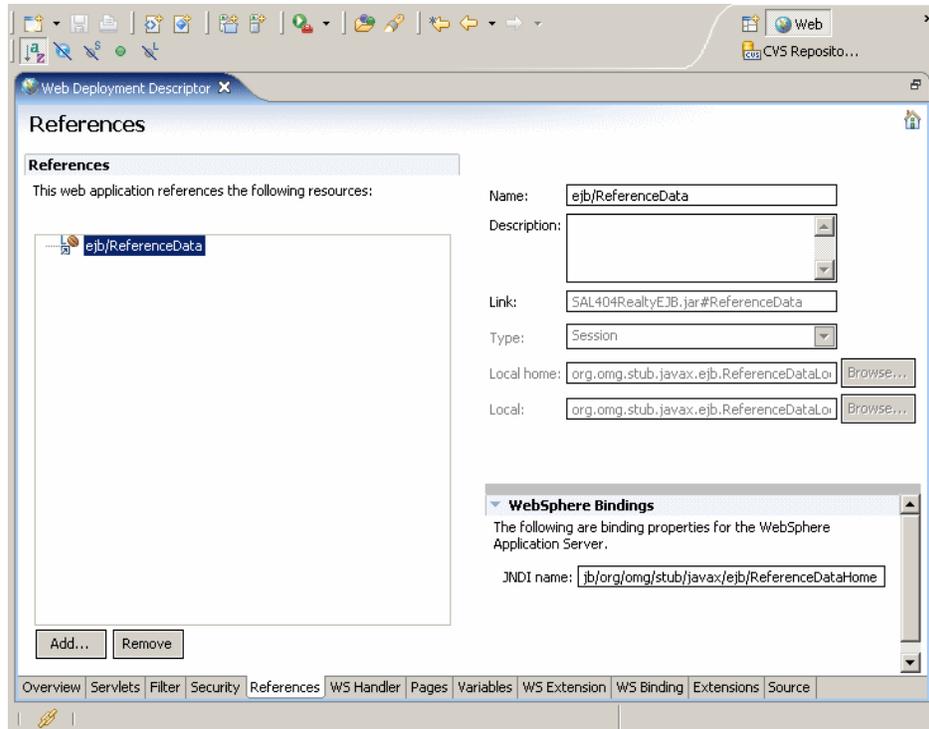
Figure 9-16   An EJB reference in the Web project deployment descriptor

9. The property management JSPs and the user management JSPs were then updated to use the new reference data system. A code snippet for using Struts tags for combo boxes is show in Example 9-2. This snippet is from the searchPropertiesByCriteria.jsp.

Example 9-2   Struts tags for combo boxes using reference data

```
<html:select property="propertyTypeId">
   <html:option value="0">- select -</html:option>
   <html:optionsCollection value="id" label="description"
name="referenceSessionData"
property="propertyTypes" />
</html:select>
```

9.2.5 Create a database connection

As a prerequisite to creating an entity EJB that accesses our database, we need to create a database connection in Rational Application Developer. An example of how to do this is as follows:

1. Switch to the Data perspective and right-click anywhere in the **Database Explorer** view. Choose **New Connection** as shown in Figure 9-17.
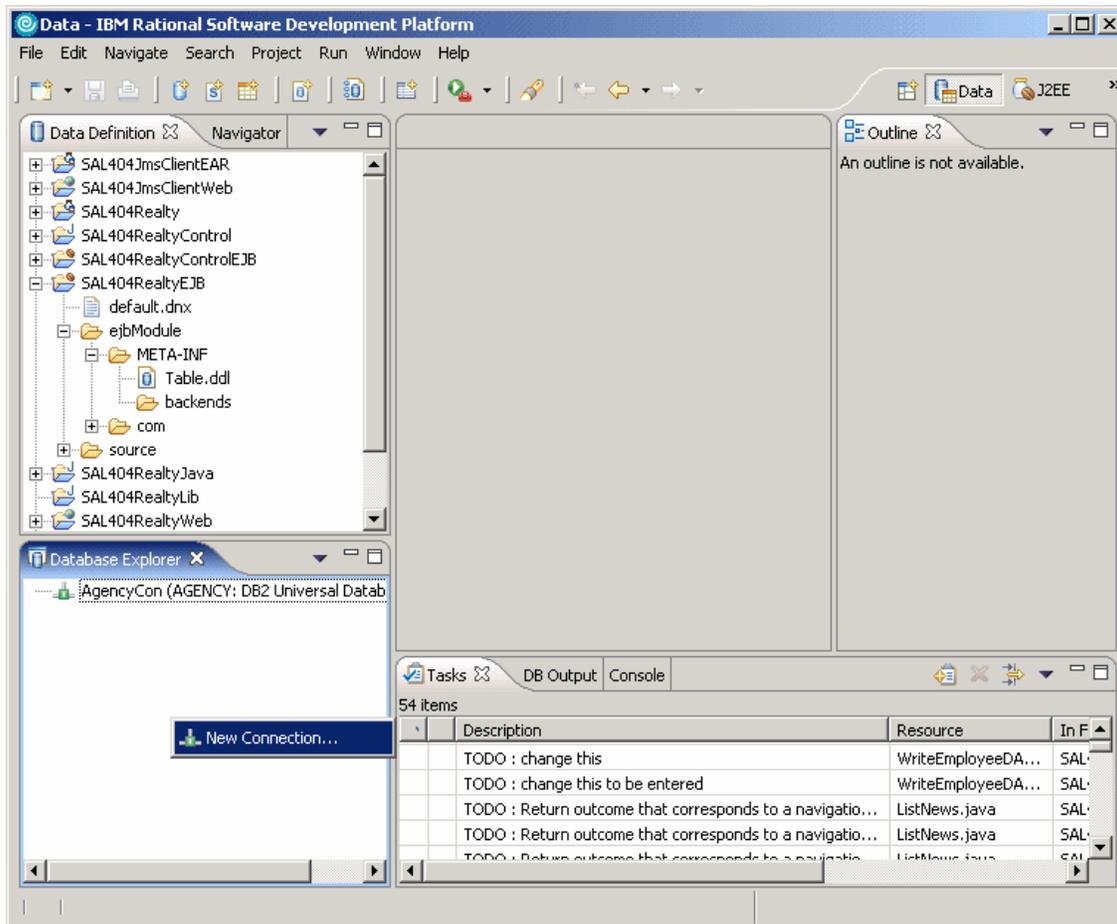
Figure 9-17   Create new database connection

2. Enter a connection name and check **Choose a DB2 alias**. Click **Next**. See Figure 9-18 on page 331 for an example.
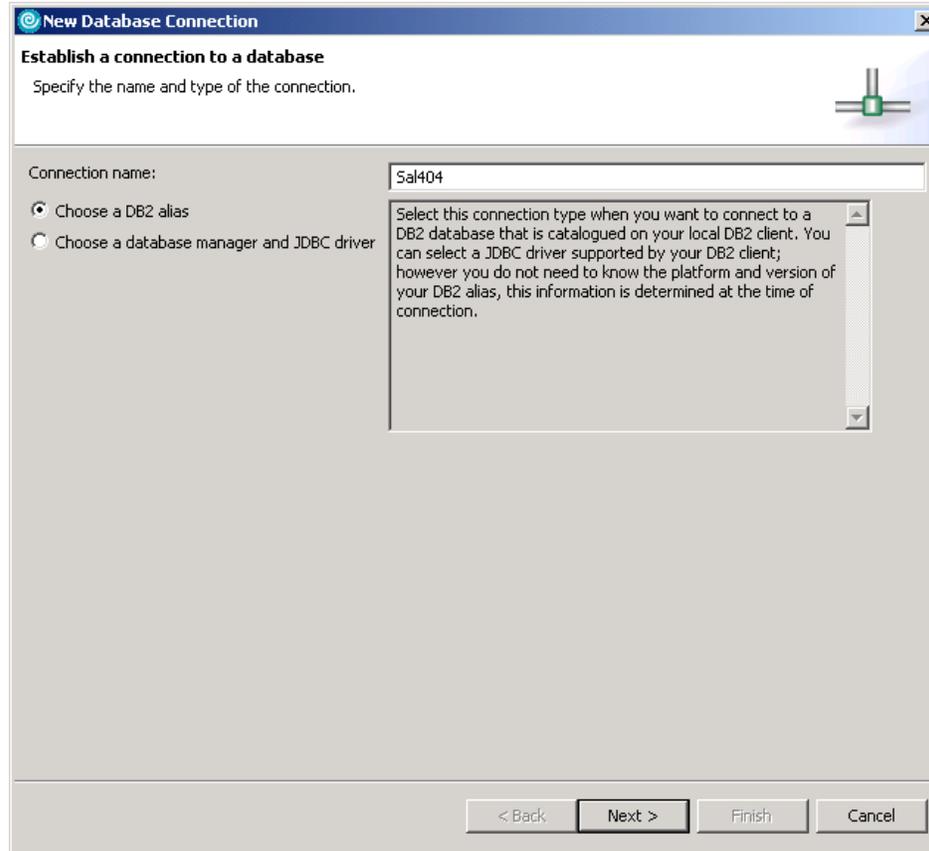
Figure 9-18 Provide connection details

3. Select the **SAL4004R** alias and enter user information. We want to enter
   db2admin as our user ID and provide the correct password. See Figure 9-19
   on page 332.
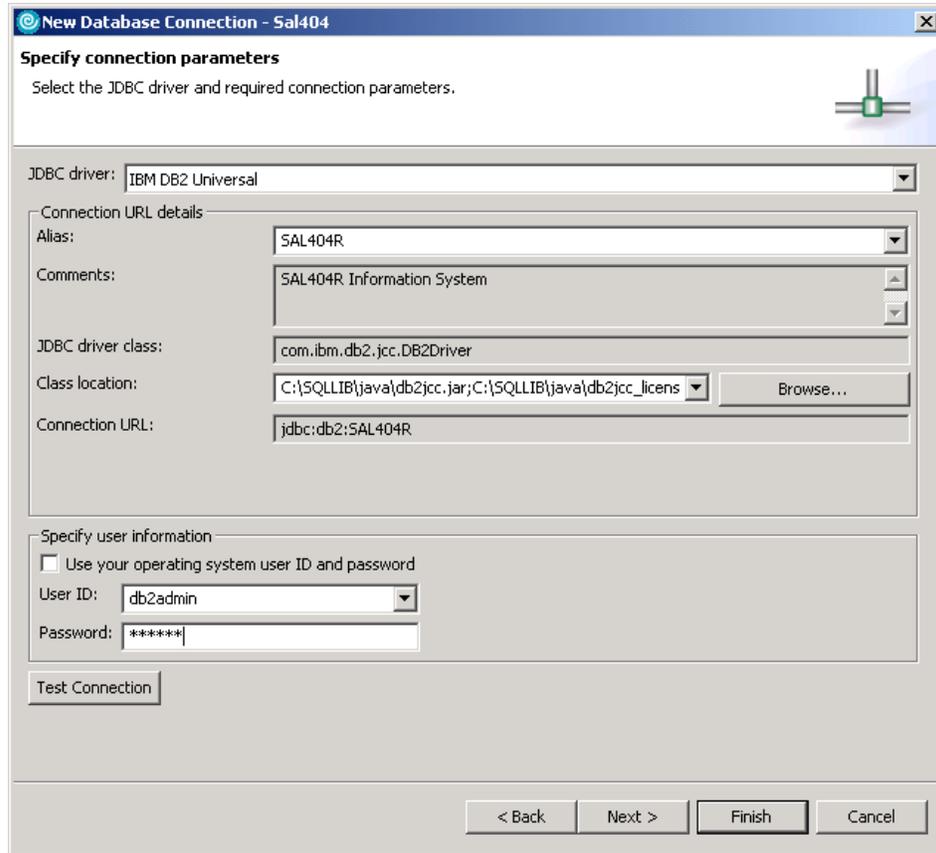
Figure 9-19   Choose alias and provide user details

4. Click **Test Connection**. A dialog box similar to that shown in Figure 9-20
    should be displayed.
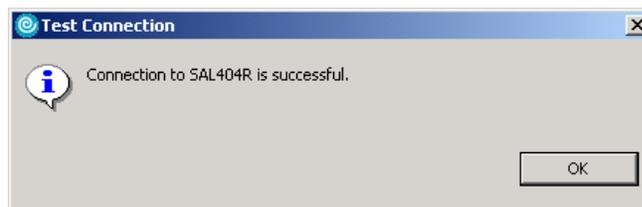
Figure 9-20   test the database connection

5. Click **OK** to close the test connection dialog box.

6. Click **Finish** to complete the database connection. You should be prompted on whether you want to copy database metadata to a project folder. See Figure 9-21 for an example.
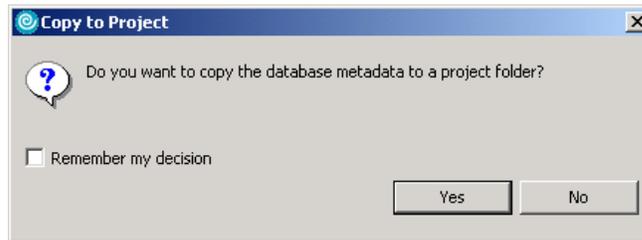
Figure 9-21   Copy database metadata to a project folder

7. Click **Yes** to copy the database metadata.

8. Click **Browse** and then select to copy the metadata to the **SAL404RealtyEJB** folder as shown in Figure 9-22.
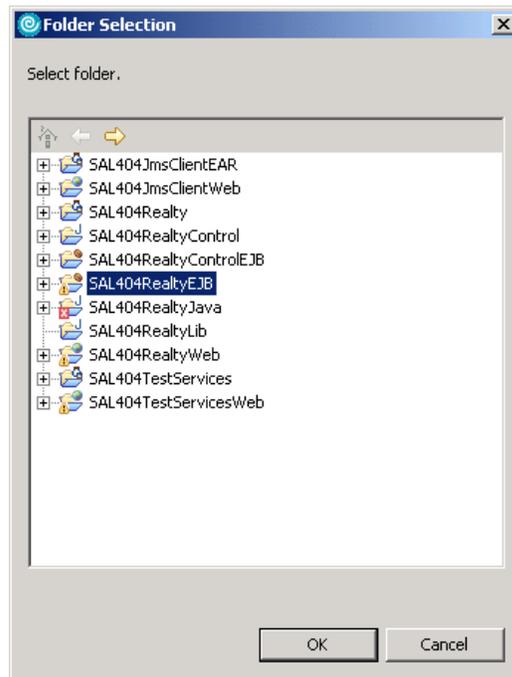
Figure 9-22   Select folder for metadata

9. Select **Use default schema folder for EJB projects** and click **Finish** as shown in Figure 9-23 on page 334.
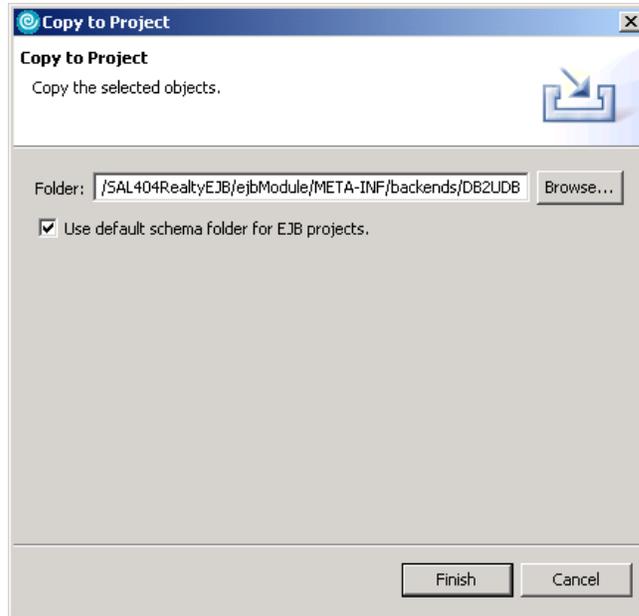
Figure 9-23   Finish connection creation

10.Click **Yes** if you are asked to confirm the creation of a new folder. See Figure 9-24.
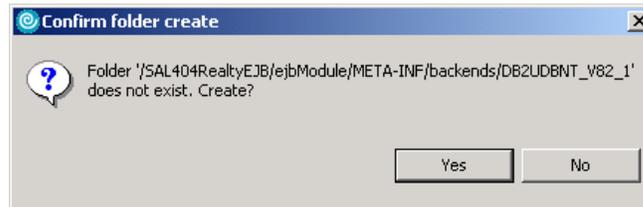
Figure 9-24   Confirm folder creation

11.Figure 9-25 on page 335 shows the new connection and the metadata that have been imported to our EJB project.
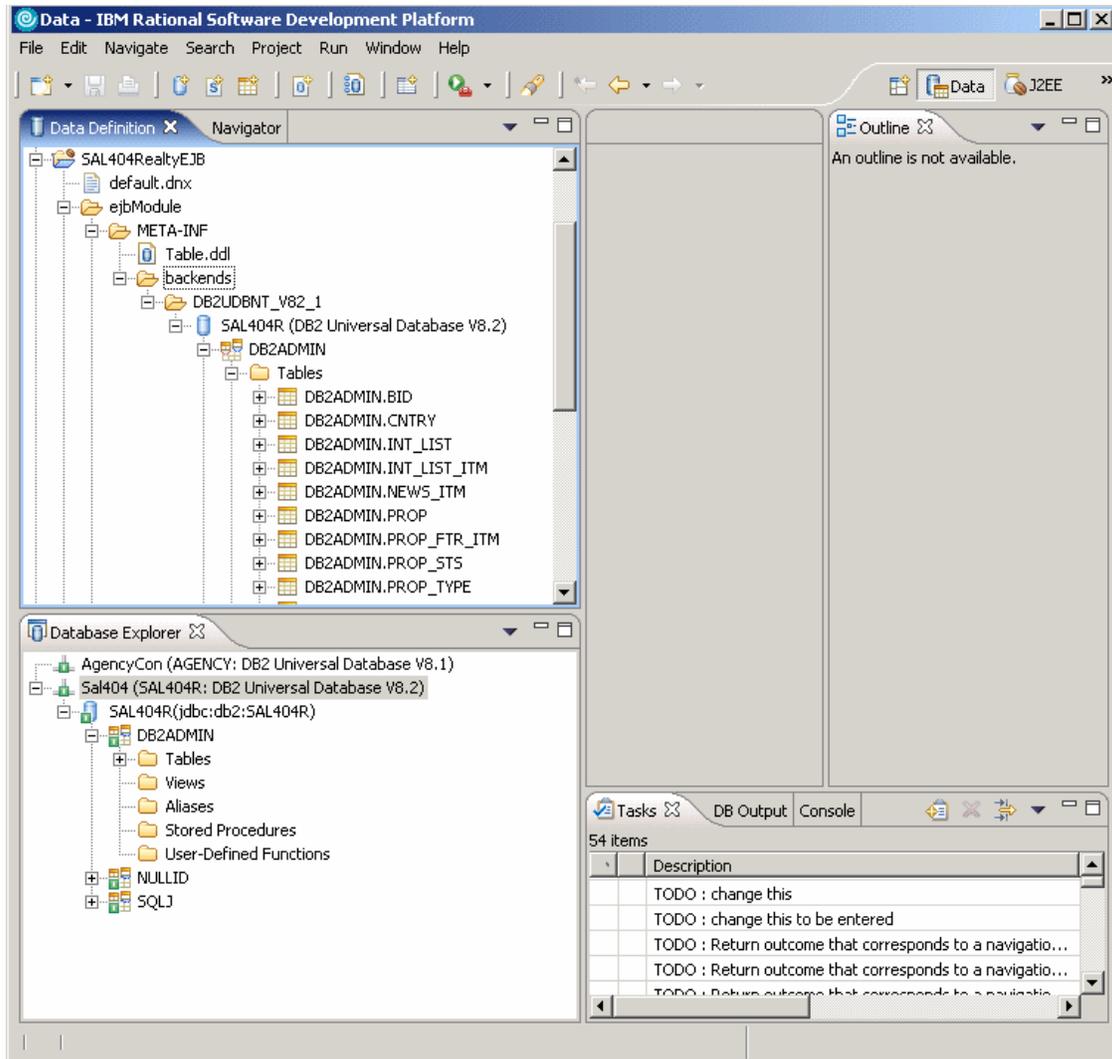
Figure 9-25   New database connection

9.2.6 Entity beans

Entity beans relate to a record in the underlying datastore. This is usually a database. The reference data implementation could have been done using entity beans. While this is true, asking the EJB container to create, for example, an entity bean for every row of the country codes would lead to performance issues and unnecessary resource consumption.

Instead, we will examine using an entity EJB to manage news items. Entity beans are located with finder methods. Our example will use finders that return a single entry.

The process we demonstrate is this section is known as *bottom up mapping*. This maps an existing database table into an entity EJB. The other two methods to create EJBs are known as top down, where a database is created for an existing EJB and meet in the middle.

1. To create the EJB from an existing table, a connection to the database must exist (and the meta data for the database will need to have been imported). Figure 9-26 shows that we have created a database connection for the SAL404R database using the method we described in 9.2.5, "Create a database connection" on page 329. For further details on how to create and manage database connections, see the *Rational Application Developer V6 Programming Guide*, SG24-6449. Right-click the **NEWS_ITM** table and choose **Create EJBs from Tables**.
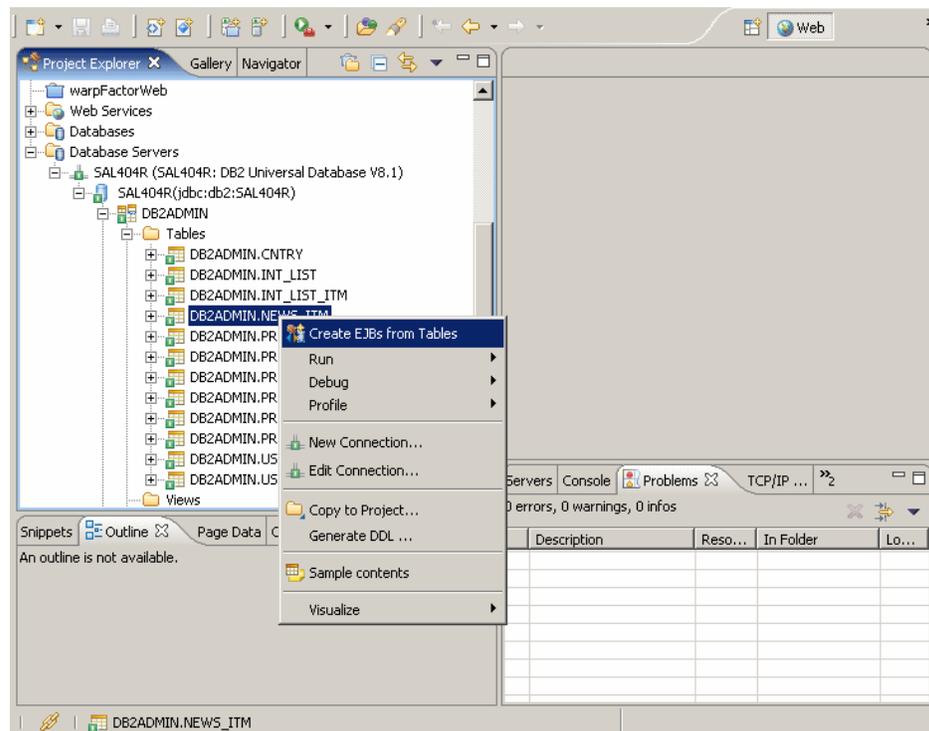
Figure 9-26   Select a table

2. Select the **SAL404RealtyEJB** project as shown in Figure 9-27 on page 337. Click **Next**.
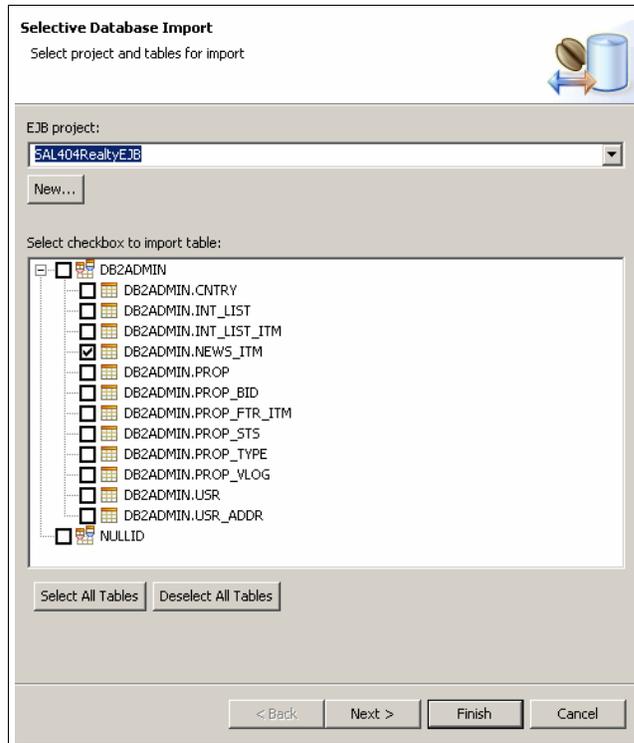
Figure 9-27   Select EJB project

3. Enter a package name for the generated EJB code and a prefix. We used `com.ibm.itso.sal404`. See Figure 9-28 on page 338. Then click **Finish**.

Figure 9-28   Provide meaningful package names

This action creates and entity bean for the news item table. The home interface created is quite simple. It has only two methods, create() and findByPrimaryKey().

Creating a news item requires that the EJB is provided with the primary key for the new item entry. Many business types have a natural and obvious primary key. An automobile has a unique vehicle registration number, for example. However it is also quite common for business types to have no obvious unique identifier. This is the case with the news items.

Several strategies exist for generating the primary key in this circumstance. A popular, quick solution is to have a database table that has an autoincrement column that can return a unique ID. This method has a couple of weaknesses. It has problems with portability because it works differently depending on the underlying database. It has scalability problems because every create will lock the table.

Another approach is to have a globally unique ID (GUID) generation system that can vend the IDs. Several GUID generation strategies are commonly discussed

on the Internet, and GUID generation will be formalized in upcoming J2EE specifications. Currently however, it is the responsibility of the developer to implement GUI generation.

The GUID generator used in our sample application is for demonstration purposes only and is not intended to be used in production systems. We recommend that you use a proven and reliable GUID generator for your production systems. Some Web references to visit when deciding on what GUID generator to use include:

► Spring globally unique identifier generator

   `http://static.springframework.org/spring-webflow/docs/pr3/api/org/springframework/util/RandomGuid.html`

► ActiveScript - J-GUID

   `http://www.activescript.co.uk/jguid.html`

Our sample implementation provides the KeyGenerator class. This class can be used as a simple Java class, or can be wrapped as a stateless session EJB. We created a stateless session bean for this class by following the same steps listed in 9.2.4, "Using stateless session EJBs" on page 316 and also use it as a simple Java class.

To test our news item entity EJB, we can use the Universal Test Client in the Rational Software Development Platform. The steps are:

1. From the Project Explorer of the J2EE Perspective, right-click **EJB Projects** → **SAL404RealtyEJB** → **Deployment Descriptor** → **Entity Beans** → **News_itm** and select **Run** → **Run on Server**.

2. When the Server Selection dialog box appears, select **WebSphere Application Server v6.0** and click **Finish**.

   The server will be started and the EJB project will be deployed, if necessary. .

3. The Universal Test Client Welcome page appears, as shown in Figure 9-29 on page 340.

> **Tip:** The default URL of the test client is `http://localhost:9080/UTC/`, so you can also access it through an external browser. If you want to access it from another machine, just substitute `localhost` with the hostname or IP address of the developer machine.
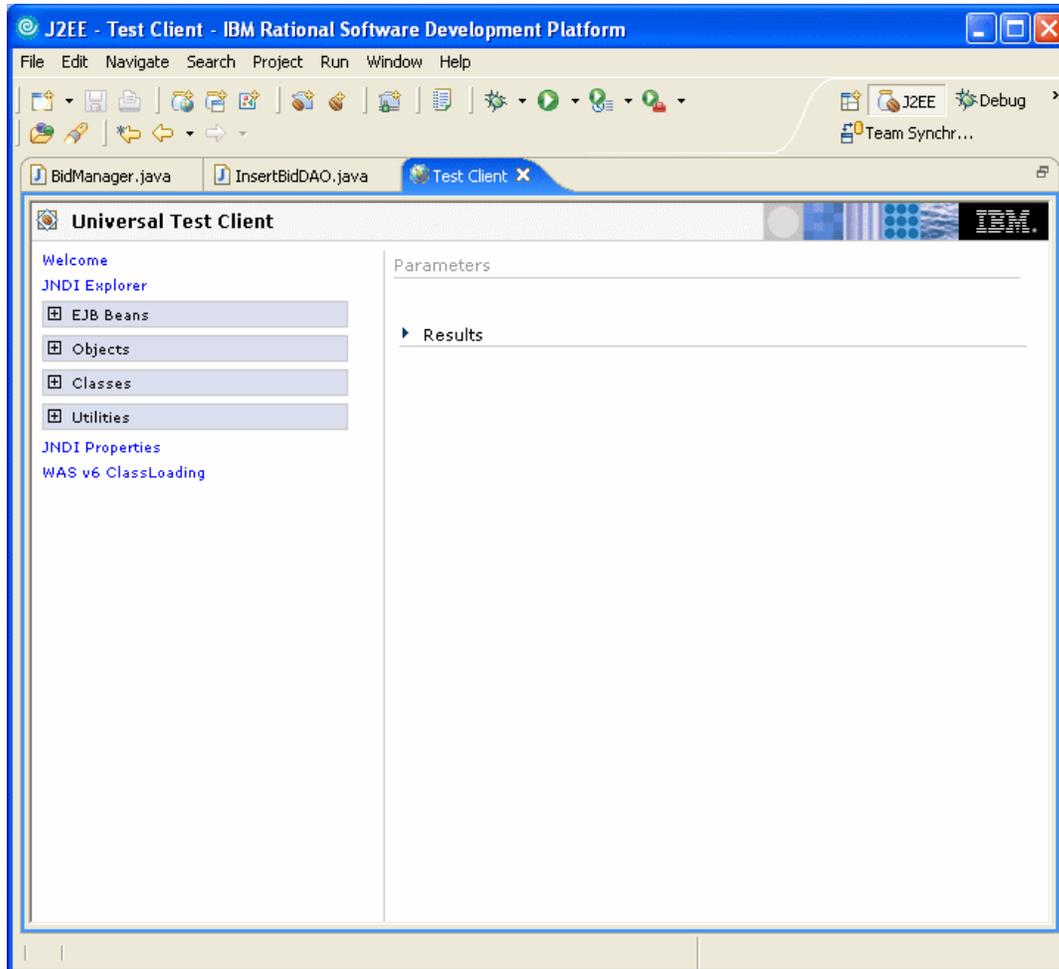
Figure 9-29   Testing an EJB with the Universal Test Client

4. Expand **EJB Beans** → **SAL404News_itmLocal** →
    SAL404News_itmLocalHome → **SAL404News_itmLocal create(String)**.
    See Figure 9-30 on page 341 for an example.
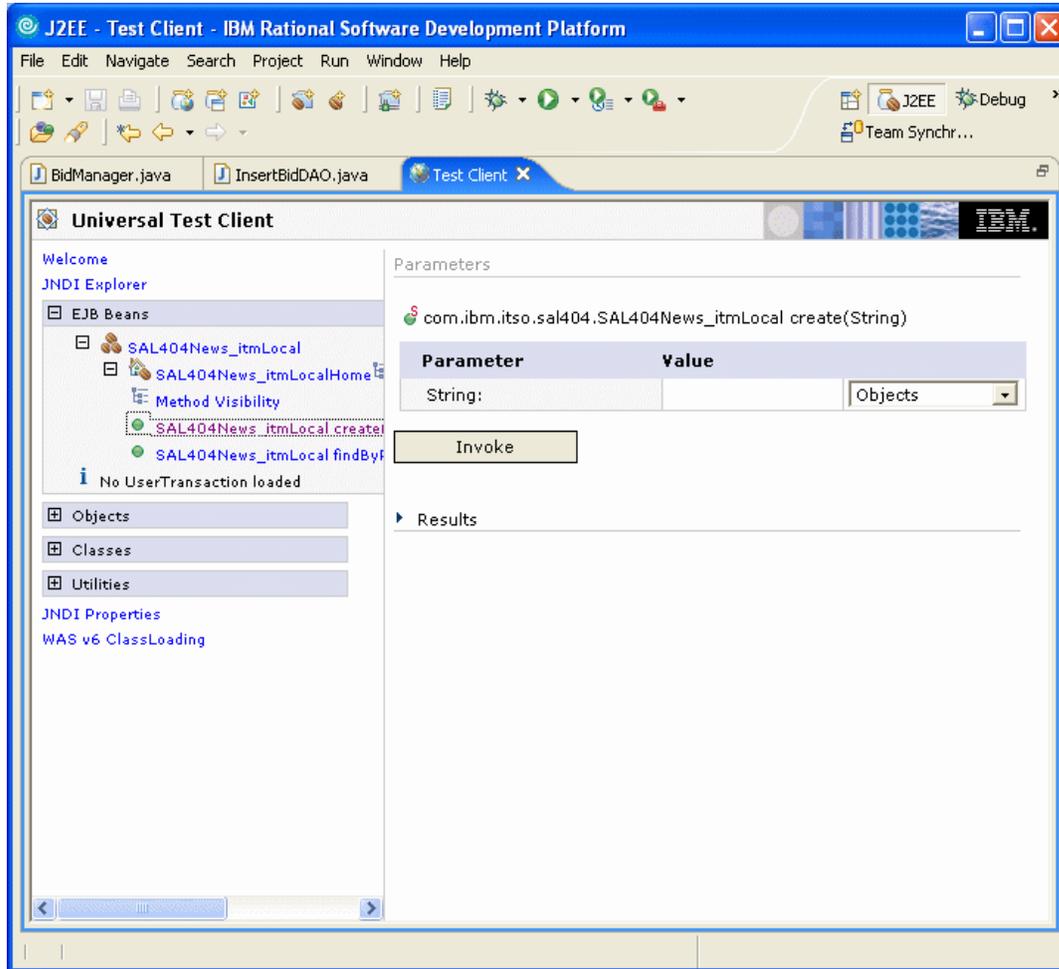
Figure 9-30   Test EJB create method

5. The create(String) method creates a new instance of the News_itm EJB with a unique key set to the entered String. Enter `AKeyString` and click **Invoke**. See Figure 9-31 on page 342 for an example.
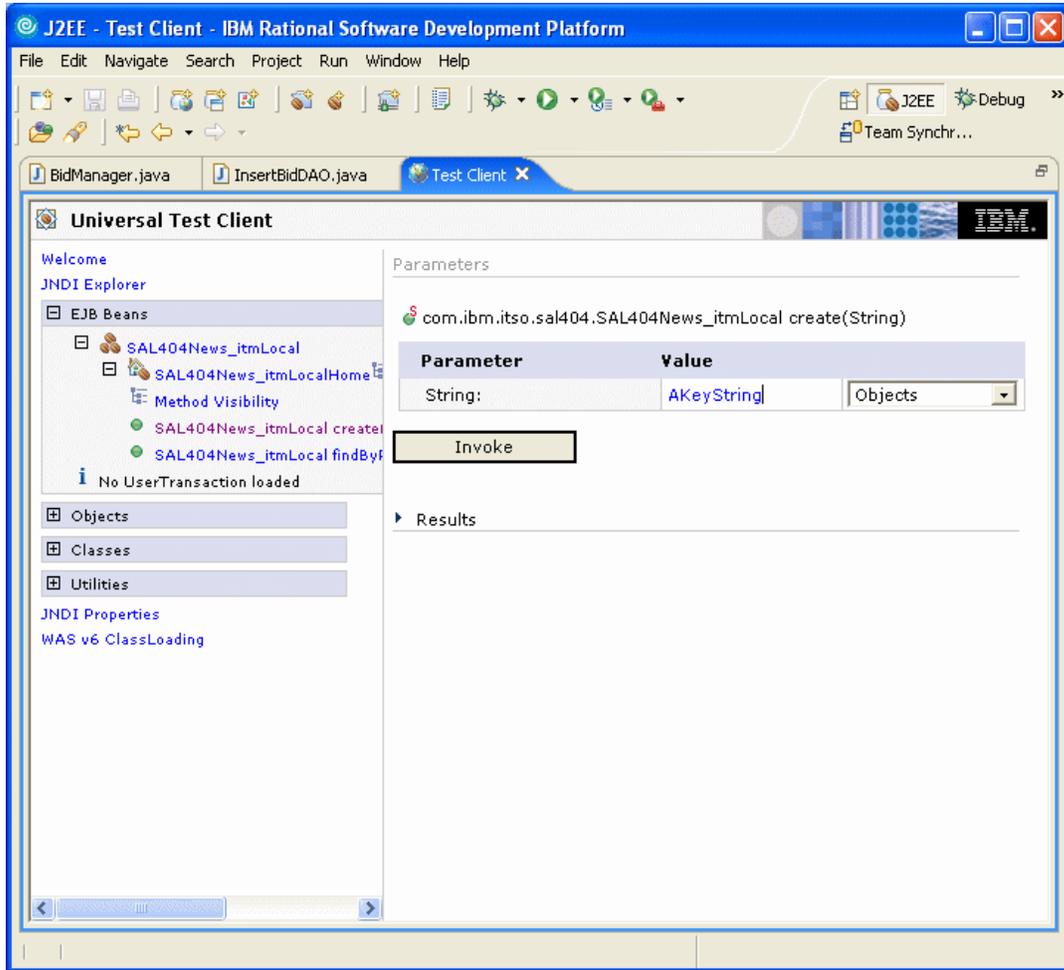
Figure 9-31   Invoke the create method

6. A new EJB instance is created so that you can work with the object as shown in Figure 9-32 on page 343.
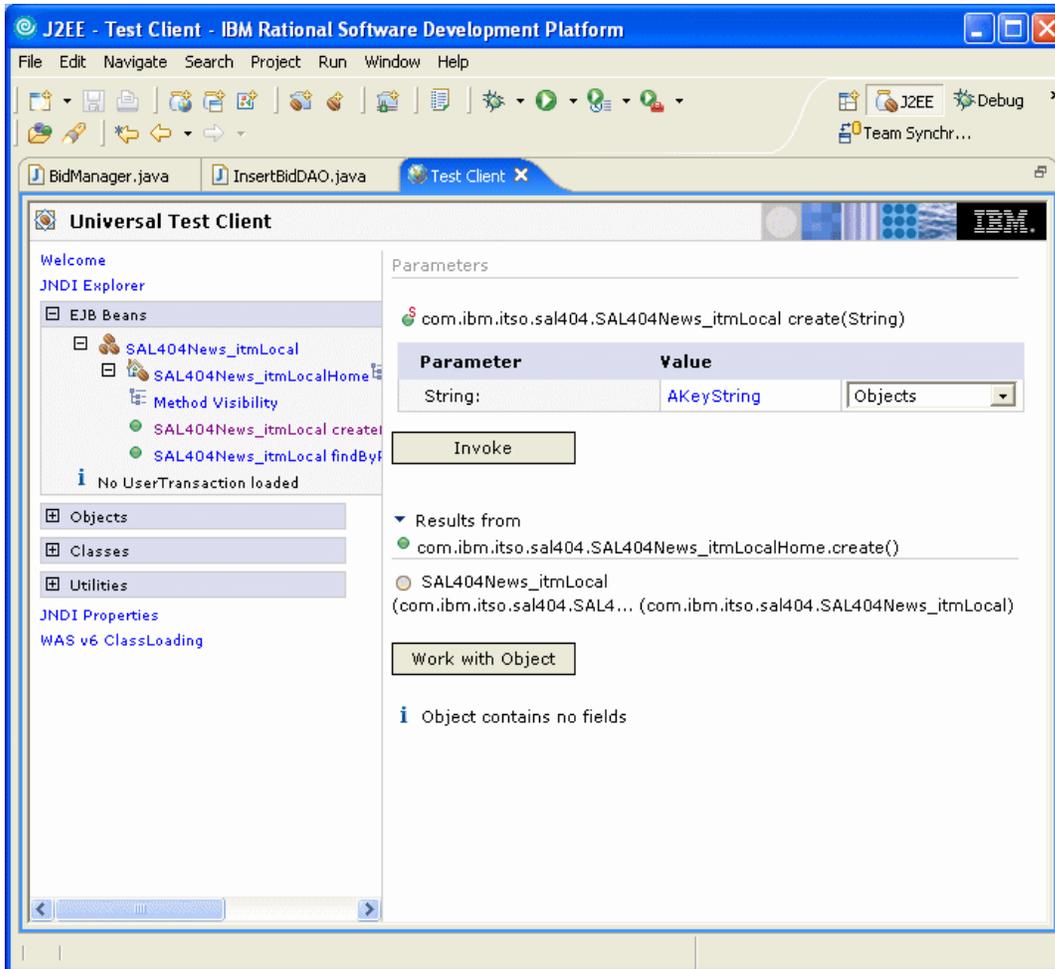
Figure 9-32   EJB created

7. Click **Work with Object** and the EJB instance is loaded under the EJB Beans heading. The Universal Test Client displays a list of available methods for the EJB, as shown in Figure 9-33 on page 344.
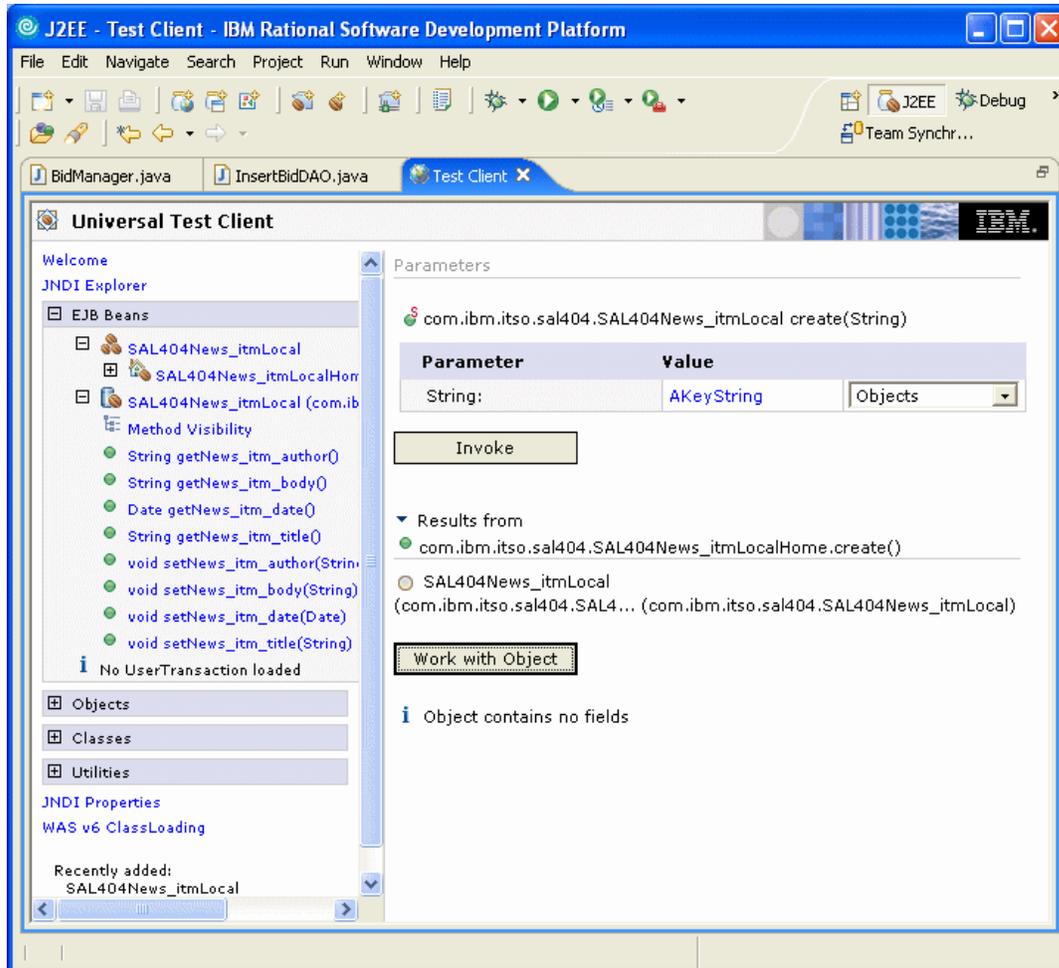
Figure 9-33   EJB method testing in the Universal Test Client

8. You can select a method, provide input parameters, and click **Invoke** to execute that method. Figure 9-34 on page 345 shows an example where we selected the **setNews_itm_date(Date)** method. In this case the Date parameter defaults to the current date so we do not need to change this.
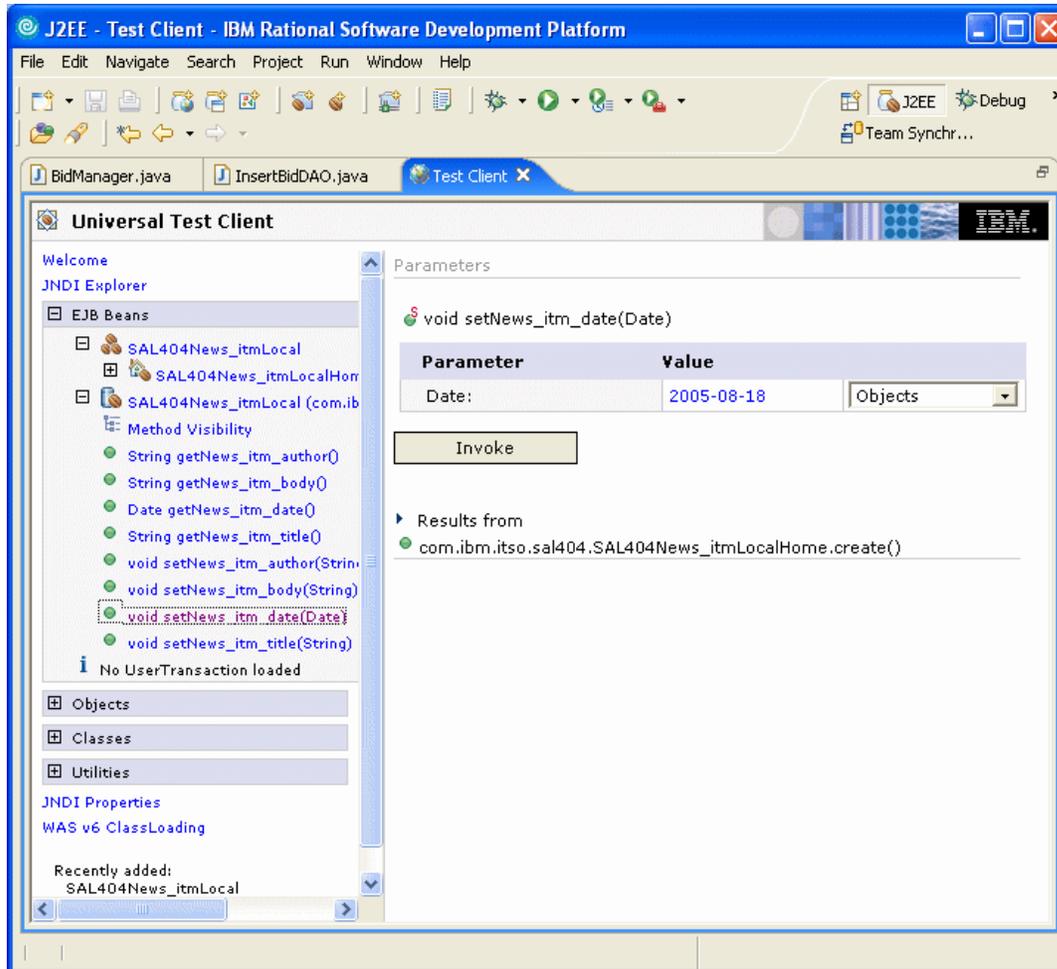
Figure 9-34   Test the setNews_itm_date(Date) method

9. Figure 9-35 on page 346 shows the results of a successful test of the setNews_itm_date(Date) method.
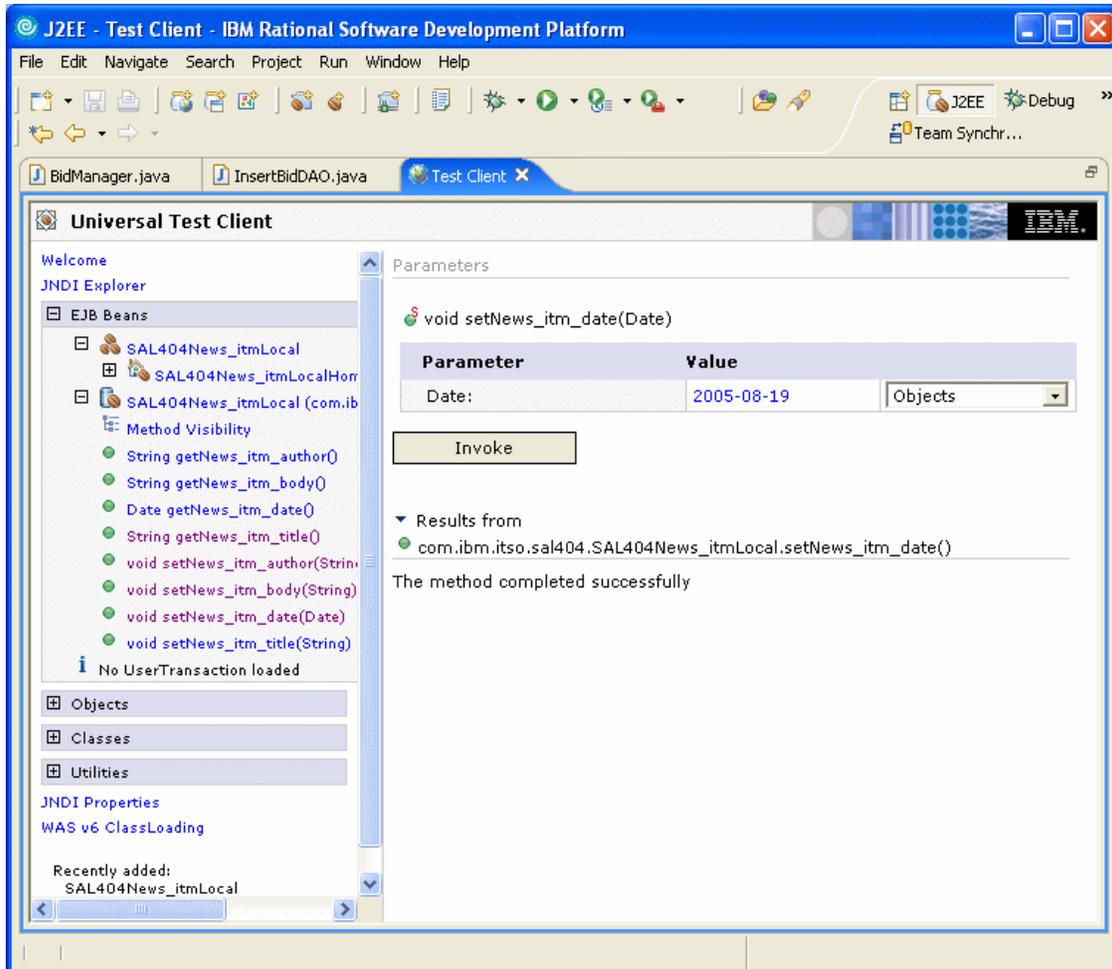
Figure 9-35   Results of testing the setNews_itm_date(Date) method

10. Figure 9-36 on page 347 shows the results of a successful test of the setNews_itm_author(String) method.
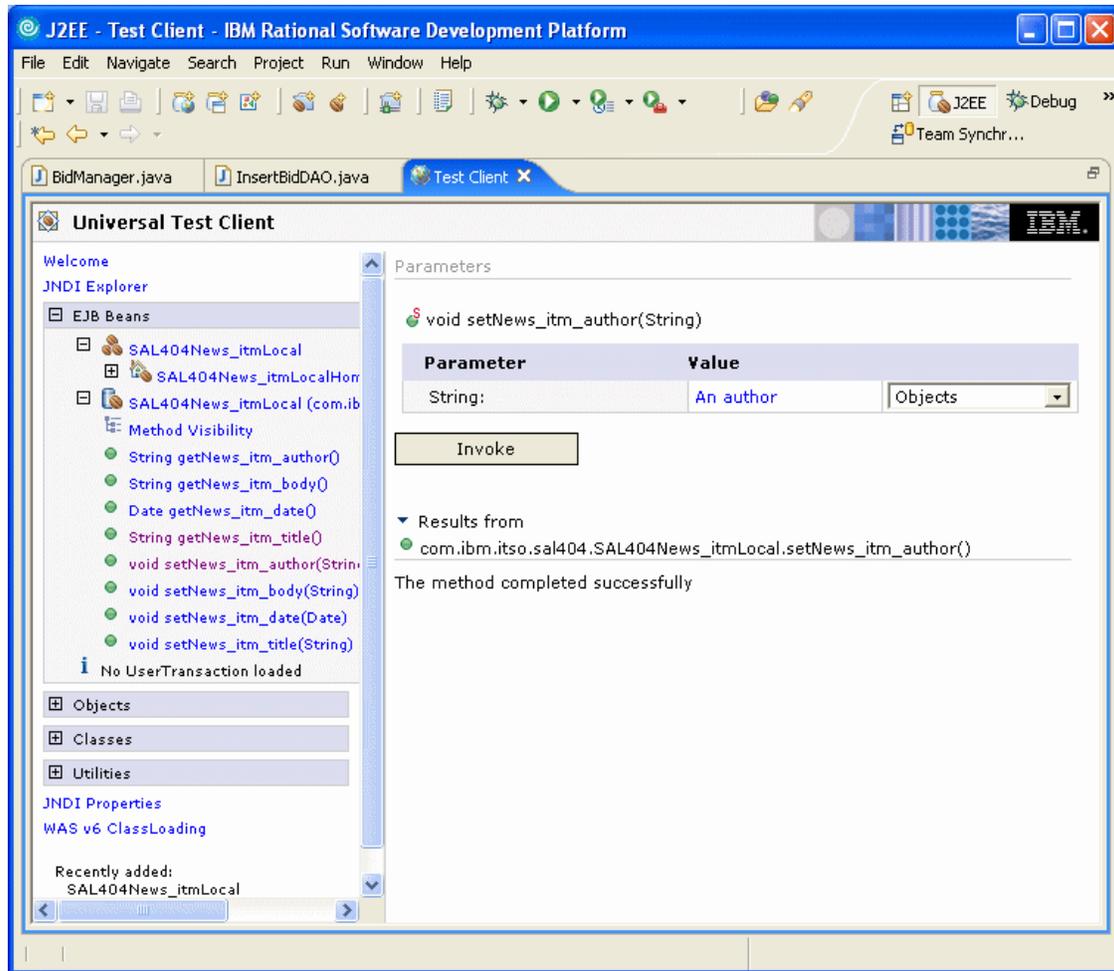
Figure 9-36   Results of testing the setNews_itm_author(String) method

11. Figure 9-37 on page 348 shows the results of a successful test of the setNews_itm_body(String) method.
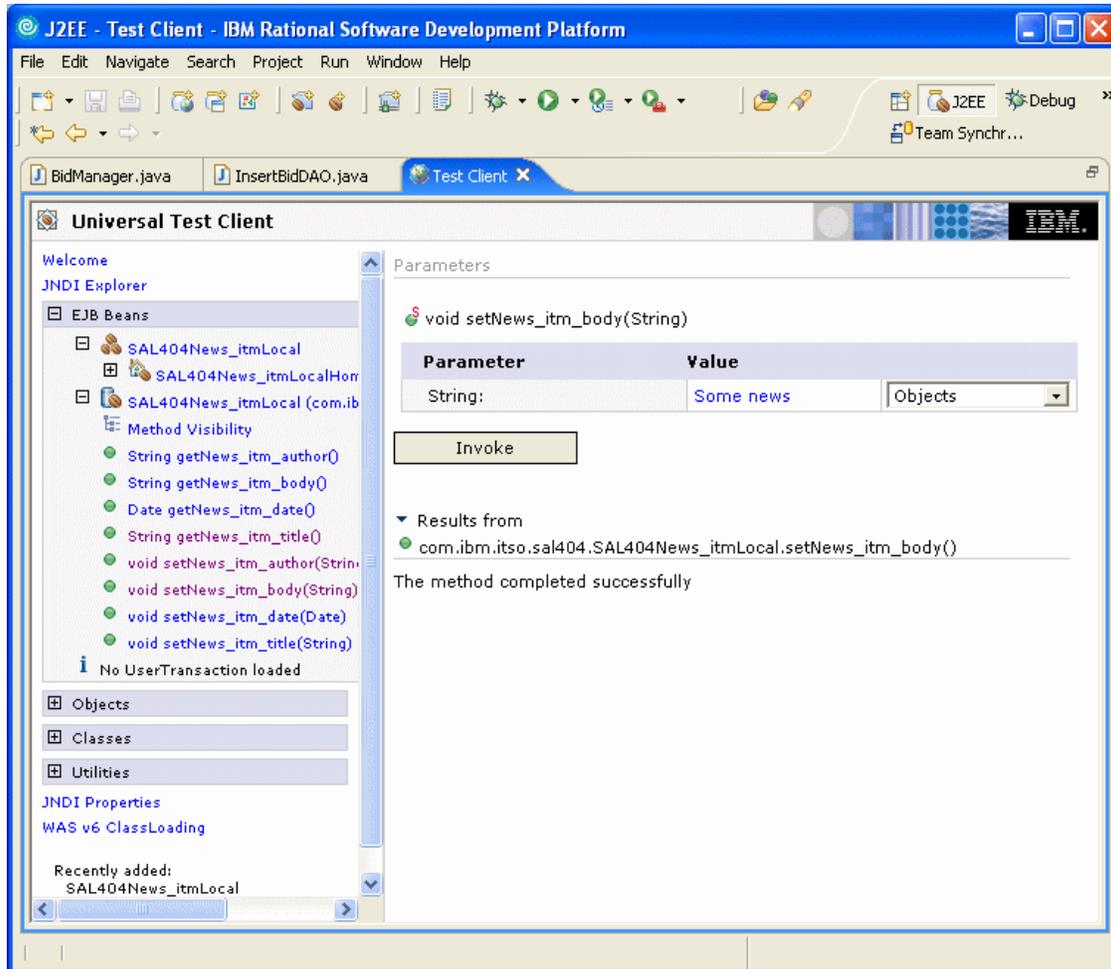
Figure 9-37   Results of testing the setNews_itm_body(String) method

12. Figure 9-38 on page 349 shows the results of a successful test of the setNews_itm_title(String) method.
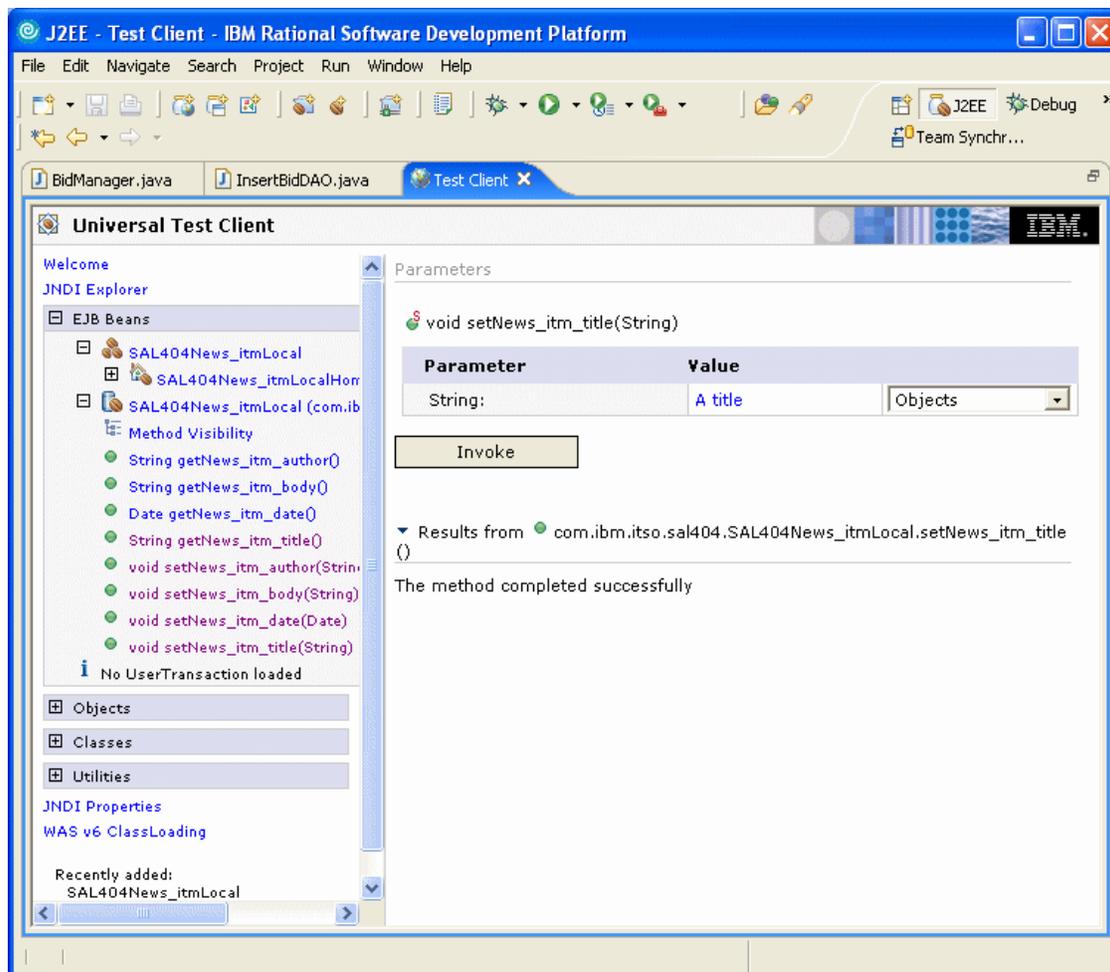
Figure 9-38   Results of testing the setNews_itm_title(String) method

Using the news item entity bean with Sal404

Now that we have successfully tested the News_itm entity EJB we can modify our Sal404 application so that data access to the news items is done using the EJB. We take a similar approach to that described in 8.5.2, "Implementing SDO-based data access" on page 298 in that we alter the manager to check if we should use EJB for our data access. This allows us to mix data access using EJB, SDO, and JDBC based on settings in our app.properties file.

Adding news with an entity EJB

To use the News_itm entity bean to add news items, follow these steps we used:

1. Modify the app.properties file in the SAL404RealtyJava project, setting a flag to determine whether the News database access will use JDBC, SDO or EJB. See Example 9-3 for the definition of the newsItemBackend flag.

Example 9-3   Modify app.properties to provide a newsItemBackend flag

```
# Flag to determine whether to use JDBC EJB or SDO database access for news
items
# Valid values are JDBC EJB and SDO
# newsItemBackend=JDBC
# newsItemBackend=SDO
newsItemBackend=EJB
```

2. Copy the existing NewsManager method addNewsItem(NewsItemDTO tempNewsItemDTO) and rename it `addNewsItemJDBC(NewsItemDTO tempNewsItemDTO)` in order to retain our JDBC access code.

3. Add a useEJB flag to the NewsManager and make sure that this is set correctly in the NewsManager constructor by looking up the app.properties file.

4. Altered the NewsManager method addNewsItem(NewsItemDTO tempNewsItemDTO) to check whether we should use EJB code to add a news item. Example 9-4 shows the code for this check.

Example 9-4   Check whether to use EJB to add news

```
if (useEJB)
            addNewsItemEJB(tempNewsItemDTO);
        else
            addNewsItemJDBC(tempNewsItemDTO);
```

> **Note:** In this code shown in Example 9-4 we do not check for SDO because we have not implemented an SDO-based add of news.

5. Create a new method called `addNewsItemEJB(tempNewsItemDTO)`.

6. Use code snippets to build EJB call code in the addNewsItemEJB(tempNewsItemDTO) method.

7. Open the addNewsItemEJB(tempNewsItemDTO) method in the Java editor.

8. Switch to the snippets view and expand the EJB snippets heading as shown in Figure 9-39 on page 351.
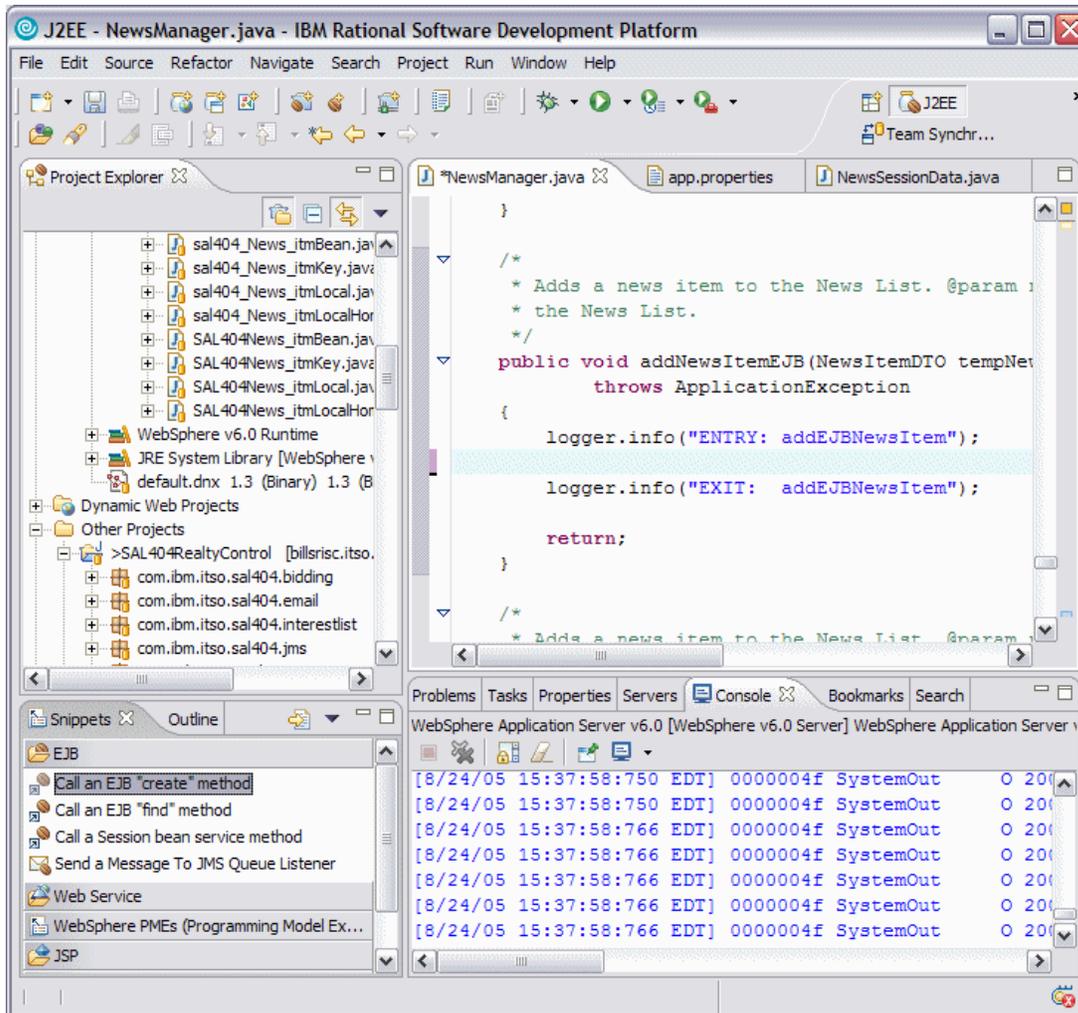
Figure 9-39   Code snippet for EJB create

9. Double-click **Call an EJB "create" method**.

10. Select the **News_itm** EJB as shown in Figure 9-40 on page 352 and click **Next**.
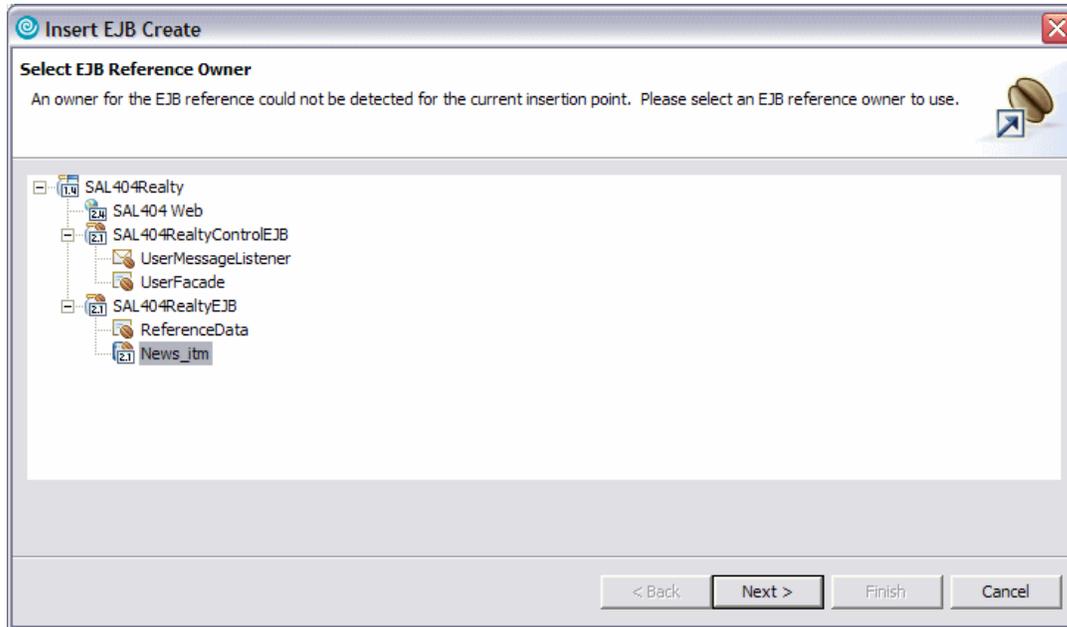
Figure 9-40   Choose a valid EJB for create

11.Select the **ejb/News_itm** reference as shown in Figure 9-41 on page 353 and click **Next**.
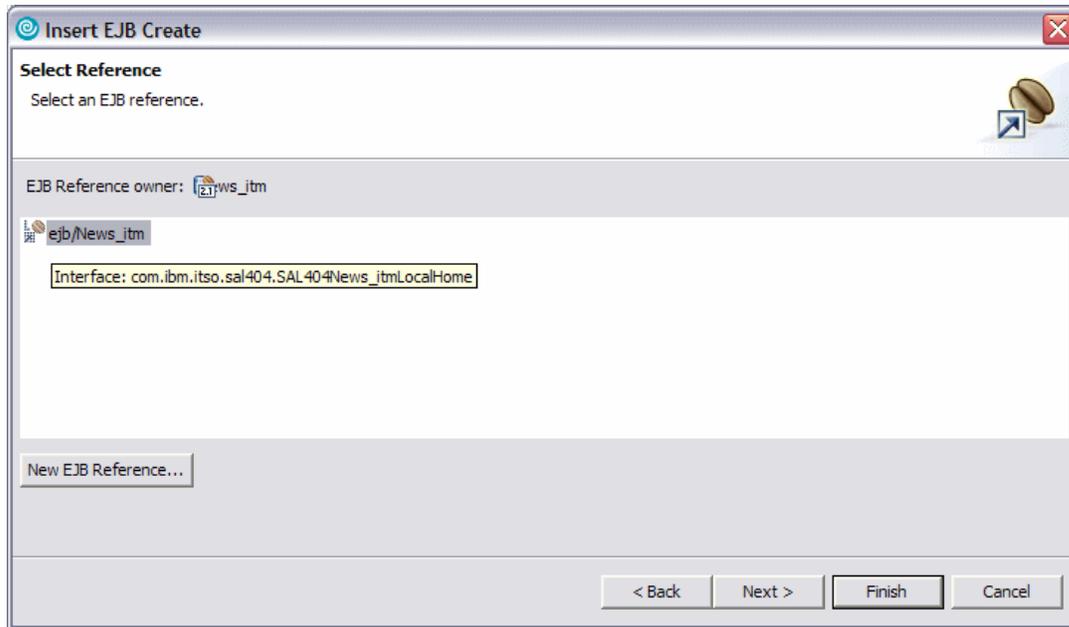
Figure 9-41   Select an EJB reference

12. Accept the parameter values shown in Figure 9-42 on page 354 and click **Finish**.
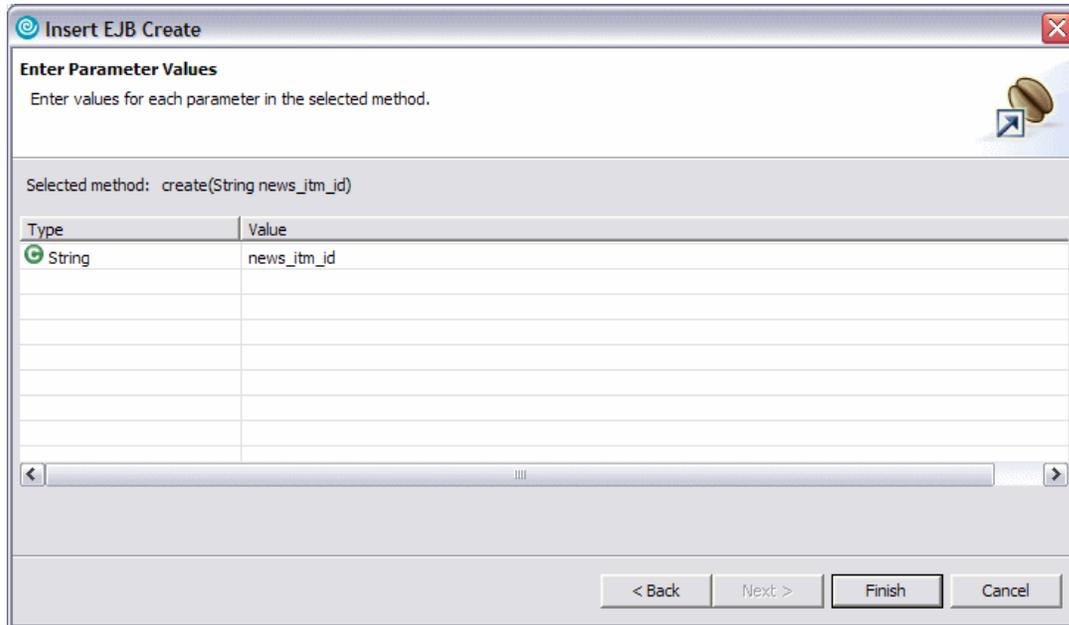
Figure 9-42   Parameter values for EJB create

13.Example 9-5 shows the generated EJB create code in the addNewsItemEJB method

Example 9-5   EJB create code

```
SAL404News_itmLocal aSAL404News_itmLocal =
createSAL404News_itmLocal(news_itm_id);
```

14.The snippet also creates the method called SAL404News_itmLocal createSAL404News_itmLocal(String news_itm_id).

15.Example 9-6 shows our completed addNewsItemEJB method where we pass values from the input DTO to the created EJB.

Example 9-6   addNewsItemEJB method

```
public void addNewsItemEJB(NewsItemDTO tempNewsItemDTO)
            throws ApplicationException
    {
        logger.info("ENTRY: addEJBNewsItem");
        KeyGenerator keyGenerator = new KeyGenerator();
```

```
        SAL404News_itmLocal ejbNewsItem =
createSAL404News_itmLocal(keyGenerator
             .getKeyString());
        ejbNewsItem.setNews_itm_author(tempNewsItemDTO.getAuthor());
        ejbNewsItem.setNews_itm_body(tempNewsItemDTO.getBody());
        java.sql.Date sqlDate = new java.sql.Date(tempNewsItemDTO.getDate()
             .getTime());
        ejbNewsItem.setNews_itm_date(sqlDate);
        ejbNewsItem.setNews_itm_title(tempNewsItemDTO.getTitle());
        logger.info("EXIT:  addEJBNewsItem");

        return;
    }
```

Modifying news with an entity EJB

To use the News_itm entity bean to modify news items, follow these steps we took:

1. Copy the existing NewsManager method modifyNewsItem(NewsItemDTO tempNewsItemDTO) and rename it `modifyNewsItemJDBC(NewsItemDTO tempNewsItemDTO)` in order to retain our JDBC access code.

2. Alter the NewsManager method modifyNewsItem(NewsItemDTO tempNewsItemDTO) to check whether we should use EJB code to modify a news item. Example 9-7 shows the code for this check.

Example 9-7   Check whether to use EJB to modify news

```
if (useSDO)
    modifyNewsItemSDO(tempNewsItemDTO);
else if (useEJB)
    modifyNewsItemEJB(tempNewsItemDTO);
else
    modifyNewsItemJDBC(tempNewsItemDTO);
```

> **Note:** In this code shown in Example 9-7, we check for both SDO and EJB otherwise we use JDBC.

3. Create a new method called `modifyNewsItemEJB(tempNewsItemDTO)`

4. Use code snippets to build EJB find code in the modifyNewsItemEJB(tempNewsItemDTO) method.

5. Open the modifyNewsItemEJB(tempNewsItemDTO) method in the Java editor.

6. Switch to the snippets view and expand the **EJB snippets** heading.

7. Double-click **Call an EJB "find" method**.

8. Select the **News_itm** EJB and click **Next**.

9. Select the **ejb/News_itm** reference and click **Next**.

10. Accept the parameter values shown in Figure 9-43 and click **Finish**.
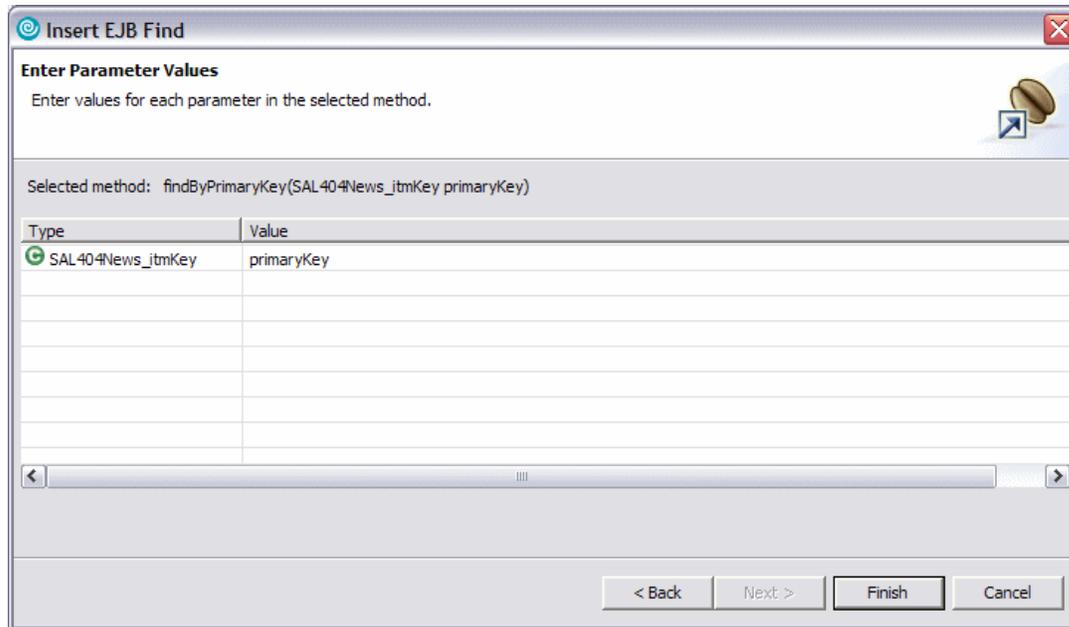
Figure 9-43   Parameter values for EJB find

11. Example 9-8 shows the generated EJB create code in the addNewsItemEJB method.

Example 9-8   EJB find code

```
SAL404News_itmLocal aSAL404News_itmLocal =
find_SAL404News_itmLocalHome_findByPrimaryKey(primaryKey);
```

12. The snippet also creates the method called SAL404News_itmLocal find_SAL404News_itmLocalHome_findByPrimaryKey(SAL404News_itmKey primaryKey).

13. Example 9-9 on page 357 shows our completed modifyNewsItemEJB method, where we pass values from the input DTO to the retrieved EJB.

Example 9-9   modifyNewsItemEJB method

```
private void modifyNewsItemEJB(NewsItemDTO tempNewsItemDTO)
    {
        logger.info("ENTRY: modifyNewsItemEJB");
        SAL404News_itmKey primaryKey = new SAL404News_itmKey(tempNewsItemDTO
                .getId());
        SAL404News_itmLocal aSAL404News_itmLocal =
find_SAL404News_itmLocalHome_findByPrimaryKey(primaryKey);
        aSAL404News_itmLocal.setNews_itm_author(tempNewsItemDTO.getAuthor());
        aSAL404News_itmLocal.setNews_itm_body(tempNewsItemDTO.getBody());
        java.sql.Date sqlDate = new java.sql.Date(tempNewsItemDTO.getDate()
                .getTime());
        aSAL404News_itmLocal.setNews_itm_date(sqlDate);
        aSAL404News_itmLocal.setNews_itm_title(tempNewsItemDTO.getTitle());
        logger.info("EXIT: modifyNewsItemEJB");
    }
```

Deleting news with an entity EJB

To use the News_itm entity bean to delete news items, follow the steps we took:

1. Copy the existing NewsManager method deleteNewsItem(String newsItemId) and rename it `deleteNewsItemJDBC(String newsItemId)` in order to retain our JDBC access code.

2. Alter the NewsManager method deleteNewsItem(String newsItemId) to check whether we should use EJB code to delete a news item.

3. Create a new method called `deleteNewsItemEJB(String newsItemId)`. Example 9-10 shows the code for our new method. This method reuses code and concepts from the EJB-based news modify.

Example 9-10   deleteNewsItemEJB(String newsItemId)

```
private void deleteNewsItemEJB(String newsItemId)
            throws ApplicationException
    {
        logger.info("ENTRY: deleteNewsItemEJB");
        SAL404News_itmKey primaryKey = new SAL404News_itmKey(newsItemId);
        SAL404News_itmLocal aSAL404News_itmLocal =
find_SAL404News_itmLocalHome_findByPrimaryKey(primaryKey);
        try
        {
            aSAL404News_itmLocal.remove();
        }
        catch (Exception e)
        {
            logger.info("removeError: " + e.getMessage());
```

```
                ApplicationException ae = new ApplicationException();
                ae.setStrutsMessage("error.news.delete");
                throw ae;
            }
            finally
            {
                logger.info("EXIT: deleteNewsItemEJB");
            }

        }
```

10

Java Message Service

In this chapter we discuss the Java Message Service and its use in WebSphere Application Server - Express. After explaining the basic concepts of messaging and JMS, we explain how we used JMS-based messaging in our sample application.

We show how to configure the default messaging provider in WebSphere Application Server - Express V6 using a simple example for sending and receiving messages.

For a detailed discussion about messaging, the service integration bus, and the other configuration options, WebSphere Application Server provides, refer to the redbook *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446.

10.1 Messaging concepts

The term *messaging* in the generic sense describes the exchange of information between two interested parties. In the context of computer science, *messaging* loosely describes a broad range of mechanisms used to communicate data. For instance, e-mail and Instant Messaging are two communication mechanisms that could be described as *messaging*. In both cases, information is exchanged between two parties, but the technology used to achieve the actual exchange is different.

10.1.1 Loose coupling

These two technologies can also be used to describe one of the main benefits of messaging, that is, *loose coupling*. We discuss two aspects of coupling in the context of messaging applications: process coupling and application coupling.

Process coupling

In the case of instant messaging, both parties involved in the exchange of messages need to be available at the same time the message is sent. Therefore, from a process point of view, the sending and receiving applications can be said to be *tightly coupled*.

In contrast, a user can send an e-mail to a recipient regardless of whether the recipient is currently online. In this case, the sender connects to an intermediary that is able to store the message until the recipient requests it. The sender and receiver processes in this situation can be described as *loosely coupled*. The intermediary in this situation is usually a mail server, but it can be generically referred to as a *messaging provider*.

Application coupling

As well as enabling loose coupling at the process level, messaging can also enable loose coupling at the application level. In this context, loose coupling means that the sending application is not dependent on any interface exposed by the receiving application. Both applications need only to be able to use the messaging provider's interface to enable them to connect and exchange data. With most messaging providers today, these interfaces are reasonably stable and, in some cases, based on open standards. These stable, open-standards-based interfaces mean that messaging applications can focus on the format of the data that is being exchanged, rather than the interface used to exchange the data. For this reason, messaging applications can be described as *datacentric*.

Contrast this with applications that make use of Enterprise JavaBeans(EJB). EJB client applications need to know about the interface exposed by the EJB. If this interface changes, then the EJB client application needs to be recompiled to prevent runtime errors. For this reason, EJBs and their clients can be described as tightly coupled. Also, due to the dependence on the interface exposed by the EJB, they can also be described as *interface-centric* applications.

10.1.2 Messaging types

The terms *tight* and *loose* coupling are not commonly used when describing messaging applications. It is more common to refer to the type of messaging that a given application uses. The messaging type describes the style of interaction between the sender and receiver.

The two messaging types are:

► Synchronous messaging

 Synchronous messaging involves tightly coupled processes, where the sending and receiving applications communicate directly and both must be available at the same time for the message exchange to occur.

► Asynchronous messaging

 Asynchronous messaging involves loosely coupled processes, where the sending and receiving applications communicate through a messaging provider. The sending application is able to pass the data to the messaging provider and then continue with its processing. The receiving application is able to connect to the messaging provider, possibly at some later point in time, to retrieve the data.

10.1.3 Destinations

With synchronous messaging, because there is no intermediary involved in the exchange of messages, the sending application must know how to connect to the receiving application. Once connected, there is no ambiguity with respect to the intended destination of a message because messages can only be exchanged between the connected parties. This is shown in Figure 10-1.
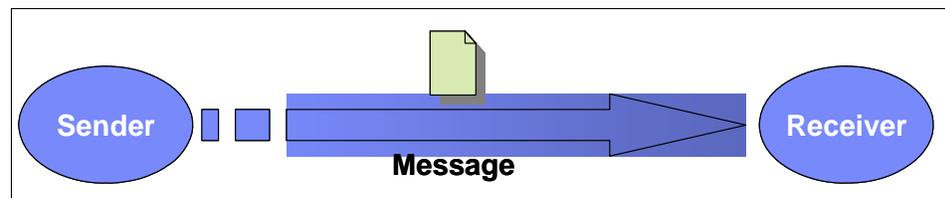
Figure 10-1   Direct communication using synchronous messaging

With asynchronous messaging, however, we need to introduce the concept of a destination. The need for a destination becomes apparent when we consider that a single messaging provider can act as an intermediary for many applications. In this situation, the sending and receiving applications must agree on a single destination that will be used to exchange messages. This destination must be specified when sending a message to or receiving a message from the messaging provider. This is shown in Figure 10-2.
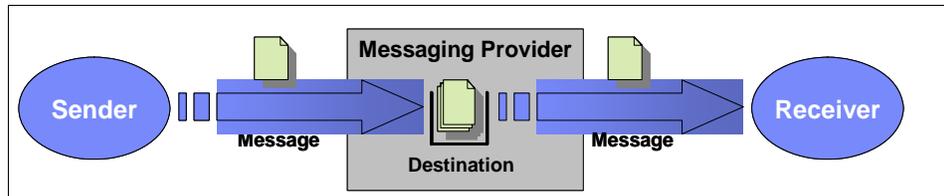
Figure 10-2   Indirect communication via a destination using asynchronous messaging

A sending application might need to exchange different messages with several receiving applications. In this situation, it would be normal for the sending application to use a different destination for each receiving application with which it wants to communicate. This is shown in Figure 10-3.
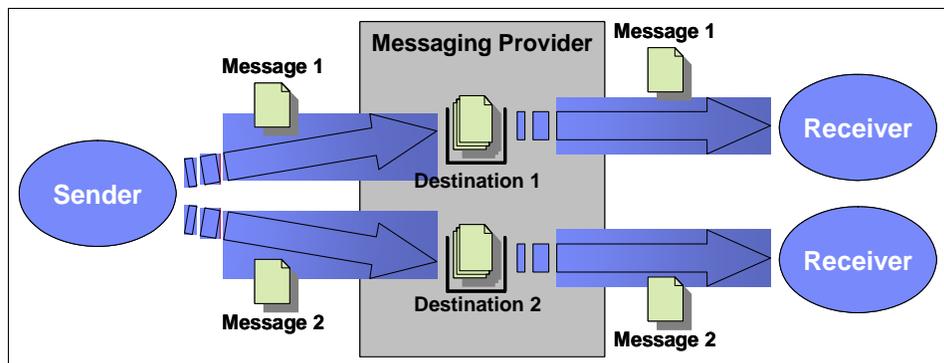
Figure 10-3   Communicating with multiple receivers using asynchronous messaging

10.1.4 Messaging models

As messaging technologies have evolved, two types of asynchronous messaging models have emerged, *Point-to-Point* and *Publish/Subscribe*. These models describe the cardinalities for the sender-receiver relationship, how the messaging provider distributes messages to the target destination. It is possible for an application to make use of both messaging models. The Point-to-Point and Publish/Subscribe messaging models are described in the following sections.

Point-to-Point

In the Point-to-Point messaging model, the sending application must specify the target destination for the message. To receive the message, the receiving application must specify the same destination when it communicates with the messaging provider. This means that there is a one-to-one mapping between the sender and receiver. This is the same situation as depicted in Figure 10-2 on page 362. In the Point-to-Point messaging model, the destination is usually referred to as a *queue*.

Publish/Subscribe

In the Publish/Subscribe messaging model, the sending application publishes messages to a destination. Multiple receiving applications subscribe to this destination in order to receive a copy of any messages that are published.

When a message arrives at a destination, the messaging provider distributes a copy of the message to all of the receiving applications who have subscribed to the destination. This means that there is potentially a one-to-many relationship between the sender and receiver. However, there might also be no receiving applications subscribed to a destination when a message arrives.

Note the difference from the situation depicted in Figure 10-3 on page 362. Figure 10-3 shows a sending application communicating with several receiving applications using the Point-to-Point messaging model. Figure 10-4 shows the Publish/Subscribe messaging model.
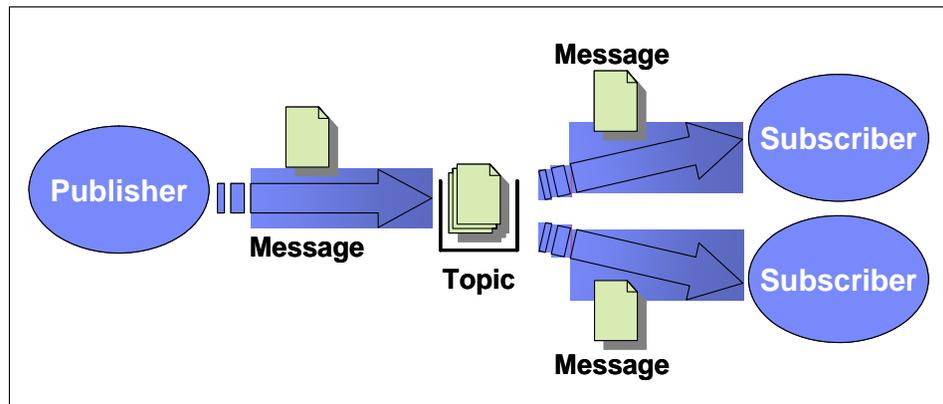
Figure 10-4   Publish/Subscribe messaging model

10.1.5 Messaging patterns

Several patterns describe the way in which messaging applications connect to, and use, messaging providers. These patterns describe whether a messaging

application interacts with the messaging provider as a *message producer*, *message consumer* or both. When a messaging application acts as both message producer and message consumer, the messaging pattern is referred to as *request-reply*. These messaging patterns are discussed in more detail in the following sections.

Message producers

In the message producer pattern, the sending application simply connects to the messaging provider, sends a message and then disconnects from the messaging provider. Because the sending application is not interested in what happens to the message once the messaging provider has accepted it, this pattern is sometimes referred to as *fire and forget*, although it is also commonly referred to as a *datagram*. The message producer pattern is shown in Figure 10-5.
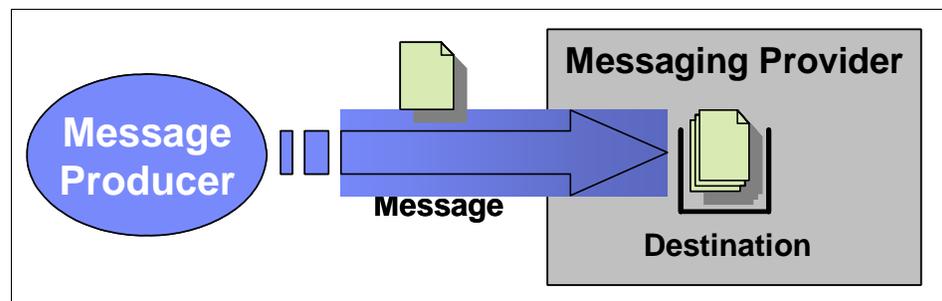
Figure 10-5   Message producer pattern

Message consumers

Message consumers operate in one of two modes:

► Pull mode

   In *pull* mode, the receiving application connects to the messaging provider and explicitly receives a message from the target destination. Obviously, there is no guarantee that a message will be available on the destination at a given point in time. The receiving application might need to retry at some later stage to retrieve a message. For this reason, the receiving application is said to *poll* the destination.

► Push mode

   In *push* mode, the messaging provider initiates the communication with the receiving application when a message arrives at a destination. The receiving application must register an interest in messages that arrive at the target destination with the messaging provider.

The message consumer pattern is shown in Figure 10-6 on page 365.
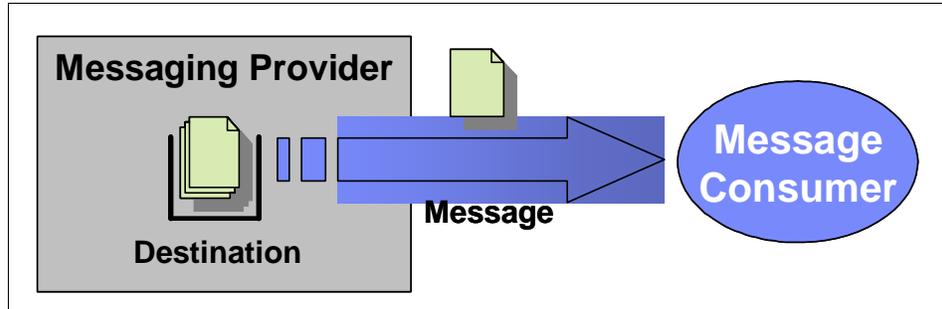
Figure 10-6   Message consumer pattern

Request-Reply

The request-reply pattern means the sending and receiving applications act as both message producers and message consumers. The sending application initiates the process by sending a message to a destination within the messaging provider and then waiting for a reply. The receiving application receives the message from the messaging provider, performs any required processing, and then sends the reply to the messaging provider. The sending application then receives this response from the messaging provider.

In this situation, the sending and receiving applications are tightly coupled processes, even though they are communicating using asynchronous messaging. For this reason, this pattern is often referred to as *pseudo-synchronous* messaging.

The request-reply pattern is shown in Figure 10-7.
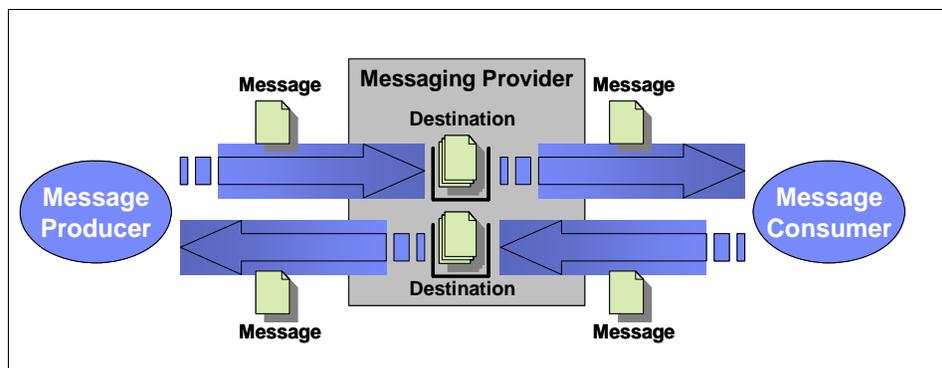
Figure 10-7   Request-reply pattern

10.2 Java Message Service API

The Java Message Service (JMS) API is the standard Java API for accessing enterprise messaging systems from Java programs. JMS is a standard API that sending and receiving applications written in Java can use to access the messaging provider create, send, receive, and read messages. We discuss some of the important features of the JMS specification in this section, such as:

► JMS API history
► JMS providers
► JMS domains
► JMS administered objects
► JMS and JNDI
► JMS connections
► JMS sessions
► JMS messages
► JMS message producers
► JMS consumers
► JMS exception handling
► Application Server Facilities
► JMS and J2EE

For a complete discussion of JMS, refer to the Java Message Service specification, Version 1.1. Visit this Web site to download a copy of the specification:

> http://java.sun.com/products/jms/docs.html

For further JMS information, see 10.9, "References and resources" on page 444.

10.2.1 JMS API history

IBM participated with Sun Microsystems™ and other organizations in the specification process that led to the original JMS API published in 1999. Several versions of the API have subsequently been released. The latest is version 1.1, which includes many changes that resulted from a review of the API by the Java community.

> **Note:** The JMS API defines a vendor-independent programming interface. It does not define how the messaging provider should be implemented, or which communication protocol clients should use to communicate with the messaging provider.

Different vendors produce different JMS implementations. While they should all be able to run the same JMS applications, implementations from different vendors will not necessarily be able to communicate directly with each other.

10.2.2 JMS providers

JMS providers are simply messaging providers that provide a JMS API implementation. However, this does not mean that the underlying messaging provider is written using the Java programming language. It simply means that the JMS provider written by a specific vendor is able to communicate with the corresponding messaging provider. As an example, the IBM WebSphere MQ JMS provider knows how to communicate with IBM WebSphere MQ.

10.2.3 JMS domains

The JMS API introduces the concept of *JMS domains*. These domains are exactly the same concepts as the messaging models described in 10.1.4, "Messaging models" on page 362, so no further discussion of them is necessary.

The JMS API also defines a set of domain-specific interfaces that enable client applications to send and receive messages in a given domain. However, version 1.1 of the JMS specification introduces a set of domain-independent interfaces, referred to as the *common interfaces*, in support of a unified messaging model. The domain-specific interfaces have been retained in version 1.1 of the JMS specification for backwards compatibility.

Today, the common interfaces are the preferred approach for implementing JMS client applications. For this reason, the JMS code examples in this chapter all make use of the common interfaces.

10.2.4 JMS administered objects

Administered objects encapsulate JMS provider-specific configuration information. They are created by an administrator and are later used at runtime by JMS clients.

The JMS specification states that the benefits of administered objects are:

► They hide provider-specific configuration details from JMS clients.

► They abstract JMS administrative information into Java objects that are easily organized and administered from a common management console.

The JMS specification defines two types of administered objects, JMS connection factories and JMS destinations. These are discussed in the following sections.

JMS connection factories

A *connection factory* encapsulates the configuration information that is required to connect to a specific JMS provider. A JMS client uses a connection factory to create a connection to that JMS provider. ConnectionFactory objects support concurrent use, meaning they can be accessed at the same time by multiple threads within a JMS client application.

The connection factory interfaces defined in the JMS specification are shown in Table 10-1.

Table 10-1   JMS Connection Factory Interfaces

Common Interface	Domain-specific Interfaces	
	Point-to-Point	**Publish/Subscribe**
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory

JMS destinations

A *destination* encapsulates address information for a specific JMS provider. A JMS client uses a destination object to address a message to a specific destination on the underlying JMS provider. Destination objects support concurrent use, meaning they can be accessed at the same time by multiple threads within a JMS client application.

The destination interfaces defined within the JMS specification are shown in Table 10-2.

Table 10-2   JMS Destination Interfaces

Common Interface	Domain-specific Interfaces	
	Point-to-Point	**Publish/Subscribe**
Destination	Queue	Topic

10.2.5 JMS and JNDI

At runtime, JMS clients need a mechanism by which to obtain references to the configured JMS administered objects. The JMS specification establishes the convention that these references are obtained by looking them up in a name space using the Java Naming and Directory Interface (JNDI) API.

The JMS specification does not define a naming policy that indicates where messaging resources should be placed in a name space. If the JMS client is a J2EE application, however, then the J2EE specification does recommend that messaging-related resources should be placed in a JMS subcontext.

Administrators require additional tools to create and bind the JMS administered objects into the JNDI name space. The JMS specification places the onus of providing these tools on the JMS provider. The tools that are provided for this purpose by WebSphere Application Server - Express form the WebSphere Administrative Console. Section 10.6, "Setup JMS the environment" on page 408 gives an example of using the WebSphere Administrative Console for the configuration of the administered object needed for the Sal404 application.

J2EE resource references and JMS

An additional consideration in this discussion is that the JMS client application needs to know where the JMS administered object was placed within the JNDI name space in order to locate it at run time. This requirement creates a dependency between the JMS client code and the actual runtime topology. If the JMS administered object is moved within the JNDI name space, the JMS client application would need to be modified. This is obviously unacceptable.

The J2EE specification provides a mechanism to add a level of indirection between the JMS client application and the JMS administered objects within the JNDI name space.

The specification defines a local JNDI name space for each J2EE component. This local JNDI name space can be accessed by performing lookups with names that begin with `java:comp/env`. When a J2EE module is assembled, the resources referenced through the local JNDI name space must be defined in the deployment descriptor for that module.

The administrator maps these references to the real JMS administered objects in the global JNDI name space when the application is deployed to the target operational environment.

At run time, when the JMS client performs a lookup in its local JNDI name space, it is redirected to the actual JMS administered object in the global name space.

Retrieving administered objects from JNDI

The code required to obtain references to a ConnectionFactory and Destination object is shown in Example 10-1.

Example 10-1   Using JNDI to retrieve JMS administered objects

```
import javax.jms.*;
import javax.naming.*

// Create the JNDI initial context
InitialContext initCtx = new InitialContext();

// Get the connection factory
```

```
ConnectionFactory connFactory
   = (ConnectionFactory)initCtx.lookup("java:comp/env/jms/myCF");

// Get the destination used to send a message
Destination destination = (Destination)lookup("java:comp/env/jms/myQueue");
```

10.2.6 JMS connections

A JMS Connection object represents the real connection that a JMS client has to
its JMS provider. The JMS specification states that a Connection encapsulates
an open connection with a JMS provider and that it typically represents an open
TCP/IP socket between a client and a JMS provider. However, this is dependent
on the JMS provider's implementation.

Creation of a Connection object normally results in resources being allocated
within the JMS provider, but outside of the process running the JMS client. For
this reason, care must be taken to close a Connection when it is no longer
required within the JMS client application. Invoking the close method on a
Connection object results in the close method being called on all of the objects
created from it.

The creation of the Connection object is also the point at which the JMS client
authenticates itself with the JMS provider. If no credentials are specified, then the
identity of the user under which the JMS client is running is used.

Connection objects support concurrent use.

ConnectionFactory objects are used to create instances of Connection objects.
The connection interfaces defined within the JMS specification are shown in
Table 10-3.

Table 10-3   JMS Connection Interfaces

Common Interface	Domain Specific Interfaces	
	Point-to-Point	**Publish/Subscribe**
Connection	QueueConnection	TopicConnection

The code required to create a Connection object is shown in Example 10-2.

Example 10-2   Creating JMS Connections

```
// User credentials
String userID = "jmsClient";
String password = "password";

// Create the connection, specifying no credentials
```

```
Connection conn1 = connFactory.createConnection();

// Create connection, specifying credentials
Connection conn2 = connFactory.createConnection(userID, password);
```

10.2.7 JMS sessions

A JMS Session is used to:

- ► Create message producers and message consumers for a single JMS provider. It is created from a Connection object.

- ► Define the scope of transactions. It can group multiple send and receive interactions with the JMS provider into a single unit of work. However, the unit of work will only span the interactions performed by message producers or consumers created from this Session object. A transacted session can complete a transaction using the commit or rollback methods of the Session object. Once the current transaction has been completed, a new transaction is automatically started.

Session objects do not support concurrent use. They cannot be accessed at the same time by multiple threads within a JMS client application. If a JMS client requires one thread to produce messages while another thread consumes them, the JMS specification recommends that the JMS client uses separate Sessions for each thread.

The session interfaces defined within the JMS specification are shown in Table 10-4.

Table 10-4   JMS Session Interfaces

Common Interface	Domain-specific Interfaces	
	Point-to-Point	**Publish/Subscribe**
Session	QueueSession	TopicSession

The code required to create a Session object is shown in Example 10-3.

Example 10-3   Creating JMS Sessions

```
// Create a non-transacted session
Session session = conn1.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

10.2.8 JMS messages

The JMS Session acts as a factory for JMS Messages. The JMS specification defines a logical format for the messages that can be sent to and received from JMS providers. Recall that the JMS specification only defines interfaces and not any implementation specifics, so the actual physical representation of a JMS message is provider specific.

The elements that make up a JMS message are:

► Headers

All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

► Properties

Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.

► Body

The JMS specification defines several types of message body.

The logical format of a JMS message is shown in Figure 10-8.
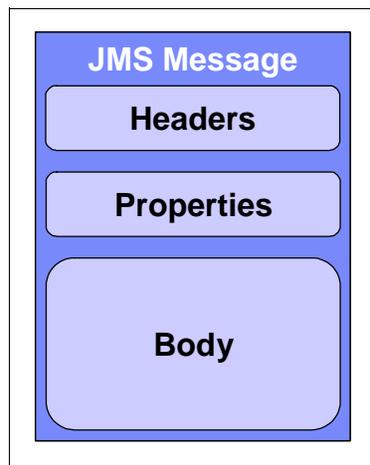
Figure 10-8   Logical format of a JMS Message

The JMS specification defines five Message interface children. These child interfaces allow for various types of data to be placed into the body of the message. The JMS message interfaces are described in Table 10-5 on page 373.

Table 10-5   JMS Message interface types

Message Type	Message Body
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
MapMessage	A set of name-value pairs, where names are strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined.
ObjectMessage	A message that contains a serializable Java object
StreamMessage	A stream of Java primitive values. It is filled and read sequentially.
TextMessage	A message containing a java.lang.String.

Types of message selectors

A JMS message selector allows a JMS client to filter the messages on a destination so that it only receives the messages in which it is interested. It must be a String whose syntax is based on a subset of the SQL92 conditional expression syntax. However, the message selector expression can only reference JMS message headers and properties, not values that might be contained in the message. An example of a message selector is shown in Example 10-4.

Example 10-4   Sample message selector

```
JMSType='car' AND color='blue' AND weight>2500
```

If a message consumer specifies a message selector when receiving a message from a destination, then only messages with headers and properties that match the selector are delivered. If the destination is a JMS queue, then the message remains on the queue. If the destination is a topic, then the message is never delivered to the subscriber. The reason is, from the subscribers perspective, the message does not exist.

For a full description of message selectors and their syntax, refer to the JMS specification, Version 1.1. Visit this Web site to download a copy of the specification:

http://java.sun.com/products/jms/docs.html

For broader JMS information, see 10.9, "References and resources" on page 444.

10.2.9 JMS message producers

The JMS Session also acts as a factory for JMS message producers. A JMS message producer is used to send messages to a specific destination on the JMS provider. A JMS message producer does not support concurrent use.

The target destination is specified when creating the message producer. However, it is possible to pass a value of null when creating the message producer. When using a message producer created in this manner, the target destination must be specified on every invocation of the send method.

The message producer can also be used to specify certain properties of messages that it sends such as, delivery mode, priority and time-to-live.

The message producer interfaces defined within the JMS specification are shown in Table 10-6.

Table 10-6   JMS MessageProducer Interfaces

Common Interface	Domain-specific Interfaces	
	Point-to-Point	**Publish/Subscribe**
MessageProducer	QueueSender	TopicPublisher

The code required to create and send a message is shown in Example 10-5.

Example 10-5   Creating and sending a JMS message

```
// Create the message producer
MessageProducer msgProducer = session.createProducer(destination);

// Create the message
TextMessage txtMsg = session.createTextMessage("Hello World");

// Send the message
msgProducer.send(txtMsg);
```

10.2.10 JMS message consumers

The JMS session also acts as factory for JMS message consumers. A JMS client uses a message consumer to receive messages from a JMS provider destination. A JMS message consumer does not support concurrent use.

The message consumer interfaces defined within the JMS specification are shown in Table 10-7.

Table 10-7   JMS MessageConsumer Interfaces

Common Interface	Domain-specific Interfaces	
	Point-to-Point	**Publish/Subscribe**
MessageConsumer	QueueReceiver	TopicSubscriber

Recall from the discussion in "Message consumers" on page 364, that message consumers can operate in pull mode or push mode. The JMS specification defines message consumers for both of these modes. The message consumers for these are modes are discussed in the following sections.

Pull mode

A JMS client operates in pull mode simply by invoking one of the receive methods on the MessageConsumer object. The MessageConsumer interface exposes a variety of receive methods that enable a client to poll the destination or wait for the next message to arrive.

The code required to receive a message using pull mode is shown in Example 10-6.

Example 10-6   Receiving a JMS message using pull mode

```
// Create the message consumer
MessageConsumer msgConsumer = session.createConsumer(destination);

// Start the connection
conn1.start();

// Attempt to receive a message
Message msg = msgConsumer.receiveNoWait();

// Make sure that we have a text message
if (msg instanceof TextMessage)
{
    // Cast the message to the correct type
    TextMessage txtMsg = (TextMessage)msg;

    // Print the contents of the message
    System.out.println(txtMsg.getText());
}
```

> **Note:** The start method must be invoked on the Connection object prior to attempting to receive a message. A connection does not need to be started in order to send messages, only to receive them. This enables the application to complete all of the required configuration steps before attempting to receive a message.

Push mode

To implement a solution that uses push mode, the JMS client must register an object that implements the javax.jms.MessageListener interface with the MessageConsumer. With a message listener instance registered, the JMS provider delivers messages as they arrive by invoking the listener's onMessage method.

> **Important:** A custom MessageListener can only be used in the client container. The J2EE specification forbids the use of the JMS MessageListener mechanism for the asynchronous receipt of messages in the EJB and Web containers.
>
> For the receiving of asynchronous messages, you have the concept of a Message Driven Bean (MDB) in the J2EE specification. The MDB enables the receiving application to listen to incoming messages asynchronously (see 10.4, "Message Driven Beans" on page 392).

The javax.jms.MessageListener interface is shown in Example 10-7 on page 376.

Example 10-7   The javax.jms.MessageListener interface

```
package javax.jms;

public interface MessageListener
{
    public void onMessage(Message message);
}
```

A simple class the implements the javax.jms.MessageListener interface is shown in Example 10-8.

Example 10-8   Simple MessageListener implementation

```
package com.ibm.itso.jms;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
```

```
import javax.jms.TextMessage;

public class SimpleListener implements MessageListener
{
    public void onMessage(Message msg)
    {
        // Make sure that we have a text message
        if (msg instanceof TextMessage)
        {
            // Cast the message to the correct type
            TextMessage txtMsg = (TextMessage)msg;

            try
            {
                // Print the contents of the message
                System.out.println(txtMsg.getText());
            }
            catch (JMSException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

An instance of the message listener can now be registered with the JMS
message consumer by the JMS client application. Once the listener is registered,
the connection needs to be started in order for messages to be delivered to the
message listener. The code required to register a message listener with a JMS
message consumer is shown in Example 10-9.

Example 10-9   Receiving a JMS message using push mode

```
import com.ibm.itso.jms.SimpleListener;

// Create the message consumer
MessageConsumer msgConsumer = session.createConsumer(destination);

// Create an instance of the message listener
SimpleListener listener = new SimpleListener();

// Register the message listener with the consumer
msgConsumer.setMessageListener(listener);

// Start the connection
conn1.start();
```

Note: In the JMS Point-to-Point domain, messages remain on a destination until they are either received by a message consumer or they expire. In the JMS Publish/Subscribe domain, messages remain on a destination until they have been delivered to all of the registered subscribers for the destination or they expire. In order for a message to be retained when a subscribing application is not available, the subscribing application must create a durable subscription.

10.2.11 JMS exception handling

Any runtime errors in a JMS application will result in a javax.jms.JMSException being thrown. The JMSException class is the root class of all JMS API exceptions.

A JMSException contains the following information:

► A provider-specific string describing the error

► A provider-specific string error code

► A reference to another exception

The JMSException is usually caused by another exception thrown in the underlying JMS provider. The JMSException class allows JMS client applications to access the initial exception using the getLinkedException method. The linked exception can then be used to determine the root cause of the problem in the JMS provider.

The implementation of JMSException does not include the embedded exception in the output of its toString method. Therefore, it is necessary to check explicitly for an embedded exception and print it out, as shown in Example 10-10.

Example 10-10   Handling a javax.jms.JMSException

```
try
{
    // Code which may throw a JMSException
}
catch (JMSException exception)
{
    System.err.println("Exception caught: " + exception);
    Exception linkedException = exception.getLinkedException();
    if (linkedException != null)
    {
        System.err.println("Linked exception: " + linkedException);
    }
}
```

However, when using a message listener to receive messages asynchronously, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to the receive methods on the message consumer.

The JMS API provides the javax.jms.ExceptionListener interface to solve this problem. An exception listener allows a client to be notified of a problem asynchronously. The JMS client must register an object that implements this interface with the connection using the setExceptionListener method. With an exception listener instance registered, the JMS provider invokes its onException method to notify it that a problem has occurred.

The javax.jms.ExceptionListener interface is shown in Example 10-11.

Example 10-11   The javax.jms.ExceptionListener interface

```
package javax.jms;

public interface ExceptionListener
{
    public void onException(JMSException exception);
}
```

A simple class implementing the javax.jms.ExceptionListener interface is shown in Example 10-12.

Example 10-12   Simple ExceptionListener implementation

```
package com.ibm.itso.jms;

import javax.jms.ExceptionListener;
import javax.jms.JMSException;

public class SimpleExceptionListener implements ExceptionListener
{
    public void onException(JMSException exception)
    {
        System.err.println("Exception caught: " + exception);
        Exception linkedException = exception.getLinkedException();
        if (linkedException != null)
        {
            System.err.println("Linked exception: " + linkedException);
        }
    }
}
```

10.2.12 Application Server facilities

The JMS specification defines a number of optional facilities that are intended to be implemented by JMS providers and Application Server vendors. These facilities extend the functionality of JMS when the JMS client executes within the context of a J2EE container. The Application Server Facilities are concerned with two main areas of functionality, concurrent message processing and distributed transactions. These functions are briefly described in the following sections.

Concurrent message consumers

Recall that Session and MessageConsumer objects do not support concurrent access from multiple threads. Such a restriction would be a huge obstacle to implementing JMS applications within an application server environment, where performance and resource usage are key concerns. The Application Server Facilities define a mechanism that enable an application server to create MessageConsumers that can process concurrently multiple incoming messages.

Distributed transactions

The JMS specification requires that a JMS provider support distributed transactions. However, it also states that if a provider supplies this support, it should be done with the JTA XAResource API. The Application Server Facilities define the interfaces that an application server should implement to correctly provide support for distributed transactions.

10.2.13 JMS and J2EE

The JMS API was included first in version 1.2 of the J2EE specification. This specification required that the JMS API definitions be included in a J2EE product, but that the platform was not required to include an implementation of the JMS ConnectionFactory and Destination objects.

Subsequent versions of the J2EE specification have placed further requirements on application server vendors. WebSphere Application Server V6.0 is fully compliant with version 1.4 of the J2EE specification, which states the following with regard to the JMS API:

Subsequent versions of the J2EE specification have placed further requirements on application server vendors. WebSphere Application Server V6.0 is fully compliant with version 1.4 of the J2EE specification, which requires that a Java Message Service provider must be included in a J2EE product. The JMS provider must support both JMS point-to-point and publish/ subscribe messaging, which means that it should implement the ConnectionFactory and Destination APIs. The JMS specification also list several optional interfaces that

are intended to help integration with an application server, but a J2EE product
need not implement these interfaces:

- ▶ javax.jms.ServerSession
- ▶ javax.jms.ServerSessionPool
- ▶ javax.jms.ConnectionConsumer
- ▶ all javax.jms XA interfaces

For further information about the full J2EE v1.4 specification, follow this link:

http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

WebSphere Application Server V6.0 also provides full support for the Application
Server Facilities described in 10.2.12, "Application Server facilities" on page 380.

10.3 Messaging in the J2EE Connector Architecture

Prior to J2EE version 1.3, there was no architecture that specified the interface
between an application server and the resource adapter for an Enterprise
Information Systems (EIS). Consequently, application server and EIS vendors
used vendor-specific architectures to provide EIS integration. This meant that, for
each application server that an EIS vendor supported, it needed to provide a
specific resource adapter. For every resource adapter that an application server
vendor supported, it needed to extend the application server.

J2EE version 1.3 required application servers to support version 1.0 of the J2EE
Connector Architecture (JCA). The J2EE Connector Architecture defines a
standard for connecting a compliant application server to an EIS. It defines a
standard set of system-level contracts between the J2EE application server and
a resource adapter.

As a result, application servers only need to be extended once to add support for
all J2EE Connector Architecture compliant resource adapters. Conversely, EIS
vendors only need to implement one J2EE Connector Architecture compliant
resource adapter, which can then be installed on any compliant application
server.

The system contracts defined by the J2EE Connector Architecture version 1.0
are described by the specification as follows:

- ▶ Connection management

  Enables an application server to pool connections to the underlying EIS and
  enables application components to connect to an EIS. This leads to a
  scalable application environment that can support a large number of clients
  requiring access to an EIS.

- ► Transaction management

  Enables an application server to use a transaction manager to manage transactions across multiple resource managers. This contract also supports transactions that are managed internal to an EIS resource manager without the necessity of involving an external transaction manager.

- ► Security management

  Provides support for a secure application environment that reduces security threats to the EIS and protects valuable information resources managed by the EIS.

> **Note:** For a full description of the system contracts listed above, refer to the J2EE Connector Architecture Version 1.0 specification. A link for this specification is included in 10.9, "References and resources" on page 444.

While version 1.0 of the J2EE Connector Architecture addressed the main requirements of both application server and EIS vendors, it left some issues unresolved. As a result, version 1.5 of the specification was produced. It is this version that application servers are now required to support by version 1.4 of the J2EE specification.

The additional system contracts defined by version 1.5 of the J2EE Connector Architecture are described by the specification as follows:

- ► Life cycle management

  Enables an application server to manage the life cycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during its deployment or application server startup, and to notify the resource adapter instance during its undeployment or during an orderly shutdown of the application server.

- ► Work management

  Enables a resource adapter to do work (monitor network endpoints, call application components, etc.) by submitting Work instances to an application server for execution. The application server dispatches threads to execute submitted Work instances. This allows a resource adapter to avoid creating or managing threads directly, and allows an application server to efficiently pool threads and have more control over its runtime environment. The resource adapter can control the security context and transaction context with which Work instances are executed.

- ► Transaction inflow management

  Enables a resource adapter to propagate an imported transaction to an application server. This contract also allows a resource adapter to transmit

transaction completion and crash recovery calls initiated by an EIS, and ensures that the ACID (Atomicity, Consistency, Isolation and Durability) properties of the imported transaction are preserved.

► Message inflow management

Enables a resource adapter to asynchronously deliver messages to message endpoints residing in the application server independent of the specific messaging style, messaging semantics, and messaging infrastructure used to deliver messages. This contract also serves as the standard message provider pluggability contract that allows a wide range of message providers (Java Message Service (JMS), Java API for XML Messaging (JAXM), etc.) to be plugged into any J2EE compatible application server via a resource adapter.

> **Note:** For a full description of the system contracts listed above, please refer to the J2EE Connector Architecture Version 1.5 specification. A link for this specification is included in 10.9, "References and resources" on page 444.

In the context of asynchronous messaging, we are interested in the message inflow system contract. The sections that follow discuss the following aspects of the message inflow system contract:

► Message endpoints
► Resource adapters
► JMS ActivationSpec JavaBean
► Administered objects

10.3.1 Message endpoints

The message inflow system contract uses the Message Driven Bean (MDB) programming model to deliver messages asynchronously from an EIS into a running application server. A message endpoint is simply a Message Driven Bean application that is running inside a J2EE application server. It asynchronously consumes messages from a message provider.

A J2EE version 1.4 compliant application server is required to support version 2.1 of the Enterprise JavaBeans™ specification. This version of the EJB specification defines additional elements for the Message Driven Bean deployment descriptor to support the message inflow system contract of the J2EE Connector Architecture. These deployment descriptor elements are discussed in more detail in 10.4.6, "Message Driven Bean activation configuration properties" on page 401.

10.3.2 MessageEndpointFactory

The J2EE Connector Architecture requires that application server vendors provide a MessageEndpointFactory implementation. A MessageEndpointFactory is used by the resource adapter to obtain references to new, or unused, message endpoint instances in order to process messages. In other words, the resource adapter uses the MessageEndpointFactory to obtain references to Message Driven Beans. Multiple message endpoint instances can be created for a single message endpoint, enabling messages to be processed concurrently.

10.3.3 Resource adapters

A *resource adapter* is the component that maps the proprietary API exposed by the EIS to the API defined by the J2EE Connector Architecture. Resource adapters are also commonly referred to as *connectors*.

The resource adapter itself runs in the same process as the application server and is responsible for delivering messages to the message endpoints hosted by the application server.

Resource adapter packaging

A resource adapter is provided typically by the messaging provider or a third party and comes packaged in a Resource Adapter Archive (RAR) file. This RAR must be packaged using the Java archive (JAR) file format and can contain:

► Any utility classes
► Native libraries required for any platform dependencies
► Documentation
► A deployment descriptor
► Java classes that implement the J2EE Connector Architecture contracts and any other functionality of the adapter

The only element of the RAR file that is required is the deployment descriptor. The deployment descriptor must called ra.xml and must be placed in the META-INF subdirectory of the RAR file.

The resource adapter is installed normally on the application server so that it is available to several J2EE applications at runtime. However, it is possible to package the resource adapter within the message endpoint application.

WebSphere Application Server V6.0 provides a preconfigured resource adapter for the default messaging JMS provider. The RAR file for this resource adapter is called sib.api.jmsra.rar and is located in the lib subdirectory of the WebSphere installation directory.

Resource adapter deployment descriptor

The resource adapter deployment descriptor contains several pieces of information that are used by the application server and the resource adapter at run time, such as:

► Supported message listener types

The resource adapter lists the types of message listener that it supports. The J2EE Connector Architecture version 1.5 and the EJB version 2.1 specifications do not restrict message listeners to using the JMS API.

► ActivationSpec JavaBean

For each message listener type supported for the resource adapter, the deployment descriptor must also specify the Java class name of the ActivationSpec JavaBean. An ActivationSpec JavaBean instance encapsulates the configuration information needed to setup asynchronous message delivery to a message endpoint. Section 10.3.4, "JMS ActivationSpec JavaBean" on page 386 discusses the ActivationSpec JavaBean for JMS providers in more detail.

► Required configuration properties

Each ActivationSpec can also specify a list of required properties. These required properties can be used to validate the configuration of an ActivationSpec JavaBean instance. Example 10-13 shows the messagelistener entry in the deployment descriptor for the default messaging JMS provider. Notice that it supports the JMS message listener (javax.jms.MessageListener) and that the ActivationSpec JavaBean has three required properties; destination, destinationType and busName.

Example 10-13   J2EE Connector Architecture message listener definition

```
<inbound-resourceadapter>
    <messageadapter>
        <messagelistener>
            <messagelistener-type>
                javax.jms.MessageListener
            </messagelistener-type>
            <activationspec>
                <activationspec-class>
                    com.ibm.ws.sib.api.jmsra.impl.JmsJcaActivationSpecImpl
                </activationspec-class>
                <required-config-property>
                    <config-property-name>destination</config-property-name>
                </required-config-property>
                <required-config-property>
                    <config-property-name>destinationType</config-property-name>
                </required-config-property>
                <required-config-property>
```

```
                    <config-property-name>busName</config-property-name>
                </required-config-property>
            </activationspec>
        </messagelistener>
    </messageadapter>
</inbound-resourceadapter>
```

▶ Administered Objects

The resource adapter deployment descriptor can also stipulate a set of administered objects. For each administered object listed, the deployment descriptor must provide the Java class name of the administered object and the interface that it implements.

These administered objects are similar to the JMS administered objects discussed in 10.2.4, "JMS administered objects" on page 367. In fact, for the default messaging JMS provider in WebSphere Application Server V6.0, the J2EE Connector Architecture administers objects that it defined to implement the relevant JMS administered object interfaces. This is shown in Example 10-14.

Example 10-14   J2EE Connector Architecture administered object definition

```
<adminobject>
    <adminobject-interface>
        javax.jms.Queue
    </adminobject-interface>
    <adminobject-class>
        com.ibm.ws.sib.api.jms.impl.JmsQueueImpl
    </adminobject-class>
    <config-property>
        <config-property-name>QueueName</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
    </config-property>

    ... additional properties removed ...

    <config-property>
        <config-property-name>BusName</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
    </config-property>
</adminobject>
```

10.3.4 JMS ActivationSpec JavaBean

An ActivationSpec JavaBean instance encapsulates the configuration information needed to setup asynchronous message delivery to a message endpoint. The J2EE Connector Architecture recommends that JMS providers

include the following properties in their implementation of an ActivationSpec JavaBean:

► destination

Recall that a JMS destination encapsulates addressing information for the JMS provider. A JMS client explicitly specifies a destination when sending a message to, or receiving a message from, the JMS provider. A message endpoint needs to specify which destination the resource adapter should monitor for incoming messages. The resource adapter is then responsible for notifying the message endpoint when a message arrives at that destination.

The J2EE Connector Architecture does not define the format for the destination property, but it does acknowledge that it is not always practical for a value to be set in the deployment descriptor for a message endpoint application. However, a value for the destination property is required when deploying the message endpoint application. For this reason, the J2EE Connector Architecture recommends that a JMS resource adapter defines the destination property as a required property on the ActivationSpec JavaBean. The resource adapter for the default messaging JMS provider within WebSphere Application Server V6.0 does just this, as shown in Example 10-13 on page 385.

The J2EE Connector Architecture also recommends that, if the destination object implements the javax.jms.Destination interface, the JMS resource adapter should provide an administered object that implements this same interface. Once again, the resource adapter for the default messaging JMS provider within WebSphere Application Server V6.0 does just this, as shown in Example 10-14 on page 386.

► destinationType

The destinationType property simply indicates whether the destination is a JMS queue or JMS topic. The valid values for this property are, therefore, javax.jms.Queue or javax.jms.Topic. The J2EE Connector Architecture recommends that a JMS resource adapter defines the destinationType property as a required property on the ActivationSpec JavaBean. The resource adapter for the default messaging JMS provider within WebSphere Application Server V6.0 does just this, as shown in Example 10-13 on page 385.

The following optional properties can also be provided for further customization of the messaging behavior.

► messageSelector

The JMS ActivationSpec JavaBean can define a messageSelector property. JMS message selectors are discussed in "Types of message selectors" on page 373.

- ► acknowledgeMode

  The JMS ActivationSpec JavaBean can define an acknowledgeMode property. This property indicates to the EJB container, how a message received by a message endpoint should be acknowledged. Valid values for this property are Auto-acknowledge or Dups-ok-acknowledge. If no value is specified, Auto-acknowledge is assumed.

  For a full description of message acknowledgement, see both the JMS version 1.1 and the EJB version 2.1 specifications. Links for these specifications are in 10.9, "References and resources" on page 444.

- ► subscriptionDurability

  The JMS ActivationSpec JavaBean defines a subscriptionDurability property. This property is only relevant if the message endpoint is receiving messages from a JMS topic (the destinationType property specifies a value of javax.jms.Topic).

  As discussed in 10.2.10, "JMS message consumers" on page 374, in the JMS Publish/Subscribe domain, in order for a message to be retained on a destination when a subscribing application is not available, the subscribing application must create a durable subscription. With MDB's, it is the EJB container that is responsible for creating subscriptions when the specified destination is a JMS topic. This property indicates to the EJB container whether it must create a durable subscription to the JMS topic.

  The valid values for the subscriptionDurability property are either Durable or NonDurable. If no value is specified, NonDurable is assumed.

- ► clientId

  The JMS ActivationSpec JavaBean can stipulate a clientId property. This property is only relevant if the message endpoint defines a durable subscription to a JMS topic (the destinationType property specifies a value of javax.jms.Topic and the subscriptionDurability property specifies a value of Durable).

  The JMS provider uses the clientId for durable subscriptions to uniquely identify a message consumer. If a message endpoint defines a durable subscription, then a value for the clientId property must be specified. A suitable value for the clientId property would normally be specified when deploying the message endpoint application.

- ► subscriptionName

  The JMS ActivationSpec JavaBean can describe a subscriptionName property. This property is only relevant if the message endpoint defines a durable subscription to a JMS topic (the destinationType property specifies a value of javax.jms.Topic and the subscriptionDurability property specifies a value of Durable).

The JMS provider uses the subscriptionName in combination with the clientId to uniquely identify a message consumer. If a message endpoint defines a durable subscription, then a value for the subscriptionName property must be specified. A suitable value for the subscriptionName property would normally be specified when deploying the message endpoint application.

10.3.5 Message endpoint deployment

Before any messages can be delivered to a message endpoint, the message endpoint must be associated with a destination. This task is performed during application installation. Therefore, the responsibility of associating a Message Driven Bean with a destination lies with the application deployer.

The application deployer creates an instance of the ActivationSpec JavaBean for the relevant resource adapter and associates it with the message endpoint during installation. In this way an ActivationSpec JavaBean, through its destination property, associates a message endpoint with a destination on the message provider. This relationship is shown in Figure 10-9 on page 389.
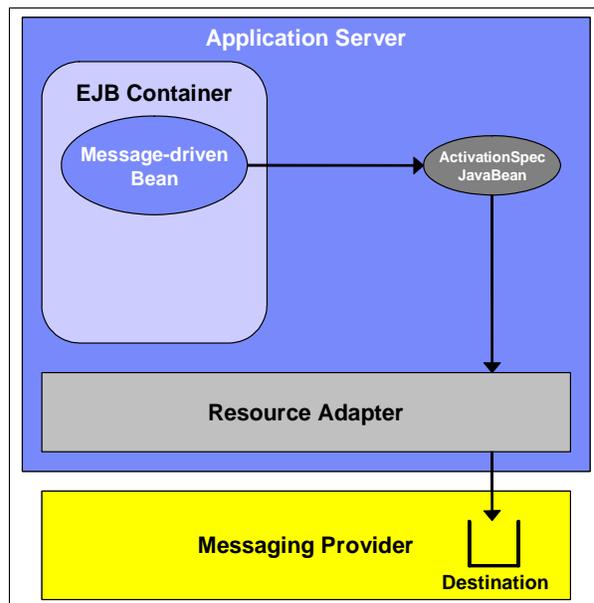
Figure 10-9   ActivationSpec JavaBean associating an MDB with a destination

10.3.6 Message endpoint activation

A message endpoint is activated by the application server when the message endpoint application is started. During message endpoint activation, the

application server passes the ActivationSpec JavaBean, and a reference to the MessageEndpointFactory, to the resource adapter by invoking its endpointActivation method.

The resource adapter uses the information in the ActivationSpec JavaBean to interact with messaging provider and setup message delivery to the message endpoint. For a JMS Message Driven Bean, this can involve configuring a message selector or a durable subscription against the destination. Once the endpointActivation method returns, the message endpoint is ready to receive messages. This process is shown in Figure 10-10 on page 390.
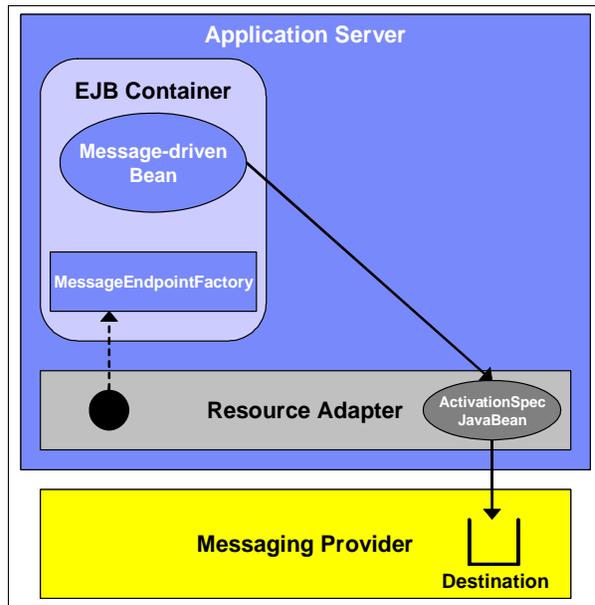
Figure 10-10   Activating a message endpoint

10.3.7 Message delivery

The following steps describe the sequence of events that occur when a message arrives at a destination:

1. The resource adapter detects the arrival of a message at the destination.

2. The resource adapter invokes the createEndpoint method on the MessageEndpointFactory.

3. The MessageEndpointFactory obtains a reference to a message endpoint. This can be an unused message endpoint obtained from a pool or, if no message endpoints are available, it can create a new message endpoint.

4. The MessageEndpointFactory returns a proxy to this message endpoint instance to the resource adapter.

5. The resource adapter uses the message endpoint proxy to deliver the message to the message endpoint.

This process is shown in Figure 10-11 on page 391.
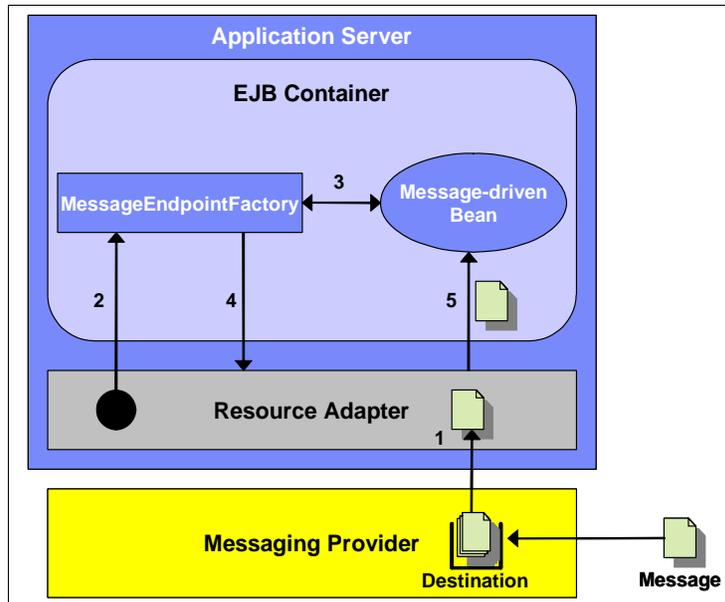
Figure 10-11   Delivering a message to a message endpoint

10.3.8 Administered objects

The resource adapter deployment descriptor defines the list of administered objects implemented by the resource adapter. However, it does not define any administered object instances. This must still be performed as an administrative task within the WebSphere Administrative Console. Because the default messaging JMS provider is specific to the JMS programming model, the WebSphere Administrative Console provides a set of JMS specific administration panels for this resource adapter. In the example provided in section 10.6, "Setup JMS the environment" on page 408 the steps required to configure administered objects for the default messaging JMS provider for the Sal404 application are shown in detail.

10.4 Message Driven Beans

The Enterprise JavaBeans specification (EJB), version 2.0 introduced a new type of EJB called the Message Driven Bean (MDB). *Message Driven Beans* are asynchronous message consumers that run within the context of an application server's EJB container. This enables the EJB container to provide additional services to the Message Driven Bean during the processing of a message, such as transactions, security, concurrency and message acknowledgement.

The EJB container is also responsible for managing the lifetime of the Message Driven Beans and for invoking Message Driven Beans when a message arrives for which a given Message Driven Bean is the consumer.

Message Driven Bean instances should not maintain any conversational state on behalf of a client. This enables the EJB container to maintain a pool of Message Driven Bean instances and to select any instance from this pool to process an incoming message. However, this does not prevent a Message Driven Bean from maintaining state that is not specific to a client, for instance, DataSource references or references to another EJB.

WebSphere Application Server V6.0 is fully compliant with version 1.4 of the J2EE specification, which requires application servers to support version 2.1 of the EJB specification.

10.4.1 Message Driven Bean types

Version 2.0 of the EJB specification defined a single type of Message Driven Bean that enabled the asynchronous delivery of messages through the Java Message Service (JMS).

However, the integration of multiple JMS providers into application servers has proven difficult. For various reasons, many application server vendors have only provided support for one JMS provider within their product. Also, the fact that Message Driven Beans within the EJB 2.0 specification only support the JMS programming model was considered too restrictive. Several other messaging providers exist that require similar functionality to Message Driven Beans within the EJB container, such as the Java API for XML Messaging (JAXM).

Because of this, version 2.1 of the EJB specification expanded the definition of Message Driven Beans to provide support for messaging providers other than JMS providers. It does this by allowing a Message Driven Bean to implement an interface other than the javax.jms.MessageListener interface. This message listener interface is specific to the messaging provider in question. The type of message listener interface that a Message Driven Bean implements determines

its type. Therefore, a Message Driven Bean that implements the javax.jms.MessageListener interface is a *JMS Message Driven Bean*.

10.4.2 Client view of a Message Driven Bean

Unlike session and entity beans, Message Driven Beans do not expose home or component interfaces. A client is not able to locate instances of a Message Driven Bean and invoke methods on it directly.

The only manner in which a client can interact with a Message Driven Bean is to send a message to the destination or endpoint for which the Message Driven Bean is the listener. The EJB container is responsible for invoking an instance of the Message Driven Bean as a result of the arrival of a message. From the clients perspective, the existence of the Message Driven Bean is completely transparent. This is shown in Figure 10-12, where the client is only able to see the messaging provider and the target destination.
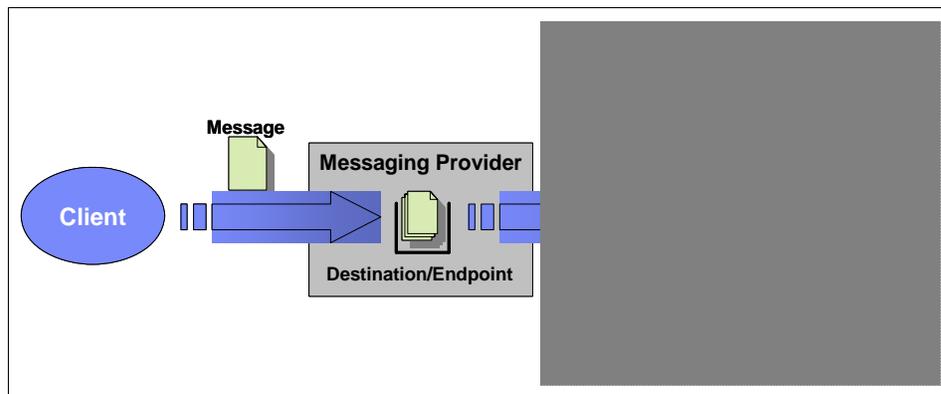
Figure 10-12   Client view of a Message Driven Bean

10.4.3 Message Driven Bean implementation

A bean provider developing a Message Driven Bean must provide a Message Driven Bean implementation class. This class must implement, directly or indirectly, the javax.ejb.MessageDrivenBean interface and a message listener interface. It must also provide an ejbCreate method implementation. These aspects of message-driven implementation are discussed in the following sections.

Message Driven Bean interface

The javax.ejb.MessageDrivenBean interface defines a number of callback methods that allow the EJB container to manage the life cycle of each Message

Driven Bean instance. Since Message Driven Beans expose no home or component interfaces, the javax.ejb.MessageDrivenBean interface defines fewer callback methods than the corresponding javax.ejb.SessionBean and java.ejb.EntityBean interfaces. The definition of the javax.ejb.MessageDrivenBean interface is shown in Example 10-15.

Example 10-15   The javax.ejb.MessageDrivenBean interface

```
public interface MessageDrivenBean extends javax.ejb.EnterpriseBean
{
    public void setMessageDrivenContext(MessageDrivenContext ctx);
    public void ejbRemove();
}
```

The purpose of each of the callback methods is described below:

► setMessageDrivenContext

   The EJB container invokes this method to associate a context with an instance of a Message Driven Bean. The Message Driven Bean instance normally stores a reference to the context as part of its state.

► ejbRemove

   The EJB container invokes this method to notify the Message Driven Bean instance that it is in the process of being removed. This gives the Message Driven Bean the opportunity to release any resources that it may be holding.

Message listener interface

As discussed in section 10.4.1, "Message Driven Bean types" on page 392, version 2.1 of the EJB specification no longer requires a Message Driven Bean to implement the javax.jms.MessageListener interface. The specification simply states that a Message Driven Bean is required to implement the appropriate message listener interface for the messaging type that the Message Driven Bean supports.

The specification also allows the message listener interface to define more than one message listener method and for these methods to specify return types. If a messaging provider has defined an interface that contains more than one message listener method, it is the responsibility of the resource adapter to determine which of these methods to invoke upon the receipt of a message.

The message listener interface for JMS Message Driven Beans is the javax.jms.MessageListener interface, as shown in Example 10-7 on page 376.

As an example of other types of message listener interfaces that might be used by messaging providers, again consider a theoretical JAXM messaging provider. A JAXM messaging provider might decide to use the

javax.xml.messaging.ReqRespListener interface as its message listener interface. This interface is shown in Example 10-16.

Example 10-16   The javax.xml.messaging.ReqRespListener interface

```
package javax.xml.messaging;

import javax.xml.soap.SOAPMessage;

public interface ReqRespListener
{
    public SOAPMessage onMessage(SOAPMessage message);
}
```

Notice that this interface is similar to the javax.jms.MessageListener interface in that it defines an onMessage method. However, you can use any method name d when defining methods within the message listener interface.

Also, notice that the onMessage method specifies a return type of SOAPMessage. The SOAPMessage can be considered to be a reply message. However, since it is the EJB container that invokes the onMessage method, the SOAPMessage will be returned to the EJB container. The EJB specification states that, if the message listener interface supports the request-reply pattern in this manner, it is the responsibility of the EJB container to deliver the reply message.

The ejbCreate method
One other requirement on the implementation class for a Message Driven Bean is that it implements the ejbCreate method. Once again, this implementation can be defined within the Message Driven Bean class itself, or within any of its superclasses. The EJB container invokes the ejbCreate as the last step in creating a new instance of a Message Driven Bean. This gives the Message Driven Bean the opportunity to allocate any resources that it requires.

10.4.4 Message Driven Bean life cycle

The EJB container is responsible for hosting and managing Message Driven Bean instances. It controls the life cycle of the Message Driven Bean and uses the callback methods within the bean implementation class to notify the instance when important state transitions are about to occur.

The life cycle of a Message Driven Bean is shown in Figure 10-13 on page 396.
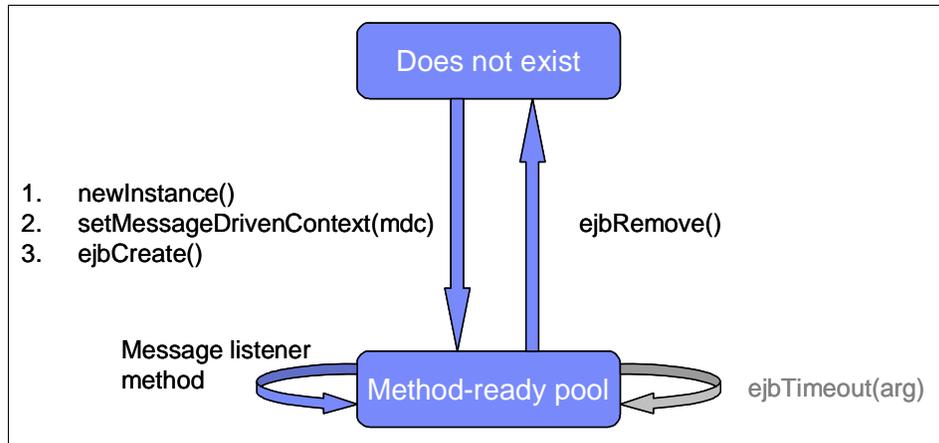
Figure 10-13   Message Driven Bean life cycle

The relevant state transitions for a Message Driven Bean are described below:

► Message Driven Bean creation

Message Driven Bean instances are created in three steps by the EJB container:

a. The EJB container invokes the Class.newInstance() method on the bean implementation class.

b. The EJB container provides the new instance with its MessageDrivenContext reference by invoking the setMessageDrivenContext method.

c. The EJB container gives the new Message Driven Bean instance the opportunity to perform one-time initialization by invoking the ejbCreate method. The Message Driven Bean is able to allocate any resources that it requires here.

► Message listener method invocation

Once in the method-ready pool, a Message Driven Bean instance is available to process any message that is sent to its associated destination or endpoint. When a message arrives at this destination, the EJB container receives the message and allocates a Message Driven Bean instance from the method-ready pool to process the message. When processing is complete, the Message Driven Bean instance is returned to the method-ready pool.

> **Note:** The EJB container performs a number of other operations during the processing of a message, such as ensuring that the processing takes place within the specified transactional context and performing any required security checks. These steps have been omitted for clarity.

► Message Driven Bean removal

The EJB container can decide at any time that it needs to release resources. To do this, it can reduce the number of Message Driven Bean instances in the method-ready pool. As part of the removal process, it will invoke the ejbRemove method on the instance being removed to give the Message Driven Bean the opportunity to release any resources that it might be holding.

10.4.5 Message Driven Beans and transactions

A bean provider can specify whether a message-driven will demarcate its own transactions programmatically or whether it will rely on the EJB container to demarcate transactions on its behalf. The bean provider does this by specifying either Bean or Container as the value for the transaction-type field for the Message Driven Bean, in the EJB module deployment descriptor.

Regardless of whether transaction demarcation is bean-managed or container-managed, a Message Driven Bean can only access the transactional context within which it is running by using the relevant methods of the MessageDrivenContext interface.

MessageDrivenContext interface

The javax.ejb.MessageDrivenContext interface extends the javax.ejb.EJBContext interface. However, unlike the SessionContext and EntityContext interfaces, the MessageDrivenContext interface does not define any additional methods. The parent EJBContext interface is shown in Example 10-17.

Example 10-17   The javax.ejb.EJBContext interface

```
package javax.ejb;

import java.security.Identity;
import java.security.Principal;
import java.util.Properties;
import javax.transaction.UserTransaction;

public interface EJBContext
{
   // EJB Home methods
   public abstract EJBHome getEJBHome();
```

```
        public abstract EJBLocalHome getEJBLocalHome();

        // Security methods
        public abstract Principal getCallerPrincipal();
        public abstract boolean isCallerInRole(String s);

        // Transaction methods
        public abstract UserTransaction getUserTransaction()
            throws IllegalStateException;
        public abstract void setRollbackOnly() throws IllegalStateException;
        public abstract boolean getRollbackOnly() throws IllegalStateException;

        // Timer service methods
        public abstract TimerService getTimerService()
            throws IllegalStateException;

        // Deprecated Methods
        public abstract Properties getEnvironment();
        public abstract Identity getCallerIdentity();
        public abstract boolean isCallerInRole(Identity identity);
}
```

Note: A Message Driven Bean instance should only invoke the transaction and timer service methods exposed by the MessageDrivenContext interface.

Attempting to invoke the EJB home methods results in a java.lang.IllegalStateException being thrown because Message Driven Beans do not define EJBHome or EJBLocalHome objects.

Attempting to invoke the getCallerPrincipal method is allowed by version 2.1. of the EJB specification. However, with a Message Driven Bean, the caller is the EJB container, which does not have a client security context. In this situation, the getCallerPrincipal method returns a representation of the unauthenticated identity. Invoking the isCallerInRole method is still not allowed by the EJB specification and will result in a java.lang.IllegalStateException being thrown.

Container-managed transactions

A Message Driven Bean with a transaction-type of Container is said to make use of *container-managed transactions*. When a Message Driven Bean is using container managed transactions, the EJB container uses the transaction attribute of the message listener method to determine the actions that it needs to take when a message arrives at the relevant destination.

The transaction attributes that can be specified for message listener method are as follows:

- ► NotSupported

  The EJB container does not create a transaction prior to receiving the message from the destination and invoking the message listener method on the Message Driven Bean. Consequently, for a JMS Message Driven Bean, the message will not be placed back on the destination for redelivery if an error occurs during the processing of the message.

- ► Required

  The EJB container creates a transaction prior to receiving the message from the destination and invoking the message listener method on the Message Driven Bean.

  If the Message Driven Bean accesses a resource manager within the message listener method, then this access takes place within the context of this transaction. Similarly, if the Message Driven Bean invokes other EJBs within the message listener method, the EJB container passes the transaction context with the invocation.

  When the message listener method completes, the EJB container attempts to commit the transaction. For a JMS Message Driven Bean, a rollback of the transaction should have the effect of placing the message back on the destination for redelivery.

When a message listener method specifies a transaction attribute of Required, it may only use the getRollbackOnly and setRollbackOnly methods of the MessageDrivenContext object. The code required to mark a transaction for rollback within a message listener method is shown in Example 10-18.

Example 10-18   Using the setRollbackOnly method

```
public class SampleMDBBean implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext msgDrivenCtx;

    // Lifecycle methods removed for clarity

    public void onMessage(Message msg)
    {
        try
        {
            // Process the message

            // Try to access a relational database
        }
        catch (SQLException e)
        {
            // An error occured, rollback the transaction
            msgDrivenCtx.setRollbackOnly();
```

```
        }
    }
}
```

Bean-managed transactions

A Message Driven Bean with a transaction-type of Bean is said to make use of
bean-managed transactions. When a Message Driven Bean is using
bean-managed transactions, the EJB container does not create a transaction
prior to receiving the message from the destination and invoking the message
listener method on the Message Driven Bean. Consequently, for a JMS Message
Driven Bean, the message will not be placed back on the destination for
redelivery if an error occurs during the processing of the message. The message
listener method is responsible for creating any transactions that it requires when
processing a message.

A Message Driven Bean using bean managed transactions may only use the
getUserTransaction method of the MessageDrivenContext object. It is then able
to use the javax.transaction.UserTransaction interface to begin, commit and
rollback transactions. The code required to use the UserTransaction interface
within a message listener method is shown in Example 10-18.

Example 10-19   Using the javax.transaction.UserTransaction interface

```
public class SampleMDBBean implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext msgDrivenCtx;

    // Lifecycle methods removed for clarity

    public void onMessage(Message msg)
    {
        // Get the UserTransaction object reference
        UserTransaction userTx = msgDrivenCtx.getUserTransaction();

        try
        {
            // Begin the transaction
            userTx.begin();

            // Process the message

            // Try to access a relational database

            // Attempt to commit the transaction
            userTx.commit();
        }
        catch (Exception e)
```

```
        {
            try
            {
                // An error occured, rollback the transaction
                userTx.rollback();
            }
            catch (SystemException e2)
            {
                e2.printStackTrace();
            }
        }
    }
}
```

> **Note:** Because of the complex nature of distributed transactions, it is
> recommended that bean providers make use of container-managed
> transactions.

10.4.6 Message Driven Bean activation configuration properties

The way in which Message Driven Beans specify deployment options within the
EJB deployment descriptor has changed significantly for EJB version 2.1. This
reflects the changes made to the J2EE Connector Architecture specification to
enable a resource adapter to deliver messages asynchronously to a Message
Driven Bean, independent of the specific messaging style, messaging semantics
and messaging infrastructure. Consequently, version 2.1 of the EJB specification
introduced a more generic mechanism to specify the messaging semantics of a
Message Driven Bean, known as *activation configuration properties*.

The EJB specification defines the following activation configuration properties for
a JMS Message Driven Bean:

► destinationType
► messageSelector
► acknowledgeMode
► subscriptionDurability

Notice that the names of these activation configuration properties match the
names of the equivalent JMS ActivationSpec JavaBean properties described in
10.3.4, "JMS ActivationSpec JavaBean" on page 386. The description of each of
the properties is also the same.

This is intentional on the part of the J2EE Connector Architecture and the EJB
specifications. The intention is that this allows the automatic merging of the
activation configuration element values with the corresponding entries in the JMS
ActivationSpec JavaBean, while configuring the JMS ActivationSpec JavaBean

during endpoint deployment. This is exactly what happens when WebSphere starts an application that contains a Message Driven Bean.

> **Note:** If a Message Driven Bean and the JMS activation specification with which it is associated both specify a value for a given property, it is the value contained in the EJB deployment descriptor for the Message Driven Bean that will be used.

Example 10-20 on page 402, shows the relevant entry for the BankListener Message Driven Bean that is packaged as part of the WebSphereBank sample in WebSphere Application Server V6.0. The elements of the deployment descriptor that are specific to messaging are shown in bold. Table 10-8 shows activation configuration properties are defined within the deployment descriptor:

Table 10-8   Configuration for the BankListener Message Driven Bean

Property name	Property value
destinationType	javax.jms.Queue
acknowledgeMode	Auto-acknowledge
messageSelector	JMSType = 'transfer'

Example 10-20   BankListener Message Driven Bean deployment descriptor

```
<message-driven id="MessageDriven_1037986117955">
   <ejb-name>BankListener</ejb-name>
   <ejb-class>com.ibm.websphere.samples.bank.ejb.BankListenerBean</ejb-class>
   <messaging-type>javax.jms.MessageListener</messaging-type>
   <transaction-type>Container</transaction-type>
   <message-destination-type>javax.jms.Queue</message-destination-type>
   <message-destination-link>BankJSQueue</message-destination-link>
   <activation-config>
      <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-property>
         <activation-config-property-name>
            acknowledgeMode
         </activation-config-property-name>
         <activation-config-property-value>
            Auto-acknowledge
```

```
                </activation-config-property-value>
            </activation-config-property>
            <activation-config-property>
                <activation-config-property-name>
                    messageSelector
                </activation-config-property-name>
                <activation-config-property-value>
                    JMSType = 'transfer'
                </activation-config-property-value>
            </activation-config-property>
        </activation-config>
        <ejb-local-ref id="EJBLocalRef_1037986243867">
            <description></description>
            <ejb-ref-name>ejb/Transfer</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <local-home>
                com.ibm.websphere.samples.bank.ejb.TransferLocalHome
            </local-home>
            <local>com.ibm.websphere.samples.bank.ejb.TransferLocal</local>
            <ejb-link>Transfer</ejb-link>
        </ejb-local-ref>
</message-driven>
```

10.4.7 Associating a Message Driven Bean with a destination

Before any messages can be delivered to a Message Driven Bean, the Message
Driven Bean must be associated with a destination. As discussed in 10.3.5,
"Message endpoint deployment" on page 389, the responsibility of associating a
Message Driven Bean with a destination lies with the application deployer.

Within WebSphere Application Server V6.0, there are two mechanisms that can
be used to associate these objects, JMS activation specifications and listener
ports. The reason is that the service integration bus within WebSphere
Application Server V6.0 is accessed using a J2EE Connector Architecture
resource adapter, while WebSphere MQ is accessed with a standard JMS API.

If the Message Driven Bean being deployed needs to be associated with a
destination defined on a Service Integration Bus, a JMS activation specification
should be used. If the Message Driven Bean being deployed needs to be
associated with a destination defined on WebSphere MQ, a listener port should
be used. The following sections discuss JMS activation specifications and listner
ports.

JMS activation specification

An ActivationSpec JavaBean, through its destination property, associates a
message endpoint with a destination. Within WebSphere Application Server

V6.0, an instance of the ActivationSpec JavaBean for the default messaging JMS provider is configured by creating a JMS activation specification using the WebSphere Administrative Console. These JMS activation specifications are normally created prior to installing the Message Driven Bean application and are stored in the JNDI name space by WebSphere Application Server.

At installation time, the deployer specifies which JMS activation specification to associate with a particular Message Driven Bean, using its JNDI name. The destination property within the JMS activation specification, specifies the JNDI name of the target JMS destination. This relationship is shown Figure 10-14.
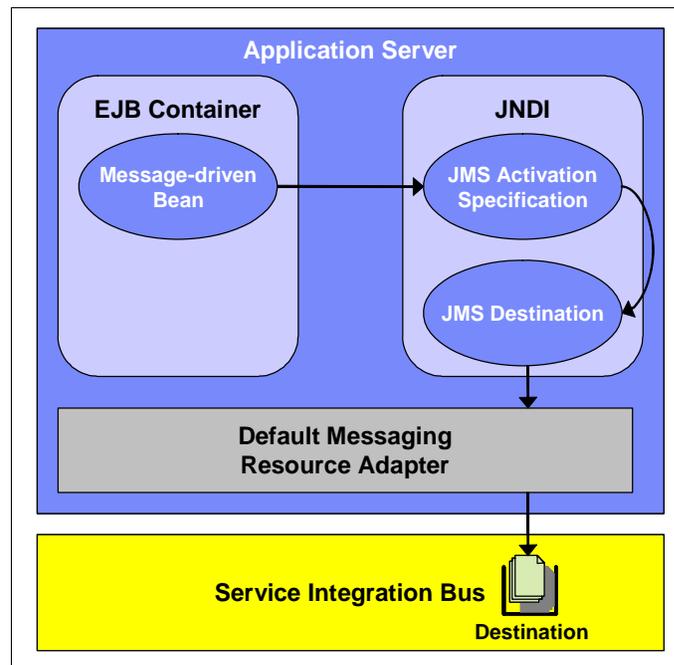
Figure 10-14   Associating an MDB with a destination using a JMS Activation Specification

The steps required to create a JMS activation specification for the default messaging JMS provider for the Sal404 application are described in "Set up an activation specification" on page 418.

Listener ports

Prior to version 1.5 of the J2EE Connector Architecture, there was no standard way to associate a Message Driven Bean with a destination. To solve this problem, WebSphere Application Server V5 introduced the concept of a listener port. Because this has been replaced in V6 with the activation specification, we do not discuss listener ports here.

If you are looking for details about how to configure listener ports in WebSphere Application Server V6.0, refer to the appropriate sections in the redbook *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446.

10.4.8 Message Driven Bean best practices

As with all programming models, certain best practices have emerged for using the Message Driven Bean programming model. These best practices are discussed in this section:

▶ Delegate business logic to another handler.

Traditionally, the role of a stateless session bean is to provide a facade for business logic. Message Driven Beans should delegate the business logic of processing the contents of a message to a stateless session bean. Message Driven Beans can focus then, on what they were designed to do, which is processing messages. This is shown in Figure 10-15.
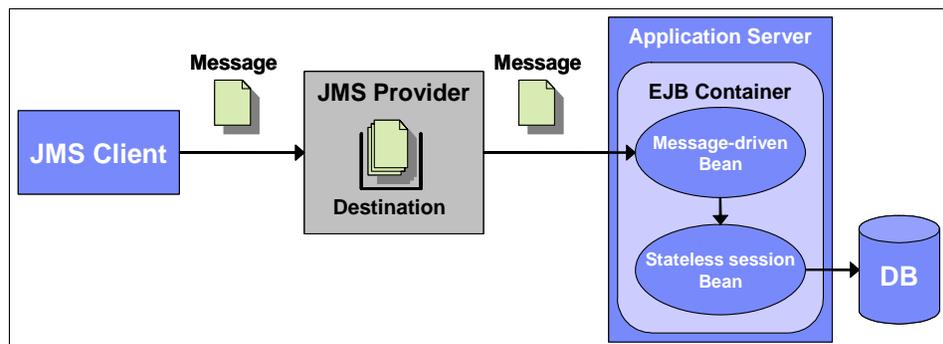
Figure 10-15   Delegating business logic to a stateless session bean

An additional benefit of this approach is that the business logic within the stateless session bean can be reused by other EJB clients. This is shown in Figure 10-16 on page 406.
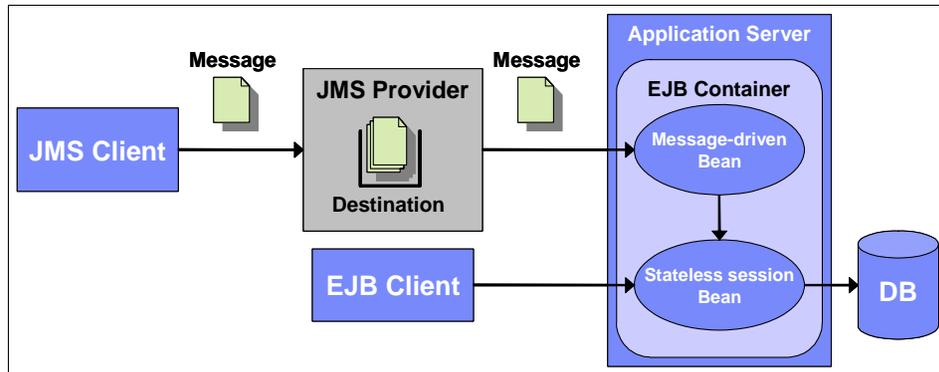
Figure 10-16   Business logic reuse

► Do not maintain client-specific state within an MDB.

As discussed earlier, Message Driven Bean instances should not maintain any conversational state on behalf of a client. This enables the EJB container to maintain a pool of Message Driven Bean instances and to select any instance from this pool to process an incoming message. However, this does not prevent a Message Driven Bean from maintaining state that is not specific to a client, for instance, DataSource references or references to another EJB.

► Avoid large message bodies.

A JMS message will probably need to travel over the network at some point in its life. It will definitely need to be handled by the JMS provider. All of these components contribute to the overall performance and reliability of the system. The amount of data contained in the body of a JMS message should be kept as small as possible to avoid impacting the performance of the network or the JMS provider.

► Minimize message processing time.

Recall from the discussion in 10.4.4, "Message Driven Bean life cycle" on page 395, that instances of a Message Driven Bean are allocated from the method-ready pool to process incoming messages. These instances are not returned to the method-ready pool until message processing is complete. Therefore, the longer it takes for a Message Driven Bean to process a message, the longer it will be unavailable for reallocation.

If an application is required to process a high volume of messages, the number of Message Driven Bean instances in the method-ready pool could be rapidly depleted if each message requires a significant processing. The EJB container would then need to spend valuable CPU time creating additional Message Driven Bean instances for the method-ready pool, further impacting the performance of the application.

Additional care must be taken if other resources are enlisted into a global transaction during the processing of a message. The EJB container does not attempt to commit the global transaction until the MDB's onMessage method returns. Until the global transaction commits, these resources cannot be released on the resource managers in question.

For these reasons, the amount of time required to process each message should be kept to a minimum.

► Avoid dependencies on message ordering.

An application should try to avoid making any assumptions about the order in which JMS messages are processed. This is due to the fact that application servers enable the concurrent processing of JMS messages by MDBs and that some messages can take longer to process than others. Consequently, a message delivered later in a sequence of messages may finish message processing before a message delivered earlier in the sequence. It can be possible to configure the application server in such a way that messaging ordering is maintained within the application, but this is usually done at the expense of performance or architectural flexibility (inability to deploy an application to a cluster).

► Be aware of poison messages.

Sometimes, a badly-formatted JMS message arrives at a destination. Such a message might cause an exception to be thrown within the MDB during message processing. An MDB that is making use of container-managed transactions would then mark the transaction for rollback, as discussed in 10.4.5, "Message Driven Beans and transactions" on page 397. The EJB container would then rollback the transaction, causing the message to be placed back on the queue for redelivery. However, the same problem occurs within the MDB the next time the message is delivered. In this situation, such a message might be received, and then returned to the queue, repeatedly. These messages are known as *poison messages*.

Fortunately, some messaging providers have implemented mechanisms that can detect poison messages and redirect them to a another destination. WebSphere MQ and the service integration bus are two such providers.

10.5 Service integration bus

The service integration bus (SIB) is a component of WebSphere Application Server V6 that provides a managed communications framework, supporting a variety of message distribution models, reliability options and network topologies. It provides support for traditional messaging applications as well enabling the implementation of service-oriented architectures within the WebSphere Application Server V6.0 environment.

The service integration bus is the underlying messaging provider for the default messaging JMS provider, and it is intended to replace the embedded messaging provider that was supported within WebSphere Application Server V5.

Section 10.6.1, "Set up the SIB" on page 409 provides the details how to setup the service integration bus for our sample application. If you want to read more about concepts, topologies and configuration of the service integration bus in WebSphere Application Server V6.0 please refer to the redbook *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446.

10.6 Setup JMS the environment

To configure WebSphere Application Server for the messaging needed by our sample application the following configuration tasks have to be performed:

► Setup the service integration bus.

    a. Create a new service integration bus.
    b. Specify the server running our application to be a member of that bus.
    c. Set up a message queue in the bus.
    d. restart the server.

► Setup the default messaging.

    – Set up a queue connection factory.
    – Set up a queue for incoming messages.
    – Set up an activation specification.
    – Set up a queue for outgoing messages.

By following these configuration steps, you are creating the administered objects in WebSphere Application Server that are needed to run the JMS functionality in the sample application.

> **Note:** Without this configuration, the application does not run because the server complains about the missing activation specification for the MDB that cannot be found. To run our sample application, you need to configure the JMS setup.

In the next sections we show you how to change your WebSphere Application Server settings in the WebSphere Administrative Console. Whenever the console states that there are unsaved changes, these changes need to be saved. You need to save the changes manually because WebSphere Application Server does not save them automatically.

You can make several changes in the configuration and save them at the end of your session, but you might also choose to save the configuration after each configuration step. The WebSphere Administrative Console will remind you the next time you enter the that you have some unsaved changes. To get confused by these messages, it is a good idea to save the changes every time the message shown in Figure 10-17, is displayed in at the top of the console.

Figure 10-17   Web message for unsaved configuration

So, if in the next sections, you are instructed to Save your configuration changes in the WebSphere Administrative Console, then you click **Save** as shown in Figure 10-18.
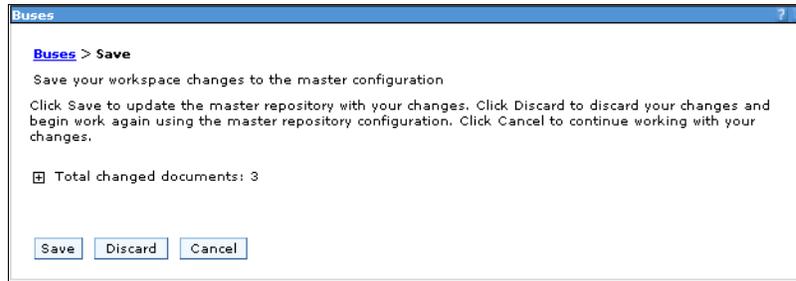
Figure 10-18   Save configuration changes in the WebSphere Administrative Console

For details about using the WebSphere Administrative Console and more options for configuring the messaging in WebSphere Application Server see the redbook *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446

10.6.1 Set up the SIB

This section shows you how to create a service integration bus for the default messaging of WebSphere Application Server V6 and configure it for our application.

Create a new service integration bus

To create a new SIB, follow these steps:

1. Open the WebSphere Administrative Console using

   ```
   http://localhost:9060/admin
   ```

2. Click **Service integration** → **Buses.**

3. Click **New** to create a new bus.

4. Enter `RealtyBus` as the name of the bus and leave the default values shown in Figure 10-19:

Figure 10-19   Setup of a new service integration bus

5. Click **OK.**

6. Save your configuration changes in the WebSphere Administrative Console.

7. Verify that you see the newly created bus in the list when you click **Service integration** → **Buses** as in Figure 10-20 on page 411.
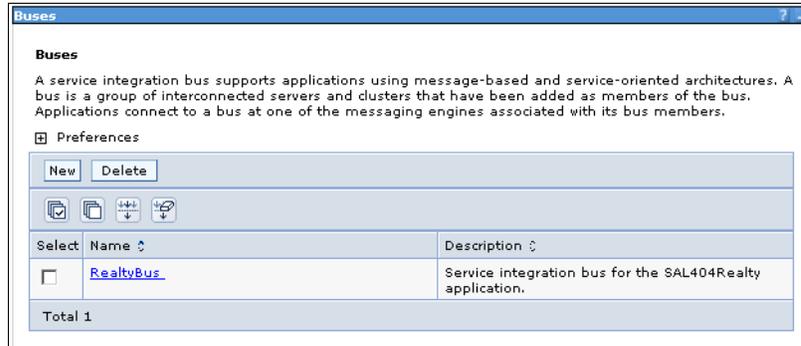
Figure 10-20   The newly created SIB

Add the current server as a bus member

To add the current server as a member of the bus, do the following:

1. Click **Service integration** → **Buses** → **RealtyBus** as shown in Figure 10-20.

2. Click **Bus member** under Additional Properties.

3. Click **Add** to add a new member to the bus.

4. Choose **server** in the radio buttons choice.

5. If you have not set up multiple server instances there should be only one choice in the next screen named *yournodename*:server1, so leave the default settings. If you have set up additional servers, choose the server that runs the Sal404 application. You may add the other servers as bus members too.

6. Leave the default settings for the data store; a JNDI name is not needed when you use the default data store . For configuring anything other than the default, refer to 10.6.3, "Data stores" on page 421)

7. Click **Next.**

8. Click **Finish.**

9. Save your configuration changes in the WebSphere Administrative Console.

10. You are then shown the screen that lists the bus members for the RealtyBus. Verify that you see your bus member or members in the list.

Note: When you click the newly created bus member, you see the messaging engine that is automatically set up and has a state of unavailable. The first time the engine is set up you will not be able to start it. You must restart the server to get the messaging engine to run.

WebSphere Application Server has a setting that lets you start the messaging engine automatically at server startup. It is not enabled on a server by default, however it is automatically enabled if you add a server to a bus. If you disable the SIB service, then any messaging engines defined on the server will not be started.

Because you just added your server to a bus, there is no need to change this setting. You find it in the WebSphere Administrative Console under **Servers** → **Application servers** → **server1** → **SIB service**. Figure 10-21 shows the configuration panel.
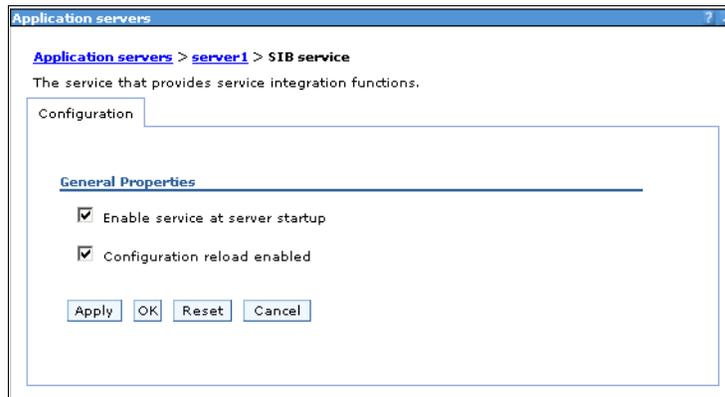
Figure 10-21   Configuration panel for SIB service

Create a queue as a destination

To create a destination for the SIB follow these steps:

1. Click **Service integration** → **Buses** → **RealtyBus** in the WebSphere Administrative Console.

2. Click **Destinations** under **Additional Properties.**

3. Click **New** to create a destination.

4. On the next screen select **Queue** as destination type and click **Next**.

5. Enter `RealtyBusQueue` as the identifier in the next screen, then click **Next**.

6. Choose the bus member from the dropdown menu (there is only one when you have not configured additional servers) and click **Next**.

7. Review settings and click **Finish**.

8. As a result, you can see the new queue under **Service integration** → **Buses** → **RealtyBus** → **Destinations**.

9. To run the messaging engine you must restart the application server. If you use the Test Environment of Rational Web Developer or Rational Application

Developer you can issue the restart from the Servers view. Verify your running messaging engine.

10. You can verify your running messaging engine after the restart if you click **Service integration** → **Buses** → **RealtyBus.**

11. Click **Messaging engine** under **Additional Properties.**

12. Check that a green arrow on the right shows the started state as seen in Figure 10-22.

13. You should also see a statement in your server SystemOut logs that says
    `Messaging engine yournodename.server1-RealtyBus is in state Started`
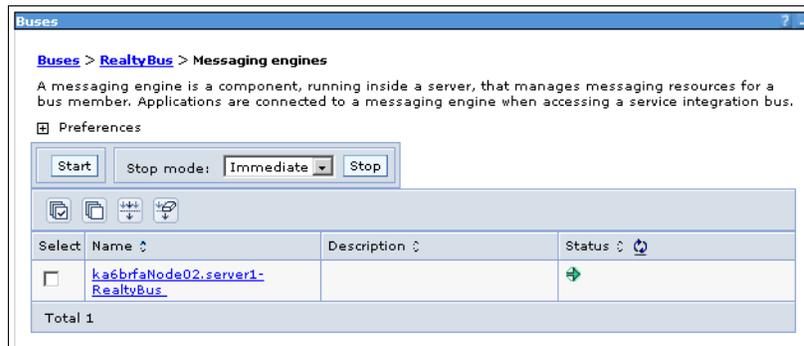
Figure 10-22   Status of messaging engine

Create a queue for outgoing messages

To create a destination for outgoing messages on the SIB, follow these steps:

1. Click **Service integration** → **Buses** → **RealtyBus** in the WebSphere Administrative Console.

2. Click **Destinations** under Additional Properties.

3. Click **New** to create a destination.

4. On the next screen select **Queue** as destination type and click **Next.**

5. Enter `RealtyBusOutQueue` as the queue identifier then click **Next.**

6. Choose the bus member from the dropdown menu and click **Next.**

7. Review the settings and click **Finish.**

8. As a result, you can see the new queue under **Service integration** → **Buses** → **RealtyBus** → **Destinations** .

Verify your queues

To verify your queues, navigate to **Service integration** → **Buses** → **RealtyBus** → **Destinations**.

Figure 10-23 shows the list of queues.
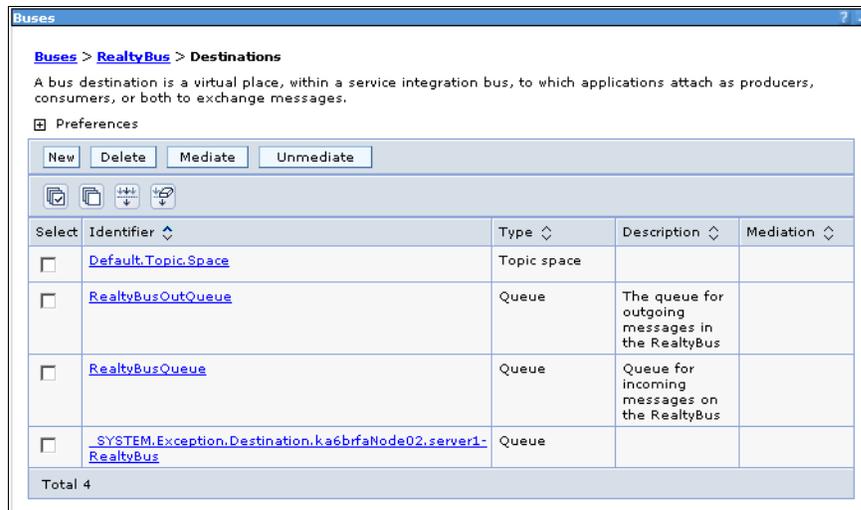
Figure 10-23   Overview of the created queues

10.6.2 Setup the default messaging

Now we can set up the JMS provider for default messaging and use the configured RealtyBus with our configured queue. The next configuration steps start from the screen shown in Figure 10-24 on page 415. Click **Resources** → **JMS Providers** → **Default messaging** in the WebSphere Administrative Console.

The default messaging provider that we configure enables messaging based on the Java Message Service (JMS). It provides connection factories to create connections for JMS destinations.

Note: We always leave the scope (see Figure 10-24 on page 415) setting as its default value of Node scope. This means your administered objects are available to any server that belongs to the node.
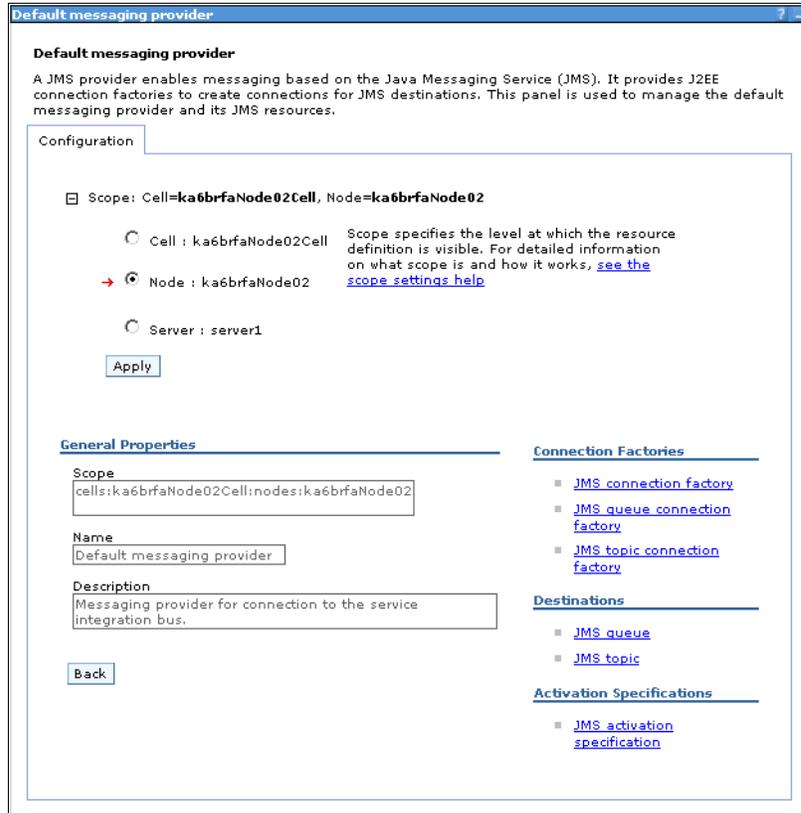
Figure 10-24   Setup of the JMS default messaging provider

Set up a queue connection factory

To set up the queue connection factory, follow these steps:

1. Click **Resources** → **JMS Providers** → **Default messaging** in the WebSphere Administrative Console.

2. Click **JMS queue connection factory** under the `Connection Factories` topic.

3. Click **New.**

4. Enter the following values as shown in Figure 10-25 on page 416:

   – Name: `RealtyConnectionFactory`
   – JNDI name: `JMS/RealtyConnectionFactory`
   – Description (optional): `Connection factory for the Sal404 application`
   – Bus name: Select **RealtyBus** from the drop-down menu.
   – Leave all the other values in their default state.
   – Click **OK** to confirm your settings.

5. Save your configuration changes in the WebSphere Administrative Console.

6. The screen that lists the connection factories is shown. Verify that you see your connection factory in the list.
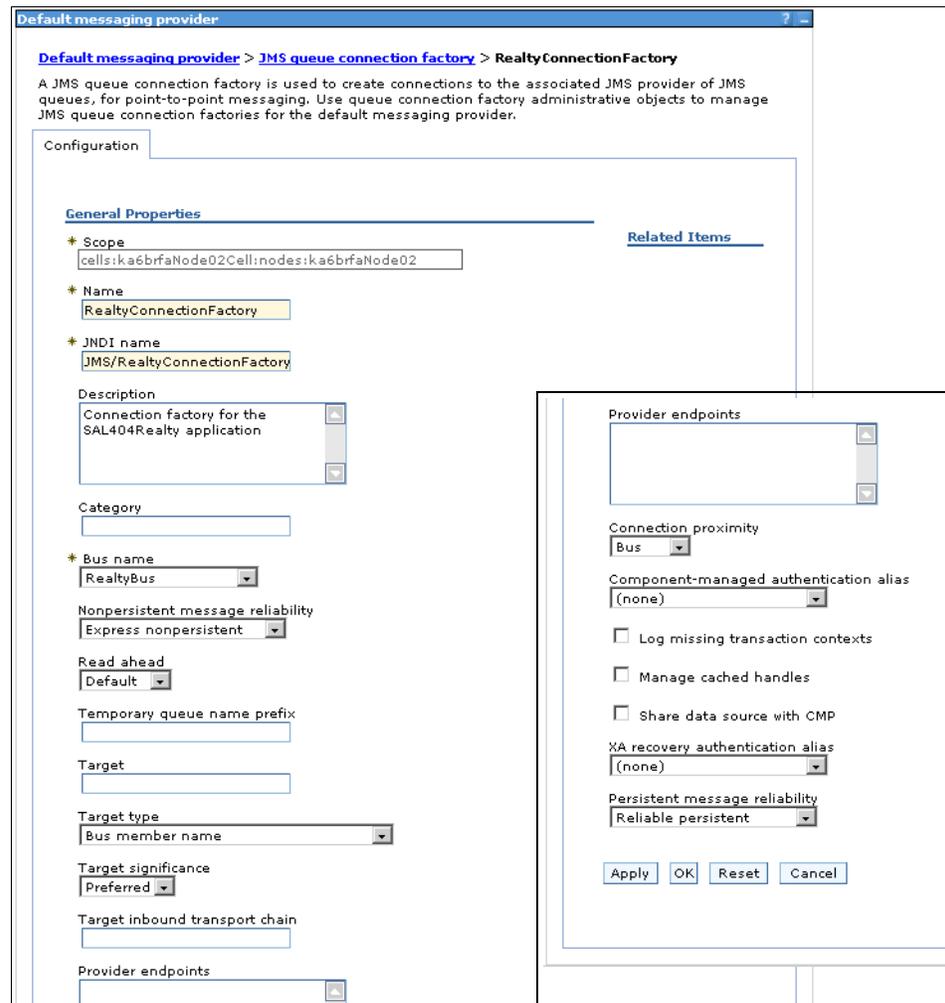
Figure 10-25   Configuration of the connection factory (top and bottom screen area)

Set up a queue for incoming messages

To set up the queue for incoming messages, follow these steps:

1. Click **Resources** → **JMS Providers** → **Default messaging** in the WebSphere Administrative Console.

2. Click **JMS queue** under the **Destinations.**

3. Click **New**.

4. Enter the following values as shown in Figure 10-26 on page 418:

   – Name: `RealtyIncomingQueue`

   – JNDI name: `JMS/RealtyIncomingQueue`

   – Description (optional): `The JMS queue for incoming messages for the Sal404 application`

   – Bus name: Select **RealtyBus** from the drop-down menu

   – If the screen does not change automatically and offer you the buses for the RealtyBus, then click **>>** as shown in Figure 10-26 on page 418.

   – Queue name: Select **RealtyBusQueue** from the drop-down menu.

   – Time to live: `5000` (milliseconds)

5. Click **OK** and verify that you see the new queue in the next screen.

6. Save your configuration changes in the WebSphere Administrative Console.

7. See the created queue in the queue list under **Resources** → **JMS Providers** → **Default messaging** → **JMS queue**.
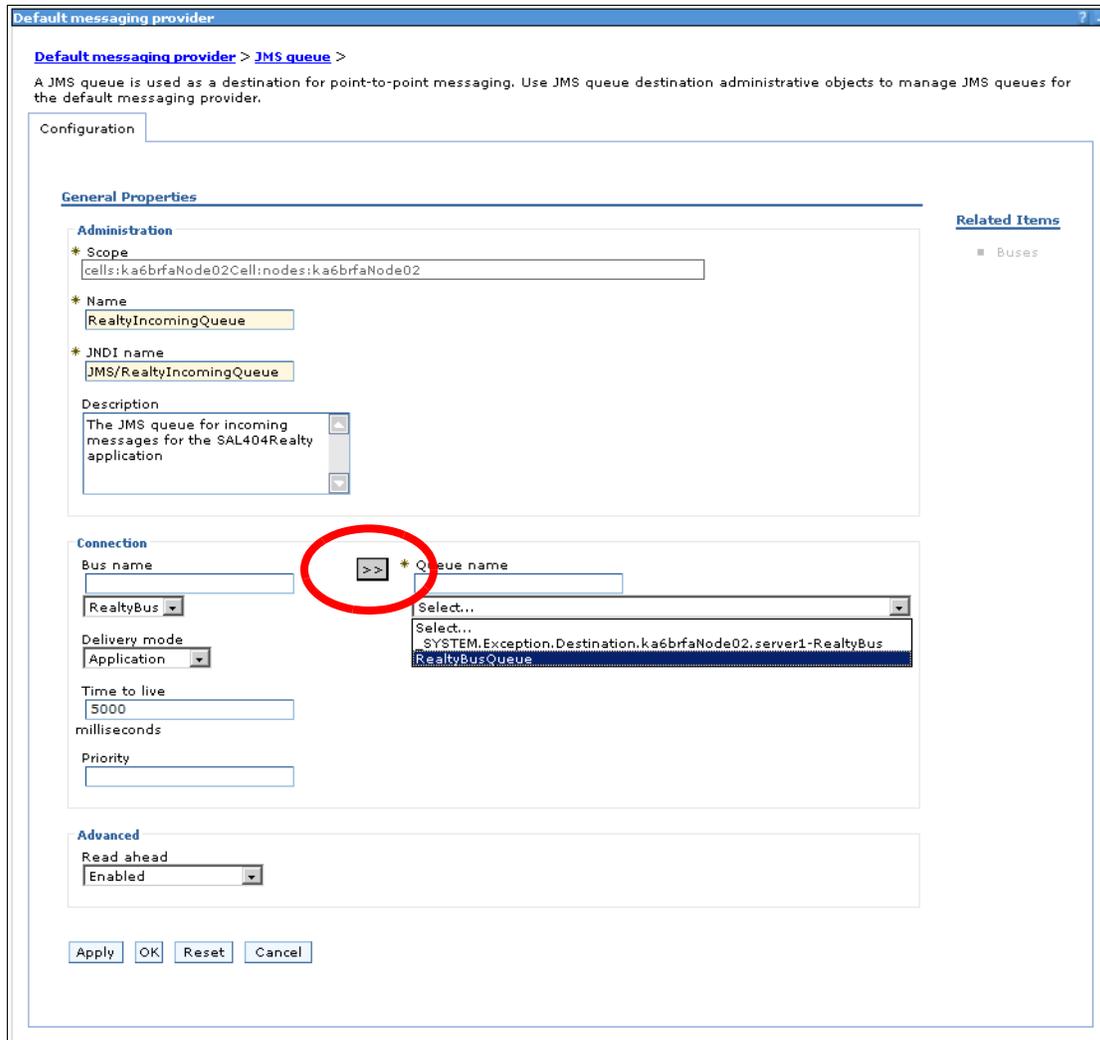
Figure 10-26   Configuration of the queue for incoming messages

Set up an activation specification

To set up an activation specification that can be associated with our MDB, follow these steps:

1. Click **Resources** → **JMS Providers** → **Default messaging** in the WebSphere Administrative Console.

2. Click **JMS activation specification** under **Activation specifications.**

3. Click **New**.

4. Enter the following values as shown in Figure 10-27 on page 419:
   – Name: `RealtyActivationSpec`
   – JNDI name: `JMS/RealtyActivationSpec`
   – Destination type: select **Queue** from the drop-down menu
   – Destination JNDI name: `JMS/RealtyIncomingQueue`
   – Bus name: Select **RealtyBus** from the drop-down menu.
   – Leave all the other values in their default state.
   – Click **OK** to confirm your settings.

5. Save your configuration changes in the WebSphere Administrative Console.

6. The screen that lists the activation specifications is displayed. Verify that you see the activation specification just created in the list.

Figure 10-27   Configuration of the activation specification (top and bottom screen area)

Set up a queue for outgoing messages

To setup a queue for outgoing messages, follow these steps:

1. Click **Resources** → **JMS Providers** → **Default messaging** in the WebSphere Administrative Console.

2. Click **JMS queue** under the **Destinations.**

3. Click **New.**

4. Enter the following values as shown in Figure 10-28 on page 421:

    – Name: `RealtyOutgoingQueue`

    – JNDI name: `JMS/RealtyOutgoingQueue`

    – Description (optional): `The JMS queue for outgoing messages for the Sal404 application`

    – Bus name: Select **RealtyBus** from the drop-down menu.

    – If the screen does not change automatically and offer you the buses for the RealtyBus then click **>>**.

    – Queue name: Select **RealtyBusOutQueue** from the drop-down menu.

5. Click **OK** and verify that you see the new queue in the next screen.

6. Save your configuration changes in the WebSphere Administrative Console.

7. You should see then the created queue in the queue list under **Resources** → **JMS Providers** → **Default messaging** → **JMS queue**.
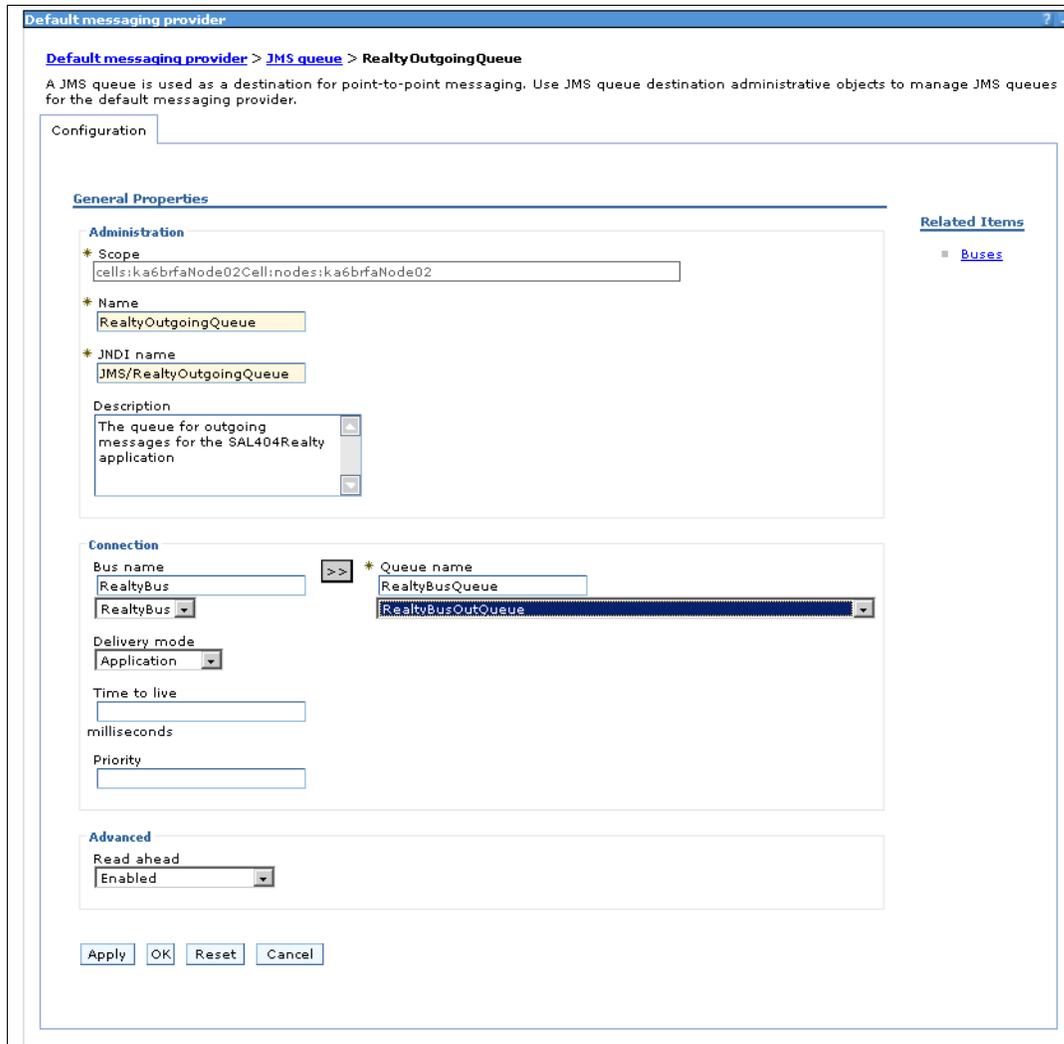
Figure 10-28   Configuration of the queue for outgoing messages

10.6.3 Data stores

Every messaging engine defined within a bus has a data store associated with it. A messaging engine uses this data store to persist durable data, such as persistent messages and transaction states.

The messaging engine can also use the data store to persist volatile data. For example, the messaging engine might write non-persistent messages to the data store to reduce the size of the Java heap when handling high message volumes.

Durable data written to the data store survives the failure of a messaging engine, regardless of the reason for the failure. Volatile data, on the other hand, does not survive the failure of a messaging engine, and might or might not survive an orderly shutdown of a messaging engine.

Each messaging engine must have exclusive access to the tables defined within its data store. This can be achieved, either by defining a separate data store for each messaging engine, or by partitioning a single data store using a unique schema name for each messaging engine within the data store.

> **Note:** If no data store is explicitly configured for a messaging engine, a data store will be created using a Cloudscape database. This is the case if you leave the check box for the default data store checked when you create the service integration bus.

Data store configuration

Figure 10-29 on page 423 shows the configuration panel of the data store when you add a server to a service integration bus the first time as is described in "Add the current server as a bus member" on page 411.

You can change the settings later by clicking **Buses** → **RealtyBus** → **Messaging engines** → **yourNodeName.server1-RealtyBus** → **Data store**.

If you want to use another than the default data store uncheck the **Default** box and specify a JNDI name for a DataSource that is already configured in WebSphere Application Server. For details about how to configure a DB2 DataSource for the sample application, refer to 4.7.2, "Creating the JDBC resources" on page 176.

Figure 10-29   Data store configuration when adding a server to a SIB

10.6.4 Databases, user names and schema names

Every messaging engine must have exclusive access to its own schema in a database. There are a variety of databases that are supported including Cloudscape, Network Cloudscape, DB2, Oracle, MS SQL Server, Sybase and Informix. Several of these have significantly different behaviors for schemas which are detailed below.

> **Note:** An alternative to configuring schemas and user IDs is for every messaging engine to have its own database.

> **Important:** If the data store for a messaging engine is configured to create tables then the user ID used to access the database must have sufficient authority to create and drop tables. Check this with your database administrator.
>
> If the user ID that accesses the database does not have authority to create tables, then the database administrator must create the tables before starting the messaging engine. See the WebSphere Application Server Version 6.0 Information Center section on Enabling your database administrator to create the data store tables.

Embedded Cloudscape (the default data store)

The service integration bus can create a data store for a messaging engine using a Cloudscape database that is embedded in the application server. This Cloudscape database enables you to get started quickly with the service integration bus. A JDBC data source to access the Cloudscape database is created at server scope on the server that has been added to the bus.

The default Cloudscape data store is not supported for cluster bus members. This is because multiple processes cannot access the database simultaneously which might happen during failover in a cluster.

The Cloudscape database is given the same name as the messaging engine, which is unique. Embedded Cloudscape does not require an authentication alias to be set up on the data store.

Networked Cloudscape and DB2

Both of these databases allow the same user ID to access a single database concurrently with different schema names to access different tables. This means that you can configure multiple messaging engines to access the same DB2 database, using the same user ID, but with its own unique schema name. Accessing the database with different user IDs automatically makes the actual schema accessed different.

When using Networked Cloudscape without security enabled, an authentication alias is still required on the data store, however the username and password only needs to contain any non-*null* values.

Other databases

Some databases do not make use of the specified schema name when a client connects. If you wish to have multiple messaging engines accessing the same database, then you will need to assign each messaging engine a unique user ID

that has permission to access the database. Refer to the documentation for your database for more information.

10.6.5 Security

Our sample application does not focus on security, and we set up our application without using the security features of WebSphere Application Server. This is so you can explore the new features provided by WebSphere Application Server - Express without having to deal with security setup. However, when it comes to real-world applications that use the security features of WebSphere Application Server, additional configuration steps are needed.

When security is enabled on WebSphere Application Server there are steps that must be taken for JMS applications using the service integration bus to authenticate themselves to the bus. This allows them to continue to use the messaging resources that they access.

► All JMS connection factory connections must be authenticated. This can be done in two ways:

– The connection factory can have a valid authentication alias defined on it.

– The JMS application can pass a valid username and password on the call to ConnectionFactory.createConnection(). An ID passed in this way will override any ID specified in an authentication alias on the connection factory.

► All activation specifications must have a valid authentication alias defined on them.

Note: Note that if a connection factory is looked up in the server JNDI from outside of the server environment, for example from the client container, any authentication alias defined on the connection factory will be unavailable. This prevents unauthorized use of an authenticated connection factory.

JMS Clients outside of the server can provide username and password on the call to create connection, or if the client is a J2EE client application running in the WebSphere application client environment it is possible to define an authenticated connection factory resource in the EAR file.

Any user that authenticates as a valid user to WebSphere Application Server will, by default, have full access to the bus and all destinations on it. It is possible to configure destination and even topic specific authentication requirement if you wish to do so.

Every bus has an optional inter-engine authentication alias which may be specified. If this property is left unset then it will be ignored, however, if an alias is

specified and security is enabled then the ID will be checked when each messaging engine starts up communication to other messaging engines in the bus. This provides additional security to prevent offenders pretending to be another messaging engine in the bus.

10.7 JMS in the Sal404 application

This section discusses the use of JMS in our Sal404 sample application. To demonstrate the technology, we implemented two basic functions with JMS:

► Sending a message to a JMS queue by using the JMS API triggered by an event that happens in our existing application

► Receiving a message on a queue from a context outside of our application by using a Message Driven Bean

10.7.1 Sending a message

The business functionality implemented is to send a message whenever any user data is changed. If an administrator changes any user details or the user changes their user data, then a message containing details of the change is sent to a queue. This could be used to synchronize an external user repository and keep it up-to-date with changes in our system.

Note: In a real-world environment, many organizations can have a user repository that would be contacted by our application. This chapter does not discuss the architecture issues of having redundant user data in our application context and of having to deal with updates to an existing repository. This example shows the principles of application communication over messaging. Even in real-world applications there could be valid reasons for keeping redundant user data. This could be the case, if connecting to the existing user repository would cause too much traffic on the network for each request and therefore would not perform well. While this problem usually is solved by some replication mechanism, there are systems around where it is necessary to have a custom implementation of that replication mechanism. Our messaging example could be a basic way to implement such replication.

In our example, there is no specific consumer for our user change message because our application is a standalone sample. In a real-world scenario, there could be many external applications needing interfaces to and from our application. One possible way to connect the applications and build the interfaces is to use messaging. Our sample provides a basic version of connecting the applications over a messaging bus. Figure 10-30 on page 427 shows the communication of the Sal404 application with an external application.
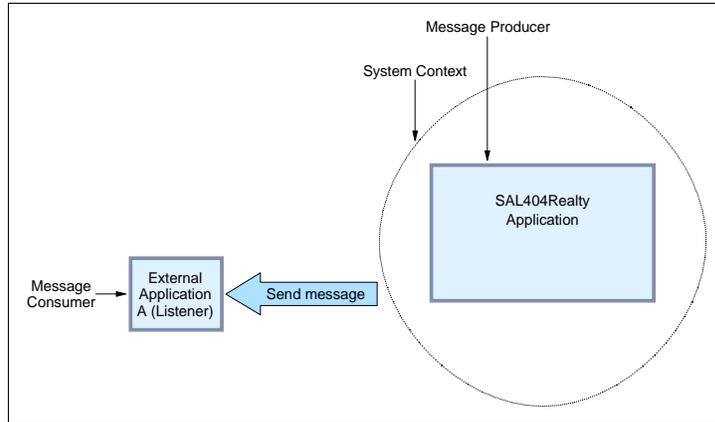
Figure 10-30   Sending a message to an external application

For demonstration purposes, we setup a message consumer that resides outside the system context of our Sal404 application just to show that messages do leave our system. See Figure 10-31.

Our current implementation of the user component uses the `UserManager` to change the user data. There is one method that updates user data in the `UserManager` called `modifyUserDetails()`. There we place the additional code for sending the messages. The code where the message is triggered is only a short piece of code that uses other components to actually send the messages.
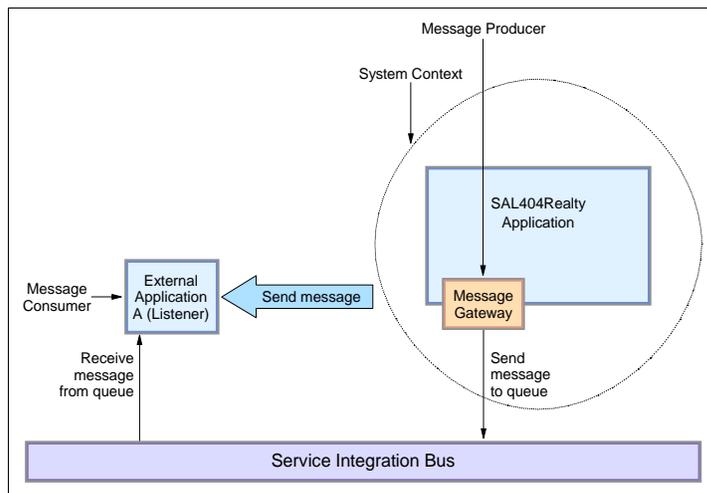
Figure 10-31   Sending a message to an external application A via the MessageGateway

The the message is sent by a component called the *MessageGateway*. This is our implementation of a gateway for JMS messages to external applications. Figure 10-31 shows the application communication with an external application via the MessageGateway. This component can be used all over the application by any other component or class that wants to send a message. Figure 10-33 on page 429 shows the user component triggering the sending of a message over the MessageGateway to a queue on the service integration bus.
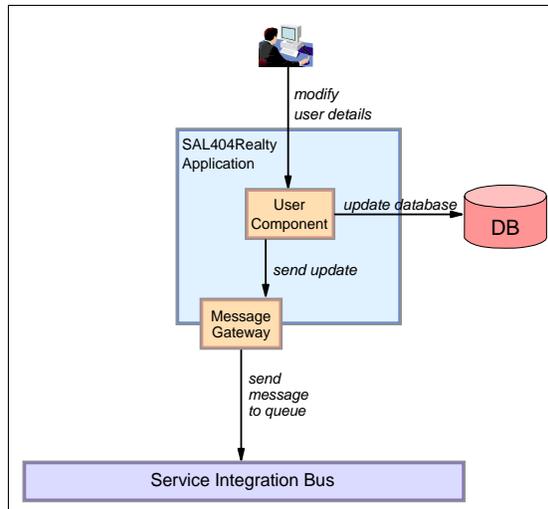
Figure 10-32   User component sending a message over the MessageGateway

10.7.2 Receiving a message

The business functionality implemented is to receive messages that let an external application create a user in our system. This might be used to synchronize our user tables with an external user repository.

The sender of the message is an external application that resides outside of our system context. That application sends a message to a queue that is connected to our Service Integration Bus. The message is then delivered to the queue endpoint, our message queue. Figure 10-34 on page 429 shows our application receiving a message from an external application.
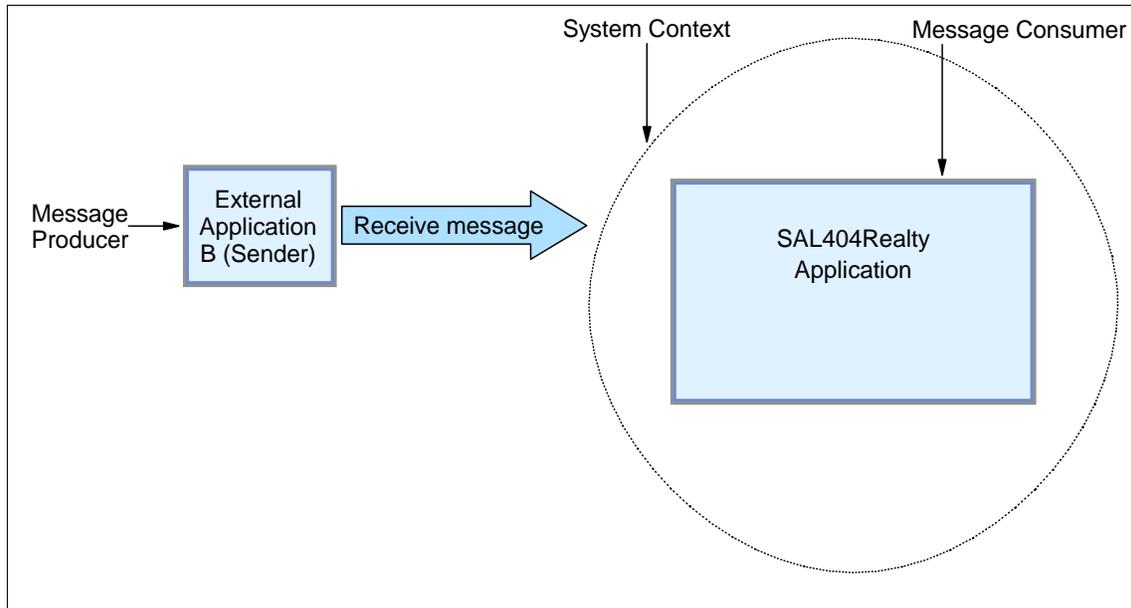
Figure 10-33   Receiving a message from an external application B

For receiving the message in our application we implement a Message Driven Bean(MDB). An instance of this bean will be created by the EJB container every time a message comes in and the resource adapter uses that bean for delivering the message. This happens when the bean is configured to listen to the endpoint (the queue). Figure 10-35 on page 430 shows the communication of the external application B over the service integration bus, using a MDB.
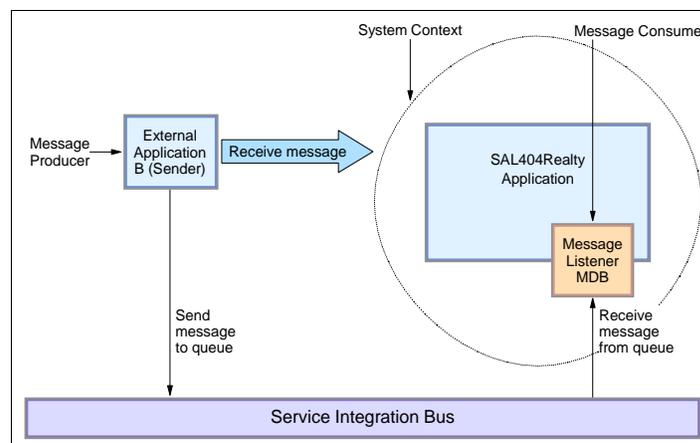
Figure 10-34   Receiving a message from some external application B with an MDB

Note: Be careful not to have multiple MDBs listening to the same queue. This is not what queues are designed for. The message would get delivered only to one of the many MDBs (by some selection criteria depending on messaging engine). Queues are meant to communicate point-to-point. If you want to have multiple beans to listen to messages then you should use the publish-subscribe mechanism and setup a topic endpoint for that communication.

Figure 10-35 shows the MDB receiving a message from the queue. The MDB parses the message and triggers the business logic in the user component to add a new user, if the message has the appropriate format. The user component is responsible for performing the business logic and updating the database.
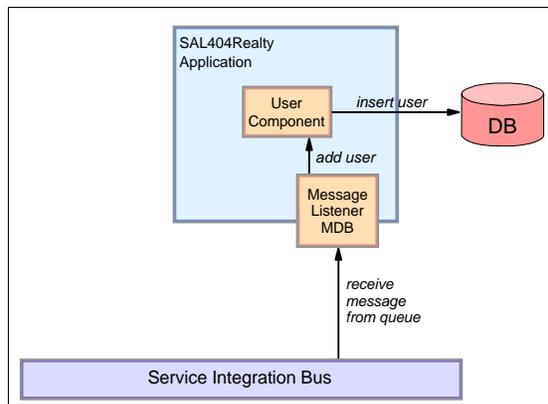
Figure 10-35   MDB adding a user via the user component

As mentioned in 10.4.8, "Message Driven Bean best practices" on page 405 no complex business processing should be done in the Message Driven Bean. To separate the business logic a stateless session facade should be used. We followed this principle by implementing the `UserFacade` as session bean. In between the MDB and the facade we have a delegate that's only purpose is to hide the JNDI lookups and to wrap the methods of the facade to expose them to any client - in our case the UserMessageListener MDB. Figure 10-36 on page 431 shows the relationship of these components.
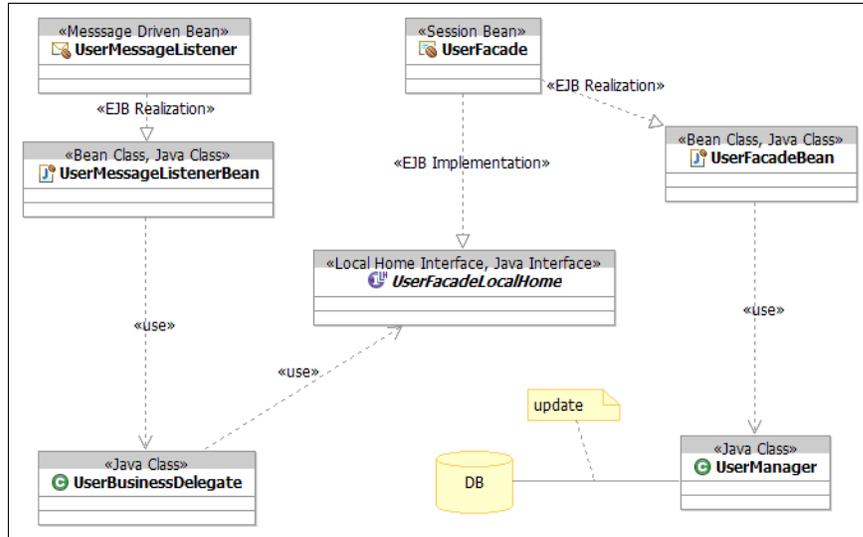
Figure 10-36   Components according to J2EE patterns

The actual JNDI lookup is done by the ServiceLocatorManager that implements the ServiceLocator pattern.

10.8 Implementation details

This section discusses some details of the implementation of the functionality described in 10.7, "JMS in the Sal404 application" on page 426.

10.8.1 Sending a message

The MessageGateway mentioned in 10.7.1, "Sending a message" on page 426 is implemented as a singleton. To minimize performance impacts, the singleton caches the JNDI lookups for the connection factory and for the queue destination.

The singleton can be accessed from every object that runs in our application. To get an instance of the singleton the code shown in Example 10-21 is used.

Example 10-21   Obtain an instance of the MessageGateway

```
MessageGateway gateway = MessageGateway.getInstance();
```

> **Note:** Be careful with singleton implementations in a J2EE environment. Usually you have to deal with multiple JVMs and multiple classloaders in a J2EE environment that affect the scope of a singleton. That means you cannot be sure to have on single instance of your singleton.
>
> The MessageGateway and the AppProperties that use the singleton pattern are very simple implementations that cache configuration properties that should be the same whatever the scope is.
>
> For a discussion about the singleton pattern in J2EE environment, see:
>
> http://ibm.com/developerworks/websphere/library/techarticles/0206_robinson/robinson.html#N10242

The MessageGateway lets you send messages of type `javax.jms.ObjectMessage`. It can easily be extended to send other JMS based message types like `javax.jms.TextMessage`. To send a message over the MessageGateway you prepare your object that you want to send to implement the `java.io.Serializable` interface. Then you call the `sendMessage()` method. This is shown in Example 10-22.

Example 10-22   Sending an ObjectMessage with the MessageGateway

```
// create some object that implements java.io.Serializable
YourObjectClass yourObject = new YourObjectClass();
// do some preparation with your object, make sure it is ready to be send
yourObject.setSomeProperty(value);
// send the object
gateway.sendMessage(yourObject);
```

Of course the receiver of the message has to be able to deal with the message. In this case, it has to be able to understand what is in the message body, that means it must know the object.

What the gateway does is just plain JMS API calls as described in 10.2, "Java Message Service API" on page 366. The method in our gateway that sends the message is shown in Example 10-23.

Example 10-23   Using the JMS API to send a message

```
        try {
                // get the connection from the factory
                Connection connection = connectionFactory.createConnection();
                // create a transacted session
                Session session = connection.createSession(true,
                        QueueSession.AUTO_ACKNOWLEDGE);
                // create the message producer for the queue
```

```
                    MessageProducer messageProducer = session.createProducer(queue);
                    // create the object message
                    ObjectMessage message = session.createObjectMessage();


                    message.setObject(object);

                    // send message
                    messageProducer.send(message);
                    // commit transaction
                    session.commit();
                    // close producer
                    messageProducer.close();
                    // close connection
                    connection.close();
                } catch (JMSException e) {
                    logger.error("sendMessage() error sending JMS message", e);
                }
```

For testing purposes we implemented a servlet that sends an object with user details through the gateway. To test this, just invoke the following URL in our sample application:

`http://localhost:9080/SAL404Realty/TestMessageGateway`

The servlet response is either `OK` or `ERROR` depending on whether it could send a message or not.

You will also get a message sent to the outgoing queue when you change the user details from the application itself. To do that, follow these steps:

1. Start the sample application in a Web browser using the URL:

   `http://localhost:9080/SAL404Realty/`

2. Enter login details: User ID=`cust1`, Password=`password`

3. Click **Log in** from the navigation menu.

4. Click the link **Modify your details here.**

5. Change the details of the user as shown in Figure 10-37 on page 434

6. Click **Submit.**

Figure 10-37   Modify user details dialog

After submitting the form, a message with the user details is send to the JMS queue.

To check that a message has been sent to the queue on the bus, go to the WebSphere Administrative Console and look for the message. Remember that the message remains in the bus until it is fetched by a message consumer or until it expires. This is the case when the message is sent to a queue (see 10.2.10, "JMS message consumers" on page 374).

To see the message in the WebSphere Administrative Console, follow these steps:

1. Open your WebSphere Administrative Console using

   `http://localhost:9060/admin`

2. Click **Service integration** → **Buses** → **RealtyBus.**

3. Click **Destinations** (under Additional Properties).

4. Click **RealtyBusOutQueue** → **Queue points**.

5. Now click the queue point
   RealtyBusOutQueue@yourNodeName.server1-RealtyBus.

6. Click the **Runtime** tab.

7. You should see a panel as shown in Figure 10-38.
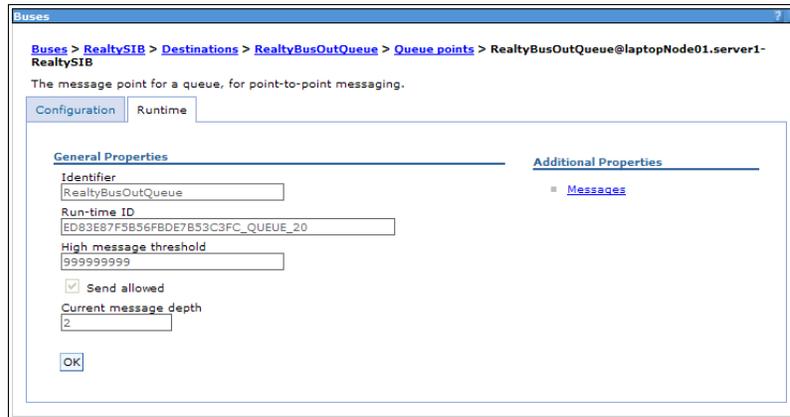
Figure 10-38   Message endpoint runtime panel

8. If you click **Messages** (under Additional Properties) you see a list of the messages that have arrived at the queue and have not yet been consumed or expired.

9. Figure 10-39 shows the list of messages.

Figure 10-39   List of messages for a queue point

10. if you click a message identifier you see the attributes of the message as shown in Figure 10-40 on page 436.
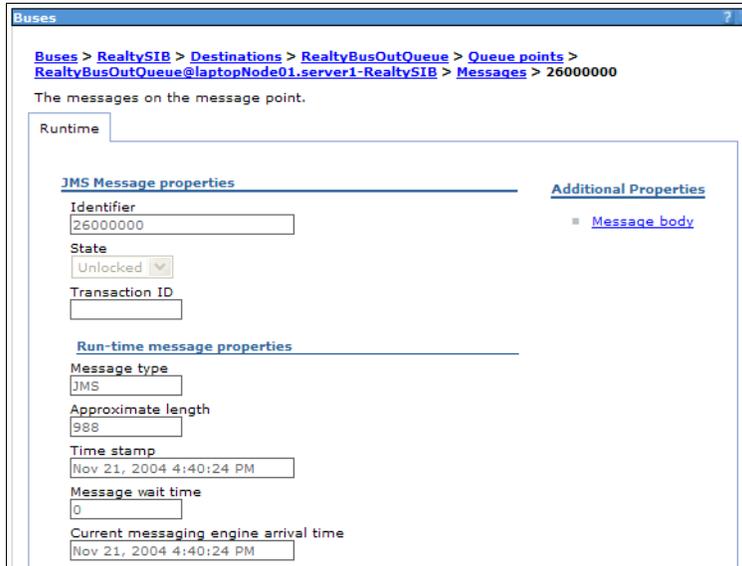
Figure 10-40 Properties of one single message (upper screen area)

11.Now click **Message body** and you see the body of the message as shown in Figure 10-41.
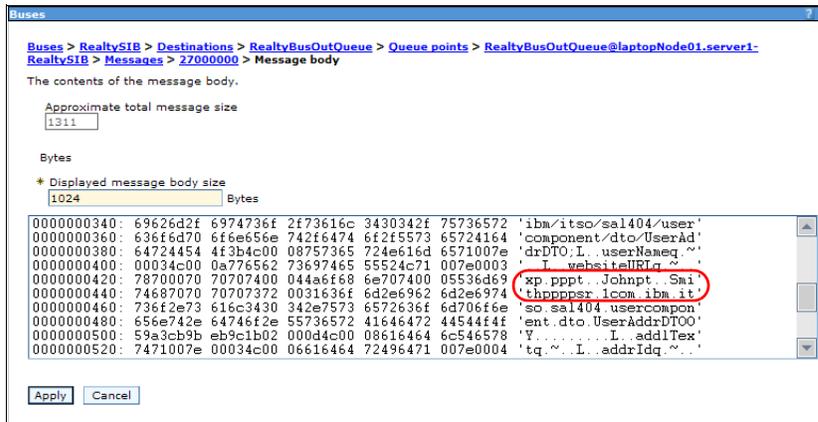
Figure 10-41 Message body with object details

Because the body of the message was a serialized object, it is not readable, but when you look at the details you can find the values for `firstName` and `lastName` (`John Smith`) of our `UserDetailsDTO` object as highlighted in Figure 10-41.

You should also a see message when you modify the user details in the Sal404 application.

10.8.2 Receiving a message

We registered an `UserMessageListener` MDB to listen to messages on the `RealtyIncomingQueue` destination. Because a Message Driven Bean is an EJB you have to use Rational Application Developer if you want to use the creation wizards for EJBs. You also could code the EJB manually but then you would need to produce the deployment descriptors and bindings manually.

Create an EJB project

To create an EJB project, follow these steps in Rational Application Developer:

1. Choose **File** → **New** → **EJB Project** as shown in Figure 10-42.
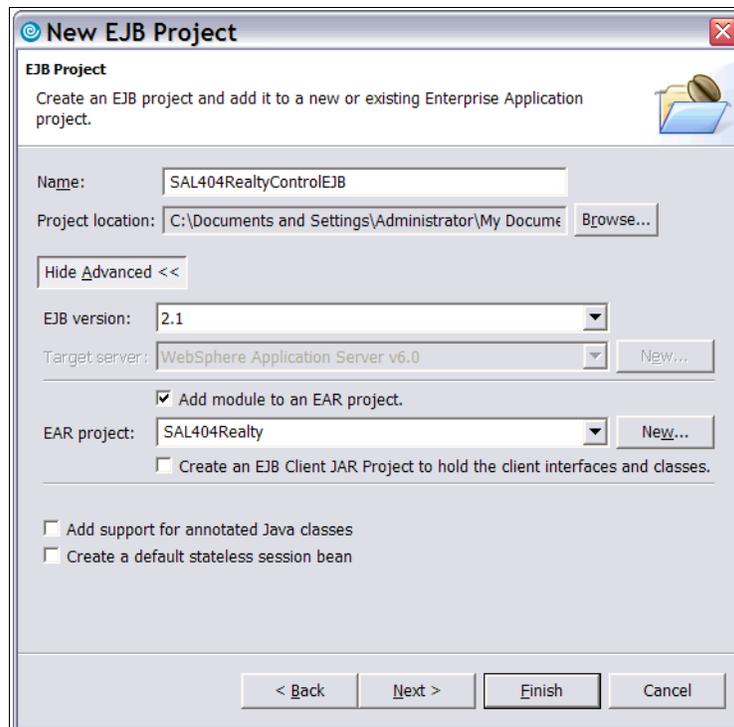
Figure 10-42   Create an EJB project

2. Make sure to specify the latest EJB specification (2.1).
3. Specify `SAL404Realty` as the EAR project.
4. Uncheck **Create an EJB Client JAR Project.**
5. Click **Finish.**

Rational Application Developer V6 lets you to specify folders for source code for the EJBs and folders for the deploy code where all the generated classes go. It is a good idea to create two separate folders, especially when you want to regenerate the deployed code and not get it confused with the code that you write yourself. To do this, follow these steps:

1. Right-click the EJB project you just created and click **Properties.**

2. Go to **Java Build Path** → **Source**.

3. Click **Add Folder...**

4. Specify a new source folder under the project folder as shown in Figure 10-43.
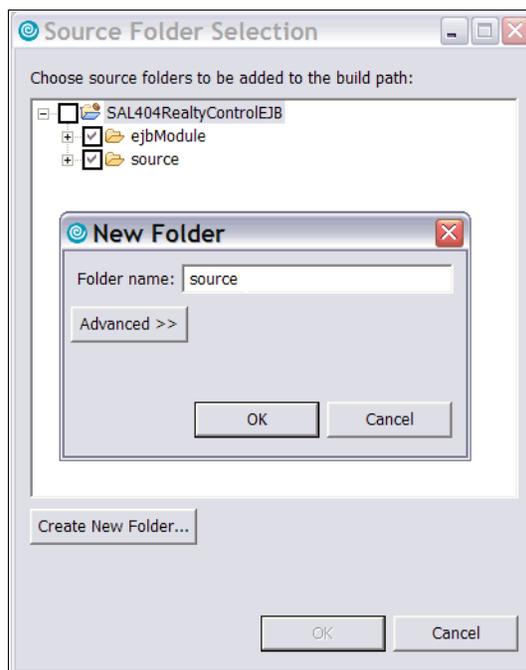
Figure 10-43   Add a new folder for java sources

Now you have the source folders source and ejbModule. Use source for your own code. It can be specified when you create an EJB with the wizard. Using the ejbModule folder for the deployed code is the default setting of for a EJB project. You can verify this with the following steps:

1. Right-click the EJB project you just created and click **Properties.**

2. Go to **EJB Deployment** and make sure that the **ejbModule** folder is selected as shown in Figure 10-44 on page 439.
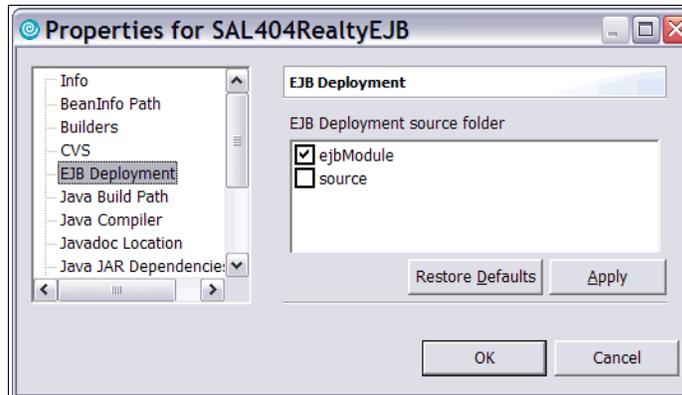
Figure 10-44   Source folder for EJB deployment code

Note: With these settings you can now remove any code that you find in the ejbModule folder. It will be redeployed automatically or when you right-click the EJB Project and chose **deploy** from the drop-down menu. The generated code by default will not be added to the CVS. T

Create a Message Driven Bean

To create a MDB with Rational Application Developer, follow these steps:

1. Switch to the J2EE perspective.

2. Right-click the **Deployment-Descriptor** of your EJB project in the Project Explorer view.

3. Click **New** → **Enterprise Bean** to open the wizard for EJBs.

4. Enter the following values as shown in Figure 10-45 on page 440:

   - Enter the name `UserMessageListener.`
   - Specify a package name as `com.ibm.itso.sal404.user.`
   - Choose the **source** folder from the drop-down menu.
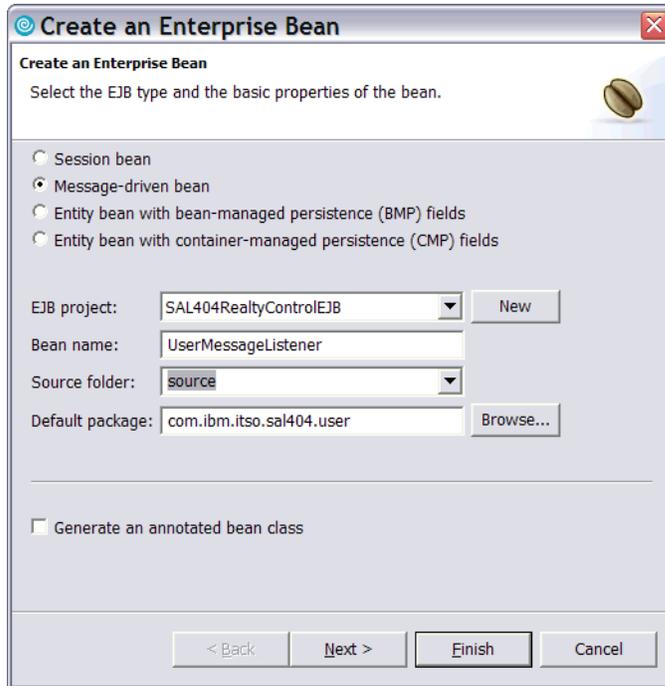
Figure 10-45   Create a new message-driven EJB

5. Leave the **javax.jms.Listener** selected as shown in Figure 10-46.

Figure 10-46   Select the JMS listener type

6. Leave all the default values in the next screen as shown in Figure 10-47. Here you could specify a message selector, but you can also change these settings in the deployment descriptor later.

Figure 10-47   Setting details for the MDB

We only changed two methods in the created MDB. We used the ejbCreate method of the MDB to create a UserBusinessDelegate object that wraps the UserFacade. This is shown in Example 10-24.

Example 10-24   ejbCreate() method of the MDB

```
public void ejbCreate() {
    ...
    // just create a reference to the delegate
    userBusinessDelegate = new UserBusinessDelegate();
    ...
}
```

The second method we changed is the `onMessage()` method. It does the filtering of the message and invokes the business logic on the delegate. The delegate invokes the session facade. The actual business logic is done by the `UserManager`. This might look like a lot of indirection but the facade and the

delegate are thin wrappers and this is a common way to separate the layers according to J2EE best practices.

The onMessage() method look like Example 10-25. First the message is filtered for an UserDetailsDTO object. This is the serializable object that is sent to the JMS message destination. When such an object was found the business logic to create a new user is invoked.

Example 10-25   The onMessage() method of the MDB

```
public void onMessage(javax.jms.Message msg) {
    ...
    UserDetailsDTO userDetailsDTO = filterforUserDetailsDTO(msg);
    if (userDetailsDTO != null) {
        logger.info("filterMessage() start processing the message");
        userBusinessDelegate.createUser(userDetailsDTO);
    }
    ...
}
```

Be careful with the implementation of the onMessage() method when using queues. Because messages do not die until they are delivered or timed out, the MDB could be created and the onMessage() method be executed forever, if an exception occurs in the method. This can produce a 100% CPU load because the application server never continually creates MDBs and tries to deliver the message.

Tip: If you stop and restart the server in an attempt to clear the 100% CPU load, the same problem can happen again. To get rid of the repeated redelivering of the message, go to the WebSphere Administrative Console and stop the application with the MDB. After having stopped the application go to the message queue point in the bus (see Figure 10-39 on page 435), delete the message that is causing the trouble, then restart the application server.

The redelivery of the message has a time limit if you specify a time to live for the messages in the queue. To set the time to live, go choose **Default messaging provider** → **JMS queue** → **RealtyIncomingQueue** and enter a timeout in milliseconds (for example 5000 for a development environment) as shown in Figure 10-48 on page 443. Save the configuration and restart the server to accept the settings.

Tip: Always use the time to live setting when developing Message Driven Beans in order not to run into trouble with an overloaded CPU.
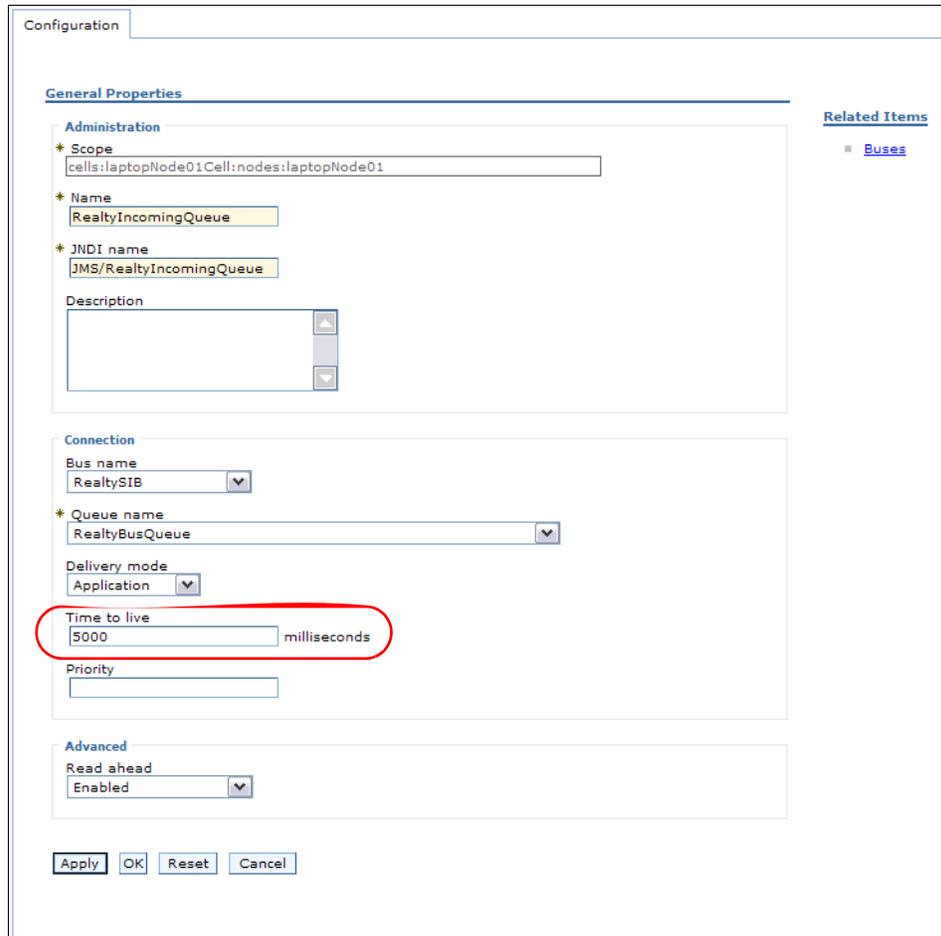
Figure 10-48   Specify a time to live for messages in a queue

Test the Message Driven Bean

To test the Message Driven Bean, we created a simple Web page that runs in another enterprise application. The application was setup to demonstrate that we can send the message from an external application that is outside of the system context of our Sal404 application even if it is actually running in the same instance of WebSphere Application Server. To use the test application, enter the following URL:

```
http://localhost:9080/AgencyWeb/faces/employee/add_employee.jsp
```

Enter some user values in the screen shown in Figure 10-49 on page 444 and click **InsertEmployee**.

The JMS Test Client Application then creates a message on the incoming queue for SAL404Realty. The MDB `UserMessageListener` will receive that message and register the new user in the system using the `UserManager`.

Figure 10-49   Add a new employee

You cannot see the message that was created in the WebSphere Administrative Console because it will be consumed immediately by the MDB and not be waiting on the queue.

To check if the employee was added to the system, login as administrator into the Sal404 application, go to the user administration, then check for the new user.

10.9 References and resources

The following references provide further information about using JMS with WebSphere Application Server - Express:

► WebSphere Application Server library

   http://www.ibm.com/software/webservers/appserv/infocenter.html

► Java Message Service documentation

   http://java.sun.com/products/jms

- ► Java 2 Platform, Enterprise Edition documentation

  `http://java.sun.com/j2ee/index.jsp`

- ► J2EE Connector Architecture

  `http://java.sun.com/j2ee/connector/index.jsp`

- ► WebSphere MQ Using Java

  `http://ibm.com/software/integration/mqfamily/library/manualsa/manuals/cross`
  `latest.html`

- ► Enterprise Messaging Using JMS and WebSphere (Kareem Yusuf), Prentice Hall, ISBN: 0-13-146863-4

- ► Java Message Service (Monson-Haefel, Chappell), O'Reilly, ISBN: 0-596-00068-5

- ► Professional JMS (Grant, Kovacs, et al), Wrox Press Inc., ISBN: 1861004931

- ► Enterprise JavaBeans, Fourth Edition (Monson-Haefel, Burke, Labourey), O'Reilly, ISBN: 0-596-00530-X

- ► EJB Design Patterns (Marinescu), Wiley, ISBN: 0471208310

11

Struts

In this chapter we review the use of Struts and provide an overview of how to implement Struts applications using the Rational Software Development Platform.

447

11.1 Struts overview

The *Jakarta Struts project*, an open-source project sponsored by the Apache Software Foundation, is a server-side Java implementation of the model-view-controller (MVC) design pattern. The Struts project was designed with the intention of providing an open-source framework for creating Web applications that easily separate the presentation layer and allow it to be abstracted from the transaction and data layers. Struts is useful when you build a Web application, but in order to understand the benefits of the framework it is important to understand MVC.

The Struts framework control layer uses technologies such as servlets, JavaBeans, and XML. The view layer is implemented using JSPs. The Struts architecture encourages the implementation of the concepts of the model-view-controller (MVC) architecture pattern. By using Struts you can get a clean separation between the presentation and business logic layers of your application.

Struts also speeds up Web application development by providing an extensive JSP tag library, parsing and validation of user input, error handling, and internationalization support.

The focus of this chapter is on the Rational Application Developer tooling used to develop Struts-based Web applications. Although we do introduce some basic concepts of the Struts framework, we recommend that you refer to the following sites:

► Apache Struts home page:

   http://struts.apache.org/

► Apache Struts User Guide:

   http://struts.apache.org/userGuide/introduction.html

Note: The Rational Software Development Platform includes support for Struts Version 1.1. As we write this redbook, the latest version of the Struts framework is V1.2.4.

11.2 MVC design pattern

Figure 11-1 on page 449 shows an overview of the model-view-controller pattern as it is used in the Struts framework.
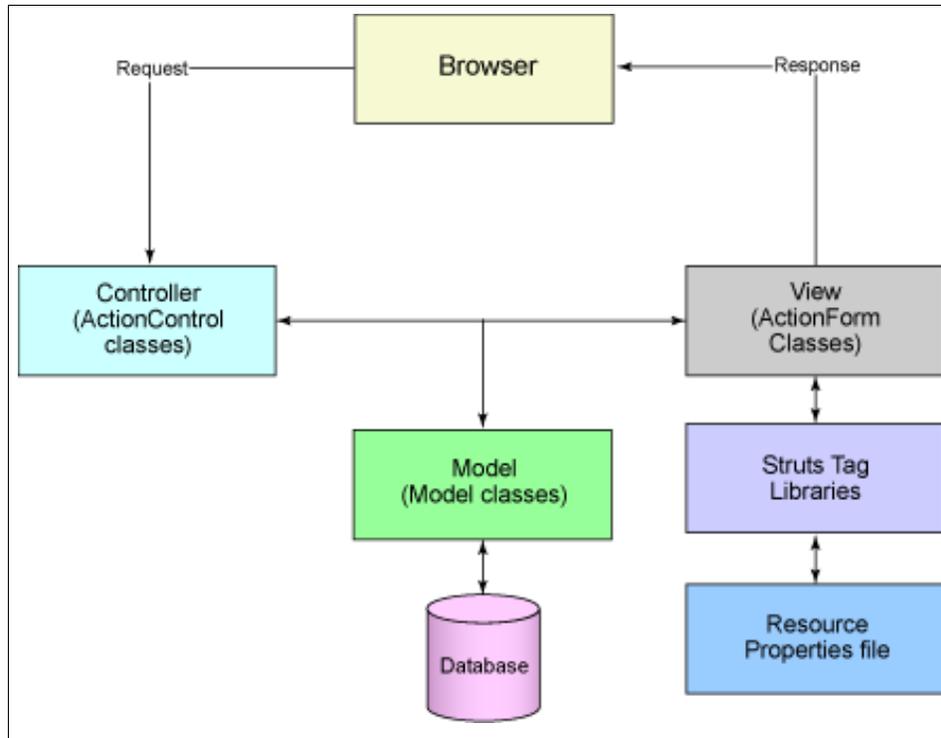
Figure 11-1   The MVC pattern and Struts

The MVC design pattern consists of three components: model, view and controllers.

Model
The model contains the core of the application's functionality and encapsulates the state of the application. Sometimes the only functionality it contains is state. It knows nothing about the view or controller.

View
The view provides the presentation of the model. It is the look of the application. The view can access the model getters, but it has no knowledge of the setters. In addition, it knows nothing about the controller. The view should be notified when changes to the model occur.

Controller
The controller reacts to the user input. It creates and sets the model.

The benefits of using MVC include:

► Reliability

The presentation and transaction layers have clear separation, enabling you to change the look and feel of an application without recompiling Model or Controller code.

► High reuse and adaptability

MVC lets you use multiple types of views, all accessing the same server-side code. This includes anything from Web browsers (HTTP) to wireless browsers.

► Very low development and life cycle costs

MVC makes it possible to have lower-level programmers develop and maintain the user interfaces.

► Rapid deployment

Development time can be significantly reduced because Controller programmers (Java developers) focus solely on transactions, and View programmers (HTML and JSP developers) focus solely on presentation.

► Maintainability

The separation of presentation and business logic also makes it easier to maintain and modify a Struts-based Web application.

11.3 Model-view-controller (MVC) pattern with Struts

Figure 11-2 on page 451 depicts the Struts components in relation to the MVC pattern.

► Model

Struts does not provide model classes. The business logic must be provided by the Web application developer as JavaBeans or EJBs.

► View

Struts provides action forms to create form beans that are used to pass data between the controller and view. In addition, Struts provides custom JSP tag libraries that assist developers in creating interactive form-based applications using JSPs. Application resource files hold text constants and error message, translated for each language, that are used in JSPs.

► Controller

Struts provides an action servlet (controller servlet) that populates action forms from JSP input fields and then calls an action class where the developer provides the logic to interface with the model.
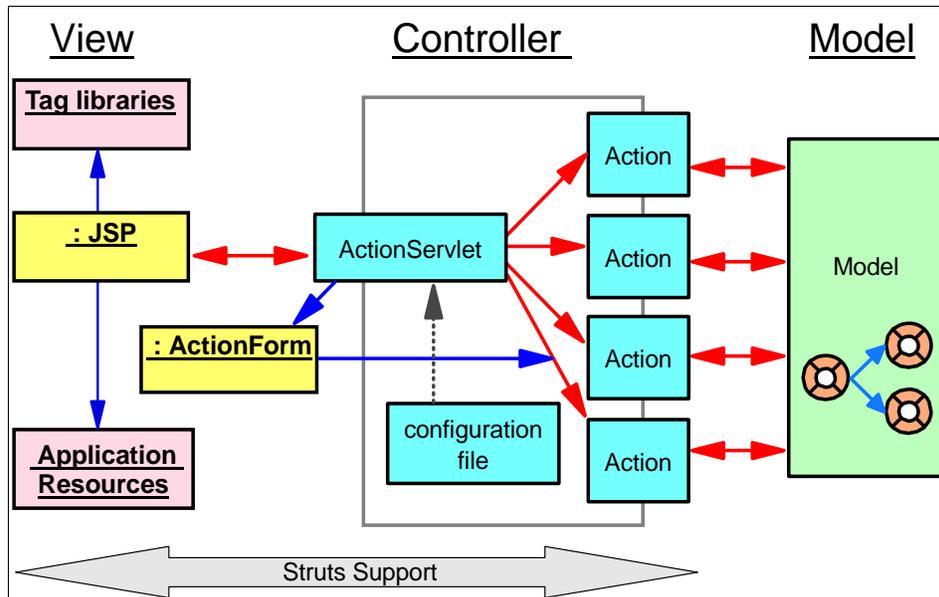
Figure 11-2   Struts components in the MVC architecture

A typical Struts Web application is composed of the following components:

► A single servlet (extending `org.apache.struts.action.ActionServlet`) implements the primary function of mapping a request URI to an action class. Before calling the action class, it populates the form bean associated to the action with the fields from the input JSP. If specified, the action servlet also requests the form bean to validate the data. It then calls the action class to carry out the requested function. If form bean validation fails, control is returned to the input JSP so the user can correct the data. The action servlet is configured by an XML configuration file that specifies the environment and the relationship between the participating components.

► Multiple JSPs that provide the end-user view. Struts includes an extensive tag library to make JSP coding easier. The JSPs display the information prepared by the actions and requests new information from the user.

► Multiple action classes (extending any one of the Struts action classes like `org.apache.struts.action.Action`) that interfaces with the model. When an action has performed its processing, it returns an action forward object which determines the view that should be called to display the response. The action class prepares the information required to display the response, usually as a form bean, and makes it available to the JSP. Usually the same form bean that was used to pass information to the action is used also for the response, but it is also common to have special view beans tailored for displaying the data. An action forward has properties for its name, address (URL) and a flag

specifying if a forward or redirect call should be made. The address to an action forward is usually hard-coded in the action servlet configuration file, but can also be generated dynamically by the action itself.

► Multiple action forms (extending any one of the Struts Action Form classes like `org.apache.struts.action.ActionForm`) to help facilitate transfer form data from JSPs. The action forms are generic Javabeans with getters and setters for the input fields available on the JSPs. Usually there is one form bean for each Web page, but you can also use more coarse-grained form beans holding the properties available on multiple Web pages. This fits very well for wizard-style Web pages. If data validation is requested (a configurable option) the form bean is not passed to the action until it has successfully validated the data. Therefore the form beans can act as a sort of firewall between the JSPs and the actions, only letting valid data into the system.

► One application resource file for each language supported by the application holds text constants and error messages and makes internationalization easy.

Figure 11-3 shows the basic flow of information for an interaction in a Struts Web application.
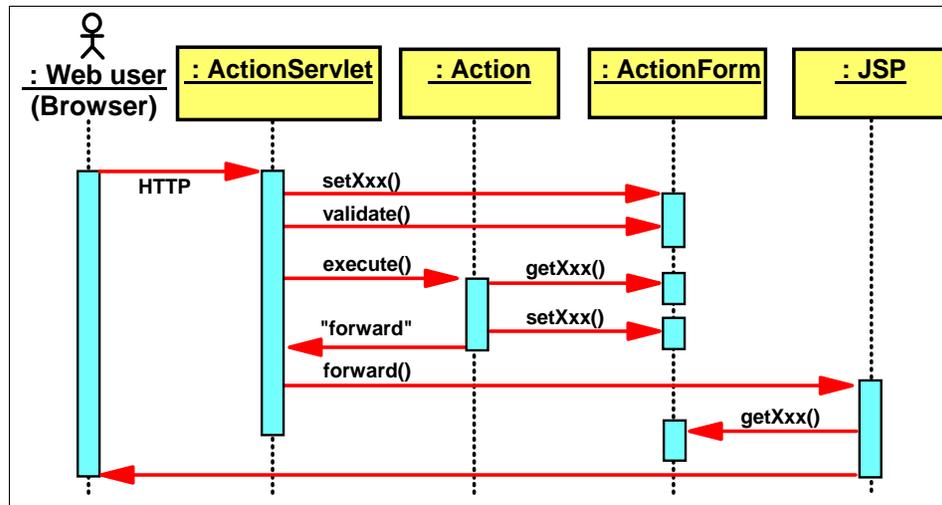
Figure 11-3   Struts request sequence

A request from a Web browser reaches the Struts `ActionServlet`. If the action that will handle the request has a form bean associated with it, Struts creates the form bean and populates it with the data from the input form. It then calls the validate method of the form bean. If validation fails, the user is returned to the input page to correct the input. If validation succeeds, Struts calls the action's execute method. The action retrieves the data from the form bean and performs

the appropriate logic. Actions often call session EJBs to perform the business logic. When done, the action either creates a new form bean (or other appropriate view bean) or reuses the existing one, populates it with new data, and stores it in the request (or session) scope. It then returns a forward object to the Struts action servlet, which forwards to the appropriate output JSP. The JSP uses the data in the form bean to render the result.

11.4 Rational Application Developer support for Struts

Rational Application Developer provides the following support for Struts-based Web applications:

▶ A Web project can be configured for Struts. This adds the Struts runtime (and dependent JARs), tag libraries, and action servlet to the project, and creates skeleton Struts configuration and application resources files. Rational Application Developer provides support for Struts 1.1, selectable when setting up the project. This field is selectable as at the time of this writing support for Struts 1.2.x is being added to Rational Application Developer.

▶ A set of Struts Component Wizards define action form classes, action classes with action forwarding information, and JSP skeletons with the tag libraries included.

▶ The Struts Configuration Editor maintains the control information for the action servlet.

▶ A graphical design tool edits a graphical view of the Web application from which components (forms, actions, JSPs) can be created using the wizards. This graphical view is called a *Web diagram*. The *Web diagram editor* provides top-down development (developing a Struts application from scratch), bottom-up development (that is, you can easily diagram an existing Struts application that you may have imported) and meet-in-the-middle development (that is, enhancing or modifying an existing diagrammed Struts application).

▶ The Project Explorer view provides a hierarchical (tree-like) view of the application. This view shows the Struts artifacts, such as Actions, Formbeans, Global Forwards, Global Exceptions and Web pages, you can expand the artifacts to see their attributes. For example, an Action can be expanded to see the formbeans, forwards and local exceptions associated with the selected Action. This is useful for understanding specific execution paths of your application. The *Project Explorer view* is available in the Web perspective.

▶ The JSP Page Designer supports rendering the Struts tags, making it possible to properly view Web pages that use the Struts JSP tags. This

support is customizable using Rational Application Developer's Preferences settings.

► Validators validate the Struts XML configuration file and the JSP tags used in the JSP pages.

11.5 Why we use Struts

The Struts project was designed with the intention of providing an open-source framework for creating Web applications that easily separate the presentation layer and allow it to be abstracted from the transaction and data layers. Since its inception, Struts has received enough developer support, and is quickly becoming a dominant factor in the open-source community.

Application development and maintenance with struts are much easier if the different components of a Web application have clear and distinct responsibilities. The Struts framework was created to make it easier for developers to build J2EE Web applications. We should have different team for each layer of the project then developers to concentrate on building the business application rather than on the infrastructure.

Struts framework has gained considerable attention because Struts combines Java Servlets, JavaServer Pages, custom tags, and message resources into a unified infrastructure, and saves the developer the time of coding an entire MVC model, a considerable task indeed.

The Sal301 sample application developed for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 was implemented as a Struts application. When the Sal301 application was written JavaServer Faces technology was not available so given that we made the decision to use the MVC pattern using Struts was an easy choice. If we were developing the sample application for our current redbook from scratch the using JavaServer Faces would be an equally valid choice. Chapter 7, "JavaServer Faces" on page 239 not only discusses JavaServer Faces and compares it with Struts, but also shows how to use JavaServer Faces to build the similar functionality as that in our Struts application. We do not attempt to convert all of our existing sample application from Struts to JavaServer Faces. Instead we concentrate on using some more features of Struts including the validation framework and Struts module support. We also use Struts to implement the new application requirements of the bidding component. Validation and module support are documented in the following sections of this chapter. For information about Struts best practices see:

http://ibm.com/developerworks/web/library/wa-struts/index.html

11.6 Struts validator framework

The Struts framework allows input validation to occur inside the ActionForm. To perform validation on data passed to a Struts application, developers must code special validation logic inside each ActionForm class. Although this approach works, it has some serious limitations. The Validator framework allows you to declaratively configure validation routines for a Struts application without programming special validation logic. The Validator has become so popular and widely used by Struts developers that it has been added to the list of Jakarta projects

To use Validator framework, we need to do a number of simple actions:

1. Define fields to validate.
2. Add the plug-in.
3. Add validator_rules.xml.
4. Configure error messages.
5. Configure validation rules.
6. Modify your old validations.

11.7 Struts validation sample

The sample application for Sal301 used Struts validations on the server side. For this project our Sal404 application sample uses validations on the client side.

One reason for using client-side validations is that you will have less traffic on the network.

The steps to implement validation in our Sal404 sample are:

1. Define the fields for your validations.

   For our project, we focused on the Register Form. The fields we decided to validate are shown in Table 11-1.

Table 11-1   Fields to validate in the register form

Field	Validation	Field	Field
User name*	Character 20	Address name*	char <20
password*	Char >6 <20	Street*	char <100
confirm password*	Equal to password	Unit	char < 10
Title	Char 5	Building	char < 10
First Name*	Char < 40	PO-Box	char < 10

Field	Validation	Field	Field
Last Name*	Char < 40	City	char <
Email*	email valid	Post/Zip Code*	decimal < 6 positions
Web site		Country*	char = 4
Phone Number	^\(?(\d{3})\)?[-|]?(\d{3})[-|]?(\d{4})	State	char < 40
Aditional Info	>=0 Char < 100		

In Table 11-1 on page 455 all the fields marked with * are required. Figure 11-4 shows the fields as laid out on the user registration form in our sample application.

Figure 11-4   Fields on the user registration form

2. The next step is to add the Validator plugin to the Struts config file. Using the Project Explorer view, navigate to the struts-config.xml file found in **SAL404R Web Project** → **Struts** → **default** → **struts-config.xml.**

3. Open the Struts configuration editor by double-clicking the **struts-config.xml** file.

4. Click the **Plug-ins** tab. Figure 11-5 shows the Struts configuration editor open on the plug-ins tab.
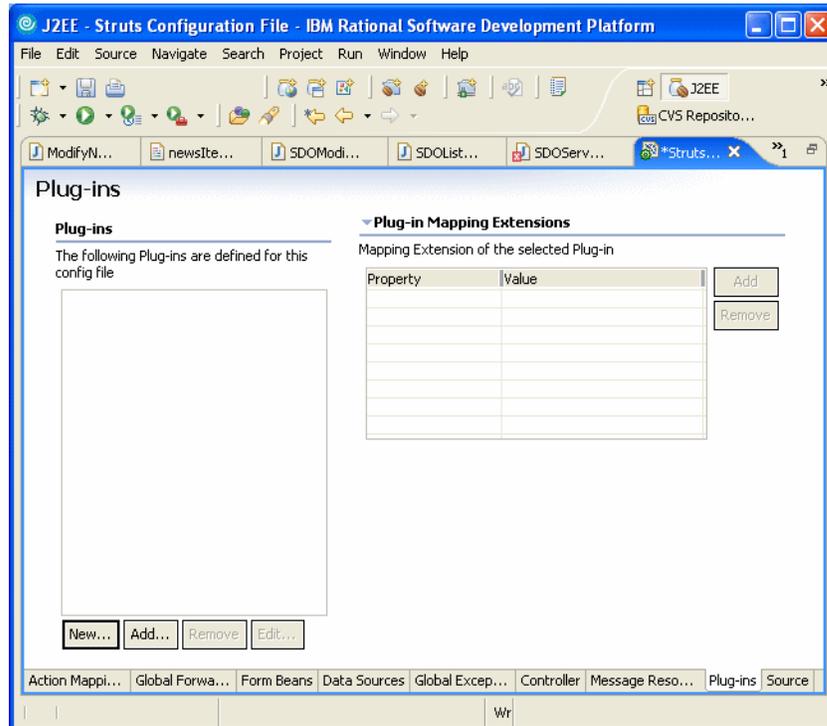
Figure 11-5   Struts plug-ins

5. Click **Add** and choose the validator plugin as shown in Figure 11-6 on page 458.
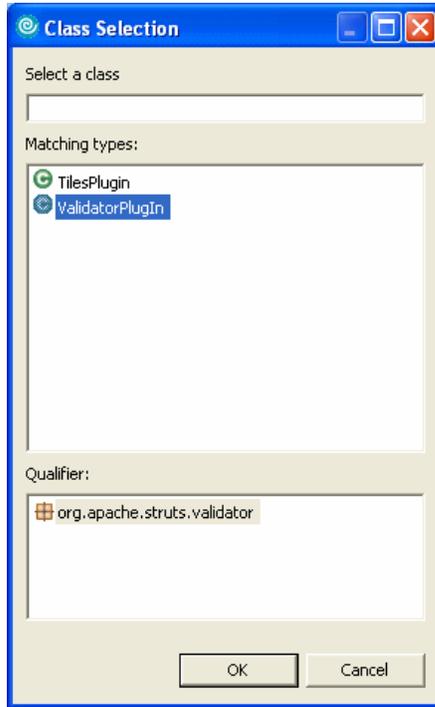
Figure 11-6   Add ValidatorPlugin

6. Click **OK.**

7. Under the heading Plug-in Mapping Extensions click **Add** to add a property
    called `pathnames` with a value of `/WEB-INF/validator-rules.xml`,
    `/WEB-INF/validation.xml.` See Figure 11-7 on page 459.
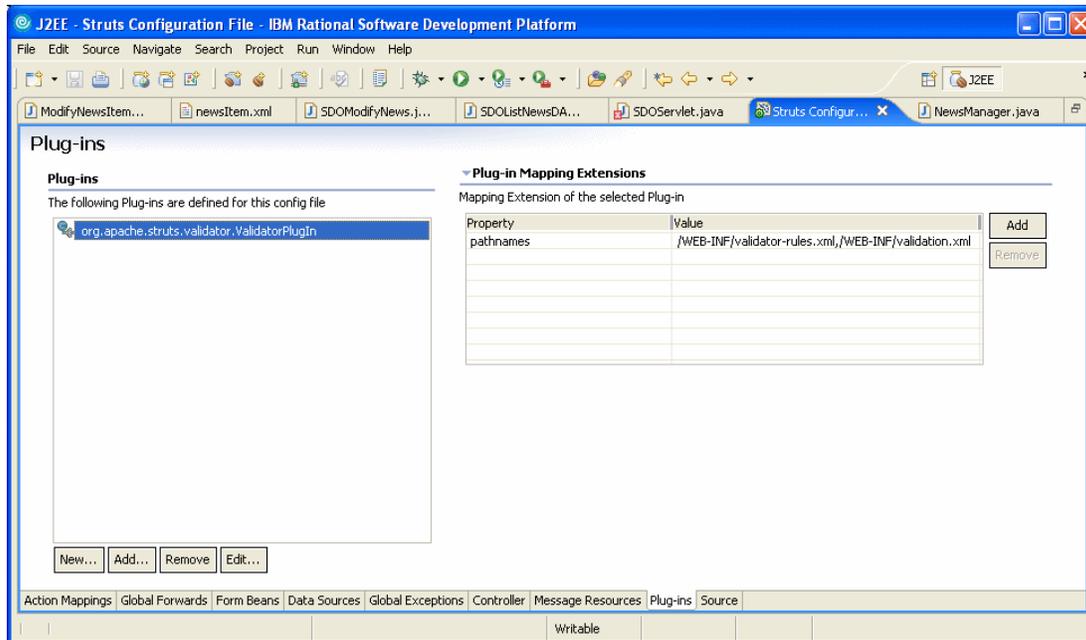
Figure 11-7   Plugin mapping extensions

8. Save the changes to the Struts config file.

> **Tip:** You can add the plug-in manually to the struts-config.xml by entering the following text:
>
> ```
> <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
> <set-property property="pathnames" value="/WEB-INF/validator-rules.xml,
> /WEB-INF/validation.xml"/> </plug-in>
> ```

9. Add the validator-rules.xml to your Web project. Right-click the **WEB-INF** folder in the Web project and choose **Import**.

10. Select to import from the **File system** and click **Next**.

11. Click **Browse** and select the **validator-rules.xml** example that is supplied with Rational Software Development Platform. This can be found in the directory

    ```
    <RSDP-install>/rwd/eclipse/plugins/com.ibm.etools.resources.common_6.0.0/St
    ruts/Struts_1.1/validator-rules.xml
    ```

12. Click **Finish** to import the file. See Figure 11-8 on page 460.
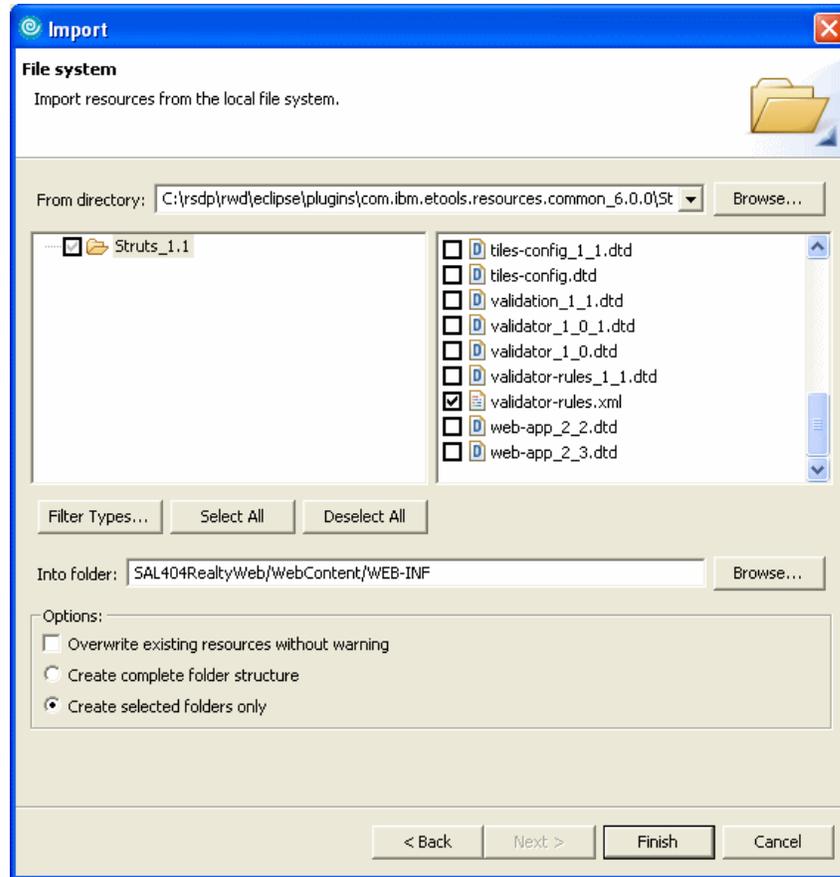
Figure 11-8   Import validator-rules.xml

13. Validator-rules.xml contains a basic set of default validations. Table 11-2 shows these validations.

Table 11-2   Validations in validator-rules.xml

required	validwhen	requiredif	maxlength
mask	byte	minlength	integer
long	float	short	date
range	floatRange	double	creditCard
email	url	doubleRange	intRange

For more details about this see the following link:

http://struts.apache.org/userGuide/dev_validator.html

14. Next we configure error messages for the default validations.

The default error messages are included by adding the lines shown in Example 11-1 to the ApplicationsResources.properties files. These are used by the validator by default and must be included even if they are not referenced.

Example 11-1   Default Validation rules

```
errors.required={0} is required.
errors.minlength={0} can not be less than {1} characters.
errors.maxlength={0} can not be greater than {1} characters.
errors.invalid={0} is invalid.
errors.byte={0} must be a byte.
errors.short={0} must be a short.
errors.integer={0} must be an integer.
errors.long={0} must be a long.
errors.float={0} must be a float.
errors.double={0} must be a double.
errors.date={0} is not a date.
errors.range={0} is not in the range {1} through {2}.
errors.creditcard={0} is an invalid credit card number.
errors.email={0} is an invalid e-mail address uuuu.
```

15. To configure messages for each new error that you want validate, you have to add a message in the ApplicationsResources.properties file:

Open the file **SAL404RealtyWeb** → **Java Resources** → **JavaSource** → **sal40realtyweb.resources** → **ApplicationResources.Properties** and add the messages shown in Figure 11-9 on page 462.
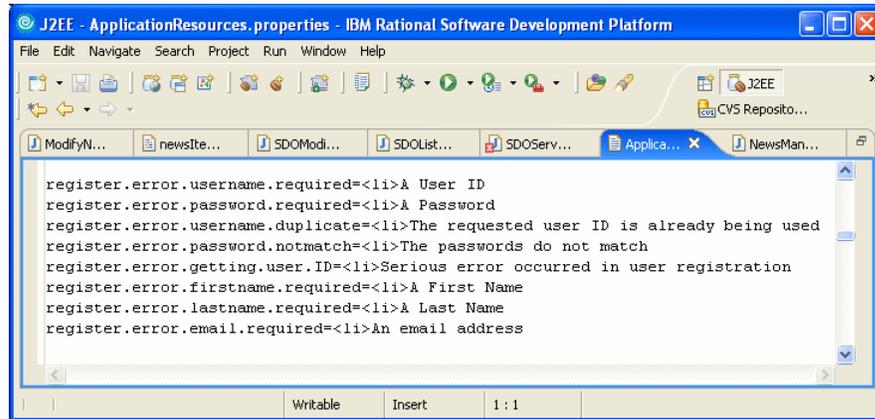
Figure 11-9   Register error messages

16. To add an error message, enter the name for the message and the text to display. For example:

`register.error.username.required=User Name is required`

Where:

`register` is the name of the form

`error` is the type of message

`username` is the field validated

`required` is the validation

17. Next configure your validations on the Validation.xml file.In the WEB-INF folder of your Web project choose **File → New → XML → XML file.**

18. Click **Next** and select **Create a file from DTD file.**

19. Click **Next** and enter the new file name as `validation.xml.`

20. Click **Next** and chose **Select XML Catalog entry**.

21. Select **-//Apache Software Foundation//DTD Struts Validation Configuration 1.1//EN** from the list of available DTDs. See Figure 11-10 on page 463.
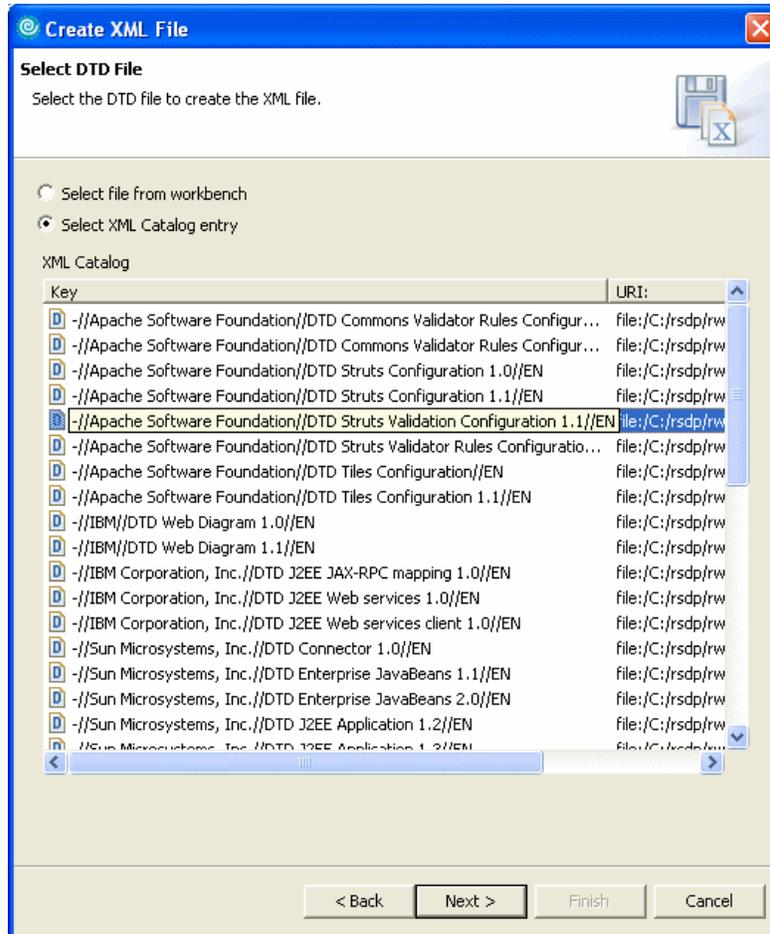
Figure 11-10   Select Struts validation DTD

22.Click **Finish**.

23.Add a validation rule to the validation.iml file for each of the fields listed in Table 11-1 on page 455.

For each field to validate, add code similar to that shown in Example 11-2:

Example 11-2   Validation example for register fields

```
<form name="registerFormBean">
<field property="userName" depends="required,mask,minlength">
          <arg0 key="register.error.username.required" />
          <arg1 name="minlength" key="${var:minlength}" resource="false" />
          <var>
```

```
                              <var-name>mask</var-name>
                              <var-value>^\w+$</var-value>
                          </var>
                          <var>
                              <var-name>minlength</var-name>
                              <var-value>5</var-value>
                          </var>
                  </field>
</form>
```

The format of a validation as defined in the XML file is:

- `formname` is the form name to be validate.

- `depends` is a list of validations, for example required.

- `mask` validation is followed by a pattern.

  This checks the field value against a Jakarta RegExp expression. For details of regular expressions see

  http://jakarta.apache.org/regexp/index.html

- `minlength` validation

- `arg0` is an Error Message defined in ApplicationsResources.properties as described in step 4 on page 455)

Tip: You can user other error messages if you use "msg" tag and the error name for a field. For example, enter the following in the validation.xml file:

```
<field property="emailAddr" depends="required,email">
            <msg name="required" key="register.error.email.required" />
            <msg name="email" key="register.error.email.valid" />
        </field>
```

Or you can modify the default values the ApplicationResources.properties file. For example enter:

```
register.error.email.required=Please type your email address
register.error.email.valid=email invalid
```

11.7.1 Using the Validator in forms and JSPs

Most of the Struts forms provided in the our sample code extend the standard Struts ActionForm and include their own validate() method. In order to use the Validation framework, the form must extend the ValidatorForm. One example of this in the sample code is the UserDetailsFormBean as shown in Example 11-3 on page 465.

Example 11-3   Using the validation framework with Struts forms

```
public class UserDetailsFormBean extends ValidatorForm

    public ActionErrors validate(
        ActionMapping mapping,
        HttpServletRequest request)
    {
        logger.info("ENTRY: validate");

        // validate super first ....
        ActionErrors errors = super.validate(mapping, request);

        if (!getPassword().equals(getPassword_conf())) {
            errors.add("register",
                new ActionError("register.error.password.notmatch"));
        }

        logger.info("EXIT:  validate");
        return errors;
    }
```

The validation for this form also includes validation performed in addition to that provided by the validator. Specifically, the password and password confirm are checked to see that they match. If no extra validation is required for a form, it must not have a validate() method. The validate method for the UserDetailsFormBean first calls the super.validate(), this allows the validation framework to apply the basic validation for the form. The extra validation required for the form is then checked and added to any errors that were detected by the validation framework.

The validation framework is capable of performing client-side validation in JavaScript. If the form is simply submitted, as is done in the addNewsItemBody.jspf, then all of the validation will be performed on the server-side.

JavaScript validation is enabled by changing the JSP as shown in Example 11-4.

Example 11-4   Enabling JavaScript validation

```
<html:form action="/register" onsubmit="return
validateUserDetailsFormBean(this);">

<html:javascript formName="userDetailsFormBean" />
```

If client-side JavaScript validation is enabled, viewing the source code for the delivered page will show the validation rules provided by the server. Failures in validation result in a JavaScript pop-up window containing the validation error messages. Server-side validation failures are shown by the standard <html:errors> tags.

This can be seen in user registration. If mandatory data is missing, a pop-up window shows the validation failures. If the password and confirmation password do not match, a server-side error message is displayed at the top of the page.

> **Important:** As can be seen from these examples, while the validation framework works with the Rational Software Development Platform tooling, it is not explicitly supported by the tooling. For example, the validation.xml file needs to be modified by hand. Changes to the validation.xml can require republishing of the application in order for them to take effect.

The validation framework provides several very useful categories. User registration makes use of email validation. Another good example for the use of the validation framework is in the news section. There, date validation is performed. The date validation is locale aware.

Developers are permitted to define their own validation rules. A common example of this would be to define a validation rule for a customer account number. This provides a centralized location for validation so the rules for a valid customer account number are defined in one place.

Also note that the validation framework supports the use of multiple validation rules XML files. For large projects, it is preferable to have individual validation rules XML configuration files for the various sections of the application. These are defined on the pathnames property in the struts-config.xml file for the validator plug-in.

11.8 Templating and Struts

The individual JSPs that make up an application can all be created from scratch. This is a poor approach because it requires a fair amount of work, does not support reuse, and requires rework if the layout of the application changes. In the Sal301 sample application developed for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301, JSP includes were used to provide a standard look and feel for the application. Using this approach, a template page is created. This template is then copied and modified for each new page.

This approach works, but has drawbacks and limitations. For one, the Rational Software Development Platform tooling does not follow JSP includes very well. Thus, if a table, for example, begins in one included JSP and ends in another, the tooling will not be able to match <TABLE></TABLE> tags and produces warnings in the JSP files.

Perhaps more importantly, the JSP design and preview modes do not work as expected. Using JSP includes leads to rendering errors in these views. This is again due to the tooling not following JSP includes.

Finally, using JSP includes as a templating method can lead to the case where the JSP template itself is changed. This can require rework in all of the existing JSPs that were copied from the template.

The Rational Software Development Platform tooling provides a templating mechanism that is very similar to Tiles. Tiles is the standard Struts templating mechanism. Tiles is provided with the Struts distribution in Rational Web Developer, but like the validation framework, it is not directly supported. The Rational Web Developer templating mechanism is very similar to Tiles. The main difference in the mechanisms is that Tiles layouts can be hierarchical. One layout can extend another. The templates provided in the Rational Web Developer tooling are static.

For our Sa404 sample application we used the Rational Web Developer templates to provide a standard look and feel for all our Struts based JSP pages. The template in this application has the standard presentation areas:

- ► Header bar at the top of the page
- ► Footer bar at the bottom of the page
- ► Menu bar on the left hand side of the page
- ► Body area
- ► Message area for error messages

The area for error messages is interesting in that it provides for a standard placement for error messages. Developers do not (and should not) need to put explicit provisions into the JSPs for displaying error messages. This leaves the placement of error messages up to the template. In this example, the errors area takes up no space on the page if there are no errors.

The Struts template is in the theme folder of our SAL404RealtyWeb project. This is the default location and is named sal404.jtpl. The template itself is comprised of JSP fragments. Each of the standard areas (menu bar, header, footer and error) are implemented as JSP fragments and are located in the general folder.

The template itself is implemented as a simple HTML table. The various JSP fragments for the content areas are then placed into the table cells. Templates

can be easily assembled using the tools provided in the Palette view under the Page Template menu of the Rational Web Developer. Finally, the template requires at least one content area. This is added to the template by dragging and dropping a Content Area from the Page Template menu into the body area of the template.

11.8.1 Using templates

The templates are quite easy to use.

1. When creating a new JSP, select **Create from page template** as shown in Figure 11-11

Figure 11-11   Using templates

2. Select the desired template as shown in Figure 11-12 on page 469.
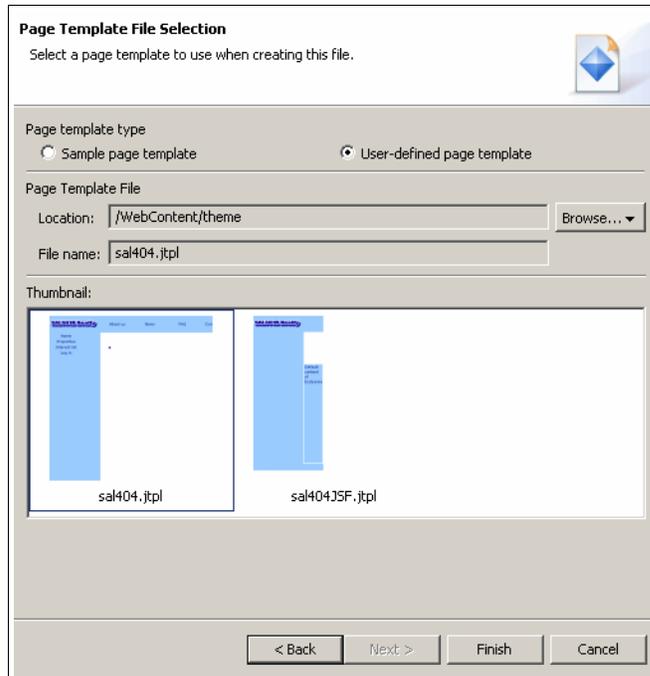
Figure 11-12   Select the template

3. Click **Finish**.

    Our examples here follow the convention of naming the JSP fragments for the body area as pageNameBody. This is not required. The page created from the template can be named pageName.jsp, while the body area JSP fragment can be named pageName.jspf.

4. When creating a body JSP fragment, select **Create as JSP Fragment** as shown in Figure 11-13 on page 470.
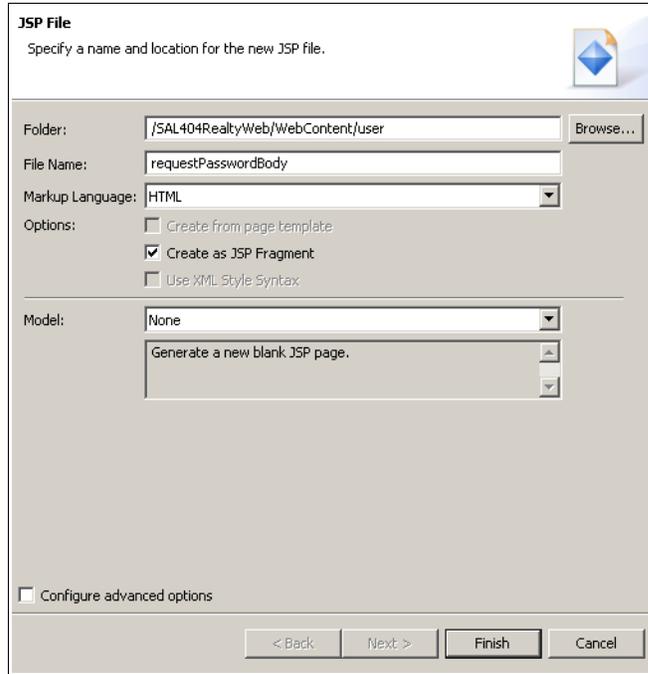
Figure 11-13   Create the page body

5. The JSP page body fragment is then placed onto the page using the Page Fragment item from the Page Template Palette. Dragging and dropping the Page Fragment onto the contents area of the new page brings up the selection dialog box shown in Figure 11-14 on page 471.
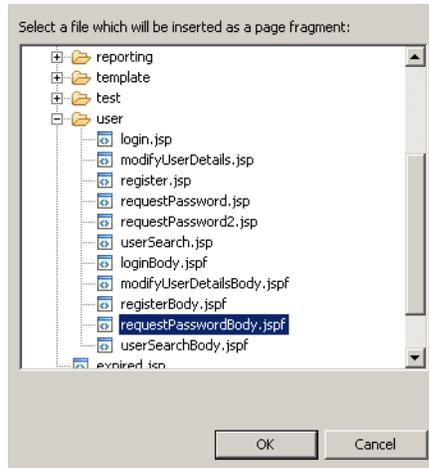
Figure 11-14   Select the body page fragment

6. The body of the request password JSP is shown in Example 11-5.

Example 11-5   Request password body

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<BR>
<h1>Request Password</h1>
<HR>
<BR>
<html:form action="/requestPassword">
    <TABLE border="0">
        <TBODY>
            <TR>
                <TD>
                <P>Please type in your registered user name and email:</P>
                <BR>
                <TD>
            </TR>
            <TR>
                <TH>User ID:</TH>
                <TD><html:text property='username' /></TD>
            </TR>
            <TR>
                <TH>Email:</TH>
                <TD><html:text property='emailAddress' /></TD>
            </TR>
```

```
            <TR>
                <TD></TD>
                <TD><BR>
                <html:submit property="submit" value="Submit" /></TD>
            </TR>
        </TBODY>
    </TABLE>
</html:form>
```

> **Note:** Compare how clean the page bodies become compared to the pages that use the JSP include templating technique. The Struts taglibs are declared. This is required. None of the enclosing taglibs and definitions from the template are available in the page body. The page body then has simply a form that contains a table.

Many of the JSP body fragments have code similar to that from modifyUserDetailsBody.jspf as shown in Example 11-6.

Example 11-6   Define a bean to be used on the page.

```
<%--
MUST use bean define so that Struts will copy the input attributes back to
the form
--%>
<c:if test="${!empty userSessionData.user}">
    <bean:define id="user" name="userSessionData"
property="user"></bean:define>
</c:if>

<c:if test="${empty userSessionData.user}">
    <bean:define id="user" name="userSessionData"
property="logInUser"></bean:define>
</c:if>
```

This code defines a bean from the session data that is then used in the JSP body fragment. This particular case uses <c:if> tags to determine which user is being modified.

11.9 Struts modules

When we wrote our previous redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301 the Web development

tools of WebSphere Studio only supported the use of one Struts module in a Web application. Multiple modules have supported Struts for some time, and support for them is are now available in the Rational Software Development Platform Web development tools.

In large projects, and even in smaller projects, it is common for multiple developers to need to modify the struts-config.xml file. Adding forms and actions updates the struts-config.xml file as does changing Struts forwards. This makes conflicts to changes in the struts-config.xml file common.

By using a different Struts module for the various sections of an application, conflicts in the configuration files can be minimized or avoided.

The Rational Software Development Platform tooling now supports Struts modules. This is an important addition to the tooling. In previous releases, only one Struts module was supported. The default module is usually named struts-config.xml, and access to this one module would become a bottleneck for team development. If multiple developers made changes to the Struts configuration simultaneously, these changes would need to be manually merged. This process is both time consuming and error prone. It is also not uncommon for one developer to overwrite the work of another developer.

Creating a new module is quite simple. Right-click the **Struts** icon in a Web project and choose **New** → **Module** as shown in Figure 11-15 on page 474.
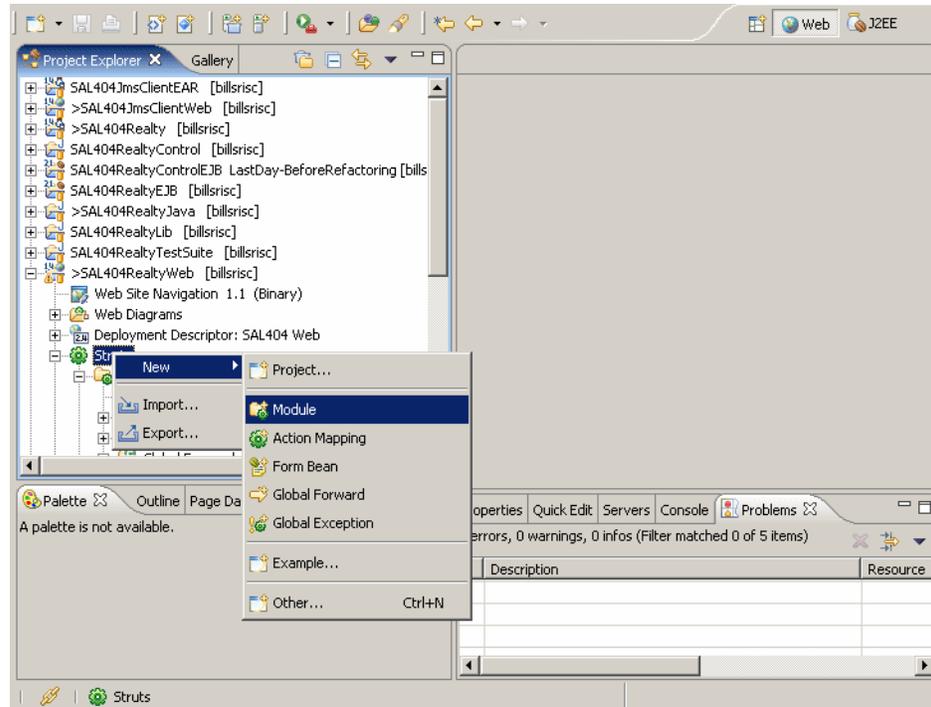
Figure 11-15   Create a Struts module

The application presented in this redbook has a bidding component, so we created a new Struts module for bidding. See Chapter 15, "Bidding component" on page 549 for details.

12

Web services

The objective of this chapter is to introduce the Web services technology and provide an example of developing a Web service in Rational Web Developer. This chapter discusses the following:

- ► Web services overview
- ► Rational Web Developer support for Web services
- ► Extend the sample application using Web services

12.1 Web services overview

The industry standard definition of *Web services* is that they are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Simply put, it is a technology that enables the invoking of applications using Internet protocols and standards. In this chapter, first we introduce the concept of a service-oriented architecture (SOA) which promises to better integrate today's highly heterogeneous environments using an approach that links services together to build complex, yet manageable solutions. We then show how Web services implement a service-oriented architecture.

12.1.1 Service-oriented architecture (SOA)

In a s*ervice-oriented architecture*, applications are made up from loosely-coupled software services that interact to provide all the functionality needed by the application. Each service is generally designed to be very self-contained and stateless to simplify the communication which takes place between them.

There are three main roles involved in a service-oriented architecture:

► Service provider
► Service broker
► Service requester

The interactions between these roles are shown in Figure 12-1.

```
                    ┌──────────┐
                    │ Service  │
                    │ Broker   │
                    └──────────┘
          look up                  register

   ┌──────────┐                  ┌──────────┐
   │ Service  │ ───── bind ────→ │ Service  │
   │Requester │                  │ Provider │
   └──────────┘                  └──────────┘
```
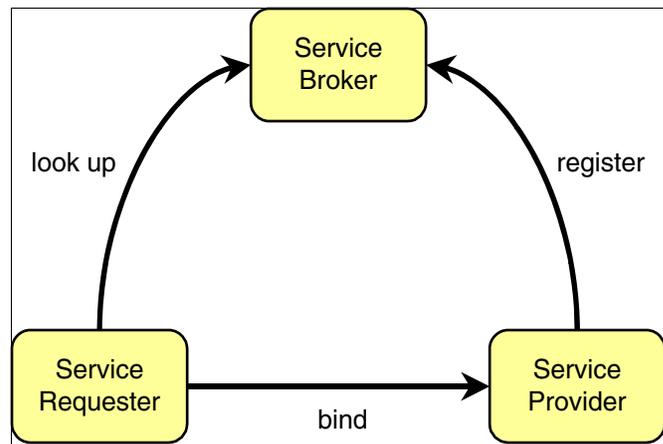
Figure 12-1   Service-oriented architecture

Service provider

The *service provider* creates a service and may publish its interface and access information to a *service broker*.

A service provider must decide which services to expose and how to expose them. There is often a trade-off between security and interoperability; the service provider must make technology decisions based on this trade-off. If the service provider is using a service broker, decisions must be made on how to categorize the service and the service must be registered with the service broker using agreed protocols.

Service broker

The *service broker,* also known as the *service registry*, is responsible for making the service interface and implementation access information available to any potential service requester.

The service broker provides mechanisms for registering and finding services. A particular broker might be public (for example, available on the Internet) or private only available to a limited audience (for example, on an intranet). The type and format of the information stored by a broker and the access mechanisms used is implementation-dependent.

Service requester

The *service requester*, also know as a *service client,* discovers services and then uses them as part of its operation.

A service requester uses services provided by service providers. Using some agreed upon protocol, the requester can find the required information about services using a broker (or this information can be obtained in some other way). Once the service requester has the necessary details of the service, it can bind or connect to the service and invoke operations on it. The binding is usually static, but the possibility of dynamically discovering the service details from a service broker and configuring the client accordingly makes dynamic binding possible.

12.1.2 Web services as an SOA implementation

Web services provides a technology foundation for implementing a service-oriented architecture. A major focus during the development of this technology is to make the functional building blocks accessible over standard Internet protocols that are independent of platforms and programming languages to ensure that very high levels of interoperability are possible.

Web services are self-contained software services which can be accessed using simple protocols over a network. They can also be described using standard mechanisms and these descriptions can be published and located using standard registries. Web services can perform a wide variety of tasks, ranging from simple request-reply to full business process interactions.

Using tools like Rational Application Developer, existing resources can be exposed as Web services very easily.

The core technologies used for Web services are as follows:

► XML
► SOAP
► WSDL
► UDDI

XML

XML (Extensible Markup Language) is the markup language that underlies Web services. XML is a generic language that can be used to describe any kind of content in a structured way, separated from its presentation to a specific device. All elements of Web services use XML extensively, including XML namespaces and XML schemas.

The specification for XML is available at:

http://www.w3.org/XML/

SOAP

SOAP (not an acronym) is a network, transport, and programming language neutral protocol that allows a client to call a remote service. The message format is XML. SOAP is used for all communication between the service requester and the service provider. The format of the individual SOAP messages depend on the specific details of the service being used.

The specification for SOAP is available at:

http://www.w3.org/TR/soap/

WSDL

WSDL (Web Services Description Language) is an XML-based interface and implementation description language. The service provider uses a WSDL document in order to specify:

► The operations a Web Service provides
► The parameters and data types of these operations
► The service access information

WSDL is one way to make service interface and implementation information available in a UDDI registry. A server can use a WSDL document to deploy a Web Service. A service requester can use a WSDL document to work out how to access a Web Service, or a tool can be used for this purpose.

The specification for WSDL is available at:

http://www.w3.org/TR/wsdl

UDDI

UDDI (Universal Description, Discovery and Integration) is both a client side API and a SOAP-based server implementation which can be used to store and retrieve information about service providers and Web services.

The specification for UDDI is available at:

http://www.uddi.org/

Figure 12-2 shows a how the Web services technologies are used to implement an SOA.
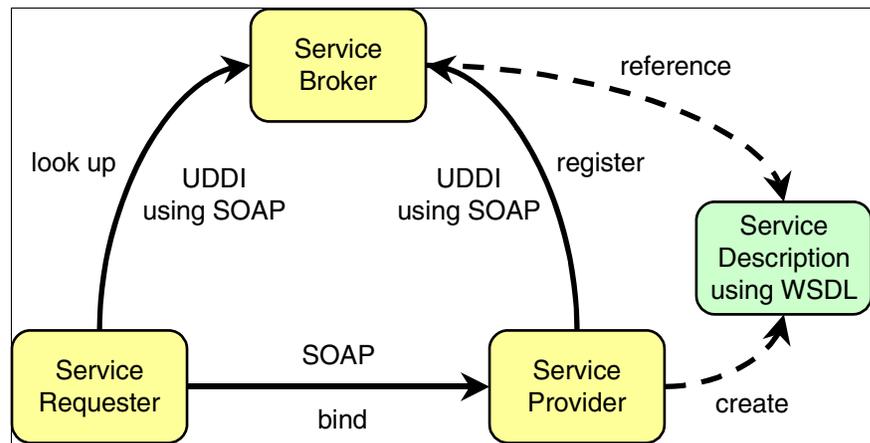
Figure 12-2   Web services implementation of an SOA

12.1.3 Properties of Web services

The key properties of Web services include:

► Web services are self-contained.

On the client side, no additional software is required. A programming language with XML and HTTP client support is enough to get you started. On the server side, merely an HTTP server and a SOAP server are required. It is

possible to Web services enable an existing application without writing a single line of code.

► Web services are self-describing.

The definition of the message format travels with the message; no external metadata repositories or code generation tools are required.

► Web services can be published, located, and invoked across the Web.

This technology uses established lightweight Internet standards such as HTTP. It leverages the existing infrastructure. Some additional standards that are required to do so include SOAP, WSDL, and UDDI.

► Web services are language independent and interoperable.

Client and server can be implemented in different environments. Existing code does not have to be changed in order to be Web service enabled. Basically, any language can be used to implement Web service clients and servers.

► Web services are inherently open and standards based.

XML and HTTP are the major technical foundation for Web services. A large part of the Web service technology has been built using open-source projects. Therefore, vendor independence and interoperability are realistic goals.

► Web services are dynamic.

Dynamic e-business can become reality using Web services because, with UDDI and WSDL, the Web service description and discovery can be automated.

► Web services are loosely coupled.

Traditionally, application design has depended on tight interconnections at both ends. Web services require a simpler level of coordination that allows a more flexible re-configuration for an integration of the services in question.

► Web services provide programmatic access.

The approach provides no graphical user interface; it operates at the code level. Service consumers have to know the interfaces to Web services but do not have to know the implementation details of services.

► Web services provide the ability to wrap existing applications.

Already existing stand-alone applications can easily be integrated into the service-oriented architecture by implementing a Web service as an interface.

12.1.4 Related Web services standards

The basic technologies of XML, SOAP, WSDL and UDDI are fundamental to Web services, but many other standards have been developed to help with developing and using them.

An excellent resource for information about standards related to Web services can be found at:

`http://ibm.com/developerworks/views/webservices/standards.jsp`

Web services in J2EE V1.4

One of the main changes in moving from J2EE V1.3 to V1.4 is the incorporation of Web services into the platform standard. J2EE V1.4 provides support for Web services clients and also allows Web services to be published. The main technologies in J2EE V1.4 which provide this support are as follows:

► Java API for XML-based Remote Procedure Calls (JAX-RPC)

   JAX-RPC provides an API for Web services clients to invoke services using SOAP over HTTP. It also defines standard mappings between Java classes and XML types.

► SOAP with Attachments API for Java (SAAJ)

   SAAJ allows SOAP messages to be manipulated from within Java code. The API includes classes to represent such concepts as SOAP envelopes (the basic packaging mechanism within SOAP), SOAP faults (the SOAP equivalent of Java exceptions), SOAP connections and attachments to SOAP messages.

► Web services for J2EE

   This specification deals with the deployment of Web Service clients and Web services themselves. Under this specification, Web services can be implemented using JavaBeans or stateless session EJBs.

► Java API for XML Registries (JAXR)

   This API deals with accessing XML registry servers, such as servers providing UDDI functionality.

The specifications for Web services support in J2EE V1.4 are available at:

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

Web services interoperability

In an effort to improve the interoperability of Web services, the Web Services Interoperability Organization (known as *WS-I*) was formed. WS-I produces a specification, known as the *WS-I Basic Profile*, which describes the technology

choices that maximize interoperability between Web services and clients running on different platforms, using different runtime systems and written in different languages.

The WS-I Basic Profile is available at this Web site:

http://ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile

Web services security

Although not all runtimes support security for Web services, a body of standards are evolving which describe how Web services can be secured. The technical basis for these standards is known as *WS-Security,* which provides the basic encryption and digital signature technologies. In addition, several other specifications now use WS-Security for defining trust models, creating secure channels between Web services and their clients, and ensuring that clients are authorized to use Web services.

The specification for WS-Security is managed by OASIS:

http://www.oasis-open.org/

Web services workflow

Business Process Execution Language for Web Services (BPEL4WS) provides a language for the specification of business processes and business interactions protocols, extending the basic Web services model to include business transaction support.

The specification for BPEL4WS is available at this Web site:

http://ibm.com/developerworks/webservices/library/ws-bpel/

Web Services Inspection Language

Web Services Inspection Language (WSIL) is also known as WS-Inspection and it can be used as an alternative to registering Web services using UDDI. With WS-Inspection, a site can be inspected for Web services and all the necessary information about the available Web services can be obtained from this inspection.

The WS-Inspection specification is available at this Web site:

http://ibm.com/developerworks/webservices/library/ws-wsilspec.html

12.2 Web services tools

Rational Web Developer provides tools to create Web services from existing Java and other resources or from WSDL files, as well as tools for Web services client development and for testing Web services. Rational Web Developer provides tools to assist with the following aspects for Web services development:

► Discover

Browse Universal Description, Discovery and Integration (UDDI) registries or Web services inspection language (WSIL) sites to find Web services for integration. The IBM Web Services Explorer provides the all necessary functions to discover a Web service.

► Create service provider

Use the Rational Web Developer tooling to create Web services from existing artifacts, such as JavaBeans, Web sites that take and return data, DB2 XML Extender calls, DB2 stored procedures and SQL queries.

► Create service consumer

Use the Web services client tools (wizard or command line), to create a client for any Web service. The generation tools analyze service WSDL files to create the client components.

► Test

Web services can be tested, running locally or remotely. for local test, the WebSphere test environment can be used. Rational Web Developer provides functions to easily create and run Web service component tests for HTTP bound Web services. See 12.2.4, "Testing tools for Web services" on page 485.

► Publish

Publish Web services to a public or private UDDI v2 or v3 Business Registry, using the Web Services Explorer, enabling access to them.

► Validate

Use the WSDL and DADX validators to check for structural and semantic problems in these types of files. This feature is useful when receiving a service WSDL file from a service provider, to check the files are valid.

► Secure

The Web service wizards and deployment descriptor editors assist to configure Web services security (WS-Security) for the WebSphere Application Server Environment.

► Run

Run Web services provider and consumer components in WebSphere Application Server or Tomcat test environments. The deployment and administration for the WebSphere test environment is integrated in, respectively can be started via Rational Web Developer.

12.2.1 Creating a Web Service from existing resources

Rational Web Developer provides wizards for exposing a variety of resources as Web services. The following resources can be used to build a Web Service:

► JavaBean:

The Web Service wizard assists you in creating a new Web Service from a simple Java class, configures it for deployment, and deploys the Web Service to a server. The server can be the WebSphere Application Server V6.0 Test Environment included with Rational Web Developer or an another application server.

► EJB

The Web Service wizard assists you in creating a new Web Service from a stateless session EJB, configures it for deployment, and deploys the Web Service to a server.

► DADX

Document access definition extension (DADX) is an XML document format that specifies how to create a Web Service using a set of operations that are defined by DAD documents and SQL statements. A DADX Web Service enables you to expose DB2 XML Extender or regular SQL statements as a Web Service. The DADX file defines the operations available to the DADX run-time environment and the input and output parameters for the SQL operation.

► URL

The Web Service wizard assists you in creating a new Web Service that directly accesses a servlet running on a server.

► ISD

An ISD file is an existing Web service deployment descriptor. It provides information to the SOAP runtime about the service that should be made available to clients (for example URI, methods, implementation classes, serializers and deserializers). When using a Web services runtime based on Apache SOAP, ISD files are concatenated into the SOAP deployment descriptor (dds.xml). This mechanism has been replaced in more recent Web services runtimes, such as Apache Axis and J2EE Web services runtimes.

12.2.2 Creating a skeleton Web service

Rational Application Developer provides the functionality to create Web services from a description in a WSDL (or WSIL) file:

► JavaBean from WSDL

The Web Service wizard assists you in creating a skeleton JavaBean from an existing WSDL document. The skeleton bean contains a set of methods that correspond to the operations described in the WSDL document. When the bean is created, each method has a trivial implementation that you replace by editing the bean.

► Enterprise JavaBean from WSDL

The Web services tools support the generation of a skeleton EJB from an existing WSDL file. Apart from the type of component produced, the process is similar to that for JavaBeans.

12.2.3 Client development

To assist in development of Web Service clients, Rational Application Developer provides these features:

► Java client proxy from WSDL

The Web Service client wizard assists you in generating a proxy JavaBean. This proxy can be used within a client application to greatly simplify the client programming required to access a Web Service.

► Sample Web application from WSDL

Rational Application Developer can generate a sample Web application, which includes the proxy classes described above and sample JSPs that use the proxy classes.

12.2.4 Testing tools for Web services

To enable developers to test Web services, Rational Application Developer provides a range of features:

► WebSphere Application Server V6.0 Test Environment

The V6.0 server is included with Rational Application Developer as a test server can be used to host Web services. It provides a range of Web services runtimes, including an implementation of the J2EE specification standards.

► Sample Web application

The Web application mentioned above can be used to test Web services and the generated proxy it uses.

► Web Services Explorer

This is a simple test environment which can be used to test any Web Service, based only on the WSDL file for the service. The service can be running on a local test server or anywhere else on the network.

► Universal Test Client

The Universal Test Client is a very powerful and flexible test application which is normally used for testing EJBs. Its flexibility makes it possible to test ordinary Java classes, so it can be used to test the generated proxy classes created to simplify client development.

► TCP/IP Monitor

The TCP/IP Monitor works like a proxy server, passing TCP/IP requests on to another server and directing the returned responses back to the originating client. In the process of doing this, it records the TCP/IP messages which are exchanged and can display these in a special view within Rational Application Developer.

12.3 Extend the sample application using Web services

In this section, we describe using Web services to extend the functionality of our Sal404 application. The Web service requirements for our sample application are to provide the following Web services implementations:

► A Web service that allows an external client to search the SAL404 property catalog

► A Web service that allows an external client to list SAL404 news items

► A Web service that allows an external client to add news item to the SAL404 news items

► An outbound service request that will not expect a response, but publish data when the property status has changed. A typical scenario would be notifying a property listing service that new properties had been listed for sale.

12.3.1 Implementing the property search Web service

This example describes how to provide a Web service that access the existing property search functionality of the Sal404 application. Our objective is to reuse the existing SAL404 PropertyCatalogManager class.

Web service preferences
We need to set the Rational Web Developer preferences so that the Web service code generation does not overwrite existing loadable classes. We do this so that

the Web Service wizards will not write any Java classes to the target project that exist in a project, module or JAR file that will be loadable from the target project when the application is running on the server.

If this option is not selected, the Web service code wizards can write Java classes to the target project that mask preexisting classes with the same name in other projects, modules or JAR files. This can result in run-time environment and compilation errors. The steps are:

1. Using Rational Web Developer, choose **Window** → **Preferences**.

2. Expand **Web Services** → **Code Generation.**

3. Check the option **Do not overwrite loadable Java classes** as shown in Figure 12-3,
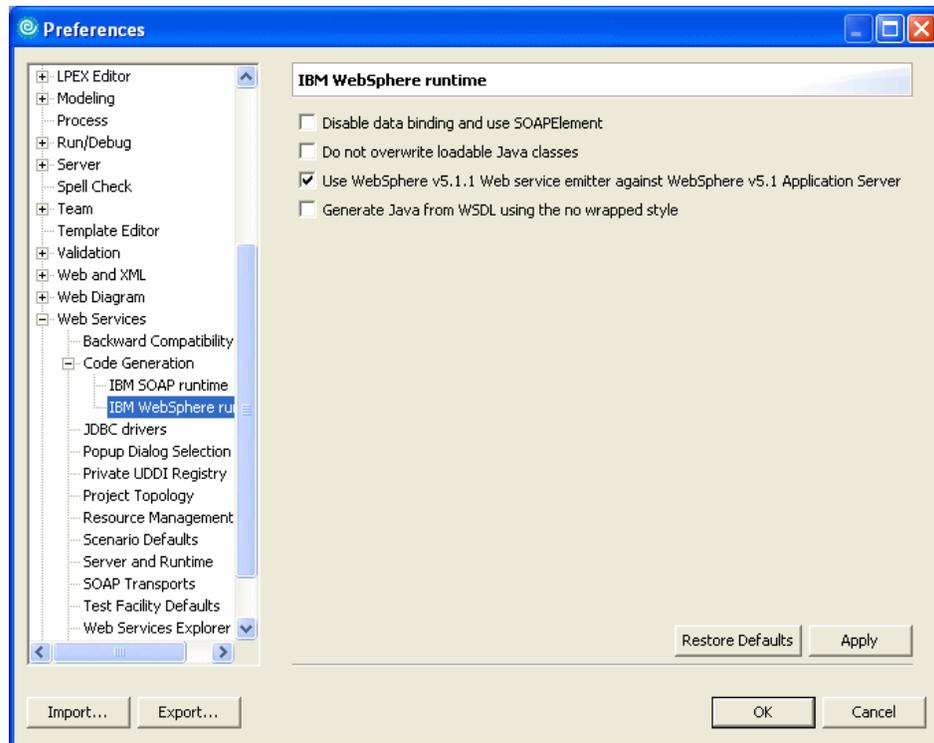
Figure 12-3   Do not overwrite loadable Java classes

4. Click **OK** to close the preferences dialog box.

PropertyCatalogServices class

We created a new class in the SAL404RealtyControl project to wrap the functionality of our PropertyCatalogManager. This is because we want to expose only some of the methods of the PropertyCatalogManager as Web services and we want our Web services to have a different interface. In our specific example the PropertyCatalogManager method searchByCriteria returns a Vector of ResultProptertiesDTO, but for our Web service we want to return an array of ResultProptertiesDTO.

The PropertyCatalogServices class creates a PrpoertyCatalogManager using the following code:

```
PropertyCatalogManager manager = new PropertyCatalogManager();
```

The key method is searchByCriteria which takes an input of a SearchPropertiesByCriteriaDTO and returns an array of ResultPropertiesDTO. It does this by calling the searchByCriteria method of the existing PropertyCatalogManager. See Example 12-1.

Example 12-1   PropertyCatalogServices searchByCriteria method

```
public ResultPropertiesDTO[] searchByCriteria(
          SearchPropertiesByCriteriaDTO searchByCriteriaDTO)
          throws ApplicationException
    {

        logger.info("ENTRY: searchByCriteria");

        Vector results = manager.searchByCriteria(searchByCriteriaDTO);
        ResultPropertiesDTO[] WSResults = new ResultPropertiesDTO[results.size()];
        Iterator iter = results.iterator();
        int i = 0;
        while (iter.hasNext())
        {
            ResultPropertiesDTO element = (ResultPropertiesDTO) iter.next();
            WSResults[i]= element;
            i++;
        }
        logger.info("EXIT:  searchByCriteria");
        return WSResults;
    }
```

Create Web service from PropertyCatalogServices

Now we can create and test our Web service that will search the property catalog by calling the PropertyCatalogServices class.

Generating the Web service and Web service test code

The steps to take are:

1. Locate the PropertyCatalogServices class in the Rational Web Developer Project Explorer, right-click and choose **Web Services** → **Create Web Service**. See Figure 12-4.
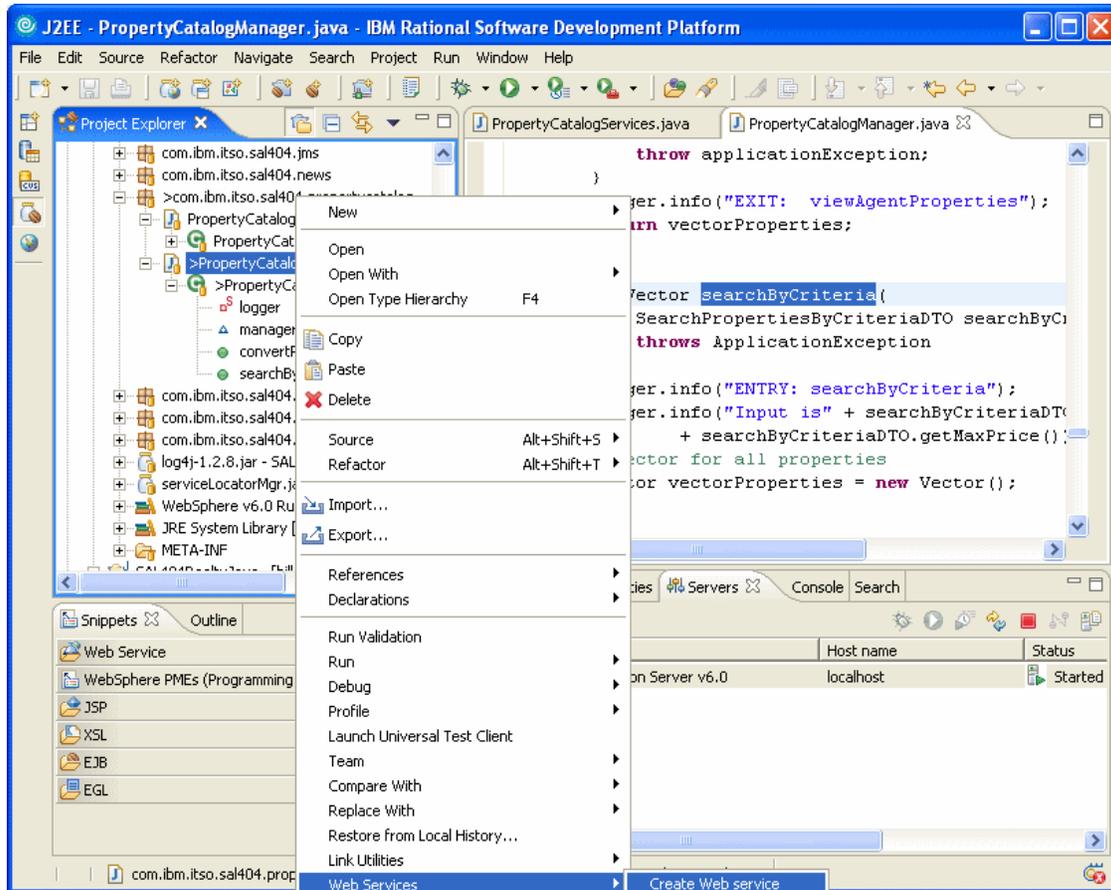
Figure 12-4   Create Web service

2. Choose a Web service type of **Java bean Web Service**, select **Generate a proxy** and **Test the Web service**. Click **Next**. See Figure 12-5 on page 490.
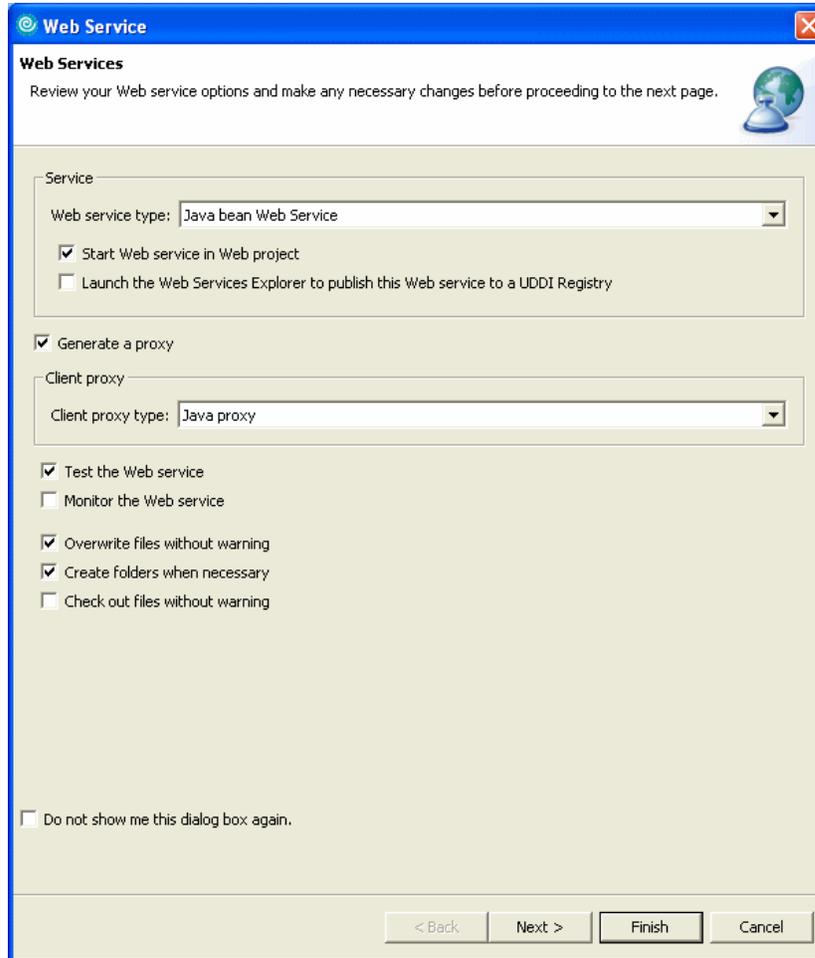
Figure 12-5   Select Web service type

3. Select **PropertyCatalogServices** as the class to use on the Object Selection Page and click **Next**. See Figure 12-6 on page 491.
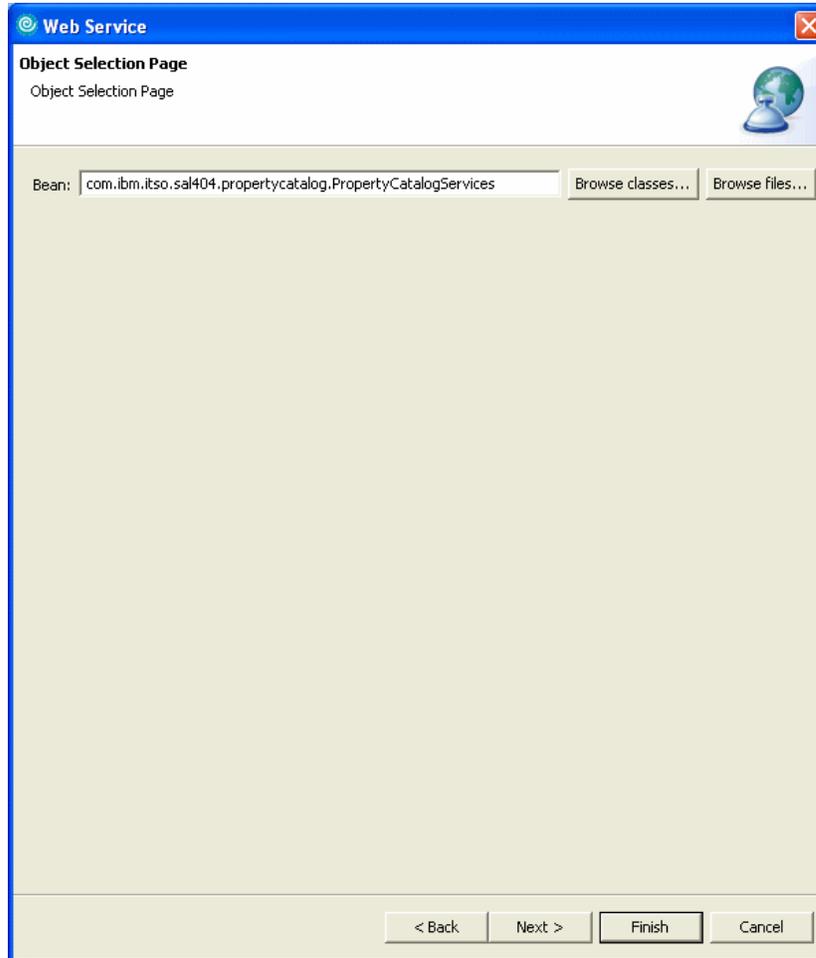
Figure 12-6   Object selection page

4. Now we choose where the generated Web service code should be deployed. On the server side, we choose to generate code in our existing SAL404RealtyWeb application. For testing purposes we created an Enterprise Application project called SAL404TestServices which we used for the client side testing of our Web services. We chose to use a Web-based client for our testing. See Figure 12-7 on page 492 for an example the options we used for our service deployment. Click **Next**.
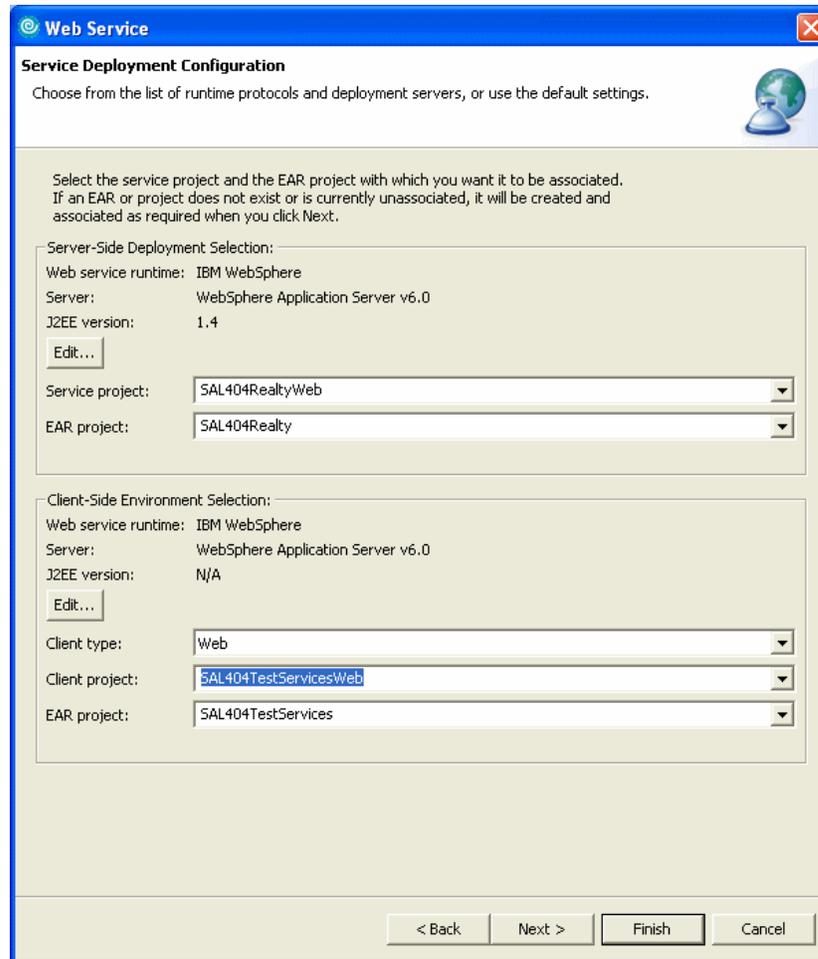
Figure 12-7   Service deployment configuration

5. Accept the defaults for the Service Endpoint Interface and click **Next**.

6. When generating the Web service, we took all the default values to configure our Java bean as a Web service:

   – WSDL file name = **PropertyCatalogServices.wsdl**
   – Style and Use = **Document/Literal**
   – Security Configuration = **No security**
   – Select **Use WSDL 1.1 Mime types exclusively**

   Select to generate a service only for the SearchByCriteria method and click **Next**. See Figure 12-8 on page 493.
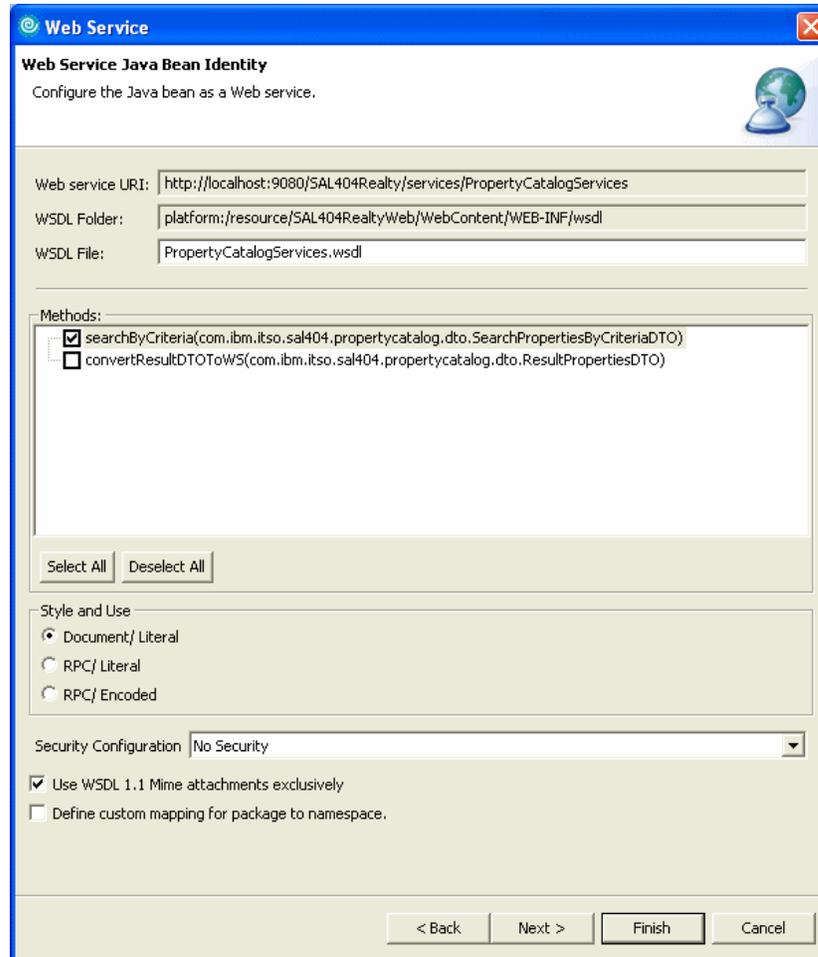
Figure 12-8   Web services generation

7. Leave the test facility set to **Web Services Explorer** as shown in Figure 12-9 on page 494 and click **Next**.
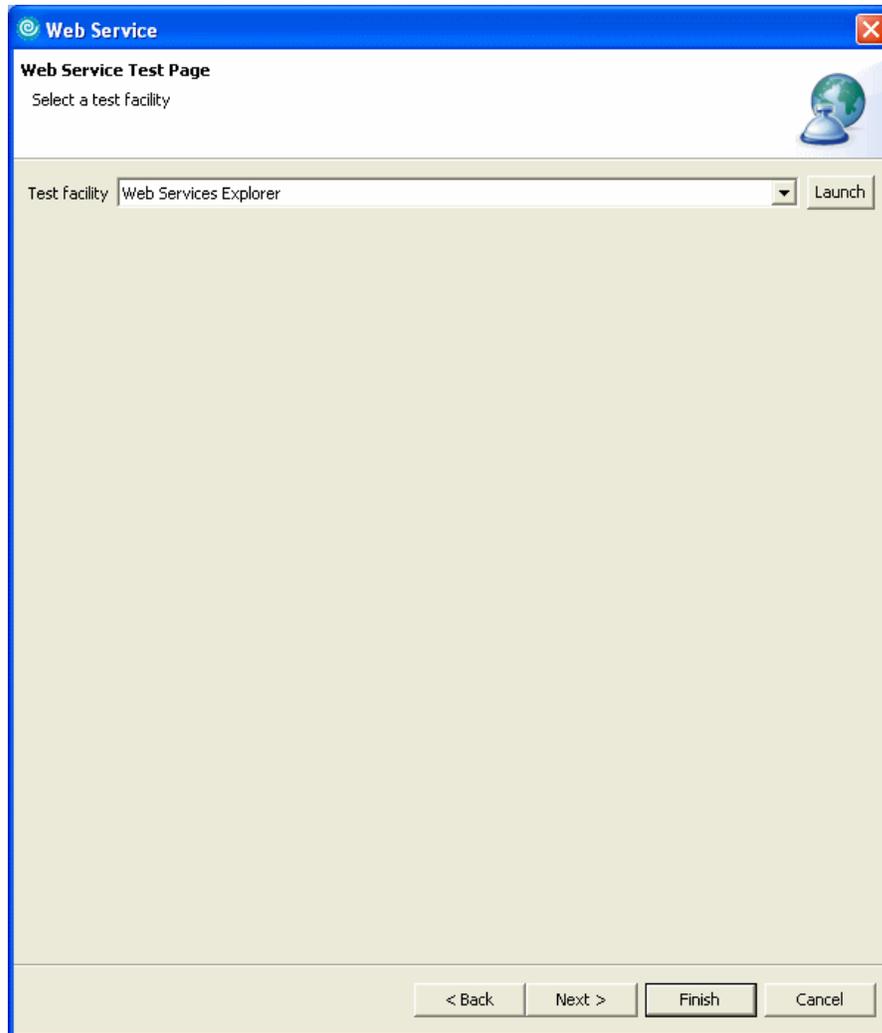
Figure 12-9   Web service test facility

8. Choose **Generate a proxy** and click **Next**. See Figure 12-10 on page 495.
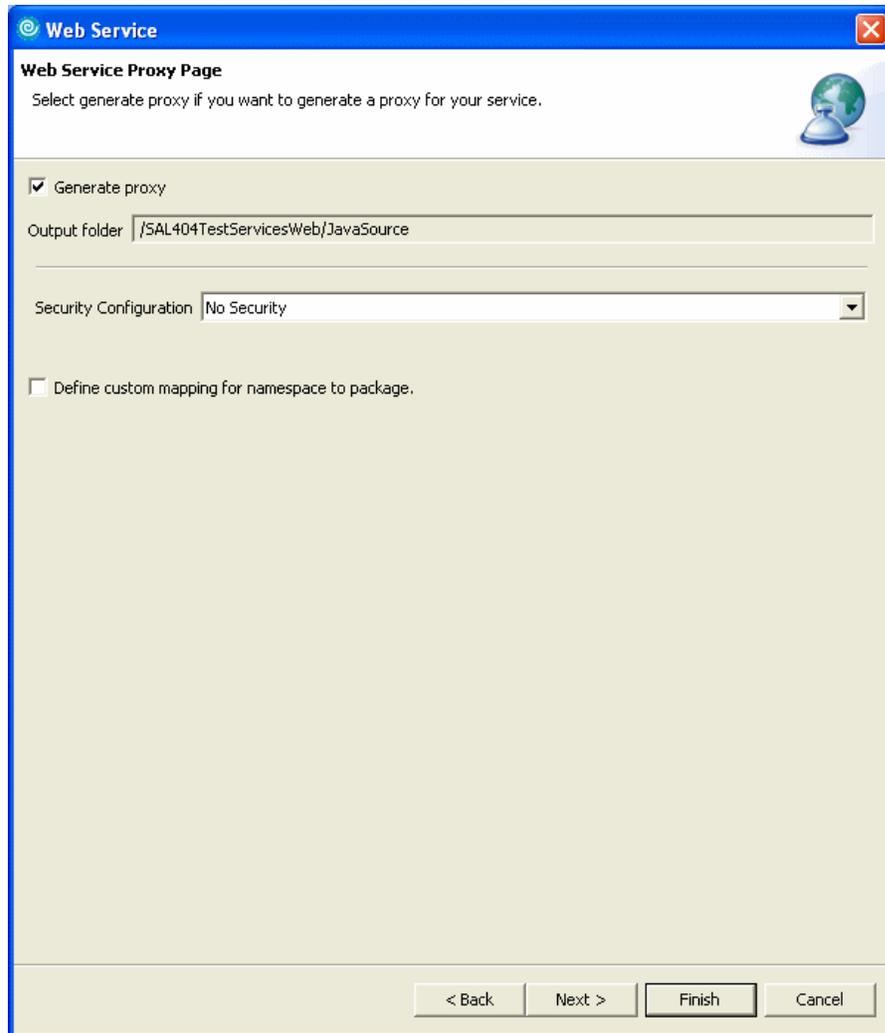
Figure 12-10   Generate a proxy

9. Choose **Test the generated proxy** and **Web service sample JSPs**. Click
   Next. See Figure 12-11 on page 496.
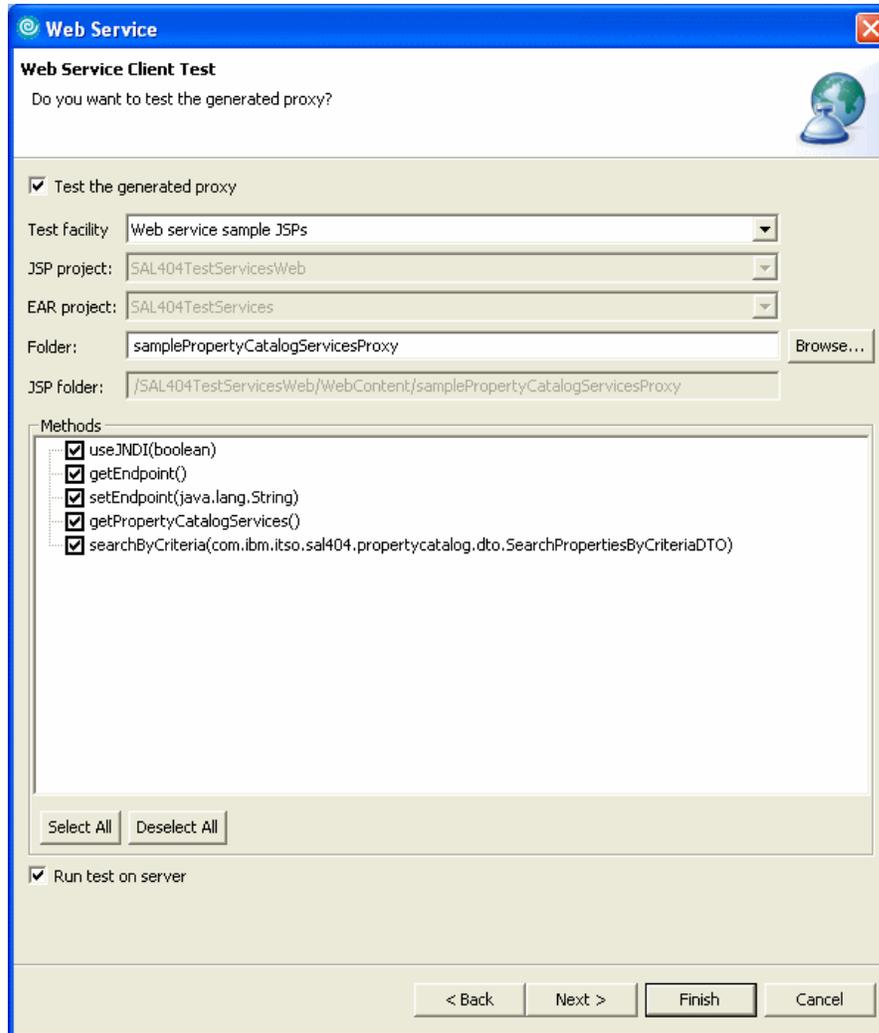
Figure 12-11   Selections for Web service client test

10.We do not need to publish our Web service, so click **Finish** as shown in
    Figure 12-12 on page 497.
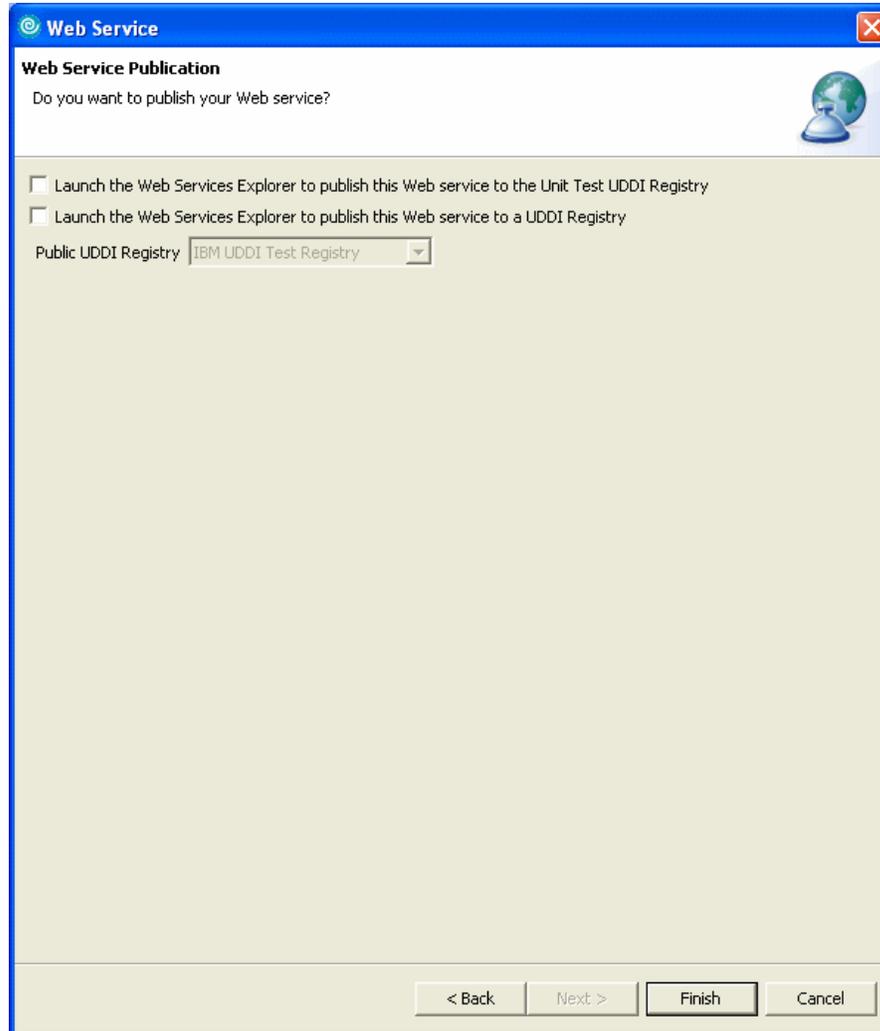
Figure 12-12   Finish Web service creation

Test the Web service

After the new Web service has been generated, the test client will be launched in the internal Web browser of Rational Web Developer as shown in Figure 12-13 on page 498.
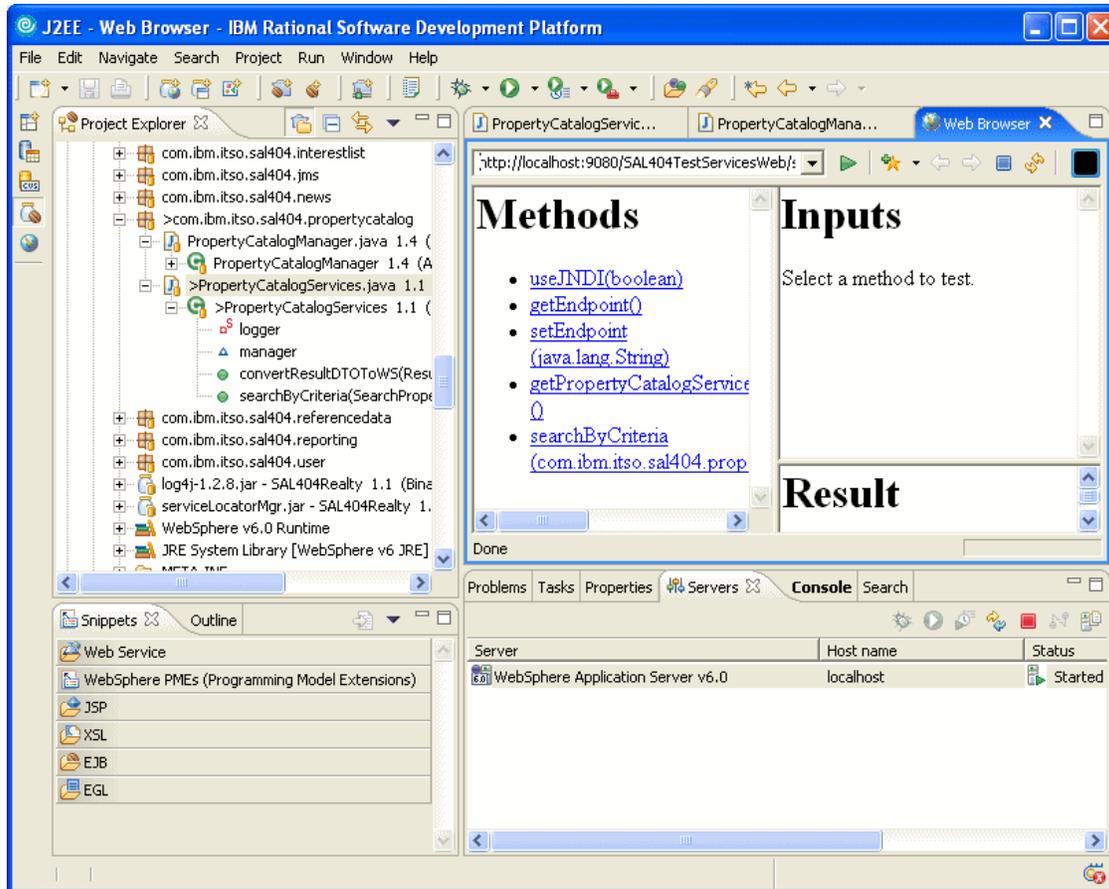
Figure 12-13   test client for Web service

The steps to test the Web service are:

1. Select a method to test as shown in Figure 12-14 on page 499.
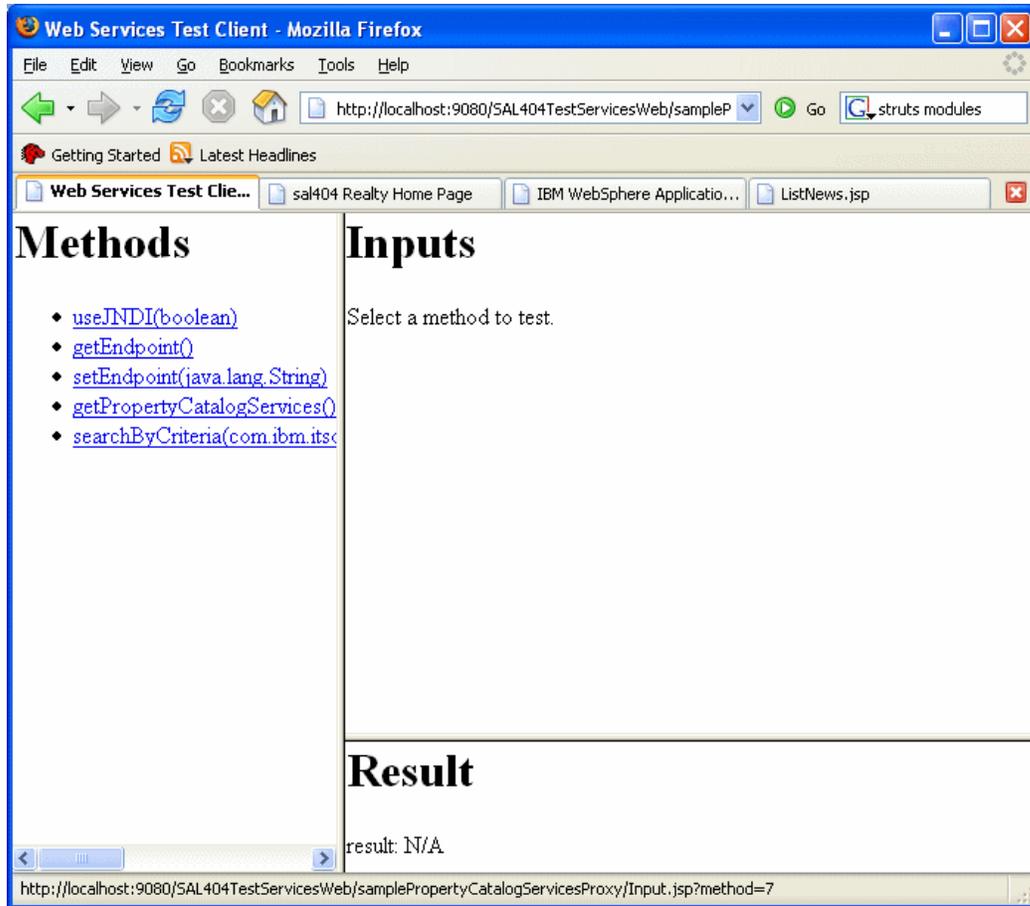
Figure 12-14   Select a method to test

2. Once you have selected a method to test, click **Invoke** to run the method. Figure 12-15 on page 500 shows the result of a successful test of the getEndpoint method.
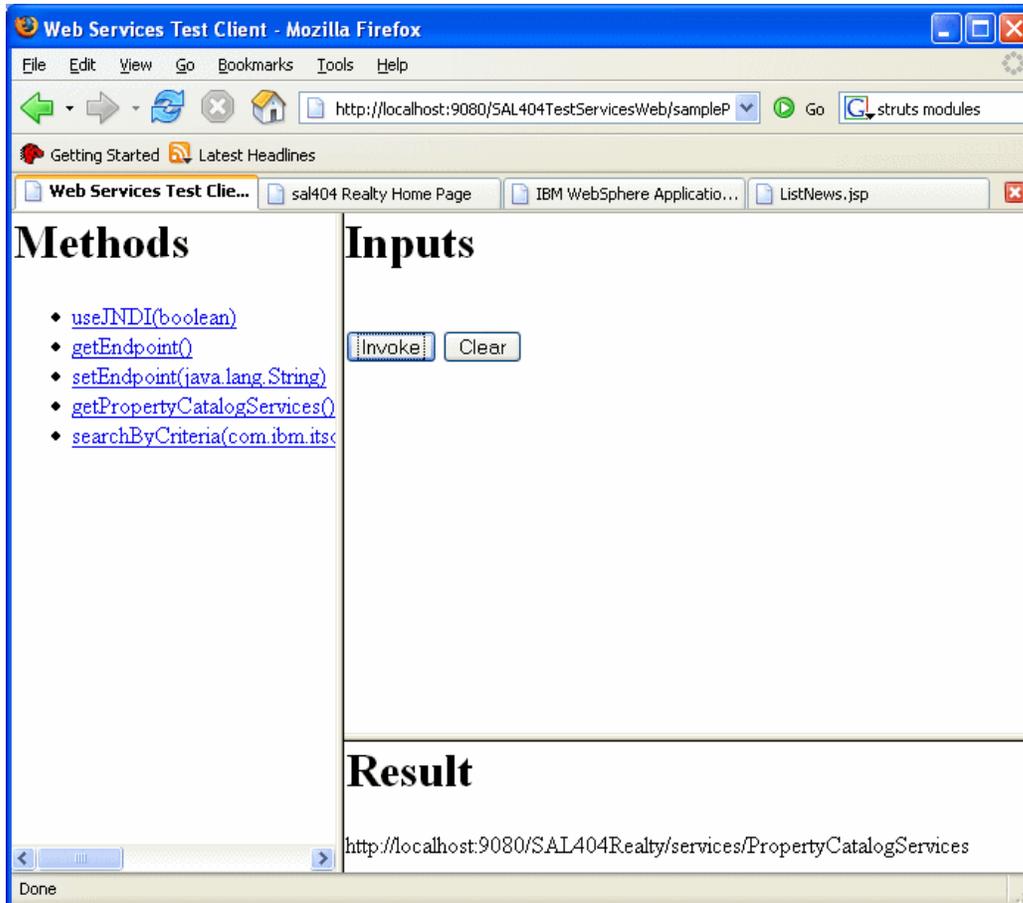
Figure 12-15   Test getEndpoint

3. To test the property search feature of our Web service, select to test the **SearchByCriteria** method, enter the required parameters to create a PropertSearchCriteriaDTO and click **Invoke**. Figure 12-16 on page 501 shows valid parameters for a property search. The propertyTypeId is set to 1 which is for properties of type Single Family Dwelling. The postalCode is set to % so that all properties that meet the other search criteria, regardless of their postal code, will be returned.
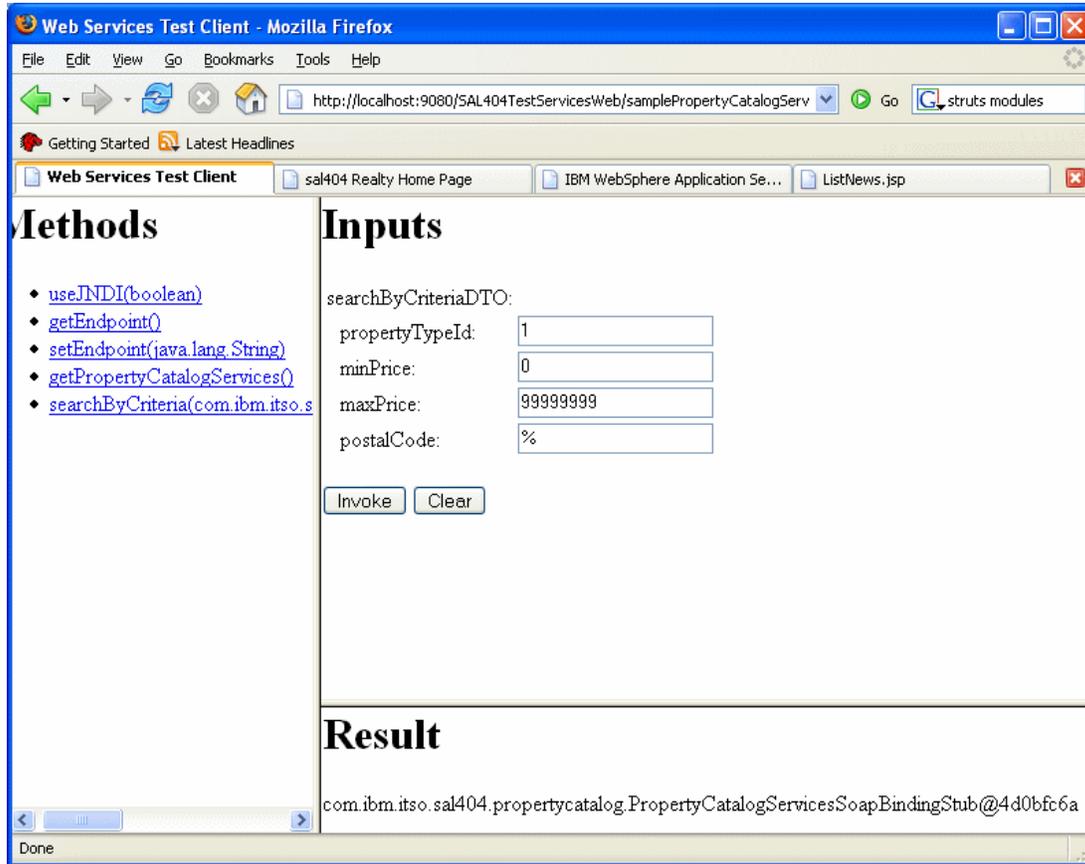
Figure 12-16   Enter parameters for a property search

4. Figure 12-16 also shows the result of this property search. Note that the return result is an array of ResultPropertiesDTO objects so we do not see details of the returned properties in the Result pane.

Test using the Web Services Explorer

One way to better see the result of a property search is to test the Web service using the Web Services Explorer. The steps to take are:

1. Use the project explore to navigate to the Web service PropertyCatalogServicesService. Right-click **PropertyCatalogServicesService** and choose **Test with Web Services Explorer** as shown in Figure 12-17 on page 502.
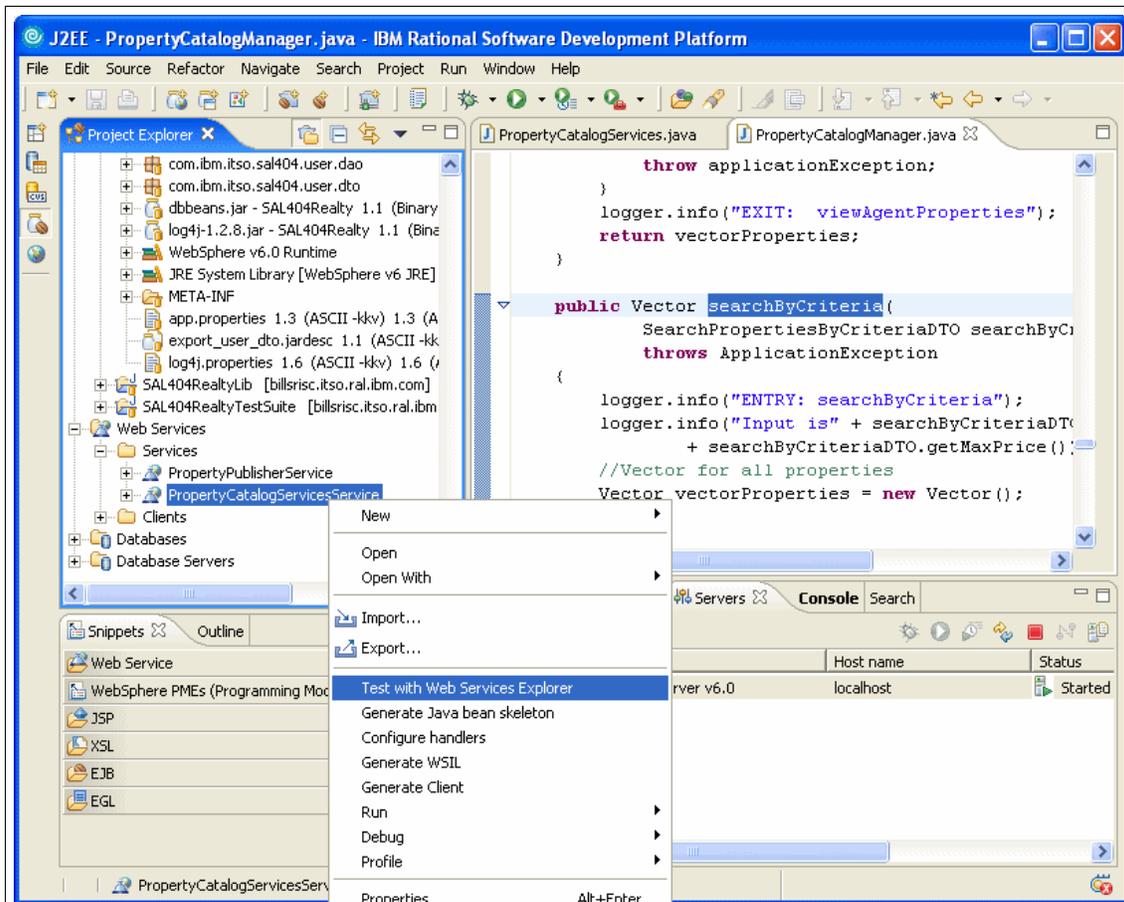
Figure 12-17   Test with Web Services Explorer

2. Invoke the searchByCriteria WSDL operation and provide values for the searchByCriteriaDTO as shown in Figure 12-18 on page 503. Click **Go**.
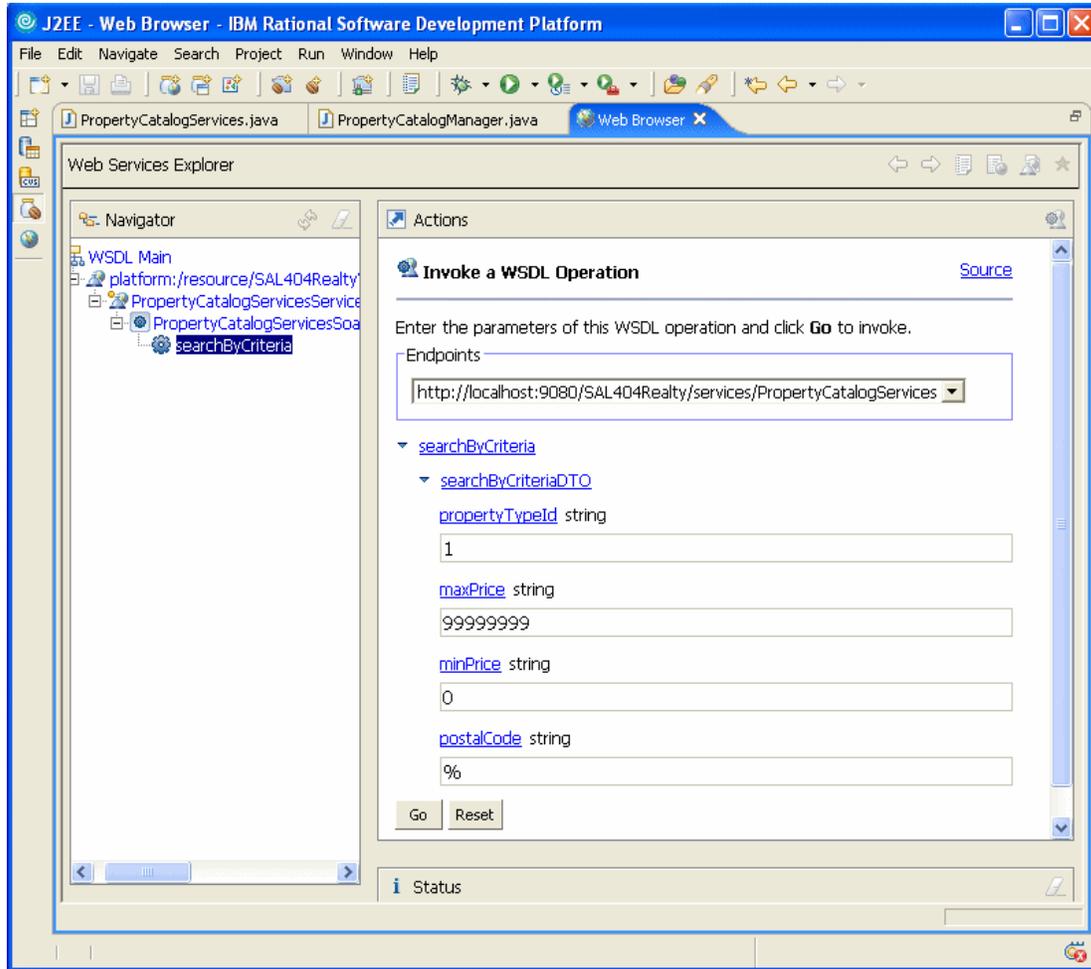
Figure 12-18   Test searchByCriteria WSDL operation

3. Figure 12-19 on page 504 shows the results of a successful test of the search criteria operation. We have expanded the status pane of the results page so that you can see the contents of the ResultPropertiesDTOs.
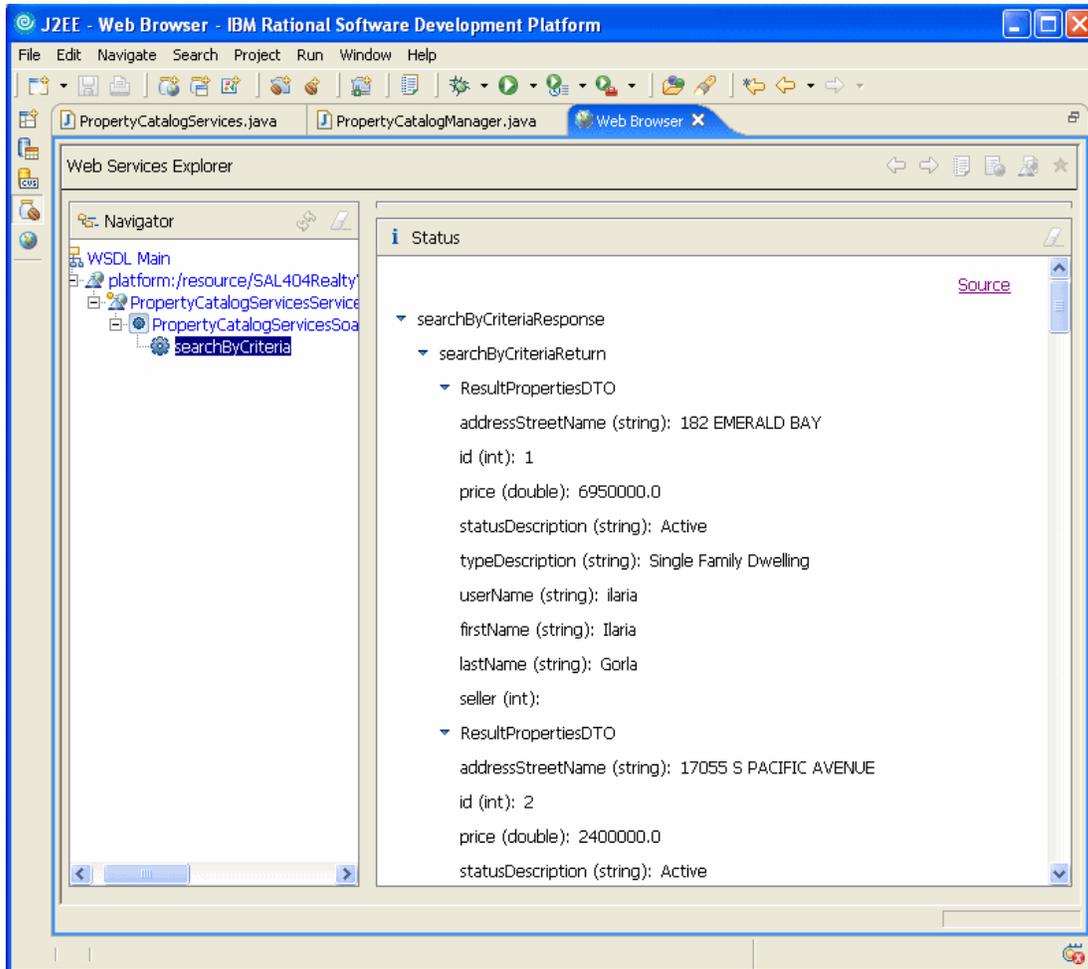
Figure 12-19   Results of testing search by criteria using the Web Services Explorer

12.3.2 Implementing News Web services

In this section we describe how we implemented sample Web services that interact with the News component of our Sal404 application.

Create the WSNewsItemDTO

We created a new DTO called WSNewsItemDTO for use with our Web services that access the News component. This is because the existing NewsItemDTO contains a Date attribute that caused problems for the generated Web services

code. The WSNewsItemDTO simply replaces the news item date with a String representation of the date.

Create Web services methods in NewsManager

We want to build our news Web services by reusing the NewsManager class, but instead of returning NewsItemDTO objects from calls to NewsManager our Web service needs to get WSNewsItemDTO objects. To do this we added some new methods to the NewsManager class. The methods are:

► WSNewsItemDTO convertNewsItemDTOToWSNews(NewsItemDTO news)

  This method converts an input NewsItemDTO into a WSNewsItemDTO

► NewsItemDTO convertWSNewsToNewsItemDTO(WSNewsItemDTO WSNews)

  This method converts an input WSNewsItemDTO into a NewsItemDTO

► Vector viewWSNewsItem()

  This method is a wrapper around the existing viewNewsItem method. The viewNewsItem() method is called to return a vector of NewsItemDTO which are then converted into a vector of WSNewsItemDTO

► WSNewsItemDTO viewWSNewsItemDetails(String newsId)

  This method is a wrapper around the existing viewNewsItemDetails method. The viewNewsItemDetails method is called to return a NewsItemDTO which is then converted into a WSNewsItemDTO

► void addWSNewsItem(WSNewsItemDTO tempWSNewsItemDTO)

  This method is a wrapper around the existing addNewsItem method. The inputWSNewsItemDTO is first converted into a NewsItemDTO and the addNewsItem method is called to insert the news item.

Create news item Web services

The steps to create these Web service are very similar to those documented in "Create Web service from PropertyCatalogServices" on page 488. In summary, you do the following:

1. Locate the NewsManager class in the Rational Web Developer Project Explorer, right-click and choose **Web Services** → **Create Web Service**.

2. Choose a Web service type of **Java bean Web Service**, select **Generate a proxy** and **Test the Web service**. Click **Next**. See Figure 12-20 on page 506.
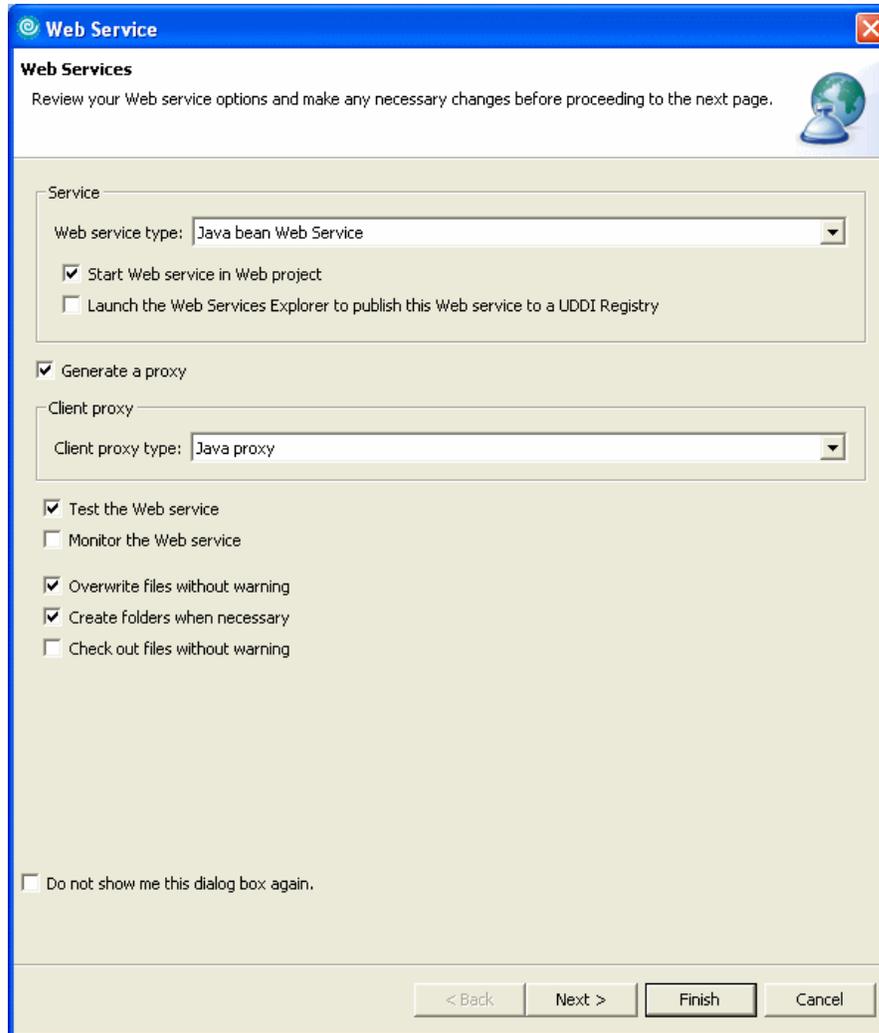
Figure 12-20   Web service creation using the NewsManager

3. Select **NewsManager** as the class to use on the Object Selection Page and click **Next**.

4. Now choose where the generated Web service code should be deployed. On the server side choose to generate code in our existing **SAL404RealtyWeb** application. Use the Enterprise Application project called **SAL404TestServices** for the client side testing of our Web service. Also choose to use a Web-based client for testing. See Figure 12-21 on page 507 for an example of the options we used for our service deployment. Click **Next**.
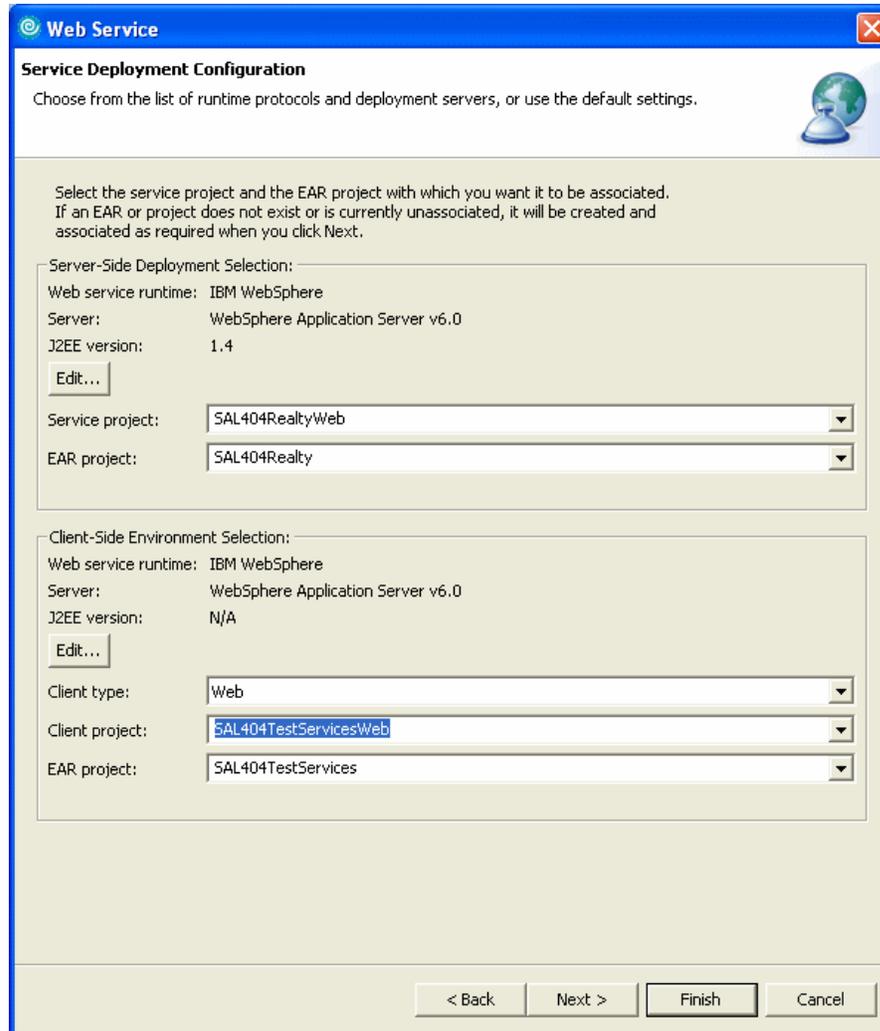
Figure 12-21   Configure service deployment

5. Accept the defaults for the Service Endpoint Interface and click **Next**.

6. When generating the Web service, we took all the default values to configure our Java bean as a Web service:

   – WSDL file name = **NewsManager.wsdl**
   – Style and Use = **Document/Literal**
   – Security Configuration = **No security**
   – Select **Use WSDL 1.1 Mime types exclusively**

   Select to generate a service for three methods:

– viewWSNewsItem
– viewWSNewsItemDetails
– addWSNewsItem

Click **Next**. See Figure 12-22.
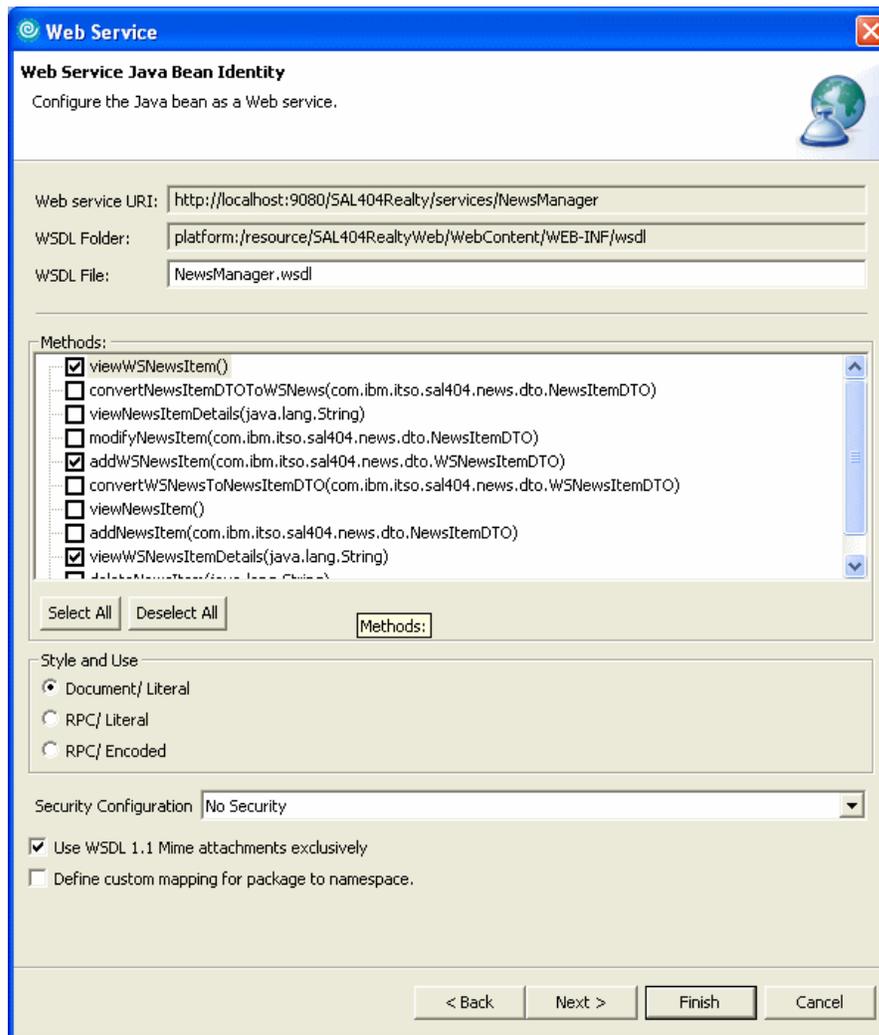
Figure 12-22   Web services generation

7. Leave the test facility set to **Web Services Explorer** and click **Next**.

8. Choose **Generate a proxy** and click **Next**.

9. Choose **Test the generated proxy** and **Web service sample JSPs**. Also remember to click **Select All** so that all methods are selected. Click **Next**. See Figure 12-23.
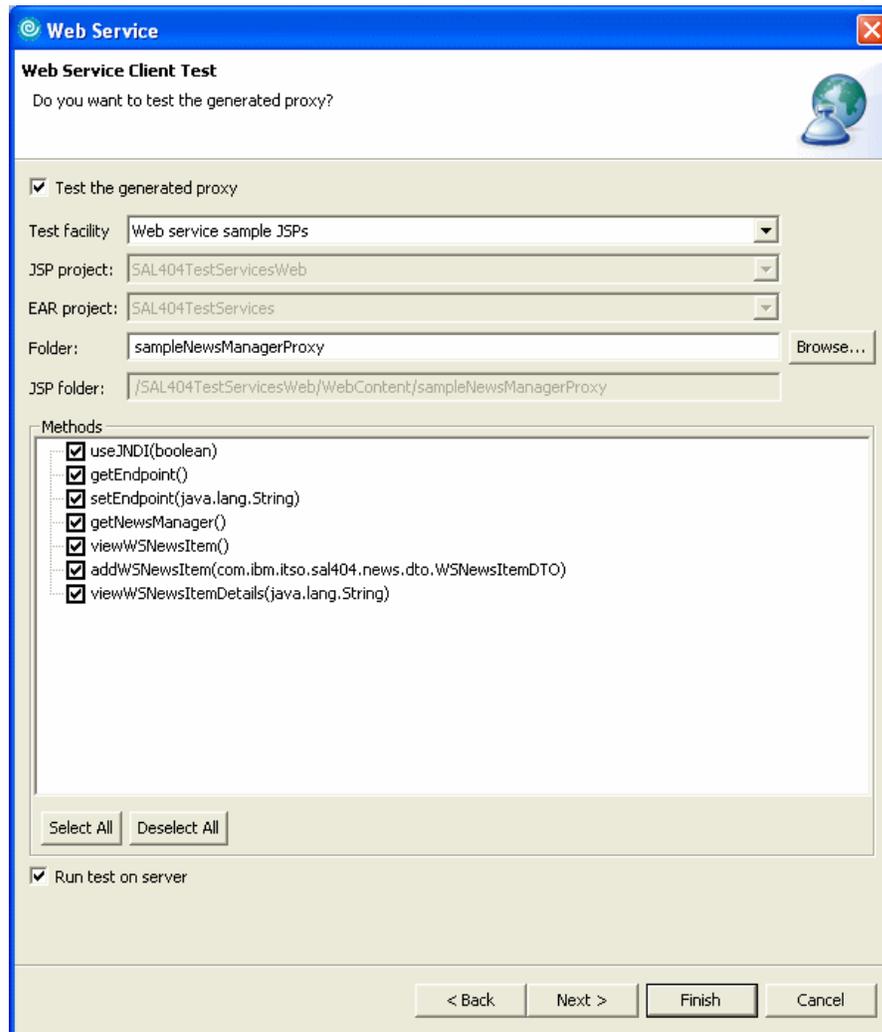
Figure 12-23   Selections for Web service client test

10.We do not need to publish our Web service so click **Finish**. See Figure 12-24 on page 510.
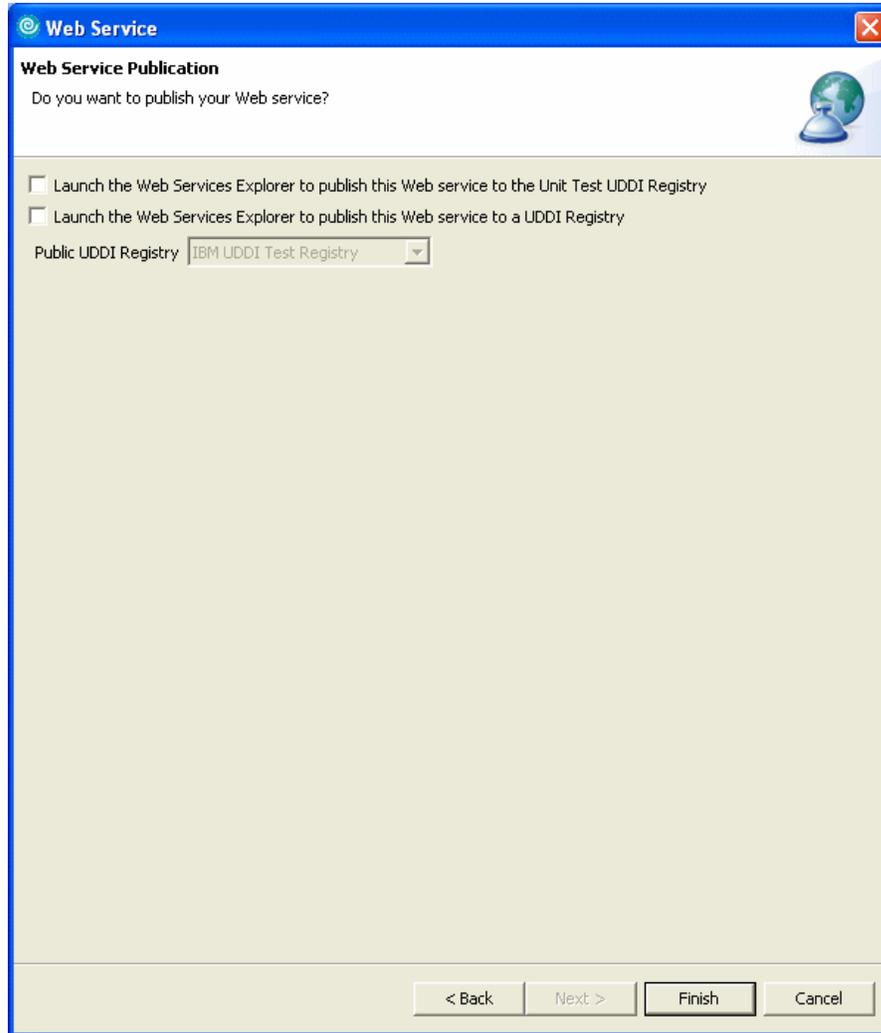
Figure 12-24   Finish Web service creation

Test the News Web services

After the Web service has been generated, the test client is launched in the internal Web browser of Rational Web Developer as shown in Figure 12-25 on page 511.
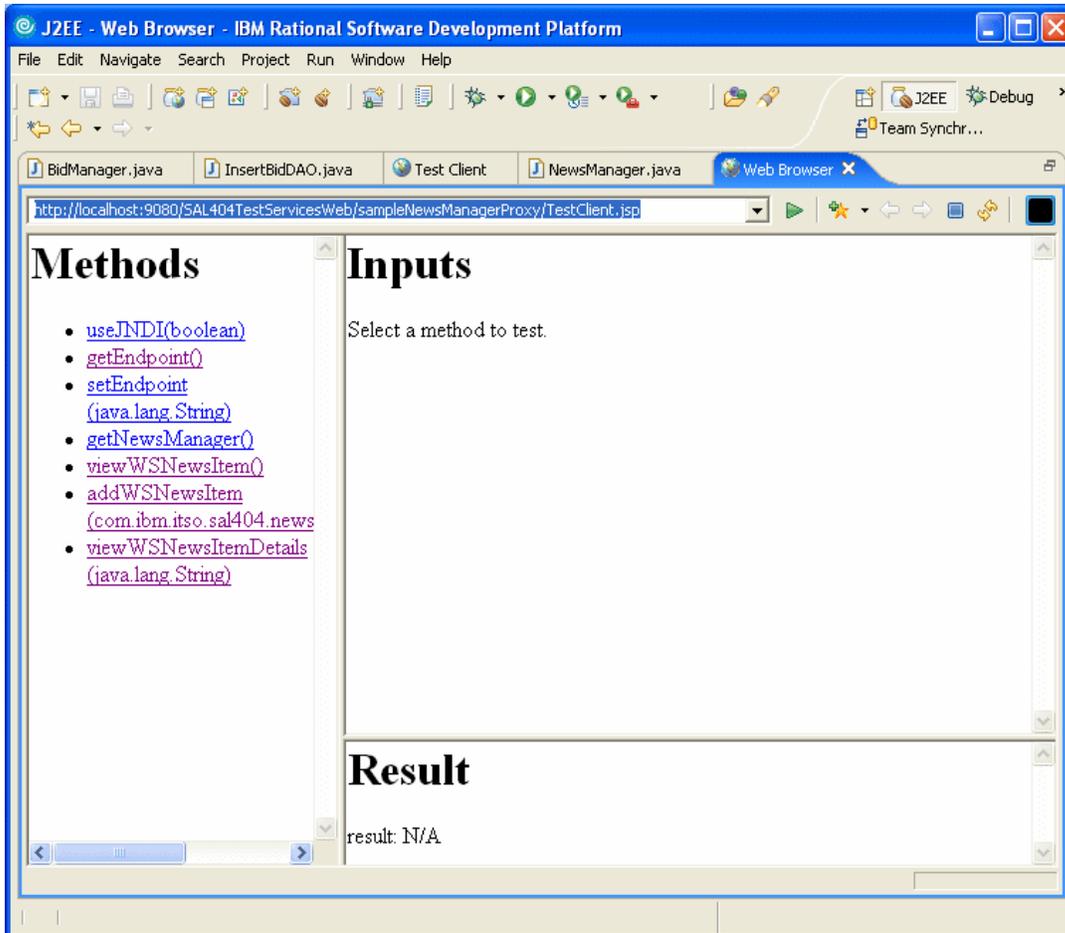
Figure 12-25   Test client for Web service

Test the key methods of the news item Web services. The steps to test the Web service are:

1. Figure 12-26 on page 512 shows a successful test of the addWSNewsItem method.
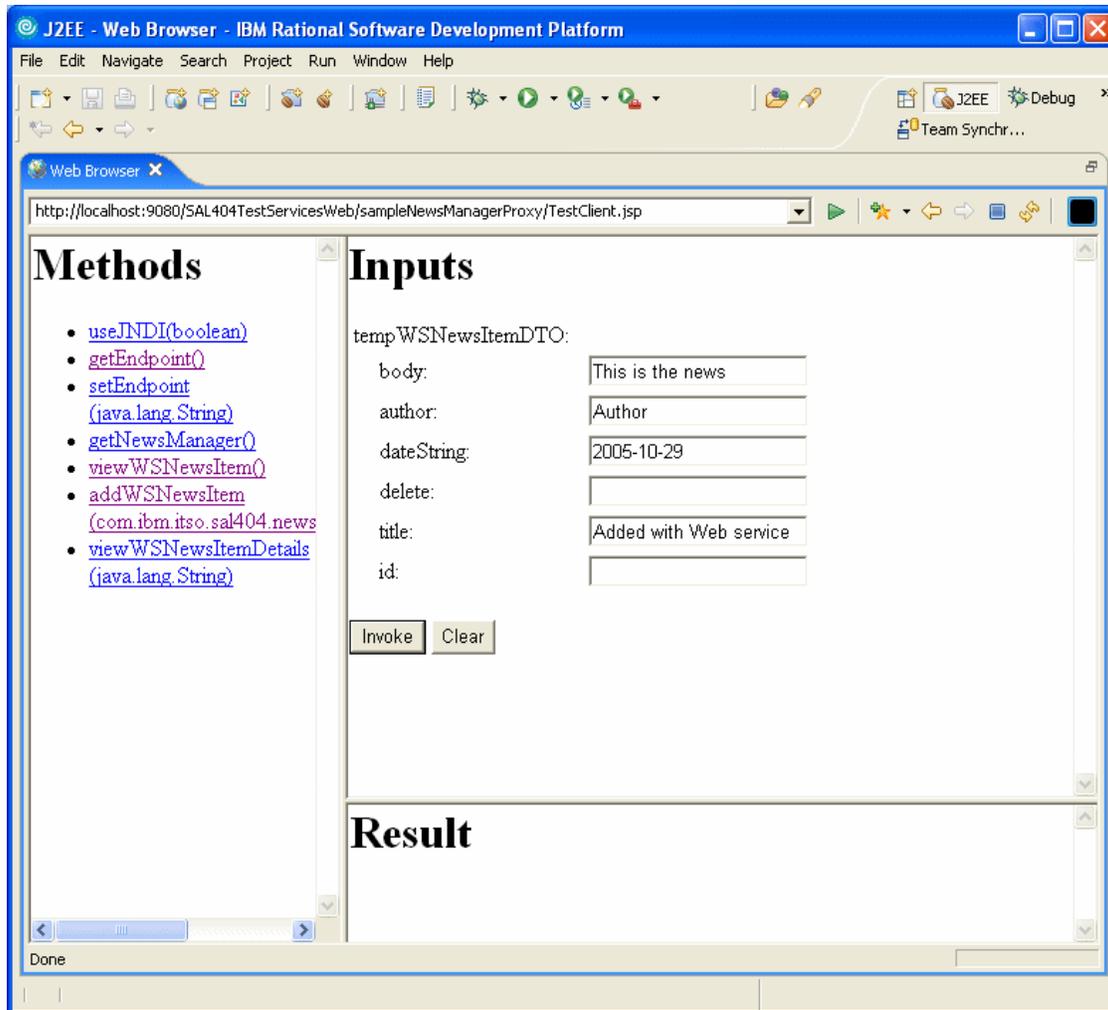
Figure 12-26   Successful test of the addWSNewsItem method

2. Figure 12-27 on page 513 shows a successful test of the viewWSNewsItemDetails method.
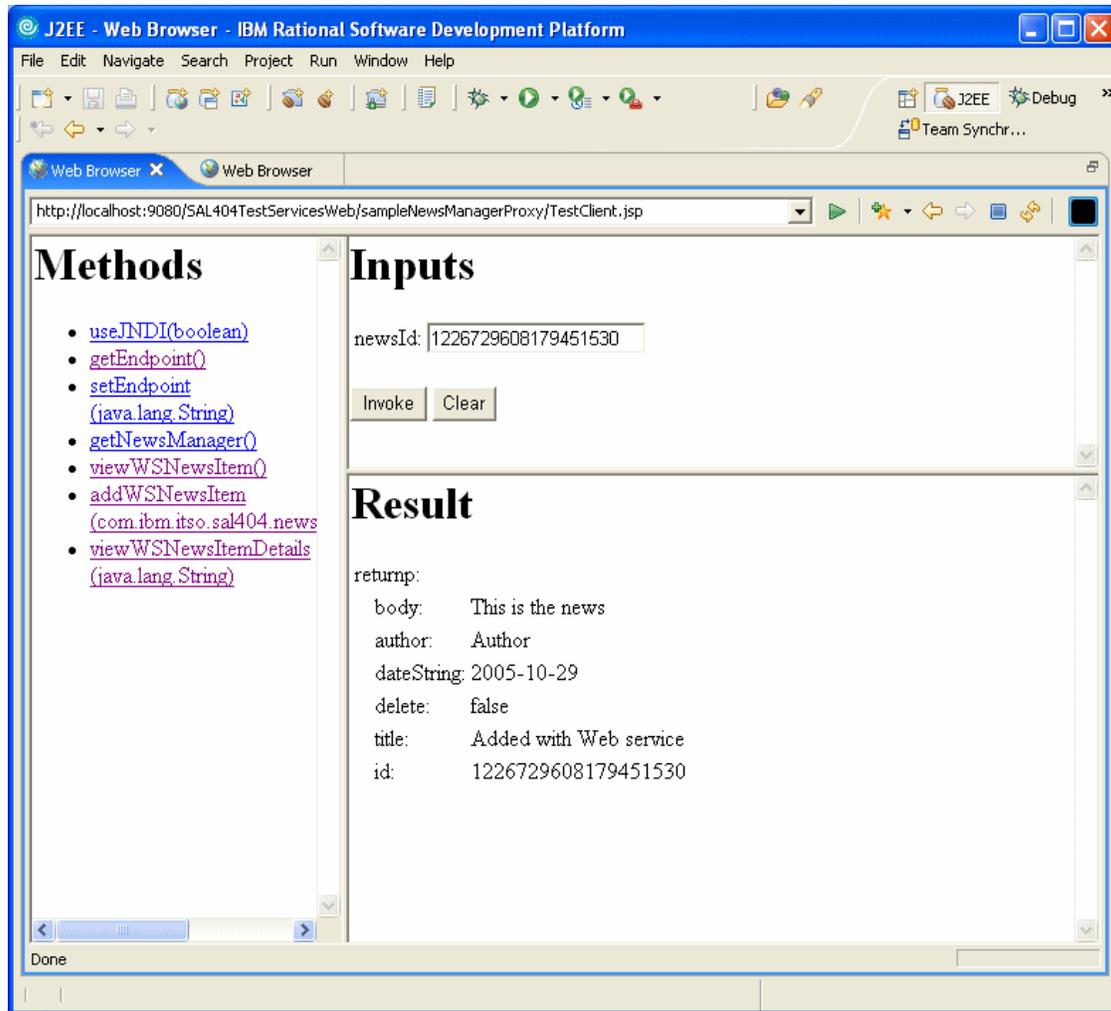
Figure 12-27   Successful test of the viewWSNewsItemDetails method

3. Figure 12-28 on page 514 shows a successful test of the viewWSNewsItem method.
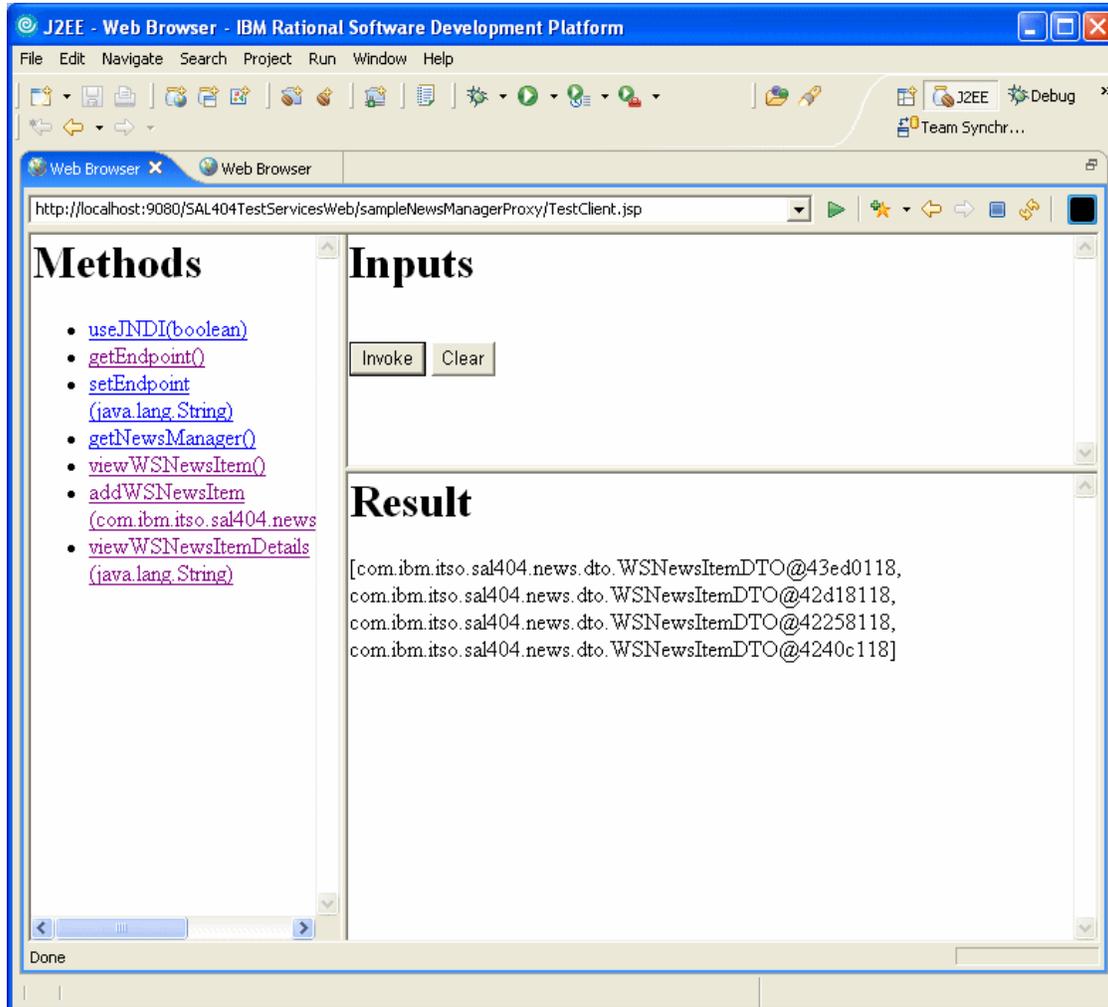
Figure 12-28   Successful test of the viewWSNewsItem method.

You can also test the new Web services with the Web Services Explorer.

Create an outbound Web service client

In this section we describe how to create a client proxy to call a Web service that is external to our Sal404 application. The sample Web service that we call is the ListingManagerService implemented in the SAL404JmsClientWeb project. The ListingManagerService has two operations:

► addListing

This operation takes an input message addListingRequest and returns an addListingResponse message.

The addListingRequest is formatted as ListingDTO complex type containing the elements shown in Example 12-2.

Example 12-2   ListingDTO type

```
<element name="listingAgent" nillable="true" type="xsd:string"/>
<element name="listingAddress" nillable="true" type="xsd:string"/>
<element name="listingDescription" nillable="true" type="xsd:string"/>
<element name="listingId" type="xsd:int"/>
<element name="listingPrice" nillable="true" type="xsd:decimal"/>
```

The addListingResponse is a String that shows the values from the input ListingDTO

► changeListing

This operation takes an input message changeListingRequest and returns an changeListingResponse message.

The changeListingRequest is formatted as ListingDTO complex type.

The changeListingResponse is a String that shows the values from the input ListingDTO

The WSDL file describing the ListingManagerService is the ListingManager.wsdl file located in the SAL404JmsClientWeb folder WebContent/WEB-INF/wsdl. The full WSDL is shown in Example 12-3.

Example 12-3   WSDL for the ListingManagerService

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
targetNamespace="http://manager.listings.sab404r.itso.ibm.com"
xmlns:impl="http://manager.listings.sab404r.itso.ibm.com"
xmlns:intf="http://manager.listings.sab404r.itso.ibm.com"
xmlns:tns2="http://dto.listings.sab404r.itso.ibm.com"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsi="http://ws-i.org/profiles/basic/1.1/xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <wsdl:types>
  <schema targetNamespace="http://dto.listings.sab404r.itso.ibm.com"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:impl="http://manager.listings.sab404r.itso.ibm.com"
xmlns:intf="http://manager.listings.sab404r.itso.ibm.com"
```

```
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
             <complexType name="ListingDTO">
              <sequence>
               <element name="listingAgent" nillable="true" type="xsd:string"/>
               <element name="listingAddress" nillable="true" type="xsd:string"/>
               <element name="listingDescription" nillable="true" type="xsd:string"/>
               <element name="listingId" type="xsd:int"/>
               <element name="listingPrice" nillable="true" type="xsd:decimal"/>
              </sequence>
             </complexType>
            </schema>
            <schema targetNamespace="http://manager.listings.sab404r.itso.ibm.com"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:impl="http://manager.listings.sab404r.itso.ibm.com"
          xmlns:intf="http://manager.listings.sab404r.itso.ibm.com"
          xmlns:tns2="http://dto.listings.sab404r.itso.ibm.com"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
             <import namespace="http://dto.listings.sab404r.itso.ibm.com"/>
             <element name="addListingResponse">
              <complexType>
               <sequence>
                <element name="addListingReturn" nillable="true" type="xsd:string"/>
               </sequence>
              </complexType>
             </element>
             <element name="changeListing">
              <complexType>
               <sequence>
                <element name="aListing" nillable="true" type="tns2:ListingDTO"/>
               </sequence>
              </complexType>
             </element>
             <element name="changeListingResponse">
              <complexType>
               <sequence>
                <element name="changeListingReturn" nillable="true" type="xsd:string"/>
               </sequence>
              </complexType>
             </element>
             <element name="addListing">
              <complexType>
               <sequence>
                <element name="aListing" nillable="true" type="tns2:ListingDTO"/>
               </sequence>
              </complexType>
             </element>
            </schema>
```

```
   </wsdl:types>

      <wsdl:message name="addListingResponse">

         <wsdl:part element="impl:addListingResponse" name="parameters"/>

      </wsdl:message>

      <wsdl:message name="changeListingRequest">

         <wsdl:part element="impl:changeListing" name="parameters"/>

      </wsdl:message>

      <wsdl:message name="changeListingResponse">

         <wsdl:part element="impl:changeListingResponse" name="parameters"/>

      </wsdl:message>

      <wsdl:message name="addListingRequest">

         <wsdl:part element="impl:addListing" name="parameters"/>

      </wsdl:message>

      <wsdl:portType name="ListingManager">

         <wsdl:operation name="addListing">

            <wsdl:input message="impl:addListingRequest"
name="addListingRequest"/>

            <wsdl:output message="impl:addListingResponse"
name="addListingResponse"/>

         </wsdl:operation>

         <wsdl:operation name="changeListing">

            <wsdl:input message="impl:changeListingRequest"
name="changeListingRequest"/>

            <wsdl:output message="impl:changeListingResponse"
name="changeListingResponse"/>

         </wsdl:operation>

      </wsdl:portType>
```

```
    <wsdl:binding name="ListingManagerSoapBinding" type="impl:ListingManager">

       <wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

       <wsdl:operation name="addListing">

          <wsdlsoap:operation soapAction=""/>

          <wsdl:input name="addListingRequest">

             <wsdlsoap:body use="literal"/>

          </wsdl:input>

          <wsdl:output name="addListingResponse">

             <wsdlsoap:body use="literal"/>

          </wsdl:output>

       </wsdl:operation>

       <wsdl:operation name="changeListing">

          <wsdlsoap:operation soapAction=""/>

          <wsdl:input name="changeListingRequest">

             <wsdlsoap:body use="literal"/>

          </wsdl:input>

          <wsdl:output name="changeListingResponse">

             <wsdlsoap:body use="literal"/>

          </wsdl:output>

       </wsdl:operation>

    </wsdl:binding>

    <wsdl:service name="ListingManagerService">

       <wsdl:port binding="impl:ListingManagerSoapBinding"
name="ListingManager">
```

```
        <wsdlsoap:address
location="http://localhost:9080/AgencyWeb/services/ListingManager"/>

      </wsdl:port>

    </wsdl:service>

</wsdl:definitions>
```

The ListingManagerService was created by using the Rational Web Developer to build a Web service for the existing Java class ListingManager found in the SAL404JmsClientWeb project in package com.ibm.itso.sab404r.listings.manager.

The ListingManagerService is our simple example of how a real estate listing company might provide a way for their customers to add and change properties that they have for sale. We want any property added or changed by our Sal404 application to also be listed by the real estate listing company that provides the ListingManagerService.

To create a client to call the ListingManagerService from the Sal404 application the steps are:

1. Obtain a WSDL file describing the ListingManagerService. In the real world we might obtain such a WSDL file by looking up a UDDI repository, by using WSIL, or directly from the company that provides the ListingManagerService. But in our simple example we used Rational Web Developer the copy the ListingManager.wsdl file from the SAL404JmsClientWeb project to the SAL404RealtyJava project.

2. Selec**t ListingManager.wsdl** file in the SAL404RealtyJava project, right-click, and choose **Web Services** → **Generate Client** as shown in Figure 12-29 on page 520.
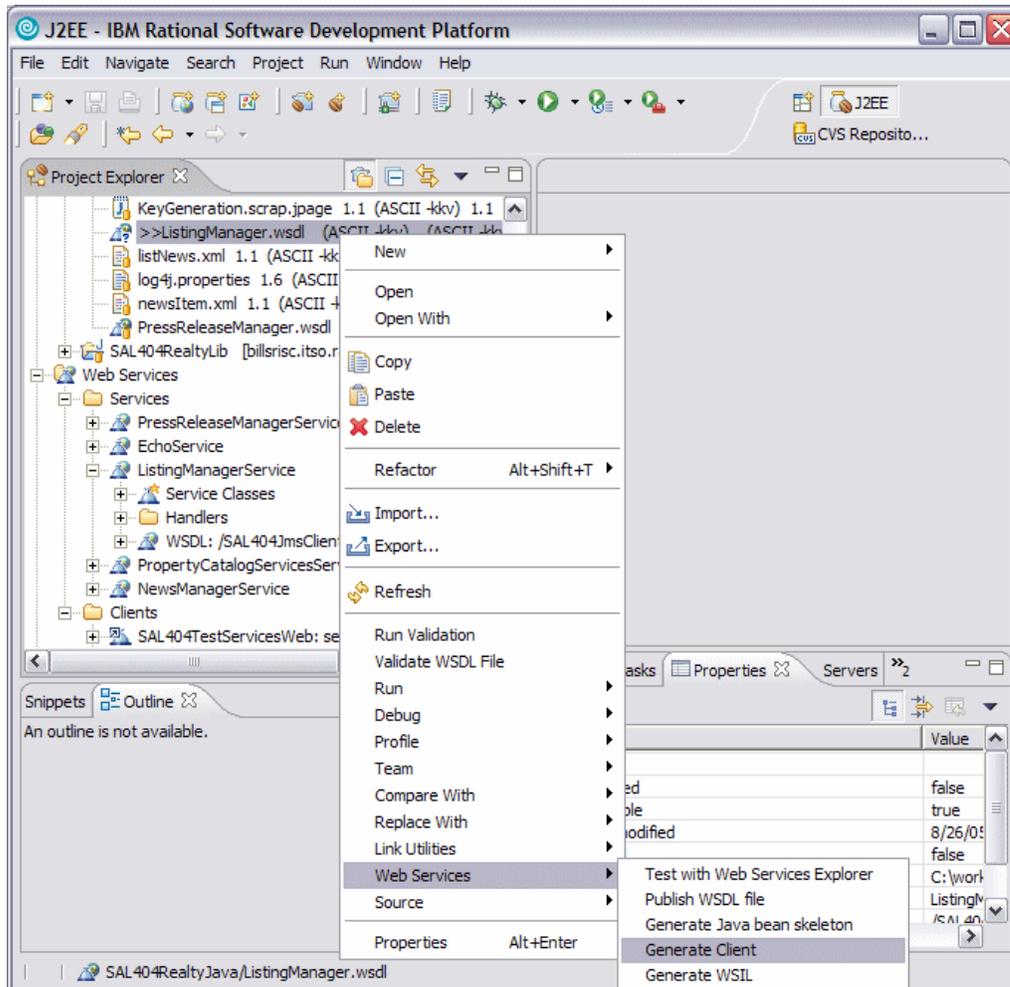
Figure 12-29   Create a Web service client

3. We do not need to test the generated client at this stage, so accept the defaults on the client proxy page as shown in Figure 12-30 on page 521 and click **Next**.
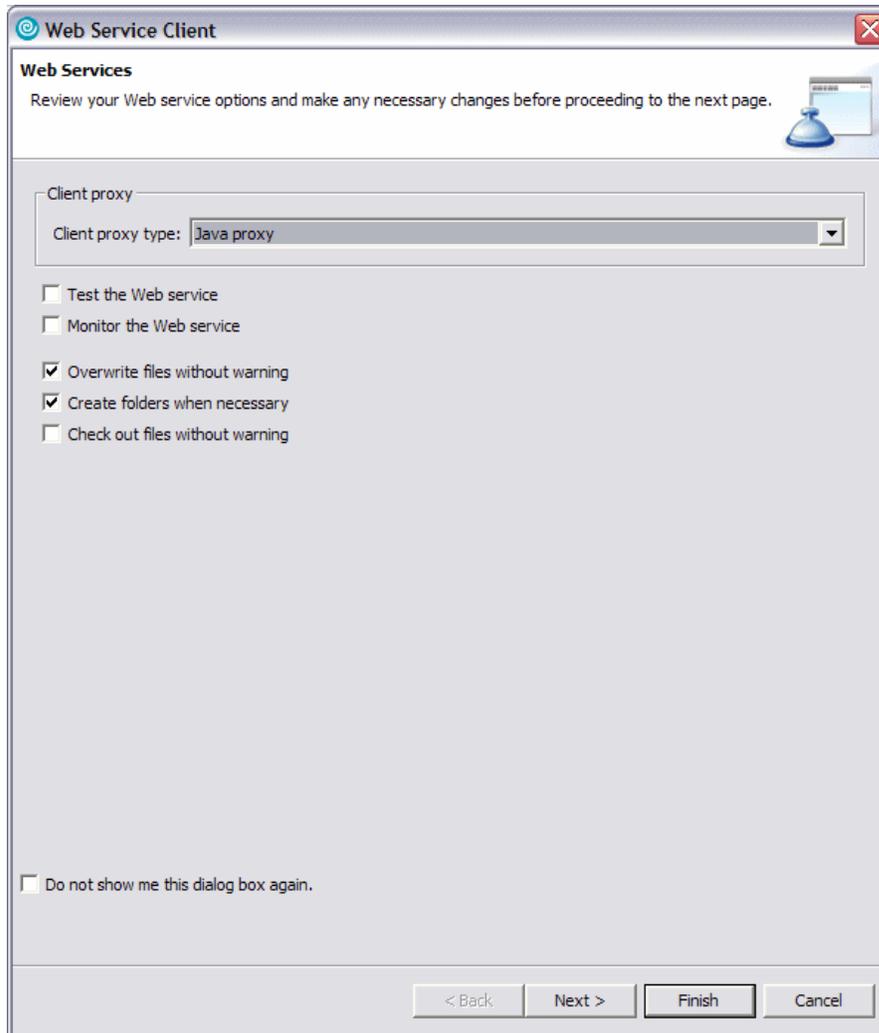
Figure 12-30   Client proxy page

4. Our ListingManager.wsdl file should be already listed as shown in Figure 12-31 on page 522, so we click **Next**.
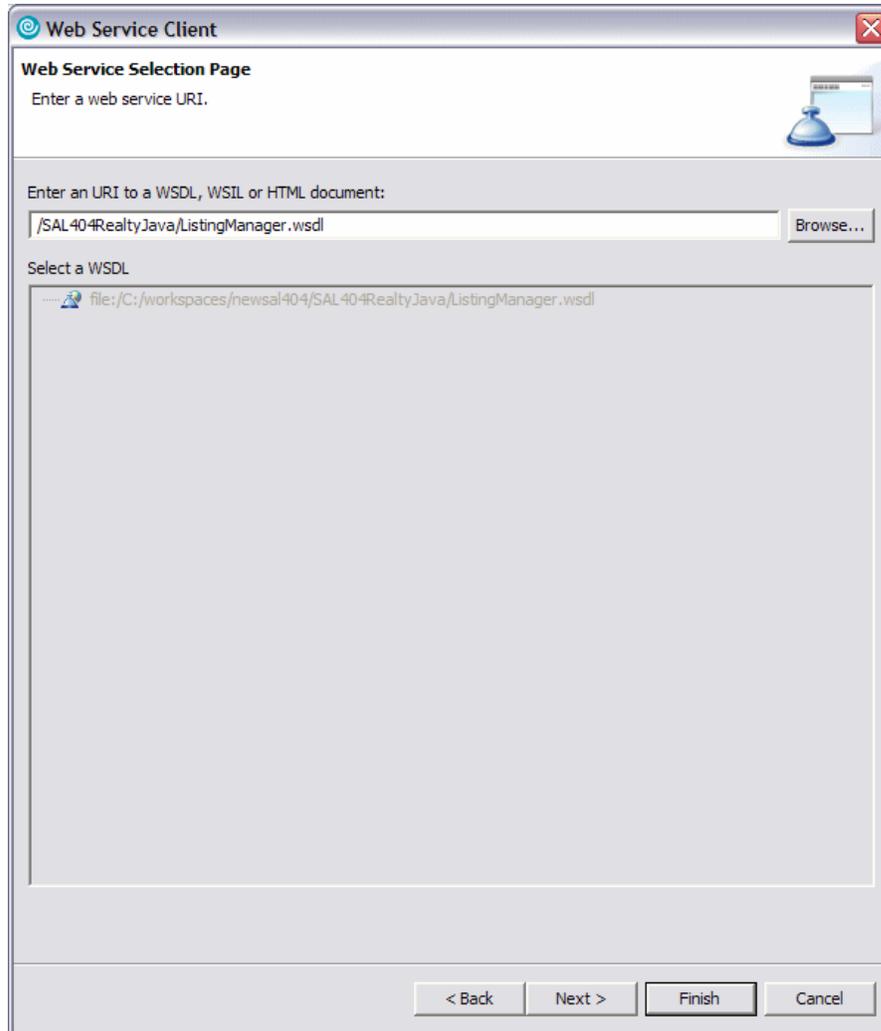
Figure 12-31   Choose WSDL for client

5. Choose a client type of **Java** and place the code in the **SAL404RealtyJava** project as shown in Figure 12-32 on page 523. Click **Next**.
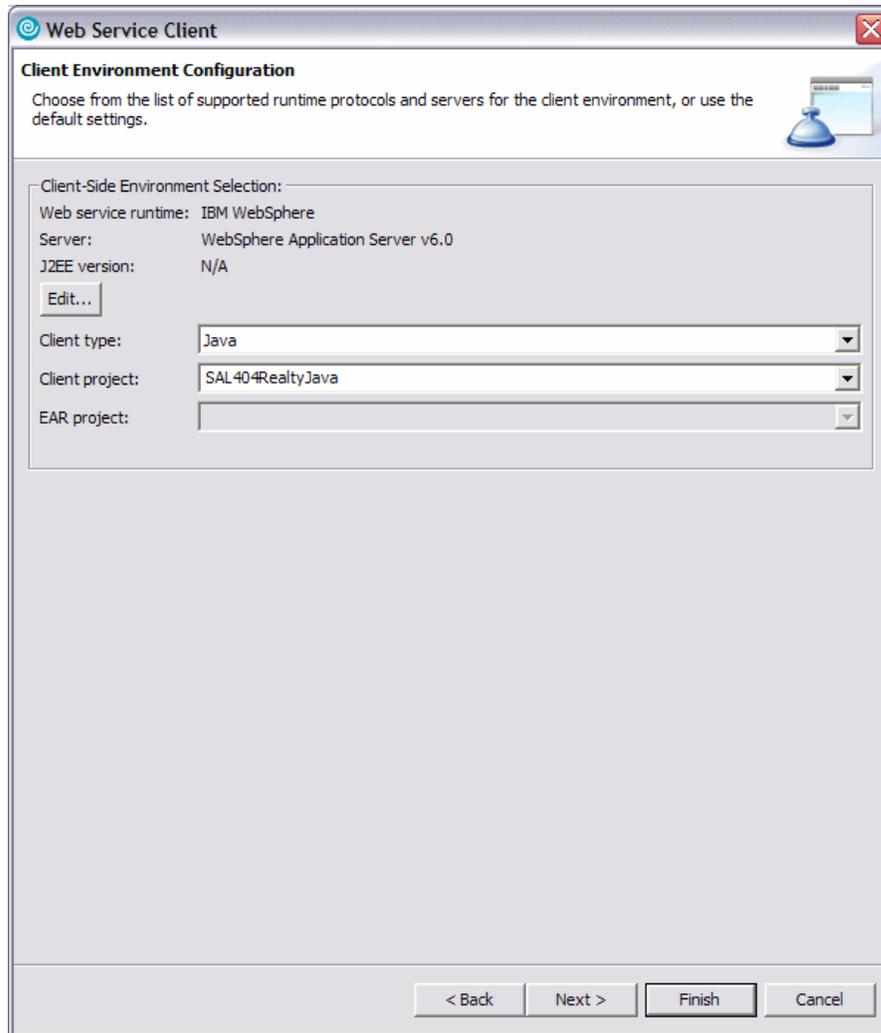
Figure 12-32   Generate a Java Web service client

6. We do not need any security configuration so click **Finish** to generate the Web service client code. See Figure 12-33 on page 524.
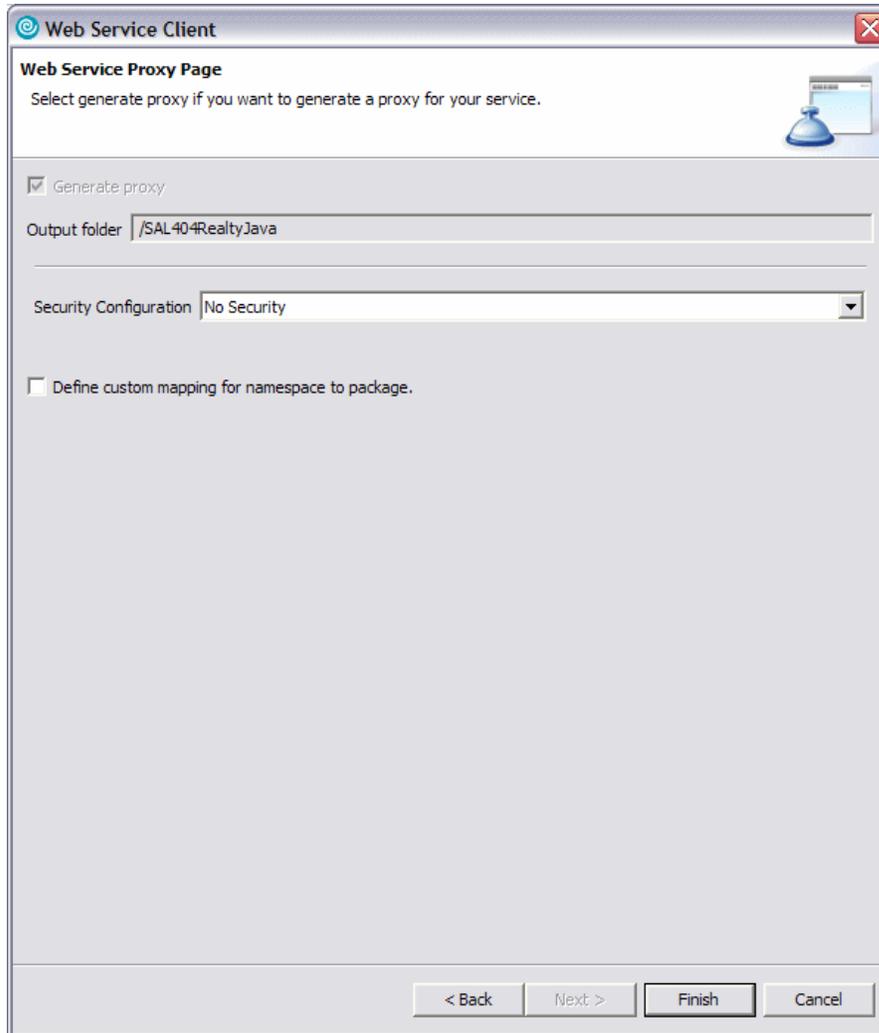
Figure 12-33   Finish Web service client

The Web service client wizard creates a proxy called ListingManagerProxy in the
com.ibm.itso.sab404r.listings.manager package of the SAL404RealtyJava
project. Other support code is also created in the
com.ibm.itso.sab404r.listings.manager and com.ibm.itso.sab404r.listings.dto
packages.

Example 12-4 on page 525 shows the addListing method of the
ListingManagerProxy class which call to access the ListingManagerService

addListing operation. A changeListing method in the proxy is also created to allow us to call the changeListing operation.

Example 12-4   addListing method of the proxy

```
public java.lang.String addListing(com.ibm.itso.sab404r.listings.dto.ListingDTO
aListing) throws java.rmi.RemoteException{
  if (listingManager == null)
    _initListingManagerProxy();
  return listingManager.addListing(aListing);
}
```

To test that the generated proxy worked we created sample calling code in a Rational Web Developer scrapbook page JavaTesting.jpage in the SAL404RealtyJava project. This test code is shown in Example 12-5.

Example 12-5   Test the ListingManagerProxy

```
ListingManagerProxy listProxy = new ListingManagerProxy();
ListingDTO listDto = new ListingDTO();
listDto.setListingAddress("An address");
listDto.setListingAgent("The agent");
listDto.setListingDescription("A great property");
listDto.setListingId(111);
listDto.setListingPrice(new java.math.BigDecimal(34567.90));
listProxy.addListing(listDto);
```

To run this test code in the scrapbook, you can select all the code, right -click and choose **Display**. A successful test returns a string similar to that shown in Example 12-6.

Example 12-6   Results of testing the ListingManagerProxy

```
(java.lang.String)
com.ibm.itso.sab404r.listings.dto.ListingDTO@64510f42*****Listing id is
111*****Listing agent is The agent*****Listing address is An
address*****Listing price is
34567.9000000000014551915228366851806640625*****Listing description is A great
property
```

To integrate a call to ListingManagerProxy into our Sal404 application we did the following:

1. Created a new DAO called ListingServiceDAO in the com.ibm.itso.sal404.propertycatalog.dao package of the SAL404RealtyJava project.

2. Created an addListing method in the ListingServiceDAO to call the addListing method of a ListingManagerProxy. See Example 12-7.

Example 12-7   addListing method in the ListingServiceDAO

```
public java.lang.String addListing(ListingDTO aListing)
         throws RemoteException

  {
      ListingManagerProxy listProxy = new ListingManagerProxy();
      return listProxy.addListing(aListing);
  }
```

3. Created a changeListing method in the ListingServiceDAO to call the changeListing method of a ListingManagerProxy.

4. Created a method called addToListingService in the PropertyCatalogManager class that creates a ListingDTO from an input PropertyDTO and then calls the addListing method of a ListingServiceDAO.

5. For testing purposes, we added a call to the addToListingService method in the addPropertyDetails method of the PropertyCatalogManager class.

> **Note:** In the code we shipped with our redbook additional material we have commented out the call to the addToListingService method. This enables you to install and test the sal404 application without needing to have the SAL404JmsClientWeb application and the ListingManagerService installed.

6. Created a method called changeListingService in the PropertyCatalogManager class that creates a ListingDTO from an input PropertyDTO and then calls the changeListing method of a ListingServiceDAO.

7. For testing purposes we added a call to the changeListingService method in the modifyPropertyDetails method of the PropertyCatalogManager class. This has also been commented out in the code in our Redbook additional material.

13

Database design

In this chapter we discuss the existing data model and SQL queries inherited from the sample application built for the redbook *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446.

We then build an improved data model and discuss the advantages this would have if used in our sample application. We did not implement this new data model in our new Sal404 application because of time constraints during our redbook project. Using the new model would have meant rewriting the entire application, so we decided to modify the old data model only where it was necessary to implement new functionality. This allowed us to be sure that the other components would work seamlessly with the components that implement the new application functionality. We provide an outline of what we changed and why and discuss how this helps the new functionality of our application.

This chapter also looks at the tooling support for implementing a database backend as part of the development process. We discuss SQL and database features such as indexes and triggers and so on, and how they should be used with the sample application.

13.1 Database features

For an online application to be functional, it needs a means to persist data including both business and end user data. Persisting data is not enough, because the application also needs a way easily and seamlessly to retrieve and manipulate the stored data. There are many data stores that applications can utilize, but for out application we are particularly interested in relational databases. WebSphere Application Server - Express supports many database platforms. In our sample application, we use IBM DB2 Universal Database V8.2, which is a powerful relational database management system provided with different editions and features and ships with WebSphere Application Server - Express V6.

We now detail some database features that should be used for developing database applications. These features are common to relational database platforms, but there can be differences where specific platform implementations are concerned, so you need to be careful of database specific features.

► Database triggers

A *trigger* defines an action or set of actions that are applied or performed when certain operations are enacted against a table within a database. Operations here include delete, update and insert operations. For example, you can specify a query or procedure that is run when an application inserts a record into a table. Our sample application uses of a single trigger that deletes an interest list item when the corresponding interest list is deleted. See the application database creation script shipped with our redbook sample material for the specifics of how this trigger is created.

► Identity column

This is a database management feature that automatically generates a unique numeric value for a particular column in a particular table for every insert. To understand this better, consider that when an application inserts a new row of data into a database table, the application will usually supply values for the different columns. However for identity columns, the database management system automatically generates the value for the columns so the application does not need to supply a value for the identity columns during an insert operation. Our sample database makes extensive use of identity columns. For larger databases, it can be beneficial to use a database table for generating the unique values, where the table is used to generate the next logical numeric value as needed. This method allows for greater flexibility, for example, character values could be appended to the generated numeric values to make more eye catching values.

► Constraints

A database constraint is a rule enforced by the database management system. It usually applies to column level data. There are numerous kinds of constraints that enforce different rules including:

– Unique constraint

When this kind of constraint is defined on one or more columns, it prevents duplicate values in those columns. This is to ensure that no two columns within the database table will have the same value. This provides the capability of enforcing business rules in the database and not in the application logic.

– Table check constraint

When defined, this constraint restricts the type or value of data inserted into a column. For example, you can define a check constraint for a column of type integer, that specifies that values entered must be between 0 and 100. When an attempt is made to insert values that are less or more than the specified range the database managements system rejects the insert by throwing an exception.

– Referential constraint

When defined, this constraint enforces a primary to secondary key relationship where column data in a table must be present in another column in a different table. Our sample database makes extensive use of this type of constraint.

– Primary key

This constraint is similar to the unique key constraint. The only difference is that there can only be one primary key per database table, although one or more columns can make up the primary key. When more than one column makes up the primary key, it is called a *composite primary key*. All tables in our sample application database have primary keys defined and some have composite primary keys.

– Index

An *index* is an ordered set of pointers to rows in a particular database table. It is calculated based on the values of data in one or more columns and an index is a separate database object from the data that is stored in the table. Indexes are primarily used to improve database performance, by making it easy for the database management system to find data when a query is issued. Note that having indexes that are not needed or used might adversely affect performance in a domain where large inserts are common. This is because the database management system must build and maintain the index as inserts occur. Our sample application database contains many indexes.

– Routines

A *routine* is a database program that consist of SQL and database specific logic. A great deal of benefit can be achieved by moving application logic into routines especially when database interaction forms part of the application logic. The decision to move application logic into routines is dependent on the problem domain. It might not always be the best solution, especially when considering database performance. Routines fall into one of the following types:

• Stored procedure
• User defined functions
• Methods

Our sample database does not make use of routines.

13.2 The Sal301 data model

Figure 13-1 shows an entity relationship diagram (ERD) of the sample database used in the Sal301 application built for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301. The diagram outlines the tables that make up the sample application database.
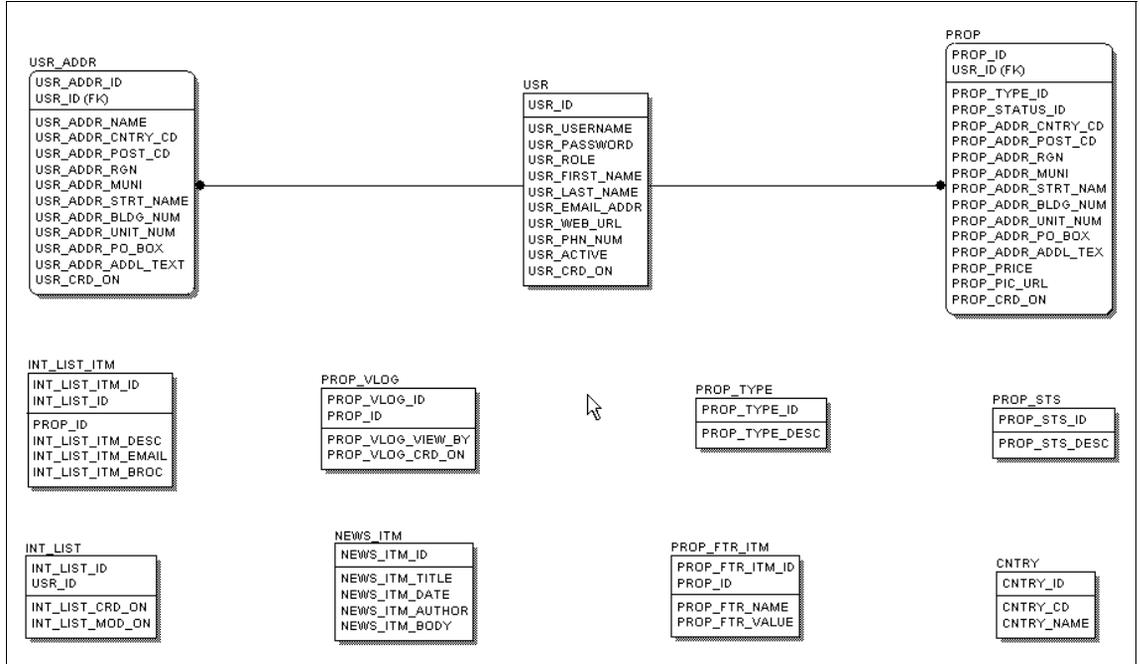
Figure 13-1   The Sal301 data model

As you can see in Figure 13-1 on page 530, there are two explicit entity relationships that involve the user_id in the PROP and USR_ADDR tables. The effect of the relationships is that the database management system enforces a rule so that when data is inserted in the PROP table, if a user_id is supplied which does not exist in the USR table, the database management system throws an exception. All other relationships are implicit. They are used within the application, but are not enforced by database constraints. This is not an ideal solution because it leaves open the possibility of having data redundancy, it also makes it difficult to query the data and usually results in business rules that are more difficult to enforce.

The different tables that make up the database shown in Figure 13-1 on page 530 are:

► USR

This table holds user personal information. These are registered users who can login and access the system functionality.

► USR_ADDR

This table holds addresses for users in the USR table.

► PROP

This table is used to hold property information.

► PROP_STS

This table is used to indicate the status of a property

► PROP_TYPE

This table is used to indicate a property type

► PROP_VLOG

This table is not used or referenced anywhere in the Sal301 application.

► PROP_FTR_ITM

This table is not used or referenced anywhere in the Sal301 application

► INT_LST

This tables is used to indicate the properties a user is interested in, and it points to the INT_LST_ITM table

► INT_LST_ITM

This table represents an interest list, which is a list of properties a user is interested in.

► CNTRY

This table is used to represent different countries.

► NEWS_ITM

This table is used to represent a news item.

A script to create this database was provided with the additional material for the redbook *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301. We have included a copy of this script in the additional material for our current redbook. The copied script is named OLDDBSCRIPT.SQL.

13.3 The new data model

We developed a new data model for numerous reasons, but primarily to capture the new functionality offered by our sample application. We also wanted to improve the Sal301 model to reduce data redundancy and streamline the process of querying the database and enforcing business rules. Due to the time constraints of our redbook project, we did not use the improved model in our sample application. Instead, we modified the old model. You can use the tools in Rational Application Developer to capture your data model.

The ideal new model consists of eleven tables. We removed PROP_FTR_ITM and PROP_VLOG and added ROLE and PROP_BID tables. We also made the table and attribute names more meaningful. This is a good practice that helps make the database easier to understand, especially when the original database designer might have moved on, or when new developers join the project. New developers can have a better chance of understanding for what purpose the tables are used. Some database design tools even provide the ability for the data model designer to include explanatory notes within the model design, and we recommend that this is always very useful when available.

The two new tables added to the data model are:

► ROLE

This table is used to hold user roles. A user role can be considered as an authority level, where users see different UI functionality dependent on their role. With this table it will be easy to add and remove roles within the application as needed. New roles can be created and added in the system if new functionality requires new roles.

► PROPERTY_BID

This table is used to hold bids made on a property. Our business rule suggests that only one bid can be made on a property, although if a bid is rejected, all customers are allowed to place bids. This rule is not meant to reflect real business conditions and from a functionally point of view is not very practical, but our noted that the sample application is for demonstration

purposes and we do not aim to model a real life scenario. A bid always has a user associated with it, the user is taken from the USR table so this means that a user must be registered before they can place a bid.

Other changes made include:

► In the NEWS_ITEM tables, the news author is now taken from the USR table, meaning an author must also be a registered use, the old design did not enforce this.

► In the Sal301 model, a user address was taken from the USR_ADDR table but a property address was hard-coded in the property table. We changed the new design to include a single address table which now holds addresses for both users and properties. In the future, if additional database objects are added that require some sort of address field, the address table can be used.

► The address table now has a country associated with it, this is taken from the COUNTRY table.

► We also created a number of indexes which make searching the database less expensive when database performance is considered. See the database creation script in our redbook additional material, for more information about the created indexes.

► The Sal301 data model did not explicitly define foreign keys although they were logically used in the application SQL statements. We changed this by the definition of explicit foreign keys in our new design.

A database script that will create our new database model called NEWDBSCRIPT.SQL can be found in our redbook additional material.

Note: Due to time pressures, we did not use the new data model with our sample application. We used a modified version of the Sal301 model with the minimum changes necessary for us to support the new functionality that allows users to bid on a property. A database script that creates this database model called DBSCRIPT.SQL can be found in our redbook additional material.

We added a bid table called BID, which is used to hold bids made on a particular property. A user and property is always associated with a bid, the association indicates who made the bid and on what property the bid was placed. The BID_STAUS is used to indicate the status of a particular bid, consult the database creation script for the different bid statuses. The bid status directly affects the property, that is where a particular property is in its property life cycle. Example 13-1 on page 534 shows the definition of the BID table. We also added a PROP_SELLER column to the PROP table to hold the user id of the property owner and inserted a new status description. Offers. to the PROP_STS table.

Example 13-1   BID table

```
CREATE TABLE BID (
      BID_ID          CHAR(36) NOT NULL,
      BID_PROP_ID     INTEGER NOT NULL,
      BID_USR_ID      INTEGER not null,
      BID_STATUS      CHAR(10) not null,
      BID_PROP_SELLER INTEGER not null,
      BID_CRD_ON      TIMESTAMP NOT NULL,
      BID_PRICE       DECIMAL(10, 2) NOT NULL
);

ALTER TABLE BID
      ADD PRIMARY KEY (BID_ID, BID_PROP_ID, BID_USR_ID);
```

14

Code standards and quality

This chapter discusses some ways in which code standards can help improve the quality of an application.

Good code standards make it easier for you and others to:

► Read the code.
► Maintain the code.
► Implement new functionality following your design and conventions.
► Implement a seamless design common for all your software components.
► Avoid errors.

We discuss standards and how we used them in our application. We show how and where Rational Web Developer can help to implement and verify the standards.

535

14.1 Coding guidelines

When you agree to have common coding guidelines in your development team and you have a set of rules, often the most difficult job is to make sure that everyone on your team follows the rules. The benefit of having common coding guidelines rises and falls depending on whether everybody on the team is conforming to those guidelines.

So you should ask three questions:

1. What are my rules?
2. Where and how do I write down those rules?
3. How can I make sure that those rules are followed?

The first two questions you have to discuss and answer in your development team. If you use a methodology to run your project, there will be most likely an approach to write your coding guidelines down in some document. Depending on the size of the development team, this might be done in workshops within the team or there might be an application architect or lead developer who defines the coding guidelines. At the very least, make sure that someone writes down the rules and every one on the team agrees on the rules.

You can help make sure that your rules are followed by using the facilities of Rational Web Developer. While Rational Web Developer cannot support every rule that you might setup, it does support enforcing many rules that are common when developing J2EE or Java applications. Rational Web Developer has a whole set of preset templates and rules that are common to these fields.

14.2 Common rules

Each file that is created manually during the development process needs comments that declare what is the purpose of the file or of the particular piece of code. These comments should always be placed at the beginning of the file so they are immediately seen when the file is opened.

It is your decision what information you want to include in the comments, but it is a common practice to include at least the following:

► Author

Who is the owner of this code or file? This does not mean that no one else in a development team can edit the code, but it gives information about the person who is responsible and to whom you can report any bugs in the code.

► Creation date

This date is the date when the file was first created. This does not change during the development process.

► Latest change

This indicates the date and time when the file was changed. If you use a configuration management tool such as CVS, it can support you by automatically filling in this information. Also you can use CVS to get the information about the person who made this last change.

► History log

This gives information about the changes to the file and who made the changes. You can get support from CVS to fill in these values. This is something that can heavily used within your development team. The history log can get very long over time, especially in central components of your code that are changed a lot.

Note: If you want CVS to fill in the person who made the last change and note this in the history log, it is a good idea to have meaningful user names for the CVS access, because these are the names CVS uses to fill the values such as Id and Log. So, if you can influence this, do not use IDs such as `dev0815` for accessing CVS, but use `your name` as is known in the development team.

The author should always include his or her full name. In addition, if you plan that each developer directly supports their own code (a common practice in open source projects), then you should also include an e-mail address.

14.2.1 Setup basic code templates for Java

For Java files, Rational Web Developer lets you specify templates that make these tasks easy. To set up the code templates our the common rules, follow these steps:

1. Choose **Window** → **Preferences** → **Java** → **Code Style** → **Code Templates.**

2. Click **Code** → **New Java files** as shown in Figure 14-1 on page 538.
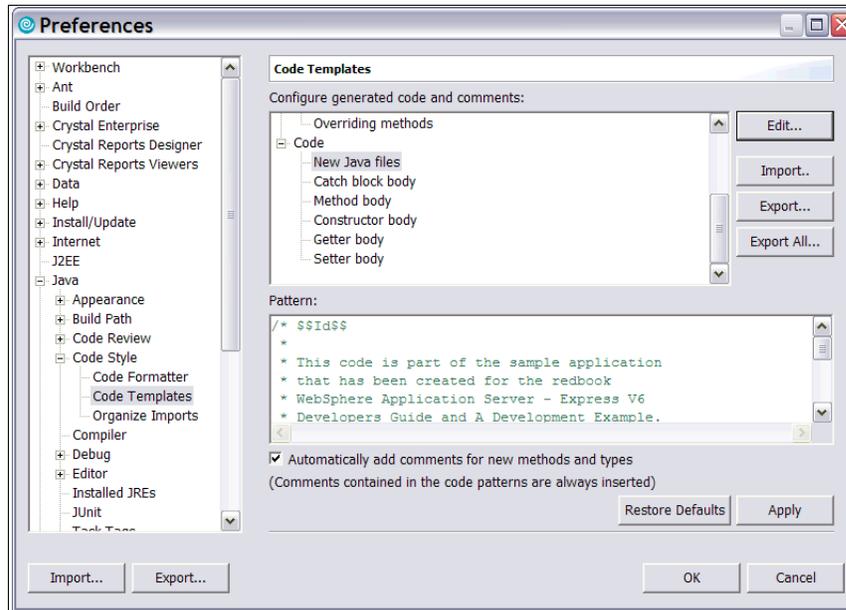
Figure 14-1   Code template for java (block-) comment in new java files

3. Click **Edit... .**
4. Enter the example text shown in Example 14-1.

> **Note:** Dollar signs are the directive for variables in templates of Rational Web Developer. Therefore the dollar has to be escaped when it is used as a character.

Example 14-1   Code template for new java files

```
/* $$Id$$
 *
 * This code is part of the sample application that has been created
 * for the redbook WebSphere Application Server - Express V6
 * Developers Guide and A Development Example (SG24-6500).
 *
 * Revision History:
 *
 *    $$Log$$
 */
${package_declaration}

${typecomment}
${type_declaration}
```

Figure 14-2 shows a view of the new code template.
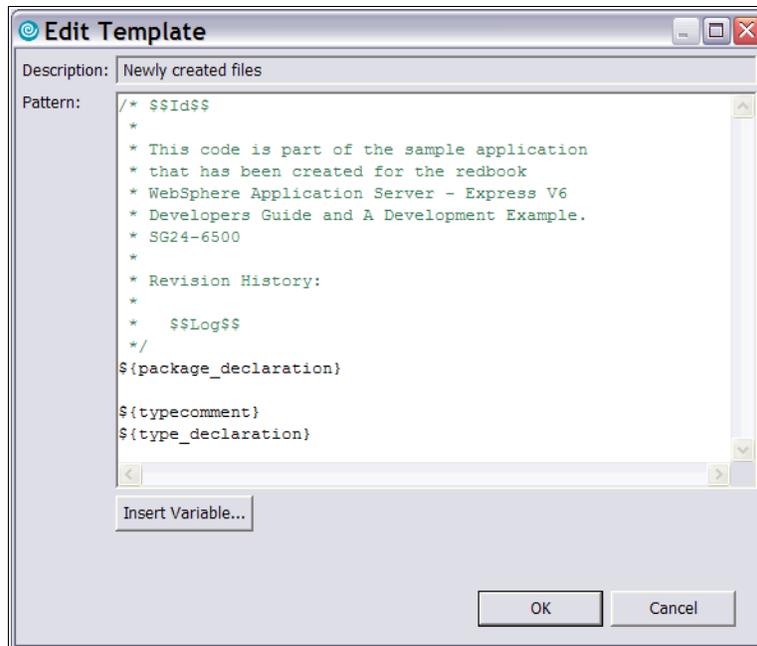
Figure 14-2   Edit the code template for new Java files

5. Click **OK** and **Apply.**

6. Now select **Comments** → **Types** to edit the Javadoc™ comment for new types.

7. Click **Edit...** .

8. Enter the text shown in Example 14-2.

Example 14-2   Code template for Javadoc in classes and interfaces

```
/**
 * One sentence about what this class or interface is doing ending with a dot.
 *
 * More detailed, arbitrarily elaborate description and information.
 *
 * <BR><BR>
 * Creation date: ${date}
 * @author ${user}
 */
```

14.2.2 CVS keyword substitution settings

To make sure that you use the CVS keyword expansion feature go to **Window** → **Preferences** → **Team** → **CVS** and make sure that **ASCII with keyword expansion (-kkv)** is selected, as shown in Figure 14-3.
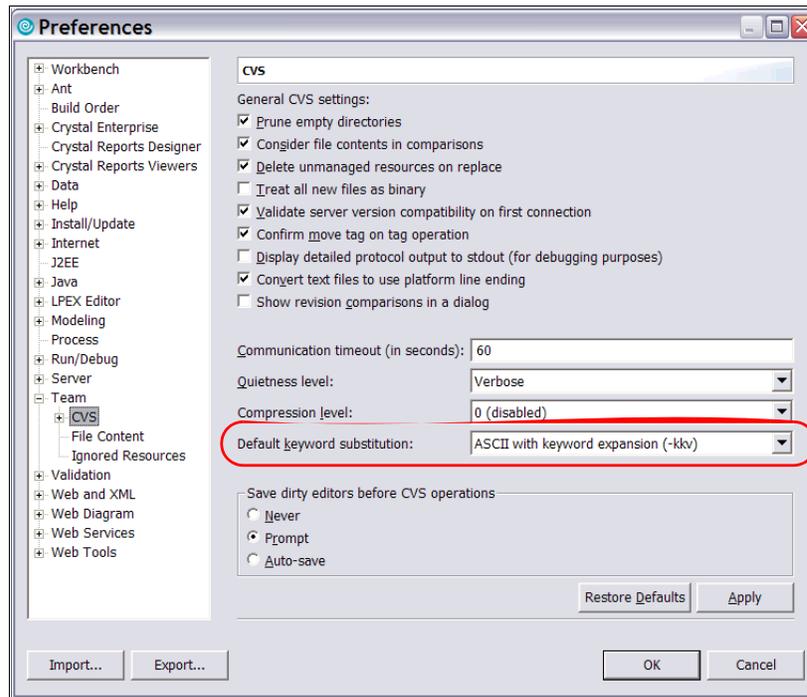
Figure 14-3   Set the keyword substitution feature of CVS

The kkv option is the default option for keyword substitution. If you use explicit locking (usually the case when you are editing documents, not code) then you should use the kkvl option that inserts the name of the person locking the file into the keyword string. This makes sense, only if `cvs admin -l` is used.

Refer to the CVS documentation for further information about keyword substitution and the substitution modes. Information about the latest stable release of CVS can be found at this Web site:

`http://ximbiot.com/cvs/wiki/index.php?title=CVS--Concurrent_Versions_System_v1.12.12.1`

14.3 Structure

In the following sections we discuss some of the issues you need to consider t when structuring a WebSphere Application Server - Express application.

14.3.1 How to organize your projects

When you are thinking about organizing the structure of your projects and folders, there are two major considerations:

► Application build
► Deployment of your application

Also consider whether your project structure works well for when you want to ship your application. The structure should be easily understood by your customers, or by a community and its users.

For the build time you want to have all the resources available for editing and compiling your code so that you can work effectively in a team. The code and other resources have to be quickly accessible, easy to edit and compile without time consuming reorganizing, configuration or packaging. Further, the structure should be easily understood. For that reason it might be a good idea to set up the structure using standards or commonly used practices.

For deployment, it is easier when your structure is close to the organization that the target infrastructure understands. For J2EE applications there are strict standards regarding the structure which your application server understands.

If you use Rational Web Developer, you get support for organizing your projects according to these standards. When you create a new enterprise application project or Web project, Rational Web Developer automatically creates a folder structure that is compliant with the standards.

However, be aware that this structure is technology-oriented. If you want to organize your projects according to your component structure or using layered design, you have to take special care not to mix up the technologies with your design components. This can be a tricky task, especially if you want to use the IDE for both development and deployment of your application.

For example, if your design uses a layered design you might like to separate the different layers into single project folders in Rational Web Developer. However, this can be quite hard. For example, consider an application where you have both hand a Web user interface and a fat client user interface. Then you would typically create a Web project for the Web user interface and a Java project for the standalone fat client. So you would have two project folders for one user interface layer. You could decide to put everything in one folder, but then you

would need a separate deployment mechanism that separates the classes to separate targets at deployment time (for example you could use Ant to do this). This would create additional effort and you would loose clarity in the developer and the deployment features of Rational Web Developer.

14.3.2 JAR file placement

When it comes to J2EE development, there are many ways you can decide to place your JAR files. Similarly, because J2EE applications run independently from each other using their own classloaders (depending on the visibility settings in WebSphere Application Server) there are many ways to control the scope of the classes that are loaded from JAR files.

You might want the classes to be visible over the whole WebSphere Application Server or in one enterprise application only. Or you might even want the classes to be visible from only one Web application.

The following sections provide some recommendations for JAR file placement.

Web application scope (WEB-INF/lib)

If your JAR file is only used in a single Web application, always put the JAR file in the Web project's WebContent/WEB-INF/lib folder. The JAR files in this folder are automatically added to the Java build path, and do not require any further setup when moved to a different server. So you can create portable WAR files with all libraries included by packing the Web projects folder.

Enterprise application scope (JAR dependencies)

If the JAR file is used by multiple modules within the same application, place the JAR file in the enterprise application (simply into the top folder of the project), then use the Java JAR Dependencies feature to set up the manifest files and the Java build class paths.

Figure 14-4 on page 543 shows our sample application structure where the JAR libraries that are placed in our enterprise application project. The JAR files are placed in the top level of the projects folder. Other projects can reference the libraries using JAR dependencies.
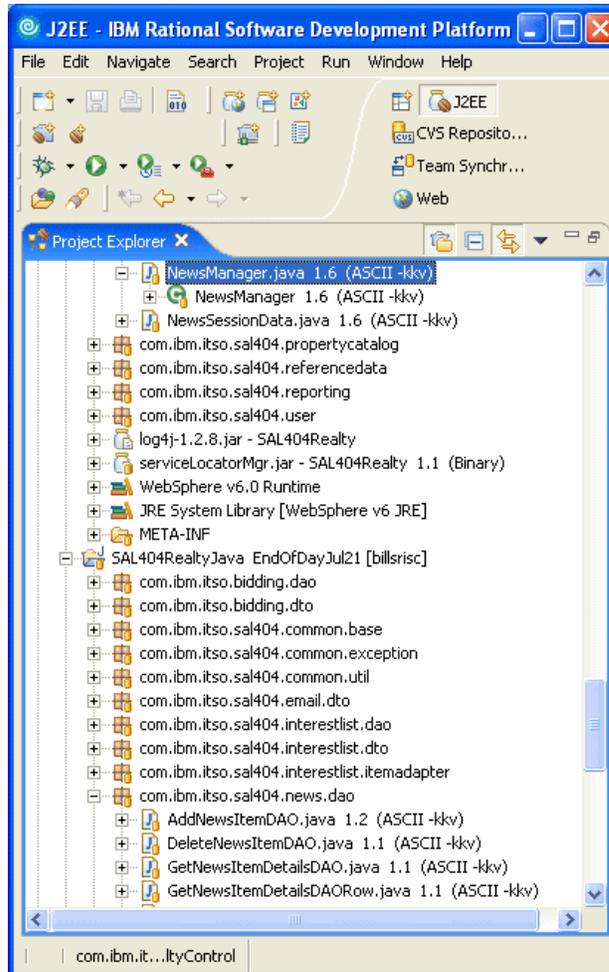
Figure 14-4   JAR libraries in the EAR project

The Java JAR Dependencies Properties page of an EJB or Web project is used to identify which JARs the EJB or Web project uses. You can access the Java JAR Dependencies by selecting **Properties** from the context menu of an EJB or Web project. Figure 14-5 on page 544 shows how to adjust the JAR dependency settings of your Web project.
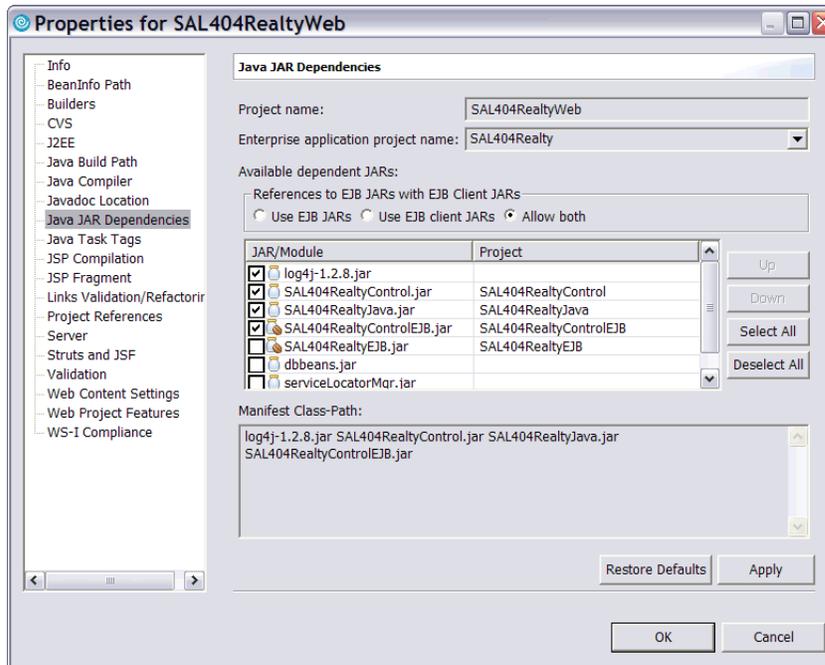
Figure 14-5   Set the JAR dependencies of a Web project

> **Note:** Remember that the JAR dependencies are for runtime while the build path is for build time when you use the Standard build within Rational Web Developer. When you add a dependency in the JAR dependencies, Rational Web Developer automatically adjusts the build path for you and adds the needed classes, library or project to the build path.
>
> If you need the classes in your deployed application at runtime on the server, then you should add them into the JAR dependencies. If you need them only to compile (for example for libraries that are already installed on WebSphere Application Server) it is sufficient to add them to the build path.

Server scope (ws.ext.dirs)

If the JAR file requires access to any J2EE or WebSphere Application Server classes, or to any other JAR files that have been added to ws.ext.dirs, it must also be placed on the ws.ext.dirs property. The ws.ext.dirs property is used for all JAR files that require access to the J2EE JAR files or WebSphere Application Server internal classes. If there are hard dependencies on the JAR file, you must update the Java build path of each project that uses the JAR file.

For WebSphere Application Server V5 servers, any changes you want to make to ws.ext.dirs can be made on the Environment page of the server editor.

For WebSphere Application Server V6 servers, this change is done using the WebSphere Administrative Console. See the *Managing shared libraries* topic in the WebSphere Application Server Version 6.0 Information Center for more details.

We do not recommend that you put JAR files on the global class path. Using one of the global class paths makes your application vulnerable to changes made to the classes on which you depend. If you have added the JAR file to the global class path or ws.ext.dirs properties you will have to publish the JAR file separately from your application, then you will have to set up the server class path again if you move to a different server.

Separate library project

Another possibility is to have a separate project for libraries that are referenced by other projects. In the case, you need to take care that these libraries are deployed to the target server when you deploy the application that needs the library. You can do a manual deploy of the library, but another way you can do this by writing scripts.

This has the advantage, that you can share the libraries on the CVS server with your team members. You do not have to care about local library paths on the development machines or about path variables in Rational Web Developer that might be different for different team members. You just add the libraries to the newly created project (simple or Java project) and let the developers synchronize with CVS.

When the version of the library changes, you just exchange the JAR file in the project. All the developers can synchronize with CVS to receive the changes. There is no need to have individual build paths.

14.3.3 Naming conventions

Naming conventions help to make it easier to read and understand code used in your application. This helps people in your project team to quickly find and understand code regardless of who originally wrote the code.

Table 14-1 on page 546 shows some of the naming conventions we used for Java code in our sample application. Note that naming standards might be very different in your project, and we do not suggest that you should use our standards, as long as you do agree to and follow standards that make sense for your project.

Table 14-1   Naming conventions for Java code

Coding level	Convention

package	► each package starts with `com.ibm.itso.sal404`
	
	► according to the component based design the following packagename is either `common` or `xyz` for the components, for example `news`
	
	► the following packages are used to group functionality:
	
	   – `form` - struts forms
	   – `action` - struts actions,
	   – `dao`- data access objects
	   – `dto` - datatransfer objects
	   – `manager` - managers
	   – other appropriate package names

class	► All classes have to use nouns that are self-explaining, for example `UserManager` ► An *Exception* to this rule are generated classes like DAO classes. All DAO classes in our application should follow the following scheme: verb + noun + DAO, for example `CreateUserDAO`
interfaces	► All interfaces end with `Interface`

Formatting

Its is also important to pick a common formatting style for your Java source code. Again this makes it easier to read and understand code. The Rational Software Development Platform can help you format code according to your preference, Choose Window → **Preferences** → **Java** → **Code Style** to access setting for various code formats and templates that you can then apply to your code.

We recommend that you also establish formatting and naming conventions for HTML and JSP files.

14.3.4 Using CVS

Working with CVS in a development team needs some common practices in order not to corrupt individual workspace or the branches in CVS. As always you need to agree to the rules in your team.

An ideal work flow is as follows:

1. Start fresh.

Before starting work, update the resources in the workspace with the current branch state. If you are sure that you have no local work that you care about, there are two ways of getting the latest code from CVS:

– If you do not have the code of the projects in your workspace, the fastest way to get caught up is to select the projects you are interested in from HEAD (or branch if your choose a branch) in the CVS perspective and select **Checkout**.

– If you already have the projects in your workspace right-click the project and select **Replace with** → **Latest from HEAD**. This overwrites your local resources with those from the HEAD (or branch if your choose **Replace with** → **Another branch or version...**).

2. Make your code changes.

   Work locally in your development environment, creating new resources, modifying existing ones, saving locally as you go.

3. Synchronize

   When you are ready to commit your work, synchronize with the repository. Right-click the project and choose **Team** → **Synchronize with Repository...**

   a. First do the updates.

      Examine incoming changes and add them to your local environment. This allows you to determine if there are changes which might affect the integrity of what you are about to commit. Resolve conflicts. Retest, run integrity checkers (for example, check for broken hypertext links, ensure your code compiles, and so on).

   b. Do your commits.

      Now that you are confident that your changes are well integrated with the latest branch contents, commit your changes to the branch. To be prudent, you can repeat the previous step if there are new incoming changes.

Important: It is always a good practice only to commit working code, at least it should be code that compiles without errors. Do not check in any code that does not compile. Compile errors and dependencies might block the whole team from testing their own code, because they cannot create any builds and deployments.

You could setup additional rules that say that every team member has to update and commit the code every day. That depends on your project setup and the project constraints.

15

Bidding component

In this chapter, we show you how to implement a bidding component for the SAL404Realty sample application that accompanies this book. We have divided this chapter into two different sections:

1. Bidding component specification (15.1, "Bidding component specification" on page 550)

   We explain what the bidding component does, how it fits in the overall design of the application, as well as the requirements it needs to implement.

2. Building the bidding component (15.2, "Building the bidding component" on page 550)

   We explain the steps necessary to build and test the add functionality of the component using Rational Web Developer.

15.1 Bidding component specification

The bidding component is used allow the buying and selling of properties. Its major functions are:

► Buyers can enter a bid against a property they want to buy.

► Sellers can accept or reject a bid.

► Administrators, the agent selling a property, and the seller can see a list of bidders for a specific property.

► The agent selling a property, and the seller cannot place a bid on that property.

The bidding component cannot be accessed directly from the menus of the application, but instead is called by other components. For example, when a buyer views the details of a property, he can make a bid for this particular property. The bid component, as any other component of the application, is developed using the layered architecture paradigm as described in Chapter 2, "Development process" on page 11.

> **Note:** We have not implemented all the requirements of the bidding component in our Sal404 sample application, but have implemented enough of the basic functionality so that you can get an idea of the key tasks needed to build such solutions.

15.2 Building the bidding component

In this part of the chapter, we demonstrate the steps to build the bidding component. In particular, we show you how to prepare your workspace, use the Struts builder to build the front-end of your component, use other Rational Web Developer wizards to build the back-end of your component, and finally we explain how to put everything together and test your component.

Our approach for building the bidding component will be a top-down one. This means that we will start with the front-end layers (presentation, controller) and move on to the back-end layers (business facade, domain, data access) of the component.

15.2.1 Preparing the workspace

Before we start writing the code for our component, we need to create the necessary Java packages and Web content folders in our workspace. We also

need to create some skeleton classes that will be used later in the building process.

Java source packages

The following Java packages need to be created:

► In the SAL404RealtyJava project, create these packages:

– com.ibm.itso.sal404.bidding.dto

All the data transfer objects classes used as place-holders of data will be created under this package.

– com.ibm.itso.sal404.bidding.dao

All the Database Access Objects classes used to perform database operations will be created under this package

► In the SAL404RealtyControl project, create these packages:

– com.ibm.itso.sal404.bidding

The Manager class which implements all the business methods of the bidding component will be created under this package. It will also contain a class to hold bid session data

► In the SAL404RealtyWeb project create packages:

– com.ibm.itso.sal404.bidding.action

All the Struts action classes will be created under this package.

– com.ibm.itso.sal404.bidding.form

All the Struts Form classes will be created under this package.

We demonstrate the steps necessary to create one of these packages, for example, `com.ibm.itso.sal404.bidding.action`. You can follow the same steps in order to create the rest of the packages:

1. Expand the **SAL404RealtyWeb** project, expand **Java Resources**, then right-click **JavaSource.**

2. When a pop-up menu appears, choose **New** → **Package**, as shown in Figure 15-1 on page 552**.**
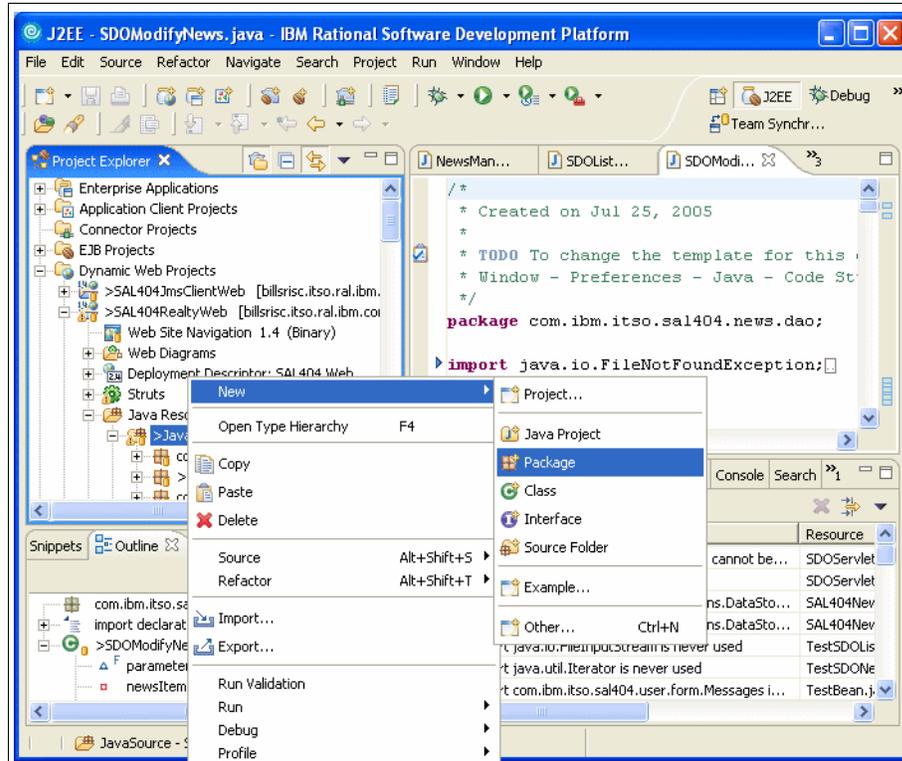
Figure 15-1   Add a Java Source package (Step 1)

3. A dialog box as the one shown in Figure 15-2 on page 553 appears. Type the name of the package in the Name text box. In this case it should be `com.ibm.itso.sal404.bidding.action`, and click **Finish**.

Follow the same procedure for the remaining packages of this component.
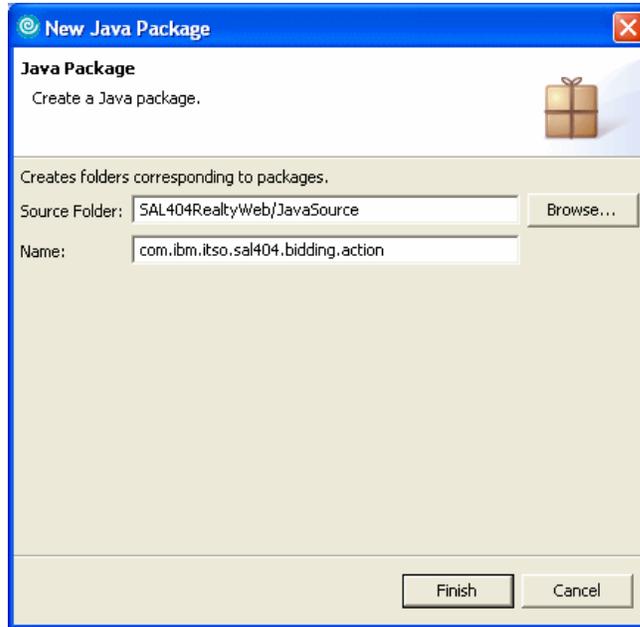
Figure 15-2   Add Java Source package (Step 2)

Web content folders

The following Web content folder needs to be created:

► bidding

  All the Web pages of the bidding component, as well as the Struts graph will be created under this folder.

The steps necessary to create this folder are:

1. Expand the **SAL404RealtyWeb** project, expand **Web Content**, then right-click **Web Content.**

2. When a pop-up menu appears, choose **New** → **Folder,** as shown in Figure 15-3 on page 554**.**
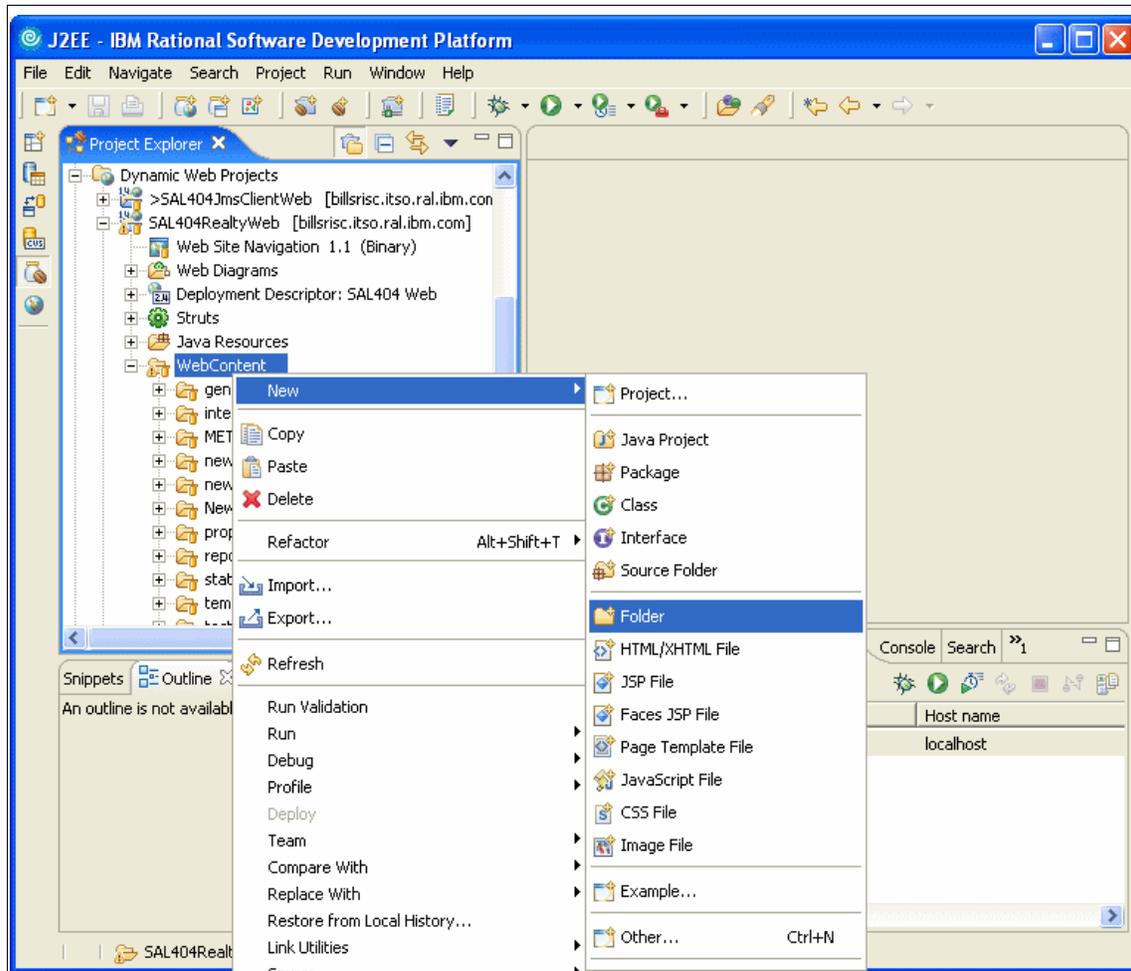
Figure 15-3   Add a Web Content folder (Step 1

3. A dialog box as shown in Figure 15-4 on page 555 appears. Type the name of the folder in the Folder name text box, `bidding`, and click **Finish**.
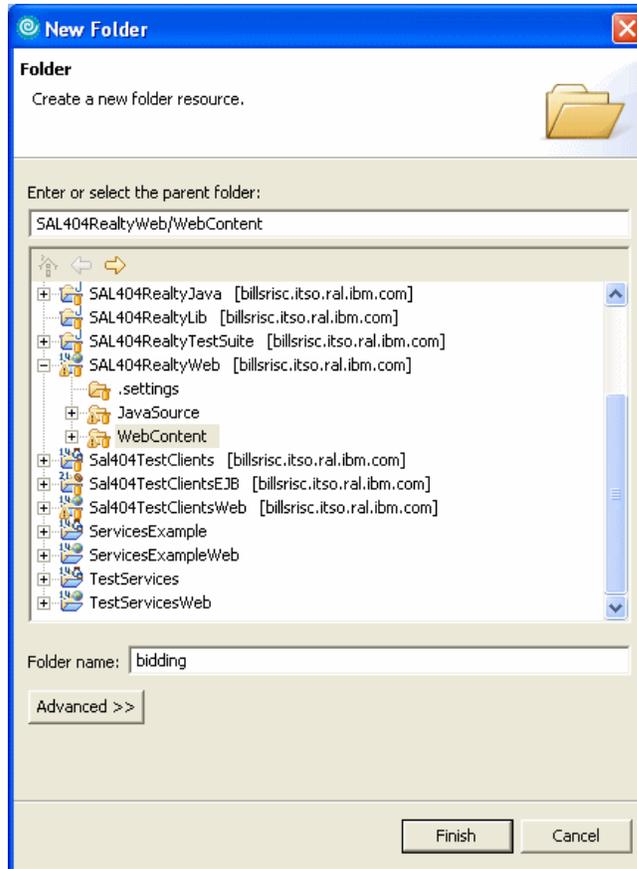
Figure 15-4   Add a Web Content folder (Step 2)

Now we are ready to start developing the code for the bidding component.

15.2.2 Changing the PropertyCatalog component

To prepare for the implementation of the bidding component, we need to make some changes to the property catalog component to allow for the fact that properties now have owners. Changes to make include:

▶ ResultPropertiesDTO

   – Add a seller attribute and generate getter and setter.

▶ PropertyDTO

   – Add a seller attribute and generate getter and setter.

▶ PropertyDetailsForm

– Add a seller attribute and generate getter and setter.

► addPropertyDetailsBody.jspf

– Add drop down to display and choose property seller.

► modifyPropertyDetailsBody.jspf

– Add drop down to display and choose property seller.

► PrpoertyCatalogManager

– Add code to addPropertyDetails method to ensure seller is saved.

15.2.3 Presentation layer

As mentioned earlier, we build the reporting component following a top-down approach. Therefore, we start by implementing the first layer of the component, the presentation layer.

The presentation layer is, essentially, the user interface of a component. This layer includes any Web pages that use forms and other UI elements to allow the user to enter data, as well as any Web pages that use tables and other UI elements to display information. The steps necessary to build the presentation layer are:

1. Create a Web diagram.
2. Add actions and Web pages to the diagram.
3. Create mappings between actions and Web pages.
4. Realize mappings, actions, and Web pages.
5. Implement the Web pages.

We describe the steps necessary to realize and implement the presentation layer for adding bids. Then, repeat the process to realize and implement the presentation layer for other functions in the bidding component.

Create a Web diagram

We start working on the presentation layer of the reporting component by creating a Web diagram. This will help us visualize the layer, its elements, and the interactions between them. It will also help us realize and fully implement these elements later on in our building process.

To add a Web diagram to your Rational Web Developer workspace:

1. Expand the **SAL404RealtyWeb** project, expand the **Web Content** folder, locate the bidding folder, and right-click it.

2. When a pop-up menu appears, choose **New** → **Other** as shown in Figure 15-5 on page 557.
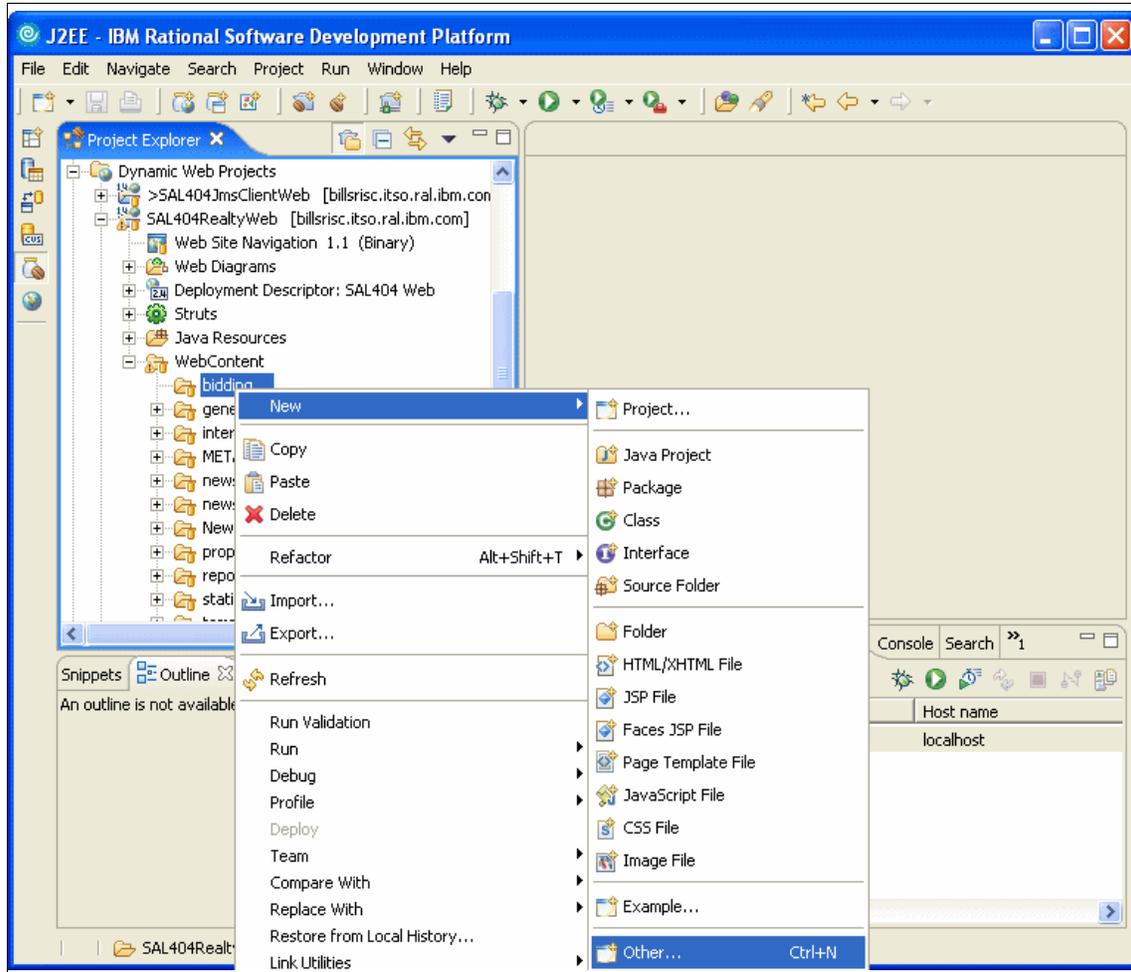
Figure 15-5   Add a Web diagram (Step 1)

3. When you are presented with a dialog box, select **Web Diagram** as shown in Figure 15-6 on page 558, and click **Next**.
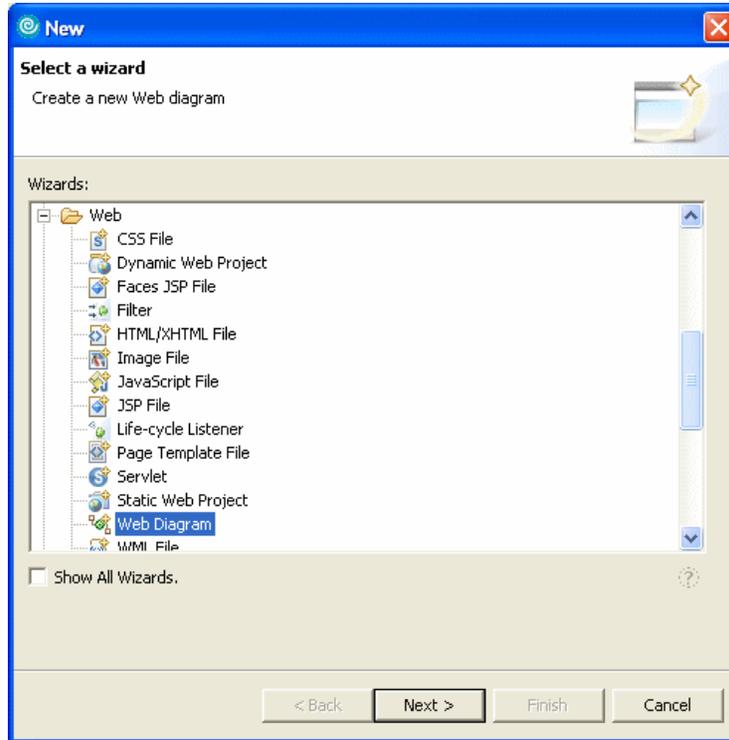
Figure 15-6   Add a Web diagram (Step 2)

4. A dialog box as the one shown in Figure 15-7 on page 559 opens. Enter `biddingGraph` in the File name textbox and click **Finish**.
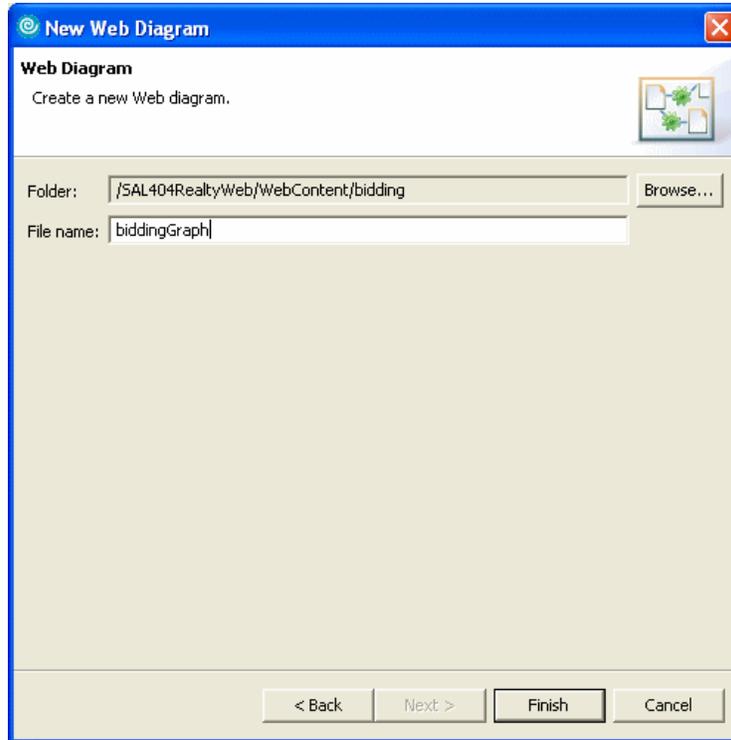
Figure 15-7   Add a Web diagram (Step 3)

5. An empty Web diagram named biddingGraph.gph is created for you, and
    opened in your workspace as shown in Figure 15-8 on page 560.
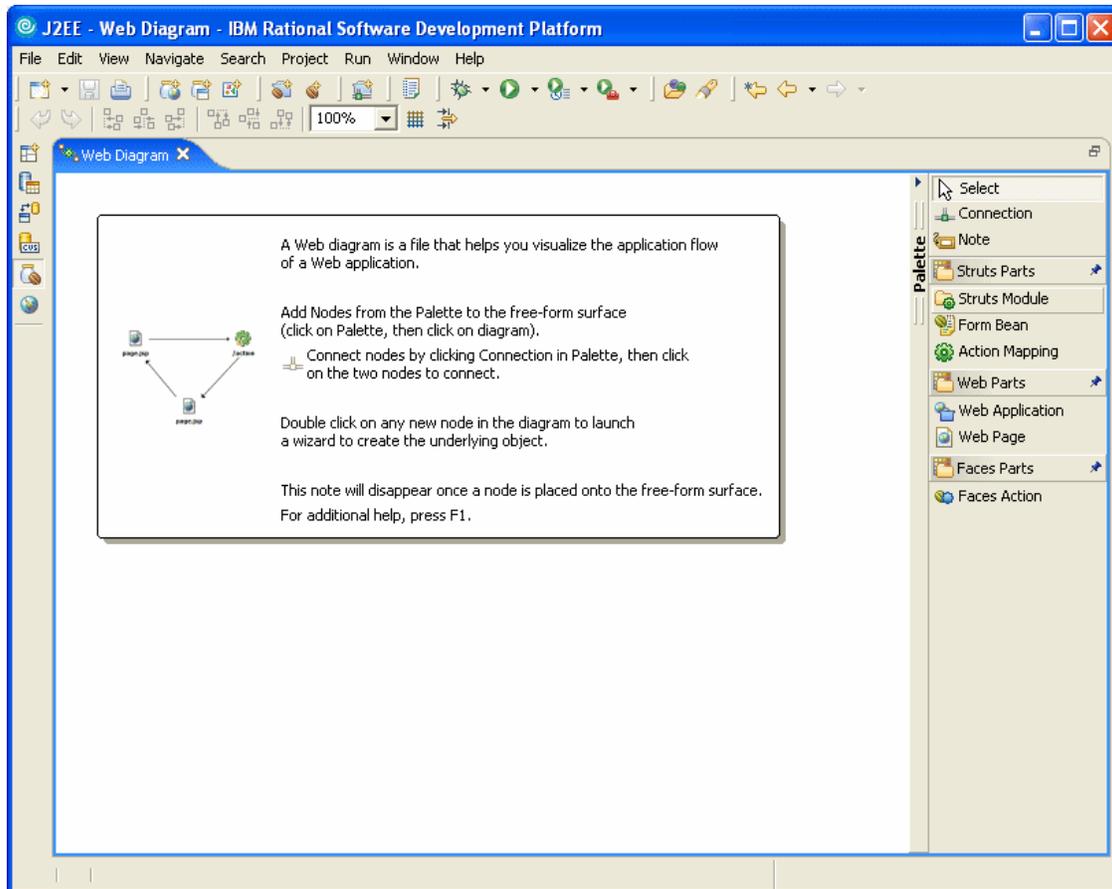
Figure 15-8   Add a Web diagram (Step 4)

6. We want to add all the Struts components for bidding into a new Struts module, so now we right-click the free form surface of the bidding graph in the Web diagram editor and choose **Change the Struts Module association** as shown in Figure 15-9 on page 561.
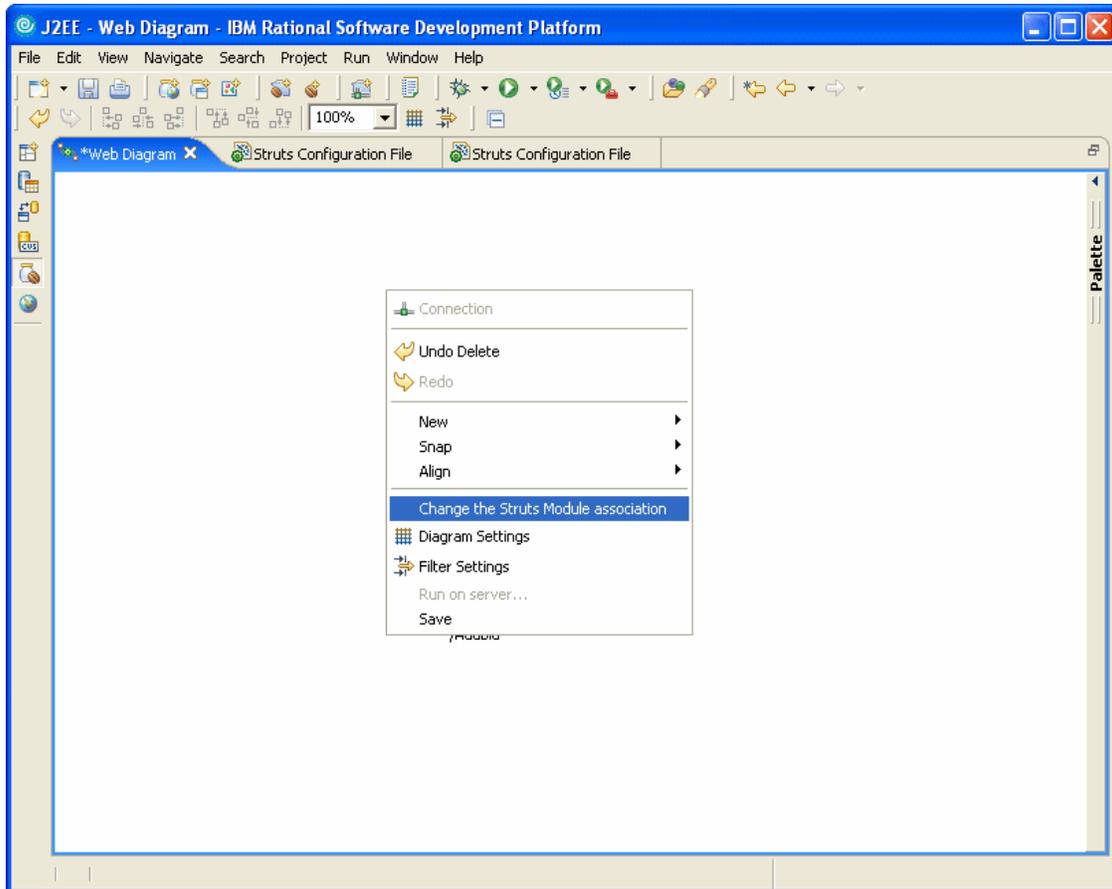
Figure 15-9   Change the Struts Module association

7. Enter `bidding` as the module name and click **OK** as shown in Figure 15-10.
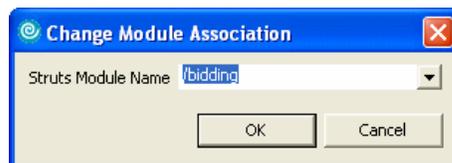
Figure 15-10   Create Struts bidding module

Add actions and Web pages to the diagram

Once the Web diagram is created, it is time to start adding nodes to the graph.

1. With the bidding.gph open in your workspace, right-click and select **New** →
   Struts Parts → **Action Mapping** as shown in Figure 15-11.
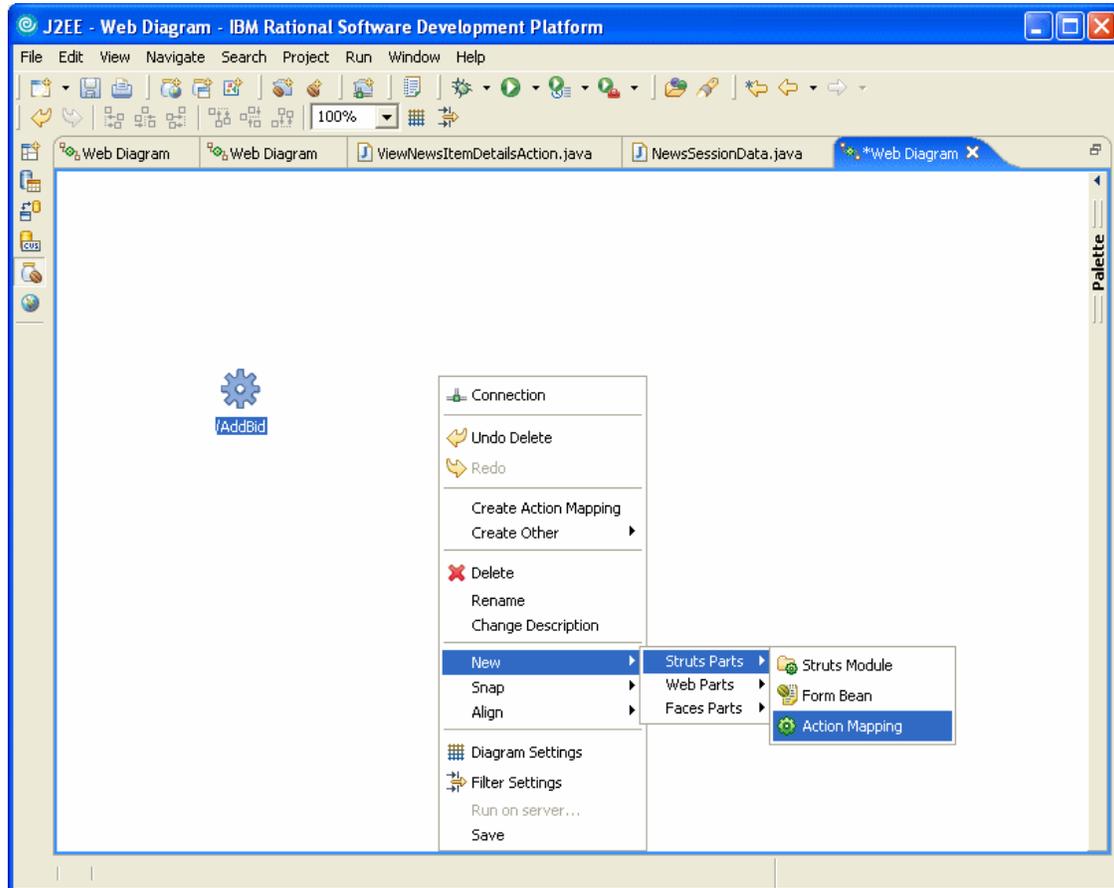
Figure 15-11   Add Struts action mapping

2. Drag the newly created icon to a place within your graph and enter the name
   you want to use. Figure 15-12 on page 563 shows an example of creating the
   ModifyBid action. Notice that the action icons are greyed-out. This means that
   the actions are not yet realized.
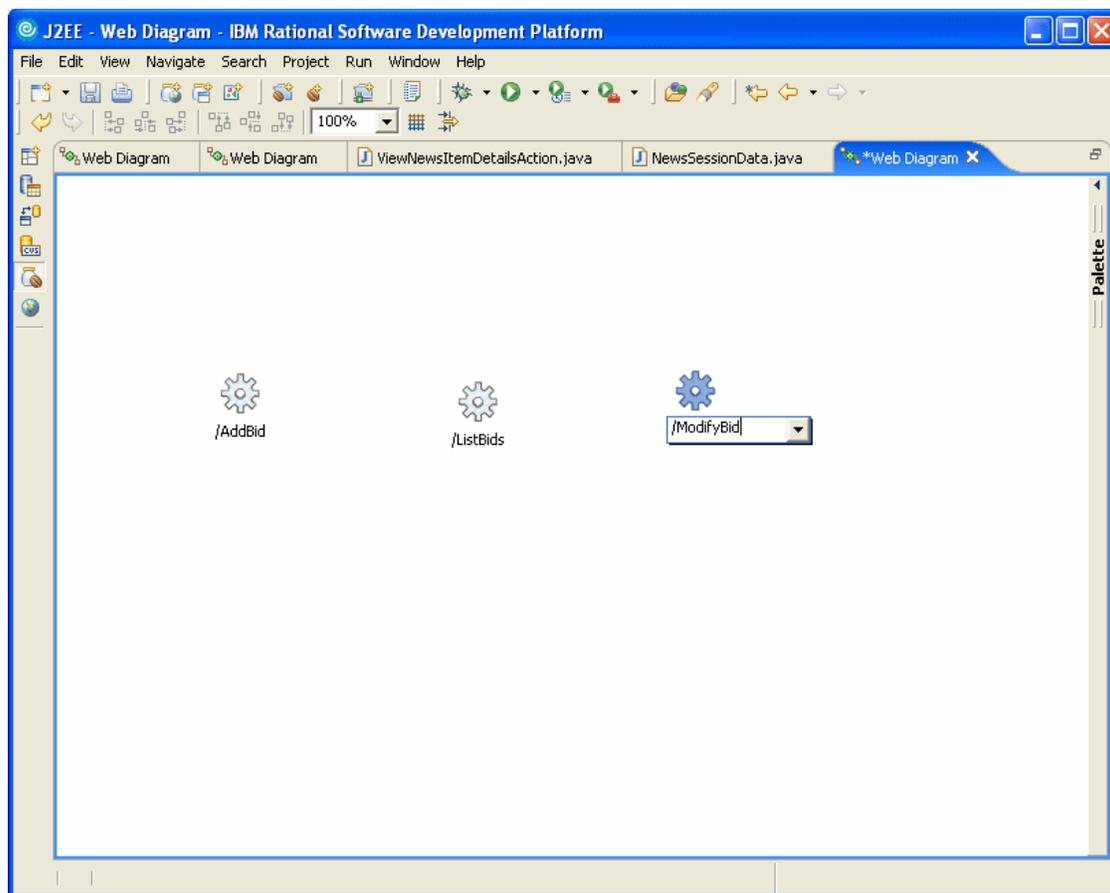
Figure 15-12   Add an action to the Web diagram

When completed, our Web diagram will have the following actions:

– AddBid
– ListBids
– ModifyBid
– SetupBid

> **Tip:** You can also add nodes to the Web diagram by choosing them from the palette at the right side of the editor.

3. We now add a Web page node to our diagram. While having the bidding.gph open in your workspace, right-click and select **New** → **Web Parts** → **Web Page** from the menu that appears, as shown in Figure 15-13 on page 564.
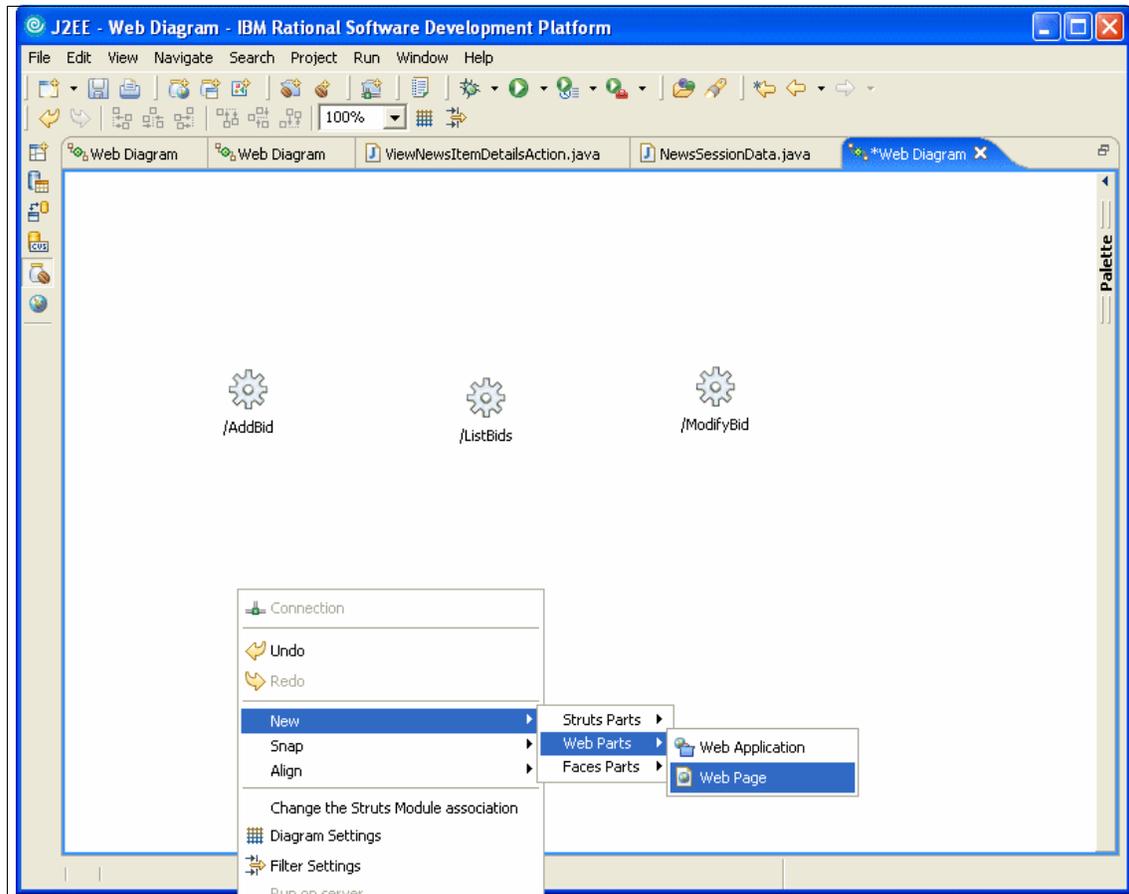
Figure 15-13   Add a Web page to the diagram

4. Drag the newly created icon to a place within your graph and enter a name to use. Figure 15-14 on page 565 shows and example of adding a Web page called /bidding/AddBid.jsp. Notice that the icon is greyed-out. This means that the Web page is not realized.
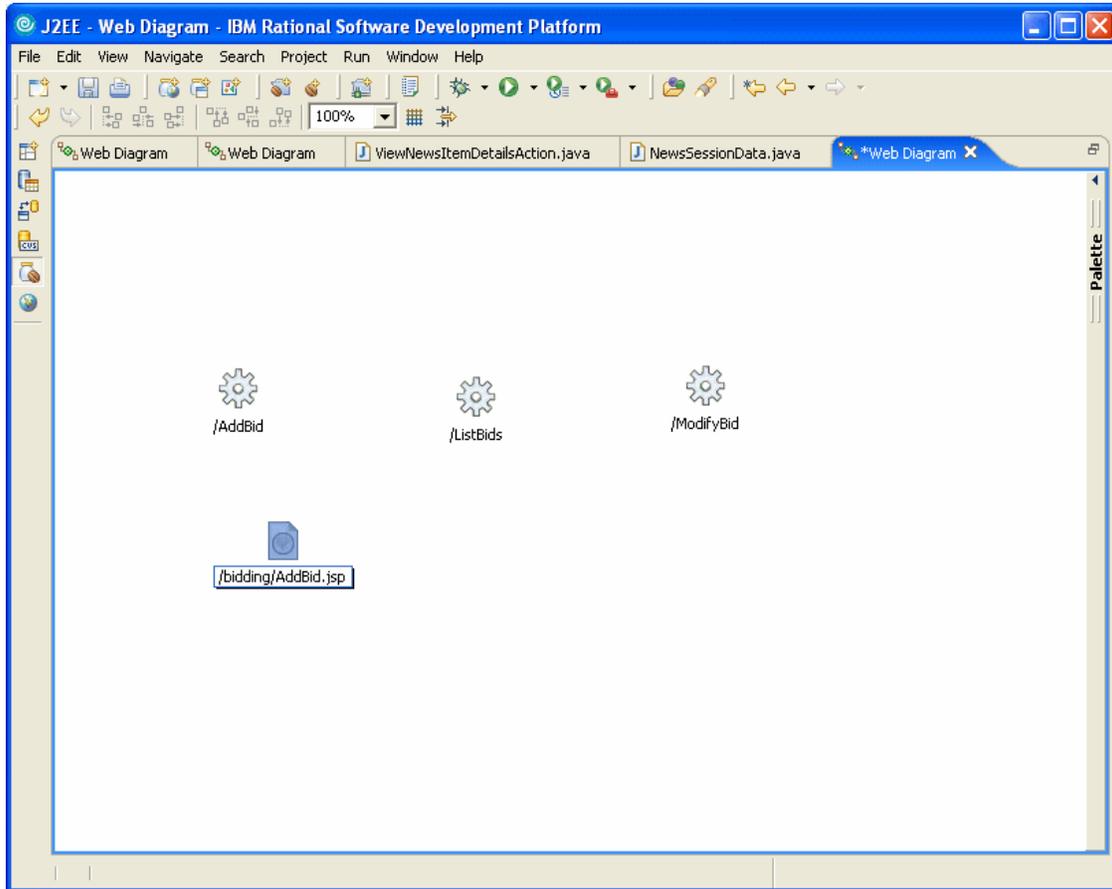
Figure 15-14   Enter a name for the Web page

When our bidding diagram is completed, we have Web pages called:

– /bidding/AddBid.jsp
– /bidding/ListBids.jsp
– /bidding/ModifyBid.jsp

Create mappings between actions and Web pages

Once we have created an action node and a node for the Web page, we need to connect them:

1. Right-click the **/AddBid** action node and select **Connection** as shown in Figure 15-15 on page 566.
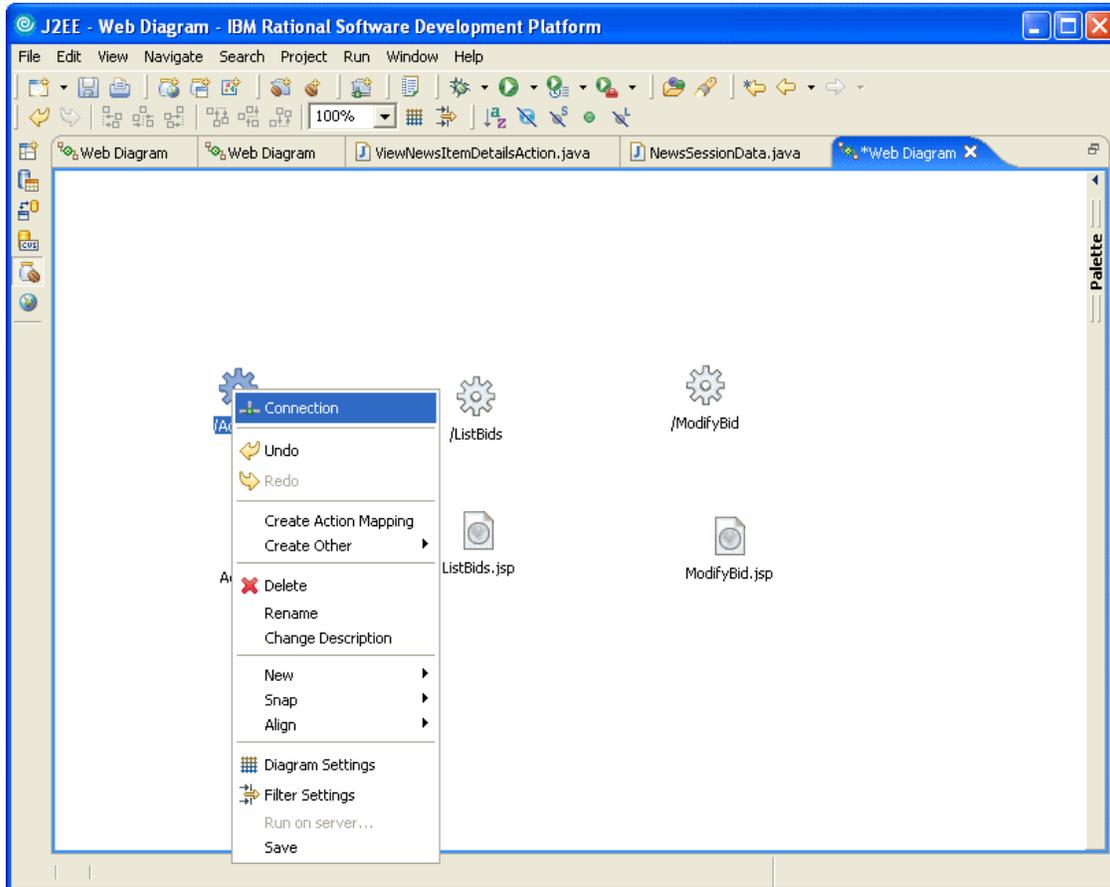
Figure 15-15   Create connections between actions and pages

2. Choose a connection type of Local Forward and click **OK,** as shown in Figure 15-16.
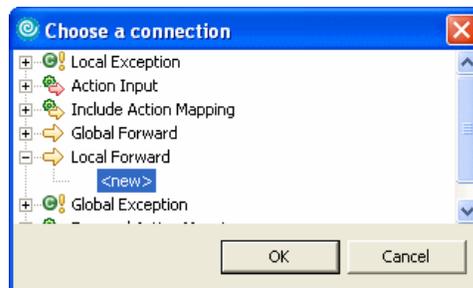
Figure 15-16   Choose connection

3. Drag the dotted line to the /bidding/AddBid.jsp Web page node and release it. Finally, type `/success` as the name of your connection, as shown in Figure 15-17 on page 568.

Notice that the two nodes are now connected with a dotted line named `/success` with an arrow pointing to the Web page node.

– The line is dotted because the connection is not yet realized.

– The arrow points to the Web page because data will move outwards from the action and inwards to the Web page.

– The connection is named /success because this is the path that is going to be followed if the execution of our AddBid action is successful.

> **Note:** In fact we are later going to modify this action forward using the Struts configuration editor as we want a successful add of a bid to return us to the view properties page we came from and this node is defined in a different Struts module. It is still useful to create the connection using the diagram editor.
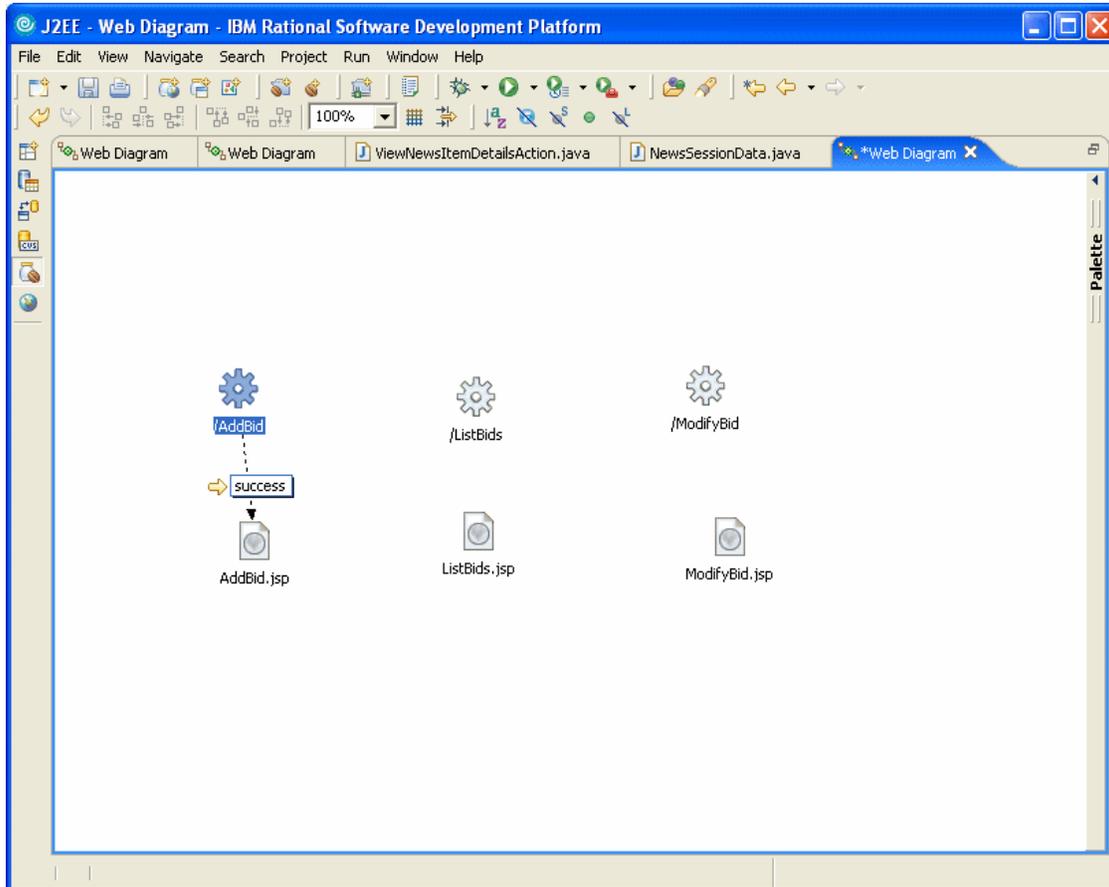
Figure 15-17 Create a connection between and action and a page

4. We also specify a /failure connection to be followed when, during the execution of our AddBid action, an error occurs. Follow the steps outlined before but this time type `/failure` as the name of the connection.

Add form beans

Next we add Struts form beans to our Web diagram.

1. While having the bidding.gph open in your workspace, right-click and select **New** → **Struts Parts** → **Form Bean** as shown in Figure 15-18 on page 569.
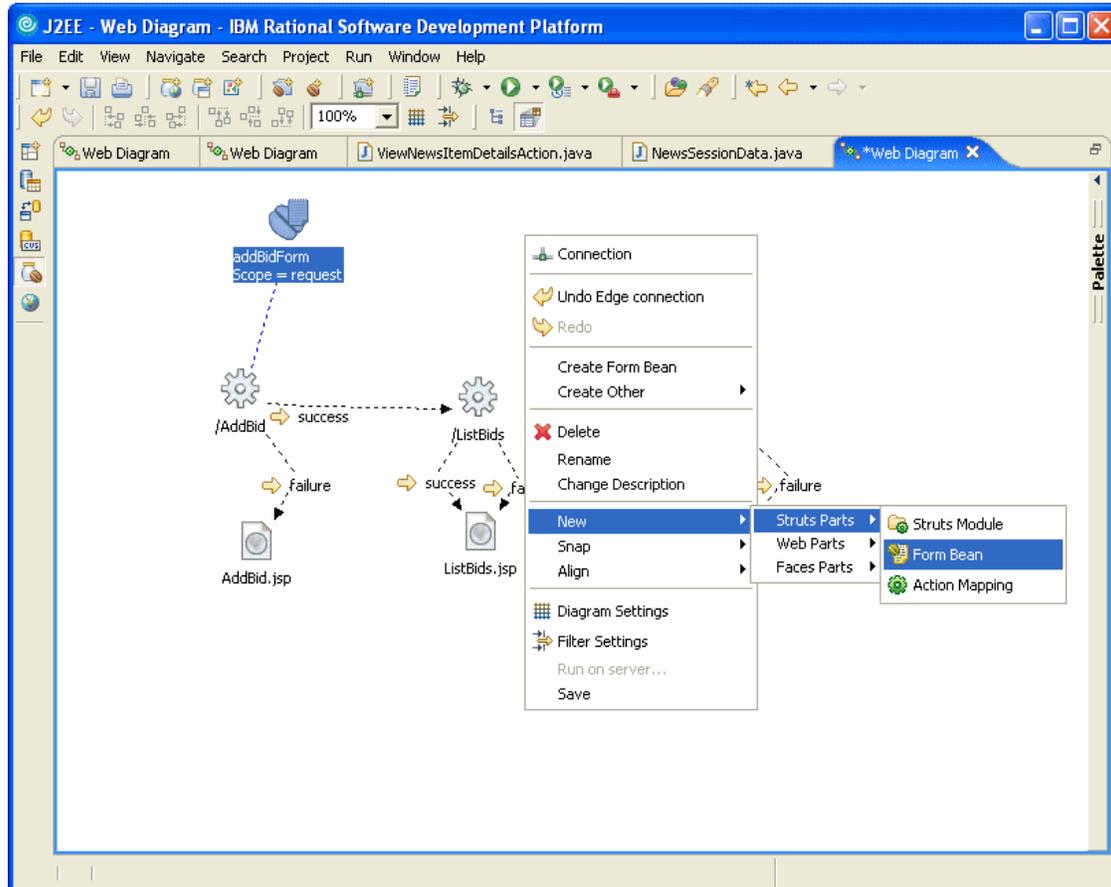
Figure 15-18   Add form bean

2. Drag the newly created icon to a place within your graph and enter the name you want to use. Figure 15-19 shows an example of creating the listBidsForm.

Figure 15-19   Add listBidsForm

3. Right-click the new form bean, choose **Connection** and drag the dotted line to the target action. For example, connect the listBidsForm to the ListBids action.

When our bidding diagram is completed we will have the following form beans:

– bidForm connected to the AddBid action
– listBidsForm connected to the ListBids action
– manageBidsForm connected to the ManageBids action
– modifyBidForm connected to the ModifyBid action

Realize forms, actions and Web pages

After creating and connecting the nodes for the forms, actions and the Web pages, we need to realize them in order for the relevant files to be created:

1. Double-click the **/AddBid** node and a dialog box similar to the one shown in Figure 15-20 on page 571 opens, click **Next**.
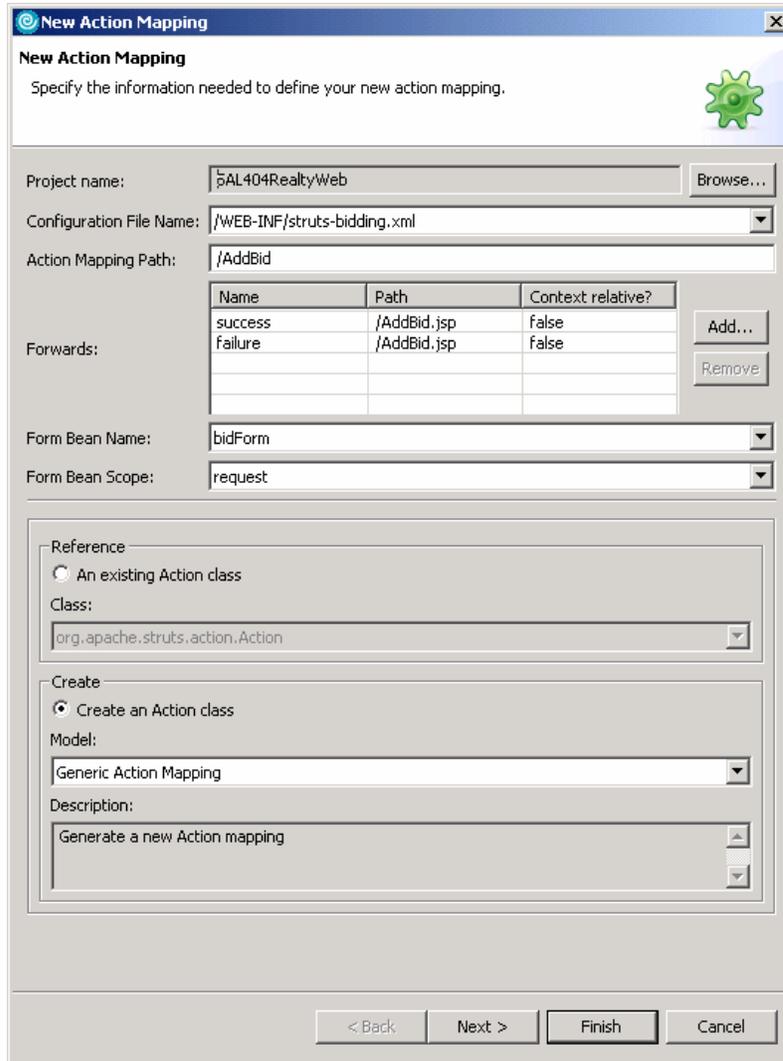
Figure 15-20   Realize AddBid action

2. Click **Browse** next to the Java package text box to select the package where your AddBidAction class will be created.

3. In the Package Selection dialog box, select the **com.ibm.itso.sal404.bidding.action** package and click **OK**.

4. You will be returned to the previous dialog box. Confirm that all the details are as shown in Figure 15-21 on page 572 and click **Finish**.
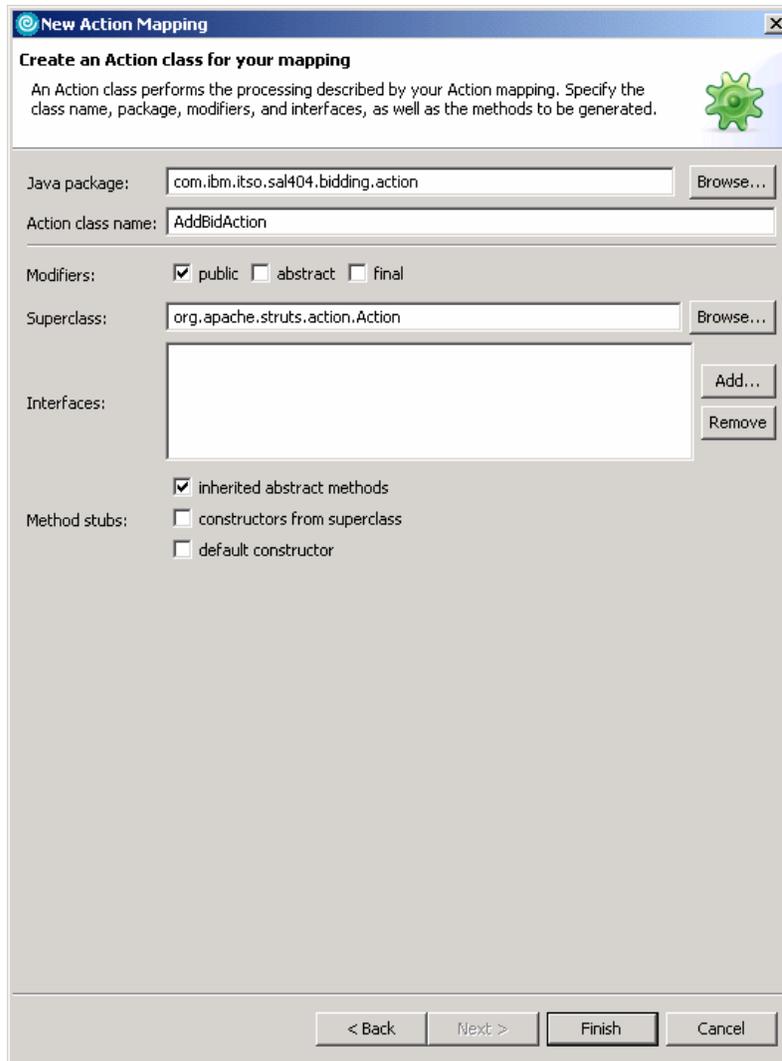
Figure 15-21   Realize AddBid action (Step 2)

5. A class for the AddBidAction action will be created on your file system under the **com.ibm.itso.sal404.bidding.action** package. Save the file and return to the Web diagram.

6. Double-click the **/AddBid.jsp** Web page node, and a New JSP File dialog box as shown in Figure 15-22 on page 573 opens.
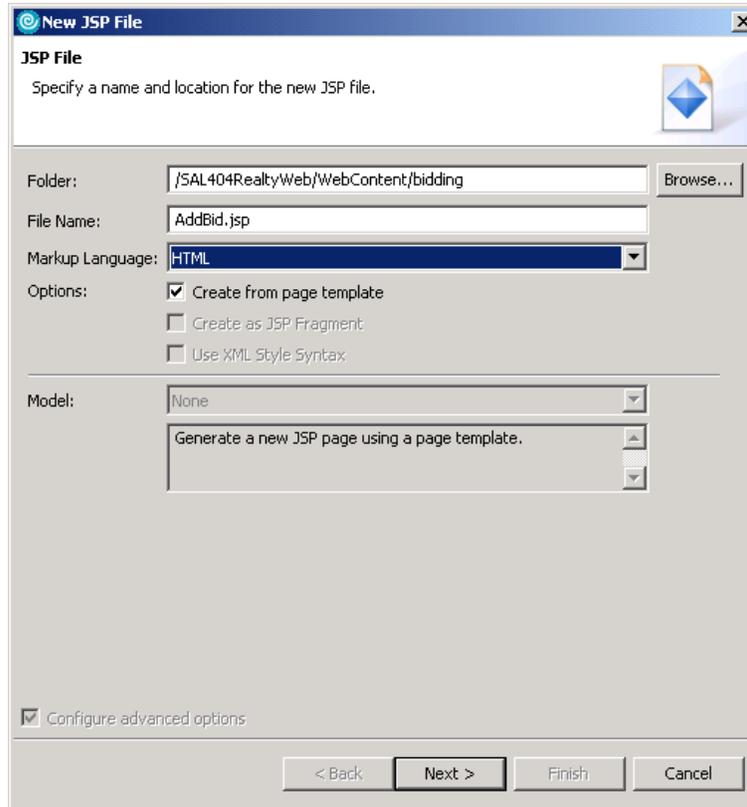
Figure 15-22   Realize AddBid.jsp

7. Select **Create from page template** and click **Next**.

8. Choose **User-defined page template**, select the **sal404.jtpl** template and click **Finish**. See Figure 15-23 on page 574 for an example.
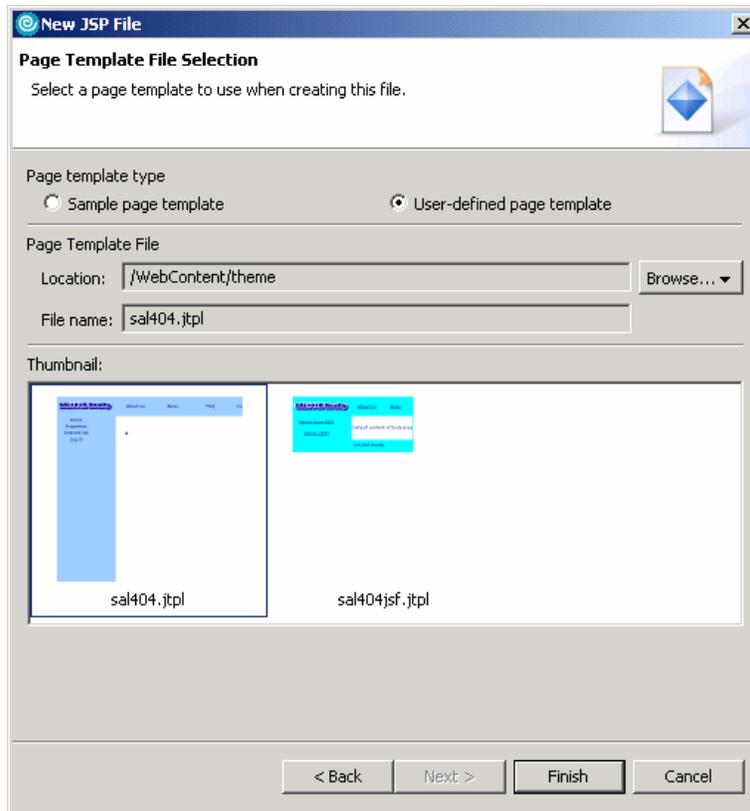
Figure 15-23   Choose user defined template

9. A JSP file is created for the /AddBid.jsp Web page under the bidding folder. Save the file.

10. Double-click the **/bidForm** form bean node, and a New Form-Bean dialog box as shown in Figure 15-24 on page 575 opens.
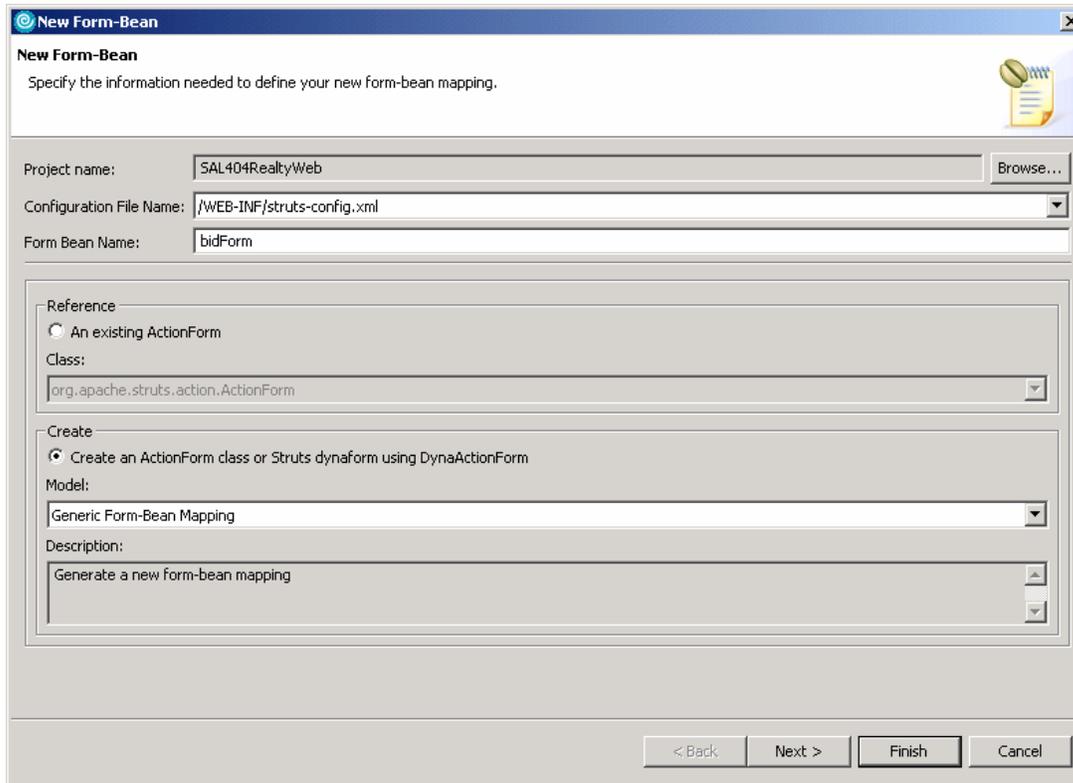
Figure 15-24   New form bean

11. Click **Next.** We do not have any existing fields to choose for the action form class, so click **Next** again.

12. We will create new fields for our form. Click **Add** and enter a new field with a name of `bidPrice` and a type of `java.math.BigDecimal`. Repeat this process to add the following fields and types:

   – `sbidPrice - String`
   – `propertyUserId - String`
   – `sbidDate - String`
   – `bidDate - java.util.Date`
   – `bidderId - String`
   – `status - String`
   – `id - String`
   – `propertyId - String`

   See Figure 15-25 on page 576 for the completed list of form fields.
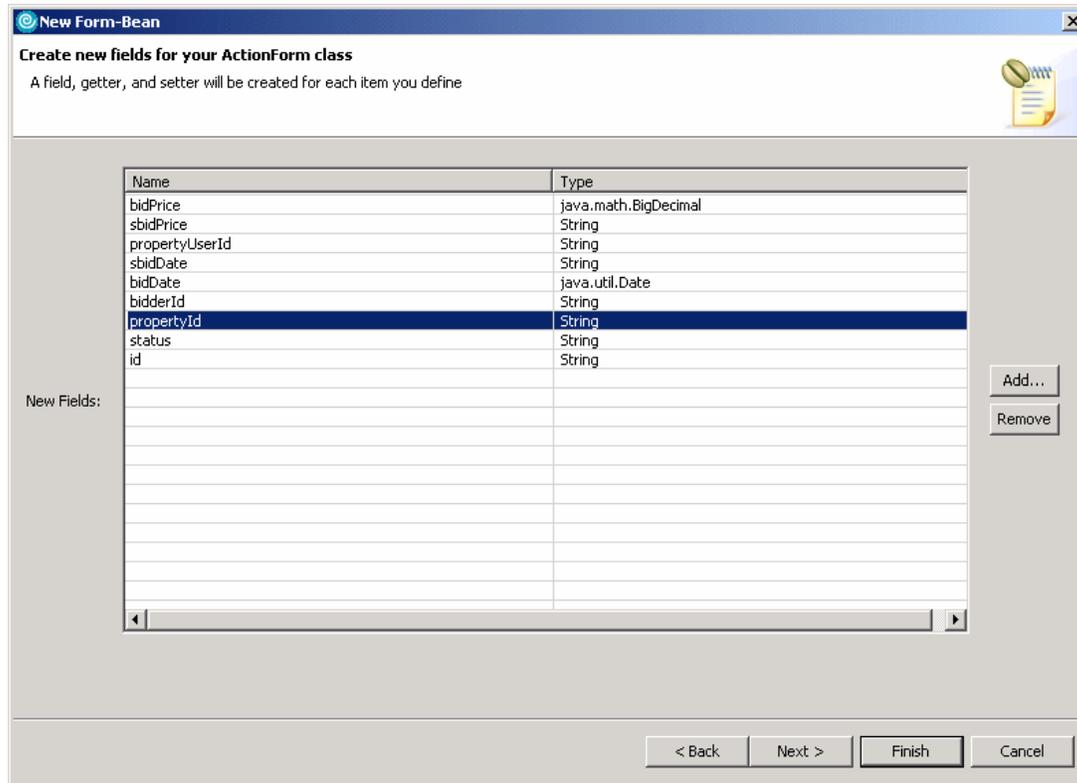
Figure 15-25   New form fields for bidForm

13. Click **Browse** to choose the correct package for the new BidForm class. The package name should be **com.ibm.itso.sal404.bidding.form**. Figure 15-26 on page 577 shows the settings for the BidForm class. Click **Finish**.
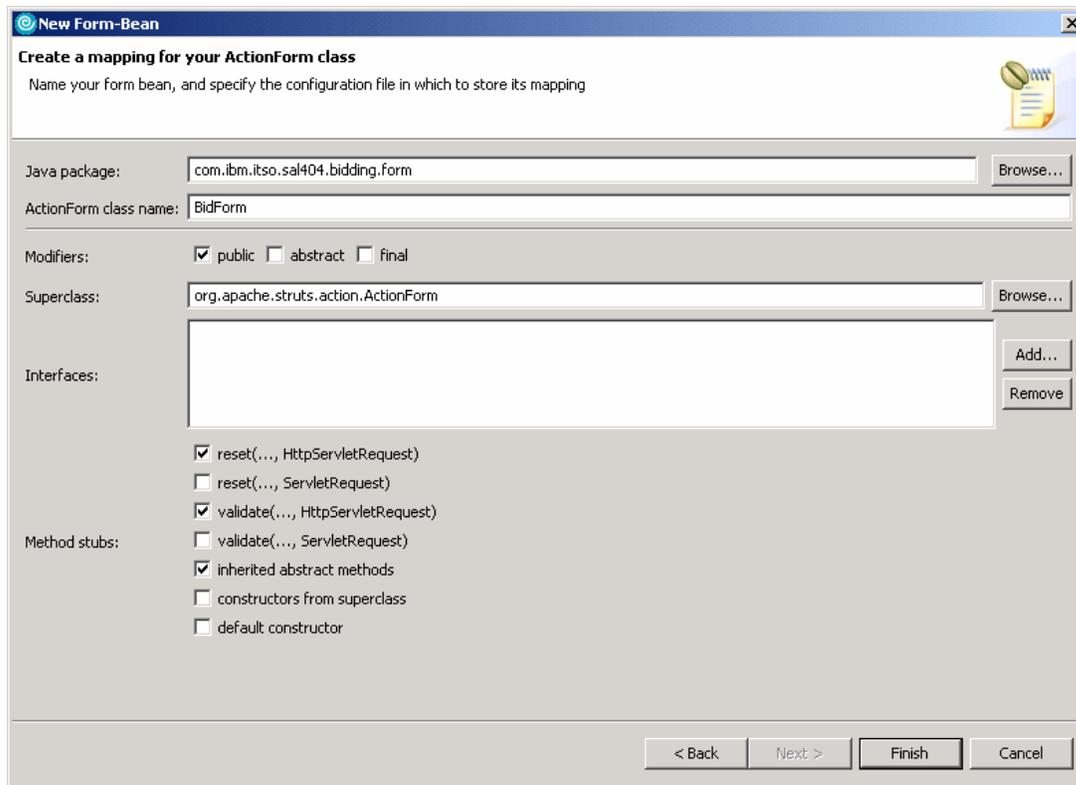
Figure 15-26 Settings for BidForm class

14. Save and close the newly created BidForm class.

15. Return to the Web page diagram and save it.

Using the Struts module switching action

As mentioned in "Create mappings between actions and Web pages" on page 565 the success action forward we created between the AddBid action and the AddBid.jsp Web page needs to be modified. The reason is our bidding application logic requires us to return to the ViewPropertyDetails page when a bid is added instead of remaining on the AddBid page. The ViewPropertyDetails page is defined in our default Struts module and not in the bidding module so to create a successful forward to a page in a different module we need to use the Struts switch action. To do this, we first have to add an action mapping to our Struts configuration files:

1. We can open the Struts configuration file for our bidding module directly from the biddingGraph Web diagram or it can be opened by using the Project Explorer.

- – To open the Struts configuration file from the Web diagram, you can double-click the connection between AddBid action and the AddBid.jsp.

- – To open the Struts configuration file from the Project Explorer navigate to the SAL404RealtyWeb project, choose **Struts** → **/bidding**, and then double click the **struts-bidding.xml** file.

2. Once the Struts configuration file editor has opened, choose the **Action Mappings** tab, click **Add** to create a new action mapping and enter /toModule as the action mapping name. See Figure 15-27 for an example.
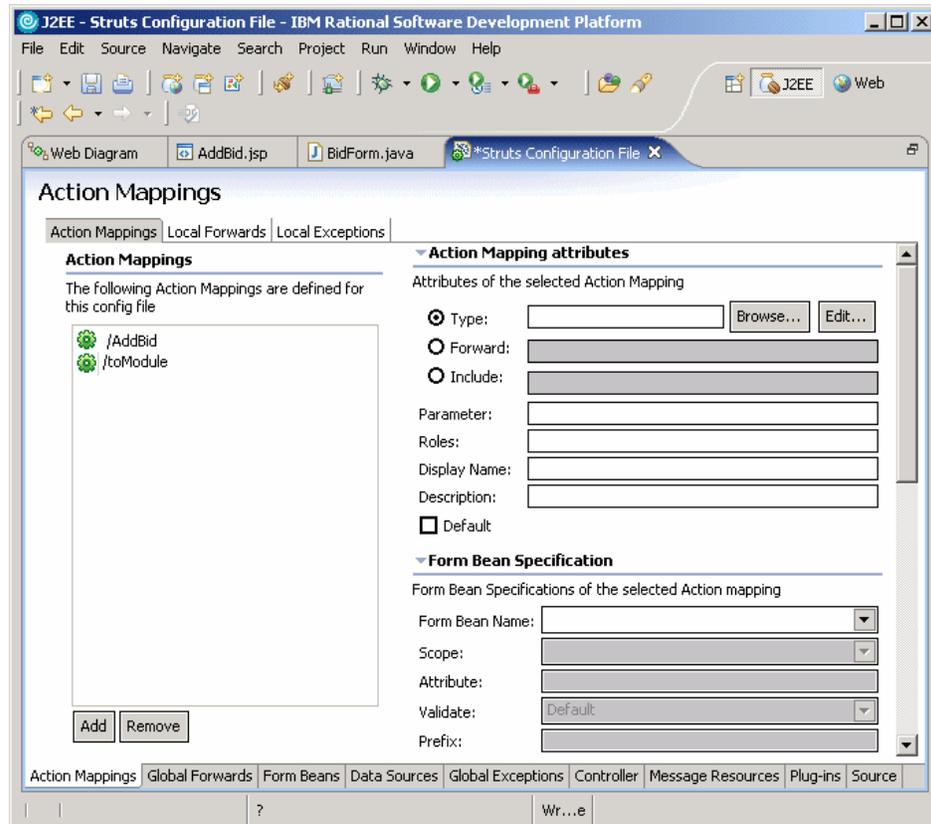
Figure 15-27   Create toModule action mapping

3. The new mapping we have created will use the Struts SwitchAction class as its implementation. To specify this, we select the **/toModule** action mapping, choose **Type** under the Attributes for the selected mapping, and then click **Browse**. From the list of available Struts action classes as shown in Figure 15-28 on page 579 select **SwitchAction** and click **OK**.
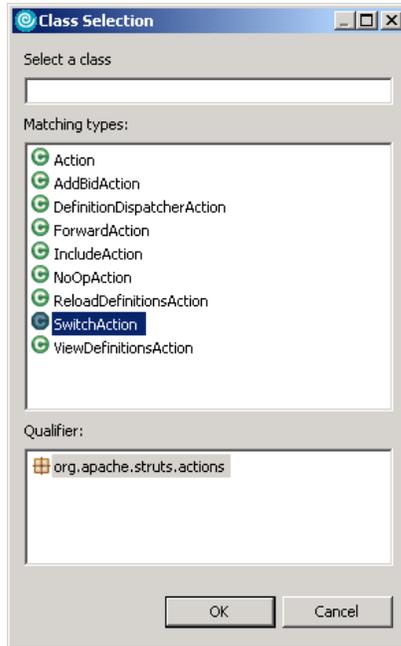
Figure 15-28   Available Struts action classes

4. Figure 15-29 on page 580 shows the completed toModule action mapping.
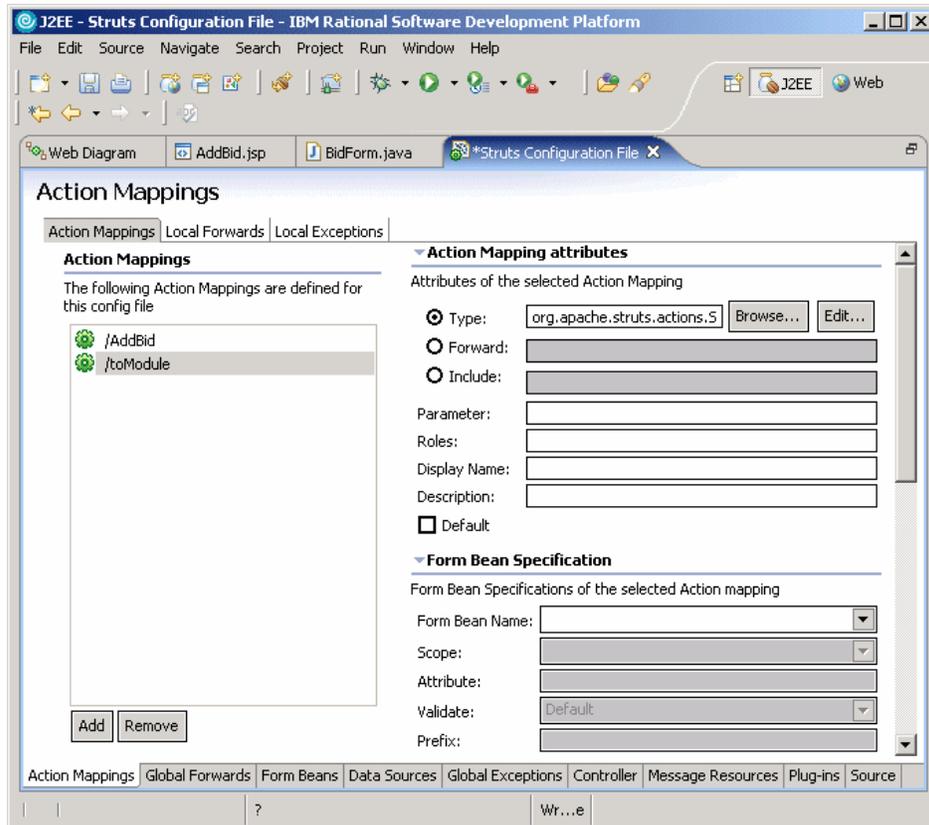
Figure 15-29   Completed toModule action mapping

   c. Now that the toModule action mapping is created we can replace the success forward for the AddBid. Select the **AddBid** action mapping, switch to the **Local Forwards** tab, and select **success**. Enter `/toModule.do?prefix=&page=/ViewPropertyDetails.do` as the path attribute for the selected forward. See Figure 15-30 on page 581 for an example.

For an explanation of the use of module switching in Struts see:

`http://struts.apache.org/userGuide/configuration.html#module_config-switching`

> **Note:** The default version of Struts used with Web projects created in Rational Web Developer is 1.1. The Struts documentation referred to in the URL above also describes using the module parameter as part of any of the Struts hyperlink tags (Include, Img, Link, Rewrite, or Forward), but this method of module switching was not introduced until Struts version 1.2.
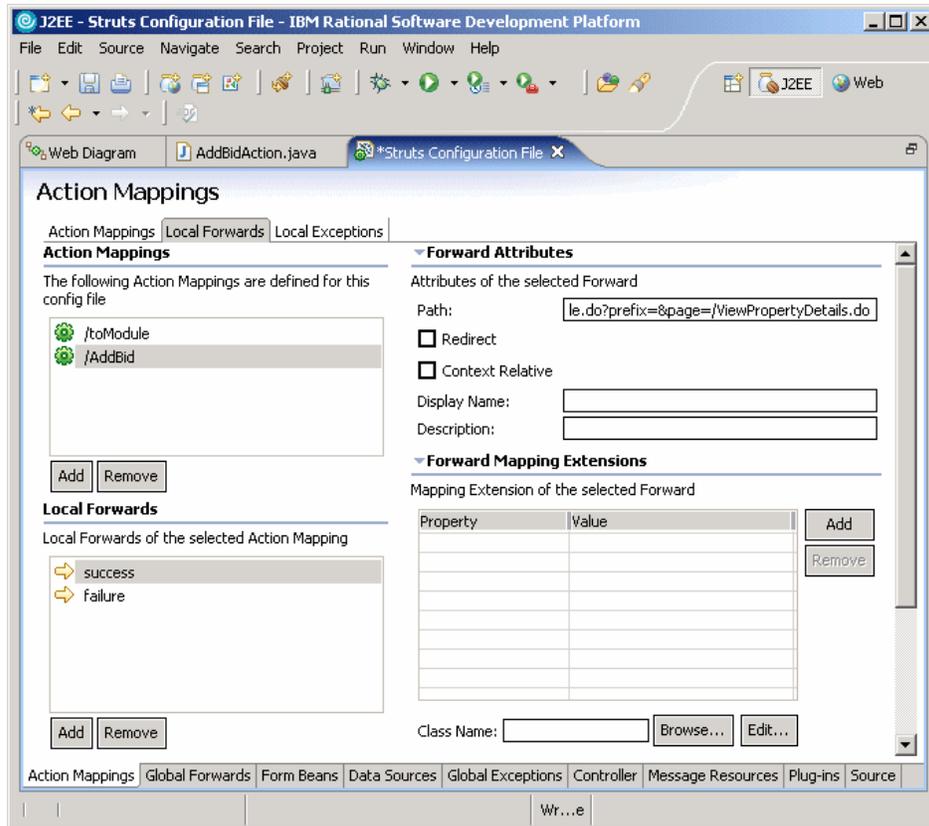
Figure 15-30   Local forward for AddBid success

5. Save the changed Struts configuration file.

> **Tip:** We also need to add the same action mapping to our default Struts
> configuration file and change the URL links in our standard menus to be
> module aware. For example, in our nav_head.jspf and nav_side.jspf menu
> include files we alter the URL links. Example 15-1 shows the original
> nav.head.jspf while Example 15-2 on page 582 shows the module-aware
> version.

Example 15-1   Original nav_head.jspf

```
<TABLE border="0" cellpadding="2" cellspacing="2" width="400">
    <TBODY>
        <TR>
            <td align="center" width="75"><html:link styleClass="menulink"
```

```
                page="/general/aboutus.jsp">About us</html:link></td>
            <td align="center" width="75"><html:link styleClass="menulink"
                page="/ListNews.do">News</html:link></td>
            <td align="center" width="75"><html:link styleClass="menulink"
                page="/general/faq.jsp">FAQ</html:link></td>
            <td align="center" width="75"><html:link styleClass="menulink"
                page="/general/contact.jsp">Contact us</html:link></td>
        </TR>
    </TBODY>
</TABLE>
```

Example 15-2   Module-aware nav_head.jspf

```
<TABLE border="0" cellpadding="2" cellspacing="2" width="400">
    <TBODY>
        <TR>
            <td align="center" width="75"><html:link styleClass="menulink"
                page="/toModule.do?prefix=&page=general/aboutus.jsp">About
us</html:link></td>
            <td align="center" width="75"><html:link styleClass="menulink"
                page="/toModule.do?prefix=&page=ListNews.do">News</html:link></td>
            <td align="center" width="75"><html:link styleClass="menulink"

page="/toModule.do?prefix=&page=general/faq.jsp">FAQ</html:link></td>
            <td align="center" width="75"><html:link styleClass="menulink"
                page="/toModule.do?prefix=&page=general/contact.jsp">Contact
us</html:link></td>
        </TR>
    </TBODY>
</TABLE>
```

Implement the Web pages

So far we have created skeleton Web pages for the presentation layer of the
bidding component. Now, we need to implement the Web pages in order to
complete the development of this layer.

We show how to build the required Web pages for the add bid functions to work.
Our Sal404 sample application also includes Web pages for listing bids and
modifying bids which were developed the same way

1. In step 6 on page 572 of the "Realize forms, actions and Web pages" on
   page 570 section we created a skeleton AddBid.jsp file, but because we now
   need to create a body JSP fragment called addBidBody.jspf for use with the
   AddBid.jsp.

2. Using the Project Explorer view, navigate to the SAL404RealtyWeb project, select **WebContent** → **bidding**, right-click and choose **New** → **JSP File** as shown in Figure 15-31.
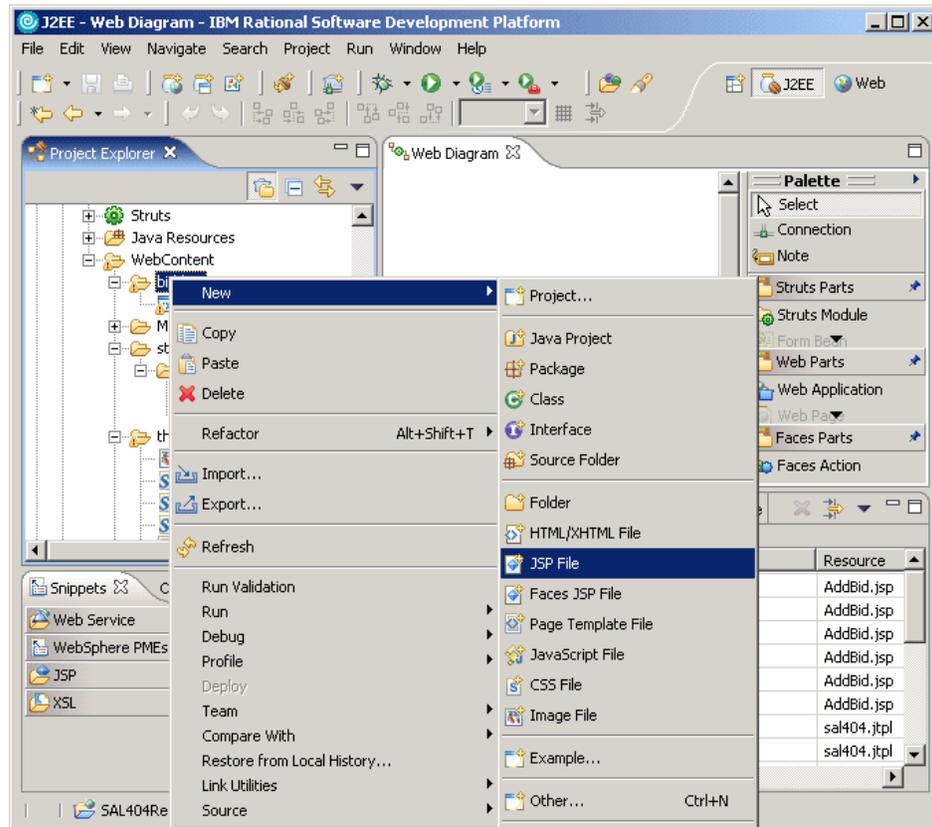
Figure 15-31   Create new JSP file

3. Select **Create as JSP Fragment** and enter a file name of `addBidBody` as shown in Figure 15-32 on page 584. Click **Next**.
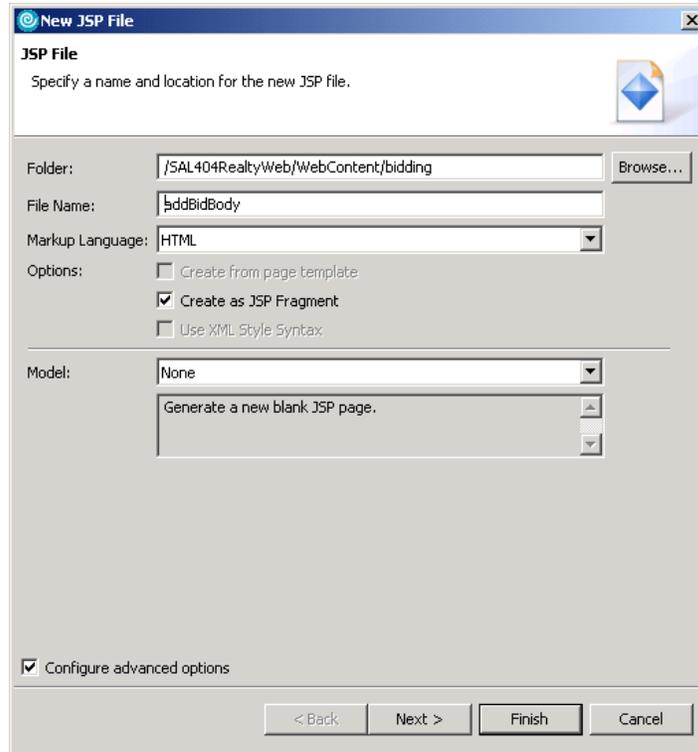
Figure 15-32   Create JSP as fragment

4. Click **Add** to select tag libraries to include in the JSP.

5. Select tag libraries from the displayed list and click **OK** as shown in Figure 15-33 on page 585.
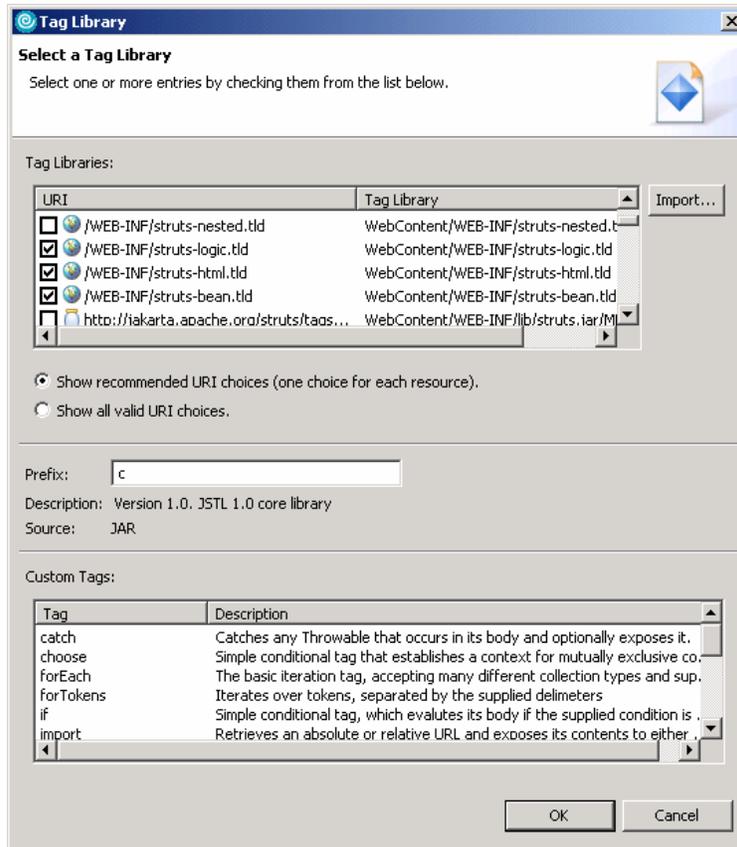
Figure 15-33   Choose tag libraries

6. Figure 15-34 on page 586 shows the tag libraries that we need to include in
   our JSP file.
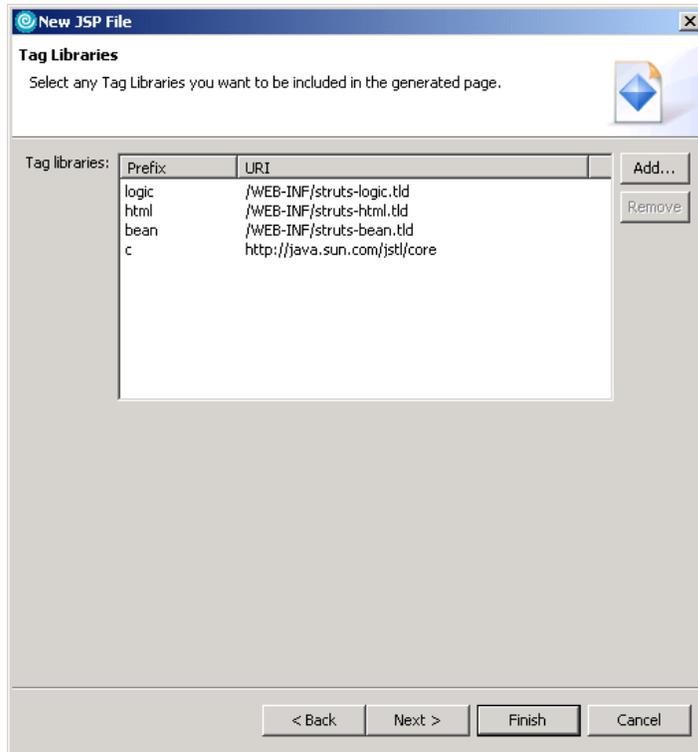
Figure 15-34   Selected tag libraries

7. Click **Finish**.

8. The addBidBoby.jspf is created and opened in the Page Designer. Replace all the existing code with the code shown in Example 15-3.

Example 15-3   Example code for addBidBody.jspf

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<BR>
<h1>Add a Bid</h1>
<HR>
<TABLE>
    <TBODY>
        <TR>
            <TD width=150>Address:</TD>
```

```
                    <TD width=250><bean:write name="propertyDTO"
                        property="streetName" />   <bean:write
                        name="propertyDTO" property="unitNumber" /> <bean:write
                        name="propertyDTO" property="buildingNumber" /></TD>
                </TR>
                <TR>
                    <TD>Postal code:</TD>
                    <TD><bean:write name="propertyDTO" property="postalCode" /></TD>
                </TR>
                <TR>
                    <TD>Municipality:</TD>
                    <TD><bean:write name="propertyDTO" property="municipality" />
                    </TD>
                </TR>

                <TR>
                    <TD>Region:</TD>
                    <TD><bean:write name="propertyDTO" property="region" /></TD>
                </TR>
                <TR>
                    <TD>Country:</TD>
                    <TD><bean:write name="propertyDTO" property="countryCode" /></TD>
                </TR>

                <TR>
                    <TD>Price:</TD>
                    <TD><fmt_rt:formatNumber minFractionDigits="2" type="number">
                        <bean:write name="propertyDTO" property="price" />
                    </fmt_rt:formatNumber></TD>
                </TR>

                <TR>
                    <TD>Additional text:</TD>
                    <TD><bean:write name="propertyDTO" property="additionalText" />
                    </TD>
                </TR>

        </TBODY>
</TABLE>

<BR>
<html:form action="/AddBid">
    <TABLE border="0">
        <TBODY>

            <TR>
                <TH>Date</TH>
                <TD><html:text property='sbidDate' /> (yyyy-MM-dd)</TD>
            </TR>
```

```
            <TR>
                <TH>Bid price</TH>
                <TD><html:text property='sbidPrice' /></TD>
            </TR>


            <TR>
                <TD><html:submit property="submit" value="Add" /></TD>
                <TD><html:reset /></TD>
            </TR>
        </TBODY>
    </TABLE>
</html:form>
```

9. The addBidBody.jspf is quite simple. It has two main purposes:

    –   It displays some details of the property that the user selected to place a bid against.

    –   It uses an HTML form so the user can enter details of their bid.

10. Save the addBidBody.jspf file.

11. Add the addBidBody.jspf to the body portion of the AddBid.jsp. The easiest way to do this is to open AddBid.jsp in the design view of the Page Designer, select **addBidyBody.jspf** from the Project Explorer and drag it onto the body section of the AddBid.jsp.

12. Save the AddBid.jsp file.

We have now completed the implementation of the presentation layer for the add bid functionality the bidding component.

15.2.4 Controller layer

Following our top-down approach, the second layer of the bidding component that we need to implement is the controller layer. Here, we implement the required action classes.

The controller layer connects the presentation layer with the business logic of the component, which is implemented in the business facade layer. In effect, the controller layer accepts a request from the presentation layer, calls the appropriate method of the business facade layer, stores any results in the request object, and returns back to the presentation layer for displaying the results. We describe how to implement the AddBidAction class.

AddBidAction class

The steps to implement this action class are:

1. Using the Project Explorer view navigate to the **SAL404RealtyWeb** project, choose **Java Resources** → **JavaSource** → **com.ibm.itso.sal404.bidding.action** → **AddBidAction.java** and double-click to open it.

2. Notice that some code has already been created from realizing the relevant action node while implementing the presentation layer.

3. Add the code highlighted in Example 15-4 to complete the AddBidAction of the controller layer. Once you have done adding the code save the file.

Example 15-4   Example code for AddBidAction

```
package com.ibm.itso.sal404.bidding.action;

import java.lang.reflect.InvocationTargetException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.beanutils.PropertyUtils;
import org.apache.log4j.Logger;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import com.ibm.itso.sal404.bidding.BidManager;
import com.ibm.itso.sal404.bidding.BidSessionData;
import com.ibm.itso.sal404.bidding.dto.BidDTO;
import com.ibm.itso.sal404.bidding.form.BidForm;
import com.ibm.itso.sal404.common.exception.ApplicationException;
import com.ibm.itso.sal404.propertycatalog.dto.PropertyDTO;
import com.ibm.itso.sal404.user.UserSessionData;

/**
 * @version 1.0
 * @author
 */
public class AddBidAction extends Action

{
    // Configure Log4J Logger
    private static Logger logger = Logger.getLogger(AddBidAction.class);
```

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
{
    logger.info("Entry: execute");
    ActionErrors errors = new ActionErrors();
    ActionForward forward = new ActionForward(); // return value
    BidForm bidForm = (BidForm) form;
    BidManager manager = new BidManager();
    PropertyDTO prop = (PropertyDTO) request.getSession().getAttribute(
            "propertyDTO");
    UserSessionData userSessionData = (UserSessionData) request
            .getSession().getAttribute("userSessionData");

    try
    {

        BidDTO bidDTO = populateBidDTO(form, prop, userSessionData);
        logger.info("BidDTO populated");
        addBid(request,errors,bidDTO);
        logger.info("After addBid");
//          request.setAttribute("propertyId",prop.getId().toString());
    }
    catch (Exception e)
    {

        // Report the error using the appropriate name and ID.
        errors.add("name", new ActionError("id"));

    }

    // If a message is required, save the specified key(s)
    // into the request for use by the <struts:errors> tag.

    if (!errors.isEmpty())
    {
        logger.info("Errors found");
        logger.info(errors);
        saveErrors(request, errors);
        forward = mapping.findForward("failure");
    }
    else
    {
        // Write logic determining how the user should be forwarded.
        forward = mapping.findForward("success");
    }

    // Finish with
```

```
            logger.info("Exit: execute");
            return (forward);

    }

    /**
     *
     * @param form
     * @param prop
     * @param userSessionData
     * @return
     */
    public BidDTO populateBidDTO(ActionForm form, PropertyDTO prop,
            UserSessionData userSessionData)
    {
        logger.info("ENTRY: populateBidDTO");
        BidForm bidFormBean = (BidForm) form;
        BidDTO bidDTO = new BidDTO();
        try
        {
            PropertyUtils.copyProperties(bidDTO, bidFormBean);
        }
        catch (IllegalAccessException e1)
        {
            e1.printStackTrace();
        }
        catch (InvocationTargetException e1)
        {
            e1.printStackTrace();
        }
        catch (NoSuchMethodException e1)
        {
            e1.printStackTrace();
        }

        bidDTO.setPropertyId((prop.getId()));
        bidDTO.setBidderId(userSessionData.getLogInUser().getId());
        bidDTO.setPropertySeller(prop.getSeller());
        bidDTO.setStatus("open");
        bidDTO.setBidDate(bidFormBean.getBidDate());
        bidDTO.setBidPrice(bidFormBean.getBidPrice());

        logger.info("EXIT:  populateBidDTO");
        return bidDTO;
    }

    /**
     *
     * @param request
```

```
 * @param errors
 * @param bidDTO
 */
public void addBid(HttpServletRequest request, ActionErrors errors,
        BidDTO bidDTO)
{
    BidSessionData bidSessionData = (BidSessionData)
request.getSession().getAttribute("bidSessionData");

    try
    {
        bidSessionData.create(bidDTO);

    }
    catch (ApplicationException ae)
    {
        // Report the error using the appropriate name and ID.
        errors.add("Bid", new ActionError(ae.getStrutsMessage()));
    }
}
}
```

Note: Saving the AddBidAction class generates errors at this stage
because it refers to other classes that we have not yet created.

15.2.5 Business facade layer

Once we have finished implementing the presentation and the controller layers of
the bidding component, we have essentially finished working with the front-end of
the component. Before we start working on the back-end layers, we explain the
purpose of the business facade layer, and describe how it should be
implemented. We show the actual implementation of the business facade layer
later on in 15.2.8, "Putting everything together" on page 607.

The business facade layer plays the role of a bridge between the front-end and
the back-end of the application. It is implemented using a manager class, which
exposes the business methods required for the implementation of the
component. In the case of the bidding component, the BidManager class has the
following public methods:

► listBids(Integer propertyId)
► addBid(BidDTO bid)
► modifyBid(BidDTO bid)

addBid(BidDTO bid) method

This is very simple and the execution flow is as follows:

1. An instance of the InsertBidDAO is created. This will be used to access the database. More on the implementation of the InsertBidDAO can be found in "Generate the data access object classes" on page 601.

2. The InsertBidDAO is executed.

15.2.6 Domain layer

We now create the data transfer object classes needed for our domain layer. The bidding component needs only one DTO called BidDTO.

BidDTO

The steps to create the BidDTO are:

1. Using the Project Explorer navigate to the **Other Projects** → **SAL404RealtyJava** project, then right-click the package **com.ibm.itso.sal404.bidding.dto** and choose **New** → **Class**.

2. Enter `BidDTO` as the class name and click **Finish**.

3. Add the code highlighted in Example 15-5 to add all the necessary attributes to your DTO.

Example 15-5   Attributes for the BidDTO class

```
private String id;
    private Integer propertyId;
    private Integer bidderId;
    private String status;
    private Integer propertySeller;
    private Date bidDate;
    private BigDecimal bidPrice;
```

4. While inside the editor, right-click and select **Source** → **Generate Getter and Setter...** as shown in Figure 15-35 on page 594.
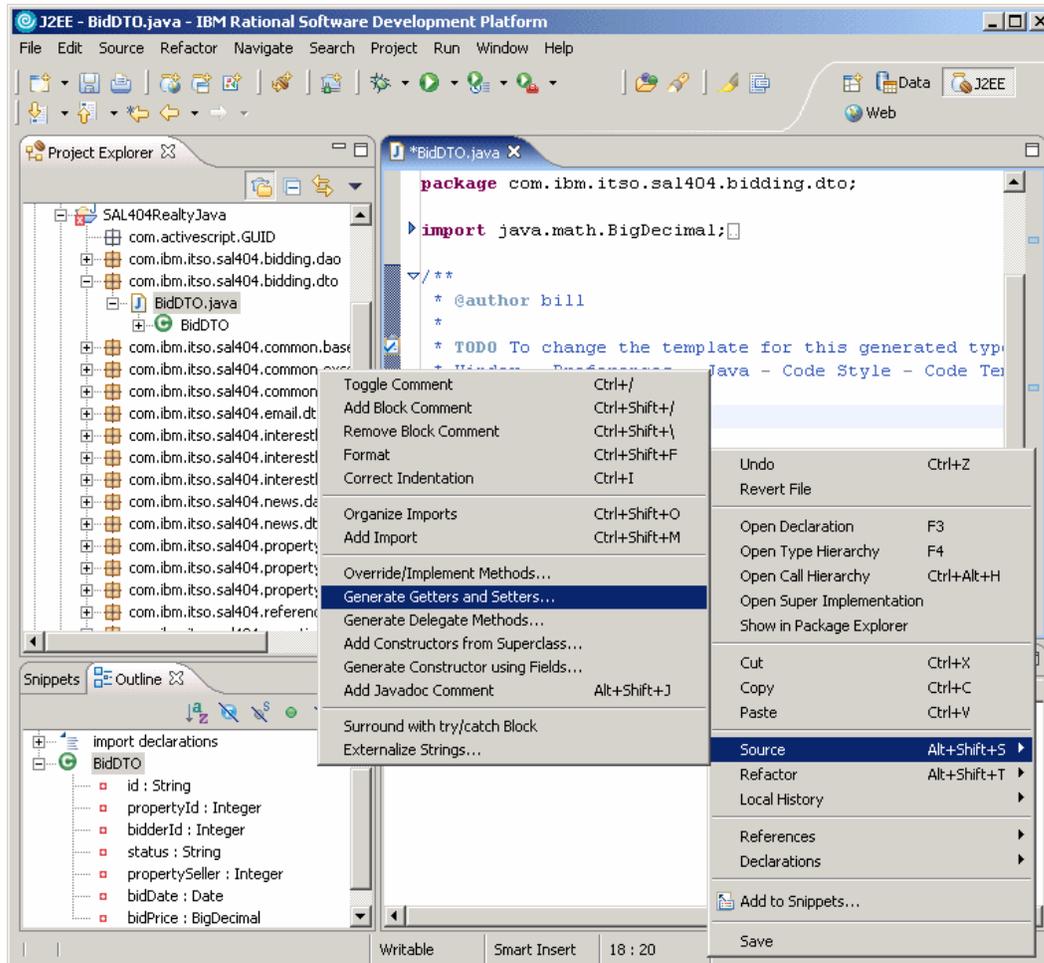
Figure 15-35   Generate getters and setters

5. When a new dialog box appears, click **Select All,** then click **OK** as shown in Figure 15-36 on page 595.
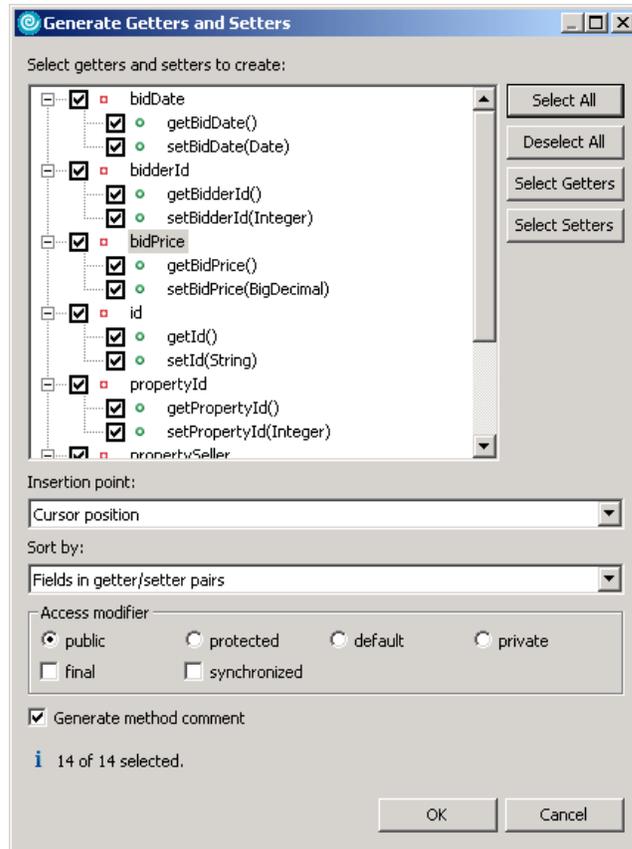
Figure 15-36   Generate Getters and Setters for BidDTO class

6. Save and close the BidDTO modified file.

Your BidDTO has now been generated.

15.2.7 Data access layer

The bidding component needs to access the database in order to retrieve information about properties and on the interest customers have shown on them. Therefore, we need to build data access objects to implement this requirement. Rational Web Developer provides wizards that you can use to build this functionality. First of all, we need to create the SQL statements necessary to retrieve data from the database, and then we need to create the data access objects that will execute these statements and return the data for further use.

Create the SQL statements

Before we create the data access objects, we need to have the SQL statements created. We demonstrate how to create a SQL statement using the SQL Builder wizard.

InsertBid SQL statement

This SQL statement will enable us to crete a new row in the BID table. To create the statement using Rational Web Developer, follow these steps:

1. Switch to the Data perspective and choose **File** → **New** → **SQL Statement**. A dialog box opens as shown in Figure 15-37.

2. Choose **INSERT** from the SQL statement drop-down menu, check the **Be guided through creating an SQL statement** option, uncheck **Create a new database connection** and click **Next**.
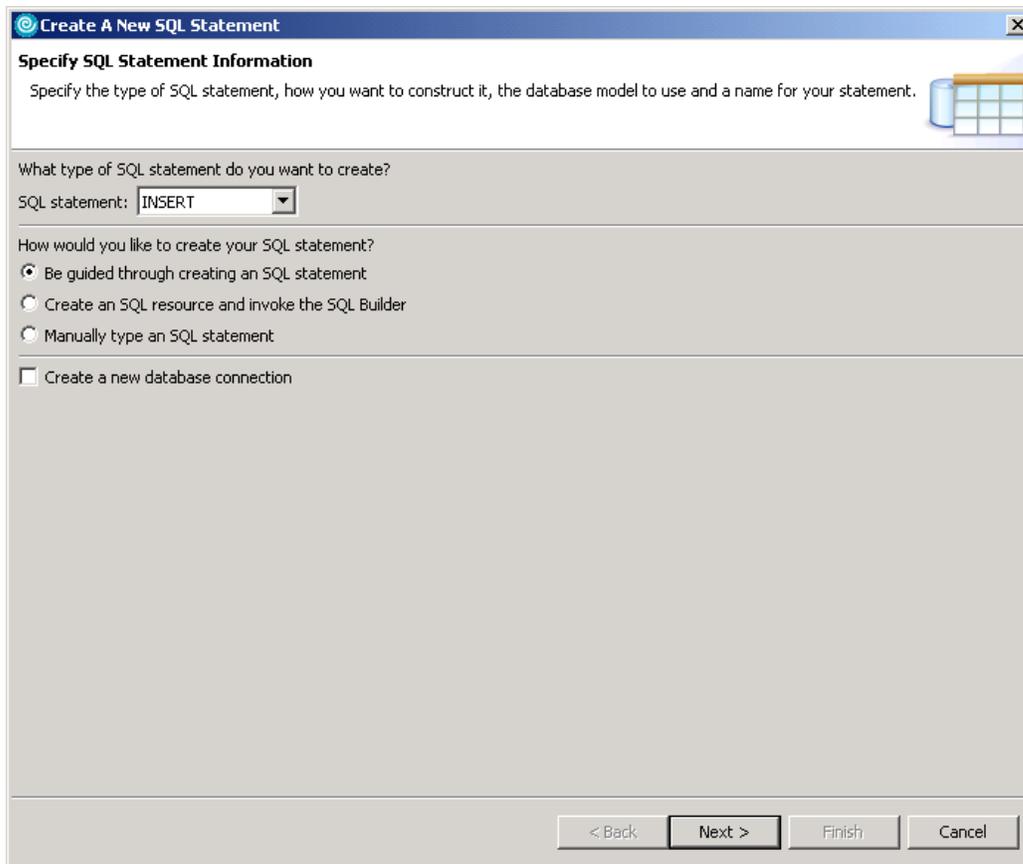
Figure 15-37   Create an Insert SQL statement

3. Click **Browse** to select an existing database model.

4. A new dialog box opens as shown in Figure 15-38. Choose
   SAL404RealtyEJB → **ejbModule** → **META-INF** → **backends** →
   DB2UDBNT_V8_1 → **SAL404(DB2 Universal Database V8.1)** and click **OK**.
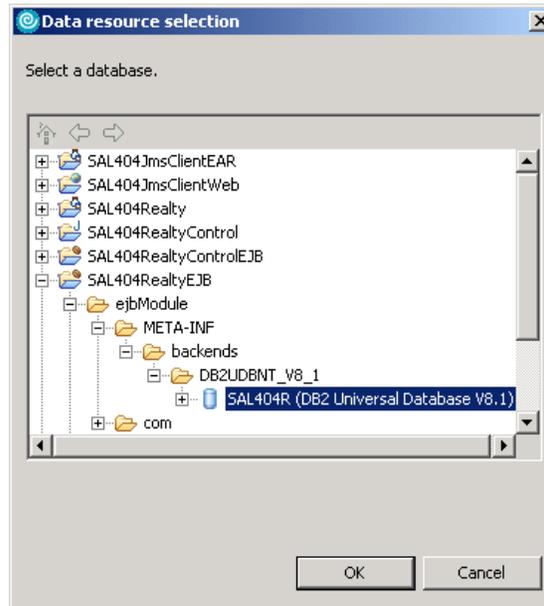
Figure 15-38   Data resource selection

5. You will be returned to the previous dialog box, notice that SAL404R appears
   as the database model used. Click **Next**.

6. Enter `InsertBid` in the SQL statement name textbox and click **Next** as shown
   in Figure 15-39 on page 598.
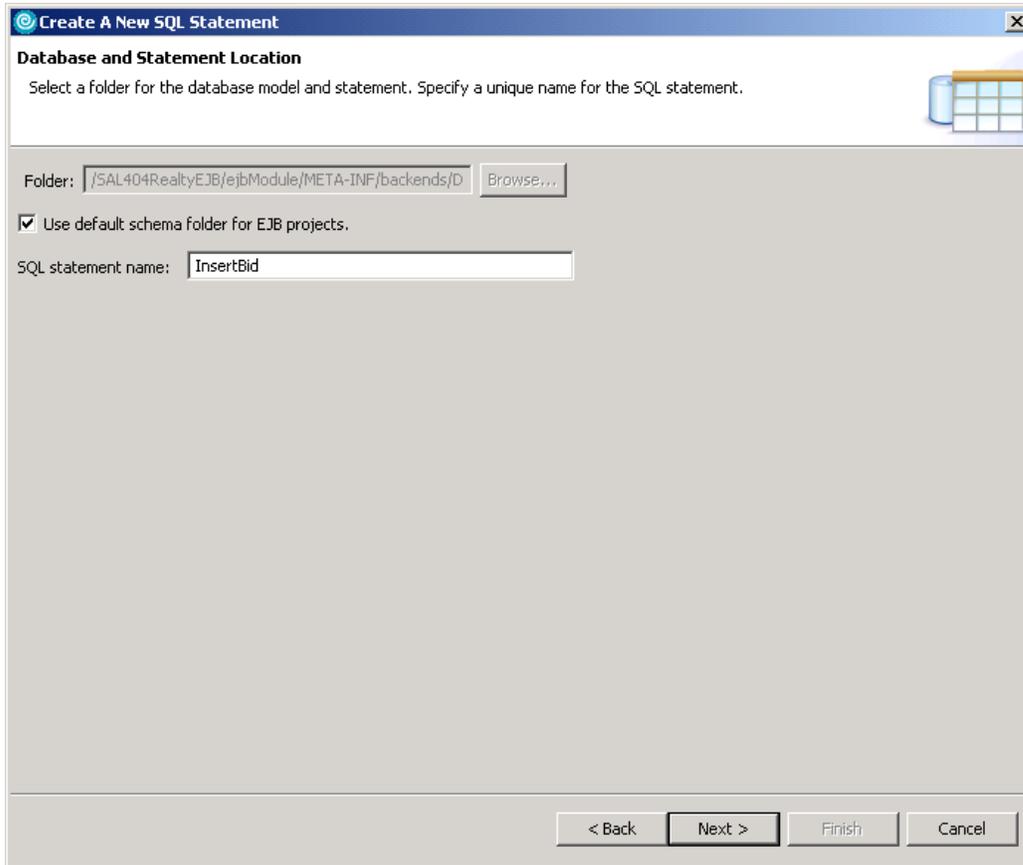
Figure 15-39   Enter SQL query name

7. From the **Tables** tab select **DB2ADMIN.BID** and click **>** to move the table to the selected table field. Figure 15-40 on page 599 shows an example of the selected table.
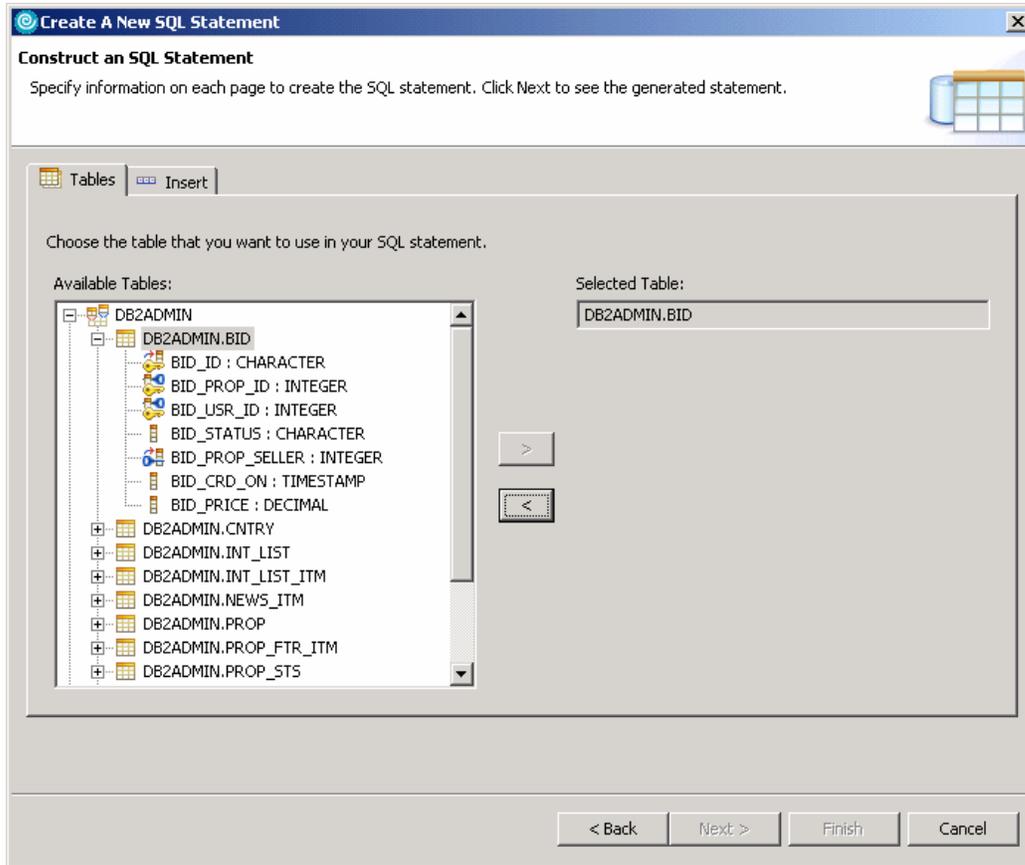
Figure 15-40   Select table for insert statement

8. Click the **Insert** tab and enter values for each column in the insert statement. We provide names of parameters that will be replaced by actual values when our query runs. See Figure 15-41 on page 600 for an example. Click **Next**.

Figure 15-41   Provide values for the insert statement

9. Figure 15-42 on page 601 shows the completed SQL statement. Click **Finish**.
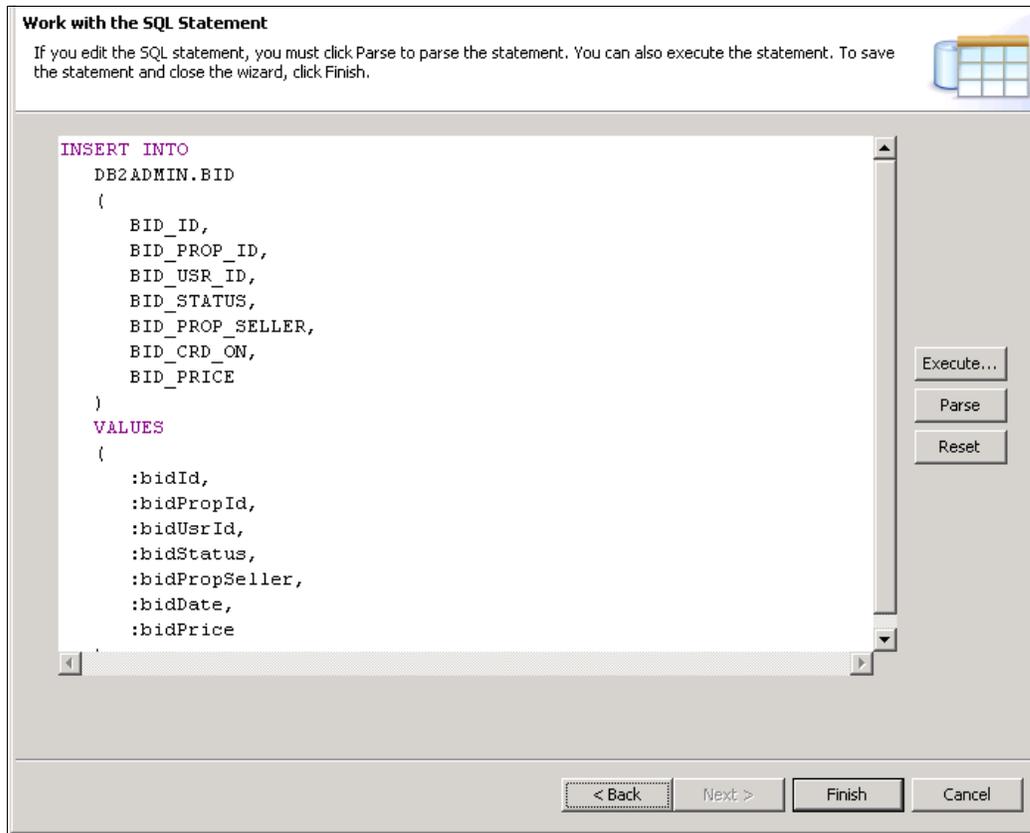
Figure 15-42   Completed SQL statement for insert bid

10. You are returned to your workspace where the new SQL statement is opened in the SQL statement editor.

You have now completed the creation of InsertBid SQL statement.

Generate the data access object classes

Once you have created an SQL statement you need to create the data access object classes that will use the statement to perform operations on the database. Our completed bidding component contains the following DAO classes:

- ► InsertBidDAO
- ► UpdateBidDAO
- ► ListBidsForPropertyDAO
- ► ListBidsForPropertyDAORow

Here, we demonstrate the steps necessary to create one of these DAO classes, InsertBidDAO using Rational Web Developer. You should follow the same steps in order to create the rest of the DAO classes:

1. Switch to the Data perspective and select **SAL404RealtyEJB** →
   ejbModule → **META-INF** → **backends** → **DB2UDBNT_V8_1** →
   SAL404(DB2 Universal Database V8.1) → **Statements.** Right-click
   InsertBid and choose **Generate Java Bean** as shown in Figure 15-43.
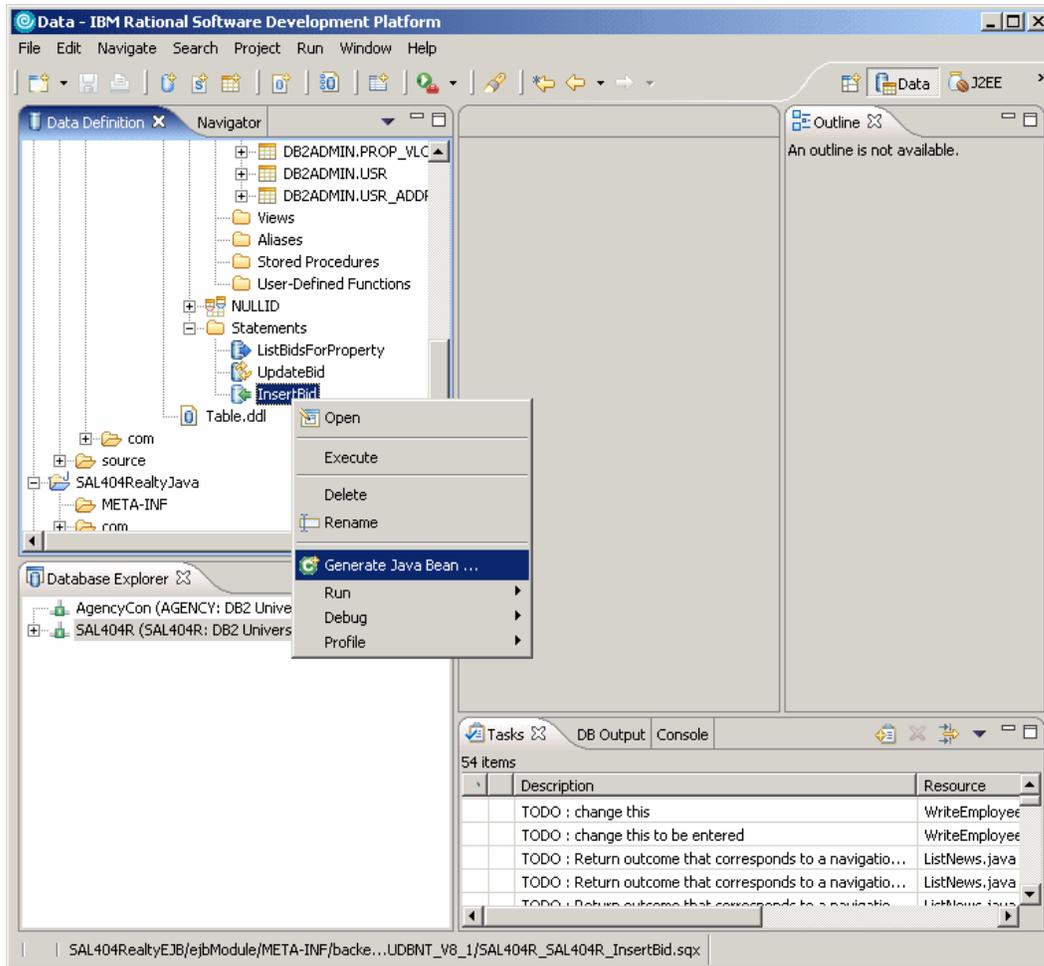
Figure 15-43   Generate Java BEAN for a SQL statement

2. A dialog box like the one shown in Figure 15-44 on page 603 will appear.
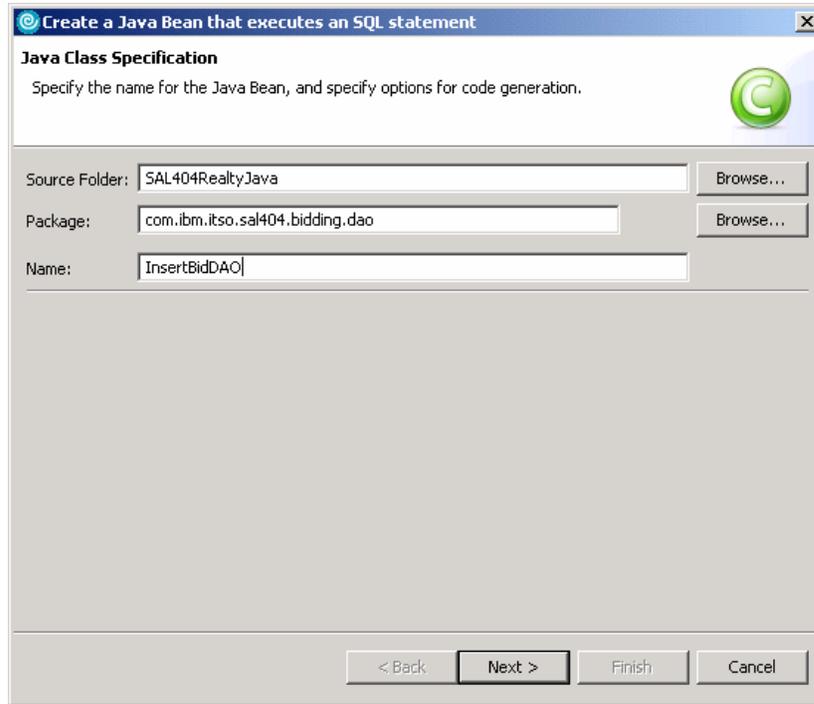
Figure 15-44   Generate a DAO class

3. Click to **Browse** a set the source folder to **SAL$04RealtyJava.**

4. You will be returned to the previous dialog box, click to **Browse** for a package, a set the package to **com.ibm.itso.sal404.bi**Generate a DAO class (Step 4).

5. You will be returned to the previous dialog box, enter `InsertBidDAO` in the Name textbox and click **Next**.

6. A new dialog box will open, choose **Use DataSource Connection** and enter in `jdbc/sal404` as your DataSourcee/JNDI Name.

7. Select **Inside the execute() method** and enter a user ID of `db2admin` and supply the correct password.`C`lick **Next** as shown in Figure 15-45 on page 604.
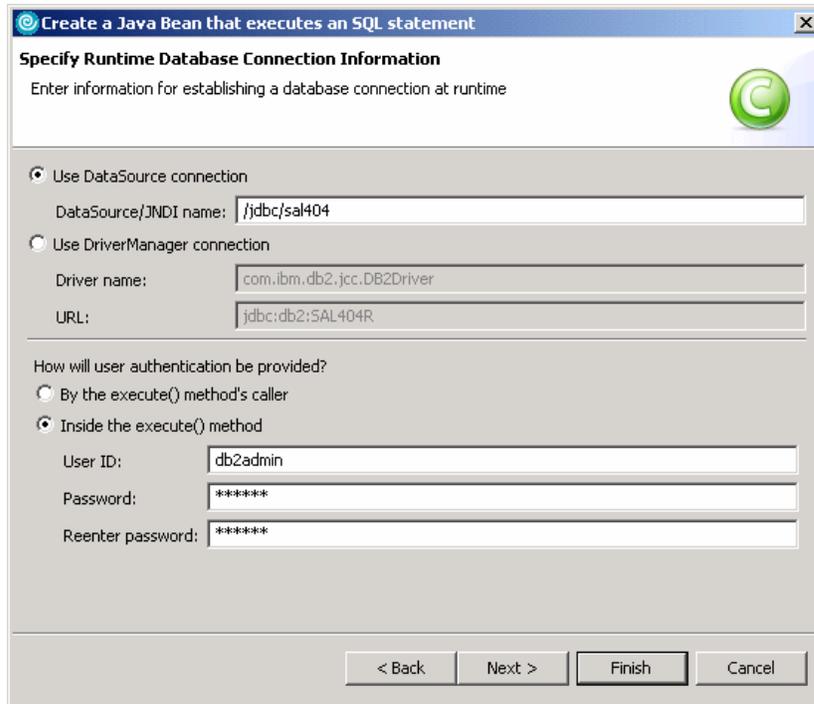
Figure 15-45   Specify runtime DataSource information for the DAO class

8. Review the details of the class to be generated and click Finish. See Figure 15-46 on page 605 for an example.
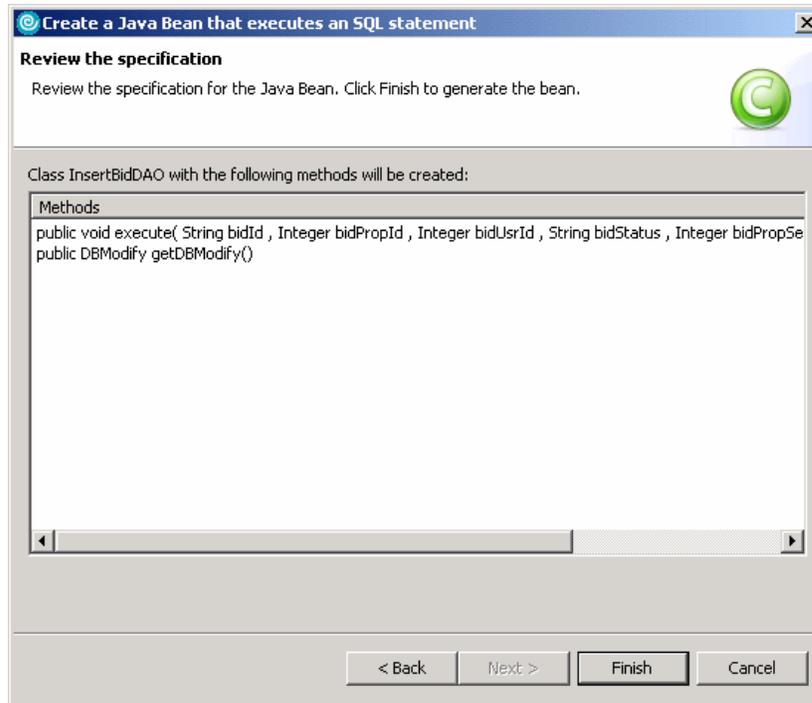
Figure 15-46   Complete generation for a DAO class

9. Open the **J2EE perspective** and navigate to the
   com.ibm.itso.sal404.bidding.dao package.

10. Double-click the **InsertBidDAO.java** file to open it, locate the execute()
    method and remove the lines that set the username and the password.
    Example 15-6 shows the generated code of the execute() method before our
    changes. We do not want to supply hard coded user name and password as
    these will be provided at runtime by our DataSource connection. The lines to
    delete are:

```
modify.setUsername("db2admin");
modify.setPassword("db2adminpassword");
```

Example 15-6   InsertBidDAO execute() method with inline user name and password

```
public void execute(String bidId, Integer bidPropId, Integer bidUsrId,
    String bidStatus, Integer bidPropSeller,
    java.sql.Timestamp bidDate, java.math.BigDecimal bidPrice)
    throws SQLException {
try {
```

```
            modify.setUsername("db2admin");
            modify.setPassword("db2adminpassword");
            modify.setParameter("bidId", bidId);
            modify.setParameter("bidPropId", bidPropId);
            modify.setParameter("bidUsrId", bidUsrId);
            modify.setParameter("bidStatus", bidStatus);
            modify.setParameter("bidPropSeller", bidPropSeller);
            modify.setParameter("bidDate", bidDate);
            modify.setParameter("bidPrice", bidPrice);
            modify.execute();
        }

        // Free resources of modify object.
        finally {
            modify.close();
        }
    }
```

11. We also want to use the GUID generator key generation facility discussed in
    9.2.6, "Entity beans" on page 335 to set the bid id for our new bid. Replace
    the line in the execute() method:

```
modify.setParameter("bidId", bidId);
```

with code that generates a key:

```
KeyGenerator keyGenerator = new KeyGenerator();
modify.setParameter("bidId", keyGenerator.getKeyString());
```

12. We also want to set the bidDate to a SQL date instead of the supplied
    java.util.Date current date and time, so we replace the line:

```
modify.setParameter("bidDate", bidDate);
```

with code to convert the date:

```
java.sql.Date sqlDate = new java.sql.Date(bidDate.getTime());
modify.setParameter("bidDate", sqlDate);
```

13. Example 15-7 shows our new execute method.

Example 15-7   Modified execute() method for the DAO

```
public void execute(String bidId,
            Integer bidPropId, Integer bidUsrId, String bidStatus,
            Integer bidPropSeller, java.util.Date bidDate, java.math.BigDecimal
bidPrice)
            throws SQLException
    {
        try
        {
            java.sql.Date sqlDate = new java.sql.Date(bidDate.getTime());
```

```
            KeyGenerator keyGenerator = new KeyGenerator();
            modify.setParameter("bidId", keyGenerator.getKeyString());
            modify.setParameter("bidPropId", bidPropId);
            modify.setParameter("bidUsrId", bidUsrId);
            modify.setParameter("bidStatus", bidStatus);
            modify.setParameter("bidPropSeller", bidPropSeller);
            modify.setParameter("bidDate", sqlDate);
            modify.setParameter("bidPrice", bidPrice);
            modify.execute();
        }

        // Free resources of modify object.
        finally
        {
            modify.close();
        }
    }
```

14. We also want to change the generated initializer() method to remove the hard coded DataSource name. Replace the line that reads:

```
modify.setDataSourceName("/jdbc/sal404");
```

Use the following code which obtains the DataSource from the application properties of our Sal404 application as a replacement:

```
modify.setDataSourceName(AppProperties.getInstance().getProperty("dataSourc
eJNDIName"));
```

The data access object class for the InsertBid statement is now complete.generated.

15.2.8 Putting everything together

By this point, we have implemented all the front-end layers (presentation and controller) and all the back-end layers (domain layer, data access layer) of the bidding component. We have also done some work on the integration of the front-end and the back-end layers in the business facade layer. We now create the BidManager class, which acts as the business facade layer of the component. We also create the associate BidSessionData class.

Create the BidManager
This section describes how to create the BidManager and shows the implementation of the add bid functionality. See our redbook sample code for the complete BidManager class that also implements bid updates and lists.

1. Using the Project Explorer navigate to the **Other Projects** →
   SAL404RealtyJavaControl project, then right-click the package
   com.ibm.itso.sal404.bidding and choose **New** → **Class**.

2. Enter `BidManager` as the class name and click **Finish**.

3. Example 15-8 shows the addBid(BidDTO) method. After you have finished
   adding the code, save the file.

Example 15-8   addBid(BidDTO) method

```
/**
     * @param bidItem
     * @throws ApplicationException
     */
    public void addBid(BidDTO bid) throws ApplicationException
    {
        InsertBidDAO bidDAO = new InsertBidDAO();
        try
        {
            bidDAO.execute("DummyId", bid.getPropertyId(), bid.getBidderId(),
bid
                    .getStatus(), bid.getPropertySeller(), bid.getBidDate(),
bid.getBidPrice());
        }
        catch (Exception e)
        {

            logger.info("DBerror " + e.getMessage());

            logger.error("addBid: " + e.getMessage());

            ApplicationException ae = new ApplicationException();
            ae.setStrutsMessage("bid.error.addBid");
            throw ae;
        } finally
        {
            logger.info("EXIT:  addBid");
        }

    }
```

4. The code shown in Example 15-8 does the following key tasks:

   a. Creates an instance of the InsertBidDAO.

      ```
      InsertBidDAO bidDAO = new InsertBidDAO();
      ```

   b. Invokes the execute() method of the InsertBidDAO providing input
      parameters extracted from the BidDTO.

```
bidDAO.execute("DummyId", bid.getPropertyId(), bid.getBidderId(),
bid.getStatus(), bid.getPropertySeller(), bid.getBidDate(),
bid.getBidPrice());
```

c. The rest of the code is concerned with logging and error handling.

> **Note:** Our current implementation of the add bid functionality does not
> satisfy the full requirements specified for our bidding component. We do
> not check that no other open bids exist for the property and we do not
> update the property status to reflect that a bid has been made.

We have now finished building the bidding component.

Create the BidSessionData

This section describes how to create the BidSessionData class and describes
the implementation of the create(BidDTO bid) method. See our redbook sample
code for the complete BidSessionData class.

1. Using the Project Explorer navigate to the **Other Projects** →
   SAL404RealtyJavaControl project, then right-click the package
   com.ibm.itso.sal404.bidding and choose **New** → **Class**.

2. Enter **BidSessionData** as the class name and click **Finish**.

3. Example 15-9 shows the create(BidDTO bid) method.

Example 15-9   create(BidDTO bid) method

```
/**
    *
    * @param bidItem
    * @throws ApplicationException
    */
   public void create(BidDTO bid) throws ApplicationException
   {
       getBidManager().addBid(bid);

       // clear the list
       setBidList(null);
   }
```

4. The code shown in Example 15-9 does the following key tasks:

a. Obtains a reference to the BidManager.

```
getBidManager()
```

b. Invokes the addBid() method of the BidManager providing input parameters extracted from the BidDTO.

```
addBid(bid)
```

c. Clears the current list of bids held in the session data so that the user will se an updated list of bids.

```
setBidList(null);
```

15.2.9 Testing the bidding component

In this section we describe how to test the add bid functionality and also describe the application flow that implements this function.

1. To test the bidding you must be logged on to the Sal404 sample application.

2. Log on to the **Properties** link from the SAL404RealtyWeb home page. The searchByCriteria page will be displayed.

3. Enter some search criteria and display a list of properties.

4. Select a property from the list and the Property Details page should be displayed as shown in Figure 15-47 on page 611.
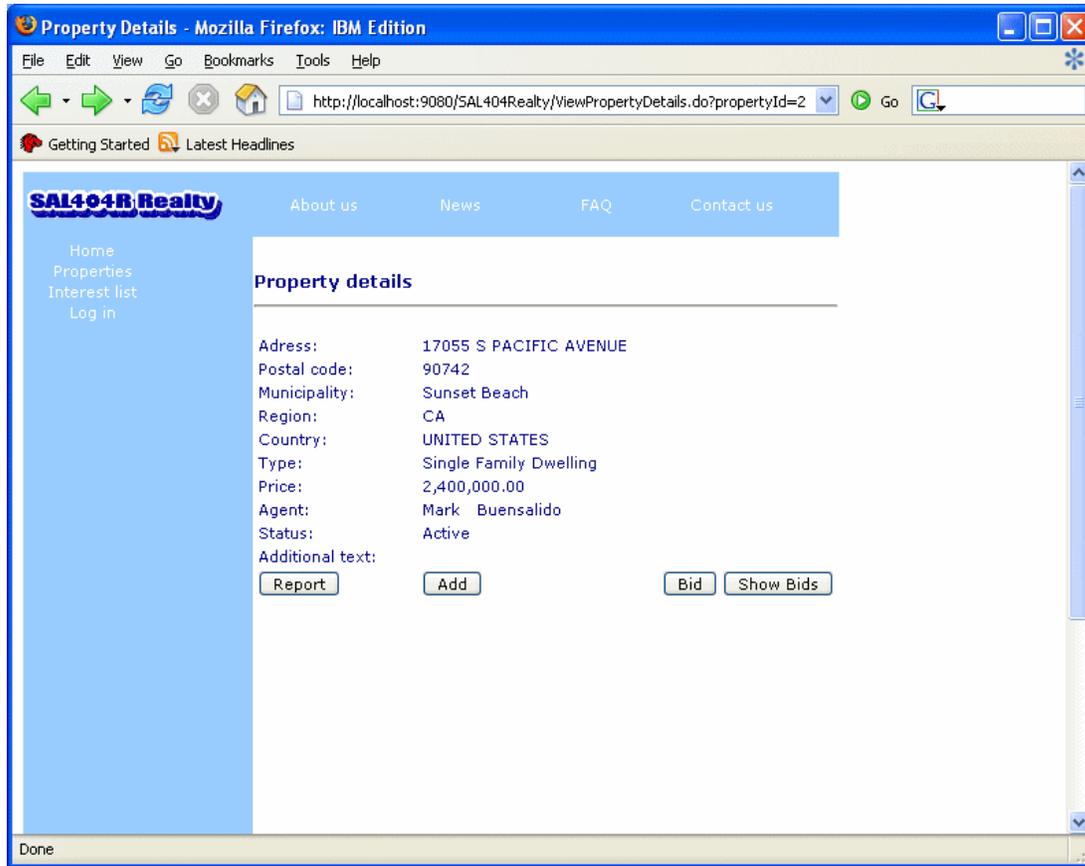
Figure 15-47   Property details page

5. The property details page is implemented by the viewPropertyDetails.jsp
   which includes the viewPropertyDetailsBody.jspf. We altered the
   viewPropertyDetailsBody.jspf to display action buttons to allow users to add
   and list bids. Example 15-10 shows the code in viewPropertyDetailsBody.jspf
   that displays the Bid button.

Example 15-10   viewPropertyDetailsBody.jspf bid button

```
<c:if test="${propertyDTO.statusDescription == \"Active\"}">
             <TD>
             <form action="/SAL404Realty/bidding/SetupBid.do"><INPUT
type="hidden"
             name="propertyId" value="${propertyDTO.id}"> <INPUT
type="submit"
```

```
                name="name" value="Bid"></form>
                </TD>
</c:if>
```

> **Note:** The code shown in Example 15-10 is part of a larger structure that
> test to make sure that the application user is logged in. It tests whether the
> property staus is Active, but we have not implemented the bidding
> requirement to make sure that the user placing a bid is not the property
> seller or agent.

6. Click **Bid**. As you can see from Example 15-10 on page 611 this submits the
   form with an action of SetupBid.do.

7. SetupBid.do is a Struts action implemented by the SetupBidAction class.This
   action uses the PropertyCatalogManager to obtain details of the property that
   the user has chosen and places a PropertyDTO in the HTTP session. This is
   so that the AddBid.jsp can display details of the property. Example 15-11
   shows the execute() method of SetupBidAction. If the execute() method
   succeeds the action will forward to the AddBid.jsp.

Example 15-11   SetupBidAction execute() method

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception
    {

        ActionErrors errors = new ActionErrors();
        ActionForward forward = new ActionForward(); // return value

        try
        {

            PropertyCatalogManager manager = new PropertyCatalogManager();
            PropertyDTO prop = manager.populatePropertyDetails(new
Integer(request.getParameter("propertyId")));
            request.getSession().setAttribute("propertyDTO", prop);

        }
        catch (Exception e)
        {

            e.printStackTrace();
            // Report the error using the appropriate name and ID.
            logger.error(e.getMessage());
            errors.add("Bid", new ActionError("No property found"));
```

```
        }

        // If a message is required, save the specified key(s)
        // into the request for use by the <struts:errors> tag.

        if (!errors.isEmpty())
        {
            logger.info("errors found in setup bid");
            saveErrors(request, errors);
            forward = mapping.findForward("failure");

        }
        else
        {

            forward = mapping.findForward("success");

        }

        // Finish with
        return (forward);

    }
```

> **Note:** The property catalog component still places DTOs directly in the
> session, which is not our recommended approach. We prefer to use the
> session data technique as described in 5.3.4, "Session data" on page 219,
> but we have not yet altered the property catalog component to use a
> session data class.

8. Figure 15-48 on page 614 shows the add bid page. This is implemented by
   the addBid.jsp which includes the addBidBody.jspf.
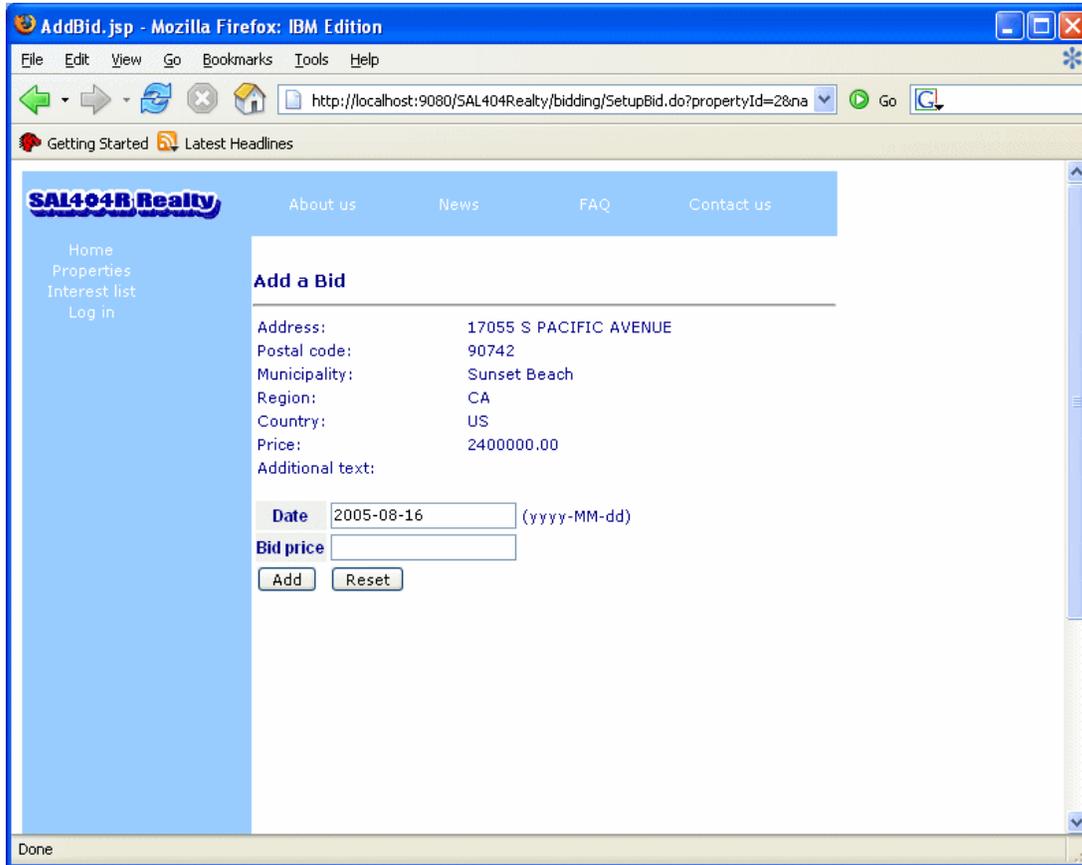
Figure 15-48   Add bid page

9. Enter a Bid price and click **Add**.

10. We use Struts validation as discussed in 11.7.1, "Using the Validator in forms and JSPs" on page 464 to validate the bid form on the add bid page. Both a bid date and a bid price are required. We also validate that the date enter can be converted to a valid date using the pattern yyyy-MM-dd and that the enter price can be converted to a float. Example 15-12 shows the validation rules for the bid form that we created in the Struts validation.xml file.

Example 15-12   Struts validation of the bid form.

```
<form name="bidForm">
    <field property="sbidPrice" depends="required,float">
        <arg0 key="error.bidPrice.required" />
    </field>
```

```
          <field property="sbidDate" depends="required,date">
            <arg0 key="error.date.required" />
            <var>
                <var-name>datePatternStrict</var-name>
                <var-value>yyyy-MM-dd</var-value>
            </var>
          </field>
        </form>
```

11. Using Struts validation means that the validate() method of our Struts form bean class BidForm does not need to do extra validation. It simply converts the string date and price fields into their correct types as shown in Example 15-13.

Example 15-13   BidForm validate() method

```
public ActionErrors validate(ActionMapping mapping,
          HttpServletRequest request)
    {

        //      ActionErrors errors = new ActionErrors();
        ActionErrors errors = super.validate(mapping, request);

        logger.info("Number of errors is " + errors.size());
        // Validate the fields in your form, adding
        // adding each error to this.errors as found, e.g.
        if (errors.isEmpty())
        {
            logger.info("No errors so parsing date and price ");
            try
            {

                setBidDate(formatter.parse(getSbidDate()));
            }
            catch (ParseException e)
            {
                e.printStackTrace();
                errors.add("Bid", new ActionError("error.bid.invalidDate"));
            }

            setBidPrice(new BigDecimal(getSbidPrice()));
        }
        return errors;

    }
```

12. After validation is completed the Struts action AddBidAction is called. The execute() method of the action does the following key tasks:

a. Retrieves the bid form.

```
BidForm bidForm = (BidForm) form;
```

b. Retrieves the user session data.

```
UserSessionData userSessionData = (UserSessionData) request
                .getSession().getAttribute("userSessionData");
```

c. Retrieves the property DTO from the session.

```
PropertyDTO prop = (PropertyDTO) request.getSession().getAttribute(
                "propertyDTO");
```

d. Calls populateBidDTO(form, prop, userSessionData) to populate fields in a bid DTO from the contents of the bid form, the property DTO and the user session data.

```
BidDTO bidDTO = populateBidDTO(form, prop, userSessionData);
```

e. Calls the addBid(request,errors,bidDTO) method to create a new bid.

```
addBid(request,errors,bidDTO);
```

13. The addBid(request,errors,bidDTO) method of the AddBidAction does the following key tasks:

a. Retrieves the bid session data.

```
BidSessionData bidSessionData = (BidSessionData)
request.getSession().getAttribute("bidSessionData");
```

b. Calls the bidSessionData.create(bidDTO) method which we previously documented in Example 15-9 on page 609.

```
bidSessionData.create(bidDTO);
```

14. The bidSessionData.create(bidDTO) method calls the BidManager method addBidd(BidDTO) which we previously documented in Example 15-8 on page 608.

15. If the bid creation succeeds, the AddBidAction forwards to the Struts ViewPropertyDetails action to redisplay the property details page.

A

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/`SG246500

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246500.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name   *Description*
SG246500.zip   Zipped code samples

617

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 20 MB minimum
Operating System: Windows
Processor: Intel Pentium III 800MHZ processor minimum or higher
Memory: Minimum 768 MB available RAM - 1GB recommended

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

The files contained in the expanded zip file are:

► In folder `\material\database`

  – DBSCRIPT.SQL

    See 4.7.1, "Running the sample database script" on page 174 for details of using this script file. Also refer to 13.3, "The new data model" on page 532 for details of the data model that this script implements.

  – NEWDBSCRIPT.SQL

    Refer to 13.3, "The new data model" on page 532 for details of the data model that this script implements.

  – OLDDBSCRIPT.SQL

    Refer to 13.2, "The Sal301 data model" on page 530 for details of the data model that this script implements

► In folder `\material\ear`

  – SAL404JmsClientEAR.ear

  – SAL404Realty.ear

    This is the packaged EAR file for our Sal404 sample application. See 4.7.5, "Installing the Sal404 application EAR" on page 190 for details of working with this EAR file.

  – SAL404TestServices.ear

► In folder `\material\projects`

  – Sal404Interchange.zip

    This is a Rational Software Development Platform project interchange file containing all our Sal404 sample code. It can be imported to a Rational Software Development Platform workspace as described in 4.9, "Installing Sal404 code in Rational Web Developer" on page 202

See 4.7, "Deploying the sample application" on page 174 for details on how to use the additional material to install our redbook samples.

Abbreviations and acronyms

ACID	Atomicity, Consistency, Isolation and Durability
ARM	Application Response Measurement
BMP	bean-managed persistence
BPEL4WS	Business Process Execution Language for Web Services
BSF	Bean Scripting Framework
CBD	Component Based Development
CGI	Common Gateway Interface
CMP	container-managed persistence
CORBA	Common Object Request Broker Architecture
CosNaming	Common Object Request Broker Architecture (CORBA) naming service
CRUD	create, read, update, delete
CSS	cascading style sheets
CVS	Concurrent Version Systems
DAD	document access definition
DADX	document access definition extension
DAO	data access object
DHTML	Dynamic HTML
DMS	data mediator service
DTD	Document Type Definition
DTO	data transfer object
EAR	Enterprise Application Archive
EGL	Enterprise Generation Language
EIS	Enterprise Information Systems
EJB	Enterprise JavaBean
EJS	Enterprise Java Server
EMF	Eclipse modelling framework
ERD	entity relationship diagram
ESB	Enterprise Service Bus
GUI	Graphical user interface
GUID	globaly unique ID
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IMAP	Internet Message Access Protocol
INS	Interoperable Naming Service
ITSO	International Technical Support Organization
J2C	Java 2 Connector Architecture
J2EE	Java 2 Platform, Enterprise Edition
J2SE	Java 2 Platform, Standard Edition
JAAS	Java Authentication and Authorization Service
JACC	Java Authorization Contract for Containers
JAF	Java Activation Framework
JAR	Java archive
JAXM	Java API for XML Messaging
JAXP	Java API for XML Processing
JAXR	Java API for XML Registries
JCA	J2EE Connector Architecture

621

JDBC	Java Database Connectivity		**POP**	Post Office Protocol
JDK	Java Development Kit		**POP3**	Post Office Protocol Version 3
JDO	Java Data Object		**RAR**	Resource Adapter Archive
JDT	Java Development Tools		**RMI**	Remote method invocation
JMS	Java Message Service		**RMIC**	RMI Compiler
JMX	Java Management Extensions		**RUP**	Rational Unified Process
JNDI	Java Naming and Directory Interface		**SCM**	software configuration management
JNLP	Java Network Launching Protocol		**SDO**	Service Data Objects
			SIB	service integration bus
JSEE	Java Secure Socket Extension		**SIBWS**	service integration bus Web services
JSF	JavaServer Faces		**SLES**	SuSE Linux Enterprise Server
JSP	JavaServer Pages		**SMTP**	Simple Mail Transfer Protocol
JSTL	JavaServer Pages Standard Tag Library		**SOA**	Service-oriented architecture
			SSL	Secure Sockets Layer
JTA	Java Transaction API		**SSO**	single signon
JTS	Java Transaction Service		**TLS**	Transport Layer Security
JVM	Java Virtual Machine		**TUI**	text user interface
LDAP	Lightweight Directory Access Protocol		**UML**	Unified Modeling Language
			URI	Uniform Resource Identifier
LTPA	Lightweight Third Party Authentication		**URL**	Uniform Resource Locator
			XSD	XML Schema Definition
MDB	Message Driven Bean		**UCS**	Universal Character Set
MIME	Multipurpose Internet Mail Extensions		**UDDI**	Universal Description, Discovery and Integration
MVC	model-view-controller		**UI**	User Interface
ORB	Object Request Broker		**WSIL**	Web services inspection language
OTS	Object Transaction Service			
OU	Organizational Unit		**WAR**	Web Application Archive
PDE	Plug-in Development Environment		**WSDL**	Web Services Definition Language
PME	Programming Model Extensions		**WAR**	Web Application Archive
			WSIF	Web Services Invocation Framework
PMI	Performance Monitoring Infrastructure		**XHTML**	Extensible HyperText Markup Language
POJO	plain old Java object			

XMI	XML Metadata Interchange
XML	eXtensible Markup Language
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 about ordering these publications, see "How to get IBM Redbooks" on page 629. Note that some of the documents referenced here may be available in softcopy only.

► *Rational Application Developer V6 Programming Guide*, SG24-6449

► *WebSphere Application Server V6 Planning and Design WebSphere Handbook Series*, SG24-6446

► *WebSphere Application Server V6 System Management & Configuration Handbook*, SG24-6451

► *WebSphere Application Server - Express V5.0.2 Administrator Handbook*, SG24-6976

► *WebSphere Application Server - Express V5.0.2 Developer Handbook*, SG24-6555

► *WebSphere Application Server - Express: A Development Example for New Developers*, SG24-6301

Other publications

These publications are also relevant as further information sources:

► *EJB Design Patterns*, Floyd Marinescu, John Wiley & Sons, Inc., 2002, ISBN: 0-471-20831-0

► *Design Patterns: Elements of Reusable Object-Oriented Software*, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1994, ISBN: 0-201-63361-2

► Enterprise Messaging Using JMS and WebSphere (Kareem Yusuf), Prentice Hall, ISBN: 0-13-146863-4

► Java Message Service (Monson-Haefel, Chappell), O'Reilly, ISBN: 0-596-00068-5

- ► Professional JMS (Grant, Kovacs, et al), Wrox Press Inc., ISBN: 1861004931
- ► Enterprise JavaBeans, Fourth Edition (Monson-Haefel, Burke, Labourey), O'Reilly, ISBN: 0-596-00530-X

Online resources

These Web sites and URLs are also relevant as further information sources:

- ► The Rational Unified Process(RUP)

  http://ibm.com/software/awdtools/rup/index.html

- ► Extreme programming resources

  http://www.xprogramming.com/

- ► Extreme programming introduction

  http://www.extremeprogramming.org/

- ► The Agile alliance

  http://www.agilealliance.org

- ► Eclipse.org

  http://www.eclipse.org/

- ► WebSphere Application Server - Express system requirements

  http://ibm.com/software/webservers/appserv/express/requirements/

- ► Java Comunnity Process - Java Specification Requests

  http://www.jcp.org/en/jsr/all

- ► Java 2 Platform, Enterprise Edition Specification, v1.4

  http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

- ► WebSphere Application Server supported hardware and software

  http://ibm.com/software/webservers/appserv/doc/latest/prereq.html

- ► Rational Web Developer system requirements

  http://ibm.com/software/awdtools/developer/web/sysreq/index.html

- ► Logging services project at Apache

  http://logging.apache.org/

- ► The Apache HTTP server project

  http://httpd.apache.org/

- ► Server watch

  http://www.serverwatch.com

- ► W3C HTML home page

  http://www.w3.org/MarkUp/

- ► W3C Hypertext Transfer Protocol overview

  http://www.w3.org/Protocols/

- ► Sun Java Servlets product page

  http://java.sun.com/products/servlet/

- ► Sun Java Servlets tutorial

  http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html

- ► Sun JavaServer Pages product page

  http://java.sun.com/products/jsp/product.html

- ► Sun JavaServer Pages tutorial

  http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPIntro.html

- ► Sun JavaBeans product page

  http://java.sun.com/products/javabeans/

- ► Sun JavaBeans tutorial

  http://java.sun.com/docs/books/tutorial/javabeans/index.html

- ► Sun Java 2 Platform, Enterprise Edition home page

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

- ► Sun Java 2 Platform, Enterprise Edition tutorial

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

- ► Java Specification Request Service Data Objects

  http://www.jcp.org/en/jsr/detail?id=235

- ► Spring globally unique identifier generator

  http://static.springframework.org/spring-webflow/docs/pr3/api/org/springframework/util/RandomGuid.html

- ► ActiveScript - J-GUID

  http://www.activescript.co.uk/jguid.html

- ► Developing and Deploying Modular J2EE Applications with WebSphere Studio Application Developer and WebSphere Application Server

  http://ibm.com/developerworks/websphere/library/techarticles/0206_robinson/robinson.html#N10242

- ► WebSphere Application Server library

  http://www.ibm.com/software/webservers/appserv/infocenter.html

- ► Java Message Service documentation

  http://java.sun.com/products/jms

- ► Java 2 Platform, Enterprise Edition documentation

  http://java.sun.com/j2ee/index.jsp

- ► J2EE Connector Architecture

  http://java.sun.com/j2ee/connector/index.jsp

- ► WebSphere MQ Using Java

  http://ibm.com/software/integration/mqfamily/library/manualsa/manuals/cross
  latest.html

- ► Apache Struts home page:

  http://struts.apache.org/

- ► Apache Struts User Guide:

  http://struts.apache.org/userGuide/introduction.html

- ► Best practices for Struts development

  http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

- ► Struts Validator guide

  http://struts.apache.org/userGuide/dev_validator.html

- ► Jakarta Regexp

  http://jakarta.apache.org/regexp/index.html

- ► W3C Soap specifications

  http://www.w3.org/TR/soap/

- ► W3C XML

  http://www.w3.org/XML/

- ► W3C WSDL:

  http://www.w3.org/TR/wsdl

- ► OASIS UDDI

  http://www.uddi.org/

- ► SOA and Web services standards

  http://ibm.com/developerworks/views/webservices/standards.jsp

- ► WS-I basic profile

  http://ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile

- ► OASIS

  http://www.oasis-open.org/

► Business Process Execution Language for Web Services version 1.1

http://ibm.com/developerworks/library/specification/ws-bpel/

► Web Services Inspection Lanaguage

http://ibm.com/developerworks/webservices/library/ws-wsilspec.html

► CVS manuals

http://ximbiot.com/cvs/wiki/index.php?title=CVS--Concurrent_Versions_System
_v1.12.12.1

► Sttus module switching

http://struts.apache.org/userGuide/configuration.html#module_config-switchi
ng

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

361, 392
   architecture   312
   create   317, 336
   deployment   324
   entity beans   315
   MDB   316
   query language   249
   session beans   316
EJB components   314
EJB container   65–66, 313–314, 317, 335, 392
EJB modules   226–227
EJB query language   249
EJB server   313
E-mail   19
E-mail component   27
EMF   296, 298
encapsulation   23
enhanced EAR   113
enhancements   209
enterprise application   226
Enterprise Application Archive *See* EAR
Enterprise Generation Language   38
Enterprise Information Systems   100
Enterprise JavaBeans *See* EJB
Enterprise Service Bus *See* ESB
Enterprise Web Services   82
entity beans   315, 335
entity relationship diagram   530
environment   102
ERD   530
ERP   100
errors   240
ESB   56, 85
event listeners   241
events
   JSF   243
exceptions   378
Express Application Server   xv, 227, 646
extensibility   21
Extensible Markup Language *See* XML
Extreme programming   11, 626

F
facade   24, 316
Faces JSP file
   create   257
faces-config.xml   282
FacesServlet   241–242

factory   372
failover   60, 105
features   17, 130
file creation wizards   234
fire and forget   364
first steps   136
form beans   450, 452
forms   20, 30–31, 219
frameworks   240, 248

G
Generic JMS provider   102
GIF   233–234
group one users   6
group three users   6
group two users   6
GUID   338

H
hardware requirements
   WebSphere Application Server - Express   121
help   145
   WebSphere Administrative Console   145
high availability   105
home page   257
   testing   257
HTML   6, 19, 30, 44–45, 48, 52, 120–121, 224, 227,
230–232, 256
HTTP   37, 65–66, 101, 210, 216, 219, 224, 226,
312, 316, 479
HTTP server   37, 224, 231, 479
HyperText Markup Language *See* HTML
Hypertext Transfer Protocol *See* HTTP

I
IBM Eclipse SDK 3.0   39
IBM HTTP Server   55, 120
IDE   112
identity column   528
IIOP   104
images   120, 233, 235
IMAP   99
implementation   15
import
   project interchange   202
IMS   81
inbound services   84

IBM

Redbooks

WebSphere Application Server - Express V6 Developers Guide and Development Examples

WebSphere Application Server - Express V6
Developers Guide and Development Examples

IBM®

Redbooks

Planning and designing your applications and databases

Developing and testing using Rational Web Developer

Building a sample application

This IBM Redbook is a practical guide for developing Web applications using the Rational Software Development Platform. We use the Rational Web Developer development environment that is provided as part of WebSphere Application Server - Express V6 to develop a sample Web application targeted to the WebSphere Application Server - Express runtime platform. We discuss a sample scenario based on realistic requirements for small and medium-sized customers, and provide a guide for the development of this scenario.

Our focus is on describing a simple process that allows nontechnical readers to understand and participate in the development of Web applications using Rational Web Developer. Our target runtime environment is the Express Application Server so we use the Rational Web Developer development environment that is part of the WebSphere Application Server - Express installation..