
SilverStream eXtend eXtend Workbench
Development Guide

Version 4.0

June 2002

SilverStream[®]

Copyright ©2002 SilverStream Software, Inc. All rights reserved.

SilverStream software products are copyrighted and all rights are reserved by SilverStream Software, Inc.

SilverStream and jBroker are registered trademarks and SilverStream eXtend is a trademark of SilverStream Software, Inc.

Title to the Software and its documentation, and patents, copyrights and all other property rights applicable thereto, shall at all times remain solely and exclusively with SilverStream and its licensors, and you shall not take any action inconsistent with such title. The Software is protected by copyright laws and international treaty provisions. You shall not remove any copyright notices or other proprietary notices from the Software or its documentation, and you must reproduce such notices on all copies or extracts of the Software or its documentation. You do not acquire any rights of ownership in the Software.

Third Party Software:

Jakarta-Regexp Copyright ©1999 The Apache Software Foundation. All rights reserved. Ant Copyright ©1999 The Apache Software Foundation. All rights reserved. Xalan Copyright ©1999 The Apache Software Foundation. All rights reserved. Xerces Copyright ©1999-2000 The Apache Software Foundation. All rights reserved. Jakarta-Regexp, Ant, Xalan and Xerces software is licensed by The Apache Software Foundation and redistribution and use of Jakarta-Regexp, Ant, Xalan and Xerces in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notices, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "The Jakarta Project", "Jakarta-Regexp", "Xerces", "Xalan", "Ant" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org <<mailto:apache@apache.org>>. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of The Apache Software Foundation. THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright ©1996-2000 Autonomy, Inc.

Copyright ©2000 Brett McLaughlin & Jason Hunter. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution. 3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org <<mailto:license@jdom.org>>. 4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org <<mailto:pm@jdom.org>>). THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun Microsystems, Inc. Sun, Sun Microsystems, the Sun Logo Sun, the Sun logo, Sun Microsystems, JavaBeans, Enterprise JavaBeans, JavaServer Pages, Java Naming and Directory Interface, JDK, JDBC, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, SunWorkShop, XView, Java WorkShop, the Java Coffee Cup logo, Visual Java, and NetBeans are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

IBM Jikes™ and Bean Scripting Framework (BSF) Copyright ©2001, International Business Machines Corporation and others. All Rights Reserved. This software contains code in executable form obtained pursuant to, and the use of which is subject to, the IBM Public License, a copy of which may be obtained at <http://oss.software.ibm.com/developerworks/opensource/license10.html>. Source code for Jikes™ is available at <<http://oss.software.ibm.com/developerworks/opensource/jikes/>>. Source code for BSF is available at <http://oss.software.ibm.com/developerworks/projects/bsf>.

This software contains code in executable form obtained pursuant to the Mozilla Public License, a copy of which may be obtained at <<http://www.mozilla.org/MPL/>>. Source code is available at <http://www.mozilla.org/rhino/download.html>.

This Software is derived in part from the SSLava™ Toolkit, which is Copyright ©1996-1998 by Phaos Technology Corporation. All Rights Reserved.

Contents

About This Book ix

- Purpose ix
- Audience ix
- Prerequisites ix
- Organization ix

Chapter 1 Developing Applications with Workbench 1

- Designing an application 1
 - Designing J2EE applications 1
 - Designing Web Services 2
- Developing an application 3
 - Setting up your project 3
 - Working on components 5
 - Updating deployment descriptors 7
 - Supporting team development 7
- Building the project 8
- Deploying project archives 9
 - Deploying from Workbench 9
 - Deploying outside Workbench 10
- Testing and debugging an application 10
 - Deploying your application to a test server 10
 - Running your application 11
 - Debugging your application 11

PART I WRITING J2EE COMPONENTS

Chapter 2 Understanding J2EE 15

- What is J2EE? 15
 - What J2EE gives you 16
 - Two kinds of applications 17
 - J2EE technologies 17
- How are J2EE applications put together? 21
 - Three tiers 21
 - J2EE applications are delivered in archive files 24
 - Roles in J2EE development 25
 - Model-View-Controller application model 26

Learning more about J2EE	26
The J2EE Blueprints	27
J2EE Web sites	27
J2EE and Workbench	28
Support for J2EE versions	28
Support for J2EE roles	28
J2EE-oriented IDE and projects	28
Wizards and editors for J2EE components	29
Build and archive facilities for J2EE modules	29
J2EE deployment services	29
Chapter 3 Writing JSP Pages	31
About JSP pages	32
SilverStream eXtend Workbench support for JSP pages	34
Looking at a sample JSP page	34
Developing JSP pages	38
Packaging the application	40
Deploying the application	42
Running the application	46
Chapter 4 Writing Servlets	49
About servlets	49
Servlet life cycle	49
Servlets and JSP pages	51
Servlets and J2EE archive structure	51
Developing a servlet	52
Creating a servlet class in Workbench	52
Processing the HTTP request	55
Generating the HTTP response	56
Specifying initialization and cleanup methods	61
Other servlet coding issues	62
Packaging the application	62
Deploying the application	63
Running a servlet	63
Chapter 5 Writing J2EE Application Clients	65
About J2EE application clients	65
Client features	65
Client container	67
Client life cycle	67
Developing a client	68
Coding client classes	68
Compiling client classes	74

Packaging a client	75
Writing the manifest file	75
Writing the deployment descriptor file	77
Creating the client JAR file	79
Deploying a client	80
Writing server-specific deployment information	80
Deploying the client JAR file	83
Running a client	86

Chapter 6 Writing Enterprise JavaBeans 87

About EJBs	87
Developing EJBs	92
What Workbench does	94
Packaging EJBs	95
Writing the deployment descriptor	95
What Workbench does	96
Creating an EJB JAR file	96
What Workbench does	96
Deploying EJBs	97
Calling EJBs	97
Finding the EJB	97
Tips for designing EJB applications	100

Chapter 7 Using Resource Adapters 101

About resource adapters	101
Deploying resource adapters	103
Using resource adapters	104

PART II PRODUCING AND CONSUMING WEB SERVICES

Chapter 8 Understanding Web Services 109

About Web Services	109
Web Service providers, consumers, and registries	110
Providing Web Services	111
Creating Web Service components	111
Creating a WSDL file	112
Publishing Web Service information	112
Using Web Services	113
Using Web Service registries	114
About registries	114
Registry data formats	115

Public and local registries	115
Learning more about Web Services	115
Popular Web Service implementations	116
Web Services and Workbench	116
jBroker Web	117
Web Service Wizard	117
Registry Manager	118
WSDL Wizard and Editor	118
Chapter 9 Generating Web Services	119
Basics	119
Steps	120
Preparing to generate	120
Generating Web Service files	122
Examining the generated files	125
Editing the generated files	131
Using the generated files	133
Choosing an implementation model	135
Tie model	135
Skeleton model	136
Scenario: starting with a Java class	137
Project setup	138
Input to the wizard	138
Generated files for the Web Service	142
Generated files for testing	154
Deployment descriptor	166
Runtime test result	166
Chapter 10 Generating Web Service Consumers	167
Basics	167
Steps	168
Preparing to generate	168
Providing a WSDL file	169
Example: WSDL file for Autoloan .NET Web Service	170
Understanding the WSDL	173
Generating the consumer files	174
Examining the generated files	177
About generated file names	178
Additional details of generation	178
Example: generated consumer files for Autoloan .NET Web Service	178

Editing the generated files	195
Editing the xxxClient.java file	195
Using the generated files	197
Running the consumer program	198
From Workbench	198
From a command line	199

About This Book

Purpose

This guide tells you how to develop J2EE and Web Service applications using SilverStream eXtend Workbench.

Audience

This guide is for J2EE application programmers who need to create, assemble, and deploy J2EE and Web Service components.

Prerequisites

This guide assumes that you are familiar with the Java programming language, the Internet, and Web applications. You can find learning materials on these topics readily available from a variety of public and commercial sources.

Organization

Here's a summary of the topics you'll find in this guide:

Topic	Description
Developing Applications with Workbench	Examines the process of developing applications in Workbench and discusses Workbench support for J2EE and Web Services
Writing J2EE Components	Provides an overview of J2EE technologies and explores how Workbench helps you develop J2EE components (JSP pages, servlets, application clients, Enterprise JavaBeans) and use supporting services (resource adapters)
Producing and Consuming Web Services	Provides an overview of Web Service technologies and explores how Workbench helps you create, publish, find, and consume Web Service components

1 Developing Applications with Workbench

This chapter explores the life cycle of a **J2EE or Web Service application**. It looks at each phase of the development process and explains how SilverStream eXtend Workbench can help you along the way. The process consists of:

1. Designing an application
2. Developing an application
3. Building the project
4. Deploying project archives
5. Testing and debugging an application

Designing an application

A comprehensive design phase is strongly recommended to help you make appropriate choices in architecture and technologies, ensuring success for your project. This includes:

- Designing J2EE applications
- Designing Web Services


You can design your application manually or with automated design and modeling tools, then implement that design using Workbench.


Designing J2EE applications

When you design an application for a J2EE (Java 2 Platform, Enterprise Edition) server, give careful consideration to the programming model it should follow. Good models, such as the **Model-View-Controller (MVC)** architecture, are available for handling the potential complexity of J2EE applications. The Jakarta **Struts** project is a popular MVC implementation.

Your application design should also specify which J2EE technologies you need. These may include:

- **Component technologies** such as application clients, servlets, JavaServer Pages (JSP pages), and Enterprise JavaBeans (EJBs)
- **Service technologies** such as Java Naming and Directory Interface (JNDI), Java Database Connectivity (JDBC), Connector architecture (resource adapters), Java Transaction API (JTA), Java Authentication and Authorization Service (JAAS), JavaMail, Java Messaging Service (JMS), Java API for XML Parsing (JAXP), and others

 For more on J2EE technologies, see Chapter 2, “Understanding J2EE”.

 For details on J2EE application design, consult the following table:


To learn about	See
Designing J2EE applications	<i>Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition</i> , from the Sun Blueprints at java.sun.com/blueprints
Using the Struts implementation of the MVC architecture	jakarta.apache.org/struts
Best practices in J2EE development recommended by SilverStream	devcenter.silverstream.com

Designing Web Services

The design of a Web Service involves several standard technologies, including:

- **Simple Object Access Protocol (SOAP)**, an XML-based messaging protocol that enables software components to communicate regardless of development platform and source language differences
- **Web Services Description Language (WSDL)**, an XML-based language that represents characteristics of a Web Service
- **Universal Description, Discovery, and Integration (UDDI)** registries, which enable you to publish to and make inquiries of a central, network-accessible repository of information about businesses and Web Services

Web Service providers must ensure that their services are highly available, reliable, and scalable, typically through careful hardware and software design choices. Web Services created in Workbench are implemented using J2EE technologies, so J2EE best practices apply to their design as well.

 For more on Web Service technologies, see Chapter 8, “Understanding Web Services”.



For details on Web Service design, consult the following table:

To learn about	See
Designing applications that implement Web Services	Chapter 9, “Generating Web Services”
Designing applications that access Web Services	Chapter 10, “Generating Web Service Consumers”

Developing an application

Using Workbench to develop a J2EE application or Web Service involves:

1. Setting up your project
2. Working on components
3. Updating deployment descriptors
4. Supporting team development

Setting up your project

In Workbench, a **project** typically represents a J2EE module that you want to build. You can create Workbench projects that build the following **J2EE archives**:

- Enterprise archive (EAR)
- Web archive (WAR)
- Resource adapter archive (RAR)
- Enterprise JavaBean archive (EJB JAR)
- Application client archive (client JAR)
- Java class archive (JAR)

These Workbench projects support the J2EE component model of development. This enables you to create, change, and build small parts of your enterprise application or the entire application.

Basic steps The typical process of setting up a J2EE development project in Workbench involves:

1. Organizing your source directories and files on the file system
2. Creating projects and subprojects in Workbench for the J2EE archives you need
3. Adding existing source directories and files to your projects in Workbench

For example, you might create a single, top-level project that represents an enterprise application. You might then create subprojects for the various modules that make up the application, including Web modules for the user interface, EJB modules for the business logic and database access, and so on.



For details on projects and subprojects in Workbench, see *Projects and Archives* in the *Tools Guide*.

Setting up a Web Service project In Workbench, Web Services are deployed as Web archives (WARs). To set up a Web Service project, you follow the same steps as when creating a WAR project.

Organizing your source directories and files


Your initial setup steps depend on whether you're creating a project from scratch or importing existing J2EE source into Workbench:


If you're starting with	Do this
No source directories or files	Create a directory tree on the file system for your project. Often this consists of a single top-level directory for the application and subdirectories to group components (JSP pages, servlets, EJBs, Java class files, and so on.). Another possible approach is to store component directories independently (in case they're used in multiple applications).
Existing source directories and files	Make sure the directory structure on your file system maps well to the J2EE modules for your project. This helps you import source into Workbench (because you can simply import entire directories). If your file system is not organized properly, you may have to import files individually (which makes project maintenance harder).

Creating projects and subprojects

Workbench provides a **New Project Wizard** that helps you create a project for each type of archive you want to build. For example, if you're creating an enterprise archive, you can select EAR as the project type then specify the project name, file system location, and J2EE version. Workbench creates a SilverStream **project file** (with **SPF** extension) in the project location.

Once you create projects for your application's J2EE modules (WARs, RARs, EJB JARs, client JARs), you can add them to the EAR project as subprojects.


 For more information on organizing projects and subprojects, see Organizing projects in the *Tools Guide*.

 To learn about choosing the J2EE version for a project, see the chapter on how to handle J2EE versions in *Getting Started*.

Adding existing source directories and files to your projects

Once your application architecture is represented in Workbench projects and subprojects, you can add any existing source directories and files to them. For example, you may already have the Java classes for some application components. You might have some standard resources (such as graphics) that you're reusing from other applications.

Wherever possible, add the **directory** that contains the files rather than the individual files. If you add a directory to a project, any files in that directory are automatically included in the project. If you specify individual files, you must add any new files created in that directory to the project manually.

 For more information on adding directories and files to a project, see Populating projects in the *Tools Guide*.

Working on components

Workbench provides **component wizards** and **source editors** to help you create and maintain J2EE components for your projects. Because Workbench adheres to J2EE standards, you also have the option of using any third-party tool to develop components for a Workbench project.

Using component wizards

Whenever you request a new file in Workbench, a wizard helps you create the kind of J2EE component or other item you want. Workbench provides wizards for JSP pages and tag libraries, servlets, EJBs, JavaBeans and Java classes, XML files, WSDL files, text files, Web Services, and more. The Web Service Wizard lets you create Web Services (SOAP-enabled servlets and supporting classes for a WAR project) or Web Service consumers (classes for accessing Web Services).

Each wizard collects information about the requested item, creates files and directories for it (including Java source where possible), and adds it to the appropriate project.



For more information, see *Creating source files and Component Wizards* in the *Tools Guide*.

Using source editors

Workbench provides a variety of editors you can use to further develop the source files in your projects:

- Java Editor
- JSP Editor
- HTML Editor
- Text Editor
- XML Editor
- WSDL Editor
- Deployment Descriptor Editor
- Deployment Plan Editor

When you open a file, Workbench automatically invokes the appropriate editor for that file type. Editor features include archive-awareness, various coding conveniences, and version control access.



For more information on using these editors, see *Source Editors* in the *Tools Guide*.



To learn about version control access, see the chapter on Workbench basics in the *Tools Guide*.

Using other tools

Workbench supports any J2EE module or component, regardless of how it was created. This means you can develop modules and components using your favorite third-party tools (such as another IDE or editor) then import them into Workbench (as described in “Adding existing source directories and files to your projects” on page 5).

Updating deployment descriptors

Workbench generates an appropriate deployment descriptor for any J2EE project or subproject you create. When you modify the contents of a project, Workbench automatically updates the corresponding deployment descriptor.

Workbench provides a **Deployment Descriptor Editor** that enables you to manually edit a deployment descriptor file. This editor offers both graphical and text-based views of the deployment descriptor information.



See Deployment Descriptor Editor in the *Tools Guide*.

Supporting team development

Because Workbench maintains projects on your file system, it's easy to share work among multiple developers. This section provides some tips on making the process flow smoothly:

- Keeping project files current
- Using relative paths

Keeping project files current

When you make changes to a project (such as adding files, directories, components, or modules), Workbench updates the project's SPF and deployment descriptor files as needed. When multiple developers work on the same set of project files, there are several ramifications of such changes. Following good source control processes usually ensures that changes in the project structure and content are handled appropriately.

You must have write access to the appropriate project files when making project-level changes. Typically, this means checking out SPF, deployment descriptor, and component files from a version control system. To share project-level changes with others on your team, you must check in your project files. Other members of the team must update their work areas to reflect the changed project structure and content.

Using relative paths

When creating components or modules in Workbench, you specify paths for archives and directories. When multiple developers work on a project, you may want to specify these paths relative to the project directory.

The advantage of using relative paths is that project files don't rely on drive letters or other absolute path structures (which can be problematic across file systems). For example, a Z: drive mapped on your computer might not exist on another developer's computer. Unless you can guarantee that all developers accessing your project have some known set of drives, you should use relative paths.

The disadvantage is that in deep directory structures, relative paths are sometimes difficult to decipher (for example, a file might be specified as `..\..\..\beans\classes\checker.class`).

Building the project

Workbench gives you flexibility in building project files and creating J2EE archives. You can:

- **Compile just the currently open Java file** without affecting the rest of your project
- **Build an entire project** (and its subprojects) with the option of compiling all classes or only those that need it
- **Generate the archive** for a project (and its subprojects)

You can perform build operations from the Workbench IDE or from the command line. In either case, your project settings are used to specify build details (such as where to generate class files and archives).



For more information, see *Compiling, building, and archiving* in the *Tools Guide*.

Validating project archives Workbench also enables you to validate the generated archive for a project (and its subprojects). Validation is a good check to perform before deployment. It makes sure the archive's deployment descriptor agrees with the appropriate J2EE deployment descriptor DTD and with the archive's content.



For more information, see *Validating archives* in the *Tools Guide*.

Deploying project archives

Once you generate the archive for a Workbench project, you can deploy it to a J2EE server. You have a choice of deployment approaches:

- Deploying from Workbench
- Deploying outside Workbench

Deploying from Workbench

Workbench provides built-in support for deployment to a variety of J2EE servers:


- BEA WebLogic Server
- IBM WebSphere Application Server
- Jakarta Tomcat
- Oracle9i Application Server
- SilverStream eXtend Application Server
- Sun J2EE Reference Implementation Server

Basic steps To deploy a project archive from Workbench to one of these servers, you:

1. Define a **server profile** that specifies configuration details about your target J2EE server.
2. Prepare **server-specific deployment information** that describes how the archive should run on your target J2EE server.

This information is typically expressed in XML, similar to the standard J2EE deployment descriptors. For example, when deploying to a SilverStream server, you provide an XML file called a **deployment plan** (which you can edit in the **Deployment Plan Editor** included in Workbench).

3. Specify **deployment settings** that tell Workbench how and where to deploy.
These settings include a **rapid deployment** option that's helpful during the development phase to quickly deploy and test changes you make.
4. Use the **Project>Deploy Archive** command to start the deployment.

 To learn more about deploying from Workbench, see Archive Deployment in the *Tools Guide*.

Deploying outside Workbench

Alternatively, you can take archives generated in Workbench and deploy them via other J2EE-compatible tools (such as the deployment facilities provided by your J2EE server). This approach should enable you to deploy to any standard J2EE server.

Testing and debugging an application

Before you can release a J2EE or Web Service application for production use, you must make sure it operates properly and with acceptable performance. Your quality control process should include:

- Deploying your application to a test server
- Running your application
- Debugging your application

Deploying your application to a test server

By deploying to a test server, you can discover application problems without exposing end users or other groups to them. Here are some common test server scenarios:

In this scenario	You might
You are unit testing your own development work	Deploy to a J2EE server on your local machine
You are integrating your development work with the work of your team	Set up an integration test machine for the team and deploy to a J2EE server on it
Your team is preparing to move its development work into production	Set up a preproduction staging machine for quality assurance and deploy to a J2EE server on it

Wherever possible, test environments should approximate the production environment in which your application will run. You can facilitate deployment to a set of test servers by defining server profiles for them in Workbench.





See the Server profile discussion in the *Tools Guide*.

Running your application

In many cases, you can test how a deployed J2EE application runs by using a Web browser to request a particular URL from your J2EE server. This approach applies when you're testing **JSP pages** and **servlets**, as well as other components or services that they then access (such as Web Services, EJBs, resource adapters, tag libraries, filters, JavaBeans, and supporting classes).

Testing a deployed J2EE **application client** requires a different approach. This essentially involves invoking the client container and asking it to start the client (although the exact process depends on your J2EE server's implementation of the client container).

 For more information on running a specific type of J2EE component, see the appropriate chapter in Part I, "Writing J2EE Components".

 For details on testing Web Services or Web Service consumers, see the appropriate chapter in Part II, "Producing and Consuming Web Services".

Debugging your application

Once you're running an application, you can use debugging tools to control program execution and monitor program status. This enables you to find and fix runtime errors. Workbench provides a **Debugger** that you can launch to debug J2EE and other Java applications (including client-side or server-side objects, on a local or remote machine).

 For more information, see the Debugger chapter in the *Tools Guide*.

Part I Writing J2EE Components

A primer on J2EE components and supporting services that prepares you for creating and using them in Workbench

- Chapter 2, "Understanding J2EE"
- Chapter 3, "Writing JSP Pages"
- Chapter 4, "Writing Servlets"
- Chapter 5, "Writing J2EE Application Clients"
- Chapter 6, "Writing Enterprise JavaBeans"
- Chapter 7, "Using Resource Adapters"

2 Understanding J2EE

The move of enterprise computing to the Internet and World Wide Web poses challenges to application providers. More than ever, enterprise applications must be responsive, easily updatable, distributed, scalable, cross-platform, and integrated with a variety of existing back-end information systems. Sun's **Java 2, Enterprise Edition** (J2EE) addresses these challenges.

This chapter provides a concise overview of J2EE and introduces the J2EE features of SilverStream eXtend Workbench. Topics include:

- What is J2EE?
- How are J2EE applications put together?
- Learning more about J2EE
- J2EE and Workbench

What is J2EE?

J2EE is a standard that provides a component-based approach to designing, implementing, and deploying multitier enterprise-level applications. With J2EE, you get reusability of components, portability, transaction support, a unified security model, and more.

This section explores the basics of J2EE, including:

- What J2EE gives you
- Two kinds of applications
- J2EE technologies

What J2EE gives you

The J2EE platform provides the following benefits:

- **J2EE applications have a standardized, component-based architecture**

J2EE applications consist of components (including servlets, JavaServer Pages, and Enterprise JavaBeans) that are bundled into modules. Because J2EE applications are component-based, you can easily reuse components in multiple applications, saving time and effort and enabling you to quickly deliver applications.

This modular development model also supports clear division of labor across development, assembly, and deployment of applications so you can best leverage the skills of individuals at your site.

- **J2EE applications are distributed and multitier**

J2EE provides server-side and client-side support for enterprise applications. J2EE applications present the user interface on the client (typically a Web browser), perform their business logic and other services on the application server in the middle tier, and are connected to enterprise information systems on the back end (these three tiers are described in a little more detail later). With this architecture, functionality exists on the most appropriate platform.

- **J2EE applications are standards-based and portable**

J2EE defines standard APIs, which all J2EE-compatible vendors must support. This ensures that your J2EE development is not tied to a particular vendor's tools or server.

This means that you have your choice of tools, components, and servers. Because J2EE components use standard APIs, you can develop them in any J2EE development tool (including Workbench), develop components or purchase them from a component provider, and deploy them on any J2EE-compatible server. You pick the tools, components, and server that make the most sense for you.

- **J2EE applications are scalable**

J2EE applications run in containers, which are part of a J2EE server. These containers can themselves be designed to be scalable, so scalability can be handled by the J2EE server provider without any effort from the application developer.

- **J2EE applications can be easily integrated with back-end information systems**

The J2EE platform provides standard APIs for accessing a variety of enterprise information systems (EISs), including relational database management systems, e-mail systems, and CORBA systems. For broader connectivity, J2EE includes the Connector architecture, which defines a standard means for accessing heterogeneous EISs.

Two kinds of applications

There are two kinds of J2EE applications:

- **Web applications** use Web browsers as clients and download static HTML, dynamic HTML, or XML generated by JavaServer Pages or servlets on the server.
- **Non-Web applications** use a standalone client (usually written in Java) or an applet embedded in a nonbrowser appliance, such as a cell phone.

The J2EE Blueprints document from Sun recommends using Web applications as much as possible. Web browsers are standard and you don't have to deploy client software onto user desktops. When used with supporting technologies (such as JavaScript, DHTML, and XML/XLS), Web applications can be made highly interactive. And browser technology continues to advance, making browsers ever more attractive as the client environment.

J2EE technologies

J2EE technologies can be divided into these categories:

- J2EE components
- J2EE services

J2EE components

J2EE includes the following kinds of components:

- Web components
- Enterprise JavaBean components
- Client components

It also supports JavaBean components, which are part of J2SE (Java 2, Standard Edition).


Web components

Web applications consist of Web components and other resources bundled together. There are two major kinds of Web components:

Web component	Description
Servlets	<p>Servlets extend the functionality of a Web server, much like Common Gateway Interface (CGI) programs. Servlets are a better choice because, unlike CGI programs, they are portable (written in Java), scale well, and are easy to maintain.</p> <p>Servlets describe how to process an HTTP request and generate a response. You can use them to deliver dynamic content.</p>
JavaServer Pages (JSP pages)	<p>Like servlets, JSP pages describe how to process and respond to HTTP requests. Unlike servlets, JSP pages are text-based documents that include a combination of HTML and JSP tags, Java code, and other information.</p> <p>JSP pages and servlets both solve the same problem, but JSP pages have the advantage of separating presentation (expressed in HTML) from application logic (coded in Java). With servlets, the presentation and application logic are mixed together in the same Java file. By using JSP pages, you can have your UI developers working on presentation of information, while your Java programmers are separately developing the application's logic.</p> <p>You should use JSP pages in most of your Web applications.</p>

Web applications can also contain some other supporting components:

- **Filters** can be used to modify the data or headers of an incoming request, or of an outgoing response.
- **Event listeners** can be used to monitor the servlet context or HTTP session for state changes and then perform any appropriate processing.

 For more on Web components, see Chapter 3, “Writing JSP Pages” and Chapter 4, “Writing Servlets”.

Enterprise JavaBean components

The business logic of a J2EE application resides in Enterprise JavaBeans (EJBs). EJBs are the layer between your application's presentation (viewed in a Web browser) and the data in your back-end enterprise information systems. There are three kinds of EJB components:

EJB component	Description
Session beans	Session beans implement logic that is specific to one client session. In a shopping cart application, for example, you would maintain a client's state (such as the items in a client's shopping cart) in a session bean. Session beans are not shared across clients.
Entity beans	Entity beans represent persistent business data, such as a row in a relational database. Entity beans are object models—they encapsulate the data along with the methods that act upon the data. Entity beans can be shared across clients and persist as long as the data they represent persists.
Message-driven beans	Message-driven beans are stateless EJBs invoked asynchronously by the arrival of a JMS (Java Messaging Service) message. After receiving a message, a message-driven bean performs business logic to process it and then waits for the next message. A client accesses a message-driven bean by sending messages to an appropriate JMS queue or topic.

 For more on EJB components, see Chapter 6, “Writing Enterprise JavaBeans”.

Client components



While most J2EE applications use a standard Web browser as the primary or sole client, J2EE also supports a couple of clients that execute a Java Virtual Machine:

- Applets
- Standalone Java application clients

 For more on client components, see Chapter 5, “Writing J2EE Application Clients”.

J2EE services

J2EE provides a wide range of standard services, including the following:

Service	Description
Deployment	<p>J2EE applications are deployed as a set of modules. Each module contains a deployment descriptor that specifies how to assemble and deploy the module in a runtime environment. Customized information can be provided at both assembly time and deployment time without the need to recompile the application objects.</p> <p> To learn who performs deployment tasks, see “Roles in J2EE development” on page 25.</p>
Naming	<p>Because J2EE applications are distributed, they need a way to look up and access remote objects and resources, such as EJBs and data sources. This is supported via the Java Naming and Directory Interface (JNDI).</p>
Data access	<p>J2EE supports both declarative and programmatic data access. It provides the Java Database Connectivity API (JDBC) for connectivity with relational database systems. It provides the Connector architecture (resource adapters) to give applications uniform access to various kinds of enterprise information systems.</p> <p> For more on the Connector architecture, see Chapter 7, “Using Resource Adapters”.</p>
Transaction	<p>J2EE supports both declarative and programmatic transactions. It provides the Java Transaction API (JTA) to handle transaction processing.</p>
Security	<p>J2EE supports both declarative and programmatic security. It provides the Java Authentication and Authorization Service (JAAS) to authenticate and enforce access controls upon users.</p>
Messaging	<p>J2EE provides JavaMail and the Java Messaging Service (JMS) to asynchronously send and receive messages. JavaMail is for e-mail messages. JMS is for program-to-program messages.</p>

Service	Description
Communication	<p>J2EE supports the following protocols:</p> <ul style="list-style-type: none"> • Internet protocols—These include TCP/IP, HTTP 1.0, and SSL 3.0 (for secure communication) • RMI protocols—Remote Method Invocation is a set of APIs used by Java distributed applications, including EJBs • OMG protocols—Object Management Group protocols allow J2EE applications to communicate with remote CORBA objects
File support	<p>J2EE implementations must support the following file types: HTML 3.2 files, GIF and JPEG files, JAR files, Java CLASS files, and XML files. XML manipulation is supported via the Java API for XML Parsing (JAXP).</p>

How are J2EE applications put together?

This section takes a closer look at the implementation of J2EE applications:

- Three tiers
- J2EE applications are delivered in archive files
- Roles in J2EE development
- Model-View-Controller application model

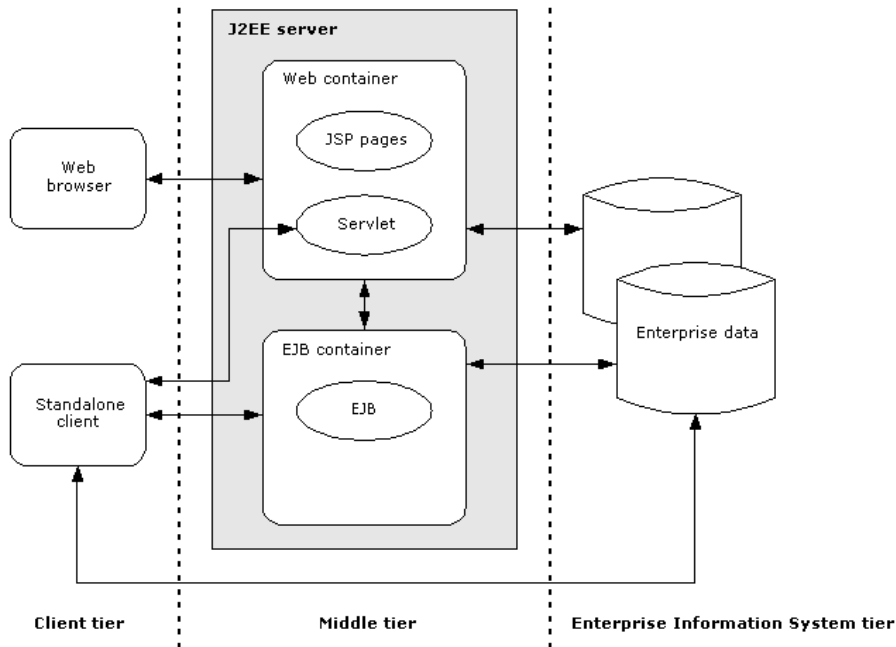
Three tiers

J2EE applications run on three tiers:

Tier	Description
Client tier	<p>Web browsers or standalone application clients. The J2EE Blueprints document recommends using Web browsers as clients whenever possible.</p>

Tier	Description
Middle tier	Consists of two subtiers: <ul style="list-style-type: none"><li data-bbox="576 331 1265 423">• Web tier. The J2EE Blueprints document recommends using JSP pages (with supporting servlets) to provide the core of the user interface for your application.<li data-bbox="576 440 1243 499">• EJB tier (or business tier). This is where the business logic, including data access, resides.
Enterprise Information System tier	Back-end databases and other information sources.

Here's a simplified illustration of these tiers:



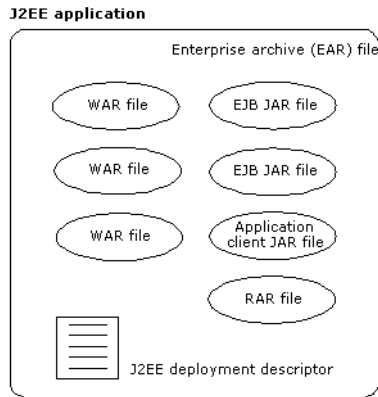
Containers At the heart of the J2EE component model are containers. Containers are the runtime environments implemented by J2EE platform providers. Containers provide life-cycle management and other services so that application developers can concentrate on the presentation and business logic of their applications.

For example, **Web containers** (which primarily contain JSP pages and servlets) provide support for receiving and responding to client requests. **EJB containers** provide built-in support for transaction management (among other things). Containers also provide built-in support for accessing enterprise information systems, such as supporting JDBC to access relational databases.

The Web and EJB containers run within the J2EE-compatible application server.

J2EE applications are delivered in archive files

A J2EE application consists of one or more J2EE modules and one deployment descriptor, packaged in an **enterprise archive** (EAR) file, which is a JAR file with the .EAR extension:



Deployment descriptors A deployment descriptor is an XML document that describes how to assemble and deploy a J2EE application or module in the runtime environment.

J2EE modules J2EE modules consist of one or more J2EE components of the same type and one component deployment descriptor. There are four kinds of J2EE modules:

Module	Description
Web modules	Consist of JSP files, classes for servlets, HTML or XML files, a deployment descriptor, and graphics files. Stored in a Web archive (WAR) file.
EJB modules	Consist of EJB classes and interfaces, plus a deployment descriptor. Stored in an EJB archive (JAR) file.
Application client modules	Consist of class files and a deployment descriptor. Stored in a client archive (JAR) file.
Resource adapter modules	Consist of class files and a deployment descriptor. Stored in a resource adapter archive (RAR) file.

Roles in J2EE development

One strength of the J2EE platform is that the implementation process is divided naturally into roles, which can be performed by different individuals with different skills.

Because of this role-based development, you can use your staff efficiently. You can have your developers do what they do best: code high-performing applications, without worrying about the details of the UI. And you can have your designers do what they do best: design attractive, easy-to-use interfaces, without having to be involved in the application's coding.

Here are the J2EE roles:

Role	Function
J2EE Product Provider	Provides the J2EE platform, including the J2EE-compatible server that supports your applications.
Application Component Provider	Creates Web components (JSP pages and servlets) and EJBs for use in J2EE applications. You can develop your own components or purchase components from others.
Application Assembler	Takes application components from component providers and assembles them into an enterprise archive (EAR) file. During this process, the assembler verifies that the components are defined properly to work together. The assembler also creates or modifies the application's deployment descriptor.
Deployer	Deploys the application in the runtime environment (the J2EE server). Defines final security, transaction, and other mappings as needed.
System Administrator	Configures and administers the runtime environment.
Tool Provider	Provides J2EE development, assembly, and deployment tools. Workbench is an example of a J2EE tool set.

Model-View-Controller application model

J2EE applications are best developed using the Model-View-Controller (MVC) application model, which consists of the following three elements:

Element	Description
Model	Represents the application data and the business rules that manage the data. In J2EE applications, the model is typically represented by EJBs .
View	Renders the content of the model to the user of the application. In J2EE applications, the view is typically provided by JSP pages .
Controller	Defines how the application works. It maps user actions (such as button clicks) to operations performed by the model (such as updating information in a database). The controller mediates between the view and the model. In J2EE applications, the controller is typically a servlet , JavaBean , or session bean .

Using the MVC architecture, you can separate the data, display, and flow of an application, allowing for greater flexibility and ease of reuse. MVC is also a very good way to develop applications that support multiple presentations of the same data.

Sample MVC applications The sample application provided with the J2EE Blueprints uses the MVC model. The Workbench Web application tutorial also uses MVC, implemented via the Struts framework from the Jakarta project.

Learning more about J2EE

This section lists other J2EE learning resources:

- The J2EE Blueprints
- J2EE Web sites

The J2EE Blueprints

The J2EE Blueprints from Sun include the following learning materials to help you gain J2EE expertise:

- The book *Designing Enterprise Applications for the Java 2 Platform, Enterprise Edition*
This is one of the best resources for learning about how to build J2EE applications and use J2EE technologies. It also illustrates best practices via the accompanying sample application.
- The sample application **Java Pet Store**
This is an e-commerce J2EE application presented through a standard Web browser. It's an excellent demonstration of how to build J2EE applications using the MVC architecture, a shopping cart metaphor, and many J2EE features (including JSP pages and EJBs).

These materials are available from the Sun Blueprints Web site (listed below). The book is also purchasable in hardcopy from major bookstores.

J2EE Web sites

Here are some J2EE Web sites that you may find helpful:

Site	URL
J2EE home page	java.sun.com/j2ee
J2EE downloads	java.sun.com/j2ee/download.html
J2EE documentation	java.sun.com/j2ee/docs.html
J2EE Blueprints	java.sun.com/blueprints

J2EE and Workbench

SilverStream eXtend Workbench provides all the capabilities you need to create, organize, maintain, and deploy J2EE applications:

- Support for J2EE versions
- Support for J2EE roles
- J2EE-oriented IDE and projects
- Wizards and editors for J2EE components
- Build and archive facilities for J2EE modules
- J2EE deployment services

Support for J2EE versions

Workbench provides built-in support for multiple versions of J2EE, including 1.2 and 1.3. It helps you handle version-related tasks throughout the life cycle of a project, including development, migration, and deployment.



See the chapter on how to handle J2EE versions in *Getting Started*.

Support for J2EE roles

Workbench maintains a separation of development, assembly, and deployment operations to support the roles and responsibilities described in the J2EE specification.

J2EE-oriented IDE and projects

Workbench provides a graphical IDE that helps you create, organize, and maintain J2EE applications at the project, archive, and source (file system) levels. You can easily see how the source directories and files for a J2EE project are mapped into the resulting archive.

Workbench gives you a natural, consistent approach to developing J2EE components and assembling them into J2EE modules and applications.



See the chapter on projects and archives in the *Tools Guide*.

Wizards and editors for J2EE components

Workbench provides automated wizards that help you create well-structured J2EE components, including:

- JSP pages and tag libraries
- Servlets
- EJBs
- JavaBeans and Java classes

Workbench also provides source editors and debugging tools that simplify maintaining these components.



See the chapter on Workbench basics in the *Tools Guide*.

Build and archive facilities for J2EE modules

Workbench provides automated compiling, building, and archiving functions that enable you to produce J2EE modules such as:

- Enterprise archives (EARs)
- Web archives (WARs)
- EJB archives (EJB JARs)
- Application client archives (client JARs or CARs)
- Resource adapter archives (RARs)
- Java class archives (JARs)



See the chapter on projects and archives in the *Tools Guide*.

J2EE deployment services

Workbench provides automated wizards that create and update deployment descriptors for your J2EE modules and applications. There are also editors for any manual changes you need to make.

Workbench provides built-in support for deployment to a variety of J2EE servers. Alternatively, you can take archives generated in Workbench and deploy them via other J2EE-compatible tools (such as the deployment facilities provided by your J2EE server).



See the chapter on archive deployment in the *Tools Guide*.

3

Writing JSP Pages

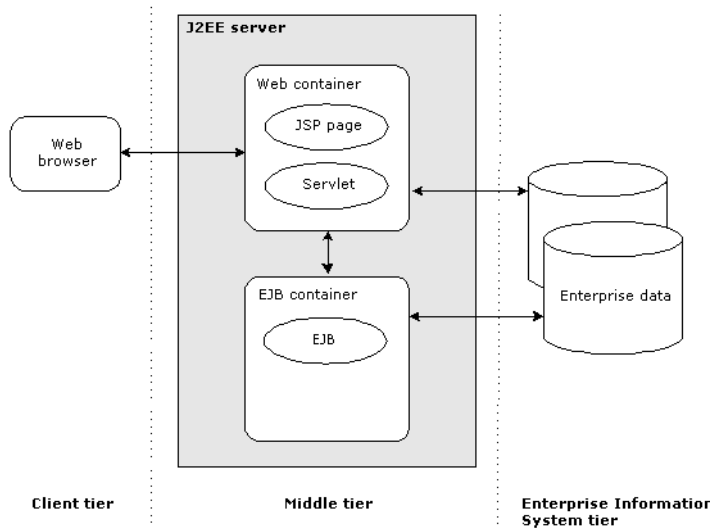
JavaServer Pages (JSP) technology provides a standard way to generate dynamic content and incorporate that content in Web-based applications. This chapter introduces you to JSP. It covers the following topics:

- About JSP pages
- Developing JSP pages
- Packaging the application
- Deploying the application
- Running the application

About JSP pages

JSP pages are an important part of Sun's J2EE platform, which recommends using JSP pages to provide the core of the user interface of your application. JSP pages are typically used in Web-based J2EE applications (*Web applications*). A Web application includes JSP pages, servlets, JavaBeans, utility classes, images, and so on that are packaged in an archive called a Web archive (*WAR*) file. These applications are accessed by browser clients.

The following diagram shows how JSP pages and servlets are part of J2EE's middle tier, sometimes called the *Web tier*.



JSP pages simplify the process of creating dynamic Web content, because they combine the power of Java with the ease of use of a Web markup language. JSP pages:

- Describe how to process and respond to HTTP requests
- Are text-based documents that include a combination of HTML and JSP tags, Java code, and other information
- Separate presentation (expressed in HTML) from application logic, coded in Java
- Allow you to extend the capabilities of a JSP page by including calls to JavaBeans components as well as embedded Java code fragments

- Can also contain **custom tags** defined in **tag libraries**
If none of the standard JSP tags provides the functionality you need for your application, you can write your own application-specific tag library and use custom tags defined by this library in your pages. Alternatively, you can use a tag library provided by a third party, such as the Jakarta project.
- Can act as a front end to Enterprise JavaBeans

About JSP pages and servlets JSP pages use the underlying servlet technology of the application server. When a JSP page is deployed to an application server, it is translated into a servlet, which is then compiled for execution. So how do servlets and JSP pages differ?

Servlets extend the functionality of a Web server, much like Common Gateway Interface (CGI) programs. Servlets are a better choice than CGI programs—because, unlike CGI programs, they are portable (because they are written in Java), scale well, and are easy to maintain. Servlets describe how to process an HTTP request and generate a response. You can use them to deliver dynamic content.

Like servlets, JSP pages describe how to process and respond to HTTP requests. Unlike servlets, which are written in Java, JSP pages are text-based documents that include a combination of HTML and JSP tags, Java code, and other information.


JSP pages and servlets both solve the same problem, but JSP pages have the advantage of separating presentation (expressed in HTML) from application logic, coded in Java. With servlets, the presentation and application logic are mixed together in the same Java file. So by using JSP pages, you can have your UI developers working on presentation of information, while your Java programmers are separately developing the application's logic.

SilverStream eXtend Workbench support for JSP pages

SilverStream eXtend Workbench provides tools that help you develop and deploy JSP pages. It specifically provides:

Workbench tool	Description
JSP Wizard	Lets you quickly specify a variety of attributes for a new JSP page and adds your JSP page to an open project For information on the JSP Wizard, see the chapter on component wizards in the <i>Tools Guide</i>
Tag Handler Wizard	Lets you quickly create a tag handler classes and TLDs for custom JSP tags For information on the Tag Handler Wizard, see the chapter on component wizards in the <i>Tools Guide</i>
Deployment Descriptor Editor	Lets you create and populate J2EE-compatible deployment descriptors
Deployment Plan Editor	Lets you create and populate a deployment plan for deploying J2EE-compatible components to a SilverStream eXtend Application Server
Deployment tool	Allows you to deploy J2EE-compatible archive files (such as WARs) to a variety of J2EE servers. You can deploy the archives to servers that support J2EE 1.2 and 1.3.

Workbench supports developing both 2.2 and 2.3 WARs. For making decisions about what WAR version you write to, see the chapter on J2EE versions.

 For more information and to access the specifications, see the Sun Java Web site at <http://java.sun.com/j2ee/docs.html>.

Looking at a sample JSP page

Here is a sample JSP page:

```
<html>
<jsp:useBean id="clock" scope="page" class="util.JspCalendar"/>
<jsp:useBean id="sql" scope="request" class="util.JspSQL"/>

<%@ taglib uri="SampleTags" prefix="SampleTags" %>
```

```
<h4>Use a tag library</h4>
<SampleTags:SimpleTag/>

<h4>Use the implicit Request object</h4>
<ul>
<li>Server name: <%= request.getServerName() %>
<li>Server port: <%= request.getServerPort() %>
<li>HTTP method: <%= request.getMethod() %>
</ul>

<h4>Use a Bean to access date information</h4>
<ul>
<li>Day of month: is <jsp:getProperty name="clock" property="dayOfMonth"/>
<li>Another form of Day of month: is <%=clock.getDayOfMonth() %>
<li>Year: is <jsp:getProperty name="clock" property="year"/>
<li>Month: is <jsp:getProperty name="clock" property="month"/>
</ul>

<h4>Call a function declared on the JSP page</h4>
<!-- Function declaration -->
<%!
    public String getAString(String x)
    {
        return x + " was passed in";
    }
%>

<ul>
<li>Call getAString: <%= getAString("Hello") %>
</ul>

<h4>Use a Bean to access a database</h4>
<%= sql.getSQL(request, "Select ID, LASTNAME, FIRSTNAME from EMPLOYEES") %>

<h4>Execute a scriptlet that has embedded text</h4>
<% if (java.util.Calendar.getInstance().get(java.util.Calendar.AM_PM) ==
java.util.Calendar.AM) {%>
Good morning!
<% } else { %>
Good afternoon
!
<% } %>

<h4>Include the output of another JSP</h4>
<jsp:include page="include.jsp"/>

</html>
```

Here is what the page looks like:



Features The sample page demonstrates most of the features of JSP including:

- Two **JavaBeans** that perform processing. The page uses an **action** (`<jsp:useBean>`) to associate each bean with an ID. Once this association has been made, the page uses the `<jsp:getProperty>` action or an **expression** (`<%= ... %>`) to get data back from the beans. The JavaBeans are in separate Java source files, which are compiled and made available to the JSP pages.

- A **tag library** that contains custom JSP tags. Tag libraries are defined in tag library descriptor (TLD) files and implemented with Java classes. The page's taglib **directive** (`<%@ taglib ... >`) specifies the uri and prefix to use to reference the tags. The uri maps to a tag library that is specified in the Web application's deployment descriptor (see "Writing the deployment descriptor" on page 40). The prefix is prepended to all tags in the library that are used on the page.
The tag used on this page (SimpleTag) returns welcome text.
- **Implicit objects** that are accessed through implicit variables.
This page uses the implicit **request** variable to call several methods associated with the servlet request that triggered the page.
- A **declaration** (`<%! ... %>`) that defines a function on the page. The declaration uses an expression to call the function.
- A **scriptlet** (`<% ... %>`) that executes some conditional logic on the page. Depending on the result of the test, the scriptlet writes the embedded text **Good Morning!** or **Good Afternoon!** directly to the output stream.
- A **<jsp:include> action** that includes the contents of another JSP page in the current page. The `<jsp:include>` action includes content at runtime. JSP also provides a compile-time include mechanism. To include content that should be evaluated at compile time, use the `<%@ include >` directive.

Mixing HTML and Java As you can see from this example, JSP pages can contain both HTML and Java code. Using both works in this example because it is very simple and is meant only to demonstrate JSP features. However, interspersing HTML and Java in the same file may not be desirable in larger applications. Web page designers don't necessarily know Java, and Java programmers often don't write HTML as well as page designers. Furthermore, by maintaining HTML and Java in the same place, you blur the distinction between static content and dynamic content.

For these reasons, you will usually want to keep your Java code separate from your JSP pages. You can do this in two ways:

- Maintain your Java code in JavaBeans components and make calls to these components from your JSP pages.
- Encapsulate your Java code in tag libraries and use custom tags to perform actions implemented in these libraries.

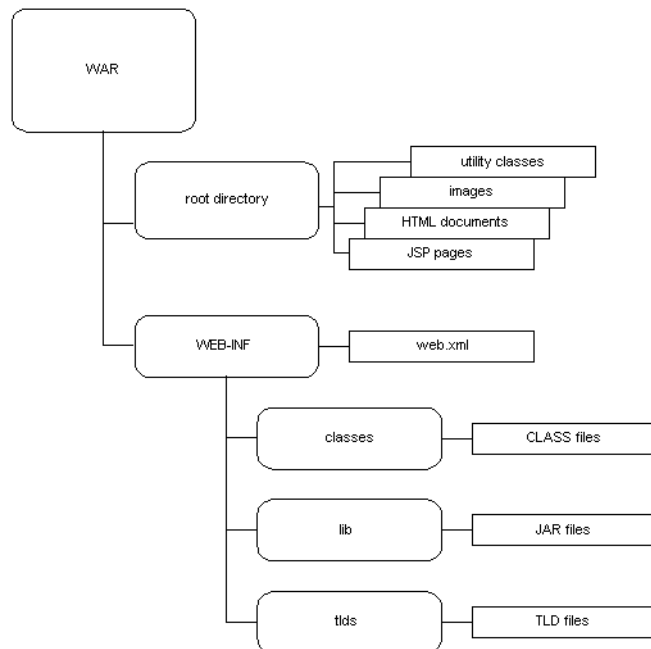
Both of these techniques are illustrated in the sample page.

Developing JSP pages

To develop JSP-based applications (Web applications), you write your JSP pages, Java servlets, JavaBean components, and other supporting Java classes, as follows.

1. Create a directory structure for your application that conforms to the format required for the Web application. The directory structure should look something like the following.

NOTE SilverStream eXtend Workbench allows you to organize your files any way you want and map the file locations to the structure required for a WAR file. If you are new to JSP pages and Workbench, you might want to first organize your files to match the WAR specification to get used to JSP development. Then later you can take advantage of the flexibility that Workbench provides to organize your files any way you want. For more information, see Projects and Archives in the *Tools Guide*.



WAR—Web archive file. Container for Web-based application.

root directory—Can contain JSP pages, HTML documents, and any other contents for the application. They could also be in subdirectories off the root. For example, you might want to put your JSP pages in a directory called **jsps**.

WEB-INF—A required subdirectory that contains all of the components of the application that should not be available directly to clients. The WEB-INF directory must contain a file called **web.xml** that is the deployment descriptor for the Web application.

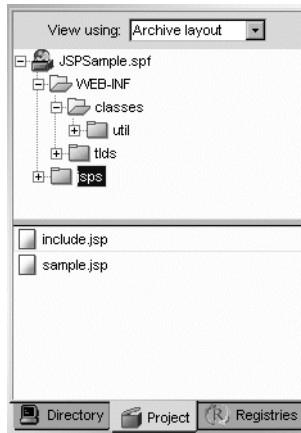
The WEB-INF subdirectory can contain the following subdirectories:

- **classes**—Directory containing servlet and utility classes
- **lib**—Directory containing JARs of servlets, JavaBeans, and other utility classes

In addition to the classes and lib subdirectories, the WEB-INF subdirectory can optionally have other subdirectories. You can give these subdirectories any names you like. For example, you might include a subdirectory named **tlcls** that contains tag library descriptor files.

2. Write your JSP pages and save them in the root directory of the Web application or a subdirectory of the root.
3. Create any Java servlets, JavaBean components, or other supporting Java classes required by the application and compile these classes.

In Workbench Here is the SilverStream eXtend Workbench project that was created for the sample application whose main JSP page was shown above.



The two JSP pages are in the jsp subdirectory. The tag library definition file (SampleTags.tld) is in the tlcls subdirectory. All the Java source files are in the classes/util directory. They are:

- JspCalendar.java and JspSQL.java, the two JavaBeans referenced in the `<jsp:useBean ... >` action
- The Java files that implement the tag library (the one used by the page is SimpleTag.java)

Before the application was packaged, all the Java files in the project were compiled.

Note that the JSP pages do not get compiled at this step. They get translated into Java servlet source files, then compiled, on the server when you deploy the application. So the files that get packaged in the WAR include:

- JSP sources
- Static resources, such as HTML pages, graphics, and style sheets (the sample application doesn't use any of these)
- Compiled servlet and utility classes, either as CLASS files or as JAR files
- Tag libraries

Packaging the application

Once you have written the components of your Web application, you package the application in a WAR. To do this, you:

1. Create a **deployment descriptor** for the application.
The file must be named **web.xml** and you must save it in the **WEB-INF** directory.
2. Create a WAR file (a JAR file with the .WAR extension) and add the JSP source files and other application components to it.

Writing the deployment descriptor

The web.xml file is the deployment descriptor for a WAR file. It contains configuration information like:

- Security mappings
- Servlet/JSP mappings
- Error pages
- Tag libraries used

Much of the information specified in the web.xml file pertains to servlets provided with the Web application. If you want to make a servlet directly accessible to the user through an URL, give the servlet a name and an URL pattern in web.xml. Note that you don't need to specify names and URL patterns for JSP pages that are placed in a public directory outside the WEB-INF directory; they are automatically available for user requests. (You can, however, map JSP pages in web.xml if you want them accessible through URLs that are different from URLs that match the location of the JSP files in the WAR file.)

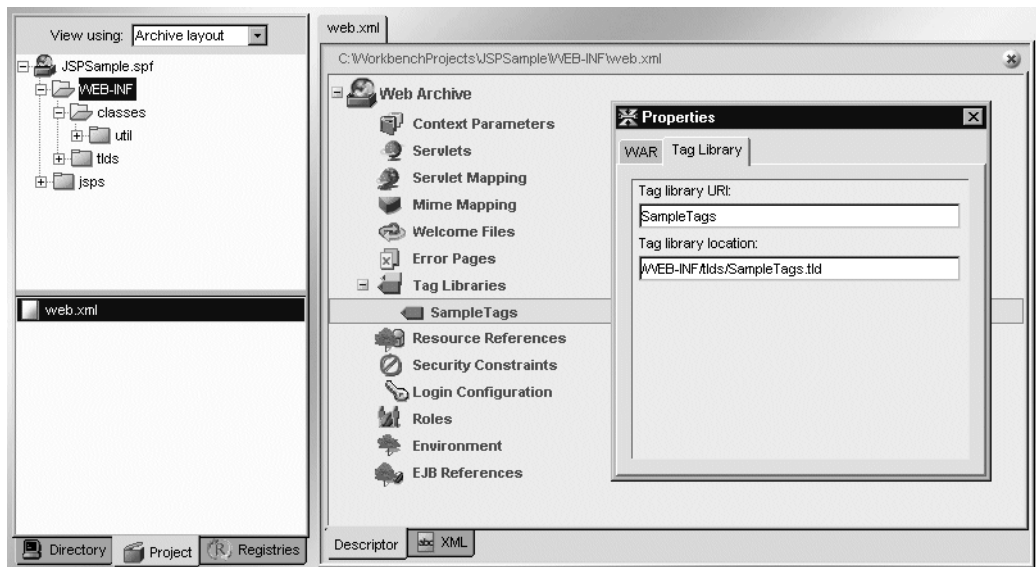
The web.xml file must follow the format specified by the Sun J2EE Web application DTD called **web-app_2_2.dtd** located in the Resources/DTDCatalog subdirectory of your Workbench installation. Version 2.2 of the Java servlet specification provides complete documentation on each tag. You can find this document on the Sun Java Web site at <http://java.sun.com/j2ee/docs.html>.

Here is the sample application's web.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <taglib>
    <taglib-uri>SampleTags</taglib-uri>
    <taglib-location>/WEB-INF/tlds/SampleTags.tld</taglib-location>
  </taglib>
</web-app>
```

The application's web.xml file is very simple. It only has one entry, <taglib>, which maps the tag library's uri (which, you'll remember, is specified in JSP pages that use the tag library) to a file location in the WAR (in the sample application, the SampleTags.tld file is in the tlds subdirectory of the WEB-INF directory).

In Workbench You can use the Deployment Descriptor Editor to easily create and maintain your application's deployment descriptor.



Creating a WAR file

A Web application must be packaged in a WAR file. You use the archive tool of your choice to create a WAR file.

In Workbench To create your archive (WAR file), you can select **Project>Build and Archive** (which compiles any Java files that need to be compiled, then creates the WAR file) or **Project>Rebuild All and Archive** (which compiles all Java files in the project, then creates the WAR file).

Deploying the application

To make your application available to users, you deploy it on a J2EE server, such as the SilverStream eXtend Application Server. You:

1. Specify in a file the runtime deployment information specific to your application.
This step is server-specific (it is not specified in the J2EE standard). Each J2EE server has its own requirements for specifying runtime deployment information. For example, the SilverStream eXtend Application Server uses a *deployment plan*, and the Sun Reference Implementation uses a *Runtime Deployment Descriptor*.
2. Deploy the application.

What happens at deployment time The server does the following:

1. Compiles all JSP pages in the WAR into Java source files.
The Java source file defines a class that implements the `HttpJspPage` interface. It imports the following packages by default:
 - `javax.servlet.*;`
 - `javax.servlet.http.*;`
 - `javax.servlet.jsp.*;`
 - `java.lang.*;`If necessary, you can import additional packages or classes by using the `import` attribute of the JSP page directive.
NOTE Some J2EE servers compile JSP pages at runtime, not at deployment time.
2. Compiles the Java sources.
The code generated for the Java class conforms to the JSP 1.1 specification.
3. Adds the results to the deployed WAR file.
4. Makes all resources in the WAR available for user requests.

In addition to the JSP pages, the deployed WAR can contain servlet classes and other supporting Java files that were compiled separately, as well as HTML documents, images, and any other files required by the application. Note that the deployment does not compile Java source files that are not .JSP files.

In Workbench To deploy your application:

1. Define a server profile for the J2EE server you want to deploy your application to.



For more information, see Setting Workbench profiles in the *Tools Guide*.

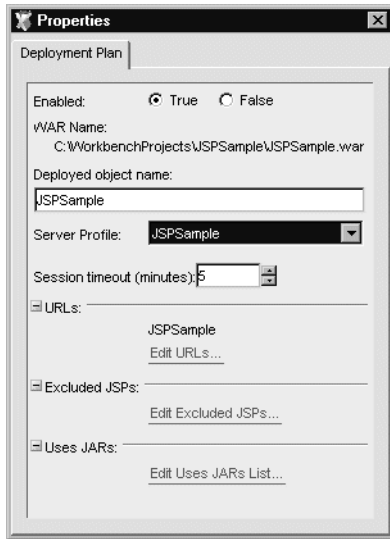
2. Make sure the server is running and accessible.
3. Select **Project>Deploy Archive**.
4. Fill in the information in the Deployment dialog.

The information you need to provide depends on the server you are deploying on. The Deployment dialog displays only information relevant to the specified server.

5. Click **OK** to deploy the application.

Here is how the sample application was deployed on the SilverStream eXtend Application Server. The SilverStream eXtend Application Server uses a *deployment plan*, an XML file that provides additional information about the contents of the WAR file and how it should be deployed in the SilverStream environment. You create your plan in the Deployment Plan Editor. For information about deployment plans, see Deployment Plan Editor in the *Tools Guide*.

Here is how the properties were specified in the Deployment Plan Editor for the sample application:

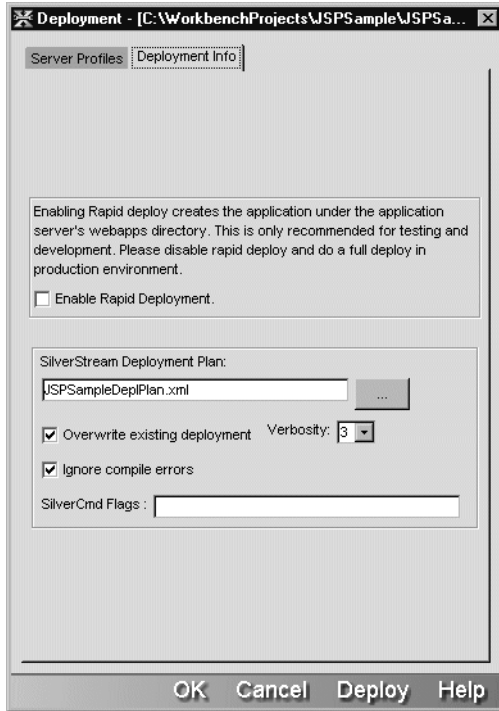


Here is the xml file that Workbench created from these specifications:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE warJarOptions PUBLIC "-//SilverStream Software, Inc.//DTD J2EE WAR Deployment
Plan//EN" "deploy_war.dtd">
<?AgMetaXML 1.0?><warJarOptions isObject="true">
  <warJar isObject="true">
    <warJarName type="String">C:\WorkbenchProjects\JSPSample\JSPSample.war</warJarName>
    <isEnabled>True</isEnabled>
    <sessionTimeout type="String">5</sessionTimeout>
    <urls type="StringArray">
      <el>JSPSample</el>
    </urls>
    <deployedObject type="String">JSPSample</deployedObject>
  </warJar>
</warJarOptions>
```

The deployment plan enables the application, sets the session timeout for the application, and defines the URL that can be used to access the application.

Finally, the application was deployed by selecting **Project>Deploy Archive**, filling in the Deployment dialog, and clicking **OK**:



Running the application

Once the application has been deployed, you can run it in your browser by specifying the appropriate URLs.

Here is the sample application:



Note the parts of the URL:

- **localhost** is the name of the server
- **JSPSampleDB** is the database

- **JSPSample** is the URL specified in the deployment plan (by default, the name of the WAR)
- **jps** is the directory containing the JSP page (there was no mapping in the deployment descriptor, so you specify the relative path from the WAR's root)
- **sample.jsp** is the JSP page

Now that you know the basics of JSP pages and their development, you'll probably want to get one of the many JSP books on the market and start developing your own.

4

Writing Servlets

This chapter tells you how to use servlets in a J2EE application and includes these topics:

- About servlets
- Developing a servlet
- Packaging the application
- Deploying the application
- Running a servlet

This chapter assumes that you understand the HTTP protocol and are familiar with the contents of HTTP request and response headers. For more information, see the JDK documentation or the Servlet home page provided by Sun at <http://java.sun.com>.

About servlets

Servlets are J2EE components that run on the server, allowing you to extend the server's functionality. A servlet is associated with one or more URLs. The servlet executes when a client (such as a browser) makes an HTTP request to one of these URLs.

Servlets can be used to:

- Access enterprise data using JDBC or EJBs
- Perform application logic on that data
- Generate an HTTP response to the client
- Maintain session data throughout a Web application

Servlet life cycle

When a client application (typically a Web browser) sends an HTTP request to an URL that is associated with a servlet, the J2EE server processes this request by handing it off to a servlet container. This container is responsible for managing the servlet life cycle from loading and initialization through request handling and servlet removal.

Servlet loading, instantiation, and initialization

Before a servlet can handle HTTP requests from clients, the container must:

- Load the servlet class
- Instantiate an object instance of the servlet class
- Initialize the servlet object by invoking the `init()` method of the servlet interface

The servlet class loading and instantiation can occur when the container starts or when the container determines that it needs the servlet to service a request.

The container calls the `init()` method only when first creating the servlet; it does not call `init()` again for each user request.

Request handling

Once a servlet is initialized, the container may use it to handle HTTP requests.

Each time the server receives an HTTP request for a servlet, the container creates an object of type `HttpServletRequest` to represent the request, and an object of type `HttpServletResponse` so the servlet can create a response. The container calls the `service()` method of the servlet interface, passing these two objects.

The `service()` method checks the HTTP request type (GET, POST, PUT, DELETE, and so on) and calls the appropriate methods in the servlet interface (`doGet()`, `doPost()`, `doPut()`, `doDelete()`, and so forth) as appropriate. Most of the servlet request processing logic appears in these methods.

The servlet can use the `HttpServletRequest` object to determine who the remote user is, what HTML form parameters may have been sent, and other data pertinent to the HTTP request. The servlet can use the `HttpServletResponse` object to create an HTTP response to send back to the client.

End of service

The servlet container may remove a particular servlet instance (for example, as the result of a specific server administration command or because the container wants to conserve memory resources). When the container determines that a servlet should be removed from service, it calls the `destroy()` method of the servlet interface.

Note that the `destroy()` method is called only when the servlet container removes the servlet as part of its regular processing. If the container is halted improperly (for example, if the server crashes), the code in this method might never be run before the servlet is removed.

Servlets and JSP pages

In J2EE, both servlets and JavaServer Pages (typically called JSP pages) can deliver dynamically generated content.

Servlets are a programmatic tool, in which your HTTP response (HTML, XML, or other format) must be coded within Java print statements. Servlets are designed to accept requests from browsers, possibly process information contained in the request, retrieve enterprise data, perform application logic on the data, and create the HTTP response.

JSP pages are a presentation-centric tool, coded in HTML-like pages. JSP pages support application logic using JavaBeans components, custom tags, and embedded Java scriptlets and expressions. JSP pages are designed to extend HTML pages to support application logic and to be modular, reusable presentation components.



For details about JSP technology, see Chapter 3, “Writing JSP Pages”.

Servlets and J2EE archive structure

In J2EE, servlets typically are packaged in Web archive (WAR) modules. WARs can contain servlets, JSP pages, and static Web content such as HTML files, pictures, sounds, movies, and so on.

In Workbench

Workbench creates a project for each major J2EE archive. When you create a project, Workbench asks you to specify what kind of archive the project is to implement—for example, an Enterprise archive (EAR), Web archive (WAR), application client JAR, Enterprise JavaBean JAR, and so on.

When you create a servlet in Workbench, your options include associating it with an existing WAR project, creating a new WAR project for it, or creating the servlet without specifying any project for it.

Developing a servlet

In J2EE, a servlet is typically a Java class that extends the standard Java class `HttpServlet`.

A servlet imports these packages:

- `javax.servlet.*`
- `javax.servlet.http.*`
- `java.io.*`
- `java.util.*`

To code the servlet, you typically override the various methods that are called by the `service()` method when handling requests. In most cases, this means you override at least the `doGet()` and `doPost()` methods to provide code that processes HTTP GET and POST requests.

In some cases, you might want to specify initialization and cleanup functionality by overriding the `init()` and `destroy()` methods.

Creating a servlet class in Workbench

Workbench provides a Servlet Wizard to help you to create a Java servlet class. When you run this wizard, Workbench creates a Java source file for your servlet based on information you supply. It also creates any directory structure resulting from project or package specifications.

Running the Servlet Wizard

To start this wizard, click **File>New** and select **Servlet** from the New File dialog.

The Servlet Wizard asks you to specify servlet characteristics such as:

- Servlet class name
- Content type of the document in the HTTP response the servlet is to generate
- Whether to allow multithreading of servlet request processes or to require that only one request process be handled at any given time
- Which WAR project (if any) is to contain the servlet
- Where on the file system the source file(s) for the servlet are to reside
- Where in the archive the class file(s) for the servlet are to reside
- What package (if any) contains the servlet

- Which HttpServlet methods you want to override
You can specify these HttpServlet methods in the wizard:
 - doGet()
 - doPost()
 - doPut()
 - doDelete()
 - init()
 - destroy()

You can override others manually after the wizard creates the servlet.

Example source file directory structure

If you specified that the servlet is to be part of a new or existing WAR or included in a package, the wizard creates the necessary file system directories to implement those choices. For example, if you specify the following when running the wizard:

- Your servlet class is called MwbiWelcomeUser
- The servlet is part of a WAR whose Workbench project directory is `d:\warProjects\welcomeUser`
- The servlet is part of a `com.mwbi.welcome` package

then the wizard creates this Java source file:

```
d:\warProjects\welcomeUser\com\mwbi\welcome\MwbiWelcomeUser.java
```

The wizard creates any directories that are specified in the wizard (for example, those resulting from package specifications, the servlet base directory, and so on) but do not yet exist in the file system.

Example servlet file source code

An example of a servlet file created by the wizard appears below. The Java code in this servlet file indicates that these characteristics were specified when running the Servlet Wizard:

- MwbiWelcomeUser is the class name
- The servlet is to be included in the package `com.mwbi.welcome`
- This servlet overrides only the `doGet()` and `doPost()` methods in the HttpServlet interface
- The output type of the HTTP response is to be HTML

4 Writing Servlets

```
package com.mwbi.welcome;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class MwbiWelcomeUser extends HttpServlet
{
    static final String CONTENT_TYPE = "text/html";

    // Handle the HTTP GET request
    public void doGet( HttpServletRequest request, HttpServletResponse response )
        throws ServletException, IOException
    {
        response.setContentType( CONTENT_TYPE );
        PrintWriter out = response.getWriter();

        /** @todo Process the HTTP "GET" request here, and write the proper
        response to the PrintWriter "out". */
        out.println( "<html><head><title>MwbiWelcomeUser</title></head><body>" );
        out.println( "<p>Servlet MwbiWelcomeUser has received an HTTP GET.</p>" );
        out.println( "<p>The servlet generated this page in response to the
request.</p>" );
        out.println( "</body></html>" );
    }

    // Handle the HTTP POST request
    public void doPost( HttpServletRequest request, HttpServletResponse response )
        throws ServletException, IOException
    {
        response.setContentType( CONTENT_TYPE );
        PrintWriter out = response.getWriter();
        /** @todo Process the HTTP "POST" request here, and write the proper
        response to the PrintWriter "out". */
        out.println( "<html><head><title>MwbiWelcomeUser</title></head><body>" );
        out.println( "<p>Servlet MwbiWelcomeUser has received an HTTP POST.</p>" );
        out.println( "<p>The servlet generated this page in response to the request.</p>"
);
        out.println( "</body></html>" );
    }
}
```


Processing the HTTP request

One of the main functions of a servlet is to process the HTTP request from the client. Typically, this includes performing some programming logic and generating content based on the data included in the request.

Typically, your servlet will handle HTTP GET and POST requests by overriding the `doGet()` and `doPost()` methods in your servlet class. These methods take two arguments:

- An object of type `HttpServletRequest` that represents the HTTP request from the client
- An object of type `HttpServletResponse` that you can use to create the HTTP response that will be returned to the client

Servlets support methods for other HTTP request types, such as PUT, DELETE, TRACE, and so forth. See your Java documentation for information on handling HTTP request types other than GET and POST in servlets.

This section describes the servlet functionality that reads information from the HTTP request. See “Generating the HTTP response” on page 56 for information about the servlet functionality that creates the HTTP response.

Reading HTML form data

One of the main reasons for generating automated Web content is to be able to base that content on user input. Typically, you obtain user data by reading the data in an HTML form that a user fills out.

Using the `getParameter()` method

Servlets can read HTML form parameters in the HTTP request using the `getParameter()` method in the `HttpServletRequest` interface. For example, if you want to read the value specified for a `login_userid` HTML form parameter into a variable in your servlet, you could code something like this:

```
userIDfromHtmlForm = request.getParameter( "login_userid" );
```

where:

- `userIDfromHtmlForm` is a predefined variable of type `String`
- `request` is an object of type `HttpServletRequest`
- `login_userid` is the name of the parameter specified in the HTML form in the HTTP request

This describes a very simple way to read and store client-supplied data in a servlet. In J2EE, there are many technologies that support this, including JavaServer Pages, custom tags, Enterprise JavaBeans, and so forth. Consult your J2EE documentation and Java programming resources for more information.

Reading HTTP request header information

HTTP request headers can contain lots of information that could be useful to your application, including:

- Cookie information
- Authorization information, such as authorization type and remote user
- Content information, such as length and type
- Date information

The `HttpServletRequest` interface supports a `getHeader()` method that can read any header you specify. For example, if you wanted to find out what character sets the client browser that sent the request can use, you could use this method:

```
request.getHeader( "Accept-Charset" );
```

Some of the more common headers have specific methods in the `HttpServletRequest` interface, such as `getCookies()`, `getAuthType()`, `getRemoteUser()`, `getContentLength()`, and so forth.

Consult the HTTP specification and your Java documentation for details about HTTP headers and how to read this information into your servlets.

Generating the HTTP response

Once the servlet reads the HTTP request information (as described under “Processing the HTTP request” on page 55), it typically generates some kind of a response in the form of an object of type `HttpServletResponse`.

The response object typically contains a status line, one or more response headers, and the actual document.

Specifying the status line

The HTTP status line contains the HTTP version, a status code, and a very short message corresponding to the status code. For example, a simple HTTP status line for a successful response could be:

```
HTTP/1.1 200 OK
```

Specifying the status code

Your server should specify a default status line (with a status code of 200) for your HTTP response as part of the processing for the methods `doGet()`, `doPost()`, and so on.

You can specify the status code explicitly using the `setStatus()` method of the `HttpServletResponse` interface.

CAUTION *If you want to specify the status code explicitly, you must do so before writing any document content. (See “Specifying the document content” on page 59 for more information.)*

This method takes an integer as an argument. However, instead of using an explicit number, you should use the constants defined in the `HttpServletResponse` interface. Examples of common status constants include:

- `SC_CONTINUE` (100)
- `SC_OK` (200)
- `SC_CREATED` (201)
- `SC_MOVED_PERMANENTLY` (301)
- `SC_MOVED_TEMPORARILY` (302)
- `SC_SEE_OTHER` (303)
- `SC_NOT_MODIFIED` (304)
- `SC_BAD_REQUEST` (400)
- `SC_UNAUTHORIZED` (401)
- `SC_FORBIDDEN` (403)
- `SC_NOT_FOUND` (404)
- `SC_INTERNAL_SERVER_ERROR` (500)

For example, to set the status code of the response to 403, you could use this method:

```
response.setStatus( response.SC_FORBIDDEN );
```

 See the Java documentation for the `HttpServletResponse` and the HTTP specification for details about the status code in HTTP responses.

Specifying HTTP response headers

HTTP response headers can provide:

- Accompanying information for particular status codes, such as locations for moved documents, authentication information, and so on
- Cookie information
- Page modification dates
- File sizes

The `HttpServletResponse` interface supports a `setHeader()` method that can define any header you specify. There are also specialized and convenience methods in `HttpServletResponse`, including:


Method	Functionality
<code>setDateHeader()</code>	Translates a Java date into a GMT time string
<code>setIntHeader()</code>	Converts an int to a String before inserting it into the header
<code>setContentType()</code>	Sets the Content-Type header
<code>setContentLength()</code>	Sets the Content-Length header
<code>addCookie()</code>	Inserts a cookie into the Set-Cookie header
<code>sendRedirect()</code>	Sets the Location header and sets the status code to 302

For example, to redirect the user to another page, you could use this method:

```
response.sendRedirect( url );
```

where **url** is a variable containing the URL to which you want to redirect the user.

CAUTION *If you want to specify any HTTP response headers, you must do so before sending any document content. (See “Specifying the document content” on page 59 for more information.)*

 For details about HTTP headers and how to read this information into your servlets, consult the HTTP specification and your Java documentation.

Specifying the document content

Writing the document content in the HTTP response that your servlet will generate requires you to specify:

- The type of the response content (HTML, XML, and so on)
- The content of the document in the response (for example, the actual HTML tags that the browser will render in the client display)

Specifying the content type

To specify the content type, you can use the `setContentType()` method of the `ServletResponse` interface. Typical response content types include:

- `text/html`
- `text/xml`
- `text/xhtml`
- `text/wml`

For example, to set the content type to HTML, you could use the following method:

```
response.setContentType( "text/html" );
```

In Workbench The Servlet Wizard creates a variable of type `String` that contains the content type you specified when running the wizard. For example, the code for a servlet that generates an HTML response would contain this variable declaration:

```
static final String CONTENT_TYPE = "text/html" ;
```

In the request-handling methods (such as `doGet()` and `doPost()`), there would be a method call like this:

```
response.setContentType( CONTENT_TYPE );
```

Writing the document content

To write the document content, you can configure a `PrintWriter` object and write the content to that object using `print()` and `println()` methods.

For example, to send a simple HTML “Hello, world” message as the response, you could use this code:

```
PrintWriter out = response.getWriter();  
  
out.println( "<HTML><HEAD></HEAD><BODY>" );
```

```
out.println( "<P>Hello, world</P>" );
out.println( "</BODY></HTML>" );
```

CAUTION *If you want to specify a status code or HTTP header for your response, you must do so before you write anything to your `PrintWriter` object. (See “Specifying the status line” on page 56 or “Specifying HTTP response headers” on page 58 for more information about the HTTP status line and headers.)*

In Workbench

The Servlet Wizard inserts code that sets the content type based on your input, defines a `PrintWriter` object to contain the HTTP response, and provides a template for writing your document content to the `HttpServletResponse` object.

For example, if you specified **HTML** under **Content Type** in the Servlet Wizard, the wizard creates this code in any method that handles HTTP requests and provides a response:

```
response.setContentType( CONTENT_TYPE );
PrintWriter out = response.getWriter();

/** @todo Process the HTTP "GET" request here, and write the proper
response to the PrintWriter "out". */

out.println( "<html><head><title>MwbiWelcomeCustomer</title></head><body>" );
out.println( "<p>Servlet MwbiWelcomeCustomer has received an HTTP GET.</p>" );
out.println( "<p>The servlet generated this page in response to the request.</p>" );
out.println( "</body></html>" );
```

where `CONTENT_TYPE` is defined as a static variable set to `text/html`, as described under “Specifying the content type” on page 59.

You must replace the `out.println()` statements to reflect your HTTP response document content.

CAUTION *If you want to specify a status code or HTTP header for your response, you must do so before you write anything to your `PrintWriter` object. (See “Specifying the status line” on page 56 or “Specifying HTTP response headers” on page 58 for more information about the HTTP status line and headers.)*

Specifying initialization and cleanup methods

If you want to define initialization and cleanup code for your servlet, you can override the `init()` and `destroy()` methods in your servlet class. (See “Servlet life cycle” on page 49 for more information about the `init()` and `destroy()` methods.)

In Workbench

When creating the servlet, the wizard asks if you want to override the `init()` and `destroy()` methods. If you specify that one or both are to be overridden, Workbench inserts skeletal method code into the servlet.

Wizard-supplied `init()` code

If you specify in the Servlet Wizard that you want to override the `init()` method, the wizard inserts this code into your servlet:

```
/**
 * This method is called once per instance of the servlet class.
 * Use this method to allocate any needed resources that should
 * be preserved for the life of the servlet instance.
 */
public void init( ServletConfig config )
    throws ServletException
{
    super.init( config );

    /** @todo Initialize any instance variables here. */
}
```

While the servlet specifies an argument of type `ServletConfig` in the `init()` method, this method can be specified without an argument. Typically, you would specify the no-argument form of `init()` if the servlet does not need to read any settings that vary from server to server.

If you do specify that `init()` takes the `ServletConfig` argument, the `super.init()` method must be the first statement in the method.

Wizard-supplied `destroy()` code

If you specify in the Servlet Wizard that you want to override the `destroy()` method, the wizard inserts this code into your servlet:


```
/**
 * This method is called once per instance of the servlet class,
 * after the application server is done with that instance.
```

```
        Use this method to free any resources owned by the
        servlet instance.
    */
    public void destroy()
    {
    }
```

Other servlet coding issues

This chapter provides only an overview of some of the issues you must confront when programming servlets. Other major topics that are outside the scope of this discussion include:

- Buffering content
- Tracking sessions
- Implementing security
- Accessing databases using JDBC and EJB
- Handling cookies
- Integrating servlets with JavaServer Pages
- Using Filters

 For detailed information about these topics, see the J2EE documentation, Java language documentation, books on programming servlets, and so forth.


Packaging the application

Once you have written the components of your Web application (including servlets, JSP pages, and other supporting components), you package the application into a Web archive (WAR) file.


This process is very similar to that described under “Packaging the application” on page 40 in Chapter 3, “Writing JSP Pages”, in that you:

1. Write a deployment descriptor for the Web application and specify the relevant information about the servlet.
2. Create a Web archive (WAR) file containing the servlet and any components required to support the servlet, such as JSP pages or supporting classes.

The main differences are in what you specify in the deployment descriptor, such as ServletContext initialization parameters, servlet mappings, servlet/JSP mappings, and so on.

 For more information about specifying servlet information in deployment descriptors, consult the servlet and J2EE documentation.

In Workbench To write your servlet information into a deployment descriptor, you can use the Deployment Descriptor Editor. To create a WAR file, you can use the appropriate archive commands on the **Project** menu in Workbench.

 For more information about writing deployment descriptors in Workbench, see Deployment Descriptor Editor in the *Tools Guide*.

Deploying the application

To make your application available to users, you deploy it on a J2EE server, such as the SilverStream eXtend Application Server. The deployment process is very similar to that described under “Deploying the application” on page 42 in Chapter 3, “Writing JSP Pages”, in that you:

1. Create a Workbench server profile for your application server, if one does not already exist.
2. Specify the runtime deployment information for your application as required by your J2EE server.
3. Deploy the application.

In Workbench To deploy the application to the J2EE server, select **Project>Deploy Archive**, as described under “Deploying the application” on page 42 in Chapter 3, “Writing JSP Pages”.

Running a servlet

Once the application has been deployed, you can run it in your browser by specifying the appropriate URLs.

5

Writing J2EE Application Clients

J2EE application clients are the standard way to provide Java-based clients that run on user machines and access J2EE servers. This chapter tells you how to use them in your own J2EE applications, including:

- About J2EE application clients
- Developing a client
- Packaging a client
- Deploying a client
- Running a client

About J2EE application clients

Although J2EE applications typically provide browser-based clients, they aren't always the answer. You may sometimes want to implement a Java-based client instead (or in addition), such as when:

- Users will access the application within an intranet
- The application requires a richer user interface (more sophisticated than the browser)
- The client needs to perform operations not supported in a browser environment

In J2EE, you do this by building a J2EE application client.

Client features

In several ways, a J2EE application client is just like a standalone Java application that runs on a user machine. It:

- Consists of one or more Java classes
- Is invoked at the `main()` method in one of those classes
- Executes in its own Java virtual machine (and runs until that VM is terminated)

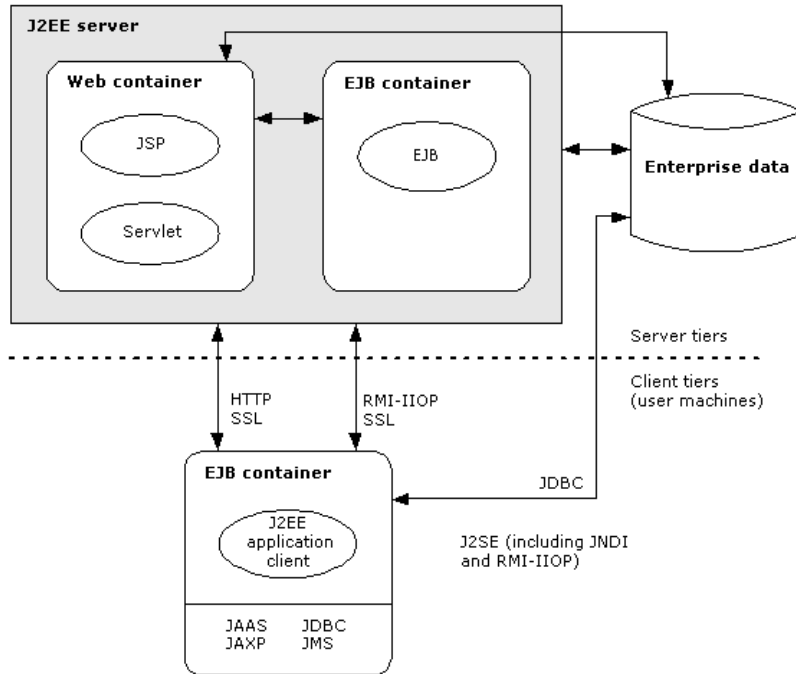
What makes a J2EE application client special is that it:

- Is a J2EE **component** that can be added to a J2EE application
- Is hosted by a J2EE **client container** on the user machine, enabling it to access J2EE services

- Is **portable** across J2EE servers

Basic architecture



The following illustration shows how J2EE application clients fit into the overall J2EE application architecture:



Note that while HTTP and JDBC are supported, RMI-IIOP is the typical means by which J2EE application clients communicate with servers.

Specifications and blueprints

Sun documentation is the authoritative source on J2EE application clients. It defines their standard features, operation, and life cycle, as well as vendor requirements for supporting them. See:

-  *Java 2 Platform Enterprise Edition Specification*, Chapter 9, "Application Clients"
-  *J2EE Blueprints*

These publications are available from the Sun Java Web site at:


java.sun.com/j2ee/docs.html

Client container

All J2EE application components rely on a container to provide supporting services. J2EE application clients are hosted by a client container that (at minimum) provides JNDI namespace access. Beyond that, the J2EE specification allows for a wide range of client container implementations, from basic to robust.

You can consult your J2EE server vendor to learn about the client container you should use. For example, SilverStream supplies a client container named **SilverJ2EEClient** that users can invoke to run J2EE application clients you've deployed to the SilverStream eXtend Application Server. SilverJ2EEClient provides a robust set of supporting services, including:

- Easy, self-updating container installation
- Automated client deployment to user machines
- User authentication and session housekeeping
- JNDI namespace access

 For more information on SilverJ2EEClient, see the *Facilities Guide* of the SilverStream eXtend Application Server Core Help.

Client life cycle

The life cycle of a J2EE application client consists of several phases, each handled by specific J2EE job roles:

Phase		What's involved	Which role handles it
1	Developing a client	Coding and compiling classes for the client	Component Provider
2	Packaging a client	Writing manifest and deployment descriptor files for the client Creating an archive (JAR file) to contain all of the client classes and other files	Component Provider or Application Assembler

Phase		What's involved	Which role handles it
3	Deploying a client	Preparing server-specific deployment information and using it to deploy the client JAR to the J2EE server	Deployer
4	Running a client	<p>Helping users install the client container on their machines</p> <p>Helping users invoke the client container and start the deployed client</p> <p>Administering the deployed client on the J2EE server</p>	Deployer or System Administrator

Depending on your organization, one or more people may take on these job roles. In particular, programmers developing client classes may need to test them by packaging, deploying, and running in their local environment.

Developing a client

Developing a J2EE application client involves:

1. Coding classes for the client
2. Compiling those classes

Coding client classes

Your J2EE application client can consist of one or more Java classes. The only requirement is that one class includes a **main()** method that can be invoked to start execution of the client.


Although you can code your client to do anything that Java allows, a common goal is to access a J2EE server—typically to call EJB session beans. When coding references to EJBs and other external entities, you should use names defined for them in the client's JNDI namespace. This helps keep deployment-specific details out of your classes, reducing the need for client code changes when external entities change.

 To learn about EJBs, see Chapter 6, “Writing Enterprise JavaBeans”.

Namespace setup

To set up the client's JNDI namespace, you need to write a **deployment descriptor** file that will accompany your classes. It defines names that let you reference:

- Environment entries
- EJB references
- Resource references:
 - JDBC data sources
 - JavaMail connections
 - JMS connections
 - URL connections

 To learn more about writing the deployment descriptor file, see “Packaging a client” on page 75.

API usage

You'll use these standard Sun APIs in the client classes you develop:

- **J2SE API** from the Java 2 Platform, Standard Edition SDK
- **J2EE APIs** from the Java 2 Platform, Enterprise Edition SDK

If you decide to use any vendor-specific APIs, remember that this can affect the portability of your client. Try isolating such code so that you can more easily replace it if that becomes necessary in the future.

Example: coding a client class

This example presents the Java code for a simple J2EE application client (which displays a one-line weather forecast for a specified day). It consists of a single class named `AppClientSample` that does the following:

- **Defines a `main()` method** to enable the client to be invoked
- **Reads a command-line argument** (passed from the client container to the client)
- **Accesses an EJB session bean** (using a bean reference defined in the deployment descriptor file) and calls one of its methods
- **Accesses an environment entry value** (using an environment entry defined in the deployment descriptor file)

- **Displays a message dialog** that includes the values obtained from the command-line argument, EJB method call, and environment entry

Here's the `AppClientSample.java` file:

```
package com.exsamp.appclient;

import java.io.*;
import javax.naming.*;
import javax.rmi.*;
import javax.swing.*;

import com.exsamp.ejb.*;

// The AppClientSample class shows how you can develop a class
// for use as a J2EE application client. It includes an example
// of using an environment entry and bean reference both defined
// separately in the deployment descriptor. The bean reference
// is used to call an EJB session bean on the server.

public class AppClientSample
{

    // Main method, used for application client startup (as
    // specified in the manifest file).

    public static void main(String[] args)
    {
        if (args.length < 1)
        {
            // Make sure all of the required command-line args have
            // been provided to the application client. Otherwise,
            // display an error message and terminate.
            JFrame frame = new JFrame();
            frame.show();
            JOptionPane.showMessageDialog(frame,
                "Required arguments:\n" +
                "* Day code (where 0=today, 1=tomorrow, etc.)" +
                "\n\nExample -- for today's forecast: 0",
                "Missing Command-Line Arguments",
                JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);
        }
        else
        {
            // Get the command-line args so the application client can
            // pass them to the AppClientSample constructor.
            try
            {

```



```

        int daycode = Integer.parseInt(args[0]);

        // Create an instance of AppClientSample. This executes the
        // constructor for the class, which calls a particular EJB
        // session bean.
        AppClientSample sample = new AppClientSample(daycode);
    }
    catch (NumberFormatException nfe)
    {
        System.out.println("AppClientSample requires one arg, " +
            "which must be an integer");
        System.exit(0);
    }
}

// Constructor for the AppClientSample class. It:
// * Finds a specific EJB session bean on the server
// * Calls a method of that session bean
// * Displays the result returned by that method (if any)
//
// It takes 1 argument: Day code (where 0=today, 1=tomorrow,
// etc.) Example -- for today's forecast: 0

public AppClientSample(int daycode)
{
    try
    {
        // Find the appropriate EJB session bean on the server.

        // Using a bean reference, do a JNDI lookup to return the
        // bean's home interface as an Object.
        InitialContext initCtx = new InitialContext();
        Object sbobj =
            initCtx.lookup("java:comp/env/ejb/myBean");

        // Narrow the Object returned by the lookup to make sure
        // it can be cast to the appropriate type (the class that
        // corresponds to your bean's home interface). Then, cast
        // it.
        sbobj = PortableRemoteObject.narrow(sbobj,
            SBMyEJBHome.class);
        SBMyEJBHome sbhome = (SBMyEJBHome) sbobj;

        // Call the home object's create() method to get an
        // instance of the bean's remote interface.
        SBMyEJB sbremote = sbhome.create();
    }
}

```

```
// Once you have the remote object, you're ready to call
// business methods of the EJB session bean. (These are
// the methods exposed by the bean's remote interface.)
String result = sbremote.getMyText(daycode);

// Now look up the value of the application client's
// environment entry reportTitle.

// Get the application client's environment naming
// context. Use the InitialContext object created earlier
// and stored in initCtx.
Context env = (Context)initCtx.lookup("java:comp/env");

// Get the reportTitle value set by the application
// client's deployer.
String title = (String)env.lookup("reportTitle");

// Display the result returned from the EJB session
// bean's business method, the title returned from the
// environment entry lookup, and the value of the passed
// command-line argument (daycode).
String day = "";
switch (daycode)
{
    case 0:
        day = "today";
        break;
    case 1:
        day = "tomorrow";
        break;
    default:
        day = Integer.toString(daycode) + " days from today";
}
JFrame frame = new JFrame();
frame.show();
JOptionPane.showMessageDialog(frame,
    "The forecast for " + day + " is:\n\n    " +
    result + "\n\n" +
    "Note: forecast obtained from getMyText() method " +
    "of EJB session bean SBMyEJB",
    title,
    JOptionPane.INFORMATION_MESSAGE);
}
catch (Exception e)
{
    System.out.println("Application error in AppClientSample");
    e.printStackTrace();
}
finally
{

```

```
        // Now that the application client is all done,  
        // make sure the VM terminates.  
        System.exit(0);  
    }  
}
```

In Workbench

To start coding a J2EE application client in SilverStream eXtend Workbench, you:

1. Create a CAR (client archive) project by using the New Project Wizard (**File>New Project**).



See the chapter on projects and archives in the *Tools Guide*.

2. Create the Java source files for your client classes. You can use the Java Class Wizard (**File>New**) to do that and add each file to your CAR project.



See the chapter on component wizards in the *Tools Guide*.

3. Edit your Java source files in the Java Editor. Use the Navigation Pane to open files you want to work on.

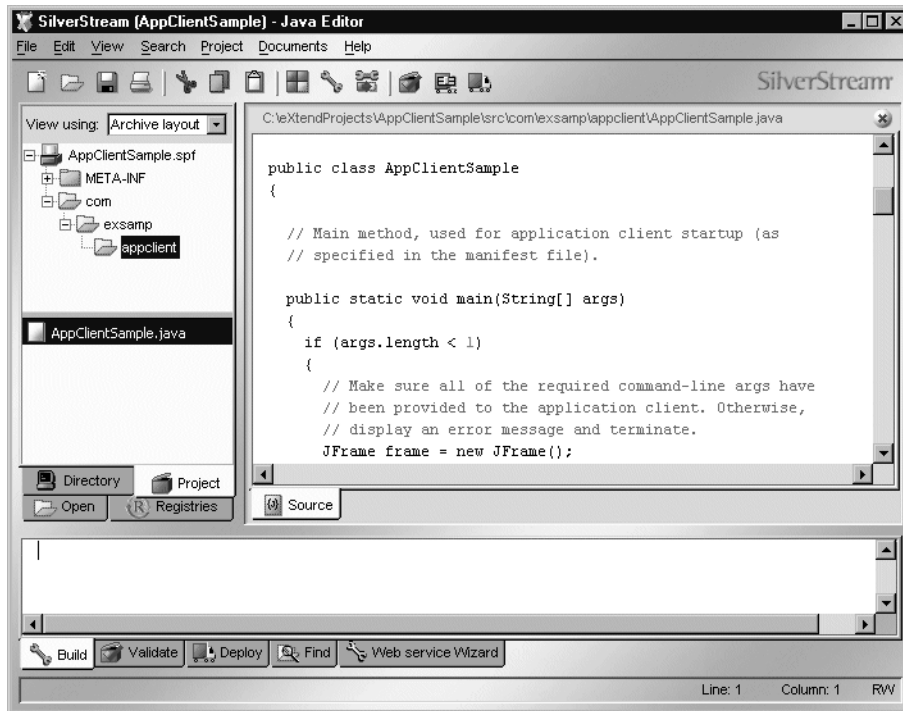


See the chapter on source editors in the *Tools Guide*.

For example, the following illustration shows the CAR project `AppClientSample.spf`. It contains the `AppClientSample.java` file and maps that class to this location in the archive:

```
com/exsamp/appclient
```

The project also includes a META-INF directory, which you'll learn more about shortly.



Compiling client classes

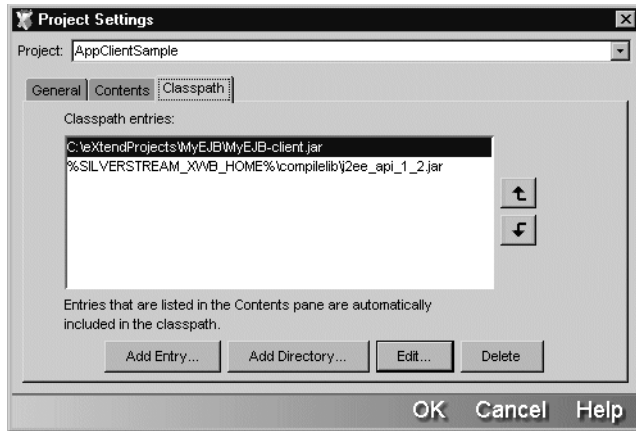
Compiling the classes you develop for a J2EE application client is just like compiling any other Java classes. You can use Sun javac or another comparable Java compiler. You just need to set your classpath so the compiler can find everything it needs, including:

- The **source files** for your client classes
- The **Java API packages** you use (both J2SE and J2EE)
- The **EJB-client JAR files** for any EJBs you access


In Workbench

Workbench automatically takes care of classpath requirements for the files in your project as well as the J2SE and J2EE packages. If you have other files to add to the project's classpath, you can do that in the Project Settings dialog (**Project>Project Settings**).

For instance, the `AppClientSample` class accesses an EJB session bean (which is independent of the CAR project). So before `AppClientSample` can be compiled, the EJB-client JAR for that bean must be added to the project's classpath:



Once your project's classpath is set, you can compile individual source files (**Project>Compile**) or build the project to compile everything (**Project>Build**).

 See the chapter on projects and archives in the *Tools Guide*.

Packaging a client

Packaging a J2EE application client involves:

1. Writing a manifest file
2. Writing a deployment descriptor file
3. Creating a JAR that contains the client files

Writing the manifest file

The manifest is a text file containing attributes that specify meta-information about a JAR file or other archive. For a client JAR, the only required attribute is **Main-Class**. It lets you specify the client class whose `main()` method is to be invoked when the client starts executing.

Make sure your manifest file ends with a new line.

Name and location

Your manifest file must be named:


```
MANIFEST.MF
```

It must be located in the following directory of your client JAR:

```
META-INF
```

Specification

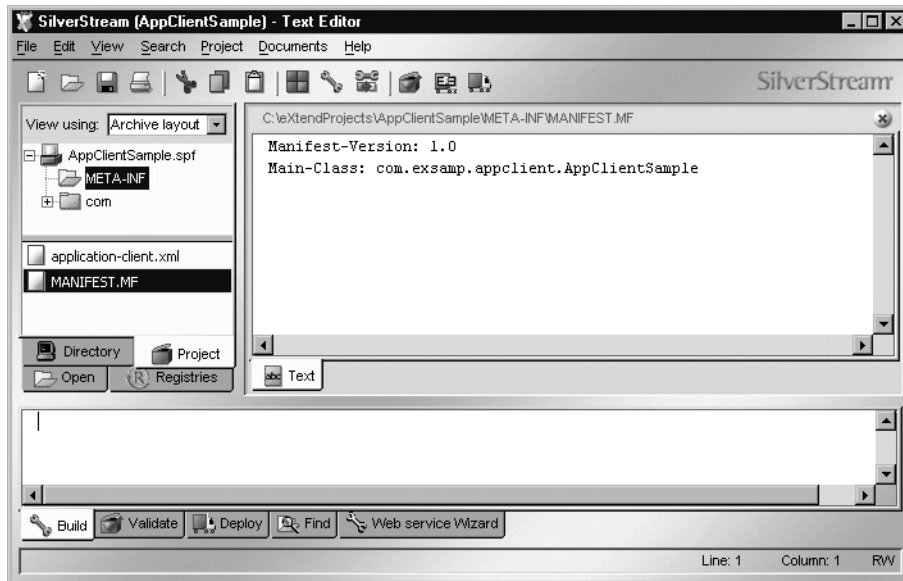
Sun documentation is the authoritative source on the JAR manifest and its attributes.

 See the *JAR File Specification* in the Java 2 Standard SDK.

In Workbench

You can select **File>New** to create a new text file for your manifest. Then you can edit it in the Text Editor and add it to your CAR project.

For example, here's the manifest file for the AppClientSample project:



It specifies the `AppClientSample` class as the `Main-Class`:

```
Manifest-Version: 1.0
Main-Class: com.exsamp.appclient.AppClientSample
```

Writing the deployment descriptor file

The deployment descriptor is an XML file that you use to define the external entities referenced by your client classes. As mentioned earlier, these include:

- Environment entries
- EJB references
- Resource references (JDBC, JavaMail, JMS, URLs)

As of J2EE 1.3, you also have the option of specifying a **callback handler** class (to be used by the client container to collect authentication information from users).

Name and location

Your deployment descriptor file must be named:

```
application-client.xml
```

It must be located in the following directory of your client JAR:

```
META-INF
```

Specification

When writing the deployment descriptor file for a J2EE application client, you enter information as a hierarchy of XML tags. The format to follow is determined by the Sun DTD (document type definition) for this file.

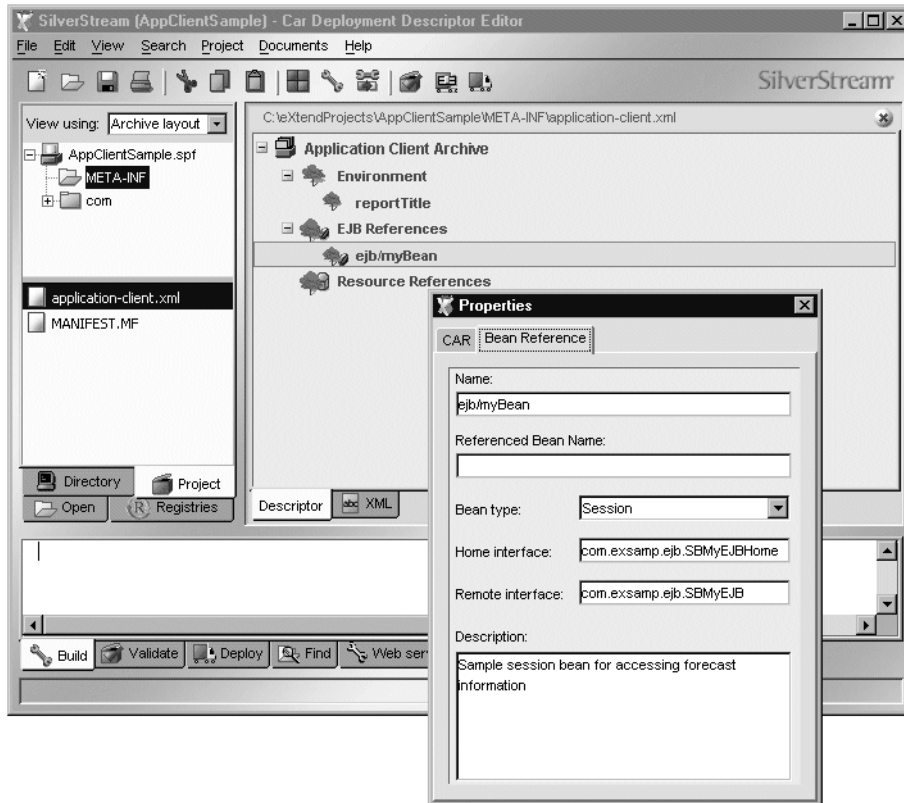
 For details, see the chapter on J2EE deployment descriptor DTDs in the *Reference*.

In Workbench

When you use the New Project Wizard (**File>New Project**) to create your CAR project, it automatically sets up a deployment descriptor file for you. Another way to create a deployment descriptor is by selecting **File>New**. Once you have this file in your project, you can edit it in the Deployment Descriptor Editor.

 See the Deployment Descriptor Editor chapter in the *Tools Guide*.

For example, this is the deployment descriptor file for the AppClientSample project:



The XML source for this deployment descriptor includes:

- Standard `<?xml ...>` and `<!DOCTYPE ...>` declarations
- Root tag `<application-client>`
- `<display-name>` and `<description>` tags that identify this J2EE application client
- An `<env-entry>` tag that defines the environment entry `reportTitle` used by the client
- An `<ejb-ref>` tag that defines the EJB session bean reference `ejb/myBean` used by the client

Here's the complete file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
Client 1.2//EN" "http://java.sun.com/j2ee/dtds/application-client_1_2.dtd">
<application-client>
  <display-name>AppClientSample</display-name>
  <description>Sample J2EE application client that calls a session bean</description>
  <env-entry>
    <description>Environment entry used to provide report title text to the client
    </description>
    <env-entry-name>reportTitle</env-entry-name>
    <env-entry-type>String</env-entry-type>
  </env-entry>
  <ejb-ref>
    <description>Sample session bean for accessing forecast information</description>
    <ejb-ref-name>ejb/myBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.exsamp.ejb.SBMyEJBHome</home>
    <remote>com.exsamp.ejb.SBMyEJB</remote>
  </ejb-ref>
</application-client>
```

Creating the client JAR file

The archive you create for a J2EE application client is a standard JAR file that includes:

- Your compiled client classes
- The manifest file
- The deployment descriptor file
- Any other supporting files this client requires

In Workbench

You can select **Project>Build and Archive** to create the client JAR file for your CAR project.



See the chapter on projects and archives in the *Tools Guide*.

For instance, performing the build and archive operation for the `AppClientSample` project generates the client JAR file `AppClientSample.jar` and displays these messages:



```
Buildfile: C:\eXtendProjects\AppClientSample\build\build-AppClientSample.xml
Building project "AppClientSample" - September 11, 2001 3:21 PM
Created dir: C:\eXtendProjects\AppClientSample\build\AppClientSample-classes
Compiling 1 source file to C:\eXtendProjects\AppClientSample\build\AppClientSample-classes
Building zip: C:\eXtendProjects\AppClientSample\AppClientSample.jar

BUILD SUCCESSFUL

Total time: 2 seconds
```


The screenshot shows a console window with a toolbar at the bottom containing buttons for Build, Validate, Deploy, Find, and Web service Wizard.

Deploying a client

Deploying a J2EE application client involves:

1. Writing deployment information for your J2EE server
2. Deploying your client JAR to that server

Deployment alternatives This chapter focuses on the simple case of deploying a lone client JAR directly to the server. But often it's advantageous to include client JARs in the context of a full J2EE application by deploying them to the server within an enterprise archive (EAR) file. Doing so provides better support for your application clients to reference other J2EE modules.

 For information on setting up an EAR project, see the chapter on projects and archives in the *Tools Guide*.



Writing server-specific deployment information

When deploying a client JAR, you'll usually need to provide server-specific information about that deployment. This includes mapping the environment entries, EJB references, and resource references defined in your standard deployment descriptor file (`application-client.xml`) to real entities in the target environment.

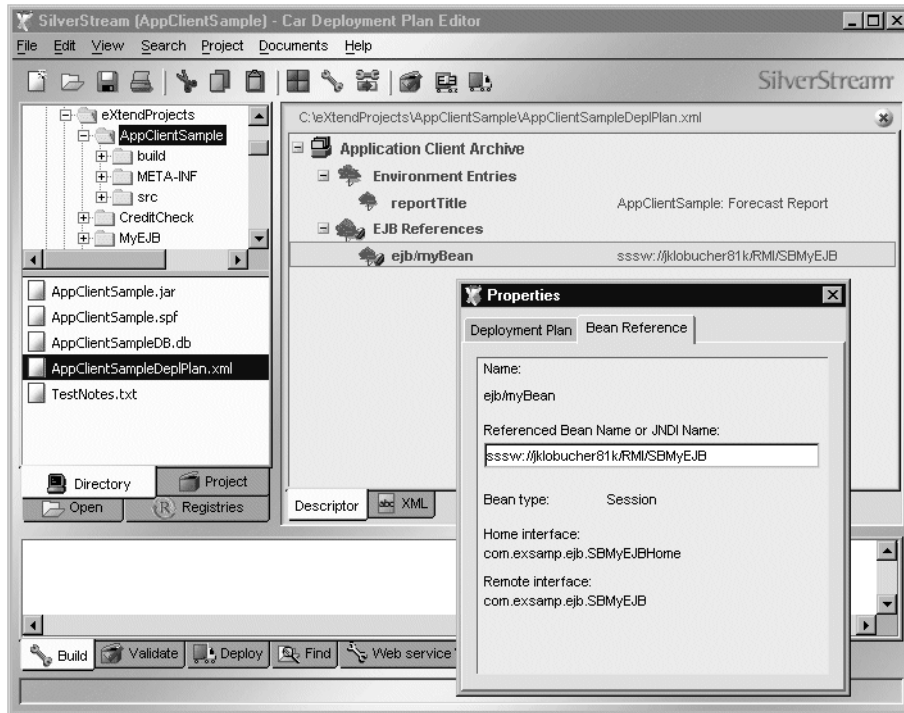
How you supply this information depends on the brand of J2EE server you're deploying to. Typically it's in the form of an XML file, similar to the standard deployment descriptor.

In Workbench

Workbench provides the following ways to create server-specific deployment information for your CAR project:

If your target server is	You can
SilverStream eXtend Application Server	<p>Select File>New to create a SilverStream deployment plan file. Then you can edit it in the Deployment Plan Editor.</p> <p>The deployment plan is an XML file. When saving it, you can specify any name and location. (You can store it with your project files on disk, but don't add it to the archive.)</p> <p>Once you have a deployment plan for your project, you can open it again later by going to the Project tab and right-clicking your SPF file.</p> <p> See the Deployment Plan Editor chapter in the <i>Tools Guide</i>.</p>
Another J2EE server	<p>Select File>New to create a new XML file for your deployment information. Then you can edit it in the XML Editor and save it with whatever name and extension (typically .xml) your server requires.</p> <p> For a summary of the deployment information required by specific J2EE servers, see the chapter on archive deployment in the <i>Tools Guide</i>.</p>

For example, suppose the client JAR from the AppClientSample project is to be deployed to the SilverStream eXtend Application Server. To prepare the required deployment information for the server, a deployment plan file named AppClientSampleDeplPlan.xml is created:



The XML source for this deployment plan includes:

- Standard `<?xml ...>`, `<!DOCTYPE ...>`, and `<?AgMetaXML ...>` declarations
- Root tag `<carJarOptions>` and the main tag it contains, `<carJar>`
- A `<version>` tag that specifies the internal version number this plan corresponds to
- An `<environmentList>` tag where details are specified for each environment entry used by this J2EE application client
 - One `<environmentEntry>` tag in this list that specifies the value to use at runtime for the reportTitle environment entry
- A `<beanReferenceList>` tag where details are specified for each EJB reference used by this J2EE application client
 - One `<beanReference>` tag in this list that maps the EJB reference ejb/myBean to the JNDI name (and server host) of a deployed EJB session bean to access at runtime

- A **<usesJars>** tag that lists additional JAR files to be downloaded from the server to user machines at runtime (and added to the classpath), along with the client JAR

Two **<el>** tags in this list that specify the EJB-client JAR file and remote EJB JAR file needed by the client to access the EJB session bean it references

Here's the complete file:


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE carJarOptions PUBLIC "-//SilverStream Software, Inc.//DTD J2EE CAR
Deployment Plan//EN" "deploy_car.dtd">
<?AgMetaXML 1.0?>
<carJarOptions isObject="true">
  <carJar isObject="true">
    <version type="String">1.0</version>
    <environmentList isObject="true">
      <environmentEntry isObject="true">
        <name type="String">reportTitle</name>
        <value type="String">AppClientSample: Forecast Report</value>
      </environmentEntry>
    </environmentList>
    <beanReferenceList isObject="true">
      <beanReference isObject="true">
        <name type="String">ejb/myBean</name>
        <beanLink type="String">sssw://jklobucher81k/RMI/SBMyEJB</beanLink>
      </beanReference>
    </beanReferenceList>
    <usesJars type="StringArray">
      <el>MyEJB-client.jar</el>
      <el>MyEJBRemote.jar</el>
    </usesJars>
  </carJar>
</carJarOptions>
```

Deploying the client JAR file

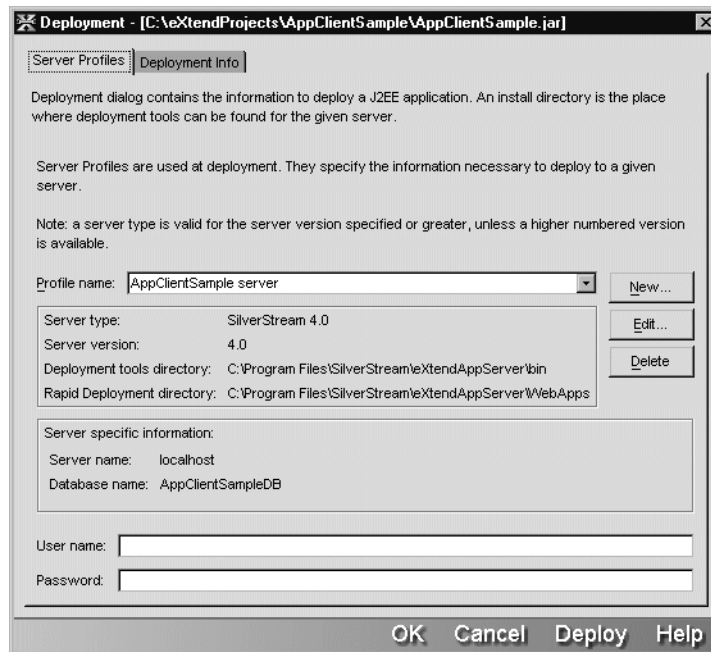
Once you have your client JAR and server-specific deployment information ready, you can deploy the J2EE application client. You can either use the native deployment tools provided with your target J2EE server or deploy from within Workbench.

In Workbench

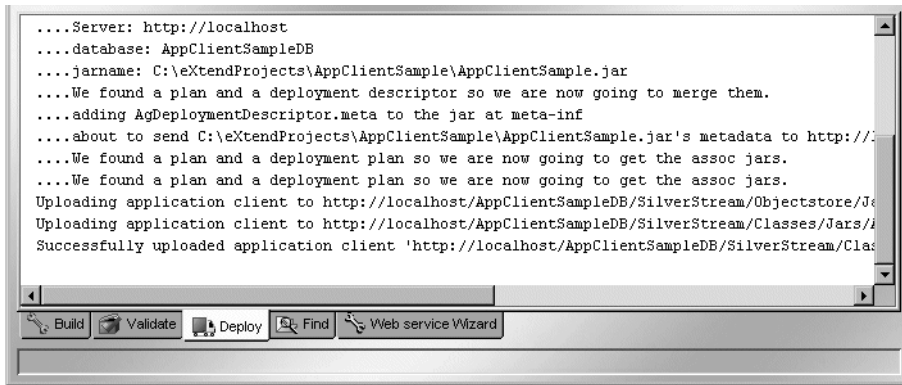
Workbench supports deployment to a variety of popular J2EE servers. Select **Project>Deployment Settings** to specify how you want to deploy. Select **Project>Deploy Archive** to deploy immediately using your current settings.

 See the chapter on archive deployment in the *Tools Guide*.

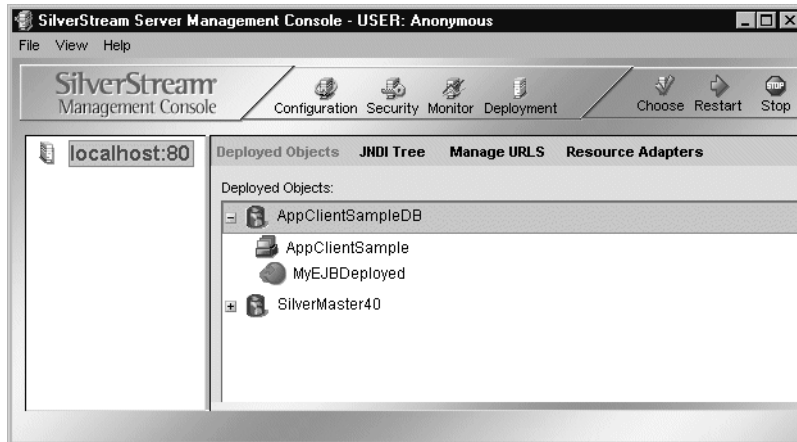
For instance, suppose the client JAR from the AppClientSample project is to be deployed to the SilverStream eXtend Application Server. This involves supplying the following deployment settings to the deploy archive operation:



Messages in the Output Pane indicate the status of the deployment:



In this case, the result is a J2EE application client deployment named AppClientSample that's ready for users to access from the SilverStream eXtend Application Server and run in the client container (SilverJ2EEClient). Here's how this deployment appears in the SilverStream Management Console (SMC):



Running a client

Running a J2EE application client involves:

1. Installing the client container on each user machine
2. Invoking the client container to start the deployed client

Consult your J2EE server vendor to learn about the client container you should use, how to install it, and how to invoke it.

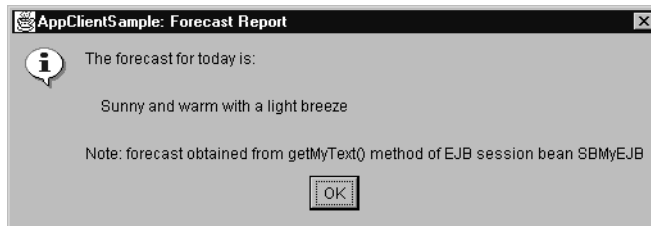


To learn about installing and invoking the SilverStream client container SilverJ2EEClient, see the *Facilities Guide* of the SilverStream eXtend Application Server Core Help.

For example, suppose the J2EE application client AppClientSample has been deployed to the SilverStream eXtend Application Server (as shown earlier) and you now want to run it. The following command line invokes the SilverJ2EEClient container, starts AppClientSample, and passes an argument (0) to the client:

```
SilverJ2EEClient jklobucher81k AppClientSampleDB AppClientSample 0
```

In this case, the client starts executing in the main() method of the AppClientSample class (as specified in the manifest). It then obtains some information (by accessing an EJB, an environment entry, and the passed argument) and displays it to the user:



6

Writing Enterprise JavaBeans

Enterprise JavaBeans (EJBs) are an important part of the J2EE application architecture. This chapter introduces EJBs. It covers these topics:

- About EJBs
- Developing EJBs
- Packaging EJBs
- Deploying EJBs
- Calling EJBs
- Tips for designing EJB applications

About EJBs

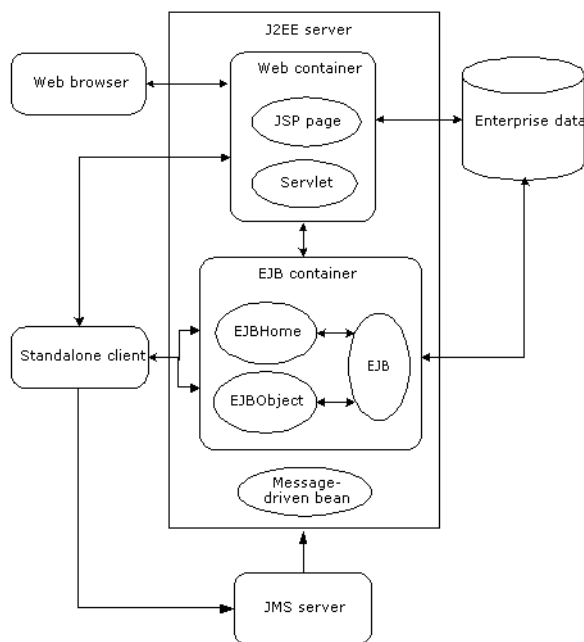
EJBs are reusable Java-based components that are transactional and secure and can be remotely accessible. You can use EJB components to provide the business logic for your application, link your application's user interface with a back-end information system, or handle JMS messages.

Sun defines the following EJB types:

EJB type	Description
Entity bean	<p data-bbox="482 343 1239 434">Represents data in an underlying data store like a relational or object database. Can also represent complex relationships among one or more related tables or components of a nonrelational data store.</p> <p data-bbox="482 454 1273 673">Since entity beans represent data in an underlying data store, the data contained in the entity instances must be synchronized with the data in the rows they represent. The process of synchronization is called <i>persistence</i>. Persistence includes creating, deleting, and modifying data rows. Creating an entity bean instance creates a row in the underlying data store; deleting an entity bean instance removes the row from the underlying data store; and so on.</p> <p data-bbox="482 694 882 725">EJBs support two persistence models:</p> <ul data-bbox="482 743 1273 986" style="list-style-type: none"><li data-bbox="482 743 1273 802">• Bean-managed persistence (BMP)—You write code that synchronizes the data to the underlying data store.<li data-bbox="482 822 1273 986">• Container-managed persistence (CMP)—You set properties in the deployment descriptor that describe how the container should synchronize the data. The most common CMP models are described in the EJB1.1 and EJB2.0 specifications. There is an EJB1.0 CMP model, but it is no longer widely supported.

EJB type	Description
Session bean	<p>Represents business processes and can be used to manage relationships among other entity or session beans. For example, you might use a session bean as a client shopping cart application, or to access and manipulate your enterprise data. Session beans are transient and do not represent persistent data.</p> <p>There are two kinds of session beans:</p> <ul style="list-style-type: none">• Stateful—A stateful session bean is bound to the client session that creates it, so it can be used to maintain values associated with that client session.• Stateless—A stateless session bean is released to the instance pool after each method call completes, so it is not guaranteed that a client will have the same instance on subsequent method calls.
Message-driven bean (2.0 only)	<p>Like a session bean, represents business processes. Resides in the EJB container and subscribes to or listens for asynchronous messages. When a message is received, the message-driven bean processes it and then waits for the next message. Message-driven beans can be used for the same types applications as session beans, but their methods cannot be called by a client; they can only respond to JMS messages.</p> <p>Message-driven beans are accessed via JMS.</p>

The following diagram shows how EJBs can be used in J2EE applications.



Benefits of EJB container From this diagram you can see that an EJB runs on a J2EE server within an **EJB container**. The EJB container (as defined by the EJB specification) provides the EJB runtime environment that includes such low-level services as naming services, remote access, security, and transaction support.

The EJB container provides two benefits:

- **You focus on business logic** Because you can rely on these services being available and accessed in a standard way, you can focus your development efforts on writing the business logic and not on low-level services.
- **Your EJBs are portable** Because all EJB containers must meet these requirements, EJBs can be portable across many EJB container implementations.

NOTE EJB container vendors can provide additional services for EJBs deployed on their systems. But EJBs developed to take advantage of nonstandard services are not portable.

How clients access the EJB You can also see from the diagram that EJB clients do not access the EJB directly. Entity beans and session beans are accessed via the **EJBObject** and the **EJBHome** object. The EJBObject provides access to the EJB's business methods; the EJBHome object provides access to the EJB's life cycle methods. A new feature for EJB2.0 allows beans within the same container to access one another using a local interface instead of a remote interface; this avoids the overhead of the remote calls. (The two new local interfaces are **EJBLocalHome** and **EJBLocalObject**.) Message-driven beans are not accessed via any interfaces. Client programs cannot access message-driven beans directly, because they are accessed only via a JMS message server.

Developing EJBs

The components you develop depend on the version of the EJB specification and the types of enterprise beans you are developing. This table shows what is required for each specification:

EJB version	Bean type	Interface or class that you must provide
EJB1.1	Entity or session beans	<ul style="list-style-type: none"> • Home and remote interface • Bean implementation class • (Optional) Primary key class (entity beans only)
EJB2.0	Entity or session beans	<ul style="list-style-type: none"> • LocalHome and local component interface and/or RemoteHome and remote interface • Bean implementation class • (Optional) Primary key class (entity beans only) • (Optional) Dependent objects (for entity beans only)
	Message-driven beans	<ul style="list-style-type: none"> • Bean implementation class

To test or deploy the EJBs you develop, you need to:

1. Package the beans and the interfaces in an EJB JAR file and include a deployment descriptor. (See “Packaging EJBs” on page 95 for more information on the deployment descriptor.)
2. (Optional) Assemble the beans (from one or more EJB JARs) into an application.
3. Deploy the EJB JAR on a J2EE-compatible server. (See “Deploying EJBs” on page 97.)
4. Write a client to call the EJB. (See “Calling EJBs” on page 97.)

Looking at a sample session bean

This sample shows the components of a stateful session bean and includes:

- The remote interface
- The home interface

- The bean implementation class

The remote interface Some things to note about this sample:

- The remote interface extends `javax.ejb.EJBObject`.
- The `doCalculation()` business method is included so that clients will be able to call it.
- All methods on the remote interface throw `java.rmi.RemoteException`.

```
/**
 * @(#)SBCalculator.java
 * SBCalculator is a Stateful session EJB (EJB v1.1).
 */
import java.rmi.RemoteException;
public interface SBCalculator extends javax.ejb.EJBObject
{
    public int doCalculation( int piFirstValue, int piSecondValue )
        throws RemoteException;
}
```

The home interface Some things to note about this sample:

- The home interface extends `javax.ejb.EJBHome`.
- The life cycle method `create()` corresponds to the `ejbCreate()` method on the session bean implementation class.
- The `create()` method throws both the `javax.ejb.CreateException` and the `java.rmi.RemoteException`.

```
/**
 * @(#)SBCalculatorHome.java
 */
import java.rmi.RemoteException;
import javax.ejb.CreateException;
public interface SBCalculatorHome extends javax.ejb.EJBHome
{
    public SBCalculator create() throws CreateException, RemoteException;
}
```

The bean implementation class Some things to note about the bean implementation class:

- It extends `javax.ejb.SessionBean`.
- It includes life cycle methods like `ejbCreate()` and `ejbRemove()`.
- It includes container callback methods like `ejbActivate()` and `ejbPassivate()`. These methods allow the container to manage the bean.
- It includes the implementation of the `doCalculation()` method.

```
/**
 * @(#)SBCalculator.java
 * SBCalculator is a Stateful session EJB (EJB v1.1).
 */
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;

public class SBCalculatorBean implements javax.ejb.SessionBean
{
    protected SessionContext m_sessionContext;
    public int m_iTotal=0;
    public void ejbCreate() throws CreateException
    {
    }
    public int doCalculation( int piFirstValue, int piSecondValue )
    {
        int iTotat = piFirstValue + piSecondValue;
        m_iTotal += iTotat;
        return m_iTotal;
    }
    public void setSessionContext( javax.ejb.SessionContext ctx )
    {
        m_sessionContext = ctx;
    }
    public void ejbActivate()
    {
    }
    public void ejbPassivate()
    {
    }
    public void ejbRemove()
    {
    }
}
```

What Workbench does

Workbench provides an EJB Wizard that prompts you for information about the EJB you want to create. It prompts you for:

- The version of the EJB specification you want to use (1.1 or 2.0)
- The kind of EJB you want to create: session bean (stateful or stateless), entity bean (BMP, CMP1.x, or CMP2.x), or message-driven bean
- The methods the EJB will contain (including parameters, return types, and exceptions)
- Information about the underlying data store (for entity beans)

When you have provided all of the information, the wizard constructs:

What the wizard constructs	Details
Remote and/or local component interface	Includes all of the implementation class's public methods You do not have to write any code
RemoteHome and/or LocalHome	Includes the life cycle methods that are required by the specification, plus any additional create() methods defined in the implementation class You do not have to write any code
Bean implementation class	Includes all necessary imports, member variables, and method skeletons for all methods you specify The resulting Java file also includes not implemented comments so you can quickly scan the class for what is complete and what is not
Primary key class	A separate primary key class is only generated when a bean field is not sufficient for use as a primary key.

Packaging EJBs

Once you have developed the components of your EJB, you package them in an EJB JAR. To do this you:

1. Write a deployment descriptor for the EJB JAR.
2. Create an EJB JAR file (a JAR with the .JAR extension) and add the EJB source files and any other utility classes needed by the EJB.

Writing the deployment descriptor

The *deployment descriptor* is an XML description of the contents of an EJB JAR file. This file can have any valid file name and be located in any directory; but in the EJB JAR file it must be named `ejb-jar.xml` and reside in a directory named `META-INF`. The `ejb-jar.xml` file must follow the format specified by the Sun Enterprise JavaBeans DTD. For more information about the DTD, see J2EE Deployment Descriptors in the online *Reference*.

An EJB deployment descriptor includes:

Contents	Defined by	Description
Description of the individual beans in the JAR	Bean developer	Information about the individual beans in the EJB JARs, such as the name of the EJB's Java class file and the names of its home and remote interfaces
Runtime attributes of the beans in the JAR	Application assembler	Information about the runtime attributes of the beans in the EJB JAR, such as entries that name roles, method permissions, and transaction attributes The deployer then uses this combination of information to install the EJB JAR on the target server and map this information to actual entities in the runtime environment

What Workbench does

When you create an EJB as part of an **EJB project**, Workbench automatically creates a deployment descriptor that complies with the EJB deployment descriptor DTD. As you add components to the EJB project, Workbench updates the deployment descriptor to keep the project and the deployment descriptor synchronized.

You use the Workbench **Deployment Descriptor Editor** to modify and update the `ejb-jar.xml` file. You can also create an EJB deployment descriptor using the Deployment Descriptor Editor.

Creating an EJB JAR file

An EJB must be packaged in an EJB JAR file. You can use the archive tool of your choice to create an EJB JAR file.

What Workbench does

Workbench automates the archiving process. You can use Workbench to compile and archive your EJBs using **Projects>Build and Archive**.

Deploying EJBs

To make your EJBs available to users, you deploy the EJB JAR on a J2EE server, such as the SilverStream eXtend Application Server. To do this you:

1. Provide the runtime deployment information specific to your application and server.
Each J2EE server has its own requirements for specifying the runtime deployment information. For example, the SilverStream eXtend Application Server uses a deployment plan, and the Sun Reference Implementation (RI) uses a Runtime Deployment Descriptor.
2. Deploy the EJB JAR.

In Workbench To deploy your EJB JAR:

1. Make sure the J2EE server is running and accessible.
2. Select **Project>Deploy Archive**.
3. Fill in the Deployment dialog.

The deployment information depends on the server you are deploying to. You use the server profile dialog to create a J2EE server profile that Workbench can use to execute the appropriate deployment tool based on the selected server.

4. Click **OK** to deploy the EJB JAR.

Workbench provides automatic deployment to several J2EE servers.



For more information, see the chapter on archive deployment in the *Tools Guide*.

Calling EJBs

To call an EJB on a J2EE server, a client must:

1. Find the EJB.
2. Create an instance of the EJB.
3. Call the bean's remote methods or send a JMS message to the appropriate topic or queue.

Finding the EJB

To find the EJB, the client application locates the EJBHome object in one of these ways:

- The JNDI name
- A bean reference using the environment context

Finding the home object using the JNDI name This example shows how to do a JNDI lookup and a `PortableRemoteObject.narrow()`:

1. Create an instance of the `javax.naming.InitialContext` class.

```
initialContext = new InitialContext();
```

2. Use it to call the `InitialContext.lookup()` method. In this example, the session bean's JNDI name is `SBCalculator` and it is located in the RMI subcontext.

```
Object obj = initialContext.lookup("RMI/SBCalculator");
```

3. Call the `javax.rmi.PortableRemoteObject.narrow()` to perform type-narrowing of the client-side representations of the home and remote interfaces. Then cast the returned object to the appropriate type (in this case `SBCalculatorHome`).

```
m_sbCalculatorHome = (SBCalculatorHome)  
    javax.rmi.PortableRemoteObject.narrow(obj,  
        com.examples.calculatordemo.SBCalculatorHome.class);
```

Finding a bean reference using the environment context This example shows how to find a bean using a bean reference from another J2EE component.

1. Create an instance of the `javax.naming.InitialContext` class.

```
m_initialContext = new javax.naming.InitialContext();
```

2. Create an instance of the environment context and call the `InitialContext.lookup()` method.

```
Context contextEnv = (Context) m_initialContext.lookup("java:comp/env");
```

3. Call the environment context lookup using the bean reference.

```
Object objEntityBeanLookup = (Object)  
    contextEnv.lookup("ejb/beanrefname");
```

4. Call the `javax.rmi.PortableRemoteObject.narrow()` to perform type-narrowing of the client-side representations of the home and remote interfaces. Then cast the returned object to the appropriate type.

```
m_myBeanHome = (myBeanHome)  
    PortableRemoteObject.narrow(objEntityBeanLookup,  
        com.examples.bankdemo.myBeanHome.class);
```

Instantiating an EJBObject

You call the `create()` method or a finder method on the resulting `EJBHome` to get an `EJBObject` for an entity bean:

```
m_myBean = m_myEntityBeanHome.findByPrimaryKey(pkCompany);
```

or a session bean:

```
m_myBean = m_mySessionBeanHome.create();
```

Calling the bean's remote methods Once your client has a remote reference to the EJB, you can call any of the exposed business methods as though the EJB were local. Your client application can call only methods exposed by the remote interface and the life cycle methods exposed by the home interface. Clients that access entity beans can also call methods on the primary key class.

The bean provider must provide some type of written documentation that describes the EJB's available business methods.

Tips for designing EJB applications

Designing a good EJB application means following the standard practices of designing any good database application—plus these EJB-specific practices:

EJB practice	Details
Use appropriate-weight components	<ul style="list-style-type: none"> • Define methods at the business logic level • Take advantage of EJB's built-in transaction support whenever possible
Keep transactions short	<ul style="list-style-type: none"> • Never start a transaction from a remote client (such as a form); if the client crashes, the database will be locked until the transaction times out • It is OK to start a transaction from a servlet or page; but the servlet or page should close the transaction before responding to the browser—don't keep transactions open across user interactions • Consider using session beans to manage transactions
Integrate with the user interface	When possible, call EJBs directly from a page or servlet—do not expose EJBs directly to remote Java clients
Always use the <code>javax.rmi.PortableRemoteObject.narrow()</code> method with bean lookups or references	Casting is not sufficient for <code>RemoteHome</code> and remote component interfaces.
Always implement an <code>equals()</code> method and a <code>hashCode()</code> method for entity bean primary key classes	These methods must override the default implementation (on <code>Object</code>) with the correct signature
Call the <code>remove()</code> method on session beans when you no longer need them	The <code>remove()</code> method will get rid of the bean instance and unexport the corresponding remote object

7 Using Resource Adapters

Resource adapters are an important part of the J2EE Connector technology. This chapter introduces resource adapters and includes the following sections:

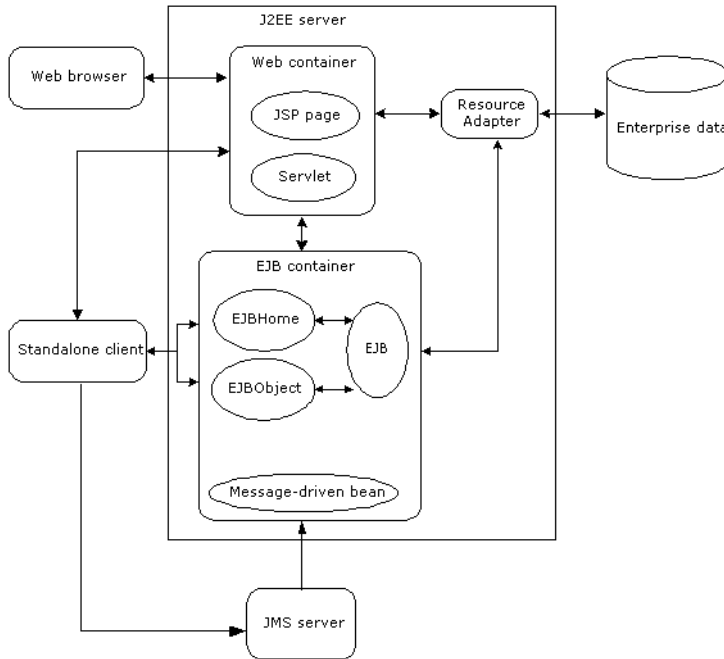
- About resource adapters
- Deploying resource adapters
- Using resource adapters

About resource adapters

Resource adapters are software components that reside on a J2EE server and allow J2EE components to interact with enterprise information systems (EIS) that reside outside of the J2EE server. Some examples of EIS systems include nonrelational databases, SAP, and PeopleSoft. A resource adapter is like a JDBC driver; it provides a standard API that J2EE application servers can use to access and provide services (like connection pooling and transaction and security management) for the EIS. Resource adapters also define and implement interfaces that J2EE client applications can use to access the resources managed by the EIS. The Connector Architecture Specification (1.0) defines a client interface called the Common Client Interface (CCI) that a resource adapter can implement for use by client applications, but it is not required.

Resource adapters are stored in resource adapter archive (RAR) files and are deployed to J2EE servers in the same way that other archive types are deployed. Once deployed, a resource adapter and its underlying EIS is not available to client applications until a Connector connection pool is created and configured using the target server's tools.

The following diagram illustrates how resource adapters can be used in J2EE applications.



Each resource adapter is developed to allow access a particular data source (EIS), so it is possible that more than one resource adapter is installed on any J2EE server. The J2EE Connector Architecture requires that the resource adapter implement the following contracts:

Contract	Description
Common Client Interface (CCI)	Defines APIs that clients can use to access data via a resource adapter. Resource adapter providers are not required to implement the CCI. The CCI API is common across heterogeneous EIS data stores and includes methods and classes to create and manage EIS connections and data. 📖 See the CCI Specification for more information.
Connection management	Defines the APIs that allow J2EE application servers to create and manage connection pools that can improve performance and scalability of applications using the resource adapter.

Contract	Description
Transaction management	<p>Defines the APIs that allow J2EE application servers to enlist EIS resources in global or local transactions via the resource adapter. The J2EE Connection Architecture defines the following types of resource adapters:</p> <ul style="list-style-type: none"> • XA (global)—transactions that can span multiple resource managers. Global transactions require coordination by an external transaction manager that will typically be bundled with the application server. XA transactions may require two-phase commit if the transaction spans multiple EIS applications. It will use a single-phase commit if only one EIS participates. • local transactions—transactions that are limited to a single EIS system and its associated resource manager (at the EIS). • nontransactional
Security	<p>Defines the APIs that allow J2EE application servers to support secure connections to EIS resources via the resource adapter. The security supported by the resource adapter is dependent on the requirements of the EIS.</p>

Deploying resource adapters

Resource adapters are stored in resource adapter archive (RAR) files and can be deployed on any J2EE-compatible server. The RAR file should include:

- The classes needed to implement the resource adapter.
- A deployment descriptor. The file must be called ra.xml and it must be stored in the META-INF file.

In addition, most J2EE servers will require a file that contains runtime deployment information, so you'll need to provide the file as required by the target J2EE server.

In Workbench To deploy your RAR:

1. Start Workbench and access the RAR from the file system.
2. Create a **deploy-only** project and add the RAR to it.

3. Make sure the J2EE server is running and accessible.
4. Select **Project>Deploy Archive**.
5. Fill in the Deployment dialog.

The deployment information depends on the server you are deploying to. You use the server profile dialog to create a J2EE server profile that Workbench can use to execute the appropriate deployment tool based on the selected server.

6. Click **OK** to deploy the RAR.

Workbench provides automatic deployment to several J2EE servers.



For more information, see the chapter on archive deployment in the *Tools Guide*.

Using Workbench to create resource adapters You can also use Workbench to create a resource adapter. For more information on using Workbench to develop J2EE components, see the *Tools Guide*.

Using resource adapters

Client applications never directly access the EIS or the resource adapter. Client applications access the resource adapter connection pool.

To access an EIS, client applications:

- Import `javax.resource.ResourceException` and any other packages necessary to use the resource adapter's client interfaces. For example: if CCI is used, the client application must import `javax.resource.cci.*` and
- Use JNDI to get the `ConnectionFactory` for the resource adapter (and the username and password values if necessary)
- Access an unused connection from the connection pool. When CCI is used, the client application would use the `getCCIConnection()` method.

Once you have the connection, you use the methods of the CCI or a proprietary interface defined by the resource adapter vendor to access the data.

The following code shows how to locate the `ConnectionFactory` (via JNDI) and to establish a connection using the CCI.

```
public void setSessionContext(SessionContext ctx) {
    try {
        m_sessionContext = ctx;
        Context ic = new InitialContext();
        username = (String) ic.lookup("java:comp/env/user");
        password = (String) ic.lookup("java:comp/env/password");
        Object obj=ic.lookup("java:comp/env/myEIS");
```

```
        cf=(ConnectionFactory) obj;
    } catch (NamingException ex) {
        ex.printStackTrace();
    }
}
```

Part II Producing and Consuming Web Services

A primer on Web Services that prepares you for creating and using them in Workbench

- Chapter 8, “Understanding Web Services”
- Chapter 9, “Generating Web Services”
- Chapter 10, “Generating Web Service Consumers”

8

Understanding Web Services

Web Services enable businesses to share application functionality regardless of the source language, operating system, or hardware used to create that functionality. Web Services overcome implementation incompatibilities by using **standard Internet protocols** and **XML-based messaging** to provide intercomponent communication.

This chapter gives an overview of Web Service technologies and SilverStream eXtend Workbench support for the development and use of Web Services. Topics include:

- About Web Services
- Web Service providers, consumers, and registries
- Providing Web Services
- Using Web Services
- Using Web Service registries
- Learning more about Web Services
- Popular Web Service implementations
- Web Services and Workbench

About Web Services

Web Services are modular software components whose application functionality is accessible over the Web using **Simple Object Access Protocol (SOAP)**, a standardized XML-based messaging protocol.

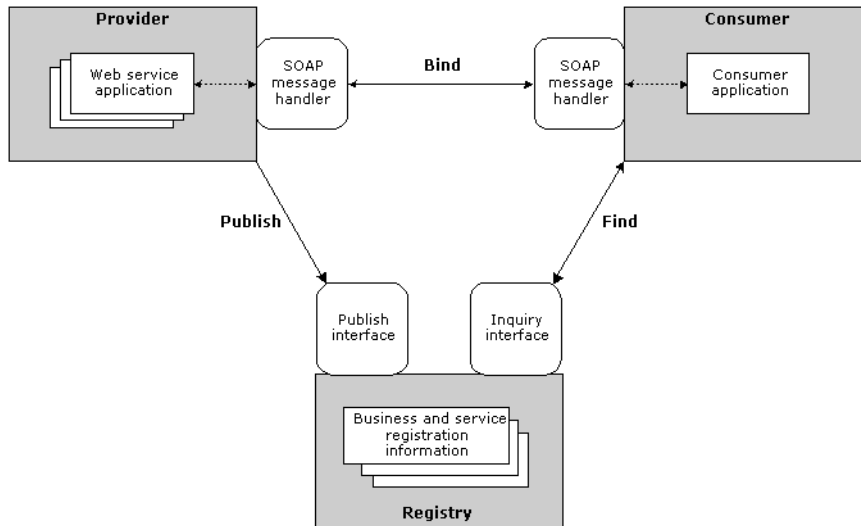
Applications invoke Web Services like remote procedure calls, except that the procedure call and response are handled using **SOAP messages** embedded in **HTTP requests and responses**. An application calls a Web Service by sending a SOAP message embedded in an HTTP request to a Web location associated with that service. The Web Service performs the application logic for that message then returns any application output in the form of another SOAP message embedded in an HTTP response.



To learn more about SOAP messages, see www.w3.org/TR/SOAP.

Web Service providers, consumers, and registries

The Web Service architecture typically consists of Web Service providers, consumers, and registries:



A Web Service **provider** is an organization that creates and hosts Web Services. Typically, a provider publishes information about their organization and the services they offer in a Web Service registry that can be queried by members of the organization or possibly by other businesses.

A Web Service **consumer** finds a Web Service (typically by querying a Web Service registry) then runs the service by establishing a connection to the provider. This is called **binding** to a Web Service.

A Web Service **registry** is a collection of business and service information that is readily accessible to providers and consumers, through programmatic publishing and querying interfaces.

Providing Web Services

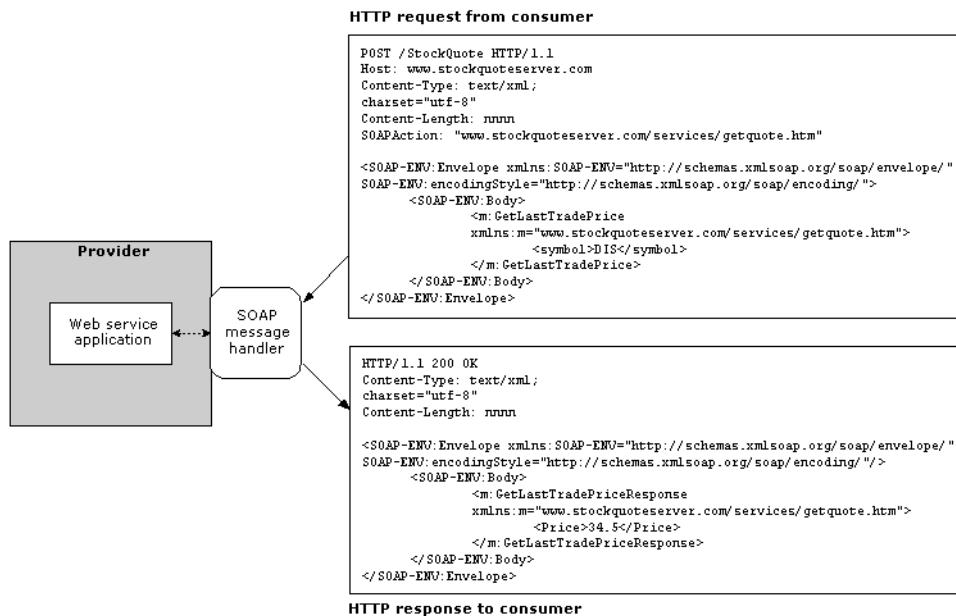
A Web Service provider:

1. Creates and deploys Web Service components
2. Creates a WSDL file to describe the Web Service
3. Publishes information about the Web Service so prospective consumers can discover and use it

Creating Web Service components

A provider creates the application logic components and deploys them to a network-accessible location, typically using a Web application server. To make these logic components into a Web Service, the provider creates and deploys a **SOAP message-handling interface** that enables HTTP requests containing well-defined SOAP messages to invoke the appropriate Web Service functionality.

When a consumer application accesses the service by sending a SOAP message embedded in an HTTP request, the provider runs the application logic and returns any application output in another SOAP message embedded in an HTTP response. For example:



Creating a WSDL file

To specify information about a Web Service in a standard form, the provider creates a **Web Services Description Language** (WSDL) document describing its characteristics. WSDL is an XML-based format that describes a Web Service by using these elements:

Element	Contains definitions of
Type	Data types specified in message content
Message	Data formats of messages
Port type	Endpoint types and the operations they support
Binding	Message formats and protocol details for a particular port type
Port	A network address for each endpoint
Service	Groups of related endpoints

In WSDL, an **endpoint** specifies a network address as well as the protocol and data format of messages exchanged with that address.

Given the flexibility of the WSDL specification, the information in a WSDL document can become complicated. For easier understanding, think of a WSDL document as essentially specifying the interface and port location of a Web Service.



To learn more about WSDL, see www.w3.org/TR/wsdl.

Publishing Web Service information

Once a Web Service has been created and deployed, the provider can publish information about the service and the provider organization in one or more registries. This enables prospective consumers to discover that the service is available and learn how to use it.



For details, see “Using Web Service registries” on page 114.

Another way to publish Web Service information is to provide the information directly to specific consumers by using Web pages, e-mail, personal communications, and so on. This is called **direct publishing**.

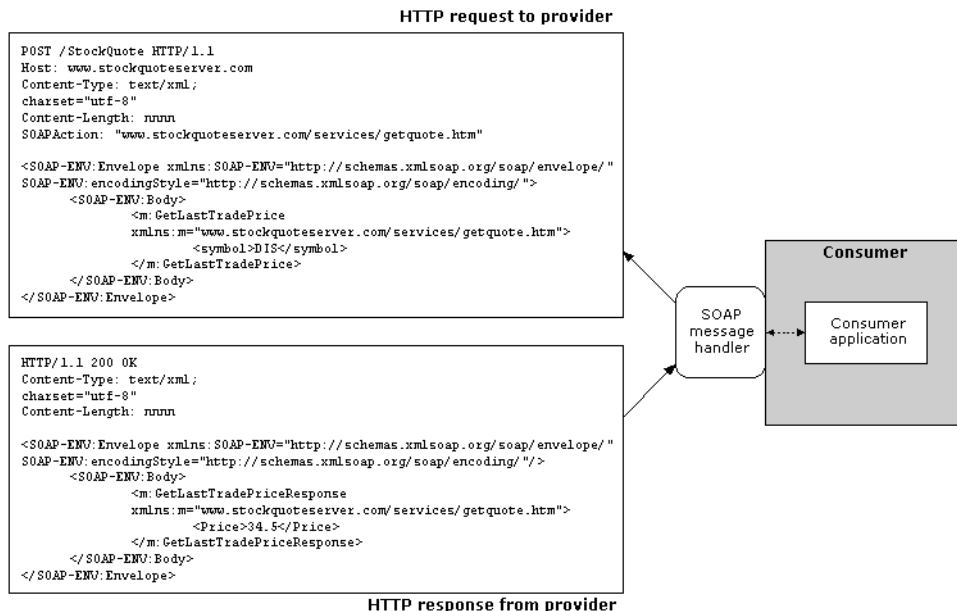
Using Web Services

A Web Service consumer creates applications that use Web Services. Typically, a consumer finds an appropriate Web Service by querying a Web Service registry (see “Using Web Service registries” on page 114).

From the WSDL information provided, the consumer can create the **SOAP message-handling code** needed to use the Web Service. When the consumer application calls the Web Service, the SOAP message-handling code **binds** to that service, as follows:

1. Establishes an HTTP connection to the provider
2. Creates and sends a SOAP message embedded in an HTTP request, instructing the provider to invoke the appropriate Web Service application logic
3. If the HTTP response contains a SOAP message, converts that message (into a data format understandable to the consumer application) then returns the data to the application

To the consumer application, this is similar to calling a remote method. However, the interaction between the application code and the Web Service uses SOAP messaging embedded in a standard HTTP request and response. For example:



Using Web Service registries

A Web Service registry is a repository of Web Service information that can be accessed programmatically over a network. Both providers and consumers can use Web Service registries:

- **Providers can publish** information about their organization and services to registries, making them visible to prospective consumers.
- **Consumers can query** registries to find the services and businesses that fit their needs and to retrieve provider-supplied information about those services (such as where and how to access them, the WSDL representation, and so on).


About registries

A registry can contain these kinds of information:

Category	Includes
Business information	Name, industry or product category, geographic location, and business identification numbers (such as NAICS or DUNS numbers)
Web Service information	General description, business process or category, and technical information (about connecting to and communicating with the Web Services for a given business)
Business service information	Corporate home page URL, sales and technical support contact information, business services not hosted on the Web, and so forth
Specification pointers	URL addresses of WSDL for services and other technical documents

Registry data formats

Registries store their business and service information in a standard XML-based format such as **Universal Description, Discovery, and Integration** (UDDI) or **Electronic Business XML** (ebXML). Businesses hosting registries typically provide Web page, GUI, or programmatic interfaces for publishing to and querying the registry (so providers and consumers don't need to know details about the internal registry implementation).

 To learn more about UDDI, see www.uddi.org. To learn more about ebXML, see www.ebxml.org.

Public and local registries

Businesses may use public or local registries:

- A **public registry** is typically visible to anyone on the Web and contains information about numerous companies and services. It may implement varying degrees of authentication and authorization security for publishing and querying.
- A **local registry** might be limited to local network access, enabling a business to share Web Services internally without exposing them to consumers outside the organization.

Learning more about Web Services

Here's a summary of Web sites you can visit to find out more about specific Web Service technologies:

Topic	Site
SOAP	www.w3.org/TR/SOAP
HTTP	www.w3.org/Protocols
WSDL	www.w3.org/TR/wsdl
UDDI	www.uddi.org
ebXML	www.ebxml.org

Popular Web Service implementations

While it's important to know about the underlying Web Service technologies (SOAP, WSDL, UDDI, ebXML, etc.), it's usually not efficient to develop applications at that level. As a result, higher-level implementations have emerged to make those technologies more accessible by wrapping them in familiar constructs. These implementations include:

- **J2EE** Java 2 Enterprise Edition provides Web Service support through its JAX-RPC (Java API for XML-based RPC) specification.
- **.NET** Microsoft provides Web Service support through its .NET platform.

For example, a programmer familiar with J2EE can more easily use a JAX-RPC implementation to develop and access Web Services. There's no need to become a SOAP expert or process SOAP messages manually.

When properly designed and built, Web Services should be **interoperable** across different implementations. For instance, a JAX-RPC client should be able to access a .NET Web Service and a .NET client should be able to access a JAX-RPC Web Service.

Web Services and Workbench

SilverStream eXtend Workbench is a J2EE-oriented IDE that providers can use to create, deploy, and maintain Web Services based on the **JAX-RPC** standard. That means Web Services are packaged in J2EE **Web archives** (WARs) that can be deployed to any J2EE-compatible server. Workbench can also be used to develop Java-based Web Service consumers that comply with JAX-RPC.

To help you implement Web Services and Web Service consumers, Workbench provides these facilities:


Facility	Description
jBroker Web	Core technologies for eXtend Web Service support, including compilers and SOAP runtime based on JAX-RPC
Web Service Wizard	Tool that helps you invoke the jBroker Web compilers to generate Java classes and WSDL files for Web Services and Web Service consumers
Registry Manager	Tool for querying and publishing to Web Service registries
WSDL Wizard and Editor	Tools for creating and editing WSDL files

jBroker Web

jBroker Web is a JAX-RPC implementation that includes **compilers** and a **runtime environment** for developing and executing Web Service provider and consumer applications.

The Web Service Wizard uses the jBroker Web compilers to create Web Service components (skeletons, ties, stubs) and WSDL files. Developers can also invoke these compilers separately from the command line.

Both provider and consumer deploy **jbroker-web.jar** (and some supporting JARs) with their applications to provide the necessary runtime environment. This includes the SOAP engine that runs when stub and skeleton components pass SOAP messages between consumer and provider applications.


 For more information, see the jBroker Web help.

Web Service Wizard

The Web Service Wizard enables you to create Web Service components from Java classes or WSDL files. It generates the Java **remote interface** for accessing an object as well as **skeleton**, **tie**, and **stub** Java classes that handle SOAP message communication between a consumer application and a Web Service. The generated code is based on JAX-RPC.

The provider deploys a Web Service as a Web archive (WAR) in which the skeleton and tie classes implement a **servlet** that processes incoming SOAP messages. A consumer application accesses Web Service functionality by calling methods in the stub class, which sends SOAP messages to the server.



 For more information, see:

- Chapter 9, “Generating Web Services”
- Chapter 10, “Generating Web Service Consumers”
- Web Service Wizard chapter in the *Tools Guide*

Registry Manager

The Registry Manager helps providers publish to Web Service registries. It helps consumers query Web Service registries.



For more information, see the Registry Manager chapter in the *Tools Guide*.

WSDL Wizard and Editor

The WSDL Wizard helps providers create new WSDL documents. The WSDL Editor helps providers edit and use existing WSDL documents.




For more information, see the WSDL Editor chapter in the *Tools Guide*.

9

Generating Web Services

This chapter walks you through the basic steps and typical scenarios for using the **Web Service Wizard** to generate Web Services from a variety of sources. Topics include:


- Basics
- Steps
- Choosing an implementation model
- Scenario: starting with a Java class


 To learn about the steps and scenarios for using the wizard when you want a program to access Web Services, see Chapter 10, “Generating Web Service Consumers”.

Basics

You can use the Web Service Wizard of Workbench to develop **standard (SOAP-based) Web Services** that are implemented as **Java remote objects** (using RMI). The wizard generates Java source files based on JAX-RPC (Java API for XML-based RPC) and jBroker Web (the JAX-RPC implementation included with SilverStream eXtend). JAX-RPC is the J2EE specification that provides Web Service support.

The generated files include a **servlet** to handle access to your Web Service and its methods from HTTP SOAP requests. You can use the generated files as is or modify them when necessary. The advantage of this Java-oriented approach is that you can deal with Web Services using the familiar technologies of RMI and J2EE instead of coding lower-level SOAP APIs.

 For an introduction to Web Service concepts, standards, and technologies, see Chapter 8, “Understanding Web Services”.

 For detailed documentation on the wizard, see the Web Service Wizard chapter in the *Tools Guide*.

Steps

The complete development process involves:

1. Preparing to generate
2. Generating Web Service files
3. Examining the generated files
4. Editing the generated files
5. Using the generated files

Preparing to generate

To prepare for using the Web Service Wizard, you:

1. Set up a **WAR project** in Workbench.


For each Web Service you generate, the wizard creates a **servlet** to handle access to that Web Service (from HTTP SOAP requests). As a result, a **WAR** is required to package your Web Services (one or more per WAR) for deployment to a J2EE server where they will run.

A possible variation is to set up a **JAR subproject** in your WAR and use that JAR to contain the servlet and other classes for a Web Service. In any case, the **servlet mapping** will be in the WAR's deployment descriptor (web.xml).

(Note that the approach of using a JAR subproject is not currently supported by the Web Service Wizard when you generate a Web Service from a WSDL file. In this situation, it only supports a WAR project.)

2. **Add these files** to the project:

Files	Details
Source files, classes, or archives from which your Web Services are to be generated	<p>You can generate a Web Service from any one of the following:</p> <ul style="list-style-type: none">• A JavaBean or other Java class• An EJB session bean• A Java remote interface• A WSDL file <p>No matter which one you provide, it should (at minimum) declare the methods you want your generated Web Service to expose.</p> <p>Compile your Java files If you provide any Java files, make sure you compile them in your project before starting the Web Service Wizard (because the wizard works from compiled classes).</p> <p>Edit your WSDL bindings If you provide any WSDL files, edit them as needed to make sure the SOAP address in the service definition specifies the correct binding URL. The Web Service Wizard will use this URL in the files it generates for your Web Service.</p>

Files	Details
<p>Archives required by jBroker Web:</p> <ul style="list-style-type: none"> • jbroker-web.jar, which contains the jBroker Web API classes needed at runtime • jaxrpc-api.jar and saaj-api.jar, which contain the Java API classes for XML-based RPC and SOAP processing • xerces.jar or another XML parser 	<p>You'll find these JARs in the Workbench compilelib directory. Depending on your J2EE server configuration, you should do one of the following:</p> <ul style="list-style-type: none"> • Add them to the WEB-INF/lib directory of your WAR project • Add them to the server classpath of your J2EE server <p> For more information, see the chapter on archive deployment in the <i>Tools Guide</i>.</p>

3. Edit the **classpath** of your project so you can compile your Web Service classes once they're generated and edited. You'll need to include:
 - j2ee_api_1_n.jar (automatically added when you create a WAR project)
 - jbroker-web.jar
 - jaxrpc-api.jar and saaj-api.jar
 - xerces.jar (or another XML parser)
 - Any application-specific entries (such as an EJB-client JAR file you've provided for a session bean Web Service)

If you use **SOAP message handlers** (an advanced JAX-RPC feature) in your application, the project will also require the following archives: activation.jar, commons-logging.jar, dom4j.jar, jaxp-api.jar, and saaj-ri.jar. You'll find these JARs in the Workbench compilelib directory.

Generating Web Service files

Once you've set up your WAR project, you're ready to use the Web Service Wizard. The wizard produces one Web Service at a time, so you'll need to use it multiple times if you have several to develop.

Each time you launch the wizard, it takes input from you about the kind of Web Service to produce. It then generates a set of source files that together make up the Web Service. Here's a summary of the process:

1. Select **File>New** to display the New File dialog and go to the **Web Services** tab.

2. Launch the Web Service Wizard by doing one of the following:

To generate a Web Service from	Select
One of these: <ul style="list-style-type: none"> • A JavaBean or other Java class • An EJB session bean • A Java remote interface 	New Web Service
A WSDL file	Existing Web Service As its name suggests, this item is mainly used to generate Web Service consumers that access deployed Web Services (based on their WSDL files). But it can also be used to read WSDL files as blueprints and generate the matching Web Services themselves.

3. When the wizard prompts you for **project location** information, specify:
- The **WAR or JAR project** you set up to contain the generated Web Service files (if you're generating from a WSDL file, the wizard currently requires you to specify a WAR project here)
 - The target **directory and package** in that project (if you're generating from a Java class, you won't have to fill in some of these settings because the wizard will automatically handle them for you)

If you specify a JAR project to contain the generated Web Service files, the wizard will also ask you for a WAR project to map the Web Service's servlet.

- When the wizard prompts you, select the **class or WSDL file** to generate the Web Service from.

The wizard then asks for additional information based on your selection:

If you select	The wizard prompts you to specify
A JavaBean or other Java class	<ul style="list-style-type: none"> Which methods to expose in the generated Web Service (in contrast, when you generate from an EJB, remote interface, or WSDL file, all methods are automatically exposed) Class-generation and SOAP options
The home interface of an EJB session bean	<ul style="list-style-type: none"> Lookup information for the EJB Class-generation and SOAP options
The remote interface of an EJB session bean or the SessionBean class itself	<ul style="list-style-type: none"> The home interface of the EJB session bean Lookup information for the EJB Class-generation and SOAP options
A Java remote interface	<ul style="list-style-type: none"> Class-generation and SOAP options
A WSDL file	<ul style="list-style-type: none"> Class-generation and SOAP options

- When the wizard prompts you for **class-generation and SOAP options**, you need to choose and configure the set of source files to generate for your Web Service.

The most important choice is whether to generate **skeletons** to be **tie-based** or not. The answer depends on the architectural model you want the implementation of your Web Service to follow. See “Choosing an implementation model” on page 135.

You can choose to generate **stubs** (which come with a simple client application) for testing your Web Service. When generating from a Java class, you can also request a **WSDL file** (for publishing the Web Service to a registry) as well as specify the **binding style** (document or RPC) and **service address** (URL) for the Web Service. When generating from a WSDL file, you can specify how **complex types** are to be mapped.

NOTE Support for jBroker Web 1.x applications is available via a **backward-compatibility** option. For more information, see “If you choose jBroker Web 1.x compatibility” on page 128.

- Click **Finish** when you’re done specifying options for the Web Service.

Examining the generated files

Once you finish the wizard, it generates everything you've specified for your Web Service and updates other parts of your project with supporting changes:

What the wizard generates	Details
Java source file for remote interface	<p>xxxWS.java This file is automatically generated whenever your input to the wizard is not a remote interface (such as when you start from a JavaBean, Java class, EJB session bean, or WSDL file). That's because a remote interface (which extends <code>java.rmi.Remote</code> and declares the methods to expose) is required to construct your Web Service.</p> <p>When you start from a WSDL file, the name of the generated remote interface is simply <code>xxx.java</code>.</p>
Java source file for skeletons	<p>xxx_ServiceSkeleton.java Abstract servlet class that handles access to the Web Service (from HTTP SOAP requests).</p> <p>In the tie model, <code>xxx_ServiceTieSkeleton</code> extends this class. In the skeleton model, you extend it yourself (with an implementation of your remote interface).</p>

What the wizard generates	Details
Java source files for tie-based skeletons	<p>xxx_ServiceTieSkeleton.java Abstract servlet class that extends <code>xxx_ServiceSkeleton</code>.</p> <p>xxxTie.java Servlet that's used in the tie model as the front end for the Web Service. It extends <code>xxx_ServiceTieSkeleton</code> to handle access to the Web Service (from HTTP SOAP requests). It delegates to one of the following to process method calls for the Web Service:</p> <ul style="list-style-type: none">• If you start with a JavaBean, Java class, or EJB session bean, <code>xxxTie</code> instantiates <code>xxxDelegate</code> and delegates to it.• If you start with a Java remote interface or WSDL file, you must edit the <code>xxxTie.java</code> file to specify a class of your own to instantiate and delegate to. <p>xxxDelegate.java This file is generated if you start with a JavaBean, Java class, or EJB session bean that implements the methods for your Web Service. <code>xxxDelegate</code> instantiates that implementation class and calls those methods on it.</p> <p>With an EJB session bean, <code>xxxDelegate</code> does a lookup and create to get the remote interface object. Then it uses that object to make the method calls.</p>

What the wizard generates	Details
Java source files for stubs	<p>xxxService.java Service interface used by JAX-RPC clients to obtain the stub for the target Web Service.</p> <p>xxxServiceImpl.java Service implementation class that handles instantiation of the stub (<i>xxx_Stub</i>). It also supports alternative ways of accessing the target Web Service, including dynamic (stubless) calls.</p> <p>(Note that, when you start from a WSDL file, the names generated for the service interface and implementation class depend on your WSDL and may omit the text Service.)</p> <p>xxx_Stub.java Facilitates method calls from a Java-based consumer to the target Web Service. <i>xxx_Stub</i> implements the remote interface corresponding to the Web Service by sending an appropriate HTTP SOAP request for each method call.</p> <p>xxxClient.java Simple client application that works as a consumer of the target Web Service. It obtains the stub (via the Service object) then uses the stub to call Web Service methods.</p> <p>You can run <i>xxxClient</i> from Workbench (select Project>Run Web Service Client Class) or from a command line.</p>
WSDL file	<p>xxx.wsdl For use when publishing your Web Service to a registry. It describes the Web Service in a standard format.</p>
Updates to deployment descriptor	<p>In the tie model (when you generate tie-based skeletons), the wizard updates your WAR project's web.xml file to declare <i>xxxTie</i> as the servlet to handle HTTP SOAP requests for your Web Service.</p> <p>In the skeleton model, you must edit web.xml yourself to declare the servlet to use (your class that extends <i>xxx_ServiceSkeleton</i>).</p>

What the wizard generates	Details
Updates to project contents	The wizard updates your project to add generated files (and other application-specific files) to it.
Updates to project classpath	The wizard updates your project classpath to include application-specific files as needed.

About generated file names


When generating file names, the Web Service Wizard follows the naming rules specified by JAX-RPC. If you start with a Java class, the resulting file names are based on the name of that class. If you start with WSDL, the resulting file names are based on the definitions in that WSDL.

For simplicity, this documentation uses *xxx* to represent the portion of a generated Web Service file name that's derived from a class name or WSDL definition.

Additional details of generation

Under the covers, the Web Service Wizard uses the **jBroker Web compilers** when generating the Web Service files listed above. In some cases, these compilers may generate additional code or files to support requirements specific to your application, such as:

- Type mapping
- Faults
- Multiple portType definitions

 For more information, see the jBroker Web help.

If you choose jBroker Web 1.x compatibility

The current version of jBroker Web provides a high degree of backward-compatibility with earlier versions. However, some changes introduced to support the JAX-RPC standard may require you to modify code when upgrading an application that originated in jBroker Web 1.x. These changes involve the conventions used for:

- **File names** JAX-RPC specifies rules for naming certain Web Service files. In order to follow these rules while keeping all generated names simple and consistent, new name patterns were adopted (for details, see Generated 1.x-compatible files below).
- **Stub access in client code** With JAX-RPC, clients use a Service object to instantiate the stub instead of looking up the stub directly via JNDI.

Although it's recommended that you upgrade to the current jBroker Web and JAX-RPC conventions, it's not required. By using the **jBroker Web 1.x compatibility** option in the Web Service Wizard, you can generate Web Service files according to the original jBroker Web conventions for file names and stub access. This enables you to take advantage of all the other improvements in the latest version of jBroker Web without altering your existing 1.x applications.

Generated 1.x-compatible files The following table describes the files generated when you use the jBroker Web 1.x compatibility option:

With 1.x compatibility on, you get	With 1.x compatibility off, this is named	Details
<code>xxx_REMOTE.java</code> Example: <code>MyObject_REMOTE.java</code>	<code>xxxWS.java</code> Example: <code>MyObjectWS.java</code>	Generated remote interface.
<code>_xxx_ServiceSkeleton.java</code> Example: <code>_MyObject_REMOTE_ServiceSkeleton.java</code>	<code>xxx_ServiceSkeleton.java</code> Example: <code>MyObjectWS_ServiceSkeleton.java</code>	Abstract servlet class.
<code>_xxx_ServiceTieSkeleton.java</code> Example: <code>_MyObject_REMOTE_ServiceTieSkeleton.java</code>	<code>xxx_ServiceTieSkeleton.java</code> Example: <code>MyObjectWS_ServiceTieSkeleton.java</code>	Abstract tie servlet class.
<code>xxx_TIE.java</code> Example: <code>MyObject_TIE.java</code>	<code>xxxTie.java</code> Example: <code>MyObjectWSTie.java</code>	Servlet for the Web Service (in the tie model).
<code>xxx_SERVICE.java</code> Example: <code>MyObject_SERVICE.java</code>	<code>xxxDelegate.java</code> Example: <code>MyObjectWSDelegate.java</code>	Delegate class for the tie servlet.

With 1.x compatibility on, you get	With 1.x compatibility off, this is named	Details
<p><code>xxxService.java</code></p> <p>Example:</p> <pre>MyObjectREMOTEService.java</pre>	<p><code>xxxService.java</code></p> <p>Example:</p> <pre>MyObjectWSService.java</pre>	<p>Service interface for the stub.</p> <p>This class is not used in 1.x-style stub access. It is generated in case you want to upgrade your client code to the JAX-RPC approach.</p>
<p><code>xxxServiceImpl.java</code></p> <p>Example:</p> <pre>MyObjectREMOTEServiceImpl.java</pre>	<p><code>xxxServiceImpl.java</code></p> <p>Example:</p> <pre>MyObjectWSServiceImpl.java</pre>	<p>Service implementation class for the stub.</p> <p>This class is not used in 1.x-style stub access. It is generated in case you want to upgrade your client code to the JAX-RPC approach.</p>
<p><code>_xxx_ServiceStub.java</code></p> <p>Example:</p> <pre>_MyObject_REMOTE_ServiceStub.java</pre>	<p><code>xxx_Stub.java</code></p> <p>Example:</p> <pre>MyObjectWS_Stub.java</pre>	<p>Stub for the Web Service.</p>
<p><code>xxx_CLIENT.java</code></p> <p>Example:</p> <pre>MyObject_CLIENT.java</pre>	<p><code>xxxClient.java</code></p> <p>Example:</p> <pre>MyObjectWSClient.java</pre>	<p>Client application for consuming the Web Service.</p> <p>The 1.x-compatible client obtains the stub directly via a JNDI lookup. In contrast, the JAX-RPC client obtains the stub indirectly via the Service object.</p>
<p><code>xxx.wsdl</code></p> <p>Example:</p> <pre>MyObject_REMOTE.wsdl</pre>	<p><code>xxx.wsdl</code></p> <p>Example:</p> <pre>MyObjectWS.wsdl</pre>	<p>WSDL file for the Web Service.</p>

Editing the generated files

Follow these guidelines when editing the files generated by the Web Service Wizard:

Guideline	Details
File you may need to edit	<ul style="list-style-type: none"> • <code>xxxTie.java</code> See “Editing the <code>xxxTie.java</code> file” on page 131.
File you must edit	<ul style="list-style-type: none"> • <code>xxxClient.java</code> See “Editing the <code>xxxClient.java</code> file” on page 132.
Files you should not edit	<ul style="list-style-type: none"> • <code>xxx_ServiceSkeleton.java</code> • <code>xxx_ServiceTieSkeleton.java</code> • <code>xxxService.java</code> • <code>xxxServiceImpl.java</code> • <code>xxx_Stub.java</code>

It’s OK to edit any of the other generated files, but not typically required.

In some cases, completing the implementation of your Web Service may require you to add one or more manually coded files to work with the generated ones. See “Creating additional files” on page 133.

Editing the `xxxTie.java` file

The generated `xxxTie.java` file includes a couple of methods you may need to edit.

init() method If you start with a JavaBean or Java class, `init()` is generated to call the `setTarget()` method of `xxx_ServiceTieSkeleton` and pass an instance of `xxxDelegate` (to delegate to it). If `xxxDelegate` provides an empty constructor, the generated code uses that constructor to do the instantiation.

But if no implicit or explicit empty constructor is available, you must modify the code to indicate which one to use. You may also want to modify it to use a constructor that expects an argument.

The wizard automatically generates calls to `setTarget()` for every public constructor of `xxxDelegate`. Each line is commented out, except the one that uses the empty constructor (if available). Uncomment the line with the constructor you want and make any related changes:

```
//super.setTarget( new MyObjectWSDelegate( java.lang.String arg0) );
//super.setTarget( new MyObjectWSDelegate( java.lang.String arg0, java.lang.String arg1)
);
super.setTarget( new MyObjectWSDelegate( ) );
```

If you start with a Java remote interface or WSDL file, `init()` is always generated with the `setTarget()` call commented out. In this case, you must provide a class of your own to instantiate and delegate to:

```
//super.setTarget( new CONSTRUCT_YOUR_SERVICE_OBJECT_HERE);
```

If you start with an EJB session bean, you shouldn't need to edit the generated `init()` method.

doGet() method This method is generated to handle HTTP GET requests sent to your Web Service. It returns the WSDL file for the Web Service, if available. Otherwise, it notifies the user that GET requests are not supported.

If you want to implement your own HTTP GET behavior, you can customize the `doGet()` code. If you want to use the default SOAP behavior, you can remove this code or comment it out.


Editing the `xxxClient.java` file

Before you can test your Web Service with `xxxClient`, you must edit the generated `xxxClient.java` file to call one or more methods of the Web Service. Look for the **process() method** in this file and you'll find comments listing all of the possible method calls:

```
// System.out.println("Test Result = " + remote.getString());
// System.out.println("Test Result = " + remote.setString(java.lang.String));
// System.out.println("Test Result = " + remote.sayHello());
```

Uncomment the method call(s) you want to test and supply appropriate argument values, as needed:

```
// System.out.println("Test Result = " + remote.getString());
System.out.println("Test Result = " + remote.setString(args[0]);
System.out.println("Test Result = " + remote.sayHello());
```

 For additional changes you may want to make to the generated `xxxClient.java` file, see Chapter 10, “Generating Web Service Consumers”.

Creating additional files

In many scenarios, once the wizard finishes generating, you'll have all of the Java source files you need for your Web Service. But there are cases where you must code additional classes yourself:

In this case	You must add
When using the skeleton model	A class that extends the generated servlet <code>xxx_ServiceSkeleton</code> and implements the remote interface for your Web Service. You'll use this manually coded class as the servlet for the Web Service.
When using the tie model and starting with a Java remote interface or WSDL file	A class that implements the remote interface for your Web Service. You must edit the generated <code>xxxTie.java</code> file to instantiate this manually coded class and delegate to it.

Using the generated files

To use the Web Service files generated by the wizard, you:

1. **Update the deployment descriptor**, if necessary.

When you use the tie model, the wizard automatically updates the WAR project's `web.xml` file with the appropriate servlet mapping for your Web Service. But with the skeleton model, you must edit `web.xml` yourself to supply this information.

In the following example, `MyService` is the servlet class that the developer has coded for the Web Service `MyRemote`:


```
<servlet>
  <servlet-name>MyService</servlet-name>
  <servlet-class>com.exsamp.rem.MyService</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>MyService</servlet-name>
  <url-pattern>MyRemote</url-pattern>
</servlet-mapping>
```

2. Update the project, if necessary.

As the wizard works, it automatically adds files to your project classpath and contents, as needed. But you should also check yourself to make sure the project has everything it requires to compile and run.

For instance, if your Web Service accesses an EJB session bean, the EJB-client JAR file should be on your project's classpath. The EJB-client JAR file and remote EJB JAR file should be in the project's WEB-INF/lib directory (assuming it's a WAR project).

 For details on setting up the required classpath and contents for your project (including what jBroker Web needs), see "Preparing to generate" on page 120.

3. Build and archive the project.

When you complete this step, you'll have a WAR file containing the Web Service(s) you've generated.

4. Set up for deployment to your J2EE server.

Prepare the server-specific deployment information required to deploy the WAR to your J2EE server. For example, if you're going to deploy to the SilverStream eXtend Application Server, create a SilverStream deployment plan file.

If you're going to deploy from Workbench, you should also set up a server profile for your J2EE server.


5. Deploy the WAR to your J2EE server.

When you complete this step, each Web Service in the WAR will be accessible as a servlet that can respond to standard HTTP SOAP requests for your exposed methods.

6. Test your Web Service(s) running on the J2EE server.

If you've generated, edited, and compiled the `xxxClient` class for a Web Service, you can use it for a quick test of your method calls. To run `xxxClient` from Workbench, select **Project>Run Web Service Client Class**. The Web Service Wizard Client Runner displays, offering you a list of client classes from the current project to choose from.

You can also run `xxxClient` from a command line (providing that you include the appropriate directories and archives on your system classpath).

 For further details on running `xxxClient`, see Chapter 10, "Generating Web Service Consumers".

Choosing an implementation model

There are two basic implementation models you can choose from when developing with the Web Service Wizard. This section explores these choices to help you select the one that's most appropriate for the Web Services you generate:

- Tie model
- Skeleton model

Tie model

Here's an overview of the tie model and when to use it:

Topic	Details
Typical use	The tie model is typically used when you have an implementation class to provide as input to the Web Service Wizard. That might be a JavaBean, Java class, or EJB session bean that already implements the methods you want to expose as a Web Service.
How it works	The tie model uses a delegation approach to hand off method calls from the generated Web Service classes (which handle the HTTP SOAP processing for your Web Service) to your implementation class (which handles the method processing).
Advantages	The tie model enables you to keep your implementation class (business logic) separate from the generated infrastructure classes that support your Web Service. A related benefit is that you can reuse existing implementation classes currently accessible via other protocols.

Topic	Details
How to generate it	<p>When you specify class-generation and SOAP options in the Web Service Wizard, check both of these items:</p> <ul style="list-style-type: none"> • Generate skeletons • Tie-based
Files generated	<p>If you start with a JavaBean, Java class, or EJB session bean, the wizard generates:</p> <ul style="list-style-type: none"> • <code>xxxWS.java</code> (remote interface) • <code>xxxDelegate.java</code> • <code>xxxTie.java</code> • <code>xxx_ServiceTieSkeleton.java</code> • <code>xxx_ServiceSkeleton.java</code>

It's possible (but not as common) to use the tie model when you have only a Java remote interface or WSDL file to provide as input to the Web Service Wizard. In this case, the wizard output leaves the delegation part of the model for you to complete later. You'll then need to code an implementation class and edit the generated tie class to instantiate it and delegate to it.

Skeleton model

Here's an overview of the skeleton model and when to use it:

Topic	Details
Typical use	<p>The skeleton model is typically used when you know the methods you want to expose as a Web Service, but don't yet have an implementation of them. In this case, you tell the Web Service Wizard about these methods by providing a Java remote interface or WSDL file as input, then implement them later in the context of the generated Web Service files.</p>
How it works	<p>In the skeleton model, you implement your Web Service methods by subclassing the servlet that the wizard generates to handle HTTP SOAP processing. As a result, the same class that supports the logistics of your Web Service also processes the method calls.</p>

Topic	Details
Advantages	The skeleton model is relatively simple, involving fewer classes to understand and maintain. At runtime, having less object overhead may also offer performance benefits.
How to generate it	When you specify class-generation and SOAP options in the Web Service Wizard, check both of these items: <ul style="list-style-type: none"> • Generate skeletons • Not tie-based
Files generated	If you start with a Java remote interface , the wizard generates: <ul style="list-style-type: none"> • <code>xxx_ServiceSkeleton.java</code> If you start with a WSDL file , the wizard generates: <ul style="list-style-type: none"> • <code>xxx.java</code> (remote interface) • <code>xxx_ServiceSkeleton.java</code>
File you add	Once the wizard is done, you must code a class that extends the generated servlet <code>xxx_ServiceSkeleton</code> and implements the remote interface for your Web Service. You'll use this manually coded class as the servlet for the Web Service.

Scenario: starting with a Java class

In this scenario, you'll see how the Web Service Wizard can be used to generate a Web Service based on an existing Java class that implements the methods to expose:

- Project setup
- Input to the wizard
- Generated files for the Web Service
- Generated files for testing
- Deployment descriptor
- Runtime test result

Implementation model This scenario illustrates use of the **tie** model. For an overview of that architecture, see "Choosing an implementation model" on page 135.

Project setup

The WAR project for this scenario is set up as follows:

- The **name** of this project is:
`WebServiceSample.spf`
- The **archive** resulting from this project will be:
`WebServiceSample.war`
- The **initial content** of this project is:

```
WEB-INF
lib
  jbroker-web.jar
  jaxrpc-api.jar
  saaj-api.jar
  xerces.jar
classes
  com
    exsamp
      obj
        MyObject.java
web.xml
```
- The **classpath** needed for this project is:

```
... \WEB-INF\lib\jbroke- web.jar
... \WEB-INF\lib\jaxrpc-api.jar
... \WEB-INF\lib\saaj-api.jar
... \WEB-INF\lib\xerces.jar
... \eXtendWorkbench\compilelib\j2ee_api_1_n.jar
```

Input to the wizard

Here's the input provided to the Web Service Wizard for this scenario:

- MyObject class
- Project location panel
- Class selection panel
- Method selection panel
- Class-generation and SOAP options panel

MyObject class

MyObject is an existing Java class from which the Web Service is to be generated. It implements the methods to expose. MyObject.java contains the following code (which must be compiled before you start the wizard):

```
package com.exsamp.obj;

public class MyObject {

    private String s;

    public MyObject() {
    }

    public MyObject(String xxx) {
    }

    public MyObject(String xxx, String yyy) {
    }

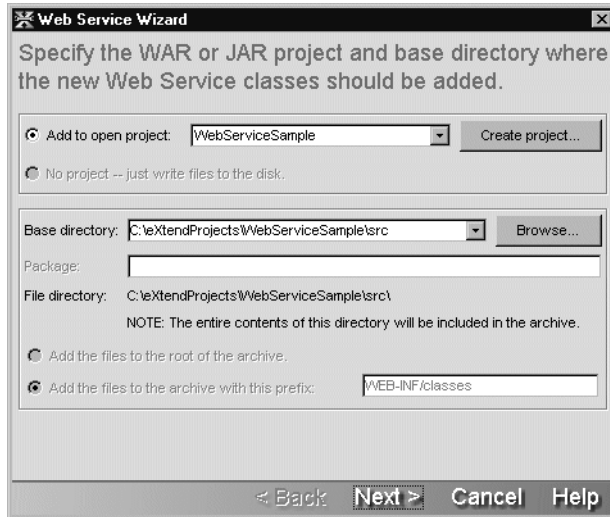
    public String getString() {
        return s;
    }

    public boolean setString(String s) {
        this.s = s;
        return true;
    }

    public String sayHello() {
        return "Hello there, I am on the server";
    }
}
```

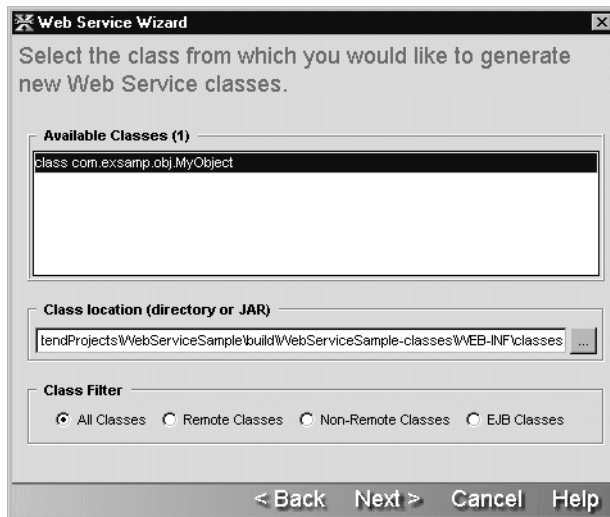
Project location panel

This wizard panel is completed as follows:



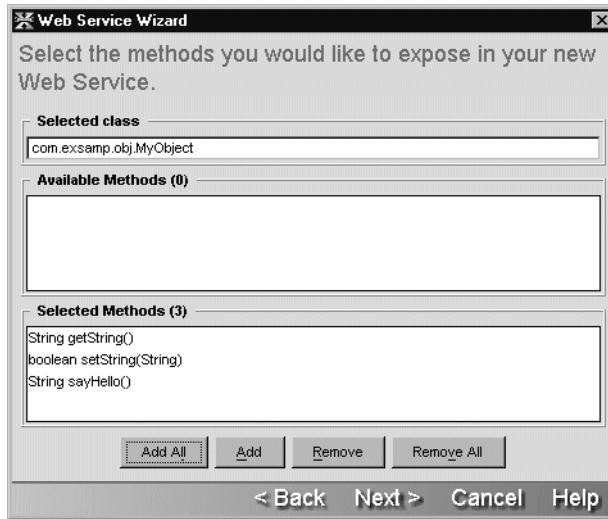
Class selection panel

This wizard panel is completed as follows:



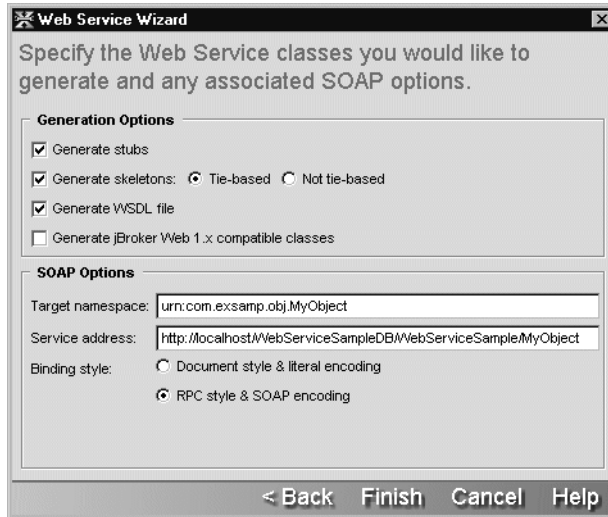
Method selection panel

This wizard panel is completed as follows:



Class-generation and SOAP options panel

This wizard panel is completed as follows:



Generated files for the Web Service

Based on the input provided for this scenario, the Web Service Wizard generates these files to implement the Web Service:

- MyObjectWS.java
- MyObjectWS_ServiceSkeleton.java
- MyObjectWS_ServiceTieSkeleton.java
- MyObjectWSTie.java
- MyObjectWSDelegate.java
- MyObjectWS.wsdl

MyObjectWS.java

MyObjectWS is the remote interface for the Web Service. The wizard generates this source code for it:

```
// The following code was generated within the SilverStream extend Workbench
// using the integrated Web Services Wizard. This code can be freely modified
// and in some cases will *require* modifications to execute as expected.
```



```
// Please keep in mind when making modifications that method signatures  
// must be consistent across all generated objects.
```

```
package com.exsamp.obj;  
  
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface MyObjectWS extends Remote  
{  
    public java.lang.String getString( )  
        throws RemoteException;  
  
    public boolean setString( java.lang.String arg0 )  
        throws RemoteException;  
  
    public java.lang.String sayHello( )  
        throws RemoteException;  
}
```

MyObjectWS_ServiceSkeleton.java

MyObjectWS_ServiceSkeleton is the abstract servlet class that handles access to the Web Service. The wizard generates this source code for it:

```
// Fri May 31 10:15:21 EDT 2002  
  
package com.exsamp.obj;  
  
import java.rmi.RemoteException;  
import java.util.Properties;  
import com.sssw.jbroker.web.encoding.TypeMappingRegistry;  
import com.sssw.jbroker.web.encoding.DefaultTypeMappingRegistry;  
  
public abstract class MyObjectWS_ServiceSkeleton  
    extends com.sssw.jbroker.web.portable.ServletSkeleton  
    implements MyObjectWS  
{  
    private static final com.sssw.jbroker.web.QName _portType =  
        new com.sssw.jbroker.web.QName("urn:com.exsamp.obj.MyObject", "MyObjectWS");  
  
    public MyObjectWS_ServiceSkeleton()  
    {  
        super(_portType);  
        _setProperty("xmlrpc.schema.uri", "http://www.w3.org/2001/XMLSchema");  
        _setProperty("version", "1.1");  
    }  
}
```

```
private static java.util.Dictionary _atable = new java.util.Hashtable();
static {
    _atable.put("\urn:com.exsamp.obj.MyObject/setString\"", new
java.lang.Integer(0));
    _atable.put("\urn:com.exsamp.obj.MyObject/getString\"", new
java.lang.Integer(1));
    _atable.put("\urn:com.exsamp.obj.MyObject/sayHello\"", new java.lang.Integer(2));
}

private static java.util.Dictionary _mtable = new java.util.Hashtable();
static {
    _mtable.put("setString", new java.lang.Integer(0));
    _mtable.put("getString", new java.lang.Integer(1));
    _mtable.put("sayHello", new java.lang.Integer(2));
}

public com.sssw.jbroker.web.portable.ServerResponse
_invoke(com.sssw.jbroker.web.portable.ServerRequest in) throws java.io.IOException
{
    com.sssw.jbroker.web.portable.ServerResponse out = null;
    String soapEncURI = "soap";
    String literalURI = "literal";

    try {

        java.lang.Integer _m = null;
        String sac = in.getAction();
        if (sac != null) _m = (java.lang.Integer) _atable.get(sac);

        if (_m == null) {
            sac = "\"" + sac + "\"";
            _m = (java.lang.Integer) _atable.get(sac);
        }

        if (_m == null) {
            String methodName = in.getMethod();
            if (methodName != null) _m = (java.lang.Integer) _mtable.get(methodName);
        }

        if (_m == null) throw new
            com.sssw.jbroker.web.ServiceException("unable to dispatch SOAP request");

        switch(_m.intValue()) {

            // setString
            case 0: {
                in.setEncodingStyleURI(soapEncURI);
                java.lang.String _arg0 = null;
                try {
```

```
        _arg0 = (java.lang.String)
            in.readObject(java.lang.String.class, "arg0");
    } catch (java.io.EOFException eofExc) {
        _arg0 = null;
    }
    boolean result = setString(_arg0);
    //create reply
    out = in.createReply();
    //set the content type
    java.lang.Object arg = null;
    arg = new java.lang.Boolean(result);
    out.writeObject(arg, "result");
    break;
}

// getString
case 1: {
    in.setEncodingStyleURI(soapEncURI);
    java.lang.String result = getString();
    //create reply
    out = in.createReply();
    //set the content type
    java.lang.Object arg = null;
    arg = result;
    out.writeObject(arg, "result");
    break;
}

// sayHello
case 2: {
    in.setEncodingStyleURI(soapEncURI);
    java.lang.String result = sayHello();
    //create reply
    out = in.createReply();
    //set the content type
    java.lang.Object arg = null;
    arg = result;
    out.writeObject(arg, "result");
    break;
}
}

} catch (java.lang.Throwable ex) {
    if (System.getProperty("SOAP_DEBUG") != null) ex.printStackTrace();
    out = in.createExceptionReply();
    out.writeException(ex, "exception");
}

return out;
```

```
    }

    public boolean isDocument(String action)
    {
        return false;
    }

    private static Properties _rootHeaders = new Properties();
    static {
        _rootHeaders.setProperty("content-type", "text/xml; charset=UTF-8");
        _rootHeaders.setProperty("content-id", "<soapbody>");
    }
}
```

MyObjectWS_ServiceTieSkeleton.java

MyObjectWS_ServiceTieSkeleton is an abstract class that extends MyObjectWS_ServiceSkeleton to support the tie model. The wizard generates this source code for it:

```
// Fri May 31 10:15:21 EDT 2002

package com.exsamp.obj;

import java.rmi.RemoteException;
import java.util.Properties;
import com.sssw.jbroker.web.encoding.TypeMappingRegistry;
import com.sssw.jbroker.web.encoding.DefaultTypeMappingRegistry;

public abstract class MyObjectWS_ServiceTieSkeleton
    extends com.exsamp.obj.MyObjectWS_ServiceSkeleton
    implements com.sssw.jbroker.web.portable.TieSkeleton
{
    private MyObjectWS _target;

    public void setTarget(java.rmi.Remote target)
    {
        _target = (MyObjectWS) target;
    }

    public java.rmi.Remote getTarget()
    {
        return _target;
    }

    public boolean setString(java.lang.String _arg0)
        throws java.rmi.RemoteException
```

```
{
    return _target.setString(_arg0);
}

public java.lang.String getString()
    throws java.rmi.RemoteException
{
    return _target.getString();
}

public java.lang.String sayHello()
    throws java.rmi.RemoteException
{
    return _target.sayHello();
}
}
```

MyObjectWSTie.java

MyObjectWSTie extends the abstract servlet classes to function as the front end for the Web Service. To process requests (method calls) it receives, this servlet instantiates and delegates to MyObjectWSDelegate. The wizard generates this source code for it:

```
// The following code was generated within the SilverStream eXtend Workbench
// using the integrated Web Services Wizard. This code can be freely modified
// and in some cases will *require* modifications to execute as expected.
// Please keep in mind when making modifications that method signatures
// must be consistent across all generated objects.

package com.exsamp.obj;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class MyObjectWSTie extends MyObjectWS_ServiceTieSkeleton
{
    public void init() throws ServletException
    {
        try
        {
            super.init();

            // The following are all public constructors for the implemented service
            // class. IMPORTANT NOTE: If available, the empty constructor has been
            // implemented by default. If no implicit or explicit empty constructor
            // is available, you *must* select one from the list below and uncomment

```

```
        // it in order to construct the generated service implementation.

        //super.setTarget( new MyObjectWSDelegate( java.lang.String arg0 ) );
        //super.setTarget( new MyObjectWSDelegate( java.lang.String arg0,
java.lang.String arg1 ) );
        super.setTarget( new MyObjectWSDelegate( ) );
    }
    catch (Exception _e)
    {
        throw new ServletException(_e);
    }
}

// The following method may be freely modified to provide custom behavior
// when an HTTP GET request is made. Comment-out or remove this method to
// provide default SOAP doGet functionality.
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    try
    {
        StringBuffer sb = new StringBuffer(1024);
        OutputStream out = null;
        InputStream in = null;
        String path = "/MyObjectWS.wsdl";

        try
        {
            // Try to load the WSDL file.
            in = getServletConfig().getServletContext().getResourceAsStream(path);
            if (in == null)
            {
                // If it can't be found, return a default message.
                sendDefaultMsg(response);
            }
            else
            {
                // Try to determine the WSDL file's character encoding for content-
type.

                byte[] buf = new byte[512];
                int read = in.read(buf);

                if (read <= 0)
                    sendDefaultMsg(response);

                String cs = getXMLEncoding(buf);
                StringBuffer ct = new StringBuffer(64);
                ct.append("text/xml");
                if (cs != null)
```

```

        {
            ct.append("; charset=");
            ct.append(cs);
        }

        // Return the WSDL file.
        response.setContentType(ct.toString());
        out = response.getOutputStream();
        do
        {
            out.write(buf, 0, read);
        } while ((read = in.read(buf)) >= 0);
    }
}
catch (Exception _e)
{
    throw new ServletException("Exception trying to return " + path, _e);
}
finally
{
    if (out != null)
        out.close();
    if (in != null)
        in.close();
}
}
catch (Exception _e)
{
    throw new ServletException(_e);
}
}

// Try to determine the character encoding of this XML document.
public static String getXMLEncoding(byte[] bytes)
{
    String lsLine = "";
    String lsEncoding = "UTF-8";

    if (bytes.length >= 2 && bytes[0] == 0xFE && bytes[1] == 0xFF)
        return "UTF-16";
    String lsState = "";
    int liDeclStart = 0;
    int liDeclLength = 0;

    for (int i=0; i < bytes.length; i++)
    {
        if (lsState.equals("") && bytes[i] == '<' && bytes[i+1] == '?')

```

```
    {
        lsState = "<?";
    }
    else
    {
        if (lsState.equals("<?") && bytes[i] == 'x' && bytes[i+1] == 'm'
            && bytes[i+2] == 'l' && bytes[i+3] == ' ')
        {
            liDeclStart = i;
            lsState = "xml";
        }
        else
        {
            if (lsState.equals("xml") && bytes[i] == '?' && bytes[i+1] == '>')
            {
                liDeclLength = i - liDeclStart;
                break;
            }
        }
    }
}

lsLine = new String(bytes, liDeclStart, liDeclLength);

int liPos = lsLine.indexOf("encoding");
if (liPos > 0)
{
    lsLine = lsLine.substring(liPos + 8);
    int liEncStart = lsLine.indexOf('"');
    int liEncEnd = lsLine.indexOf('"', liEncStart + 1);
    if (liEncStart < 0 && liEncEnd < 0)
    {
        liEncStart = lsLine.indexOf('"');
        liEncEnd = lsLine.indexOf('"', liEncStart + 1);
    }

    if (liEncStart >= 0 && liEncEnd >= 0)
        lsEncoding = lsLine.substring(liEncStart + 1, liEncEnd);
}

return lsEncoding;
}
```

```
static private final String DEFAULT_MESSAGE =
"<html><head><title>SilverStream eXtend Web Service</title>" +
"</head><body><h3 align=\"center\">SilverStream eXtend Web Service</h3>" +
"By default, SOAP servers do not communicate via HTTP GET requests. The SilverStream "
```

+


```

"eXtend Web Service Wizard has generated an overloaded version of the " +
"<i>doGet()</i> method for your convience. This method, found in your " +
"generated _TIE code, is producing this message. If the WSDL file for this Web Service
" +
" is available in the root of your Web Service WAR, this method will return the WSDL
instead " +
"of this default message. You may add any custom code you like in your generated
_TIE's " +
"<i>doGet()</i> method to handle HTTP GET support.</body></html>";

private void sendDefaultMsg(HttpServletResponse response) throws IOException
{
    PrintWriter out = null;

    try
    {
        response.setContentType("text/html");
        response.setContentLength(DEFAULT_MESSAGE.length());
        out = response.getWriter();
        out.print(DEFAULT_MESSAGE);
    }
    finally
    {
        if (out != null) out.close();
    }
}
}

```

MyObjectWSDelegate.java

MyObjectWSDelegate instantiates the implementation class (MyObject) and makes the requested method calls against that instance. The wizard generates this source code for it:

```

// The following code was generated within the SilverStream eXtend Workbench
// using the integrated Web Services Wizard. This code can be freely modified
// and in some cases will *require* modifications to execute as expected.
// Please keep in mind when making modifications that method signatures
// must be consistent across all generated objects.

package com.exsamp.obj;

import java.rmi.Remote;
import java.rmi.RemoteException;

public class MyObjectWSDelegate implements MyObjectWS
{
    private MyObject m_objMyObject;

```

```
public MyObjectWSDelegate( java.lang.String arg0 )
{
    m_objMyObject = new MyObject( arg0 );
}

public MyObjectWSDelegate( java.lang.String arg0, java.lang.String arg1 )
{
    m_objMyObject = new MyObject( arg0, arg1 );
}

public MyObjectWSDelegate( )
{
    m_objMyObject = new MyObject( );
}

public java.lang.String getString( )
    throws RemoteException
{
    return m_objMyObject.getString( );
}

public boolean setString( java.lang.String arg0 )
    throws RemoteException
{
    return m_objMyObject.setString( arg0 );
}

public java.lang.String sayHello( )
    throws RemoteException
{
    return m_objMyObject.sayHello( );
}
}
```

MyObjectWS.wsdl

This generated file describes the Web Service in standard WSDL format (useful when publishing to a registry):

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MyObjectWSService"
    targetNamespace="urn:com.exsamp.obj.MyObject"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tns="urn:com.exsamp.obj.MyObject"
    xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types/>
```

```
<message name="setStringRequest">
  <part name="arg0" type="xsd:string"/>
</message>
<message name="setStringResponse">
  <part name="result" type="xsd:boolean"/>
</message>
<message name="getStringRequest"/>
<message name="getStringResponse">
  <part name="result" type="xsd:string"/>
</message>
<message name="sayHelloRequest"/>
<message name="sayHelloResponse">
  <part name="result" type="xsd:string"/>
</message>
<portType name="MyObjectWS">
  <operation name="setString" parameterOrder="arg0">
    <input message="tns:setStringRequest"/>
    <output message="tns:setStringResponse"/>
  </operation>
  <operation name="getString">
    <input message="tns:getStringRequest"/>
    <output message="tns:getStringResponse"/>
  </operation>
  <operation name="sayHello">
    <input message="tns:sayHelloRequest"/>
    <output message="tns:sayHelloResponse"/>
  </operation>
</portType>
<binding name="MyObjectWSBinding" type="tns:MyObjectWS">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="setString">
    <soap:operation soapAction="urn:com.exsamp.obj.MyObject/setString"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:com.exsamp.obj.MyObject" use="encoded"/>
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:com.exsamp.obj.MyObject" use="encoded"/>
    </output>
  </operation>
  <operation name="getString">
    <soap:operation soapAction="urn:com.exsamp.obj.MyObject/getString"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:com.exsamp.obj.MyObject" use="encoded"/>
    </input>
  </operation>
</binding>
</service>
```

```
</input>
<output>
  <soap:body
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:com.exsamp.obj.MyObject" use="encoded"/>
</output>
</operation>
<operation name="sayHello">
  <soap:operation soapAction="urn:com.exsamp.obj.MyObject/sayHello"/>
  <input>
    <soap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:com.exsamp.obj.MyObject" use="encoded"/>
  </input>
  <output>
    <soap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:com.exsamp.obj.MyObject" use="encoded"/>
  </output>
</operation>
</binding>
<service name="MyObjectWSService">
  <port binding="tns:MyObjectWSBinding" name="MyObjectWSPort">
    <soap:address
      location="http://localhost/WebServiceSampleDB/WebServiceSample/MyObject"/>
    </port>
  </service>
</definitions>
```

Generated files for testing

Based on the input provided for this scenario, the Web Service Wizard generates these files so you can test the Web Service once it's deployed:

- MyObjectWSService.java
- MyObjectWSServiceImpl.java
- MyObjectWS_Stub.java
- MyObjectWSClient.java

MyObjectWSService.java

MyObjectWSService is the service interface that's used in JAX-RPC to help clients obtain the stub for the Web Service. The wizard generates this source code for it:

```
// Fri May 31 10:15:21 EDT 2002

package com.exsamp.obj;

import javax.xml.rpc.ServiceException;

public interface MyObjectWSService extends javax.xml.rpc.Service
{
    public MyObjectWS_Stub getMyObjectWSPort()
        throws ServiceException;
}
```

MyObjectWSServiceImpl.java

MyObjectWSServiceImpl is the service implementation class that handles instantiation of the stub (MyObjectWS_Stub). The wizard generates this source code for it:

```
// Fri May 31 10:15:21 EDT 2002

package com.exsamp.obj;

import java.io.FileNotFoundException;
import java.util.Iterator;
import java.util.Hashtable;
import java.util.Properties;
import java.util.ArrayList;
import java.net.URL;
import java.net.MalformedURLException;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceException;
import com.sssw.jbroker.web.Binding;
import com.sssw.jbroker.web.encoding.DefaultTypeMappingRegistry;

public class MyObjectWSServiceImpl
    extends com.sssw.jbroker.web.xml.rpc.ServiceImpl
    implements MyObjectWSService
{
    public MyObjectWSServiceImpl()
    {
        try {
            createCalls();
        }
    }
}
```

```
        } catch (ServiceException ex) {
            throw new javax.xml.rpc.JAXRPCException("failed to create the call objects: "
+ ex.getMessage());
        }
    }

    public QName getServiceName() { return _serviceName; }

    public Iterator getPorts() { return _portMapping.keySet().iterator(); }

    public void setProxyMode(boolean proxy) { _proxy = proxy; }

    public boolean getProxyMode() { return _proxy; }

    public URL getWSDLDocumentLocation()
    {
        return null;
    }

    public java.rmi.Remote getPort(Class serviceDefInterface)
        throws ServiceException
    {
        if (serviceDefInterface == null)
            throw new ServiceException("No Service class specified.");
        if (!java.rmi.Remote.class.isAssignableFrom(serviceDefInterface))
            throw new ServiceException("Class is not a valid Interface.");

        String stubName = (String) _intfMapping.get(serviceDefInterface);
        Binding binding = (Binding) _intfBinding.get(serviceDefInterface);

        if (stubName == null)
            return getPort(serviceDefInterface, binding,
                _classInfo, _typeMappingRegistry, null);
        else
            return getPort(stubName, binding, _typeMappingRegistry);
    }

    public java.rmi.Remote getPort(QName portName, Class serviceDefInterface)
        throws ServiceException
    {
        return getPort(portName, serviceDefInterface, getProxyMode());
    }

    public java.rmi.Remote getPort(QName portName, Class serviceDefInterface, boolean
proxy)
        throws ServiceException
    {
        if (((proxy==false) || (serviceDefInterface == null)) &&
            (portName != null)) {
            String stubName = (String) _portMapping.get(portName);
```

```

        Binding binding = (Binding) _portBinding.get(portName);

        if (stubName == null) return getPort(null, serviceDefInterface);

        try {
            return getPort(stubName, binding, portName,
                _typeMappingRegistry);
        } catch (Exception ex) {
            return getPort(null, serviceDefInterface);
        }
    } else {
        if (serviceDefInterface == null)
            throw new ServiceException("No Service class specified.");
        if (!java.rmi.Remote.class.isAssignableFrom(serviceDefInterface))
            throw new ServiceException("Class is not a valid Interface.");

        Binding binding = (Binding) _intfBinding.get(serviceDefInterface);
        String uri = (portName == null) ? null : portName.getNamespaceURI();
        return getPort(serviceDefInterface, binding, _classInfo,
            _typeMappingRegistry, uri);
    }
}

public Call[] getCalls(QName portName)
    throws ServiceException
{
    ArrayList callslst = (ArrayList) _calls.get(portName);
    if (callslst == null) return null;
    Call[] calls = new Call[callslst.size()];
    return (Call[]) callslst.toArray(calls);
}

private void addCall(QName portName, Call call)
{
    ArrayList callslst = (ArrayList) _calls.get(portName);
    if (callslst == null) {
        callslst = new ArrayList();
        _calls.put(portName, callslst);
    }
    callslst.add(call);
}

public MyObjectWS_Stub getMyObjectWSPort()
    throws ServiceException
{
    try {
        return (MyObjectWS_Stub) getPort(new QName(
            "urn:com.exsamp.obj.MyObject", "com.exsamp.obj.MyObjectWSPort"), null,
false);
    }
}

```

```
    } catch (Exception ex) {
        return (MyObjectWS_Stub) getPort(com.exsamp.obj.MyObjectWS.class);
    }
}

private void createCalls()
    throws ServiceException
{
    Call call = null;

    call = createCall(new QName("urn:com.exsamp.obj.MyObject",
"com.exsamp.obj.MyObjectWSPort"),
        new QName("urn:com.exsamp.obj.MyObject", "setString"));
    call.addParameter("arg0", new QName("http://www.w3.org/2001/XMLSchema", "string"),
java.lang.String.class, ParameterMode.IN);
    call.addParameter("result", new QName("http://www.w3.org/2001/XMLSchema",
"boolean"), boolean.class, ParameterMode.OUT);
    call.setReturnType(new QName("http://www.w3.org/2001/XMLSchema", "boolean"),
boolean.class);
    call.setProperty(Call.OPERATION_STYLE_PROPERTY, "rpc");
    call.setProperty(Call.SOAPACTION_URI_PROPERTY,
"\urn:com.exsamp.obj.MyObject/setString\");

call.setTargetEndpointAddress("http://localhost/WebServiceSampleDB/WebServiceSample/MyObje
ct");
    addCall(new QName("urn:com.exsamp.obj.MyObject", "com.exsamp.obj.MyObjectWSPort"),
call);

    call = createCall(new QName("urn:com.exsamp.obj.MyObject",
"com.exsamp.obj.MyObjectWSPort"),
        new QName("urn:com.exsamp.obj.MyObject", "getString"));
    call.addParameter("result", new QName("http://www.w3.org/2001/XMLSchema",
"string"), java.lang.String.class, ParameterMode.OUT);
    call.setReturnType(new QName("http://www.w3.org/2001/XMLSchema", "string"),
java.lang.String.class);
    call.setProperty(Call.OPERATION_STYLE_PROPERTY, "rpc");
    call.setProperty(Call.SOAPACTION_URI_PROPERTY,
"\urn:com.exsamp.obj.MyObject/getString\");

call.setTargetEndpointAddress("http://localhost/WebServiceSampleDB/WebServiceSample/MyObje
ct");
    addCall(new QName("urn:com.exsamp.obj.MyObject", "com.exsamp.obj.MyObjectWSPort"),
call);

    call = createCall(new QName("urn:com.exsamp.obj.MyObject",
"com.exsamp.obj.MyObjectWSPort"),
        new QName("urn:com.exsamp.obj.MyObject", "sayHello"));
    call.addParameter("result", new QName("http://www.w3.org/2001/XMLSchema",
"string"), java.lang.String.class, ParameterMode.OUT);
```



```

        call.setReturnType(new QName("http://www.w3.org/2001/XMLSchema", "string"),
java.lang.String.class);
        call.setProperty(Call.OPERATION_STYLE_PROPERTY, "rpc");
        call.setProperty(Call.SOAPACTION_URI_PROPERTY,
"\urn:com.exsamp.obj.MyObject/sayHello\");

call.setTargetEndpointAddress("http://localhost/WebServiceSampleDB/WebServiceSample/MyObjec
t");
    addCall(new QName("urn:com.exsamp.obj.MyObject", "com.exsamp.obj.MyObjectWSPort"),
call);

    }

    static boolean _proxy = true;
    static final QName _serviceName;
    static final Hashtable _intfMapping = new Hashtable();
    static final Hashtable _intfBinding = new Hashtable();
    static final Hashtable _portBinding = new Hashtable();
    static final Hashtable _portMapping = new Hashtable();
    static final Hashtable _classInfo = new Hashtable();
    private final Hashtable _calls = new Hashtable();

    static {
        _serviceName = new QName("urn:com.exsamp.obj.MyObject",
"com.exsamp.obj.MyObjectWSService");

        _intfBinding.put(MyObjectWS.class, new Binding("soap",
"http://localhost/WebServiceSampleDB/WebServiceSample/MyObject"));
        _portBinding.put(new QName("urn:com.exsamp.obj.MyObject",
"com.exsamp.obj.MyObjectWSPort"),
            new Binding("soap",
"http://localhost/WebServiceSampleDB/WebServiceSample/MyObject"));
        _intfMapping.put(MyObjectWS.class, "com.exsamp.obj.MyObjectWS_Stub");
        _portMapping.put(new QName("urn:com.exsamp.obj.MyObject",
"com.exsamp.obj.MyObjectWSPort"), "com.exsamp.obj.MyObjectWS_Stub");

        Hashtable _methodInfo;
        Hashtable _paramInfo;
        Properties _props;

        _methodInfo = new Hashtable();
        _paramInfo = new Hashtable();
        _props = new Properties();

        _props.setProperty("jbroker.web.soap.action", "\urn:com.exsamp.obj.MyObject/setString\");
        _paramInfo.put("Properties", _props);
        _props = new Properties();
        _props.setProperty("jbroker.web.parameter.name", "arg0");
        _props.setProperty("jbroker.web.parameter.inout", "1");

```

```
    _paramInfo.put("Param0", _props);
    _props = new Properties();
    _props.setProperty("jbroker.web.parameter.name", "result");
    _props.setProperty("jbroker.web.parameter.inout", "2");
    _paramInfo.put("Result", _props);
    _methodInfo.put("setString", _paramInfo);
    _paramInfo = new Hashtable();
    _props = new Properties();

    _props.setProperty("jbroker.web.soap.action", "\"urn:com.exsamp.obj.MyObject/getString\"");
    _paramInfo.put("Properties", _props);
    _props = new Properties();
    _props.setProperty("jbroker.web.parameter.name", "result");
    _props.setProperty("jbroker.web.parameter.inout", "2");
    _paramInfo.put("Result", _props);
    _methodInfo.put("getString", _paramInfo);
    _paramInfo = new Hashtable();
    _props = new Properties();

    _props.setProperty("jbroker.web.soap.action", "\"urn:com.exsamp.obj.MyObject/sayHello\"");
    _paramInfo.put("Properties", _props);
    _props = new Properties();
    _props.setProperty("jbroker.web.parameter.name", "result");
    _props.setProperty("jbroker.web.parameter.inout", "2");
    _paramInfo.put("Result", _props);
    _methodInfo.put("sayHello", _paramInfo);
    _classInfo.put("com.exsamp.obj.MyObjectWS", _methodInfo);
}
}
```

MyObjectWS_Stub.java

MyObjectWS_Stub is used by clients as a proxy for accessing the Web Service. This stub class implements the remote interface (MyObjectWS) to handle the logistics of each method call. The wizard generates this source code for it:

```
// Fri May 31 10:15:21 EDT 2002

package com.exsamp.obj;

import java.util.Properties;
import com.sssw.jbroker.web.core.Constants;
import com.sssw.jbroker.web.encoding.TypeMappingRegistry;
import com.sssw.jbroker.web.encoding.DefaultTypeMappingRegistry;

public class MyObjectWS_Stub
    extends com.sssw.jbroker.web.portable.Stub
    implements MyObjectWS
```

```

{
    private static com.sssw.jbroker.web.QName _portType =
        new com.sssw.jbroker.web.QName("urn:com.exsamp.obj.MyObject", "MyObjectWS");

    private static final com.sssw.jbroker.web.Binding[] _bindings =
        new com.sssw.jbroker.web.Binding[] {
            new com.sssw.jbroker.web.Binding("soap",
"http://localhost/WebServiceSampleDB/WebServiceSample/MyObject"),
        };

    public MyObjectWS_Stub()
    {
        this(null);
    }

    public MyObjectWS_Stub(DefaultTypeMappingRegistry tmr)
    {
        super(_portType, _bindings);
        _setProperty("xmlrpc.schema.uri", (Object)
"http://www.w3.org/2001/XMLSchema".intern());
        _setProperty("version", (Object) "1.1");
        TypeMappingRegistry _tm = null;
        try {
            if (tmr != null)
                _tm = tmr;
            else {
                _tm = new DefaultTypeMappingRegistry();
            }
            _setTypeMappingRegistry(_tm);
        } catch (Exception ex) {
            throw new javax.xml.rpc.JAXRPCException("failed to initialize type mapping
registry: " + ex.getMessage());
        }
    }

    public boolean setString(java.lang.String _arg0)
        throws java.rmi.RemoteException
    {
        com.sssw.jbroker.web.portable.ClientResponse in = null;

        try {
            // create an output stream
            _getDelegate().setProperty("xmlrpc.soap.operation.name",
                new com.sssw.jbroker.web.QName("urn:com.exsamp.obj.MyObject", "setString"));
            //create request
            com.sssw.jbroker.web.portable.ClientRequest out =
                _request("setString", true, "soap", false,
"\urn:com.exsamp.obj.MyObject/setString\");

```

```
        _getDelegate().setProperty("soapAction", (Object)
"\urn:com.exsamp.obj.MyObject/getString\");
        _getDelegate().setProperty(Constants.HTTP_CONTENT_TYPE, (Object) "text/xml;
charset=utf-8");
        out._setProperties(_getDelegate().getProperties());
        Object arg = null;

        // marshal the parameters
        arg = _arg0;
        out.writeObject(arg, "arg0");

        // do the invocation
        in = _invoke(out);
        // unmarshal the results

        // return
        java.lang.Boolean retWrapper = (java.lang.Boolean)in.readObject(boolean.class,
"result");
        boolean ret = retWrapper.booleanValue();
        return ret;

    } catch (java.lang.Throwable t) {

        // map to remote exception
        throw com.sssw.jbroker.web.ServiceException.mapToRemote(t);
    }
}

public java.lang.String getString()
    throws java.rmi.RemoteException
{
    com.sssw.jbroker.web.portable.ClientResponse in = null;

    try {
        // create an output stream
        _getDelegate().setProperty("xmlrpc.soap.operation.name",
            new com.sssw.jbroker.web.QName("urn:com.exsamp.obj.MyObject", "getString"));
        //create request
        com.sssw.jbroker.web.portable.ClientRequest out =
            _request("getString", true, "soap", false,
"\urn:com.exsamp.obj.MyObject/getString\");
        _getDelegate().setProperty("soapAction", (Object)
"\urn:com.exsamp.obj.MyObject/getString\");
        _getDelegate().setProperty(Constants.HTTP_CONTENT_TYPE, (Object) "text/xml;
charset=utf-8");
        out._setProperties(_getDelegate().getProperties());
        Object arg = null;

        // do the invocation
        in = _invoke(out);
```

```
// unmarshal the results

// return
java.lang.String ret = null;
try {
    ret = (java.lang.String)
        in.readObject(java.lang.String.class, "result");
} catch (java.io.EOFException eofExc) {
    ret = null;
}
return ret;

} catch (java.lang.Throwable t) {

    // map to remote exception
    throw com.sssw.jbroker.web.ServiceException.mapToRemote(t);
}
}

public java.lang.String sayHello()
    throws java.rmi.RemoteException
{
    com.sssw.jbroker.web.portable.ClientResponse in = null;

    try {
        // create an output stream
        _getDelegate().setProperty("xmlrpc.soap.operation.name",
            new com.sssw.jbroker.web.QName("urn:com.exsamp.obj.MyObject", "sayHello"));
        //create request
        com.sssw.jbroker.web.portable.ClientRequest out =
            _request("sayHello", true, "soap", false,
                "\"urn:com.exsamp.obj.MyObject/sayHello\"");
        _getDelegate().setProperty("soapAction", (Object)
            "\"urn:com.exsamp.obj.MyObject/sayHello\"");
        _getDelegate().setProperty(Constants.HTTP_CONTENT_TYPE, (Object) "text/xml;
            charset=utf-8");
        out._setProperties(_getDelegate().getProperties());
        Object arg = null;

        // do the invocation
        in = _invoke(out);
        // unmarshal the results

        // return
        java.lang.String ret = null;
        try {
            ret = (java.lang.String)
                in.readObject(java.lang.String.class, "result");
        } catch (java.io.EOFException eofExc) {
```

```
        ret = null;
    }
    return ret;

} catch (java.lang.Throwable t) {

    // map to remote exception
    throw com.sssw.jbroker.web.ServiceException.mapToRemote(t);
}

}

private static Properties _rootHeaders = new Properties();
static {
    _rootHeaders.setProperty("content-type", "text/xml; charset=UTF-8");
    _rootHeaders.setProperty("content-id", "<soapbody>");
}
}
```

MyObjectWSClient.java

MyObjectWSClient is a simple client application that accesses the Web Service by:

1. Instantiating MyObjectWSService via JNDI lookup
2. Using the MyObjectWSService object to obtain the stub (MyObjectWS_Stub)
3. Calling Web Service methods via the MyObjectWS_Stub object

The wizard generates this source code for it:

```
// The following code was generated within the SilverStream eXtend Workbench
// using the integrated Web Services Wizard. This code *requires* process() method
// modification in order to execute as expected. Please keep in mind when making
// modifications that method signatures must be consistent across all
// generated objects.

package com.exsamp.obj;

import javax.naming.*;

public class MyObjectWSClient
{
    public void process(String[] args) throws Exception
    {
        MyObjectWS remote = getRemote(args);

        // The following code has been generated for your testing convenience. In
        // order to successfully test your Web Service, you must uncomment one or
        // more of these lines and supply meaningful arguments where necessary.
        // Once you have modified the test method(s) below, compile this class and
```

```

// execute it from a command line with your class path set appropriately.

// System.out.println("Test Result = " + remote.getString());
// System.out.println("Test Result = " + remote.setString(java.lang.String));
// System.out.println("Test Result = " + remote.sayHello());

}

public MyObjectWS getRemote(String[] args) throws Exception
{
    InitialContext ctx = new InitialContext();

    String lookup = "xmlrpc:soap:com.exsamp.obj.MyObjectWSService";
    MyObjectWSService service = (MyObjectWSService)ctx.lookup(lookup);
    MyObjectWS remote = (MyObjectWS)service.getMyObjectWSPort();

    return remote;
}

public static void main(String[] args)
{
    try
    {
        MyObjectWSClient client = new MyObjectWSClient();
        client.process(args);
    }
    catch (Exception _e)
    {
        System.out.println("*** Error Executing Generated Test Client ***");
        _e.printStackTrace();
    }
}
}

```

Modifications needed The process() method of the generated MyObjectWSClient.java file must be edited to uncomment the Web Service method call to be tested. Here's the change:

```

// System.out.println("Test Result = " + remote.getString());
// System.out.println("Test Result = " + remote.setString(java.lang.String));
System.out.println("Test Result = " + remote.sayHello());

```

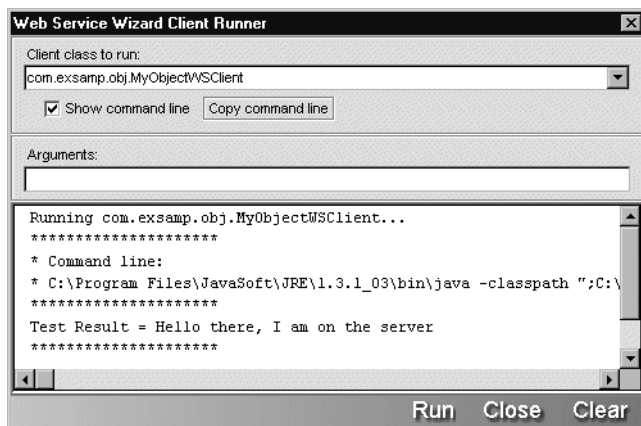
Deployment descriptor

Because this scenario uses the tie model, the Web Service Wizard automatically updates the **web.xml** file to declare **MyObjectWSTie** as the servlet class to handle requests for the **MyObject** Web Service:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
    <servlet-name>MyObject</servlet-name>
    <servlet-class>com.exsamp.obj.MyObjectWSTie</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyObject</servlet-name>
    <url-pattern>MyObject</url-pattern>
  </servlet-mapping>
</web-app>
```


Runtime test result

Once this project is built and the WAR file is created and deployed to the J2EE server, the **MyObject** Web Service is ready for a test run. Here's the result of using the **Client Runner** in Workbench to execute the **MyObjectWSClient** application:



10 Generating Web Service Consumers

This chapter walks you through the basic steps and a typical scenario for using the **Web Service Wizard** to generate a Web Service consumer (a program that accesses a Web Service).

 To learn about the steps and scenarios for using the wizard when you want to create a Web Service, see Chapter 9, “Generating Web Services”.

Basics


You can use the Web Service Wizard of Workbench to generate the code needed for a **Java-based consumer program** to access any **standard (SOAP-based) Web Service**. The generated code handles all HTTP SOAP processing under the covers, enabling the consumer program to call the Web Service as a **Java remote object** (using RMI) and invoke its methods.


For input, the wizard requires a **WSDL file** that describes the Web Service to access. It can handle a wide variety of Web Service implementations, including:

- Document-style and RPC-style bindings
- Basic and complex types
- J2EE providers, Microsoft .NET providers, and others

The wizard generates Java source files based on JAX-RPC (Java API for XML-based RPC) and jBroker Web (the JAX-RPC implementation included with SilverStream eXtend). JAX-RPC is the J2EE specification that provides Web Service support.

You can use the generated files as is or modify them when necessary. The advantage of this Java-oriented approach is that you can deal with Web Services using the familiar technologies of RMI and J2EE instead of coding lower-level SOAP APIs.

 For an introduction to Web Service concepts, standards, and technologies, see Chapter 8, “Understanding Web Services”.

 For detailed documentation on the wizard, see the Web Service Wizard chapter in the *Tools Guide*.

Steps

The process of developing your consumer program involves:

1. Preparing to generate by setting up your project
2. Providing a WSDL file that describes the Web Service for which you want the wizard to generate consumer code
3. Generating the consumer files by using the wizard
4. Examining the generated files that the wizard creates, including Java source for:
 - A **remote interface**, **service classes**, and a **stub class** that facilitate the Web Service access
 - Any **type classes** needed for method arguments and return values
 - A **simple Java client class** that uses the other classes to make method calls
5. Editing the generated files to adjust the method calls to make and the Web Service location to point to
6. Using the generated files either as is or by including the consumer code in some other Java application
7. Running the consumer program in your development environment (for testing) and in the production environment

Preparing to generate

To prepare for using the Web Service Wizard, you:

1. Set up an appropriate **project** in Workbench.

The type of project you should create depends on how you ultimately plan to use the consumer code that the wizard will generate. For instance:

If you plan to use the consumer code in	You should create
A standard Java application (perhaps based on the simple Java client class that the wizard generates)	A JAR project
A J2EE application client	A CAR project
A JSP page or servlet	A WAR project
An Enterprise JavaBean	An EJB JAR project

2. Add the **archives required by jBroker Web** to your project:
 - **jbroke-*web.jar***, which contains the jBroker Web API classes needed at runtime
 - **jaxrpc-*api.jar*** and **saaj-*api.jar***, which contain the Java API classes for XML-based RPC and SOAP processing
 - **xerces-*jar*** or another XML parser

You'll find these JARs in the Workbench **compilelib** directory.

3. Edit the **classpath** of your project so you can compile your consumer classes once they're generated and edited. You'll need to include:
 - `jbroke-web.jar`
 - `jaxrpc-api.jar` and `saaj-api.jar`
 - `xerces-jar` (or another XML parser)
 - Any application-specific entries

For **J2EE projects**, you'll also need `j2ee-api_1_n.jar` (it's included automatically when you create a J2EE project in Workbench).

If you use **SOAP message handlers** (an advanced JAX-RPC feature) in your application, the project will also require the following archives: `activation-jar`, `commons-logging-jar`, `dom4j-jar`, `jaxp-api.jar`, `saaj-ri.jar`, and `j2ee-api_1_n.jar` (for mail support). You'll find these JARs in the Workbench `compilelib` directory.

Providing a WSDL file

To generate consumer code, you'll need to provide the Web Service Wizard with a WSDL file that describes the target Web Service. It's a good idea to obtain the file location or URL of this WSDL file before you start the wizard.

These are common scenarios:

- **For a Web Service developed in your organization**, you might have the WSDL file on your file system or even in your project.
- **For an external Web Service**, you should be able to get the WSDL file's URL from the appropriate Web site or registry.

Example: WSDL file for Autoloan .NET Web Service

Suppose you want to generate consumer code to use the **Autoloan** .NET Web Service, which is listed on the XMethods public registry under the name **Equated Monthly Instalment (EMI) Calculator**. That Web Service calculates and returns the monthly loan payment for a given term (number of months), interest rate, and loan amount.

In this case, you can go to the Web site www.xmethods.net to discover the URL for the corresponding WSDL file:

```
http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
```

When you provide this URL to the Web Service Wizard, it will read the WSDL file to learn what it needs to know about the Autoloan Web Service:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:s0="http://circle24.com/webservices/"
  targetNamespace="http://circle24.com/webservices/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema attributeFormDefault="qualified" elementFormDefault="qualified"
      targetNamespace="http://circle24.com/webservices/">
      <s:element name="Calculate">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="Months" type="s:double" />
            <s:element minOccurs="1" maxOccurs="1" name="RateOfInterest" type="s:double" />
            <s:element minOccurs="1" maxOccurs="1" name="Amount" type="s:double" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="CalculateResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="CalculateResult" nillable="true"
              type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="string" nillable="true" type="s:string" />
    </s:schema>
  </types>
  <message name="CalculateSoapIn">
```

```

    <part name="parameters" element="s0:Calculate" />
</message>
<message name="CalculateSoapOut">
  <part name="parameters" element="s0:CalculateResponse" />
</message>
<message name="CalculateHttpGetIn">
  <part name="Months" type="s:string" />
  <part name="RateOfInterest" type="s:string" />
  <part name="Amount" type="s:string" />
</message>
<message name="CalculateHttpGetOut">
  <part name="Body" element="s0:string" />
</message>
<message name="CalculateHttpPostIn">
  <part name="Months" type="s:string" />
  <part name="RateOfInterest" type="s:string" />
  <part name="Amount" type="s:string" />
</message>
<message name="CalculateHttpPostOut">
  <part name="Body" element="s0:string" />
</message>
<portType name="AutoloanSoap">
  <operation name="Calculate">
    <input message="s0:CalculateSoapIn" />
    <output message="s0:CalculateSoapOut" />
  </operation>
</portType>
<portType name="AutoloanHttpGet">
  <operation name="Calculate">
    <input message="s0:CalculateHttpGetIn" />
    <output message="s0:CalculateHttpGetOut" />
  </operation>
</portType>
<portType name="AutoloanHttpPost">
  <operation name="Calculate">
    <input message="s0:CalculateHttpPostIn" />
    <output message="s0:CalculateHttpPostOut" />
  </operation>
</portType>
<binding name="AutoloanSoap" type="s0:AutoloanSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
  <operation name="Calculate">
    <soap:operation soapAction="http://circle24.com/webservices/Calculate"
      style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>

```

```
        <soap:body use="literal" />
    </output>
</operation>
</binding>
<binding name="AutoloanHttpGet" type="s0:AutoloanHttpGet">
  <http:binding verb="GET" />
  <operation name="Calculate">
    <http:operation location="/Calculate" />
    <input>
      <http:urlEncoded />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<binding name="AutoloanHttpPost" type="s0:AutoloanHttpPost">
  <http:binding verb="POST" />
  <operation name="Calculate">
    <http:operation location="/Calculate" />
    <input>
      <mime:content type="application/x-www-form-urlencoded" />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<service name="Autoloan">
  <documentation>This Web Service mimics a Simple Autoloan calculator.</documentation>
  <port name="AutoloanSoap" binding="s0:AutoloanSoap">
    <soap:address location="http://upload.eraserver.net/circle24/autoloan.asmx" />
  </port>
  <port name="AutoloanHttpGet" binding="s0:AutoloanHttpGet">
    <http:address location="http://upload.eraserver.net/circle24/autoloan.asmx" />
  </port>
  <port name="AutoloanHttpPost" binding="s0:AutoloanHttpPost">
    <http:address location="http://upload.eraserver.net/circle24/autoloan.asmx" />
  </port>
</service>
</definitions>
```

Understanding the WSDL

In the Autoloan WSDL, you can ignore the definitions for **HttpGet** and **HttpPost** (including message, portType, binding, and service port). Only the **Soap** definitions apply to the Web Service consumer program you're developing.

Notice that this Web Service exposes one method named **calculate()**. It takes a **Calculate** object containing three doubles (Months, RateOfInterest, and Amount) and returns a **CalculateResponse** object containing one string (CalculateResult). The Web Service Wizard will generate a corresponding remote interface in Java to support calling this method.

The **types** section specifies the **XML Schema** definitions for Calculate and CalculateResponse. The Web Service Wizard will generate corresponding type classes in Java to represent these objects.

If you look in the **binding** section for AutoloanSoap, you'll see that this Web Service is defined as **document style** (as opposed to **RPC style**). That's typical of .NET Web Services. Binding style describes the format of SOAP messages and can affect interoperability with other Web Service environments:

Binding style	What it means
Document (with literal use)	The SOAP message body contains just the XML document being exchanged and message parts map to elements literally defined in the WSDL file's XML schema.
RPC (with encoded use)	The SOAP message body contains argument and return values, individually wrapped in ad hoc elements that the recipient must interpret by applying specified encoding rules to each message part's type.

The Web Service Wizard will generate the Java code needed to handle the specified binding style.

The **port** definition for AutoloanSoap (at the end of the WSDL file) specifies the **address** (URL) where the Web Service can be accessed:

```
http://upload.eraserver.net/circle24/autoloan.asmx
```

The Web Service Wizard will use this URL in the service and stub classes it generates for calling the Web Service.

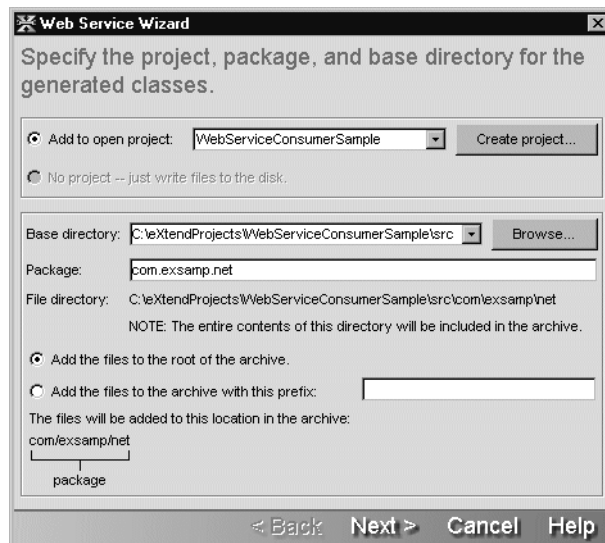
Generating the consumer files

Once you've set up your project and located the appropriate WSDL file, you're ready to use the Web Service Wizard. The wizard produces one Web Service consumer at a time, so you'll need to use it multiple times if you have several to develop.

Each time you launch the wizard, it uses the WSDL file and other input you provide to generate a set of consumer source files. Here's a summary of the process:

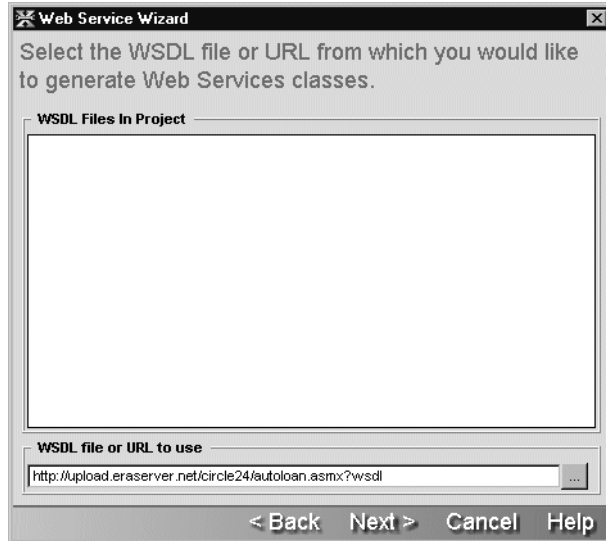
1. Select **File>New** to display the New File dialog and go to the **Web Services** tab.
2. Launch the Web Service Wizard by selecting **Existing Web Service**.
3. When the wizard prompts you for **project location** information, specify:
 - The **project** you set up to contain the generated Web Service consumer files
 - The target **directory and package** in that project

For example, suppose you're generating a consumer for the Autoloan Web Service. You might specify `WebServiceConsumerSample` as the target JAR project and `com.exsamp.net` as the package for generated classes:



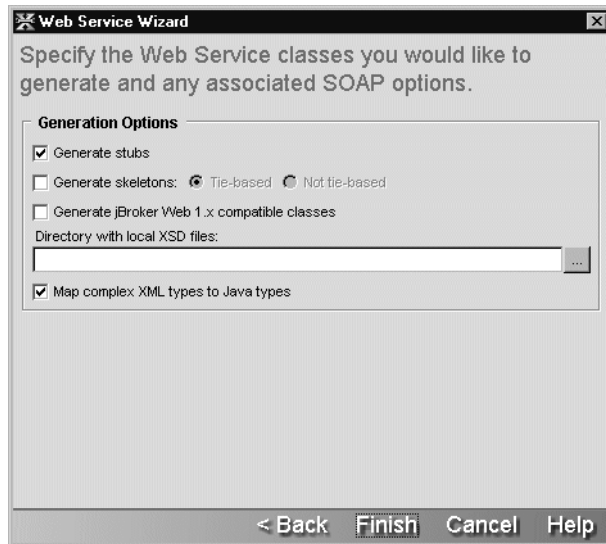
4. When the wizard prompts you, specify the **WSDL file** that describes your target Web Service.

For example, when generating a consumer for the Autoloan Web Service, you specify the WSDL file URL obtained from the XMethods public registry:



5. When the wizard prompts you for **class-generation and SOAP options**, you must specify details about the code to create:
 - To get the files needed for a Web Service consumer, check **Generate stubs** (and leave **Generate skeletons** unchecked).
 - To automatically generate type classes for any complex types in the WSDL, check **Map complex XML types to Java types**.

For example, these options will generate the appropriate consumer source files (including type classes) for the Autoloan Web Service:



NOTE Support for jBroker Web 1.x applications is available via a **backward-compatibility** option. For more information, see If you choose jBroker Web 1.x compatibility (in the previous chapter).

6. Click **Finish** when you're done specifying options for the Web Service consumer.

Examining the generated files

Once you finish the wizard, it generates everything you've specified for your Web Service consumer and updates other parts of your project with supporting changes:

What the wizard generates	Details
Java source file for remote interface	<p>xxx.java An interface that extends <code>java.rmi.Remote</code> and declares the methods exposed by the target Web Service (as determined from the WSDL file). The generated stub class <code>xxx_Stub</code> implements this interface to support method calls for the Web Service.</p>
Java source files for stubs	<p>xxxService.java Service interface used by JAX-RPC clients to obtain the stub for the target Web Service.</p> <p>xxxServiceImpl.java Service implementation class that handles instantiation of the stub (<code>xxx_Stub</code>). It also supports alternative ways of accessing the target Web Service, including dynamic (stubless) calls.</p> <p>(Note that the names generated for the service interface and implementation class depend on your WSDL and may omit the text Service.)</p> <p>xxx_Stub.java Facilitates method calls from a Java-based consumer to the target Web Service. <code>xxx_Stub</code> implements the generated remote interface by sending an appropriate HTTP SOAP request for each method call.</p> <p>xxxClient.java Simple client application that works as a consumer of the target Web Service. It obtains the stub (via the Service object) then uses the stub to call Web Service methods.</p> <p>You can run <code>xxxClient</code> from Workbench (select Project>Run Web Service Client Class) or from a command line.</p>
Updates to project contents	<p>The wizard updates your project to add generated files to it.</p>

About generated file names

When generating file names, the Web Service Wizard follows the naming rules specified by JAX-RPC. For a Web Service consumer, the resulting file names are based on the definitions in the WSDL.

For simplicity, this documentation uses *xxx* to represent the portion of a generated Web Service consumer file name that's derived from a WSDL definition.

Additional details of generation

Under the covers, the Web Service Wizard uses the **jBroker Web compilers** when generating the Web Service consumer files listed above. In some cases, these compilers may generate additional code or files to support requirements specific to your application, such as:

- Type mapping
- Faults
- Multiple portType definitions



For more information, see the jBroker Web help.

Example: generated consumer files for Autoloan .NET Web Service

The consumer code that the Web Service Wizard generates for the Autoloan Web Service consists of these **standard files for Web Service access**:

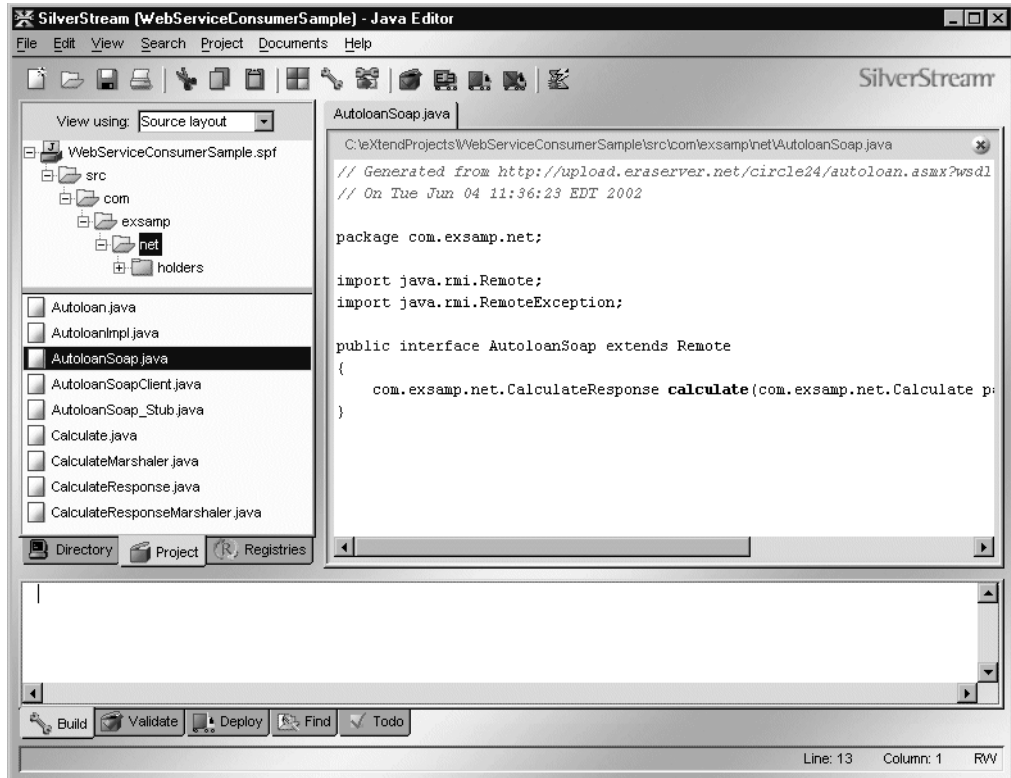
- AutoloanSoap.java (remote interface)
- Autoloan.java (service interface)
- AutoloanImpl.java (service implementation class)
- AutoloanSoap_Stub.java
- AutoloanSoapClient.java

And these **application-specific files for mapping the complex types** defined in the WSDL:

- Calculate.java
- CalculateMarshaler.java
- CalculateHolder.java
- CalculateResponse.java
- CalculateResponseMarshaler.java

- CalculateResponseHolder.java
- autoloan.asmx.xmlrpc.type.mappings

When creating these files, the wizard adds them to your project on the directory path you've specified:



AutoloanSoap.java

This is the remote interface used by the stub class to support method calls for the Autoloan Web Service.

```

// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Jun 04 11:36:23 EDT 2002

```

```
package com.exsamp.net;
```

```
import java.rmi.Remote;
```

```
import java.rmi.RemoteException;

public interface AutoloanSoap extends Remote
{
    com.exsamp.net.CalculateResponse calculate(com.exsamp.net.Calculate parameters)
        throws RemoteException;
}
```

Autoloan.java

This is the service interface that's used in JAX-RPC to help clients obtain the stub for the Web Service.

```
// Tue Jun 04 11:36:23 EDT 2002

package com.exsamp.net;

import javax.xml.rpc.ServiceException;

public interface Autoloan extends javax.xml.rpc.Service
{
    public AutoloanSoap_Stub getAutoloanSoap()
        throws ServiceException;
}
```

AutoloanImpl.java

This is the service implementation class that handles instantiation of the stub (AutoloanSoap_Stub).

```
// Tue Jun 04 11:36:23 EDT 2002

package com.exsamp.net;

import java.io.FileNotFoundException;
import java.util.Iterator;
import java.util.Hashtable;
import java.util.Properties;
import java.util.ArrayList;
import java.net.URL;
import java.net.MalformedURLException;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceException;
import com.sssw.jbroker.web.Binding;
import com.sssw.jbroker.web.encoding.DefaultTypeMappingRegistry;
```

```
public class AutoloanImpl
    extends com.sssw.jbroker.web.xml.rpc.ServiceImpl
    implements Autoloan
{
    public AutoloanImpl()
    {
        try {
            _typeMappingRegistry.importTypeMappings(_tmprops);
        } catch (Exception ex) {
            throw new javax.xml.rpc.JAXRPCException("failed to populate default type
mapping registry: " + ex.getMessage());
        }
        try {
            createCalls();
        } catch (ServiceException ex) {
            throw new javax.xml.rpc.JAXRPCException("failed to create the call objects: "
+ ex.getMessage());
        }
    }

    public QName getServiceName() { return _serviceName; }

    public Iterator getPorts() { return _portMapping.keySet().iterator(); }

    public void setProxyMode(boolean proxy) { _proxy = proxy; }

    public boolean getProxyMode() { return _proxy; }

    public URL getWSDLDocumentLocation()
    {
        return null;
    }

    public java.rmi.Remote getPort(Class serviceDefInterface)
        throws ServiceException
    {
        if (serviceDefInterface == null)
            throw new ServiceException("No Service class specified.");
        if (!java.rmi.Remote.class.isAssignableFrom(serviceDefInterface))
            throw new ServiceException("Class is not a valid Interface.");

        String stubName = (String) _intfMapping.get(serviceDefInterface);
        Binding binding = (Binding) _intfBinding.get(serviceDefInterface);

        if (stubName == null)
            return getPort(serviceDefInterface, binding,
                _classInfo, _typeMappingRegistry, null);
        else
            return getPort(stubName, binding, _typeMappingRegistry);
    }
}
```

```
    }

    public java.rmi.Remote getPort(QName portName, Class serviceDefInterface)
        throws ServiceException
    {
        return getPort(portName, serviceDefInterface, getProxyMode());
    }

    public java.rmi.Remote getPort(QName portName, Class serviceDefInterface, boolean
proxy)
        throws ServiceException
    {
        if ((proxy==false) || (serviceDefInterface == null)) &&
            (portName != null) {
            String stubName = (String) _portMapping.get(portName);
            Binding binding = (Binding) _portBinding.get(portName);

            if (stubName == null) return getPort(null, serviceDefInterface);

            try {
                return getPort(stubName, binding, portName,
                    _typeMappingRegistry);
            } catch (Exception ex) {
                return getPort(null, serviceDefInterface);
            }
        } else {
            if (serviceDefInterface == null)
                throw new ServiceException("No Service class specified.");
            if (!java.rmi.Remote.class.isAssignableFrom(serviceDefInterface))
                throw new ServiceException("Class is not a valid Interface.");

            Binding binding = (Binding) _intfBinding.get(serviceDefInterface);
            String uri = (portName == null) ? null : portName.getNamespaceURI();
            return getPort(serviceDefInterface, binding, _classInfo,
                _typeMappingRegistry, uri);
        }
    }

    public Call[] getCalls(QName portName)
        throws ServiceException
    {
        ArrayList callslst = (ArrayList) _calls.get(portName);
        if (callslst == null) return null;
        Call[] calls = new Call[callslst.size()];
        return (Call[]) callslst.toArray(calls);
    }

    private void addCall(QName portName, Call call)
    {
        ArrayList callslst = (ArrayList) _calls.get(portName);
```



```

        if (callslist == null) {
            callslist = new ArrayList();
            _calls.put(portName, callslist);
        }
        callslist.add(call);
    }

    public AutoloanSoap_Stub getAutoloanSoap()
        throws ServiceException
    {
        try {
            return (AutoloanSoap_Stub) getPort(new QName(
                "http://circle24.com/webservices/", "AutoloanSoap"), null, false);
        } catch (Exception ex) {
            return (AutoloanSoap_Stub) getPort(com.exsamp.net.AutoloanSoap.class);
        }
    }

    private void createCalls()
        throws ServiceException
    {
        Call call = null;

        call = createCall(new QName("http://circle24.com/webservices/", "AutoloanSoap"),
            new QName("http://circle24.com/webservices/", "Calculate"));
        call.addParameter("{http://circle24.com/webservices/}Calculate",
            new QName("http://circle24.com/webservices/", "Calculate"),
            com.exsamp.net.Calculate.class, ParameterMode.IN);
        call.addParameter("{http://circle24.com/webservices/}CalculateResponse",
            new QName("http://circle24.com/webservices/", "CalculateResponse"),
            com.exsamp.net.CalculateResponse.class, ParameterMode.OUT);
        call.setReturnType(new QName("http://circle24.com/webservices/",
            "CalculateResponse"), com.exsamp.net.CalculateResponse.class);
        call.setProperty(Call.OPERATION_STYLE_PROPERTY, "document");
        call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY, null);
        call.setProperty(Call.SOAPACTION_URI_PROPERTY,
            "\http://circle24.com/webservices/Calculate\");

        call.setTargetEndpointAddress("http://upload.eraserver.net/circle24/autoloan.asmx");
        addCall(new QName("http://circle24.com/webservices/", "AutoloanSoap"), call);
    }

    static boolean _proxy = true;
    static final QName _serviceName;
    static final Hashtable _intfMapping = new Hashtable();
    static final Hashtable _intfBinding = new Hashtable();
    static final Hashtable _portBinding = new Hashtable();
    static final Hashtable _portMapping = new Hashtable();

```

```
static final Hashtable _classInfo = new Hashtable();
static final Properties _tmprops = new Properties();
private final Hashtable _calls = new Hashtable();

static {
    _serviceName = new QName("http://circle24.com/webservices/",
"com.exsamp.net.Autoloan");

    _intfBinding.put (AutoloanSoap.class,
        new Binding("soap", "http://upload.eraserver.net/circle24/autoloan.asmx"));
    _portBinding.put (new QName("http://circle24.com/webservices/", "AutoloanSoap"),
        new Binding("soap", "http://upload.eraserver.net/circle24/autoloan.asmx"));
    _intfMapping.put (AutoloanSoap.class, "com.exsamp.net.AutoloanSoap_Stub");
    _portMapping.put (new QName("http://circle24.com/webservices/",
"AutoloanSoap"), "com.exsamp.net.AutoloanSoap_Stub");

    Hashtable _methodInfo;
    Hashtable _paramInfo;
    Properties _props;

    _methodInfo = new Hashtable();
    _paramInfo = new Hashtable();
    _props = new Properties();

    _props.setProperty("jbroker.web.soap.action", "\"http://circle24.com/webservices/Calculate\
");
    _paramInfo.put ("Properties", _props);
    _props = new Properties();
    _props.setProperty("jbroker.web.parameter.name", "parameters");
    _props.setProperty("jbroker.web.parameter.inout", "1");
    _paramInfo.put ("Param0", _props);
    _props = new Properties();
    _props.setProperty("jbroker.web.parameter.name", "parameters");
    _props.setProperty("jbroker.web.parameter.inout", "2");
    _paramInfo.put ("Result", _props);
    _methodInfo.put ("Calculate", _paramInfo);
    _classInfo.put ("com.exsamp.net.AutoloanSoap", _methodInfo);

    _tmprops.put ("tm1", "com.exsamp.net.CalculateResponse
com.exsamp.net.CalculateResponseMarshaler com.exsamp.net.CalculateResponseMarshaler
http://circle24.com/webservices/ CalculateResponse none");
    _tmprops.put ("tm0", "com.exsamp.net.Calculate com.exsamp.net.CalculateMarshaler
com.exsamp.net.CalculateMarshaler http://circle24.com/webservices/ Calculate none");
}
}
```

AutoloanSoap_Stub.java

This is the stub class. It passes method calls to the Autoloan Web Service as HTTP SOAP requests.

```
// Tue Jun 04 11:36:23 EDT 2002

package com.exsamp.net;

import com.exsamp.net.holders.*;

import java.util.Properties;
import com.sssw.jbroker.web.core.Constants;
import com.sssw.jbroker.web.encoding.TypeMappingRegistry;
import com.sssw.jbroker.web.encoding.DefaultTypeMappingRegistry;

public class AutoloanSoap_Stub
    extends com.sssw.jbroker.web.portable.Stub
    implements AutoloanSoap
{
    private static com.sssw.jbroker.web.QName _portType =
        new com.sssw.jbroker.web.QName("http://circle24.com/webservices/",
"AutoloanSoap");

    private static final com.sssw.jbroker.web.Binding[] _bindings =
        new com.sssw.jbroker.web.Binding[] {
            new com.sssw.jbroker.web.Binding("soap",
"http://upload.eraserver.net/circle24/autoloan.asmx"),
        };

    public AutoloanSoap_Stub()
    {
        this(null);
    }

    public AutoloanSoap_Stub(DefaultTypeMappingRegistry tmr)
    {
        super(_portType, _bindings);
        _setProperty("xmlrpc.schema.uri", (Object)
"http://www.w3.org/2001/XMLSchema".intern());
        _setProperty("version", (Object) "1.1");
        TypeMappingRegistry _tm = null;
        try {
            if (tmr != null)
                _tm = tmr;
            else {
                _tm = new DefaultTypeMappingRegistry();
                if (_tmprops.size() > 0) _tm.importTypeMappings(_tmprops);
            }
        }
    }
}
```

```
        _setTypeMappingRegistry(_tm);
    } catch (Exception ex) {
        throw new javax.xml.rpc.JAXRPCException("failed to initialize type mapping
registry: " + ex.getMessage());
    }
}

public com.exsamp.net.CalculateResponse calculate(com.exsamp.net.Calculate _arg0)
    throws java.rmi.RemoteException
{
    com.sssw.jbroker.web.portable.ClientResponse in = null;

    try {
        // create an output stream
        _getDelegate().setProperty("xmlrpc.soap.operation.name",
            new com.sssw.jbroker.web.QName("http://circle24.com/webservices/",
"Calculate"));
        //create request
        com.sssw.jbroker.web.portable.ClientRequest out =
            _request("Calculate", true, "literal", true,
"\http://circle24.com/webservices/Calculate\");
        _getDelegate().setProperty("soapAction", (Object)
"\http://circle24.com/webservices/Calculate\");
        _getDelegate().setProperty(Constants.HTTP_CONTENT_TYPE, (Object) "text/xml;
charset=utf-8");
        out._setProperties(_getDelegate().getProperties());
        Object arg = null;

        // marshal the parameters
        arg = _arg0;
        out.writeObject(arg, "http://circle24.com/webservices/", "Calculate");

        // do the invocation
        in = _invoke(out);
        // unmarshal the results

        // return
        com.exsamp.net.CalculateResponse ret = null;
        try {
            ret = (com.exsamp.net.CalculateResponse)
                in.readObject(com.exsamp.net.CalculateResponse.class,
"http://circle24.com/webservices/", "CalculateResponse");
        } catch (java.io.EOFException eofExc) {
            ret = null;
        }
        return ret;
    } catch (java.lang.Throwable t) {

        if (t instanceof com.sssw.jbroker.web.ServiceException) {
```

```

        com.sssw.jbroker.web.ServiceException sex =
            (com.sssw.jbroker.web.ServiceException) t;
        if (sex.getTargetException() != null)
            t = sex.getTargetException();
    }

    // map to remote exception
    throw com.sssw.jbroker.web.ServiceException.mapToRemote(t);
}
}
static final Properties _tmprops = new Properties();

static {

    _tmprops.put("tml", "com.exsamp.net.CalculateResponse
com.exsamp.net.CalculateResponseMarshaler com.exsamp.net.CalculateResponseMarshaler
http://circle24.com/webservices/ CalculateResponse none");
    _tmprops.put("tm0", "com.exsamp.net.Calculate com.exsamp.net.CalculateMarshaler
com.exsamp.net.CalculateMarshaler http://circle24.com/webservices/ Calculate none");
}

private static Properties _rootHeaders = new Properties();
static {
    _rootHeaders.setProperty("content-type", "text/xml; charset=UTF-8");
    _rootHeaders.setProperty("content-id", "<soapbody>");
}
}
}

```

AutoloanSoapClient.java

This is a simple client application that obtains the stub (via the Service object) then uses it to call the calculate() method of the Autoloan Web Service. (Notice that this method call is generated as a comment. You'll learn what to do with it a little later in "Editing the generated files".)

```

// The following code was generated within the SilverStream eXtend Workbench
// using the integrated Web Services Wizard. This code *requires* process() method
// modification in order to execute as expected. Please keep in mind when making
// modifications that method signatures must be consistent across all
// generated objects.

package com.exsamp.net;

import javax.naming.*;

public class AutoloanSoapClient
{
    public void process(String[] args) throws Exception
    {

```

```
AutoloanSoap remote = getRemote(args);

// The following code has been generated for your testing convenience. In
// order to successfully test your Web Service, you must uncomment one or
// more of these lines and supply meaningful arguments where necessary.
// Once you have modified the test method(s) below, compile this class and
// execute it from a command line with your class path set appropriately.

// System.out.println("Test Result = " +
remote.calculate(com.exsamp.net.Calculate));

}

public AutoloanSoap getRemote(String[] args) throws Exception
{
    InitialContext ctx = new InitialContext();

    String lookup = "xmlrpc:soap:com.exsamp.net.Autoloan";
    Autoloan service = (Autoloan)ctx.lookup(lookup);
    AutoloanSoap remote = (AutoloanSoap)service.getAutoloanSoap();

    return remote;
}

public static void main(String[] args)
{
    try
    {
        AutoloanSoapClient client = new AutoloanSoapClient();
        client.process(args);
    }
    catch (Exception _e)
    {
        System.out.println("*** Error Executing Generated Test Client ***");
        _e.printStackTrace();
    }
}
}
```

Calculate.java

This class represents the complex type Calculate that's defined in the WSDL.

```
// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Jun 04 11:36:21 EDT 2002
```

```
package com.exsamp.net;

public class Calculate implements java.io.Serializable
{
    public Calculate() {}

    public Calculate(double monthsVal, double rateOfInterestVal, double amountVal) {
        _months = monthsVal;
        _rateOfInterest = rateOfInterestVal;
        _amount = amountVal;
    }
    private double _months;
    public double getMonths() {
        return _months;
    }
    public void setMonths(double monthsVal) {
        _months = monthsVal;
    }
    private double _rateOfInterest;
    public double getRateOfInterest() {
        return _rateOfInterest;
    }
    public void setRateOfInterest(double rateOfInterestVal) {
        _rateOfInterest = rateOfInterestVal;
    }
    private double _amount;
    public double getAmount() {
        return _amount;
    }
    public void setAmount(double amountVal) {
        _amount = amountVal;
    }
    public java.lang.String toString()
    {
        StringBuffer buffer = new StringBuffer();
        buffer.append("{");
        buffer.append("months=" + _months);
        buffer.append(",");
        buffer.append("rateOfInterest=" + _rateOfInterest);
        buffer.append(",");
        buffer.append("amount=" + _amount);
        buffer.append("}");
        return buffer.toString();
    }
}
```

CalculateMarshaler.java

This class handles serialization and deserialization for Calculate.

```
// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Jun 04 11:36:21 EDT 2002

package com.exsamp.net;

import java.io.IOException;
import org.xml.sax.Attributes;
import com.sssw.jbroker.web.*;
import com.sssw.jbroker.web.encoding.*;
import com.sssw.jbroker.web.portable.InputStream;
import com.sssw.jbroker.web.portable.OutputStream;

public class CalculateMarshaler implements Marshaler
{
    // attributes
    // elements
    private static final java.lang.String _MONTHS = "Months";
    private static final java.lang.String _RATEOFINTEREST = "RateOfInterest";
    private static final java.lang.String _AMOUNT = "Amount";

    public Attribute[] getAttributes(Object obj)
    {
        return null;
    }

    public void serialize(OutputStream os, Object obj) throws IOException
    {
        Calculate jt = (Calculate) obj;
        os.writeObject(new java.lang.Double(jt.getMonths()), _MONTHS);
        os.writeObject(new java.lang.Double(jt.getRateOfInterest()), _RATEOFINTEREST);
        os.writeObject(new java.lang.Double(jt.getAmount()), _AMOUNT);
    }

    public Object deserialize(InputStream is, Class javaType)
        throws IOException
    {
        if (javaType != Calculate.class)
            throw new
                ServiceException("can't deserialize " + javaType.getName());

        try {
            // instantiate the object
            Calculate jt = (Calculate) javaType.newInstance();
            try {
                // read elements
                jt.setMonths(is.readDouble(_MONTHS));
            }
        }
    }
}
```



```

        jt.setRateOfInterest(is.readDouble(_RATEOFINTEREST));
        jt.setAmount(is.readDouble(_AMOUNT));
    } catch (java.io.EOFException eofExc) {}

    return jt;
} catch (Exception ex) {
    if (ex instanceof IOException)
        throw (IOException) ex;
    throw new ServiceException(ex);
}
}

public java.lang.String getMechanismType() { return null; }
}

```

CalculateHolder.java

This is the Holder class required by JAX-RPC to implement type mapping support for Calculate. Note that this class is generated in the **holders** subdirectory.

```

// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Jun 04 11:36:21 EDT 2002

package com.exsamp.net.holders;

import com.exsamp.net.Calculate;

public final class CalculateHolder implements javax.xml.rpc.holders.Holder
{
    public com.exsamp.net.Calculate value;

    public CalculateHolder() { }

    public CalculateHolder(com.exsamp.net.Calculate val)
    {
        value = val;
    }
}

```

CalculateResponse.java

This class represents the complex type CalculateResponse that's defined in the WSDL.

```

// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Jun 04 11:36:21 EDT 2002

package com.exsamp.net;

```

```
public class CalculateResponse implements java.io.Serializable
{
    public CalculateResponse() {}

    public CalculateResponse(java.lang.String calculateResultVal) {
        _calculateResult = calculateResultVal;
    }
    private java.lang.String _calculateResult;
    public java.lang.String getCalculateResult() {
        return _calculateResult;
    }
    public void setCalculateResult(java.lang.String calculateResultVal) {
        _calculateResult = calculateResultVal;
    }
    public java.lang.String toString()
    {
        StringBuffer buffer = new StringBuffer();
        buffer.append("(");
        buffer.append("calculateResult=" + _calculateResult);
        buffer.append(")");
        return buffer.toString();
    }
}
```

CalculateResponseMarshaler.java

This class handles serialization and deserialization for CalculateResponse.

```
// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Jun 04 11:36:21 EDT 2002

package com.exsamp.net;

import java.io.IOException;
import org.xml.sax.Attributes;
import com.sssw.jbroker.web.*;
import com.sssw.jbroker.web.encoding.*;
import com.sssw.jbroker.web.portable.InputStream;
import com.sssw.jbroker.web.portable.OutputStream;

public class CalculateResponseMarshaler implements Marshaler
{
    // attributes
    // elements
    private static final java.lang.String _CALCULATERESULT = "CalculateResult";

    public Attribute[] getAttributes(Object obj)
    {

```

```

        return null;
    }

    public void serialize(OutputStream os, Object obj) throws IOException
    {
        CalculateResponse jt = (CalculateResponse) obj;
        os.writeObject(jt.getCalculateResult(), _CALCULATERESULT);
    }

    public Object deserialize(InputStream is, Class javaType)
        throws IOException
    {
        if (javaType != CalculateResponse.class)
            throw new
                ServiceException("can't deserialize " + javaType.getName());

        try {
            // instantiate the object
            CalculateResponse jt = (CalculateResponse) javaType.newInstance();
            try {
                // read elements
                jt.setCalculateResult((java.lang.String)is.readObject(java.lang.String.class,
                    _CALCULATERESULT));
            } catch (java.io.EOFException eofExc) {}

            return jt;
        } catch (Exception ex) {
            if (ex instanceof IOException)
                throw (IOException) ex;
            throw new ServiceException(ex);
        }
    }

    public java.lang.String getMechanismType() { return null; }
}

```

CalculateResponseHolder.java

This is the Holder class required by JAX-RPC to implement type mapping support for CalculateResponse. Note that this class is generated in the **holders** subdirectory.

```

// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Jun 04 11:36:21 EDT 2002

package com.exsamp.net.holders;

import com.exsamp.net.CalculateResponse;

```

```
public final class CalculateResponseHolder implements javax.xml.rpc.holders.Holder
{
    public com.exsamp.net.CalculateResponse value;

    public CalculateResponseHolder() { }

    public CalculateResponseHolder(com.exsamp.net.CalculateResponse val)
    {
        value = val;
    }
}
```

autoloan.asmx.xmlrpc.type.mappings

The settings specified in this file tell jBroker Web how to configure the type mappings for Calculate and CalculateResponse. These mappings apply when data is converted from XML to Java or vice versa.

Since the generated stub and service classes automatically configure the mappings, this mappings file is not typically needed. It is provided for special situations (such as when you want to override a mapping).

The mappings file is generated in the base directory of the source tree (**src**).

```
Calculate=com.exsamp.net.Calculate com.exsamp.net.CalculateMarshaler
com.exsamp.net.CalculateMarshaler http://circle24.com/webservices/ Calculate none
CalculateResponse=com.exsamp.net.CalculateResponse
com.exsamp.net.CalculateResponseMarshaler com.exsamp.net.CalculateResponseMarshaler
http://circle24.com/webservices/ CalculateResponse none
```

Editing the generated files

Follow these guidelines when editing the files generated by the Web Service Wizard:

Guideline	Details
File you must edit	<ul style="list-style-type: none"> • <code>xxxClient.java</code>
Files you should not edit	<ul style="list-style-type: none"> • <code>xxxService.java</code> • <code>xxxServiceImpl.java</code> • <code>xxx_Stub.java</code>

It's OK to edit any of the other generated files, but not typically required.

Editing the `xxxClient.java` file

Before using the generated `xxxClient.java` file, you:

- **Must edit the `process()` method** to call one or more methods of the target Web Service
- **May need to edit the `getRemote()` method** to specify the correct location (binding) for accessing the target Web Service

`process()` method

The `process()` method is where the generated client application calls methods of the Web Service. Here you'll find commented code for calling each method defined in the generated remote interface and displaying return values on the console. For example:

```
public void process(String[] args) throws Exception
{
    AutoloanSoap remote = getRemote(args);

    // The following code has been generated for your testing convenience. In
    // order to successfully test your Web Service, you must uncomment one or
    // more of these lines and supply meaningful arguments where necessary.
    // Once you have modified the test method(s) below, compile this class and
    // execute it from a command line with your class path set appropriately.

    // System.out.println("Test Result = " + remote.calculate(com.exsamp.net.Calculate));
}
```

You need to modify this code as follows:

1. **Uncomment one or more method calls** you want to execute.
2. **Provide appropriate arguments** for each method call, either as hardcoded values or as parameters to be furnished at runtime. For runtime arguments, you may also want to add code that validates the values supplied.
3. **Check the return data type** to make sure it can be converted using `toString()`. If not, use an alternative to `System.out.println` for displaying the data returned.

Here's what the line with the `calculate()` method call looks like after editing:

```
System.out.println("Autoloan Web Service\n " +
    "Loan input data:\n    24 months, 8%, $15000\n " +
    "Output from the Web Service:\n    " +
    remote.calculate(new com.exsamp.net.Calculate(24, 8, 15000));
```

getRemote() method

This section explains the basic use of the `getRemote()` method and how to modify it when you need to specify binding information.

Basic use The `getRemote()` method is where the generated client application obtains the remote object to handle its method calls to the Web Service. That remote object is an instance of the generated stub class (`xxx_Stub`). To create the stub instance, `getRemote()` does the following:

1. **Instantiates the Service object** (from the service interface and implementation classes, `xxxService` and `xxxServiceImpl`) via JNDI lookup
2. **Calls a method** that the Service object provides (in the service interface) to get the stub

Here's an example of the typical code generated for `getRemote()`. Normally, you don't need to edit it:

```
public AutoloanSoap getRemote(String[] args) throws Exception
{
    InitialContext ctx = new InitialContext();

    String lookup = "xmlrpc:soap:com.exsamp.net.Autoloan";
    Autoloan service = (Autoloan)ctx.lookup(lookup);
    AutoloanSoap remote = (AutoloanSoap)service.getAutoloanSoap();

    return remote;
}
```

Specifying binding information The wizard includes the binding information for your target Web Service in the generated stub class (*xxx_Stub.java*) and service implementation class (*xxxServiceImpl.java*). The binding provides the **service endpoint address** where the Web Service can be accessed. In a WSDL file, this address is the URL in the **soap:address location** element.

As an alternative, you can specify the binding to use when creating the stub instance in the `getRemote()` method. This enables you to override the binding in the stub class (such as when the Web Service has moved to a new location). You just need to add a line of code to set the address property for the stub:

```
public AutoloanSoap getRemote(String[] args) throws Exception
{
    InitialContext ctx = new InitialContext();

    String lookup = "xmlrpc:soap:com.exsamp.net.Autoloan";
    Autoloan service = (Autoloan)ctx.lookup(lookup);
    AutoloanSoap remote = (AutoloanSoap)service.getAutoloanSoap();

    ((javax.xml.rpc.Stub)remote)._setProperty("javax.xml.rpc.service.endpoint.address",
        "http://upload.eraserver.net/circle24/autoloan.asmx");

    return remote;
}
```

Using the generated files

How you use the Web Service consumer code that you have at this point depends on the nature of the application you're developing. Sometimes you might want to enhance the generated *xxxClient.java* file and include it in your application. At other times you may just copy syntax from *xxxClient.java* into your own classes. But in either case, you'll always need the generated remote interface, service, and stub files.

Before you start any application-specific coding, it's a good idea to test the basic *xxxClient* to make sure your consumer code works as expected. You'll first need to build your project to compile the source files. Then you can run *xxxClient* as described in the next section.

Running the consumer program

The generated Web Service consumer program `xxxClient` is a standard Java application. You can run it in either of these ways:

- From Workbench
- From a command line

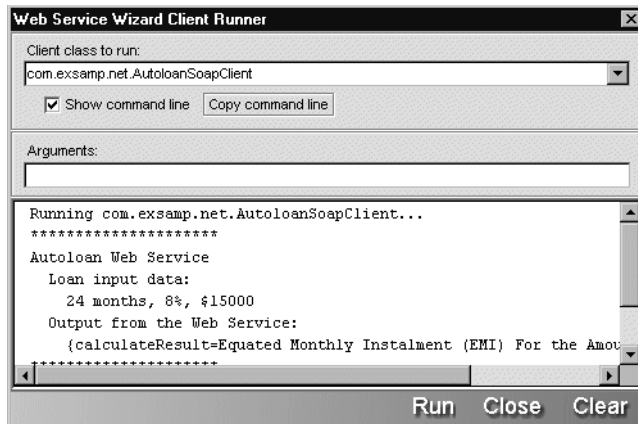
From Workbench

To help you test your generated client quickly and easily, Workbench provides the **Web Service Wizard Client Runner**. This facility lists the client applications in your current project and lets you select one to execute. For each run, it automatically sets the classpath to include all required files and lets you supply command-line arguments.

➤ To use the Client Runner:

1. Open the **project** that contains the compiled client class you want to run.
2. Select **Project>Run Web Service Client Class** to display the Client Runner window.
3. Select a client from the **Client class to run** dropdown.
This dropdown lists every compiled class in your project that has a `main()` method.
4. Check **Show command line** if you want to:
 - See the complete command line that the Client Runner uses to execute your client (it will appear in the display console portion of the window after you click Run)
 - Optionally copy that command line to the system clipboard by clicking **Copy command line** (after a run)
5. Type any command-line **Arguments** required by your client (use a space to separate each argument).
6. Click **Run** to execute your client and see its output in the display console portion of the window.

For example, here's what it looks like to execute the generated `AutoloanSoapClient` class using the Client Runner:



When `AutoloanSoapClient` runs, it calls the `calculate()` method of the `Autoloan Web Service` and passes a `Calculate` object containing loan data (term, rate, amount). The `calculate()` method returns a `CalculateResponse` object containing a string of payment information, which `AutoloanSoapClient` displays on the screen:

```
Running com.exsamp.net.AutoloanSoapClient...
*****
Autoloan Web Service
Loan input data:
  24 months, 8%, $15000
Output from the Web Service:
  {calculateResult=Equated Monthly Instalment (EMI) For the Amount $15000 is $678}
*****
```

From a command line

You can also execute the generated client from the command prompt of your operating system. Doing so demands that you set the classpath to include all required files (such as the generated consumer classes, `jbroker-web.jar`, and so on).

The recommended approach is to use the `Web Service Wizard Client Runner` to display and copy the command line for your client (as described in the previous section). Then you can paste that line to your command prompt and run it.

If you plan to run the client on other computers (beyond your development machine), make sure they have access to all of the files listed in this command line.

Index

A

- application clients
 - about 19, 65
 - API usage 68
 - classpaths 74
 - coding classes 68
 - compiling 74
 - containers 67, 86
 - creating 65, 68
 - creating a client archive 79
 - deploying 65, 80, 83
 - deployment descriptors 77
 - deployment documents 80
 - designing 65
 - example 69
 - in J2EE modules 24
 - JNDI namespace 68
 - life cycle 67
 - manifest files 75
 - packaging into an archive 75
 - running 65, 86
- archives
 - application clients in 66
 - creating 8
 - deploying 9
 - deploying Web Services as WAR files 117
 - deployment descriptors 24
 - directory structure considerations 4
 - EJBs in 87, 95
 - J2EE 24
 - JavaServer Pages in 38
 - servlets in 51
 - validating 8

B

- bindings
 - from consumers to Web Services 195

C

- classpaths
 - for application clients 74
- Client Runner facility
 - for testing Web Service consumers 198
- consumers
 - see Web Services
- containers
 - see J2EE

D

- deployment descriptors
 - about 24
 - creating 40
 - EJB JARs 95
 - for application client archives 68
 - for Web archives 40
 - in J2EE application clients 77
 - modifying 7
- deployment documents
 - about 9
 - application clients 80
 - EJBs 97
 - for Web archives 42
- deployment plans
 - about 9

E

- Electronic Business XML (ebXML)
 - see Web Services
- Enterprise JavaBeans (EJBs)
 - about 19, 87
 - containers 87
 - creating 87, 92
 - deploying 87, 97
 - designing 87
 - entity beans 19
 - home interfaces 92

- implementation classes 92
- JNDI lookup 97
- message-driven beans 19
- packaging in an archive 95
- remote interfaces 92
- running 87, 97
- session beans 19
- tips for designing applications 100

J

J2EE

- about 15, 26
- application clients 65
- architecture 21, 66
- archives 3
- Blueprints 27
- client tier 21
- components 13, 17
- containers 21, 32, 49, 67, 87
- creating JavaServer Pages 31
- creating servlets 49
- data access services 20
- deployment descriptors 75
- deployment services 20
- designing applications 1, 21, 26
- developing applications 3
- Enterprise Information System tier 21
- file support 20
- Internet protocols 20
- manifest files 75
- messaging services 20
- META-INF directories 75
- middle tier 21
- Model-View-Controller (MVC) model 26
- modules 24
- naming services 20
- OMG protocols 20
- RMI protocols 20
- roles 25
- security services 20
- technologies 17
- testing and debugging applications 10
- transaction services 20
- WEB-INF directories 38

- web.xml 40
- Workbench support for 28

JavaServer Pages

- about 18, 31
- creating 31, 38
- deploying 42
- designing 38
- example 34
- in Web applications 32
- in Web archives 38, 40
- in Web modules 24
- mixing HTML and Java 37
- running 46
- servlets and JSP pages 33

JAX-RPC

- about 116
- generating consumers for 167
- support for 119

JBroker Web

- about 117
- packaging jbroker-web.jar with generated consumers 168
- packaging jbroker-web.jar with generated Web Services 120

M

- manifest files
 - see J2EE, application clients
- META-INF directories
 - see J2EE
- Microsoft .NET
 - about 116
 - generating consumers for 167
- Model-View-Controller (MVC)
 - see J2EE

P

- project files
 - creating 5
 - saving 7

projects

- adding source files and directories 5, 8
- compiling, building, and archiving 8
- creating 5
- creating components 5
- creating enterprise archive (EAR) projects 5
- deploying 9
- designing 3
- organizing 4
- supporting team development 7

providers

- see Web Services

R

registries

- see Web Services

S

servers

- creating profiles 9

servlets

- about 18, 49
- containers 49
- creating 49, 52
- deploying 63
- designing 49
- event listeners 18
- example 52
- filters 18
- generating an HTTP response 56
- in Web modules 24
- JavaServer Pages and servlets 33, 51
- life cycle 49
- packaging into a Web archive 62
- processing HTTP requests 55
- reading HTML form data 55
- reading HTTP request header information 56
- running 63
- specifying init() and destroy() methods 61
- specifying the HTTP document content 59

SilverStream eXtend Application Server

- deploying Web archives to 43
- deployment documents 80
- deployment plans 9
- deployment plans for EJB JARs 97
- SilverJ2EEClient 67, 86

skeleton model

- for Web Services 135

SOAP (Simple Object Access Protocol)

- see Web Services

source files

- adding to projects 4, 5
- creating components 5
- directory structure considerations 4
- editing 5

SPF files

- see project files

subprojects

- see projects

T

tie model

- for Web Services 135

U

UDDI

- see Web Services

W

Web Service consumers

- binding style 173
- binding to services 195
- generating 167
- J2EE 167
- Microsoft .NET 167
- packaging jbroker-web.jar with 168
- running 198
- type mapping 173
- using JAX-RPC 167

- Web Service Wizard
 - Client Runner facility 198
 - generating consumers with 167
 - generating Web Services with 119
 - implementation model choices 135
 - using jbroker-web.jar with 120, 168
- Web Services
 - about 109, 115
 - browsing registries 114
 - consumers 110
 - creating components 111, 117
 - designing applications 1
 - developing applications 3
 - ebXML 114, 115
 - generating 119
 - HTTP 109, 111, 113, 115, 117
 - implementation models for 135
 - JAX-RPC 116
 - JAX-RPC support 119
 - jBroker Web 117
 - local registries 115
 - Microsoft .NET 116
 - packaging jbroker-web.jar with 120
 - providers 110, 111
 - publishing to registries 112, 114
 - registries 110, 112, 114, 115
 - SOAP 109, 111, 113, 115, 117
 - testing and debugging applications 10
 - tools provided in Workbench 116
 - UDDI 114, 115
 - using 113, 167
 - WSDL 111, 113, 115
- WEB-INF directories
 - see J2EE
- Workbench
 - J2EE support 28
- WSDL (Web Services Description Language)
 - see Web Services