Web Services Developer's Guide

VERSION 9

Borland®
JBuilder®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JBE0090WW21003websvcs 2E1R0503
0304050607-9 8 7 6 5 4 3 2 1
PDF

Contents

Tables

Figures

Tutorials

Chapter

1

Introduction

This is a feature of JBuilder Enterprise

The *Web Services Developer's Guide* explains how to use the JBuilder web services features to create, browse, consume, and publish web services.

The *Web Services Developer's Guide* contains the following chapters:

- Chapter 2, "Introducing web services"

  Provides an overview of web services and the web services features available in JBuilder.

- Chapter 3, "Configuring projects for web services"

  Explains how to create a web services server to host a web service using the Web Services Configuration wizard and how to run the web services server in the JBuilder IDE.

- Chapter 4, "Monitoring SOAP messages"

  Describes how to monitor SOAP messages using tools provided by the web services toolkits.

- Chapter 5, "Working with WSDL"

  Gives an overview of the Web Services Description Language (WSDL) and how it's used in web services.

- Chapter 6, "Developing EJBs as web services"

  Provides an overview of how to develop Enterprise JavaBeans as web services.

- Chapter 7, "Using the Apache Axis toolkit"

  Describes how to use the Apache Axis toolkit to develop web services. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

- Chapter 8, "Using the WebLogic toolkit"

  Describes how to use the WebLogic toolkit to develop and test web services.

- Chapter 9, "Using the Apache SOAP 2 toolkit"

  Describes how to use the Apache SOAP 2 toolkit to develop web services. Support for the Apache SOAP 2 toolkit is not provided with the JBuilder WebLogic Edition.

- Chapter 10, "Browsing and publishing web services"

  Describes how to use the Web Services Explorer to browse and publish web services.

- Web services samples

  Web services samples are available in the following JBuilder directories:

  - `samples/webservices/axis`
  - `samples/webservices/weblogic`
  - `thirdparty/xml-axis/java/samples`

- Chapter 11, "Axis tutorials"

  - "Tutorial: Creating a simple web service" on page 11-1

    Explains how to use the Export As A Web Service wizard to export a JavaBean as a web service with the Axis toolkit, exposing selected methods to the web service consumer.

  - "Tutorial: Generating a web service from a WSDL document" on page 11-8

    Explains how to use the Import A Web Service wizard to generate Java classes for a web service with the Axis toolkit, then implement the classes to provide access to AltaVista's BabelFish translation service.

Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

For an explanation of documentation conventions, see "Documentation conventions" on page 1-4.

Documentation conventions

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
`Monospaced type`	Monospaced type represents the following: • text as it appears onscreen • anything you must type, such as "Type `Hello World` in the Title field of the Application wizard." • file names • path names • directory and folder names • commands, such as `SET PATH` • Java code • Java data types, such as `boolean`, `int`, and `long`. • Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events • argument names • field names • Java keywords, such as `void` and `static`
Bold	Bold is used for java tools, bmj (Borland Make for Java), bcj (Borland Compiler for Java), and compiler options. For example: **javac**, **bmj**, **-classpath**.
Italics	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
Keycaps	This typeface indicates a key on your keyboard, such as "Press *Esc* to exit a menu."
[ ]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.
< >	Angle brackets are used to indicate variables in directory paths, command options, and code samples. For example, `<filename>` may be used to indicate where you need to supply a file name (including file extension), and <username> typically indicates that you must provide your user name. When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (< >). For example, you would replace `<filename>` with the name of a file, such as `employee.jds`, and omit the angle brackets. **Note:** Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as <font color=red> and <ejb-jar>. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.

Table 1.1 Typeface and symbol conventions (continued)

Typeface	Meaning
Italics, serif	This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, `<url="jdbc:borland:`*jbuilder*`\\samples\\guestbook.jds">`
...	In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

Table 1.2 Platform conventions

Item	Meaning
Paths	Directory paths in the documentation are indicated with a forward slash (/). For Windows platforms, use a backslash (\).
Home directory	The location of the standard home directory varies by platform and is indicated with a variable, `<home>`. • For UNIX and Linux, the home directory can vary. For example, it could be `/user/<username>` or `/home/<username>` • For Windows NT, the home directory is `C:\Winnt\Profiles\<username>` • For Windows 2000 and XP, the home directory is `C:\Documents and Settings\<username>`
Screen shots	Screen shots reflect the Metal Look & Feel on various platforms.

Developer support and resources

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

Contacting Borland Technical Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support on installation of the Borland product to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at http://www.borland.com/devsupport/, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

Online resources

You can get information from any of these online sources:

World Wide Web	http://www.borland.com/ http://www.borland.com/techpubs/jbuilder/
Electronic newsletters	To subscribe to electronic newsletters, use the online form at: http://www.borland.com/products/newsletters/index.html

World Wide Web

Check www.borland.com/jbuilder regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- http://www.borland.com/jbuilder/ (updated software and other files)
- http://www.borland.com/techpubs/jbuilder/ (updated documentation and other files)
- http://community.borland.com/ (contains our web-based news magazine for developers)

Borland newsgroups

When you register JBuilder you can participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at http://www.borland.com/newsgroups/.

Usenet newsgroups

The following Usenet groups are devoted to Java and related programming issues:

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

Note These newsgroups are maintained by users and are not official Borland sites.

Reporting bugs

If you find what you think may be a bug in the software, please report it to Borland at one of the following sites:

- Support Programs page at http://www.borland.com/devsupport/namerica/. Click the "Reporting Defects" link to bring up the Entry Form.
- Quality Central at http://qc.borland.com. Follow the instructions on the Quality Central page in the "Bugs Reports" section.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email jpgpubs@borland.com. This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

Chapter

2

Introducing web services

This is a feature of JBuilder Enterprise

A web service is a software module performing a discrete task or set of tasks that can be found and invoked over a network including and especially the World Wide Web. The developer can create a client application that invokes a series of web services through remote procedure calls (RPC) or a messaging service to provide some or most of the application's logic. A published web service describes itself so that developers can locate the web service and evaluate its suitability for their needs.

As an example, a company could provide a web service to their customers to check inventory on products before they order. Another example is the Federal Express package tracking service that customers can use to track their shipments.

Web services use SOAP (Simple Object Access Protocol) for the XML payload and uses a transport such as HTTP to carry the SOAP messages back and forth. SOAP messages are actually XML documents that are sent between a web service and the calling application.

Web services can be written in any language and run on any platform. A client of a web service can also be written in any language and run on any platform. So, for example, a client written using Delphi and running on Windows could call a web service written in Java and running on Linux.

Web services architecture

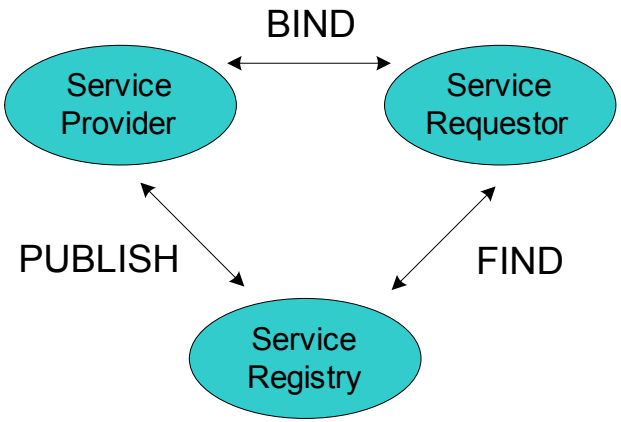
The web services architecture permits the development of web services that encapsulate all levels of business functionality. In other words, a web service can be very simple, such as one that returns the current temperature, or it can be a complex application. The architecture also allows multiple web services to be combined to create new functionality.

The web services architecture has three distinct roles: a provider, a requestor, and a broker. The provider creates the web service and makes it available to clients who want to use it. A requestor is a client application that consumes the web service. The requested web service can also be a client of other web services. The broker, such as a service registry, provides a way for the provider and the requestor of a web service to interact.

The three roles of provider, requestor, and broker interact with each other through the operations of publish, find, and bind. A provider informs the broker about the existence of the web service by using the broker's publish interface to make the service accessible to clients. The information published describes the service and specifies where the service is located. The requestor consults the broker to locate a published web service. With the information it gained from the broker about the web service, the requestor is able to bind, or invoke, the web service.

The following diagram summarizes how the provider, requestor, and broker interact with each other.

Figure 2.1 Web service roles and operations

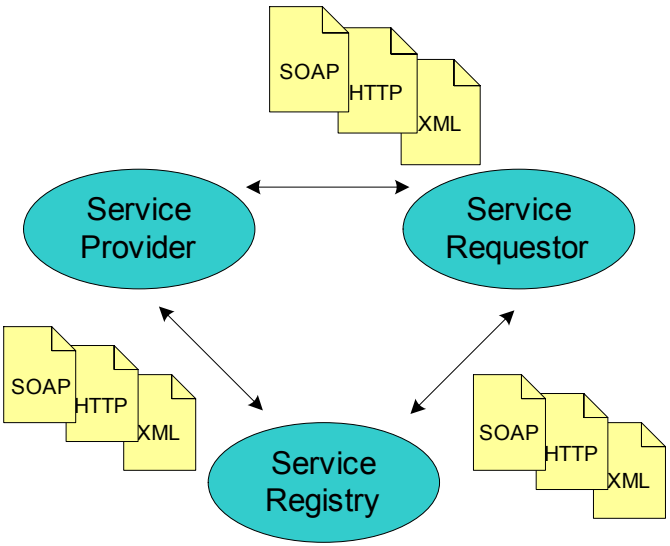


Web services standards

The standards on which web service development is based are evolving technologies. The primary players are SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), UDDI (Universal Description, Discovery and Integration), and WSIL (Web Services Inspection Language).

Simple Object Access Protocol (SOAP)

SOAP is a transport-independent messaging protocol. Each SOAP message is an XML document. SOAP uses one-way messages, although it's possible to combine messages into request-and-response sequences. The SOAP specification defines the format of the XML message but not its content and how it's actually sent. SOAP does, however, specify how SOAP messages are routed over HTTP.



Each SOAP document has a root `<Envelope>` element. The *root element*, the first element in an XML document, contains all the other elements in the document. Within the "envelope" are two parts: a header and a body. The header contains routing or context data. It can be empty. The body contains the actual message. It too can be empty.

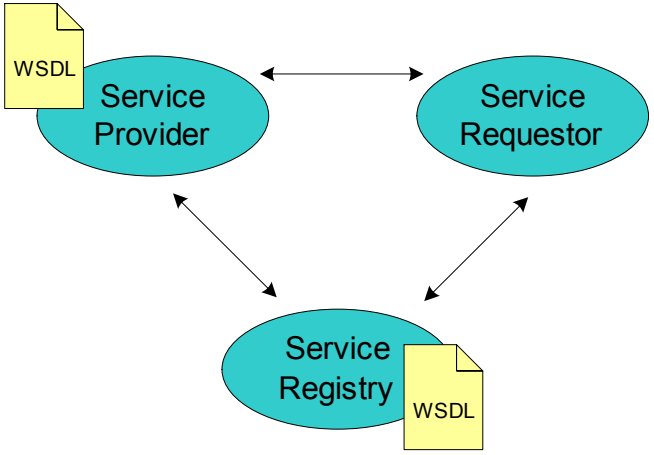
Here is an example of a simple SOAP message sent over HTTP that requests the current stock price of Borland:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "urn:stock-quote-services"

<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <symbol>BORL</symbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For more information about SOAP, start with the SOAP documents on the World Wide Consortium web site at `http://www.w3.org/2002/ws/`. Also visit the Apache SOAP site at `http://xml.apache.org/soap/`.

Web Services Description Language (WSDL)

A web service is useless unless others can find out what it does and how to call it. Developers must know enough information about a web service so they can write a client program that calls it. WSDL is an XML-based language used to define web services and describe how to access them. Specifically, it describes the data and message contracts a web service offers. By examining a web service's WSDL document, developers know what methods are available and how to call them using the proper parameters.

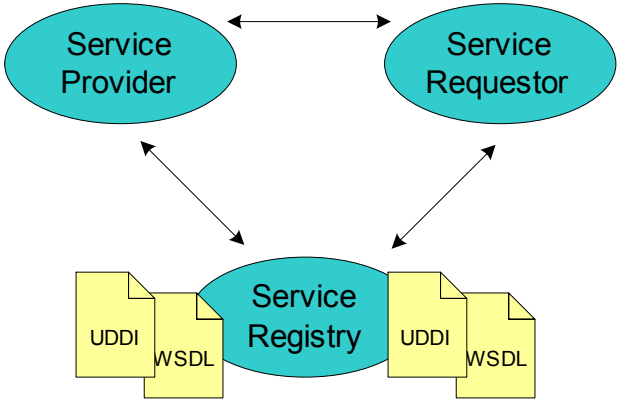


For in-depth information about WSDL, see the Web Services Description Language 1.1 specification at http://www.w3.org/TR/wsdl on the World Wide Web Consortium web site.

Universal Description, Discovery and Integration (UDDI)

UDDI is an evolving standard for describing, publishing, and discovering the web services that a business provides. It's a specification for a distributed registry of information on web services. Once a web service is developed and a WSDL document describing it is created, there needs to be a way to get the WSDL information into the hands of the users who want to use the web service it describes. When a web service is published in a UDDI registry, potential users have a way to look up and learn about the web service's existence.

The content of a UDDI registry is similar to a telephone directory. In the "white pages" section of the registry is information such as the name, address, and telephone number of the business that offers one or more web services. The "yellow pages" section identifies the business type and categorizes it by industry. The "green pages" section provides the data about the web services the business offers.

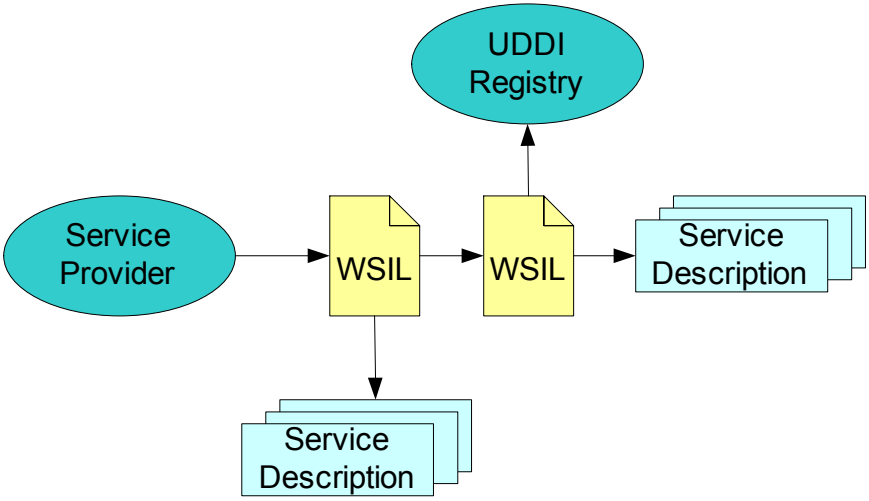


To read more about UDDI, see the UDDI.org web site at http://www.uddi.org.

Web Services Inspection Language (WSIL)

WSIL, like UDDI, provides a method of service discovery for web services. Unlike UDDI, WSIL uses a de-centralized, distributed model, rather than a centralized model. WSIL documents, which are essentially pointers to lists of services, allow consumers of web services to browse available services on web sites. The WSIL specification provides standards for using XML-formatted documents to inspect a site for services and a set

of rules for how the information is made available. A WSIL document gathers multiple references to pre-existing service description documents in one document. The WSIL document is then hosted by the provider of the service, so consumers can find out about available services.



For more information on the Web Services Inspection Language (WS-Inspection) 1.0 specification, see `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html`.

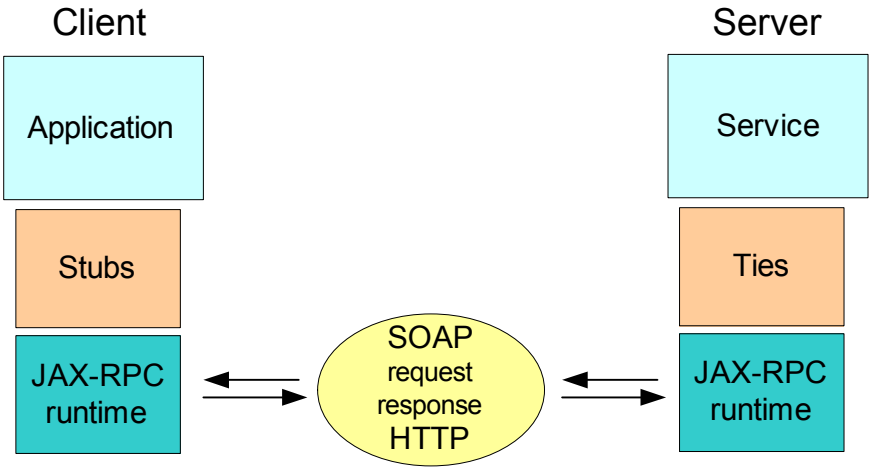
Java APIs for XML-based Remote Procedure Call (JAX-RPC)

JAX-RPC defines Java APIs that Java developers can use in their applications to develop and consume web services. Using JAX-RPC, a Java client can consume a web service on a remote server over the Internet, even if the service is written in another language and running on a different platform. A JAX-RPC service can also be consumed by non-Java clients.

JAX-RPC uses an XML-messaging protocol, such as SOAP, to transmit a remote procedure call over a network. For example, a web service that returns a stock quote would receive a SOAP HTTP request containing a method call from the client. Using JAX-RPC, the service extracts the method call from the SOAP message, translates it into a method call, and invokes it. Then the service uses JAX-RPC to convert the method response back into SOAP and send the results back to the client. The client receives the SOAP message and uses JAX-RPC to translate it into a response.

The JAX-RPC runtime generates stubs and ties, which are classes that allow communication between the client and the service. A stub, which is on the client side, is a local object that represents a remote service and acts

as a proxy for the service. A tie, which is on the server side, acts as a proxy on the server.



For more information on JAX-RPC, see `http://java.sun.com/xml/jaxrpc/index.html` and the JAX-RPC tutorial at `http://java.sun.com/webservices/docs/1.0/tutorial/doc/JAXRPC.html`.

JBuilder and web services

Although it's useful to learn about the technologies behind web services, you don't have to do such things as actually create the SOAP messages and WSDL descriptions yourself. JBuilder can do these things for you.

JBuilder uses the web services toolkit you select when configuring the project for its SOAP support. JBuilder can generate the WSDL document for a web service you've created. It can also take an existing WSDL for a web service and create the Java class files or EJBs, so you can create a client that invokes the web service. Another option is to use the generated Java code to implement the web service yourself. JBuilder can also quickly create a web application that hosts a web services server. Tools are also available for testing the deployed web services and monitoring SOAP messages between the client and the server. These tools vary by toolkit. JBuilder also includes the Web Services Explorer for searching for web services that fit your needs, as well as publishing to a UDDI registry.

To see a sample JavaServer Page client that invokes a web service to translate words from one language to another, open the `BasicWebService.jpx` project in the JBuilder `samples/webservices/axis` directory.

Consuming and creating web services with JBuilder wizards

JBuilder provides wizards for consuming and creating web services. The Import A Web Service wizard generates Java classes from an existing WSDL document or EAR. You can then implement these classes to consume the service specified in the WSDL. You can also create server-side classes for a web service or implement a web service as an Enterprise JavaBean (EJB) from the imported WSDL. The Export As A Web Service wizard generates a WSDL document from an existing Java class and exposes selected methods of the class as a web service. The WebLogic toolkit has an additional wizard, the Export As An Asynchronous wizard, which is available for WebLogic Server 8.1. You can use this wizard to create asynchronous web services. Available web services features vary by web services toolkit.

These wizards are available on the Web Services page of the object gallery (File | New). They are also available on the context menu in the project pane for the appropriate nodes.

Developing EJBs as web services varies according to the toolkit selected. For more information on EJBs, see the appropriate toolkit chapter.

Web services samples

Web services samples are available in the following JBuilder directories:

- `samples/webservices/axis`
- `samples/webservices/weblogic`
- `thirdparty/xml-axis/java/samples`

Supported enterprise application servers

The JBuilder WebLogic Edition provides support for WebLogic Servers only

The JBuilder web services features are supported on the following enterprise application servers:

- WebLogic toolkit: WebLogic Server 7.0 SP2 and 8.1
- Axis and SOAP toolkits:
  - Borland Enterprise Server 5.1.x, 5.2, and 5.2.1
  - WebLogic Server 7.0 SP2 and 8.1
  - WebSphere Application Server 4.0 AES/AE and 5.0
  - Sybase 4.1.3 and 4.2

For information about how to use these servers with the JBuilder web services features, see Appendix A, "Modifying enterprise application server settings for web services."

Chapter

3

Configuring projects for web services

This is a feature of JBuilder Enterprise

Before you can work with web services in JBuilder, you need to configure the project for web services. The Web Services Configuration wizard creates a configuration for hosting a web service. This involves creating a WebApp and configuring it with the selected web service toolkit library. On application servers, the wizard also creates an eargrp node and links it to the WebApp. In addition, the wizard creates a custom, server toolkit-specific runtime configuration reflecting the archive selections. The configuration is sensitive to the server configuration and allows the choice of valid toolkits for a given configuration which make a server web service capable. When invoking other web services wizards, JBuilder checks if the project is configured for web services and, if it isn't configured, invokes the Web Services Configuration wizard first.

Working with the Web Services Configuration wizard

JBuilder provides the Web Services Configuration wizard for quickly creating a WebApp to host a web service with the appropriate deployment files. The Web Services Configuration wizard creates a SOAP implementation for the project based upon the selected toolkit. It also creates a Web Services Server runtime configuration for deploying and running the service or an implementation of the service. Once a web service is deployed to the web services server, the web service can receive and send SOAP messages to and from client applications. For more information on SOAP, see Chapter 4, "Monitoring SOAP messages."

The web service must be hosted by a web application. If your project doesn't have an existing web application, you can create one from the

Web Services Configuration wizard. For more information on web applications, see "Working with WebApps and WAR files" in the *Web Application Developer's Guide*.

The Web Services Configuration wizard generates the following nodes and files and adds them to your WebApp. The contents of the nodes vary according to the toolkit selected.

- Web Service Components node

  The name of this node is dependent upon the toolkit selected in the wizard. This node contains two child nodes:

  - The EJB-based Services node contains any Enterprise JavaBean (EJB) deployment files. By default, all stateless session beans with business methods in the remote interface are automatically deployed. The deployment information is generated in one deployment file per EJB module. At build time the EJB classes in the deployment files are deployed to the server.
  - The Java-based Services node contains any Java class deployment files.
  - The Async Component-based Service node contains any Java class deployment files for classes that are exported as asynchronous services. This is a feature of the WebLogic toolkit and WebLogic Server 8.1.

- Deployment Descriptors node

  This node contains any deployment descriptors, which varies according to the selected toolkit. For example, if the Axis or SOAP toolkits are selected, the wizard adds the appropriate SOAP information to the `web.xml` file that describes the WebApp deployment.

- Root Directory node

  The wizard adds contents to this node according to the toolkit selected.

To open the Web Services Configuration wizard,

1. Choose File | New to open the object gallery.
2. Click the Web Services tab and double-click the Web Services Configuration icon.
3. Choose an EAR in the project or click New to create an EAR. This is only required if you're using an enterprise application server which requires an EAR.
4. Choose a WebApp in the project to host the service or click New to create a new WebApp.
5. Select a toolkit for the project.

6 Click Next to see that the wizard creates a Web Services Server runtime configuration for the project.

7 Click Finish to configure the project for web services.

Selecting an EAR

If the project is configured to use an application server, the Web Services Configuration wizard requires you to select or create an EAR. The EAR is automatically configured by the wizard to include the WebApp archive and is automatically updated at build time to include any referenced EJB JARs used in the web service.

Naming the WebApp

The web context, also known as the contextURI, is the name of the WebApp, the WebApp's directory, and the WebApp archive file.

Note The name of the WebApp is important, because it's part of the address for invoking the web service. An example of an address to invoke a web service running on your local server might be `http://localhost:8080/<web context>/<serviceURI>`. The web context is used in the SOAP address in the WSDL document. The serviceURI is the Uniform Resource Identifier of the service.

For more information about WebApps, see "Working with WebApps and WAR files" in the *Web Application Developer's Guide*.

Selecting a web services toolkit

Support for the Apache Axis and Apache SOAP 2 toolkits is not provided with the JBuilder WebLogic Edition.

The Toolkit field in the Web Services Configuration wizard gives you a choice of all registered web services toolkits. The Toolkit field could contain additional toolkit plugins if they are available on your machine.

The Copy Admin/Console To WebApp option copies files into the Root Directory folder so that you can use the toolkits' UI to administer the web services server. Check this option if you would like these files added to the WebApp. The Copy Admin/Console To WebApp option could potentially be disabled if you are using a toolkit that doesn't have an admin/console. Both the Apache Axis toolkit and the Apache SOAP 2 toolkit have an admin/console, so this option is enabled for both of them.

Apache Axis toolkit

Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

The Apache Axis toolkit is an open source implementation of SOAP, the next generation of Apache SOAP 2.0. Axis is a rewrite of Apache SOAP 2.0 that uses SAX instead of DOM. It's a more modular, flexible, and a higher performance implementation of SOAP than Apache SOAP 2.0. The

Apache Axis toolkit is JAX-RPC (Java APIs for XML-based Remote Procedure Call) compliant and supports WSDL 1.1.

The Apache Axis toolkit generates files for administering, listing deployed services, and validating. For more information on these files, see the toolkit documentation.

Note The Axis toolkit combines each deployment file (`deploy.wsdd`) in your project into a `server-config.wsdd`. Any manual edits to the deployment files are overwritten by the toolkit. If you edit `server-config.wsdd`, turn off the Regenerate Deployment option on the Web Services tab of the Build page of Project Properties or `server-config.wsdd` will be overwritten by the toolkit when you build the project. See "Setting build options" on page 3-6 for more information.

See also

- Chapter 7, "Using the Apache Axis toolkit"
- Axis at `http://xml.apache.org/axis/`
- Apache Axis documentation at `<jbuilder>/thirdparty/xml-axis/java/docs/index.html`

WebLogic toolkit

WebLogic Server 7.x and 8.1 have built-in web services capability. The WebLogic toolkit is available if your project specifies WebLogic Server 7.x or 8.1 as the server. WebLogic supports WSDL 1.1 and is JAX-RPC compliant.

See also

- Chapter 8, "Using the WebLogic toolkit"
- "Programming WebLogic Web Services" at `http://edocs.bea.com/wls/docs70/webserv/index.html` for version 7.0 or `http://edocs.bea.com/wls/docs81/webserv/index.html` for version 8.1

Apache SOAP 2 toolkit

Support for the Apache SOAP 2 toolkit is not provided with the JBuilder WebLogic Edition.

Apache SOAP 2 toolkit is an open-source implementation of SOAP 1.1 developed by the Apache SOAP community. This implementation of SOAP uses DOM.

Caution Note that Apache SOAP 2 is an older version of Axis and does **not** support JAX-RPC or WSDL. Due to these limitations, it's recommended that you use Axis instead. If you're using the Apache SOAP 2 toolkit, you need to make modifications in the JBuilder wizards and do additional hand coding. For more information, see Chapter 9, "Using the Apache SOAP 2 toolkit."

See also

- Apache SOAP at `http://ws.apache.org/soap/`
- Apache SOAP 2 documentation at `http://ws.apache.org/soap/`

Defining a runtime configuration for the web services server

On the last page of the Web Services Configuration wizard, you can optionally define a runtime configuration for the web services server. The web services server requires a Server type runtime configuration for the server to run.

If no Server type runtime configuration exists and you don't choose to define one on this page, a runtime configuration called Web Services Server is automatically added for you. This runtime configuration uses the default Server configuration for the project as defined on the Server page of the Project Properties dialog box (Project | Project Properties | Server). You can edit the Web Services Server configuration on the Run page of the Project Properties dialog box (Project | Project Properties | Run).

For more information on runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*.

Examining the WebApp node

After creating a web services server with the Web Services Configuration wizard, you can expand the WebApp node in the project pane to view the child nodes the wizard added. You'll see a toolkit node, a Deployment Descriptors node, and a Root Directory node. If you added files to administer the WebApp in the Root Directory, expand that node to see the files.

For more information about WebApps, see "Working with WebApps and WAR files" in the *Web Application Developer's Guide*.

Starting the web services server

To run the web services server, you must have a Server runtime configuration which uses an appropriate web server. The Web Services Configuration wizard adds the Web Services Server runtime configuration to your project automatically. For more information about runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*.

Once you've completed the Web Services Configuration wizard, you can start the web services server by choosing Run | Run Project when Web

Services Server runtime configuration is the default runtime configuration or by clicking the small arrow next to the Run icon on the main toolbar and selecting the Web Services Server runtime configuration from the list. You can view the server's progress in the message pane. If you're using the Axis toolkit, you can also right-click any servlet, JSP, or HTML file in the project pane and select Web Run Using Web Services Server from the context menu.

If you chose to copy the toolkit's Admin/console files, if available, to the WebApp node when you created the web services server, you can now expand the Root Directory node, right-click the `index.html` file and select Web Run from the context menu to reach the admin UI for the Axis toolkit. The file to Web Run for the Apache Soap 2 toolkit is `admin/index.html`. With the default IDE settings, Web Run when the web server is already running will cause the file to be accessed off the web server.

Setting build options

The Build page of the Project Properties dialog box has a build option for controlling deployment at build time.

To access the build option:

1. Choose Project | Project Properties to open the Project Properties dialog box.
2. Choose the Build tab and click the Web Services tab.

The Regenerate Deployment option is set by default. If this option is set, deployment files are regenerated whenever the project is built. For more details on this option, choose the Help button on the Web Services tab of the Build page.

Chapter

4

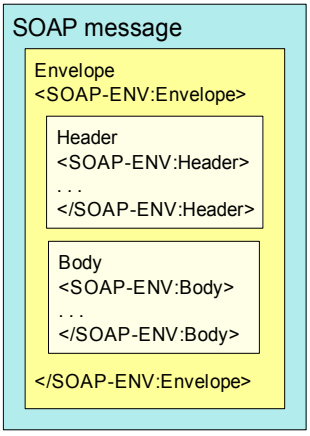
Monitoring SOAP messages

This is a feature of JBuilder Enterprise

Web services use the Simple Object Access Protocol (SOAP), which provides a messaging protocol for clients and servers to send messages back and forth. SOAP is a transport-independent messaging protocol that allows access to remote objects. SOAP uses a XML as the messaging protocol and a transport layer such as HTTP. SOAP messages are one-way messages, although it's possible to combine messages into request-and-response sequences. The SOAP specification defines the format of the XML message, but not its content and how it's actually sent. SOAP does, however, specify how SOAP messages are routed over HTTP.

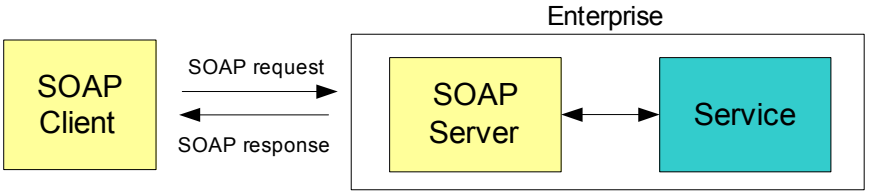
Each SOAP message has a root `<Envelope>` element. Within the "envelope" are two parts: a header and a body. The envelope and body are required elements in SOAP messages, while the header is optional. The header contains routing or context data and can be empty. The body, which contains the actual message, can also be empty.

Figure 4.1 Soap message



SOAP messages are sent and received by SOAP clients and web services servers. The SOAP client generates and sends SOAP requests to the SOAP server over HTTP. The SOAP server receives these requests and generates the appropriate response over HTTP to the client.

Figure 4.2 Soap architecture



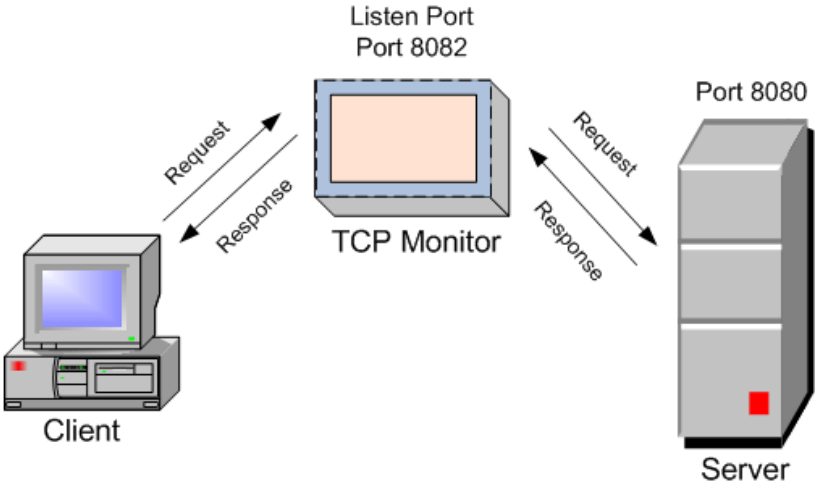
Some of the web services toolkits include tools for monitoring the SOAP messages between the client and the server. For example, the WebLogic toolkit includes the Web Services Console. The Axis toolkit includes the Axis TCP Monitor.

Using the Axis TCP Monitor

This is a feature of JBuilder Enterprise and the Axis toolkit. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

The Axis TCP Monitor, which is available with the Axis toolkit, allows you to monitor the SOAP envelopes as they're transported between the client and the server. TCP, Transmission Control Protocol, is the most common Internet transport layer protocol. The TCP Monitor actually sits between the SOAP client and server. The client sends its request to the TCP Monitor, which then forwards it on to the server. Responses from the server are in turn sent to the TCP Monitor and then to the client. You can monitor these messages locally to test the service or listen over a connection.

Figure 4.3 TCP Monitor



The TCP Monitor supports these features:

- Adding multiple listener ports
- Editing SOAP messages
- Resending SOAP messages
- Viewing a history of requests and responses
- Saving requests and responses to a file

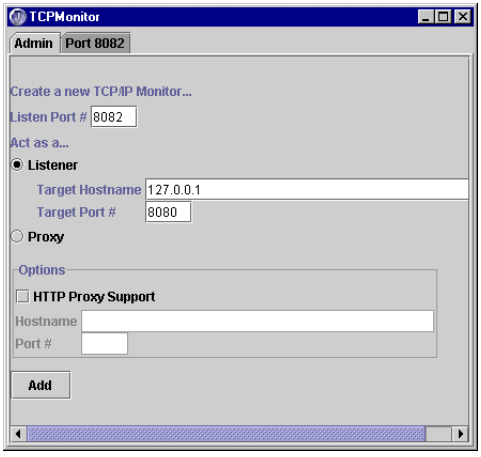
You can set up the TCP Monitor to act as a listener or use a proxy if you're behind a firewall. The various web services toolkits have different levels of support for proxy servers and different usages. Consult the toolkit documentation for information on how to set up for use with a proxy server.

Creating a new TCP/IP Monitor

By default, the TCP monitor sets the client port to 8082 and the server port to 8080.

To create a new port to listen to, complete these steps:

1. Choose Tools | TCP Monitor to open the TCP Monitor.
2. Choose the Admin tab.
3. Enter a port number to listen on, such as 8082 for localhost. This should be a port that won't be in use by the service.
4. Choose Act As A Listener.
5. Enter the Target Host Name for the destination server that you want to monitor, such as 127.0.0.1 for localhost. If you enter a name instead of an address, such as services.xmethods.net, be sure to remove the protocol from the address (`http://`).
6. Enter the Target Port Number for the server, such as 8080. This is the destination port that you want to monitor.



7 Click the Add button to add this new TCP/IP Monitor. A new page is added.

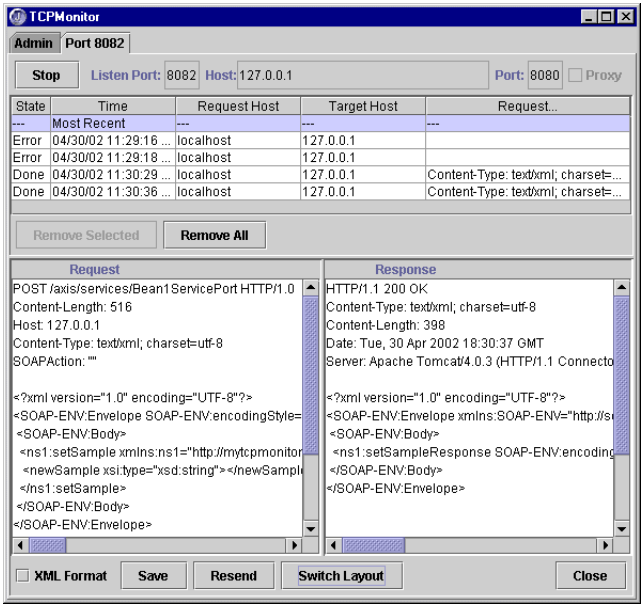
8 Choose the new Port tab to monitor the messages.

9 Modify the client to send its messages to and receive its messages from the TCP Monitor on the listen port. Change the address and port number in your code but don't remove the context and service target in the address. For example, the context and service target in this address `http://localhost:8080/axis/services/rpcrouter` is `axis/services/rpcrouter` and should remain unchanged.

10 Rebuild the client to save the changes.

11 Run the application on the web server.

12 Interact with the web service and view the SOAP requests and responses in the TCP Monitor. You can also make any necessary edits to requests in the TCP Monitor and resend them, as well as save request and response messages to disk in text or XML format.



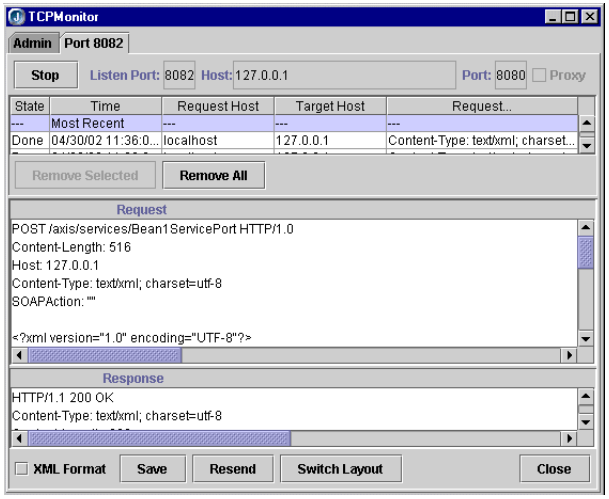
Monitoring a service's SOAP messages

Let's look at a web services sample and monitor the SOAP messages with the TCP Monitor. In this example, you'll create a web service, modify the client's URL to point to the port that the TCP Monitor listens on, rebuild the client, run the service test case, and monitor the messages forwarded between the client and the server by the TCP Monitor.

1. Create a web service as described in "Tutorial: Creating a simple web service" on page 11-1.
2. Open `Bean1ServiceLocator.java` in the `<project name>.generated` package and change the URL for the service port address from the 8080 to 8082. The TCP Monitor is set up by default to listen to messages on port 8082. You want to redirect the client to send the messages to the listening port so the Monitor can receive the messages and send them on to the service.
3. Choose Tools | TCP Monitor to open the TCP Monitor. Notice that the listen port is set to 8082 and the destination port to 8080. These are the default TCP Monitor settings.

Note You can change the default settings at any time. Choose the Stop button and edit the Listen Port, Host, and Port fields. You can also create a new TCP Monitor on the Admin page of the TCP Monitor.

4. Expand the axis node's Root Directory, right-click `index.html`, and choose Web Run Using "Web Services Server" to run the web services server.
5. Right-click the test case, `Bean1ServiceTestCase.java`, and choose Run Test Using Defaults.
6. Return to the TCP Monitor to see the request and response sent from the client and server to the Monitor.



See also

- Step 8 of "Tutorial: Generating a web service from a WSDL document" on page 11-8

Enabling the SOAP Monitor feature on the server side

To enable the Axis SOAP Monitor feature on the server side using Axis handler and the Axis SOAPMonitor Applet, follow these steps:

1 Copy the following into any one of the `deploy.wsdd` files in the project source path above or below the <service> element:

```
<globalConfiguration>
  <requestFlow>
    <handler type="java:org.apache.axis.handlers.JWSHandler">
      <parameter name="scope" value="session"/>
    </handler>
    <handler type="java:org.apache.axis.handlers.JWSHandler">
      <parameter name="scope" value="request"/>
      <parameter name="extension" value=".jwr"/>
    </handler>
    <handler type="java:org.apache.axis.handlers.SOAPMonitorHandler"/>
  </requestFlow>
  <responseFlow>
    <handler type="java:org.apache.axis.handlers.SOAPMonitorHandler"/>
  </responseFlow>
</globalConfiguration>
```

2 Rebuild the project and restart the server or redeploy for the changes to take effect.

3 Click the SOAPMonitor hyperlink on the main page of the Axis Admin Console to launch the applet.

Using the Web Services Console

This is a feature of JBuilder Enterprise and the WebLogic toolkit

Every web service deployed on WebLogic Server has a testing home page, which you can view in the Web Services Console. From the testing home page, you can view the exposed methods and the SOAP requests and responses, test each operation, view the WSDL that describes the service, and copy example code that invokes the service. Before testing the deployed service, be sure that the server has a web browser configured for the Admin Console in the Configure Servers dialog box (Tools | Configure Servers). For more information on configuring a web browser for the server, see "Using JBuilder with BEA WebLogic servers" in *Developing J2EE Applications*. After configuring the web browser, run the server and deploy the service to the server as described in Chapter 8, "Using the WebLogic toolkit."

To open the testing home page for your web service, choose Tools | Web Services Console. The Web Services Console automatically finds the URL for the web service. If multiple ServiceUri names exist in the project, the Console displays a main page that lists all the ServiceUri names, which are hyperlinked to the deployed services. If a single service is deployed or

multiple services with the same ServiceURI name, the Web Services Console displays the WebLogic Web Services Home Page.

The URL for the web service consists of the following:

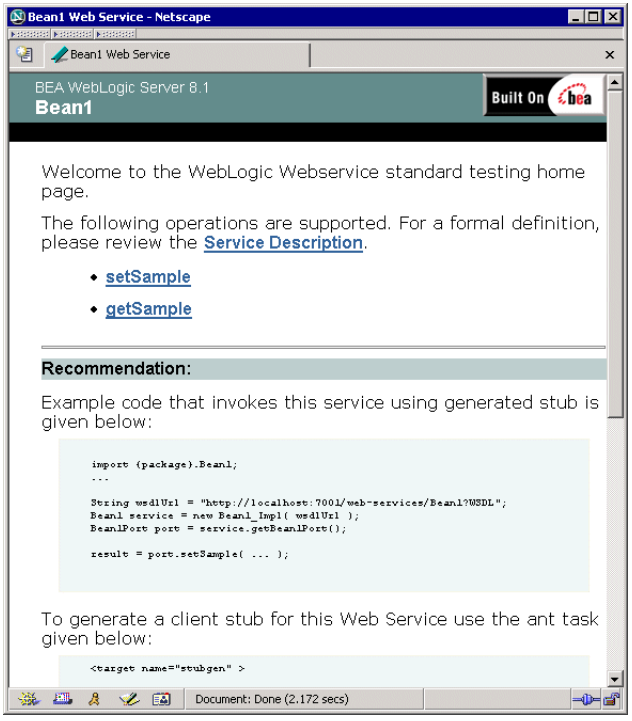
```
<protocol>://<host:port>/<contextURI>/<serviceURI>
```

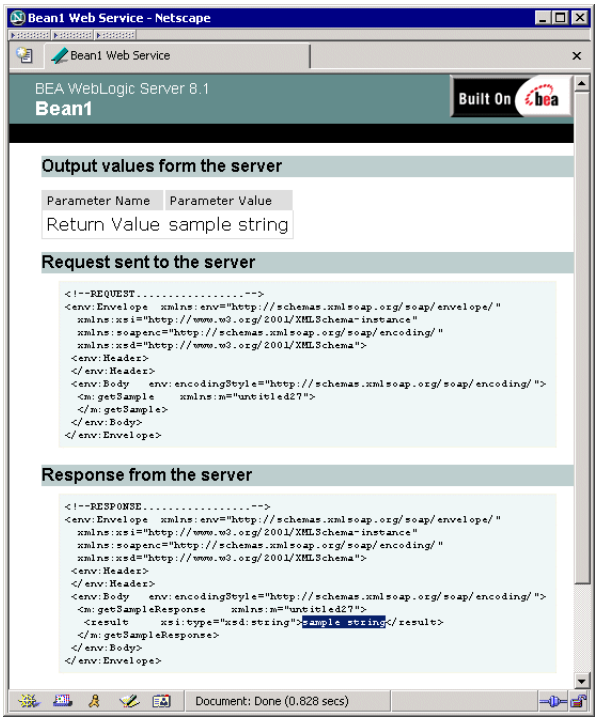
where:

`<protocol>`	The protocol for the service, such as http.
`<host>`	The computer running WebLogic Server, such as `localhost`.
`<port>`	The port number where WebLogic Server is listening, such as `localhost:7001`.
`<contextURI>`	The context root of the WebApp or the name of the WebApp archive file., such as `web-services`.
`<serviceURI>`	The Uniform Resource Identifier of the service.

For example, if you're running WebLogic on the default port 7001 locally, the WebApp is `web-services`, and the ServiceUri is Bean1, the address for the web service would be: `http://localhost:7001/web-services/Bean1`.

You can also find the ServiceURI in the `servicegen.wldu` file or right-click the Web Service Components node in the WebApp node in the project pane and choose Properties.





To open the WSDL for your web service on the home page, choose the Service Description link or enter this URL:

```
<protocol>://<host:port>/<contextURI>/<serviceURI>?WSDL
```

See also

- "Testing deployed services" on page 8-31
- "The WebLogic Web Services Home Page and WSDL URLs" in "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com`.

Chapter

5

Working with WSDL

This is a feature of JBuilder Enterprise

To write a client application that invokes a web service, developers need information about where the service is located and how to access it. The Web Services Description Language (WSDL) provides this information. A WSDL document, written in XML, describes a web service and how to call it.

A WSDL document exposes the web service's method signature, protocol to be used, network address, and data format. With the information available in the WSDL document, a programmer can write an application to interface with a web service. By examining a web service's WSDL document, developers know what methods are available and how to call them using the proper parameters.

To discover other web services and their corresponding WSDL documents and to publish web services, programmers can use the UDDI Business Registry where businesses register their services. JBuilder provides the Web Services Explorer, available on the Tools menu, for discovering and publishing web services. For more information, see Chapter 10, "Browsing and publishing web services."

WSDL terms

To better understand the terms used by WSDL documents, see the following table. For in-depth information about WSDL, see the Web

Services Description Language 1.1 specification at http://www.w3.org/TR/wsdl on the World Wide Web Consortium web site.

Table 5.1 WSDL terms

Term	WSDL element	Definition
Endpoint		A port. See port.
Message	`<message name="GetQuoteRequest">`	An abstract, typed definition of the data (parameter).
Type	`<part name="symbol" type="xsd:string"/>`	A container for data type definitions, such as XSD, which is an attribute of the `<part>` element in a message.
Port Type	`<portType name="GetQuote">`	An abstract set of operations supported by one or more endpoints.
Binding	`<binding name="GetQuoteBinding" type="tns:GetQuote">`	A protocol and data format for a particular port type. This is the binding between the WSDL abstract definition and the implementation. SOAP is an example of a binding type.
Operation	`<operation name="getQuote">`	An abstract definition of an action of the service (method).
Port	`<port name="GetQuote" binding="tns:GetQuoteBinding"> <soap:address location= "http://localhost: 8080/<contextURI><serviceURI>"/>`	An address for a binding and a communication endpoint defined as a combination of a binding and a network address.
Service	`<service name="GetQuoteService">`	A collection of related endpoints (ports).

Sample WSDL document

```
<?xml version="1.0" ?>
<definitions name="urn:GetQuote"
             targetNamespace="urn:xmltoday-delayed-quotes"
             xmlns:tns="urn:xmltoday-delayed-quotes"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- message declns -->
  <message name="GetQuoteRequest">
    <part name="symbol" type="xsd:string"/>
  </message>

  <message name="GetQuoteResponse">
    <part name="result" type="xsd:float"/>
  </message>
```

```
<!-- port type declns -->
  <portType name="GetQuote">
    <operation name="getQuote">
      <input  message="tns:GetQuoteRequest"/>
      <output message="tns:GetQuoteResponse"/>
    </operation>
  </portType>

  <!-- binding declns -->
  <binding name="GetQuoteBinding" type="tns:GetQuote">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getQuote">
      <soap:operation soapAction="getQuote"/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:xmltoday-delayed-quotes"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="urn:xmltoday-delayed-quotes"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>

  <!-- service decln -->
  <service name="GetQuoteService">
    <port name="GetQuote" binding="tns:GetQuoteBinding">
      <soap:address location="http://localhost:8080/<contextURI>/<serviceURI>"/>
    </port>
  </service>
</definitions>
```

WSDL samples

For sample WSDL documents, see the following directories:

- `<jbuilder>/samples/webservices/axis/wsdl`
- `<jbuilder>/samples/weblogic`

See also

- "Tutorial: Generating a web service from a WSDL document" on page 11-8
- Chapter 10, "Browsing and publishing web services"

Chapter

6

Developing EJBs as web services

This is a feature of JBuilder Enterprise

The JBuilder web services features are designed to assist you in building enterprise class web services applications on the J2EE platform. The solutions are standards-based and are portable across application servers. The J2EE platform has evolved over the years and is currently the platform of choice for enterprise Java development. Enterprise JavaBean (EJB) containers, which provide a solid platform for business logic and enterprise data, in conjunction with web containers, such as servlets and JSPs, have extended the functionality of applications by leveraging the penetration of browsers and HTML.

Web services extend the functionality of the J2EE platform still further in providing cross-platform and language-independent solutions. In addition, the J2EE platform can itself leverage web services built outside of its domain. Web services are not confined to a browser environment and can be embedded just as easily in other applications and in application servers.

Typically, as is the case with web containers, coarse-grained business methods are the right candidate and appropriate design for exposing functionality. JBuilder makes this effortless by automatically exposing the appropriate methods in the stateless session beans in the project. You can also override this default behavior and select only the EJB modules, beans, and methods that you want to expose as web services. In addition to creating EJB-based web services, JBuilder provides features for quickly implementing existing web services as EJBs.

As described in the sections that discuss toolkits, JBuilder automatically exports your EJBs as web services by default. Simply develop your EJB application as usual, configure the project for web services with the Web Services Configuration wizard, and run the project with the Web Services Server runtime configuration created by the wizard. All stateless session beans with business methods in the remote interface are automatically deployed to the server as web services without any additional steps. Although some application servers may require an additional step to deploy the EAR to the server.

Developing EJBs as web services varies according to the toolkit selected. For more information on EJBs, see the appropriate chapter for the toolkit.

Important For configuration issues with application servers, see Appendix A, "Modifying enterprise application server settings for web services" and "Release Notes" (Help | Release Notes).

For a list of enterprise application servers that JBuilder supports for the web services features, see "Supported enterprise application servers" on page 2-8.

Chapter

7

Using the Apache Axis toolkit

This is a feature of JBuilder Enterprise. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

Apache Axis is an open source implementation of Simple Object Access Protocol (SOAP), an XML-based protocol for exchanging information. When you choose Apache Axis as the toolkit in a JBuilder web services wizard, the wizard uses Axis to generate a WSDL, Java classes, deployment files, administration files, and so on, depending on which wizard you're using.

For example, when the Axis toolkit is selected in the Import A Web Service wizard, the wizard generates Java classes from an existing WSDL document, as well as deployment files. You can then implement these classes to consume the service specified in the WSDL. You can also create server-side classes for a web service. If the Axis toolkit is use in the Export As A Web Service wizard, the wizard generates a WSDL document from an existing Java class and exposes selected methods of the class as a web service.

For more information on the Apache Axis toolkit, see the documentation available in `<jbuilder>/thirdparty/xml-axis/java/docs`.

Exporting a class as a web service

The Export As A Web Service wizard is used to export a class as a web service. It has options to choose the methods you want to make available to the service. You can use this wizard to export any Java class as a web service; for example, a JavaBean. However, if the class uses non-bean types as parameters or return values, you need to provide serializers and deserializers. The wizard has options to generate the server-side and client-side implementations of the service. It also generates a WSDL, which describes the service, and deployment information for the Axis toolkit.

Note Exporting a class is different from exporting an EJB. You must explicitly export a Java class with the Export As A Web Service wizard. EJBs, however, are exported automatically if the project is configured for web services with the Web Services Configuration wizard.

If the project isn't configured for hosting the web service, the Web Services Configuration wizard displays first before the Export As A Web Service wizard displays.

Depending on the settings you select, the Export As A Web Service wizard generates some or all of the following files from the Java class or interface. If you uncheck the Generate Client Stub option, no Java files are generated; only the WSDL and WSDD are generated. The examples for the file names in the following table are from the `AddressBook.wsdl` sample in `<jbuilder>/samples/webservices/axis/wsdl/AddressBook.wsdl`.

Table 7.1 Files generated by the Export As A Web Service wizard

File name	Description	WSDL element
class name[PortType]`.wsdl`	The WSDL document which describes the web service.	
service name`.java`	A service interface which defines a get method for each port listed in the `service` element of the WSDL. This service interface defines a factory class to get a stub instance.	`<service name= "AddressBookService">`
service name`Locator.java`	A locator class which is the client-side server implementation of the service interface.	`<service name= "AddressBookService">`
service name`TestCase.java`	An optional JUnit test case for testing the web service.	`<service name= "AddressBookService">`
portType name`.java`	An interface for each `portType` in the WSDL. You will use the implementation of this interface to call the remote methods.	`<portType name= "AddressBookPortType">`
binding name`Stub.java`	A client-side stub class which acts as a proxy for a remote web service. This allows you to call the web service as if it were a local object. This class implements the *`class name`*`[PortType].java` interface.	`<binding name= "AddressBookSOAPBinding">`

Table 7.1 Files generated by the Export As A Web Service wizard (continued)

File name	Description	WSDL element
data types	Java files for all other types and holders needed for the web service.	
`deploy.wsdd`	An XML file for each service that provides deployment information to the web services server and can be used with the AdminClient.	
`server-config.wsdd`	An XML file that provides deployment information to the server.	

The wizard generates a package name based on the package name of the class with `.generated` appended to it.

The Export As A Web Service wizard can optionally generate client stub classes. If the Generate Client Stub option is selected on the first page of the wizard, the wizard has additional steps. The additional steps provide the same functionality as the Import A Web Service wizard and generate a client stub from a WSDL document.

The Export As A Web Service wizard is available on the Web Services tab of the object gallery. You can also right-click any `.java` file in the project pane and select Export As A Web Service from the context menu.

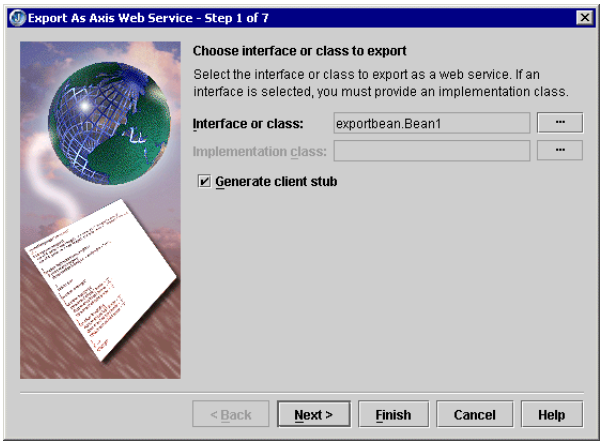
To export a class as a web service and generate Java classes and a WSDL for the service, follow these basic steps:

1. Create a new project (File | New Project).
2. Create a JavaBean (File | New | General | JavaBean), choose `java.lang.Object` as the base class, and check the options: Public and Generate Sample Property. Uncheck the Generate Main Method option.
3. Right-click the bean in the project pane and choose Export As A Web Service to open the Export As A Web Service wizard.

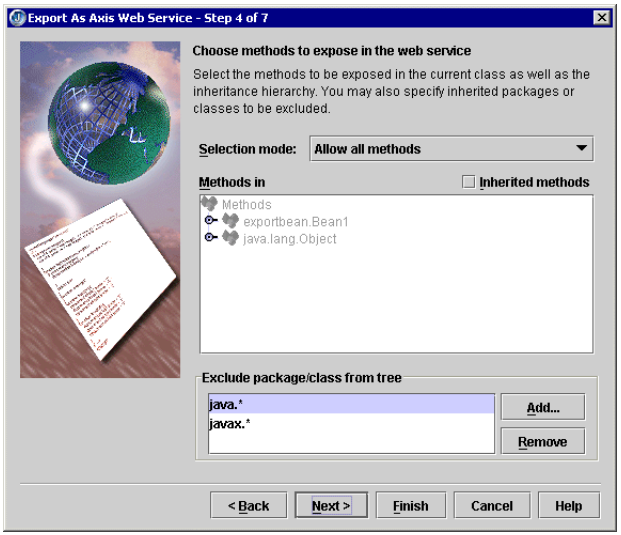
   Because the project isn't configured for web services yet, the Web Services Configuration wizard displays before the Export As A Web Service wizard.

4 Create a SOAP implementation with the Axis toolkit to host the web service as described in "Working with the Web Services Configuration wizard" on page 3-1.

Once the project is configured for web services, the Export As A Web Service wizard displays.

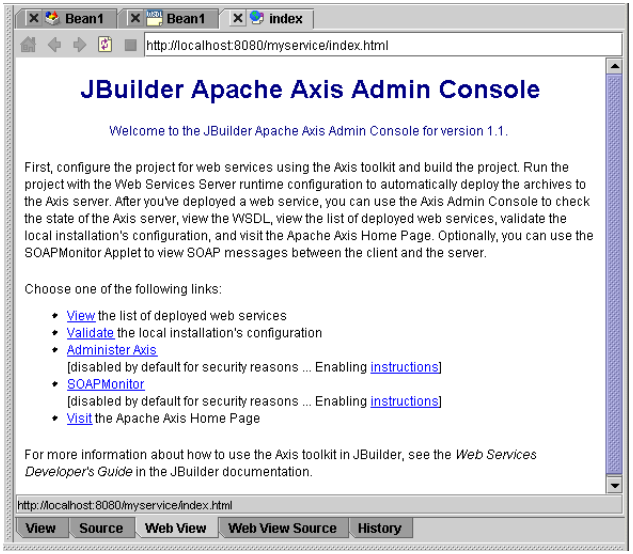


5 Continue through the steps of the wizard and choose the desired options and the methods you want to expose as a service in the wizard. For more information on the Export As A Web Service wizard options, choose the Help button in the wizard.



6 Expand the web services server node, right-click `index.html`, and choose Web Run Using "Web Services Server" to deploy the service to the web services server. You can also choose Run | Run Project to run the Web Services Server configuration created by the Web Services Configuration wizard.

The service is deployed to the web server and the Axis Admin Console displays where you can administer, view, and validate the service.



7 Expand the package generated by the wizard,`<package-name>.generated`. Right-click the JUnit test case, `<class name>ServiceTestCase.java`, and choose Run Test Using Defaults to test the generated service.

Important If you make any changes to a class after you've exported it, you need to recompile, export as a web service again, and redeploy to the web services server for the changes to take effect.

For a tutorial which uses the Export As A Web Service wizard to export a JavaBean as a web service, see "Tutorial: Creating a simple web service" on page 11-1. For more information about SOAP, see Chapter 3, "Configuring projects for web services."

Importing a WSDL

The Import A Web Service wizard generates Java classes that implement the web service defined in the WSDL. It can generate both the server-side and client-side classes for creating and consuming services. The wizard can also generate only client-side classes for consuming a service. It can optionally generate a JUnit test case to test interaction with the generated web service. You add code to the generated classes to implement the

methods as desired. The Import A Web Service wizard also generates a `deploy.wsdd` file which provides deployment information to the web services server.

Depending on the settings you select, the Import A Web Service wizard generates some or all of the following files from the WSDL. The examples for the file names are from the `AddressBook.wsdl` sample in `<jbuilder>/samples/webservices/axis/wsdl/AddressBook.wsdl`.

Table 7.2 Files generated by the Import A Web Service wizard

File name	Description	WSDL element
complexType name`.java`	A class for each `complexType` in the types section of the WSDL. A holder class if this type is used as an inout/out parameter.	`<xsd:complexType name="phone">`
service name`.java`	A service interface for each service which defines a get method for each port listed in the `service` element of the WSDL. This service interface defines a factory class to get a stub instance.	`<service name="AddressBookService">`
service name`Locator.java`	A locator class for each service which is the implementation of the service interface.	`<service name="AddressBookService">`
service name`TestCase.java`	An optional JUnit test case for testing the web service.	`<service name="AddressBookService">`
portType name`.java`	An interface for each `portType` in the WSDL. You will use the implementation of this interface to call the remote methods.	`<portType name="AddressBookPortType">`
binding name`Impl.java`	An implementation class for each `portType` interface.	`<binding name="AddressBookSOAPBinding">`
binding name`Skeleton.java`	An optional skeleton class to encapsulate an implementation for the server.	`<binding name="AddressBookSOAPBinding">`
binding name`Stub.java`	A stub class for each binding which acts as a proxy for a remote web service. This allows you to call the web service as if it were a local object.	`<binding name="AddressBookSOAPBinding">`

Table 7.2 Files generated by the Import A Web Service wizard (continued)

File name	Description	WSDL element
`deploy.wsdd`	An XML file for each service that provides deployment information to the web services server and can be used with the AdminClient.	
`server-config.wsdd`	An XML file that provides deployment information to the server.	

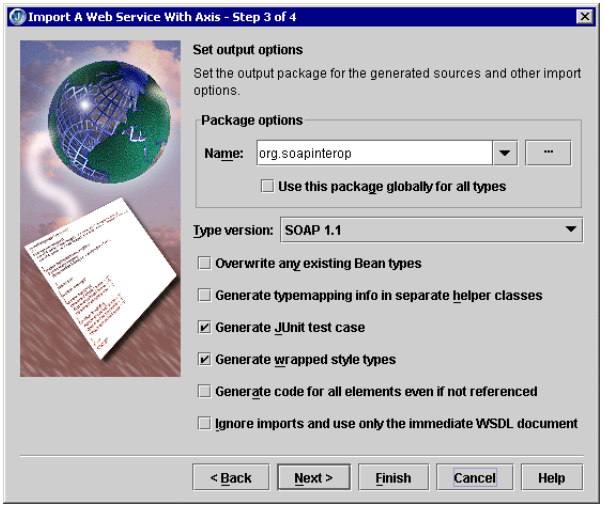
The wizard generates a package name based on the WSDL target namespace or uses one that you specify, as well as adding any necessary project libraries.

The Import A Web Service wizard is available on the Web Services tab of the object gallery. You can also right-click any WSDL document in the project pane and select Import A Web Service from the context menu. The Import A Web Service wizard is also available in the Web Services Explorer when a node specifying a WSDL is selected.

To import a WSDL and generate Java classes from it, follow these basic steps:

1. Create a new project (File | New Project).
2. Create a SOAP implementation with the Axis toolkit to host the web service as described in "Working with the Web Services Configuration wizard" on page 3-1 (File | New | Web Services | Web Services Configuration).
3. Choose one of these methods to import a WSDL and open the Import A Web Service wizard:
   - Add a WSDL to your project, right-click it in the project pane, and choose Import A Web Service.
   - Add a WSDL to your project, select it in the project pane, choose File | New | Web Services, and double-click the Import A Web Service icon.
   - Choose Tools | Web Services Explorer, browse to a WSDL, and choose File | Import A Web Service.
4. Accept the WSDL URL or click the ellipsis (…) button to browse to a WSDL. Enter a user name and password if the WSDL requires one. Click Next to continue.

5 Choose the server-side and output options you want on step 2 and 3. Notice the package name in Package Options. This is the package where the wizard puts the generated Java class files. For more information on the Import A Web Service wizard options, choose the Help button on any step of the wizard.



6 Customize package names on Step 4 and click Finish to close the wizard.

7 Expand the new package created by the wizard to see the generated Java class files.

8 Expand the web services server node, right-click `index.html`, and choose Web Run Using "Web Services Server" to deploy the service to the web services server. You can also run the project with the Web Services Server runtime configuration created by the Web Services Configuration wizard. For more information, see "Starting the web services server" on page 3-5.

9 Run the client to consume the service or edit the server classes to implement the service locally.

10 Right-click the JUnit test case in the project pane if you created one and choose Run Test Using Defaults to test the generated service.

For a tutorial which uses a WSDL document to consume a web service, see "Tutorial: Generating a web service from a WSDL document" on page 11-8.

Exporting EJBs as web services

Exporting Enterprise JavaBeans (EJBs) as web services in JBuilder requires no additional work other than configuring the project for web services. Simply develop your EJB application as usual, configure the project for web services with the Web Services Configuration wizard, and run the project with the Web Services Server runtime configuration created by the wizard. By default, JBuilder automatically exposes all the stateless session beans with business methods in the remote interface. You can also override this default behavior and select only the Enterprise JavaBean (EJB) modules, beans, and methods that you want to expose as web services.

Note Exporting an EJB is different from exporting a Java class. You must explicitly export a Java class with the Export As A Web Service wizard. EJBs, however, are exported automatically if the project is configured for web services with the Web Services Configuration wizard.

Important For configuration issues with application servers, see Appendix A, "Modifying enterprise application server settings for web services," and the "Release Notes" (Help | Release Notes).

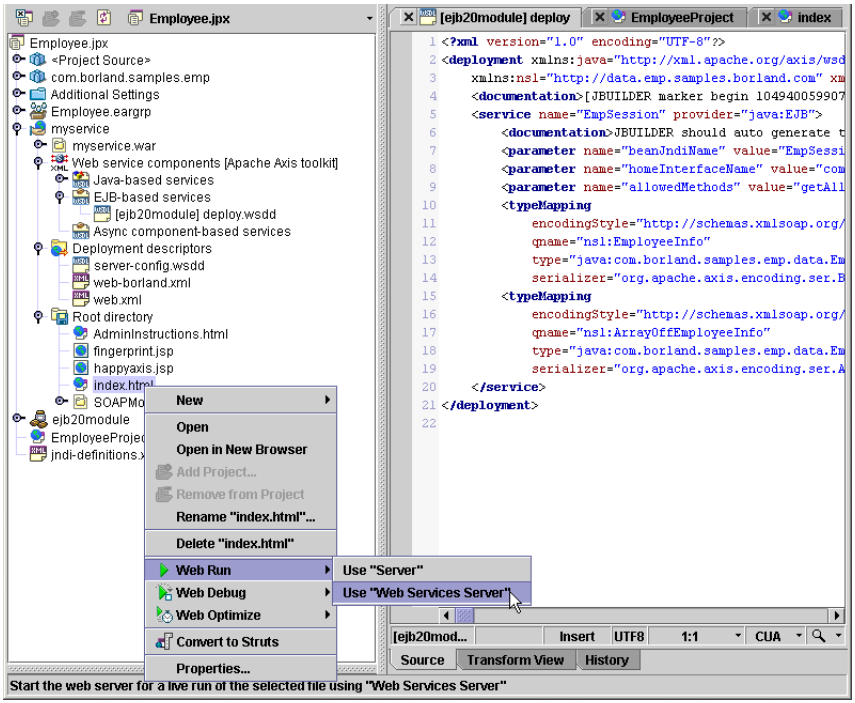
To create an EJB-based web service, complete the following steps:

1. Use the Web Services Configuration wizard and create the Axis WebApp as you normally would while creating a standalone web services application. This creates a runtime configuration based on the application server for the project. For more information on the Web Services Configuration wizard, see "Working with the Web Services Configuration wizard" on page 3-1.
2. Create one or more EJB modules and populate them with session and entity beans. Implement them as you would with any EJB application. You may already have coarse-grained stateless session beans. If you don't, you need to create one, as only stateless session beans are exposed as web services. The stateless session beans must have at least one valid method in the remote interface. This model parallels the access of beans from a web container, such as Servlet/JSPs. For more information on EJBs, see *Developing Enterprise JavaBeans*.
3. Build the project.

Note The Apache Axis toolkit combines each deployment file (`deploy.wsdd`) in your project into a `server-config.wsdd` upon building the project. Any manual edits to the deployment files are overwritten by the toolkit. If you edit `server-config.wsdd`, turn off the Regenerate Deployment option on the Web Services tab of the Build page of Project Properties or `server-config.wsdd` will be overwritten by the toolkit when you build the project. For more information, see "Setting build options" on page 3-6.

For more information on deploying web services, see "Deploying web services" on page 7-13.

4 Expand the Axis WebApp node, right-click `index.html`, and choose Run Using "Web Services Server" to run the SOAP implementation and the EJB application server.



Note that if you modify the EJB application after you've exported it as a service, you'll need to rebuild and redeploy the application again for the changes to take effect.

For a tutorial that demonstrates the complete web services cycle using EJBs, see "Tutorial: Importing a web service as an EJB application" on page 11-29.

Importing services as EJB applications

The Import A Web Service wizard provides several choices when importing a service from a WSDL file. If you want to implement the service as an EJB, choose the Implementation option, As EJB, on Step 2 of the wizard. You must have an application server specified for your project on the Server page (Project | Project Properties) and an EJB module in your project. You can create an EJB module before using the wizard or create one as you use the wizard.

To implement a web service as an EJB,

1 Add a WSDL document to your project or browse to a WSDL in the Web Services Explorer.

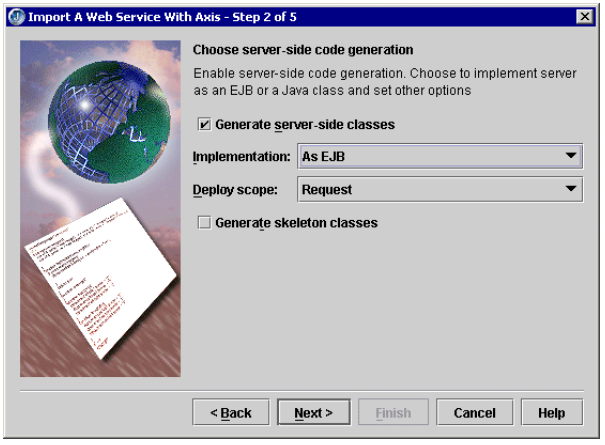
Important Your project must have an application server specified as the server for the project.

2 Right-click the WSDL in the project pane and choose Import A Web Service to open the Import A Web Service wizard. If you're using the Web Services Explorer, choose File | Import A Web Service.

3 Accept Apache Axis as the toolkit and click OK. If you choose the WebLogic toolkit, the steps will be different. See Chapter 8, "Using the WebLogic toolkit."

4 Accept the WSDL name in the WSDL field and click Next. You don't need to enter a user name or password.

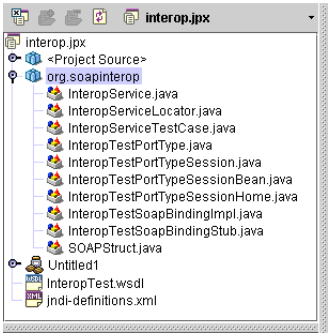
5 Choose the Implementation option, As EJB, on Step 2 of the wizard and click Next.



6 Set the output package, choose the import options, uncheck the Generate JUnit Test Case option, and click Next.

7 Create a new EJB Module with the EJB Module wizard or choose an existing EJB Module from the drop-down list on Step 4 and click Next.

8 Customize package name mapping and click Finish to close the wizard. Notice that the wizard generates a package based on the targetNamespace of the WSDL. Expand the package to see the `.java` files generated by the wizard.



9 Build the project.

10 Create an EJB test client (File | New | Enterprise), modify it, run the server, and test the EJB application.

For a tutorial on importing a WSDL as an EJB application with Borland Enterprise Server, see "Tutorial: Importing a web service as an EJB application" on page 11-29.

Importing services as EJB applications in the Web Services Explorer

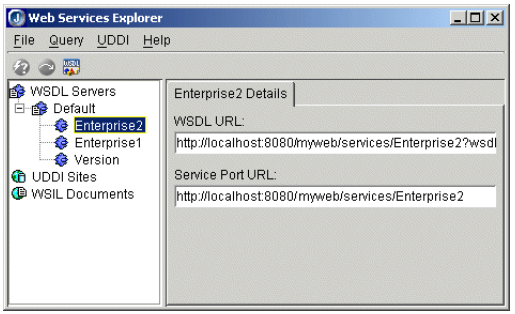
Once you've built and exported your application as a web service to an Axis server, you can build a client to use the service you just created in the Web Services Explorer. In the next few steps, you'll use the Web Services Explorer to browse all services available on the server and to import the WSDL to build a client application.

1 Choose Tools | Web Services Explorer to open the Web Services Explorer. For more information, see Chapter 10, "Browsing and publishing web services."

2 Expand the WSDL Servers node and select the Default node in the tree.

3 Modify the URL field on the Details page with the WebApp context and the server location of the web service. For example, `http://localhost:8080/axis/services`, where `http://localhost:8080` is the server location and `axis` is the web context (WebApp name).

4 Click the Display Services button on the Details page to display the published EJB web services. Notice that beans are available in the list. Note that the application server must be running before you can display its services.



5 Select the EJB in the tree and click the Import A Web Service toolbar button. The Import A Web Service wizard provides several implementation options: implement the service as a class, an EJB, or client code only. For a description of the files generated by the Import A Web Service wizard, see "Importing a WSDL" on page 7-5. For a tutorial on creating a web services from a WSDL, see "Tutorial: Generating a web service from a WSDL document" on page 11-8.

6 You can then publish the web service to a UDDI registry. For more information, see "Publishing web services from an Axis server" on page 10-24.

Deploying web services

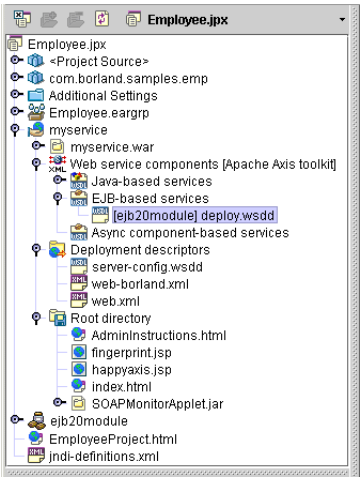
The Import A Web Service and Export As A Web Service wizards create an XML web services deployment descriptor file called `deploy.wsdd` that provides deployment information to a web services server. When EJBs are exported as web services, which JBuilder automatically exports without using a wizard, a `deploy.wsdd` file is generated for each EJB module in the project.

Understanding WSDD files

The WSDD files are Web Services Deployment Descriptors generated by JBuilder to represent the services that are exposed as web services. You can customize the generated information and add additional elements, such as custom handlers, chains, and so on.

deploy.wsdd

The `deploy.wsdd` file displays in the WebApp node hosting the web service. If a Java class is exported, then it displays in the Java-based Services node of the WebApp (WebApp | Web Service Components | Java-based Services). EJB WSDDs display in the EJB-based Services node of the WebApp. The file name is preceded by `[<package name>]` for exported classes and `[<EJB module name>]` for EJBs to differentiate it from other `deploy.wsdd` files which may have been generated for other packages in the project. All WSDD files in the `src` tree are collected in the Web Service Components node for convenience. Note however, that the Web Service Components node does not correspond to an actual physical location on disk. The `deploy.wsdd` file is saved in the same directory as the generated stubs and skeletons.



For exported classes, values in the `deploy.wsdd` file are derived from the information in the WSDL document and from exposed methods chosen in the Export As A Web Service wizard. Settings in the Import A Web Service wizard do not change values in the deployment descriptor. For EJBs, which are automatically deployed by JBuilder without using the wizard, the values for the `deploy.wsdd` are derived from the stateless session beans with business methods in the remote interface. Note that the WSDD format is a proprietary format specified in the Axis toolkit. For more information, see the Axis documentation in `<jbuilder>/thirdparty/xml-axis/java/docs`.

You can change the values in the `deploy.wsdd` for Java classes and EJBs at any time, as well as add additional elements, such as custom handlers, chains, and so on. For more information on customizing EJB deployment, see "Editing WSDD files for EJBs" on page 7-15.

Sample deploy.wsdd file

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:ns="http://untitled33"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
   <service name="Bean1" provider="java:RPC">
      <parameter name="className" value="untitled33.Bean1"/>
      <parameter name="allowedMethods" value="*"/>
      <parameter name="scope" value="Request"/>
  </service>
</deployment>
```

See also

- "Custom Deployment - Introducing WSDD" in the Axis documentation in `<jbuilder>/thirdparty/xml-axis/java/docs/user-guide.html`
- "Axis Web Services Deployment Descriptor (WSDD)" in the Axis documentation in `<jbuilder>/thirdparty/xml-axis/java/wsdd/docs/index.html`
- "Deployment (WSDD) Reference" in the *Axis Reference Guide* in `<jbuilder>/thirdparty/xml-axis/java/docs/reference.html#Deployment`

Editing WSDD files for EJBs

You can customize the EJB deployment of your web services in the following two ways:

- "Modifying the deployment node properties" on page 7-15
- "Editing deploy.wsdd and modifying the <documentation> element" on page 7-16

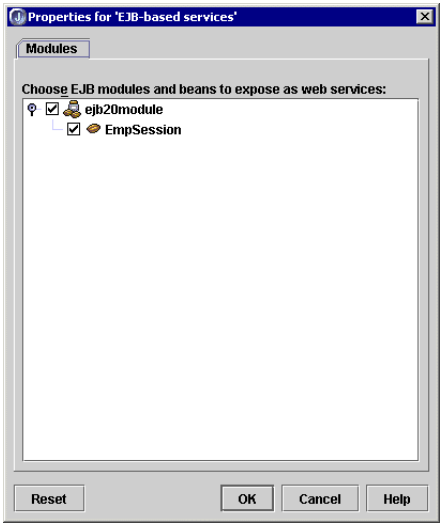
Modifying the deployment node properties

By default, JBuilder automatically exposes all the stateless session beans with business methods in the remote interface. You can override this default behavior and select only the Enterprise JavaBean (EJB) modules, beans, and methods that you want to expose as web services.

You can change the automatic deployment behavior on the Modules page of the EJB-based Services Properties dialog box:

1. Expand the WebApp node that hosts the web service in the project pane, expand the Web Service Components node, right-click the EJB-based Services node, and choose Properties. On the Modules page, choose the EJB modules and stateless session beans to expose as web services. The stateless session beans must have at least one valid

method in the remote interface. The default behavior is to expose all of these beans, which are selected on the Properties page.



2 Uncheck any EJB modules and/or beans that you don't want exposed as a web service.

3 Click OK to close the dialog box.

4 Rebuild the project.

Editing deploy.wsdd and modifying the <documentation> element

If you modify the `deploy.wsdd` for an exported EJB, you need to edit the `<documentation>` element to prevent the toolkit from overwriting your changes. In `deploy.wsdd` for EJBs, there's a `<documentation>` element for each `<service>` element. Change the entry from YES to NO in the `<documentation>` element for each `<service>` element that you modified:

```
<documentation>JBUILDER should auto generate this entry: NO(YES/NO)</documentation>
```

After you modify this entry, the toolkit will **not** overwrite your edits in the `deploy.wsdd`. Rebuild the project for the changes to take effect.

server-config.wsdd

At build time, JBuilder generates web services for the entire project using the individual `deploy.wsdd` as input. The result of this operation is a `server-config.wsdd`, which is located in the Deployment Descriptors node of the WebApp. You can edit this file, but you must turn off a build option to prevent the toolkit from overwriting it on the next build. For more information, see the Web Services tab of the Build page (Project | Project Properties | Build).

Sample server-config.wsdd file

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/" xmlns:java="http://
xml.apache.org/axis/wsdd/providers/java">
<globalConfiguration>
  ...
  <handler name="MsgDispatcher"
      type="java:org.apache.axis.providers.java.MsgProvider"/>
  <service name="Bean1" provider="java:RPC"/>
  <parameter name="allowedMethods" value="*"//>
  <parameter name="scope" value="Session"//>
  <parameter name="className" value="untitled33.Bean1"//>
  </service>
  ...
</deployment>
```

Editing server-config.wsdd

To edit `server-config.wsdd`,

1. Build the project to generate the `server-config.wsdd` file from the web services `deploy.wsdd` files in the project.
2. Open the `server-config.wsdd` in the Deployment Descriptors node of the WebApp hosting the web service.
3. Look for and change whatever is of interest. For example, you may choose to restrict the methods exposed from `<parameter name="allowedMethods" value="getSample, setSample"/>` to only allow the get method and no set method: `<parameter name="allowedMethods" value="getSample"/>`.
4. Use the same mechanism to add requestFlow, typemappings, or any other deployment elements to the WSDD file and JBuilder will deploy them for you.
5. Disable the Regenerate Deployment option on the Web Services tab of the Build page (Project | Project Properties | Build). This prevents the toolkit from overwriting the file on the next build.
6. Rebuild the project.

Chapter

8

Using the WebLogic toolkit

This is a feature of JBuilder Enterprise

JBuilder provides wizards for working with the WebLogic toolkit to easily consume and create web services. The Export As A Web Service wizard exports a Java class or classes as a web service. EJBs, however, are exported automatically if the project is configured for web services with the Web Services Configuration wizard. The Export As An Asynchronous Web Service wizard exports a web service using asynchronous messaging. The Import A Web Service wizard allows you to import a web service from a WSDL document or an EAR.

These wizards can create a client JAR, which can be used by a client to invoke the service. The client JAR is generated in a GeneratedWebServiceClients node in the project pane. To see the Java classes generated by the wizard and the WebLogic toolkit, open the client JAR file. The file, `<ServiceName>Port.java`, contains the interface definition of your web service, where `<ServiceName>` refers to the name of the web service. The `<ServiceName>_Impl` stub implements the JAX-RPC `Service` interface. The WebLogic toolkit uses an implementation of JAX-RPC (Java APIs for XML-based Remote Procedure Call) compliant client stubs.

Table 8.1 JAX-RPC interfaces and classes

Interface or class	Description
`Service`	Main client interfaces used for static and dynamic invocations.
`ServiceFactory`	Factory class for creating `Service` instances.
`Stub`	The client proxy for invoking the operations (methods) used for static invocations of a service.
`Call`	Used to dynamically invoke a web service.

The WebLogic toolkit in JBuilder is supported on WebLogic Server 7.0 and 8.1. For more information on WebLogic Server and web services see the

BEA documentation at `http://edocs.bea.com/wls/docs70/webserv/index.html` for version 7.0 or `http://edocs.bea.com/wls/docs81/webserv/index.html` for version 8.1.

Exporting a class as a web service

The Export As A Web Service wizard is used to export a class as a web service. It has options to choose the methods you want to make available to the service. You can use this wizard to export any Java class or classes as a web service; for example, a JavaBean. However, if the class uses non-bean types as parameters or return values, you need to provide serializers and deserializers. The wizard has options to generate the client-side implementations of the service. It also generates a WSDL, which describes the service, and deployment information for the WebLogic toolkit.

The wizard provides the option to generate a client stub or only the WLDU deployment files. If you don't generate a client stub, the wizard has two steps and only a `servicegen.wldu` file, which provides deployment information to the WebLogic Server, is generated. For more information on the deployment file, see "Understanding WLDU files" on page 8-25. If you check the Generate Client Stub option, the wizard has seven steps and generates a `<client>.jar`, which contains the classes and the WSDL. The `<client>.jar` displays as a node in the GeneratedWebServiceClients node in the project pane. The wizard also adds the GeneratedWebServiceClients library to the project (Project | Project Properties | Paths | Required Libraries), so JBuilder can find the classes in the JAR.

If the project isn't configured for hosting the web service, the Web Services Configuration wizard displays first before the Export As A Web Service wizard displays.

Note Exporting a class is different from exporting an EJB. You must explicitly export a Java class with the Export As A Web Service wizard. EJBs, however, are exported automatically if the project is configured for web services with the Web Services Configuration wizard.

To create a web service from a Java class with the WebLogic toolkit, follow these basic steps:

1. Choose File | New Project to create a project with the Project wizard.

   Important Be sure to create the project in a directory **without** any spaces in the name or WebLogic Server may generate errors.

2. Choose Project | Project Properties and click the Server tab.
3. Specify WebLogic Application Server 7.x or 8.1 as the single server for the project.

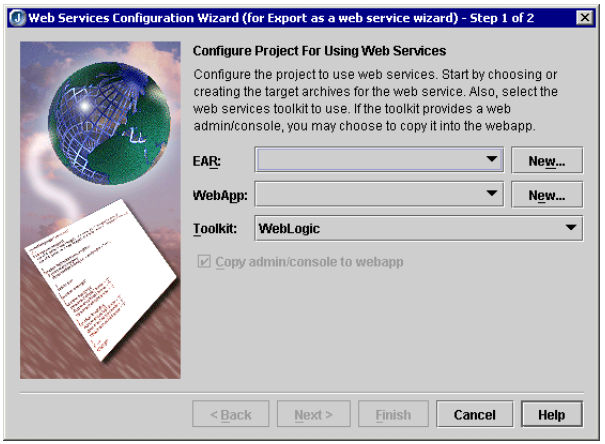
4 Choose File | New | General and double-click the JavaBean icon to create a JavaBean to export as a service.

5 Choose `java.lang.Object` as the base class.

6 Check the Public and Generate Sample Property options in the JavaBean wizard. Uncheck the Generate Main Method option.

7 Click Finish to create the JavaBean.

8 Right-click the JavaBean in the project pane and choose Export As A Web Service to open the Export As A Web Service wizard. Because the project isn't yet configured for web services, the Web Services Configuration wizard opens, so you can create an EAR, a WebApp to host the service, and choose the web services toolkit.



9 Configure the project for web services as follows:

a Choose the New button next to the EAR field and create an EAR with the EAR wizard. Click Finish to return to the Web Services Configuration wizard.

b Choose the New button next to the WebApp field to create a SOAP WebApp with the Web Application wizard. A WebApp is required to host the service. Enter the name and directory for the WebApp. Click OK to return to the Web Services Configuration wizard.

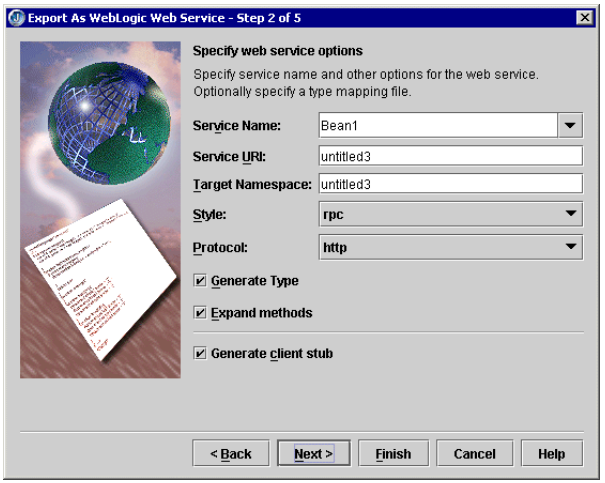
c Choose WebLogic as the web services toolkit and click Next to go to the last step.

d Accept the default Web Services Server runtime configuration. You'll use this later to build and run the project.

e Click Finish to close the Web Services Configuration wizard. Now the Export As A Web Service wizard displays.

10 Accept the class to export, EAR name, and WebApp name on Step 1 of the Export As A Web Service wizard. Notice that the Generate Client Stub option is checked. If you don't want to generate the client stub, uncheck this option. When this option is unchecked, the wizard has only two steps. If Generate Client Stub is checked, the wizard has additional steps and generates a client JAR in the GeneratedWebServiceClients node in the project pane.

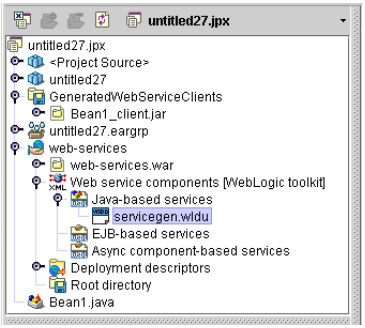
11 Continue to Step 2 and take note of the ServiceURI on Step 2 of the wizard. The service URI is used in the URL for invoking the web service. The values on this page of the wizard are saved in the WLDU deployment file, `servicegen.wldu`. You can modify any of these values on the WebLogic Properties page and then export the Java class again. For more information on modifying these values, see "Setting service naming defaults" on page 8-30.



12 Click Next and continue through the remaining steps of the Export As A Web Service wizard. Notice the default names for the package and the client JAR on Step 4: `<project-name>.generated` and `<class-name>_client.jar`. Click the Help button to learn more about the options.

13 Click Finish to close the wizard. Expand the WebApp node, then the Web Service Components node, and the Java-based Services node to see the WebLogic deployment file, `servicegen.wldu`, which provides deployment information to the server. The WLDU file contains such information as the service name, service URI, web context (contextURI), namespace, and the classes which are exported. For more information on this deployment file, see "Understanding WLDU files" on page 8-25.

Expand the GeneratedWebServiceClients node and double-click the JAR to see the class files and WSDL created by the WebLogic toolkit.



14 Choose Project | Make Project to build the project.

15 Choose Run | Run Project to run the project with the Web Services Server runtime configuration created by the Web Services Configuration wizard. This runtime configuration runs the WebLogic Server.

16 Right-click the eargrp node and choose Deploy Options | Deploy to deploy the EAR to the server.

17 Test the deployed web service on the home page in the Web Services Console (Tools | Web Services Console). For more information, see "Testing deployed services" on page 8-31.

18 Copy the example code in the Web Services Console for invoking the service and modify it to create a client to test the service. See "Writing a test client" on page 8-34.

For more information on writing clients to invoke a web service, see "Writing the Java Client Application Code" in "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com` and "Step 5: Writing a client to test the service" on page 12-7.

Note If you modify any classes in your web service, you need to recompile, export as a service again, and redeploy for the changes to take effect.

For a tutorial on exporting a class as a web service with WebLogic, see "Tutorial: Creating a simple web service" on page 12-1.

Exporting multiple classes as a web service

You can also expose multiple classes as a web service simultaneously as described in the following steps:

1 Select and right-click a class or classes in the project pane and choose Export As A Web Service.

2 Choose the Add button in the wizard if any want to add any additional classes to expose as services.

3 Choose any options you want in the Export As A Web Service wizard and click Finish.

4 Compile and deploy to the server.

5 Test the deployed service in the Web Services Console (Tools | Web Services Console).

Importing a WSDL or an EAR as a web service

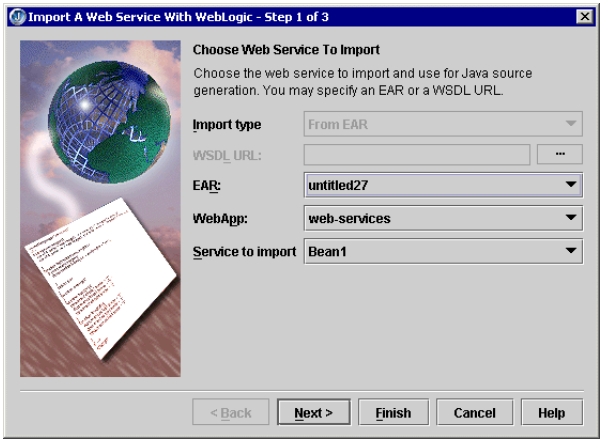
The Import A Web Service wizard generates a web service client. When you use the Import A Web Service wizard with the WebLogic toolkit, you can import a WSDL or import an EAR containing a WebLogic-specific web service. The wizard generates JAX-RPC (Java APIs for XML-based Remote Procedure Call) compliant client stubs. The generated code is packaged in a JAR file in the GeneratedWebServiceClients node of the project. A project library of the same name is automatically created and added to the project.

Importing an EAR as a web service

Follow these basic steps to import an EAR as a web service:

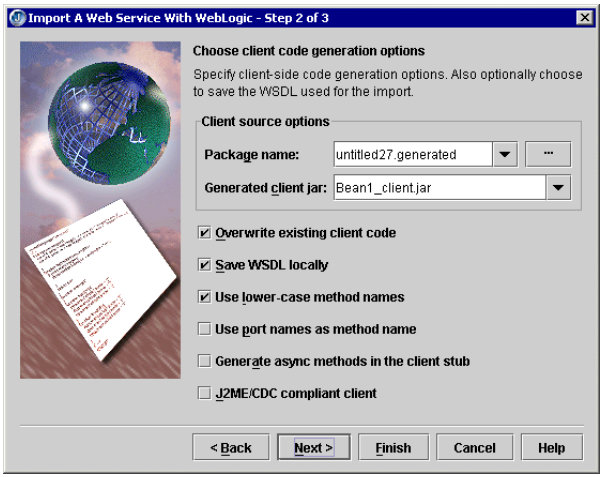
1 Complete the steps for creating a deployed web service as described in "Exporting a class as a web service" on page 8-2.

2 Right-click the eargrp node in the project pane and choose Import A Web Service to open the Import A Web Service wizard.



3 Accept the EAR file name to import and select the service you want to import from the drop-down list, if available, and click Next.

4 Choose the client-side code generation options and click Next.



5 Specify the package name and click Finish.

6 Expand the GeneratedWebServiceClients node to see the client JAR generated by the WebLogic Server.

7 Test the deployed service in the Web Services Console (Tools | Web Services Console) as described in "Testing deployed services" on page 8-31.

8 Copy the example code from the web service home page in the Web Services Console for invoking the service and create a client to test the service. See "Writing a test client" on page 8-34.

For more information on writing clients to invoke a web service, see "Writing the Java Client Application Code" in "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com` and "Step 5: Writing a client to test the service" on page 12-7.

Importing a WSDL as a web service

Importing a WSDL is very similar to importing an EAR. Follow these basic steps:

1 Choose File | New Project to create a project with the Project wizard.

Important Be sure to create the project in a directory **without** any spaces in the name or WebLogic Server may generate errors.

2 Choose Project | Project Properties and click the Server tab. Choose WebLogic 7.0 or 8.1 as the single server for the project.

3. Add a WSDL document to your project with the Add Files/Packages/Classes button on the project pane toolbar or use the Web Services Explorer to import the WSDL. For more information, see "Generating Java classes from WSDL documents" on page 10-26.
4. Right-click the WSDL in the project pane and choose Import A Web Service to open the Import A Web Service wizard.
5. Choose WebLogic as the toolkit and click OK.
6. Accept the WSDL URL and choose the service to import if the WSDL has multiple services.
7. Click Next and choose the client-side code options that you want.
8. Click Next and accept the package name or enter a new one.
9. Click Finish to close the wizard and generate the JAR file. The wizard creates the JAR file of Java classes from the WSDL and displays it in the project pane in a GeneratedWebServiceClients node.
10. Write a client to test the service and run the test. See "Writing a test client" on page 8-34.

For more information on writing clients to invoke a web service, see "Writing the Java Client Application Code" in "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com/` and "Step 5: Writing a client to test the service" on page 12-7.

Exporting EJBs as web services

Exporting Enterprise JavaBeans (EJBs) as web services in JBuilder requires no additional work other than configuring the project for web services. Simply develop your EJB application as usual, configure the project for web services with the Web Services Configuration wizard, and run the project with the Web Services Server runtime configuration created by the wizard. Any EJBs in the project are automatically deployed to the WebLogic Server as web services without any additional steps. By default, JBuilder automatically exposes all the stateless session beans with business methods in the remote interface. You can also override this default behavior and select only the Enterprise JavaBean (EJB) modules, beans, and methods that you want to expose as web services.

Note Exporting an EJB is different from exporting a Java class. You must explicitly export a Java class with the Export As A Web Service wizard. EJBs, however, are exported automatically if the project is configured for web services with the Web Services Configuration wizard.

To create an EJB-based web service, complete the following steps:

1. Choose File | New Project to create a new project.
2. Set WebLogic Application Server 7.x or 8.1 as the server for the project on the Server tab of Project Properties (Project | Project Properties).
3. Use the Web Services Configuration wizard to configure the project for web services. You'll create a web services WebApp and an EAR. This wizard creates a runtime configuration based on the project application server. For more information on the Web Services Configuration wizard, see "Working with the Web Services Configuration wizard" on page 3-1.
4. Create one or more EJB 2.0 modules and populate them with session and entity beans. Implement them as you would with any EJB application. You may already have coarse-grained stateless session beans. If you don't, you need to create one, as only stateless session beans are exposed as web services. The stateless session beans must have at least one valid method in the remote interface. This model parallels the access of beans from a web container, such as Servlet/JSPs. For more information on EJBs, see *Developing Enterprise JavaBeans*.
5. Choose Project | Make Project to build the project.

Note The wizard creates a deployment file, `servicegen.wldu`. Any manual edits to the deployment files are overwritten by the toolkit. If you edit `servicegen.wldu`, turn off the Regenerate Deployment option on the Web Services tab of the Build page of Project Properties or `servicegen.wldu` will be overwritten by the toolkit when you build the project. For more information, see "Setting build options" on page 3-6.

6. Choose Run | Run Project to run the project with the Web Services Server runtime configuration. This runtime configuration builds the project, deploys the EJBs, and runs the server. JBuilder deploys all the 2.0 compliant EJB modules and session beans in the project automatically. If you want to change this web services deployment behavior, right-click the EJB-based Services node in the web services WebApp and choose Properties. Uncheck any modules or beans that you don't want to expose.
7. Right-click the eargrp node, which contains the EAR, and choose Deploy Options | Deploy to deploy the EAR to the WebLogic Server.
8. Test the EJB deployment using the Web Services Console (Tools | Web Services Console) as described in "Testing deployed services" on page 8-31.
9. Copy the example code from the Home page and modify it to create a client to invoke and test the exported web service. For an example, see "Step 5: Writing a client to test the service" on page 12-7.

Note that if you modify the EJB application after you've exported it as a service, you'll need to rebuild and redeploy the application again for the changes to take effect.

For a tutorial that demonstrates the complete web services cycle using EJBs, see "Tutorial: Creating a web service from an EJB application with WebLogic Server" on page 12-9.

Creating asynchronous web services

This is a feature of WebLogic Server 8.1

JBuilder provides the Export As An Asynchronous Web Service wizard, a feature of WebLogic Server 8.1, for developing web services that use asynchronous messaging. *Asynchronous* messaging is one-way messaging and, unlike two-way synchronous messaging, it doesn't require an immediate response. This allows the requestor of the message to continue processing other tasks. In asynchronous messaging, Java Message Service (JMS) is used as the messaging framework and message-driven beans receive or send messages on or from the JMS destination. Using asynchronous messaging, a client can send a one-way message containing data to a web service without requiring a response from the service, such as posting information to a database. The client can also invoke another operation in the service to receive a message back from the service.

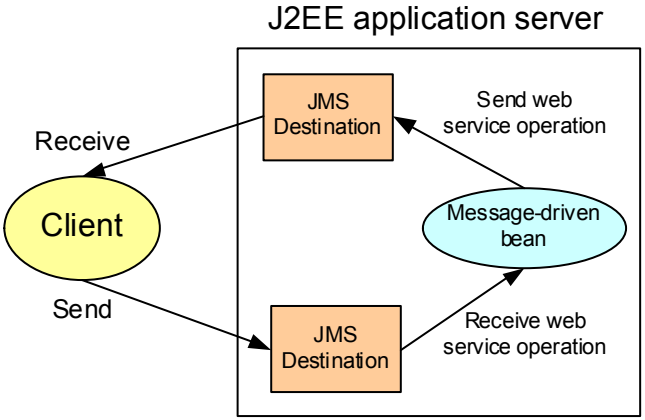
Java Message Service (JMS)

JMS is a Java API that establishes a common set of enterprise messaging concepts that supports asynchronous messaging. In asynchronous messaging, a JMS provider delivers messages to a client as they arrive, without any action on the client's part. In synchronous messaging, such as SOAP over HTTP, an application (requestor) sends a message to another application (provider) to invoke a function from the service. The requestor is blocked until a response is returned and is unable to continue processing. Because some activities don't require a response, asynchronous messaging may be a better choice for messaging. For example, a message might only trigger an activity, as in ordering more widgets from manufacturing. Manufacturing could take days to execute this request. With asynchronous messaging, the requestor isn't blocked and can continue executing other processes.

JMS is commonly used in a distributed enterprise where communication between enterprise systems and J2EE applications is required. JMS can also be used to access and communicate with available web services on these systems. A web service, located on a J2EE application server, can be accessed using JMS messaging and message-driven beans. The message-driven bean, which resides on the J2EE server with the web service, receives and processes the JMS message sent by the client. For more

information on message-driven beans, see "Developing message-driven beans" in *Developing Enterprise JavaBeans*.

In the following diagram, there are two JMS destinations, one for each web service operation. One operation receives messages; the other sends messages. The client can invoke one operation to send data to the service in a message. The message-driven bean listens for the message on the first JMS destination, processes it, and sends it to the second JMS destination. The client can then invoke the second web service operation to receive the processed message located on the second JMS destination.



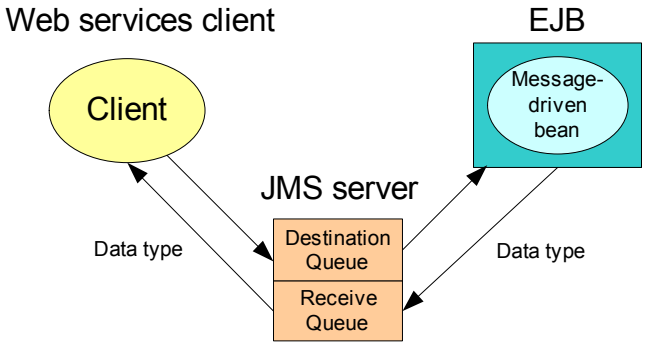
JMS has two message models: a point-to-point message queue and publish/subscribe. The queue retains all messages until they are consumed or have expired. Typically, the queue model has a single consumer. The publish/subscribe model has a sender (publisher) and a receiver (subscriber/client). Clients subscribe to a topic. When messages are published on that topic, the publisher sends them to all subscribers. Generally, the publish/subscribe model has multiple consumers.

For more information on JMS, see `http://java.sun.com/products/jms`.

Using the Export As An Asynchronous Web Service wizard

The WebLogic toolkit uses a JMS implementation with queue support for creating asynchronous web services. The JMS implementation requires a JMS server that queues the messages, a message-driven bean that processes JMS messages, and a JMS message type (data) that's sent between the client and the server. The JMS server must be configured with a destination queue and optionally a receive queue if the message requires a response. Because asynchronous messaging doesn't require a response, the receive queue is optional. The message-driven bean must be configured to listen on the destination queue for any incoming messages. Then, you can use the Export As An Asynchronous Web Service wizard to

create a web services implementation to send the data to the destination and optionally to receive data from the receive queue, if required.



The Export As An Asynchronous Web Service wizard, a feature of WebLogic Server 8.1, can be used to create an asynchronous web service in which a data type is sent between the client and the server. Because the WebLogic toolkit supports JMS-implemented web services using queue as the messaging model, you must configure your WebLogic Server for JMS. Since JMS also requires a message-driven bean to process the messages, your project must contain an EJB module and a message-driven bean before you can create the service.

The typical steps involved in developing an asynchronous web service include:

1. Configure a JMS server, a destination queue, and an optional receive queue on WebLogic Server 8.1.
2. Create an EJB module and a message-driven bean, which processes JMS messages from the client.
3. Create a data type, such as a serialized JavaBean, to send between the client and the server.
4. Create a web services implementation to send the data to the server and optionally receive return data.

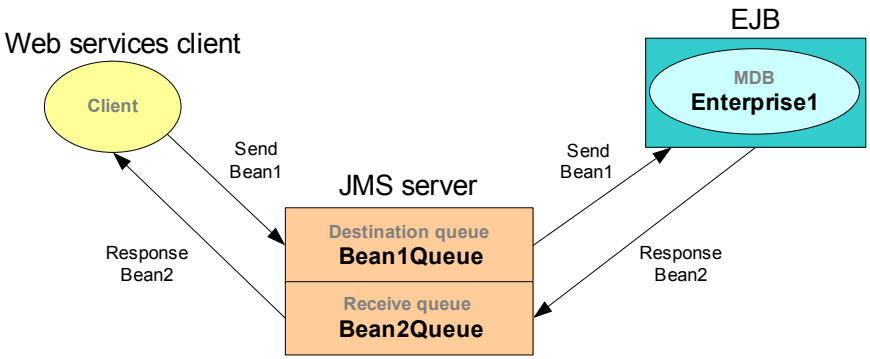
When you use the Export As An Asynchronous Web Service wizard to create an asynchronous web service, the wizard creates an Async Component-based Service node as a child node of the WebApp node in the project pane. This node contains a WLDU file with JMS-specific deployment information. The wizard also generates a JMS class, if a receive queue name is specified, that is used to send and receive point-to-point (queue) messages. The Export As An Asynchronous Web Service wizard also shares some of the same functionality as the Export As A web service wizard, such as creating a WSDL, and creating an optional client JAR to invoke the service.

The WebLogic toolkit supports the point-to-point messaging (queue) model and wraps the Java objects in a `javax.jms.ObjectMessage` object. Because the JMS topic messaging model is deprecated in WebLogic Server, only queue is available in the wizard.

For a tutorial on creating an asynchronous web service, see "Tutorial: Creating an asynchronous web service with WebLogic Server 8.1" on page 12-15. For more information on asynchronous web services and the WebLogic toolkit, see "Writing an Asynchronous Client Application" in "Programming WebLogic web services" in the WebLogic documentation at `http://edocs.bea.com/wls/docs81/webserv/index.html`.

Let's look at a simple example project to demonstrate how to create an asynchronous web service. This example project contains an EJB module, a message-driven bean called Enterprise1, two serialized JavaBeans that are the data types the web service sends and receives, and a JMS class that supports the point-to-point messaging model (queue). This example uses `JMSFactory` for the factory name, and `Bean1Queue` and `Bean2Queue` for the two destination queues. You'll also configure WebLogic Server with a JMS connection factory, a JMS server, and two destination queues. For more detailed information about creating EJBs in JBuilder, see *Developing Enterprise JavaBeans*.

The following diagram demonstrates what this example does:



First, you'll configure the server, create a project, and populate it with the necessary classes.

1. Choose File | New Project to create a project and name it async.
2. Set the server to WebLogic 8.1 on the Server page of Project Properties (Project | Project Properties).
3. Run the WebLogic Server in JBuilder with a Server runtime configuration. Open the WebLogic Admin Console (Tools | WebLogic Admin Console) and configure the server for JMS. For this example, configure a JMS Connection factory with a JNDI name of `JMSFactory`. You also need to set up a JMS server with two JMS destinations of type

queue and name them `Bean1Queue` and `Bean2Queue`. Only a send operation is required if the message-driven bean doesn't need to send back a response. For more information on configuring the server, refer to the WebLogic documentation at `http://edocs.bea.com`. Once you've finished the configuration setup, restart the server to commit the new configuration information.

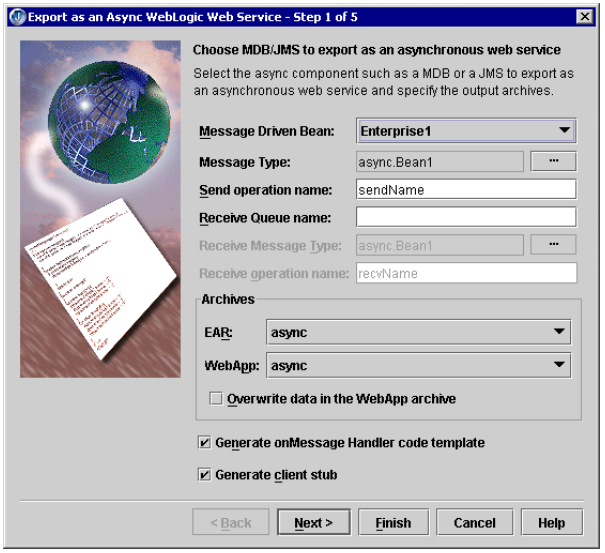
4. Create an EJB 2.0 module called Untitled1 with the EJB Module wizard (File | New | Enterprise).
5. Create a message-driven bean called Enterprise1 in the EJB Designer. Click the name of the bean in the EJB Designer and make these changes in the bean properties:
   a. Accept `Queue` as the Destination Type.
   b. Enter `Bean1Queue` as the destination name.
   c. Enter `JMSFactory` as the Connection Factory Name.
   d. Enter the appropriate pool sizes.
6. Create a serialized JavaBean called Bean1 and populate it with the data you want to send as described here.
   a. Choose File | New | General and double-click the JavaBean icon to open the JavaBean wizard.
   b. Enter Bean1 in the Name field.
   c. Choose Object as the base class, and check the options: Public and Generate Sample Property. Uncheck the Generate Main Method option. Click OK to close the JavaBean wizard and generate the JavaBean.
   d. Choose the Bean tab at the bottom of the editor and check Support Serialization.
   e. Enter the code and the data you want to send. In this example, change the data to "Bean1 Sample" as shown here:

      ```
      private String sample = "Bean1 Sample";
      ```
7. Create a serialized JavaBean called Bean2. Bean2 is the return data. Repeat the steps for creating Bean1 but make the following changes:
   a. Enter Bean2 in the Name field.
   b. Change the data to "Bean2 Sample" as shown here:

      ```
      private String sample = "Bean2 Sample";
      ```

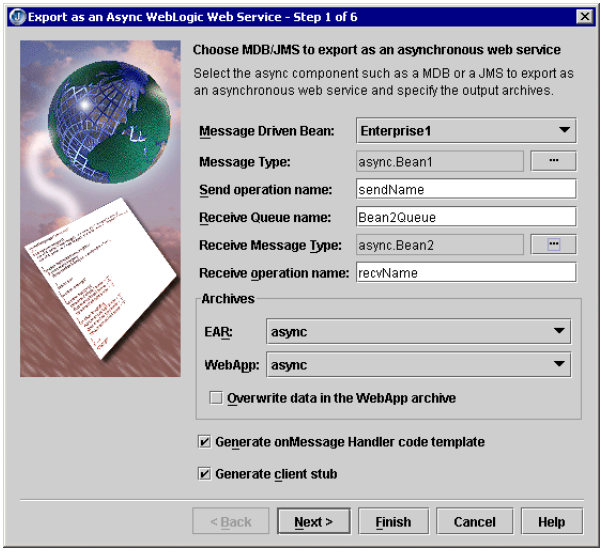
Now, you're ready to configure the project for web services and export the service as an asynchronous web service.

1 Right-click `Bean1.java` in the project pane and choose Export As An Asynchronous web service to export it. First, the web services Configuration wizard displays so you can configure the project for web services. Choose WebLogic as the toolkit, create an EAR named async, and create a WebApp named async. Click Finish to close the wizard. Next, the Export As An Asynchronous Web Service wizard displays. Notice that `async.Bean1` is automatically entered as the JMS Message Type in the wizard. Bean1 is the data that will be sent to the message-driven bean on the server.



2 Enter `Bean2Queue` in the Receive Queue Name field since this example will have a response message from the message-driven bean on the server. Notice that the Generate onMessage Handler Template Code option is enabled. When you have a receive queue, the onMessage handler code is automatically generated.

3 Click the ellipsis (…) button next to the Receive Message Type field. The message that the service receives from the server will be Bean2. Enter `Bean2` in the Search field and choose `async.Bean2` and press *Enter*. The Export As An Asynchronous Web Service wizard should look like this:



4 Click Next twice to go to step 3 of the wizard. Here you'll see that the wizard will generate a Jms1 class. Accept all the defaults and click Finish to close the Export As An Asynchronous Web Service wizard.

5 Expand the async WebApp node in the project pane and the web service Components child node to see that the wizard created a node called Async Component-based Services node which contains the WLDU deployment file. Open the WLDU file to see that the entries you just made in the wizard are stored here.

6 View the source of the message-driven bean, Enterprise 1, to see the generated `onMessage()` method. This is where you would typically insert your business logic to process the message. In this example, you won't add any business logic. You'll simply use the default code generated by the wizard. The default code gets the data out of the queue, in this case,

Bean1. Then it creates a new instance of the return type and send it back. The `onMessage()` method looks like this:

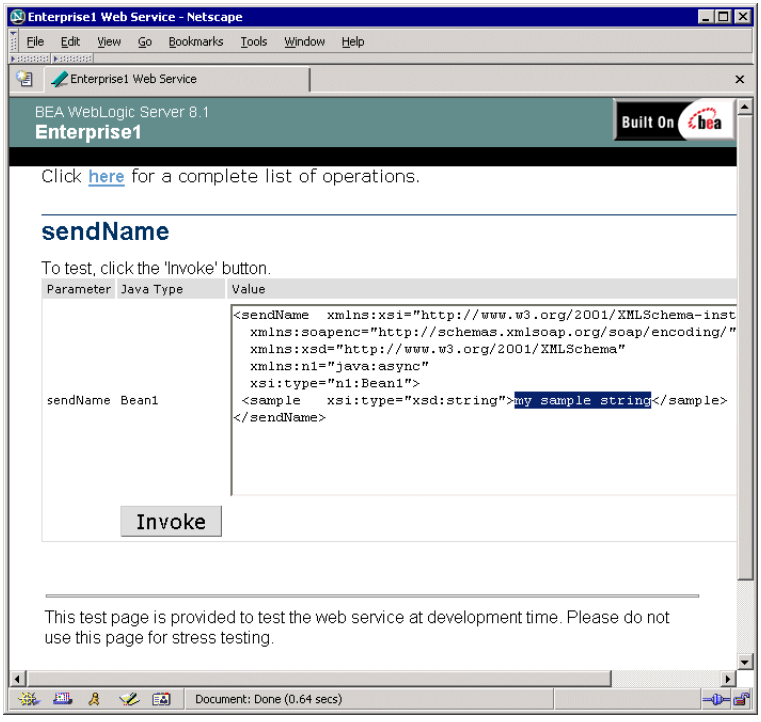
```
public void onMessage(Message msg) {
    /**
     * @todo
     * The OnMessage handler is called when data arrives in the Queue
           from a web services client.
     * The following template code retrieves the data from the Queue and
     * recreates an instance of a strongly typed data of type Bean1,
           exactly as sent by the client.
     */

    Bean2 bean2 = null;

    Bean1 bean1 = null;
    try {
      if (msg instanceof ObjectMessage) {
        bean1 = (Bean1) ( (ObjectMessage) msg).getObject();
      }
    }
    catch (JMSException jmsEx) {
      jmsEx.printStackTrace();
    }
// Implement your business logic using data in Bean1.

    bean2 = new Bean2();
// Populate bean2 with response data before sending to client.

    try {
      Jms1 jms1 = new Jms1();
      jms1.sendObject(bean2);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

7. Run the project with the web services Server runtime configuration to build the project and run the server.
8. Right-click the eargrp node and choose Deploy Options | Deploy to deploy the web service to the server.

Now that the asynchronous service is created, you can test it in the Web Services Console, which displays the testing home page for the web service. You can also copy example code from the testing page to write the test client.

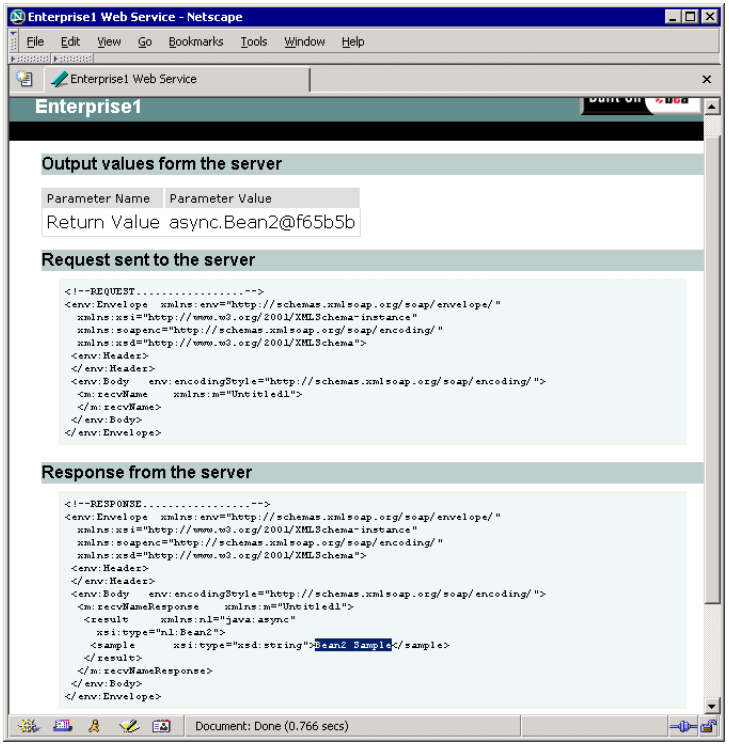
1 Open the testing page (Tools | Web Services Console) to test the deployed operations and to view the SOAP requests and responses sent between the client and the server.

   a Click the sendName operation hyperlink and click the Invoke button.



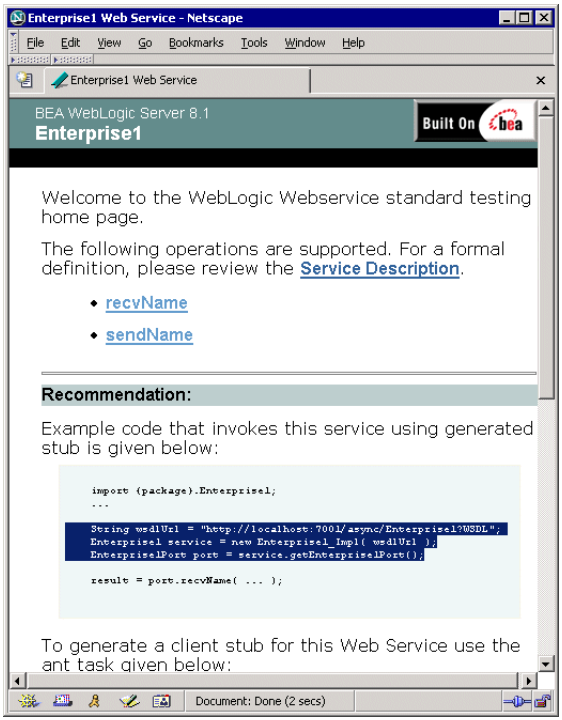
   b Return to the first page, click the recvName operation hyperlink and click the Invoke button. The receive operation, Bean2 Sample, is the response from the server and contains the data, Bean2 Sample. Here

you see in the response that Bean2 Sample is returned. `<sample xsi:type="xsd:string">Bean2 Sample</sample>`.



2 Return to the first page and copy these three lines of service lookup code from the testing page:

```
String wsdlUrl = "http://localhost:7001/async/Enterprise1?WSDL";
Enterprise1 service = new Enterprise1_Impl( wsdlUrl );
Enterprise1Port port = service.getEnterprise1Port();
```



3 Use the Class wizard (File | New Class) to write a test client. Note that in this example, we use the service lookup code from the testing home page. Here is an example of a test client for this sample project. There are two methods to invoke in the service: `pollingMethod()` and `callbackMethod()`. The `pollingMethod()` keeps polling to see if the invocation is complete. The `callbackMethod()` registers a callback, which will be called when the system receives a response.

```
package async;
import weblogic.webservice.async.FutureResult;
import weblogic.webservice.async.AsyncInfo;
import weblogic.webservice.async.ResultListener;
import weblogic.webservice.async.InvokeCompletedEvent;
import async.generated.*;
import javax.xml.rpc.*;
import java.io.*;
import java.rmi.RemoteException;
```

```
public class Test {

  public static void main(String[] args) {
    System.out.println("calling polling method");
    pollingMethod();
    System.out.println("calling callback method");
    callbackMethod();
  }

  private static void pollingMethod() {
    try {

      String wsdlUrl = "http://localhost:7001/async/Enterprise1?WSDL";
      Enterprise1 service = new Enterprise1_Impl( wsdlUrl );
      Enterprise1Port port = service.getEnterprise1Port();

      //For the send operation
      //Make the request call. Since this is an async call, it is done with
      //a pair of methods - start<MethodName> and end<MethodName>.
      //The start method returns a future result data type, where a result
      //will be populated by the system at a later time
      FutureResult futureResult = port.startSendName(
          new async.generated.Bean1(), null);
      //Keep polling FutureResult, if it isn't done
      while( !futureResult.isCompleted() ){
        Thread.sleep( 300 );
      }
      //Future result is done, so get result data. In this case, it's a void
      //so there isn't a return value ... just want to know it's done.
      port.endSendName(futureResult);


      // For the Recv operation
      //The Server MDB, which is listening on the Queue, will be invoked
      //asynchronously. The MDB processes the request from the Queue.
      //The following client code picks up that response data asynchronously.
      futureResult = port.startRecvName(null);
      while( !futureResult.isCompleted() ){
        Thread.sleep( 300 );
      }
      //Future result is done, so get result data. In this example, the result
      //data is a Bean2 data type. Get and print data out.
      async.generated.Bean2 bean2 = port.endRecvName(futureResult);
      System.out.println("bean2: " + bean2.getSample());
      //flush the stream
      System.out.println(" ");
    }
    catch (Exception ex) {
    }
  }

  private static boolean done;
  private static void callbackMethod() {
    try {

      String wsdlUrl = "http://localhost:7001/async/Enterprise1?WSDL";
```

```
        Enterprise1 service = new Enterprise1_Impl( wsdlUrl );
        Enterprise1Port port = service.getEnterprise1Port();

        //For the send operation
        //Make the request call. Since this is an async call, it is done with a
        //pair of methods - start<MethodName> and end<MethodName>.
        //The start method returns a future result data type, where a result will
        //be populated by the system at a later time
        FutureResult futureResult = port.startSendName
           (new async.generated.Bean1(), null);
        //Keep polling FutureResult, if it isn't done
        while( !futureResult.isCompleted() ){
          Thread.sleep( 300 );
        }
        //Future result is done, so get result data. In this case, it's a void so
        //there isn't a return value ... just want to know it's done.
        port.endSendName(futureResult);

        // For the Recv operation
        // In the callback method, instead of polling again and again, we just
        // register a callback and wait.
        //The callback is defined by the AsyncInfo class. The onCompletion()
        //method will be called by the system, when response is available.
        AsyncInfo asyncInfo = new AsyncInfo();
        asyncInfo.setResultListener( new ResultListener(){
          public void onCompletion( InvokeCompletedEvent event ){
            Enterprise1Port source = (Enterprise1Port) event.getSource();
            try {
              //Since we are in the onComplete event, we know that the
              //response is available right now, so get response data.
              async.generated.Bean2 bean2 =
                  source.endRecvName(event.getFutureResult());
              System.out.println("bean2: " + bean2.getSample());
              //flush the stream
              System.out.println(" ");
            }
            catch (RemoteException e) {
              e.printStackTrace(System.out);
            }
            finally {
              done = true;
            }
          }
        });
        done = false;
        //Initialize the call by the start method.
        futureResult = port.startRecvName(asyncInfo);
        while(!done) {
          Thread.sleep( 300 );
        };
      }
      catch (Exception ex) {
      }
    }
  }
```

4 Right-click `Test.java` in the project pane and choose Run Using Test. This test client returns these results:

```
calling polling method
bean2: Bean2 Sample
calling callback method
bean2: Bean2 Sample
```

Understanding the asynchronous client stub

The special asynchronous methods used in the WebLogic toolkit take the following form:

```
FutureResult startMethod (params, AsyncInfo asyncInfo);
result endMethod (FutureResult futureResult);
```

where:

- *Method* is the name of the synchronous method used to invoke the web service operation.
- *params* is the list of parameters of the operation.
- *result* is the result of the operation.
- `FutureResult` is a WebLogic object used as a placeholder for the impending result.
- `AsyncInfo` is a WebLogic object used to pass additional information to WebLogic Server.

Note If the operations of the web service are document-oriented, rather than RPC-oriented, the Export As An Asynchronous Web Service wizard also generates the following `end()` method, in addition to the methods listed above:

```
result endConvenienceMethod (FutureResult futureResult);
```

If you use convenience methods when invoking document-oriented web service operations, then use this flavor of the `end()` method when invoking the operation asynchronously. For example, assume the standard generated stub contains the following method to invoke a web service operation called setSample:

```
setSample();
```

The Export As An Asynchronous Web Service wizard generates the following additional asynchronous methods in the generated stub:

```
FutureResult startEchoString (String str, AsyncInfo asyncInfo);String
endEchoString (FutureResult futureResult);
```

For more information about the `FutureResult` interface and the `AsyncInfo` class, see the `weblogic.webservice.async` Javadoc in the WebLogic documentation.

Writing asynchronous client code

When you write a Java client application to asynchronously invoke a web service operation, you must first import the following classes:

```
import weblogic.webservice.async.FutureResult;
import weblogic.webservice.async.AsyncInfo;
import weblogic.webservice.async.ResultListener;
import weblogic.webservice.async.InvokeCompletedEvent;
```

There are two steps involved in invoking an asynchronous operation: the first starts the invocation and the second optionally retrieves the results of the completed operation.

Assume that your client application uses the following Java code to get an instance of the Enterprise1 stub implementation:

```
String wsdlUrl = "http://localhost:7001/async/Enterprise1?WSDL";
Enterprise1 service = new Enterprise1_Impl( wsdlUrl );
Enterprise1Port port = service.getEnterprise1Port();
```

Also assume that you want to invoke the recvName and sendName operations of the web service. The following paragraphs show two ways you can invoke this operation asynchronously. The simplest way is to simply execute the `startSendName()` client method, do some other task, then execute the `endSendName()` client method. You can also use the `FutureResult.isCompleted()` method to poll and ask whether the results have returned from the web service, as shown in the following excerpt:

```
FutureResult futureResult = port.startSendName(new Bean1(), null);
while( !futureResult.isCompleted() ){
  Thread.sleep( 300 );
}
port.endSendName(futureResult);
```

The `endMethod()` method, in this case `endSendName()`, blocks until the result is ready.

Alternatively, you can register an event and be notified when the data is ready. This is done using the `ResultListener` and `InvokeCompletedEvent` classes to set up a listener in your client application that listens for a callback indicating that the results of the operation have returned, as shown in the following excerpt:

```
// For the Recv operation
      AsyncInfo asyncInfo = new AsyncInfo();
      asyncInfo.setResultListener( new ResultListener(){
        public void onCompletion( InvokeCompletedEvent event ){
          Enterprise1Port source = (Enterprise1Port) event.getSource();
          try {
            Bean2 bean2 = source.endRecvName(event.getFutureResult());
            System.out.println("bean2: " + bean2.getSample());
            System.out.println(" ");
          }
```

```
        catch (RemoteException e) {
          e.printStackTrace(System.out);
        }
        finally {
          done = true:
      }
    });
    done = false;
    futureResult = port.startRecvName(asyncInfo);
    while(!done) {
      Thread.sleep( 300 );
    };
  }
  catch (Exception ex) {
  }
```

For a tutorial that uses polling and callbacks in a test client application, see "Tutorial: Creating an asynchronous web service with WebLogic Server 8.1" on page 12-15.

For more information on asynchronous web services and how to write test clients, see the WebLogic documentation at `http://edocs.bea.com`.

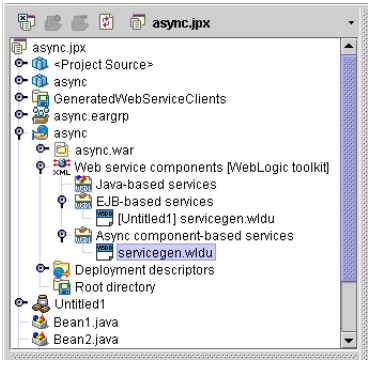
Deploying web services

The Export As A Web Service and Export As An Asynchronous Web Service wizards create XML web services deployment descriptor files called `servicegen.wldu` that provide deployment information to the WebLogic Server. When EJBs are exported as web services, which JBuilder automatically exports without using a wizard, a WLDU file is generated for each EJB module in the project. If you're exporting a Java class as an asynchronous web service, two WLDU files are generated, one for the EJB and one for the asynchronous service.

Understanding WLDU files

The `servicegen.wldu` file displays in the project pane as a child node of the WebApp node hosting the web service. If a Java class is exported, then it displays in the Java-based Services node of the WebApp (WebApp | Web Service Components | Java-based Services). EJB WLDU files display in the EJB-based Services node of the WebApp. If you have multiple EJB modules in your project, a `servicegen.wldu` file is generated for each EJB module, such as `[ejbmodule1]servicegen.wldu`, `[ejbmodule2]servicegen.wldu`, and so on. If you are exporting a class as an asynchronous web service, two WLDU files are generated: `[ejbmodule-name]servicegen.wldu` in the EJB-

based Services node and `servicegen.wldu` file in the Async Component-based Services node, which contains JMS-specific information.



The WLDU file, which contains everything within the `<servicegen>` element in `servicegen.xml`, provides the information the server needs to know about the service to create the classes, such as the EAR name, WAR name, class name, service name, service URI, and so on. You can modify this file at any time and change any of the values. For more on `servicegen.xml`, see "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com/`.

Sample WLDU file

```
<?xml version="1.0"?>
<servicegen destEar="bean.ear" overwrite="true" warName="weblogic.war">
    <service javaClassComponents="bean.Bean1" serviceName="Bean1"
        serviceURI="bean" targetNamespace="bean" protocol="http"
        style="rpc" expandMethods="true" generateTypes="true"/>
</servicegen>
```

Sample EJB WLDU file

```
<?xml version="1.0" encoding="UTF-8"?>
<servicegen destEar="Employee.ear" overwrite="true"
    warName="web-services.war" contextURI="web-services">
    <documentation>[JBUILDER marker begin 1048543643843 JBUILDER marker end]</
documentation>
    <service serviceName="EmpSession" serviceURI="ejb20module"
        targetNamespace="ejb20module" protocol="http" style="rpc"
        expandMethods="true" generateTypes="true"
        ejbJar="ejb20module.jar" includeEJBs="EmpSession">
        <documentation>JBUILDER should auto generate this entry: YES(YES/NO)
            </documentation>
    </service>
</servicegen>
```

Sample asynchronous WLDU file

```
<?xml version="1.0" encoding="UTF-8"?>
<servicegen destEar="untitled3.ear" overwrite="true"
    warName="asynchronous.war" contextURI="asynchronous">
    <service serviceName="Enterprise1" serviceURI="Untitled1"
        targetNamespace="Untitled1" protocol="http" style="rpc"
        expandMethods="true" generateTypes="true"
        JMSConnectionFactory="fact" JMSDestinationType="queue"
        JMSDestination="dest" JMSAction="send"
        JMSOperationName="sendName" JMSMessageType="untitled3.Bean1"/>
     <service serviceName="Enterprise1" serviceURI="Untitled1"
        targetNamespace="Untitled1" protocol="http" style="rpc"
        expandMethods="true" generateTypes="true"
        JMSConnectionFactory="fact" JMSDestinationType="queue"
        JMSDestination="rec" JMSAction="receive"
        JMSOperationName="recvName" JMSMessageType="untitled3.Bean1"/>
</servicegen>
```

Editing WLDU files for EJBs

You can customize the EJB deployment of your web services in the following two ways:

- "Modifying the EJB deployment node properties" on page 8-27
- "Editing the WLDU and modifying the <documentation> element" on page 8-28

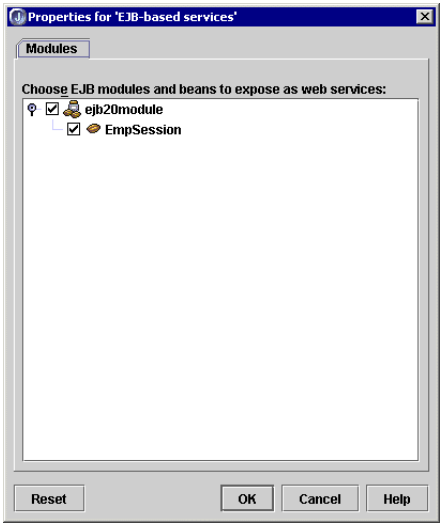
Modifying the EJB deployment node properties

By default, JBuilder automatically exposes all the stateless session beans with business methods in the remote interface. You can override this default behavior and select only the Enterprise JavaBean (EJB) modules, beans, and methods that you want to expose as web services.

You can change the automatic deployment behavior on the Modules page of the EJB-based Services Properties dialog box:

1 Expand the WebLogic WebApp node in the project pane, expand the Web Service Components node, right-click the EJB-based Services node, and choose Properties. On the Modules page, choose the EJB modules and stateless session beans to expose as web services. The stateless session beans must have at least one valid method in the

remote interface. The default behavior is to expose all of these beans, which are selected on the Properties page.



2. Uncheck any EJB modules and/or beans that you don't want exposed as a web service.
3. Click OK to close the EJB-based Services Properties page.
4. Rebuild your project and redeploy.

Editing the WLDU and modifying the <documentation> element

In the case of EJBs, once you've modified the WLDU file, you need to instruct the WebLogic toolkit that you don't want the WLDU file to be auto-generated and overwritten by JBuilder when the project is built. The `servicegen.wldu` for an exported EJB has a `<documentation>` element for each `<service>` element that you can modify to prevent the toolkit from overwriting your changes. Change the entry from YES to NO in the `<documentation>` element for each `<service>` element that you modified:

```
<documentation>JBUILDER should auto generate this entry: NO(YES/NO)
    </documentation>
```

After you modify this entry, the toolkit will **not** overwrite your edits in the `servicegen.wldu`. Rebuild the project for the changes to take effect.

web-services.xml

When you build your project and deploy it to the WebLogic Server, the WLDU deployment files are transferred to the WebLogic toolkit-specific `servicegen.xml` file which is used to execute the `servicegen` Ant task. The servicegen Ant task takes the EJB JAR or a list of Java classes and generates the components for the web service and a web services deployment file, `web-services.xml`. The `web-services.xml` deployment descriptor file, which is located in the WAR of the EAR, contains information that describes a WebLogic web service, such as the backend components that implement the web service, the non-built-in data types used as parameters and return values, the SOAP message handlers that intercept SOAP messages, and so on. For more information on the servicegen Ant task and `web-services.xml`, see the WebLogic documentation, "Programming WebLogic Web Services" at `http://edocs.bea.com/wls/docs70/webserv/index.html` for version 7.0 or `http://edocs.bea.com/wls/docs81/webserv/index.html` for version 8.1.

Note The WebLogic toolkit works directly with the EAR. Therefore, the `web-services.xml` and other web services components are only present in the EAR and not in the WebApp.

Sample web-services.xml file

```
<web-services>
          <web-service name="stockquotes" targetNamespace="http://example.com"
                       uri="/myStockQuoteService">
               <components>
                    <stateless-ejb name="simpleStockQuoteBean">
                         <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
                    </stateless-ejb>
               </components>
               <operations>
                    <operation method="getLastTradePrice"
                               component="simpleStockQuoteBean" />
               </operations>
          </web-service>
     </web-services>
```

Manually deploying services with web-services.xml

If you choose to create web service components manually and add other elements, such as typemapping and handler chains, you need to uncheck the Regenerate Deployment option on the Build page (Project | Project Properties | Build | Web Services). When this option is unchecked, JBuilder uses the manually created components and doesn't generate web services automatically.

To manually create components and add other elements, follow these basic steps:

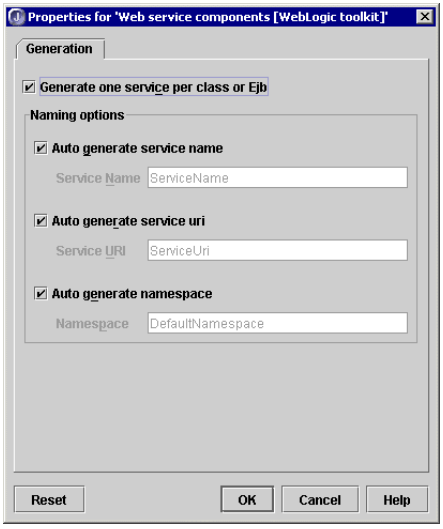
1. Create `web-services.xml` and save it to the `<web context>/WEB-INF` directory of your project.
2. Create the web service helper classes in the project, such as serializers.
3. Uncheck the Regenerate Deployment option on the Web Services tab of the Build page (Project | Project Properties | Build | Web Services).
4. Build the project.

For more information on manually creating `web-services.xml`, see "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com`.

Setting service naming defaults

You can modify the service naming defaults in the Web Service Components [WebLogic toolkit] Properties dialog box. Right-click the Web Service Components node in the project pane and choose Properties. Here you can choose how to generate the service and choose naming options. If you choose to auto generate the service name and the service URI, the bean name is used by default. For more information on these options, choose the Help button.

If you modify these properties, they are immediately reflected in the WLDU files for EJBs. But for Java classes, since they are statically created by wizards, the changes won't be reflected unless you export the Java classes again as a web service.



Testing deployed services

After you've deployed a web service with the WebLogic toolkit, you can use the Web Services Console to view the WSDL, test the service, and view the SOAP messages between the client and the server. If multiple ServiceUri names exist in the project, the Console displays a main page that lists all the ServiceUri names, which are hyperlinked to the deployed services. If a single service is deployed or multiple services with the same ServiceURI name, the Web Services Console displays the WebLogic Web Services Home Page.

Important To use the Web Services Console, a web browser must be configured for the WebLogic Server. For more information, see "Using JBuilder with BEA WebLogic servers" in *Developing J2EE Applications*.

Figure 8.1 Web Services Console: multiple ServiceURIs

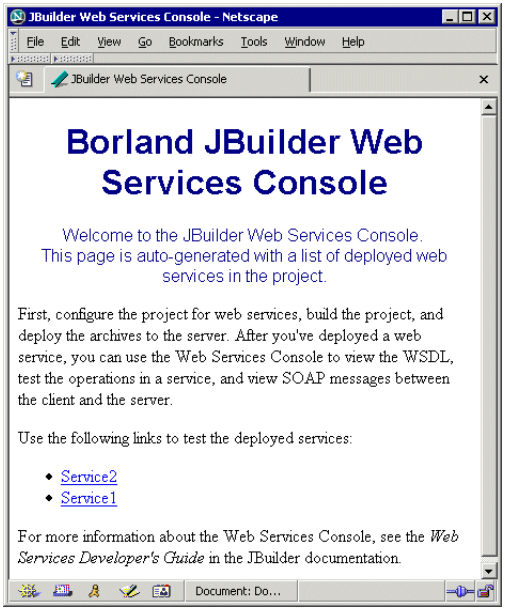
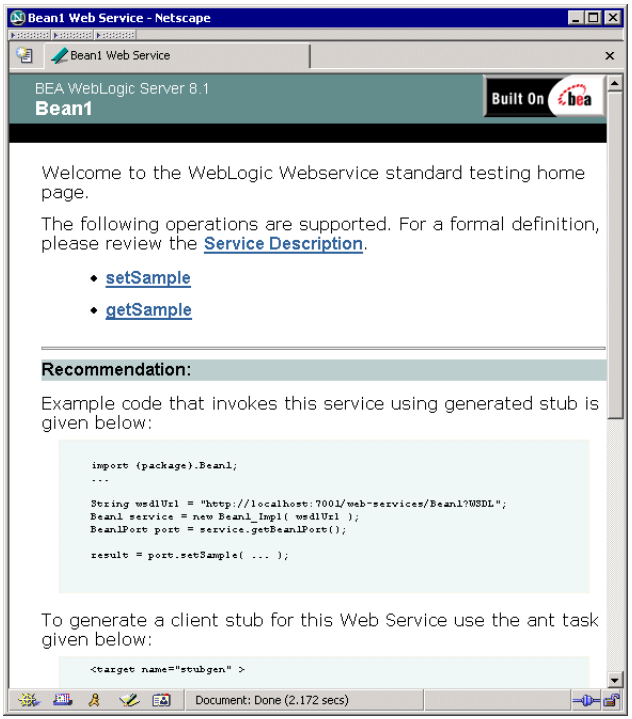


Figure 8.2 Web Services Console: WebLogic Web Services Home Page



After running the server and deploying the service as described in "Exporting a class as a web service" on page 8-2, use the Web Services Console to test the deployed service as described in the following steps:

1 Choose Tools | Web Services Console to open the Console where you'll find the home page for the web service.

   The home page displays the name of the service, a link to the Service Description (WSDL), the methods exposed in the service, the SOAP requests and responses, and example code for invoking the service. The address for the deployed service is automatically entered in the address field of the Console.

   The address to invoke the web service is in this form:

   ```
   <protocol>://<host:port>/<contextURI>/<serviceURI>
   ```

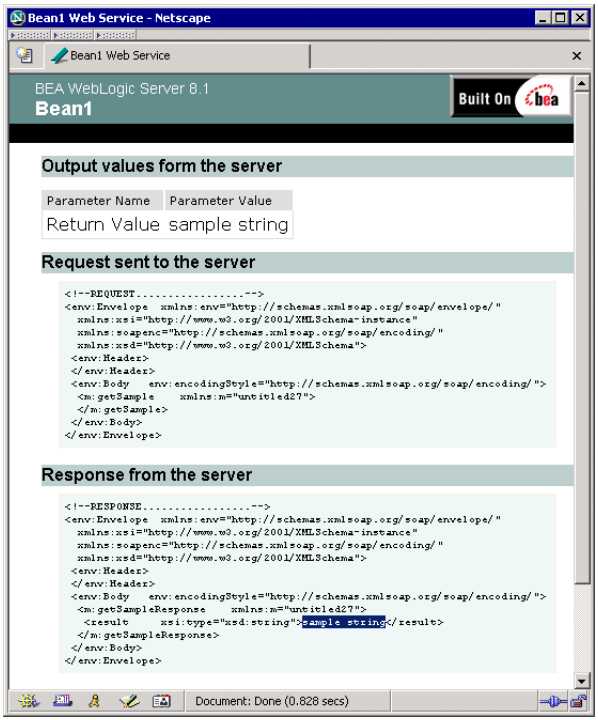
   where: `<protocol>` is the protocol for the service; `<host>` is the computer running WebLogic Server; `<port>` is the port number where WebLogic Server is listening; `<contextURI>` is the context root of the WebApp or the name of the WebApp archive file; and `<serviceURI>` is the Uniform Resource Identifier of the service.

   For example, if you're running WebLogic on the default port 7001 locally, the WebApp is `web-services`, and the ServiceUri is `Bean1`, the

address for the web service would be: `http://localhost:7001/web-services/Bean1`

2. Click the setSample link and choose the Invoke button to see the SOAP request and response to and from the server. Notice that the setSample value is set to "Sample."

3. Return to the first page, click the getSample link, and click the Invoke button to get the value of sample. The server response returns a value of "Sample."



4. Click the Services Description link to see the generated WSDL for the service.

The testing home page also has example code for invoking the service that you can copy, modify, and use to test the service you've deployed.

Writing a test client

After you've deployed a web service, you can use the example code from the Web Services Console to invoke the web service.

Write a test client as follows:

1. Choose Tools | Web Services Console to open the Web Services Console, which automatically loads the testing home page for the deployed service.
2. Copy the example code on the testing home page.
3. Choose File | New Class in JBuilder to create a new class file and click OK.
4. Paste the sample code into the new class.
5. Modify the code to invoke and test the service.

   For example,

```
package untitled1;

import untitled1.generated.*; //<client>.jar
import java.io.*;

public class Untitled1 {
  public static void main(String[] args){
  try {
          String wsdlUrl = "http://localhost:7001/web-services/untitled1?WSDL";
          Bean1 service = new Bean1_Impl(wsdlUrl);
          Bean1Port port = service.getBean1Port();
          System.out.println(port.getSample());
  }
  catch (IOException ex) {
    ex.printStackTrace();
    }
  }
}
```

6. Right-click the new test class and choose Run to run the test and invoke the web service.

See also

- "Using the Web Services Console" on page 4-6
- "The WebLogic Web Services Home Page and WSDL URLs" in "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com`

Chapter

9

Using the Apache SOAP 2 toolkit

This is a feature of JBuilder Enterprise. Support for the Apache SOAP 2 toolkit is not provided with the JBuilder WebLogic Edition.

If you're using the Apache SOAP 2 toolkit, you'll need to make some modifications when using the JBuilder wizards and do additional hand coding. Because SOAP 2 is an older version of Apache Axis and pre-dates the WSDL specification, there isn't any built-in WSDL support for SOAP. However, you can use the Import A Web Service and Export As A Web Service wizards and the Axis toolkit, because JBuilder provides an Axis to Apache SOAP cross-deployer that converts Axis deployment information into Apache SOAP.

Caution Note that Apache SOAP 2 is an older version of Axis and does **not** support JAX-RPC or WSDL. Due to these limitations, it's recommended that you use Axis instead.

For more information on the Apache SOAP 2 toolkit, see the documentation at the Apache web site at `http://ws.apache.org/soap/`

Create this simple example, and then make the appropriate changes in the wizards and the WSDL document.

1. Create a new project named `Untitled1` (File | New Project).
2. Create a JavaBean named `Bean1` with the JavaBean wizard (File | New | General), choose `java.lang.Object` as the base class, and check the options: Public and Generate Sample Property. Uncheck the Generate Main Method option.
3. Right-click `Bean1.java` in the project pane and choose Export As A Web Service. Because the project isn't configured for web services, the Web Services Configuration wizard opens first so you can configure the project.

4 Create a new WebApp named `soapsample` to host the service, choose Apache SOAP 2 as the toolkit in the Web Services Configuration wizard, and click Finish. Now the Export As A Web Service wizard opens.

5 Choose Axis as the toolkit in the Export As A Web Service wizard and make the changes specified in "Exporting a class as a web service" on page 9-2. Note that there isn't a SOAP toolkit available in the wizard because SOAP doesn't support WSDL.

Exporting a class as a web service

When exporting a class as a web service, you need to make the following changes in the Export As A Web Service wizard:

1 Uncheck the Generate Client Stub option on Step 1.

2 Change the Location URL on Step 2 of the wizard from `"http://localhost:8080/axis/services/Bean1"` to `"http://localhost:8080/<soapsample>/servlet/rpcrouter"`, where `<soapsample>` is the WebApp name.

3 Choose the methods you want to expose on Step 4.

   a Choose Allow All Methods from the Selection Mode drop-down list.

   b Select the methods you want to expose in the tree.

4 Click Finish to close the wizard.

Importing a WSDL

Before importing a WSDL to create a client, modify the WSDL. Then import the WSDL with the Import A Web Service wizard.

1 Modify the operation input and output namespace attributes in the binding section of the WSDL to match the port of the service. For example, change `namespace="http://untitled1"` to the port name, `"Bean1"`. The reason this change is required is that SOAP's dispatcher servlet needs to know about the service to dispatch and namespace is the mechanism it uses. Note that in the next generation of toolkits, such as Axis, this change isn't required because URL mapping is used.

WSDL before editing

```
<?xml version="1.0" encoding="UTF-8"?>
      <wsdl:definitions targetNamespace="http://untitled1"
       xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:apachesoap=
           "http://xml.apache.org/xml-soap"
       xmlns:impl="http://untitled1-impl" xmlns:intf="http://untitled1"
       xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
       xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     [...  rest of WSDL snipped for brevity  ...]
   <wsdl:binding name="Bean1SoapBinding" type="intf:Bean1">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/
          http"/>
      <wsdl:operation name="getSample">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="getSampleRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                  namespace="http://untitled1" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="getSampleResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                  namespace="http://untitled1" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="setSample">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="setSampleRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                  namespace="http://untitled1" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="setSampleResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                  namespace="http://untitled1" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="Bean1Service">
      <wsdl:port binding="intf:Bean1SoapBinding" name="Bean1">
         <wsdlsoap:address location="http://localhost:8080/soapsample/
             servlet/rpcrouter"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Edited WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
   <wsdl:definitions targetNamespace="http://untitled1"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://untitled1-impl"
    xmlns:intf="http://untitled1"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     [...  rest of WSDL snipped for brevity  ...]
   <wsdl:binding name="Bean1SoapBinding" type="intf:Bean1">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org
         /soap/http"/>
      <wsdl:operation name="getSample">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="getSampleRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
                soap/encoding/"
                 namespace="Bean1" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="getSampleResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/
                encoding/"
                 namespace="Bean1" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="setSample">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="setSampleRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/
                encoding/"
                   namespace="Bean1" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="setSampleResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/
                encoding/"
                   namespace="Bean1" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="Bean1Service">
      <wsdl:port binding="intf:Bean1SoapBinding" name="Bean1">
         <wsdlsoap:address location="http://localhost:8080/soapsample/servlet/
             rpcrouter"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

2 Right-click the edited WSDL file and choose Import A Web Service to open the Import A Web Service wizard.

3 Choose Apache Axis as the toolkit.

4 Uncheck the Generate Server-side Classes on Step 2 of the Import A Web Service wizard.

5 Change the package name on step 3 of the wizard to `untitled1.generated`. If you don't change the package name, the wizard will overwrite the `Bean1` class you're exporting.

6 Click Finish.

7 Build and run the project with the Web Services Server configuration.

8 Right-click the test case and run it to test the service.

If you prefer to hand code the client, follow this client example.

Client example

```
public class Test
{
  public static void main(String[] args) throws Exception
  {

    URL url = new URL("http://localhost:8082/soapsample/servlet/rpcrouter");
//  String nameToLookup = "Bean1";

    Call call = new Call();
    call.setTargetObjectURI("Bean1");
    call.setMethodName("getSample");
    call.setEncodingStyleURI(encodingStyleURI);

//  Vector params = new Vector();
//
//  params.addElement(new Parameter("sample", String.class,
        nameToLookup, null));
//  call.setParams(params);

    // Invoke the call.
    Response resp;
    try
    {
      resp = call.invoke(url, "");
    }
    catch (SOAPException e)
    {
      System.err.println("Caught SOAPException (" +
                         e.getFaultCode() + "): " +
                         e.getMessage());
      return;
    }
```

```
      // Check the response.
      if (!resp.generatedFault())
      {
        Parameter ret = resp.getReturnValue();
        Object value = ret.getValue();

        System.out.println(value != null ? "\n" + value : "I don't know.");
      }
      else
      {
        Fault fault = resp.getFault();

        System.err.println("Generated fault: ");
        System.out.println ("  Fault Code   = " + fault.getFaultCode());
        System.out.println ("  Fault String = " + fault.getFaultString());

      }
    }
  }
```

Chapter

10

Browsing and publishing web services

This is a feature of JBuilder Enterprise. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

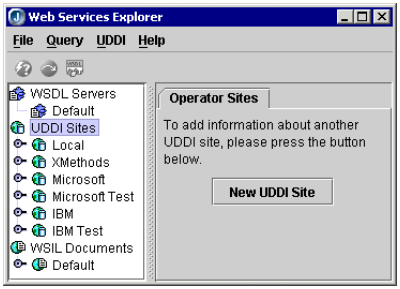
Because web services can be created by anyone and consumed by anyone, it's important to have a mechanism for registering, advertising, and locating a service. Currently, the UDDI (Universal Description, Discovery and Integration) framework can be used to register and discover a service. In addition to searching UDDI registries for web services, you can also search web sites for available web services using WSIL documents, as well as search for services located on Axis servers. JBuilder provides the Web Services Explorer to assist you in searching for and publishing web services.

Web Services Explorer overview

JBuilder provides the Web Services Explorer, available from the Tools menu, for searching available services at sites and for publishing web services to UDDI registries. Using the Web Services Explorer, you can search for businesses, web services, or tModels, as well as add business and tModel entries. You can also monitor message traffic to and from the UDDI sites in the UDDI Message Monitor. For more information about UDDI, see "UDDI overview" on page 10-3.

Important The UDDI feature of the Web Services Explorer requires the use of Sun's Java Secure Socket Extension (JSSE). If you're using JDK 1.3 or earlier, you must download and install the JSSE Extension from Sun at `http://java.sun.com/products/jsse/index-103.html`.

The Web Services Explorer also supports browsing network sites that run Apache Axis-hosted web services. These sites provide overview information, which describes the web services available at that site. Each web service is described by a WSDL document. For more information about the Axis features in the Web Services Explorer, see "Axis overview" on page 10-5.



In addition, the Web Services Explorer supports browsing of Web Services Inspection Language (WSIL) documents. WSIL documents are essentially a collection of references to available web services on a web site that are hosted by the web service provider. These references may point to other WSIL documents, UDDI entries, or WSDL documents.

The Web Services Explorer also provides a link to the Import A Web Service wizard, which generates Java classes from a selected Web Services Description Language (WSDL) document. Browse to any web service available on a UDDI site, Axis site, or other web site listing services and select the node that specifies the WSDL document. Then choose the Import A Web Service button in the Explorer to quickly generate Java classes. If you don't have a project open, the Project wizard opens first, then the Import A Web Service wizard opens. After creating a web service, you can then use the publishing features of the Web Services Explorer to publish your web service to UDDI registries.

See also

- "Adding and deleting nodes in the Explorer's tree" on page 10-6
- "Web Services Explorer menus" in the online help
- Chapter 5, "Working with WSDL"
- "Publishing businesses and services" on page 10-22
- "Tutorial: Browsing UDDI web services" on page 13-1

UDDI overview

The *Universal Description, Discovery and Integration* (UDDI) framework provides a mechanism for locating, describing, and registering web-based services in a central business registry over the Internet. Through this registry, industries can find business partners and dynamically connect to their web services, as well as publish services.

This XML-based framework uses Simple Object Access Protocol (SOAP) and HTTP to discover a particular service in the registry using XML messages to make remote procedure calls and send information to and from the registry. XML, HTTP, and SOAP provide the added advantage of language independent, cross-platform programming.

A UDDI registry doesn't contain the actual specifications about how businesses do things electronically to share their products and services. Instead, it contains pointers or references that link to this information. Some of these pointers may be WSDL (Web Services Description Language) files, which describe a web service.

UDDI registries can be private, public, or even local on your own computer. Some registries, such as the UDDI Business Registry, are updated and replicated with the same information by several UDDI operator sites. Other registries are public but do not replicate their data with other sites.

The UDDI Business Registry is a universal UDDI registry that is maintained and operated as a distributed service by several operator sites. These operator sites, such as Microsoft and IBM, replicate data with each other to synchronize the instances of the registry. They also have UDDI test sites where you can test your web services thoroughly before you publish them.

UDDI accepts and organizes three types of business information:

- Publish - "white pages" containing business information, such as contact and address information.
- Find - "yellow pages" containing business service information, organized by industrial categories and geographic location.
- Bind - "green pages" containing technical information and specifications about services, necessary for interfacing programmatically.

See also

- "Searching a UDDI registry" on page 10-7
- "Publishing web services to a UDDI registry" on page 10-21

- The UDDI project at `http://www.uddi.org`

- Microsoft's UDDI site at `http://uddi.microsoft.com`
- IBM's UDDI site at `https://www-3.ibm.com/services/uddi/protect/registry.html`

UDDI terms and definitions

The Web Services Explorer uses many UDDI data types and terms that are found in the UDDI specification. Although this documentation assumes familiarity with the UDDI specification, some key UDDI terms are defined here. For complete information, see the UDDI specification at `http://uddi.org/specification.html`.

Table 10.1 UDDI terms

UDDI term	Definition
accessPoint	An entry point address for accessing a web service. This could be a URL, email address, or phone number.
businessKey	A unique identifier for a given business entity.
category	Category information that is similar to an identifier but that uses a predefined taxonomy, such as industry, product, or geographic codes. It also uses name/value pairs. Examples of these codes include: NAICS (North American Industry Classification System), UNSPSC 3.1 (United Nations Standard Products and Services Code System), and SIC (Standard Industrial Classification) codes.
identifier	A name/value pair that acts as an identifier for a business. Examples include the D-U-N-S (Dun & Bradstreet's Data Universal Numbering System) ID and Thomas Register ID.
keyName	Commentary that aids in readability but usually isn't required. For example, the D-U-N-S identifier for IBM has a keyName of D-U-N-S, whereas the keyValue is the actual nine-digit D-U-N-S number, 00-136-8083. However, the keyName does have important meaning in the uddi-org:misc-taxonomy category.
keyValue	Specifies a business classification, such as the nine-digit D-U-N-S number used for businesses worldwide.
operator site	Each instance of a UDDI business registry. For example, Microsoft and IBM maintain instances of the UDDI Business Registry.
overview document URL	A pointer to the location of the overview document description.
serviceKey	A unique key for a business service that is generated when the service is registered.
tModel	A reference to a technical specification of a service. TModels are descriptions of web services that define service types. Each TModel has a unique identifier and points to a specification that describes the web service. tModels provide a common point of reference that allow compatible services to be easily identified.

Table 10.1 UDDI terms (continued)

UDDI term	Definition
tModelInstance details	A list of tModel information structures which acts as a fingerprint for the service. Includes such details as description, tModel name, tModel key, and documentation.
tModelKey	A unique identifier for a tModel that is assigned by UDDI when a service is registered.

Axis overview

This is a feature of the Axis toolkit. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

The Web Services Explorer allows you to browse network sites that run Apache Axis-hosted web services. Searching these sites returns information that describes the web services available in the form of a WSDL document. JBuilder builds Axis servers with remote access disabled. To browse services on Axis servers, remote access must first be enabled. See "Accessing remote Axis servers" on page 10-17.

See also

- "Searching an Axis server for web services" on page 10-15
- "Publishing web services from an Axis server" on page 10-24
- Chapter 7, "Using the Apache Axis toolkit"

WSIL overview

The Web Services Inspection Language (WSIL), like UDDI, provides a method of service discovery for web services. Unlike UDDI, WSIL uses a decentralized, distributed model, rather than a centralized model. WSIL documents, which are essentially pointers to lists of services, allow consumers of web services to browse available services on web sites not listed in the UDDI registries. The WSIL specification provides standards for using XML-formatted documents to inspect a site for services and a set of rules for how the information is made available. A WSIL document gathers multiple references to pre-existing service description documents in one document. The WSIL document is then hosted by the provider of the service, so consumers can find out about available services.

See also

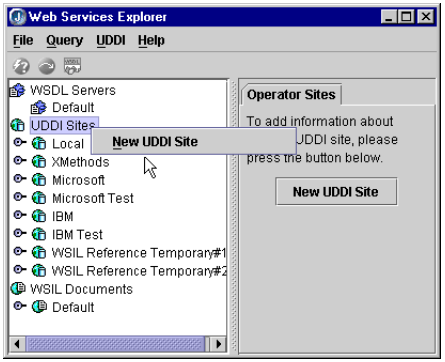
- "Searching web services with WSIL documents" on page 10-18

Adding and deleting nodes in the Explorer's tree

The Web Services Explorer has several nodes available in the tree view to help you search for available web services at different locations:

- WSDL Servers, a feature of the Axis toolkit
- UDDI Sites
- WSIL Documents

Expand the WSDL Servers node, a feature of the Axis toolkit, and see that it has a Default node, which points to a common location for a test environment. The UDDI Sites node has multiple UDDI nodes already available, such as Local, XMethods, Microsoft, IBM, and UDDI test sites. The WSIL Documents node also contains a Default node. You can add nodes to the existing nodes and delete nodes in the Explorer's tree.



The contents of the WSDL Servers and UDDI Sites nodes in the Explorer are read from XML files delivered in the JBuilder `defaults` directory. Once you make a change to one of these nodes, the changes are saved to your `<home>/<.jbuilder>` directory. The WSIL Documents file is generated after you add a node and is also saved to the `<home>/<.jbuilder>` directory. The file names are as follows:

- WSDL Servers node (Axis servers): `AxisServers.xml`
- UDDI Sites node: `UDDIOps.xml`
- WSIL Documents node: `WSILDocs.xml`

To add a new node to the top-level nodes in the Explorer's tree,

1. Choose the node in the tree view. A page displays on the right.
2. Create a new node using one of the following methods:
   - Click the New button on the page to the right.
   - Choose File | New.
   - Right-click the node and choose New.

3 Enter any name you like for the new node in the Input dialog box and click OK.

4 Select the new node in the tree to display the detail information on the right.

5 Enter the appropriate information, such as Publish and Inquiry URL for UDDI sites or location URL for a WSIL document.

6 Choose the Save button.

To delete a node from the Explorer's tree,

1 Choose the node you want to delete to display the Details page on the right.

2 Choose one of these methods to delete the selected node:

- Click the Delete button on the Details page.
- Choose File | Delete.
- Right-click the node in the tree and choose Delete.

Note The top-level nodes can't be deleted.

Searching a UDDI registry

There are several ways to search for web services in a UDDI registry with the Web Services Explorer:

- Search for businesses
- Search for services
- Search for tModels

Important You can increase the timeout for the Web Services Explorer. Choose UDDI | Connection Settings and enter the new timeout in the Timeout The server may also truncate your query results if it determines that the results are too large.

Searching for businesses

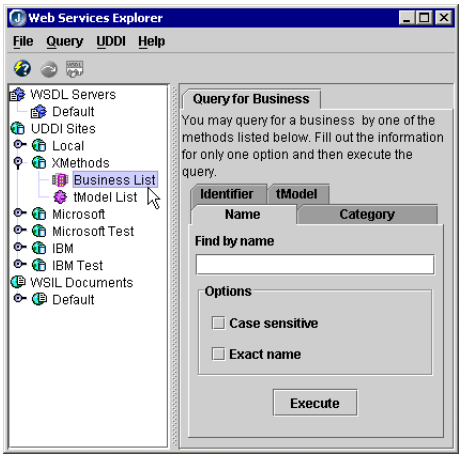
You might want to find a particular business that you like to work with and see what types of services they offer. Or you may want to find businesses in a particular industry, such as software publishing. There are several ways to search for a business using the Web Services Explorer:

- Search by business name
- Search by business category
- Search by business identifier
- Search by tModel

Searching by name

To search for a specific business by name in the Web Services Explorer, complete the following steps:

1. Choose Tools | Web Services Explorer to open the Explorer.
2. Expand the UDDI Sites node in the tree on the left and double-click a UDDI operator site node to expand it.
3. Select the Business List node to display the Query For Business page on the right.



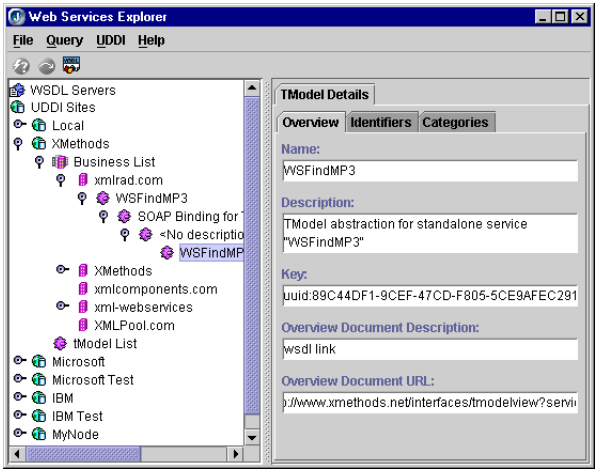
4. Enter the business name or the first few letters of the business name in the Find By Name field on the Name tab. If you leave this field blank, all the businesses at the site are found.

Note The Web Services Explorer supports the % symbol as a wildcard.

5. Execute the query with any of these methods:

- Choose Query | Execute.
- Click the Execute Query button on the toolbar.
- Click the Execute button on the Query For Business page to the right of the tree.
- Press *Enter* in the Find By Name field after entering the search name.

6 Expand the Business List node to see the query results and drill down to find out more about a business and its services.



Searching by category

Another method of searching for a business is to search by business category. There are two business classifications you can search by:

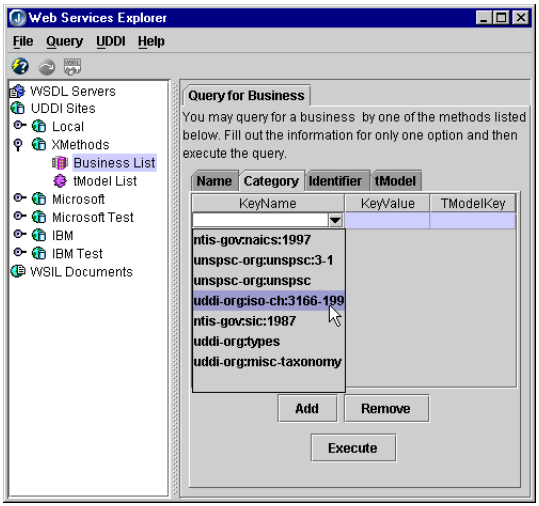
- ntis-gov:naics:1997 (North American Industry Classification System)
- unspsc-org:unspsc:3-1 (United Nations Standard Products and Services Code System)
- unspsc-org:unspsc (Universal Standard Products and Services Classification)
- uddi-org:iso-ch:3166-1999 (codes for geographic location)
- ntis-gov:sic:1987 (Standard Industrial Classification)
- uddi-org:types
- uddi-org:misc-taxonomy

Note Other classification systems can also be used. If a scheme is not available in the drop-down list, you can paste the key for the scheme into the tModelKey field. You can also add and remove search criteria using the Add and Remove buttons.

Each of these business classification systems has its own codes for the various categories. Large businesses that do a wide variety of business may be classified in several of these systems and under several classifications in each system. For example, a company might sell computer hardware and software. In NAICS, this business might be listed with several classifications, such as computer training, data processing services, and software publishers. This same business could also be

classified in UNSPSC as computer programmed instructions, database software, and mainframe computers.

These business classifications are listed in the KeyName drop-down list on the Category page of the Web Services Explorer. For further classification, a KeyValue is used to give a more specific description of the business. Each business classification also has a corresponding tModelKey.



To search by business category,

1. Choose Tools | Web Services Explorer to open the Explorer.
2. Expand the UDDI Sites node in the tree on the left and double-click a UDDI operator site node to expand it.
3. Select the Business List node to display the Query For Business page on the right.
4. Choose the Category tab on the Query For Business page.
5. Click in the KeyName field to activate the drop-down list and choose one of the business categories.
6. Enter an appropriate code for the selected business category in the KeyValue column.

   For example, if you want to search for all software publishers at a UDDI site, you could choose `ntis-gov:naics:1997` from the KeyName drop-down list, and enter the NAICS software publisher's code in the KeyValue field: `51121`. Each business category type has its own set of codes for classification.
7. Press *Enter* to commit the KeyValue.

8 Click the Execute button on the Query For Business page to execute the query.

9 Expand the Business List node to see the list of businesses registered as the selected category.

Searching by identifier

Another method of searching for a business is to search by business identifier. There are several built-in identifier schemes you can search by:

- thomasregister-com:supplierID
- dnb-com:D-U-N-S (Dun & Bradstreet Number Identifier System)

These IDs are listed in the KeyName drop-down list on the Identifier page of the Web Services Explorer. Each of these classifications has a corresponding tModelKey.

To search by business identifier,

1 Choose Tools | Web Services Explorer to open the Explorer.

2 Expand the UDDI Sites node in the tree.

3 Expand a UDDI operator site node and select the Business List node to display the Query For Business page on the right.

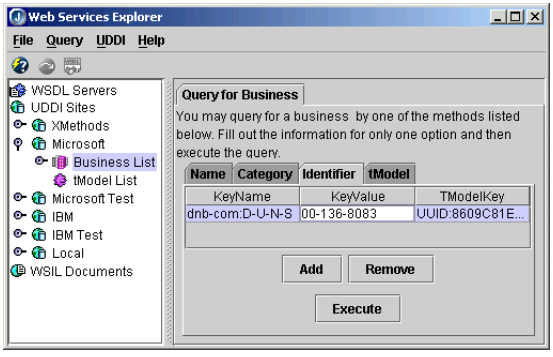
4 Choose the Identifier tab on the Query For Business page.

5 Choose one of these business identifiers from the KeyName drop-down list:

   - thomasregister-com:supplierID
   - dnb-com:D-U-N-S

6 Enter an appropriate code for the selected business identifier in the KeyValue column.

   For example, if you want to search for a business with a D-U-N-S ID of 00-136-8083 at a UDDI site, you would choose `dnb-com:D-U-N-S` from the KeyName drop-down list, and enter the ID number in the KeyValue field: `00-136-8083`. This is IBM's D-U-N-S ID.



7 Press *Enter* to commit the KeyValue.

8 Click the Execute button to execute the query.

9 Expand the Business List node to see the business registered with the selected ID.

Searching for services

Querying by service can only be done within a particular business. You must first search for a business and select the business node in the tree on the left before querying for a service.

To query for a service, complete the following steps:

1 Choose Tools | Web Services Explorer to open the Explorer.

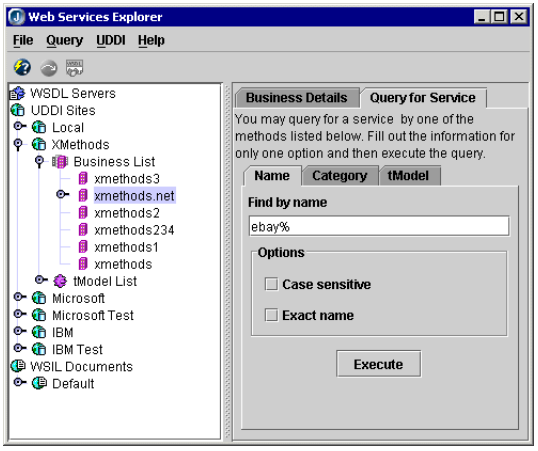
2 Expand the UDDI Sites node in the tree on the left.

3 Expand a UDDI operator site node and execute a Query For Business by name as explained in "Searching by name" on page 10-8.

4 Expand the Business List node to see the query results.

5 Select a business in the tree and choose the Query For Service tab on the right.

6 Enter the name of the web service in the Find By Name field.



7 Click Execute to run the query.

8 Expand the business node to see the services returned by the query.

Note You can also search for a service by category or tModel.

Searching for tModels

You can also search for tModels in the Web Services Explorer. TModels represent a technical specification for a web service. When searching for tModels you can search by tModel name, category, and identifier.

Searching by name

To search for tModels by name,

1 Choose Tools | Web Services Explorer to open the Explorer.

2 Expand the UDDI Sites node in the tree on the left and double-click a UDDI operator site node to expand it.

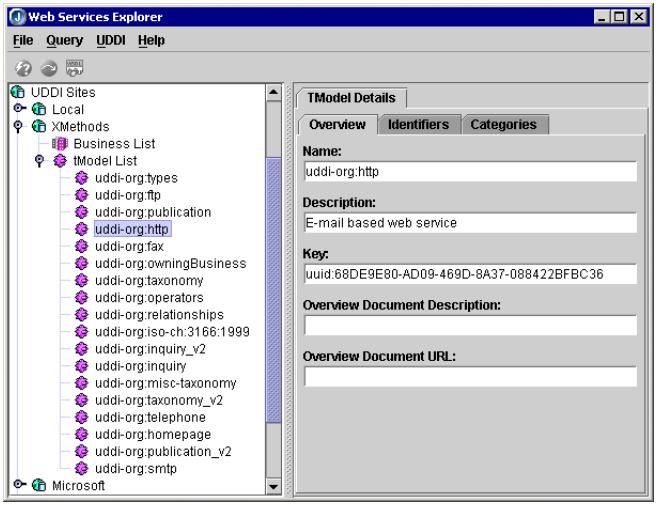
3 Choose the tModel List node in the tree to display the Query For TModel page on the right.

4 Enter the tModel name in the Find By Name field.

For example, if you wanted to find all tModels beginning with the uddi-org name, you would enter `uddi-org`.

5 Click the Execute button to execute the query.

6 Expand the tModel List node to see the results of the query.



Examining UDDI query results

Once you've executed a UDDI query, you can browse detailed information about a web service in the Web Services Explorer. Each node in the Explorer's tree displays different information about a selected service. When you select a node in the tree, the appropriate detail

information displays to the right of the tree. For example, when you select the business node in the tree, the Business Details page is displayed on the right with overview information about the business, as well as any contact information, identifiers, and categories. When you expand the business node and select a business service, the Service Details page displays, and so on.

UDDI detail pages

Queries for businesses and services return multiple nodes in the tree, such as the business name, service name, binding information, tModel Instance, and tModel details. Depending on the web service and how it's published, a service may not have all of these detail pages. Queries for tModels return only tModel nodes in the tree and display the TModel Details page only. Some of the details on these pages are required while others are optional, so some fields may be blank.

There are several UDDI detail pages in the Web Services Explorer:

- Details
- Business Details
- Service Details
- Binding Details
- TModel Instance Details
- TModel Details

Details page

The Details page displays when a UDDI operator site node is selected. This page displays operator site information, such as name, inquire URL, and publish URL.

Business Details page

The Business Details page displays when the business node is selected. This page displays company information, such as a company overview with business name and description, contact information, identifiers, and business categories.

Service Details page

The Service Details page displays when the web service node is selected in the tree. This page displays overview information about the service, such as the service name, description, and service key.

Binding Details page

The Binding Details page displays when the binding node is selected in the tree. This page displays a description, the access point to the service, and the URL type.

TModel Instance Details page

The TModel Instance Details page displays when the TModel Instance Details node, which often defaults to <No Description>, is selected in the tree. This page displays a description, tModel key, an overview document description, and an overview document URL.

TModel Details page

The TModel Details page displays when the tModel node, the last node in the Business List, is selected in the tree. It also displays when a tModel node in the TModel List is selected. This page displays overview information, such as the tModel name and description, and the tModel key, an overview document description, an overview document URL, and any relevant identifiers and categories.

If a WSDL document is available on this page, you can use the Import A Web Service wizard to generate Java classes from the WSDL. Choose File | Import A Web Service or click the Import A Web Service button. If you don't have a project open, the Project wizard opens first, then the Import A Web Service wizard opens. For more information on the Import A Web Service wizard, see Chapter 5, "Working with WSDL."

Searching an Axis server for web services

This is a feature of the Axis toolkit. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

Some web services are deployed to and available on network sites that run Apache Axis-hosted web services. You can browse these with the Web Services Explorer. If you're searching a remote Axis server, remote access on that server must be enabled. See "Accessing remote Axis servers" on page 10-17. In addition, you can publish an Axis web service to a UDDI registry. See "Publishing web services from an Axis server" on page 10-24.

You can also create Axis web services locally and browse them as follows:

1. Create a web service as described in "Tutorial: Creating a simple web service" on page 11-1.
2. Expand the axis node in the project pane and expand the Root Directory of the axis WebApp.
3. Right-click `index.html` in the project pane and choose Web Run Using "Web Services Server" to deploy the service and run the web server.
4. Choose Tools | Web Services Explorer to open the Explorer.
5. Display the web service as described in "Displaying services."

Displaying services

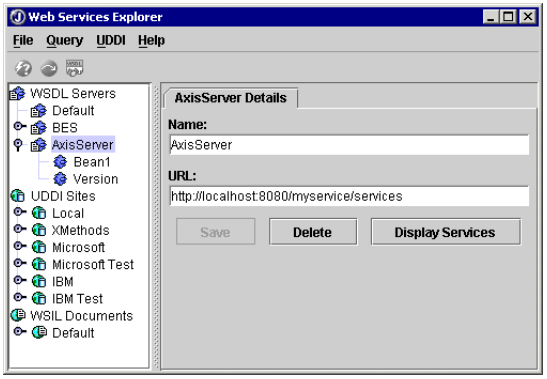
When web services are deployed to an Axis server, you can browse those services in the Web Services Explorer.

To display available services on an Axis server,

1. Expand the WSDL Servers node and create a new WSDL Servers child node for the Axis service as follows:
   a. Right-click the WSDL Servers node and choose New WSDL Server Site or click the New WSDL Server button on the WSDL Servers page.
   b. Enter a name for the new node, select Axis Server as the server type, and click OK.
2. Choose the new WSDL Server node and enter the URL for the server in the URL field or create a new WSDL Server node and enter the URL, including the web context of the service. The URL for a service on an Axis server is in one of these forms:

   ```
   <protocol:>//<port:port number>/<web context>/services/
   <protocol:>//<port:port number>/<web context>/servlet/AxisServlet/
   ```

   Where protocol is the protocol for the service; the port is the computer running the Axis server; port is the port number where the Axis server is listening; web context is the context root or name of the WebApp hosting the web service. For example, `http://localhost:8080/myservice/services`.
3. Click the Display Services button on the Details page.
4. Expand the node to view the available services. Note that Version is one of the services listed. This is an Axis built-in service that gets the version of Axis.



Importing a WSDL and publishing Axis web services

Once you've displayed the Axis-hosted services, you can import a WSDL for the service and generate Java classes for implementing and/or creating the service. To import a WSDL for a service and generate Java classes, select one of the Axis-hosted services and choose File | Import A Web Service or click the Import A Web Service button on the toolbar. Complete the steps of the Import A Web Service wizard, click Finish, and see the generated WSDL document and the Java classes. For more information, see Chapter 5, "Working with WSDL," or click the Help button in the wizard.

You can also publish Axis web services to a UDDI site in the Web Services Explorer. See "Publishing web services from an Axis server" on page 10-24.

Note Remote access must be enabled on remote Axis servers to browse their services. See "Accessing remote Axis servers" on page 10-17.

Accessing remote Axis servers

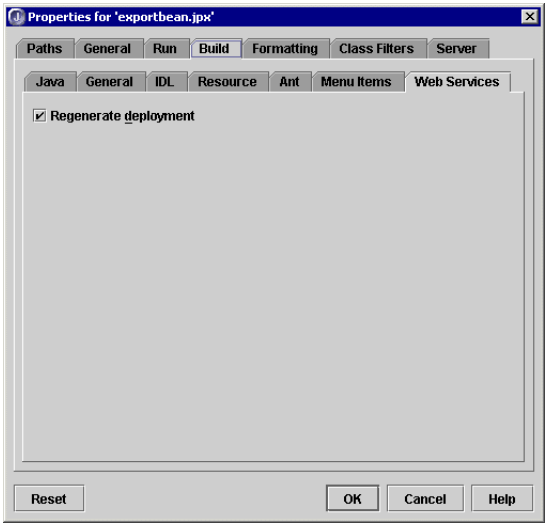
JBuilder builds Axis servers with remote access disabled. If you want to change this default behavior, you'll need to modify the `server-config.wsdd` file as follows.

Warning Enabling remote administration may give unauthorized parties access to your machine. Make sure to add security to your configuration.

To enable remote access:

1. Build the project.
2. Open the `server-config.wsdd` file located in the toolkit node of the WebApp hosting the service.
3. Change the value for the `AdminService` element from `false` to `true`: `<parameter name="enableRemoteAdmin" value="true"/>`.
4. Disable the regeneration of `server-config.wsdd` during deployment by unchecking the project's build SOAP option as follows:
   a. Choose Project | Project Properties to open the Project Properties dialog box.
   b. Choose the Build tab.

c Choose the Web Services tab on the Build page and uncheck the Regenerate Deployment option.



d Click OK to close the Project Properties dialog box.

Searching web services with WSIL documents

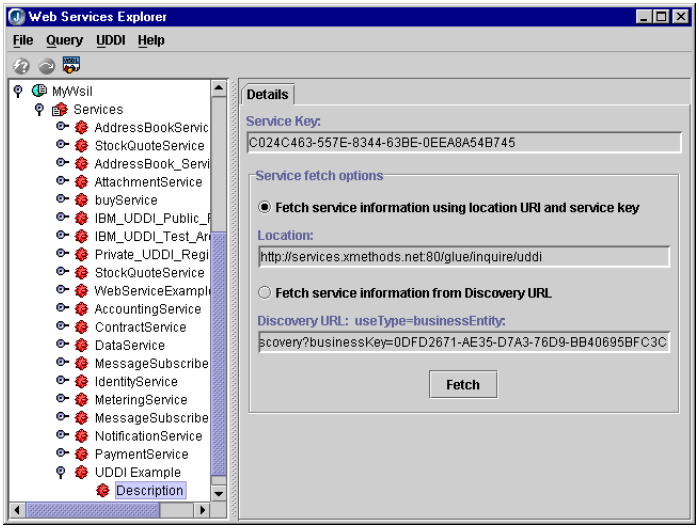
Another method of searching available web services is to search with WSIL documents. WSIL documents are collections of pointers to other documents that list web services available on a web site. WSIL documents can point to other WSIL documents, a UDDI business or service entry, and WSDL documents. Usually web sites with available web services gather their links into one WSIL document at a default location, such as `http://www.xmethods.net/inspection.wsil`. Once you've found the service you want at a site, you can import the WSDL document with the Import A Web Service wizard.

According to the Web Services Inspection Language specification, WSIL documents must have at least one `<service>` element or one `<link>` element and may have both. Services and links are displayed in the Web Services Explorer as child nodes of a WSIL node.

Services node

The Services node displays the `<service>` elements in the WSIL document. The `<service>` element specifies the WSDL document for the service or it can refer to a UDDI entry that specifies the WSDL. Expand the Services node to display the available services on the site. Expand a web service and select the Description node beneath it to display the Details page. This

page provides such information as the location of the WSDL document, whether the service is an endpoint, and binding information.



If the reference is to a UDDI entry, you can search for services by two methods:

- Service key and Location URI
- Service key and Discovery URL

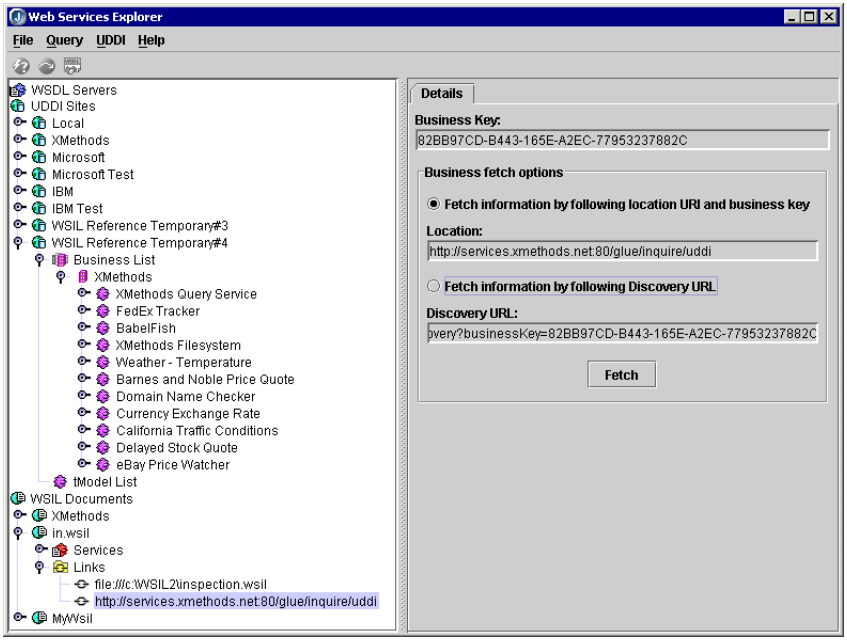
Choose an option and click the Fetch button to get the information. The search results are displayed in a temporary node, WSIL Reference Temporary, in the UDDI tree.

Links node

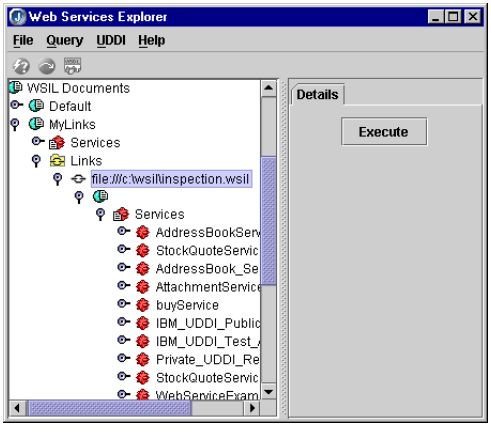
If any <link> elements are contained in the WSIL document, they display as nodes under the Links node. Links can be addresses to another WSIL document or to a UDDI business entry. Expand the Links node to see the <link> elements in the WSIL document and select a link to display the Details page.

If the link is a reference to a UDDI business entry, the Details page displays UDDI information you can fetch from the UDDI registry. The

search results are displayed in a WSIL Reference Temporary node under the UDDI node.



If the link refers to another WSIL document, you can click the Execute button on the Details page to display the list of services in the WSIL.



Executing a search with a WSIL document

To search for available services at a web site,

1. Select the WSIL Documents node in the Explorer.
2. Click the New WSIL Location on the WSIL Documents page on the right to create a new WSIL node.

Tip You can also right-click the WSIL Documents node and choose New or choose File | New.

3. Enter a name for the node and click OK.
4. Enter the web site URL or a local URL that you want to search. For example, `http://www.xmethods.net/inspection.wsil` or `file:///c:\wsil\inspection.wsil`.
5. Click Save to save the changes.
6. Expand the new WSIL node to see the available Services and Links nodes.
7. Do one of the following:
   - Expand the Services node and choose a Description node under one of the services to see the location of the WSDL document. You can then use the Import A Web Service wizard to import it and generate Java classes to consume the service.
   - Expand the Links node and choose a link to see what's available on the Details page.

     If there's a reference to a UDDI business entry, choose Fetch to obtain the information. The results are displayed in a WSIL Reference Temporary node under the UDDI Sites node. You can then drill down to a service's tModel node where the WSDL is located and use the Import A Web Service wizard to import it and generate Java classes to consume the service.

     If the reference is to another WSIL document, you can also browse those services and links. Choose the link with the WSIL reference and click the Execute button on the Details page. Expand the link node to display the services listed in the WSIL.

Publishing web services to a UDDI registry

The Web Services Explorer supports publishing new businesses, web services, and their corresponding tModels. In some cases, UDDI sites require registration before you can publish to them. Some UDDI sites have test sites where you can publish and test new services before distributing them on the official UDDI site. Most UDDI sites also require you to specify

a Publish URL before publishing your service. The Publish URL is available on the site's web site. For example, the Publish URL for the Microsoft UDDI site is `https://uddi.microsoft.com/publish`.

Publishing web services to a UDDI site in JBuilder involves the following steps:

1. Registering at the UDDI site through a web browser.
2. Creating and deploying a web service.
3. Publishing a business and service at the UDDI site.
4. Publishing a tModel for the service.

Registering at the UDDI site through a web browser

Before publishing businesses and web services, you need to register at the UDDI site where you want to publish your service. In many cases, you'll want to publish to a test site first and test your service. IBM and Microsoft provide test sites for this purpose. Visit the UDDI operator site with your web browser to register.

Creating and deploying a service

Use the JBuilder web services wizards to create your web service and deploy it according to the toolkit selected. For more information, see the appropriate chapter for the toolkit.

Publishing businesses and services

To publish a web service, you must first publish a business, if it doesn't exist, and then add information for the business, service, binding template, and tModel. If you already have a business published at a UDDI site, you can skip this step.

First, add the new business as follows:

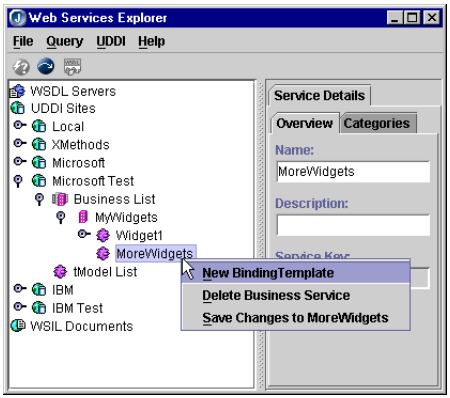
1. Choose Tools | Web Services Explorer to open the Explorer.
2. Choose a UDDI operator node, such as Microsoft Test, under the UDDI Sites node in the tree. The operator site must have a Publish URL associated with it. If the Publish URL field is blank, go to the operator's web site to find the Publish URL for that particular site. To enter the Publish URL for an operator site, choose an operator node and enter the URL in the Publish URL field on the Details page. Choose Save to save the entry.
3. Right-click the Business List node in the tree and choose New Business or choose File | New Business.

4 Enter the name of the business in the dialog box and click OK.

5 Select the new business node and enter the appropriate information on the Overview tab of the Business Details page on the right side. Some of these details are filled in by the UDDI site, but the Explorer allows you to edit these fields.

6 Choose the Contact tab and click Add. Enter the appropriate contact information.

7 Continue to add information to this business by right-clicking the business node and selecting New again to create the service node. Then right-click the service node and choose New to create the binding node, and so on.



8 Choose the UDDI operator node, the Business List node, or the business node and choose File | Save Changes. This saves all the changes and posts them to the UDDI site. You can also right-click a node and choose Save Changes or use the toolbar button. If the UDDI site requires a login, you'll be prompted for a user name and password.

Important The changes that are saved are dependent upon the node selected. You can choose a child of a node and only save changes to that node and nodes beneath it. You can also select the UDDI operator node or the Business List node and save all the changes.

Publishing tModels

To add and publish a new tModel,

1 Complete steps 1 and 2 in "Publishing web services."

2 Right-click the tModel List node in the tree and choose New tModel. A new tModel node is added to the tree.

3 Select the New tModel node and enter the appropriate information on the tModel Details page. If the UDDI site requires a login, you'll be prompted for a user name and password.

4 Select the tModel node and choose File | Save Changes to save the new tModel and post it to the UDDI site.

Publishing web services from an Axis server

This is a feature of the Axis toolkit. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

Publishing web services to a UDDI site from an Axis server in JBuilder involves the following steps:

1 Registering at the UDDI site through a web browser.

2 Creating and deploying a web service hosted on an Axis server.

3 Displaying an Axis-hosted web service.

4 Publishing from the Axis server.

Creating and deploying web services hosted on an Axis server

Next, create a web service that's hosted on a local Axis server.

1 Create a local Axis web service as described in "Tutorial: Creating a simple web service" on page 11-1.

2 Deploy the service and run the web server.

Displaying Axis-hosted web services

Now that the web service is deployed to the Axis server and the server is running, you can display the service in the Web Services Explorer.

1 Choose Tools | Web Services Explorer to open the Explorer.

2 Expand the Axis node and choose the Default Axis node.

3 Modify the URL on the Details page with the WebApp context, in this example `foo`, and the server location of the web service. For example, `http://localhost:8080/foo/servlet/AxisServlet`.

4 Choose the Display Services button on the details page to display the service as a child node of the Default Axis node.

Publishing from the Axis server

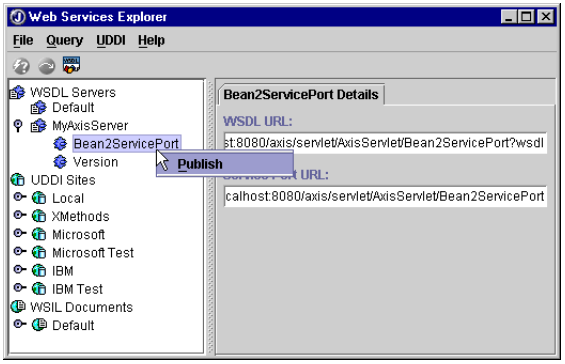
Once you have the Axis server running, you can publish the service to the UDDI site. First, you need to set the default for publishing. Then you need to run the Axis service locally on the web server and publish it.

1 Choose Tools | Web Services Explorer to open the Explorer.

2 Expand the UDDI Sites node and expand the UDDI operator site where you want to publish. Choose the Business List node and search for the business where you want to publish the service. If you haven't created a business yet, create one as described in "Publishing businesses and services" on page 10-22.

3 Right-click the business node and choose Set As Publishing Default.

4 Right-click the service displayed as a child of the WSDL Servers Default node and choose Publish or choose File | Publish.



5 Click OK in the Publish Axis Service To UDDI dialog box to close it.

6 Log into the UDDI registry if it requires it. A dialog box appears indicating that the service is published and the new nodes are created for that service below the business node set as the publishing default.

Monitoring UDDI messages

The Web Services Explorer has a UDDI Message Monitor that allows you to view the SOAP messages to and from the selected UDDI operator site.

To view these messages,

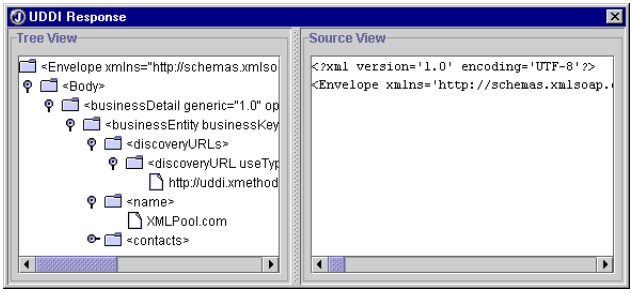
1 Expand the UDDI Sites node in the Explorer's tree and choose a UDDI node.

2 Choose UDDI | Cache UDDI Messages. This enables the UDDI Message Monitor, which saves the requests to and responses from the UDDI operator site.

3 Execute a search at the site.

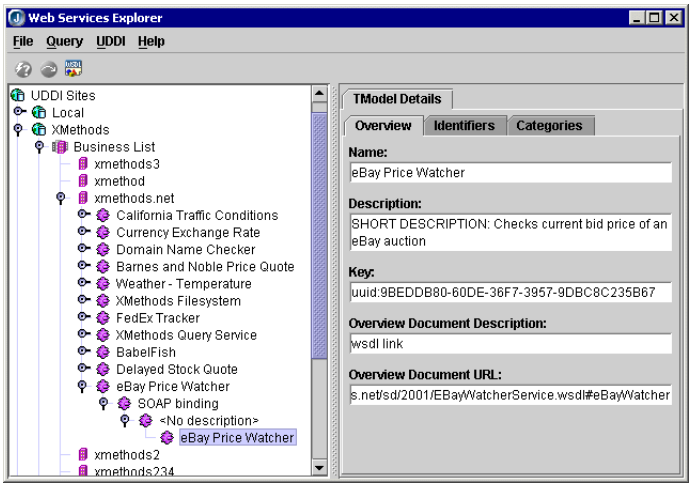
4 Choose the UDDI operator node and choose UDDI | View Messages to open the UDDI Message Monitor.

5 Select a response or request from the list and click View to view the message. Here you can see the SOAP envelope and the XML messages it contains in the Tree View on the left. The Source View on the right displays the source code for the message sent or received.



Generating Java classes from WSDL documents

The Web Services Explorer provides access to the Import A Web Service wizard for generating Java classes based on the WSDL document. The Import A Web Service wizard is available as a menu command, File | Import A Web Service, and as a button on the toolbar. The menu command and the toolbar button are only active when a WSDL document is available. For example, if the tModel Details node is selected in the tree and a WSDL document is specified in the Overview Document URL field, the Import A Web Service button and the menu command are enabled.



Use the menu command or the toolbar button to open the Import A Web Service wizard. If you don't have a project open, the Project wizard opens first. After you've created a project, the Import A Web Service wizard opens. Choose a web services toolkit and click OK. Notice that the WSDL document name is automatically entered in the WSDL URL field of the Import A Web Service wizard. When a web site requires a user name and password, the first step of the wizard has Username and Password fields that you must fill out. For example, some UDDI sites require you to register as a user. Continue completing the steps of the wizard to generate the appropriate classes. Choose the Help button on any step of the wizard for more information.

Chapter

11

Axis tutorials

This is a feature of JBuilder Enterprise. Support for the Apache Axis toolkit is not provided with the JBuilder WebLogic Edition.

The following tutorials are available for the Axis web services toolkit:

- "Tutorial: Creating a simple web service" on page 11-1 - Explains how to use the Export As A Web Service wizard to publish a JavaBean as a web service, exposing selected methods to the web service consumer.
- "Tutorial: Generating a web service from a WSDL document" on page 11-8 - Explains how to use the Import A Web Service wizard to generate the Java classes for a web service that provides a translation service and how to implement the service.
- "Tutorial: Creating a web service from an EJB application with Borland Enterprise Server" on page 11-21 - Explains how to create a web service from an Enterprise JavaBean application using Borland Enterprise Server.
- "Tutorial: Importing a web service as an EJB application" on page 11-29 - Describes how to import a WSDL as an EJB application using the Axis toolkit and Borland Enterprise Server.

Tutorial: Creating a simple web service

This tutorial uses features in JBuilder Enterprise. Support for the Apache Axis toolkit and Tomcat is not provided with the JBuilder WebLogic Edition.

This tutorial teaches you how to use the Export As A Web Service wizard to export a JavaBean as a web service, create a WSDL to describe the service, and create a web services server using the Apache Axis toolkit to host the service.

In this tutorial, you'll complete the following tasks:

- Create a sample JavaBean.
- Export the sample bean as a web service and configure the project for web services.
- Deploy, run, and test the service.

This tutorial assumes you are familiar with Java, and the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder*.

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Creating a sample JavaBean

In this step, you'll create a new project using the Project wizard. Then you'll create a JavaBean that you'll export as a web service.

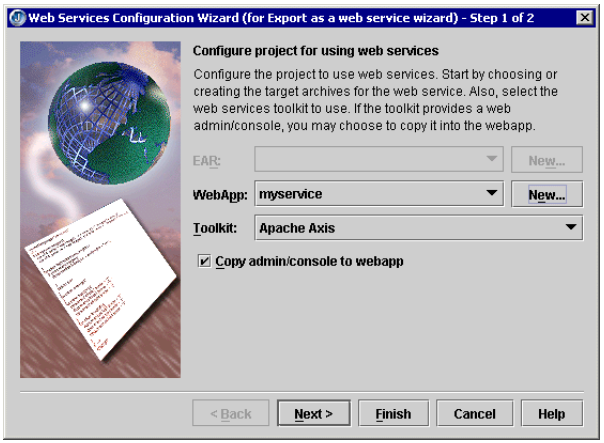
1. Select File | New Project to display the Project wizard.
2. Enter `exportbean` in the Name field.
3. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.
4. Check to make sure that the server for the project is set to Tomcat 4.0 on the Server page of Project Properties (Project | Project Properties).
5. Select File | New to display the object gallery and choose the General tab.
6. Select JavaBean and click OK to launch the JavaBean wizard.
7. Choose `java.lang.Object` as the base class.
8. Check the Public and Generate Sample Property options and uncheck the Generate Main Method option.
9. Click OK to close the wizard. A `Bean1.java` file is generated in the `exportbean` package.

Step 2: Exporting the sample bean as a web service and configuring the project for web services

In this step, you'll configure the project for web services and use the Export As A Web Service wizard to export the bean as a web service. The wizard will expose all the bean methods in the service, create a WSDL to describe the service, create a web services deployment file, and generate the Java classes for the service.

1. Right-click `Bean1.java` in the project pane and choose Export As A Web Service from the context menu. Because your project isn't configured for web services yet, the Web Services Configuration wizard displays so you can create a WebApp to host the web service. After the project is configured, the Export As A Web Service wizard displays.
2. Configure the project for web services using the Web Services Configuration wizard as follows:
   a. Choose the New button next to the WebApp field to create a WebApp with the Web Application wizard. A WebApp is required to host the service.
   b. Enter `myservice` as the name and directory for the WebApp. The web context, which is the WebApp name and directory, is used in the WSDL and `Bean1ServiceLocator.java` generated by the toolkit. In this example, the default SOAP address in the WSDL document generated by the wizard is `<wsdlsoap:address location="http://localhost:8080/myservices/services/Bean1"/>`. The address for the service port in `Bean1ServiceLocator.java` is `http://localhost:8080/myservice/services/Bean1`.
   c. Click OK to return to the Web Services Configuration wizard.
   d. Choose Apache Axis from the Toolkit drop-down list.

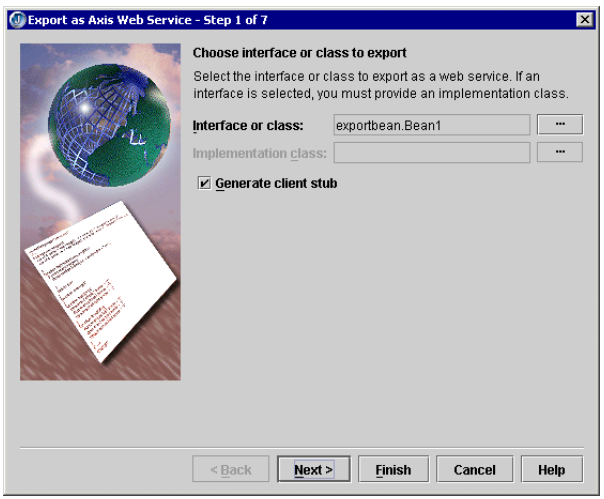
The Web Services Configuration wizard should look like this:



e Click Next in the Web Services Configuration wizard to see that a runtime configuration named Web Services Server will be created by the wizard. You'll use this runtime configuration later to run the project.

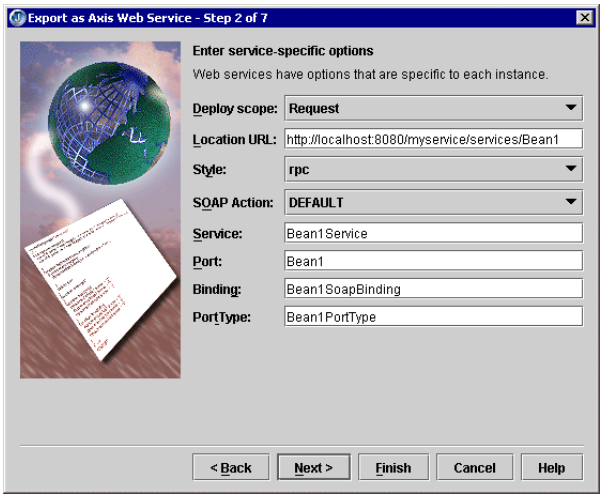
f Click Finish to close the Web Services Configuration wizard and continue on to the Export As A Web Service wizard.

Now that you've configured the project for web services, you'll use the Export As A Web Service wizard to export the class as a web service. The Export As A Web Service wizard looks like this:



3 Accept the class on Step 1 of the Export As A Web Service wizard and click Next.

4 Enter `Bean1PortType` in the PortType field.



5 Continue through the remaining steps of the wizard and accept all the defaults. Notice that on the Methods page, the Selection Mode is to allow all methods to be exposed in the service. Choose Help on any page of the wizard to read about the options.

6 Click Finish to close the wizard and generate the WSDL and Java classes for the service.

A WSDL document called `Bean1.wsdl` is generated. This file contains the WSDL information which defines how to connect to the web service. The Java files which make up the web service are also generated and placed in a package called `exportbean.generated`. These include the following files:

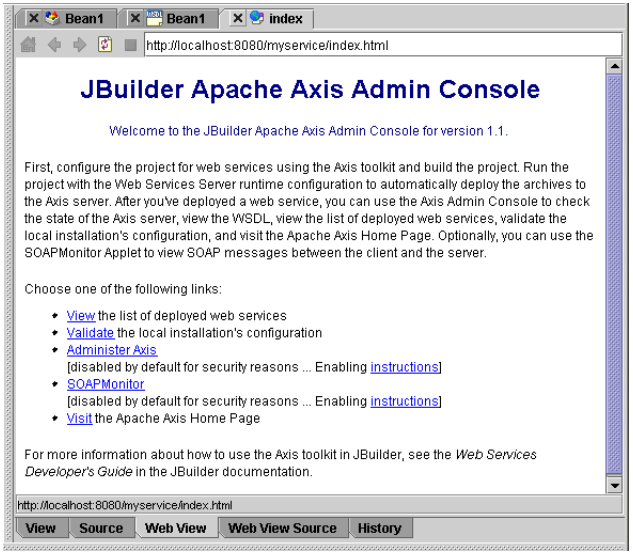
- `Bean1PortType.java` is the interface for the service.
- `Bean1Service.java` is an abstract interface which defines a factory class to get a stub instance.
- `Bean1ServiceLocator.java` is the implementation of the abstract interface.
- `Bean1ServiceTestCase.java` is the JUnit test case for the service. For more information about JUnit test cases, see "Unit Testing" in *Building Applications with JBuilder*.
- `Bean1SoapBindingStub.java` is the client stub that serializes the Java call and parameters into SOAP.

Step 3: Deploying, running, and testing the web service

In this step, you'll deploy the service and run the web services server. Then you'll run the JUnit test case, which was generated by the Export As A Web Service wizard, to test the JavaBean running as a web service.

1 Choose Run | Run Project to run the runtime configuration created by the Web Services Configuration wizard, Web Services Server. This runtime configuration deploys the service and runs the web services server. For more information on runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*.

   The web services server is started and the service is deployed to the server. Once the service is deployed, the Axis Admin Console loads in the JBuilder content pane.



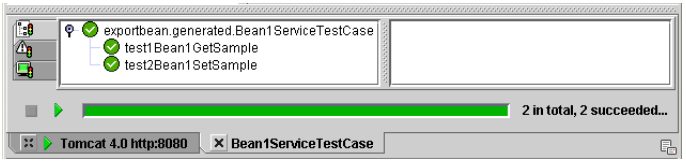
2 Confirm that the service is deployed. Click the View hyperlink in the Axis Admin Console to display the deployed services. You'll see that

Bean1 and its two methods, getSample() and setSample(), have been deployed.

3 Click the Bean1 (wsdl) hyperlink in the Axis Admin Console or double-click Bean1.wsdl in the project pane to view the WSDL document generated by the wizard for the service. Notice that both the getSample() and setSample() methods are exposed as operations in the service description.

4 Expand the exportbean.generated package node in the project pane.

5 Right-click Bean1ServiceTestCase.java and select Run Test Using Defaults from the context menu. This runs the generated JUnit test case in the default test runner. Here is how JBTestRunner looks after running the test case:



The test case accesses the public methods of the JavaBean running as a web service on the web services server. For more information on unit testing, see "Unit Testing" in *Building Applications with JBuilder*.

Important If you make any changes to Bean1.java after you've exported it, you would need to recompile, export as a web service again, and redeploy to the web services server for the changes to take effect.

Congratulations! You've completed the tutorial. You've exported a JavaBean as a web service, configured the project for web services, and tested the service. For more information about WSDL, see Chapter 5, "Working with WSDL." For more information about the Web Services Configuration wizard, see Chapter 3, "Configuring projects for web services." For more information on unit testing, see "Unit Testing" in *Building Applications with JBuilder*.

Tutorial: Generating a web service from a WSDL document

This tutorial uses features in JBuilder Enterprise. Support for the Apache Axis toolkit and Tomcat is not provided with the JBuilder WebLogic Edition.

This tutorial illustrates how to build a web service application from an existing WSDL document, call an external web service, and implement that same web service locally. You'll use the Import A Web Service wizard and the Axis toolkit to generate Java classes from the WSDL document. Then you'll complete the remaining steps to access a web service which provides translation to and from foreign languages using AltaVista's BabelFish. For more information on the files generated by the Import A Web Service wizard, see "Importing a WSDL" on page 7-5.

In this tutorial, you'll complete the following tasks:

- Configure the project for web services.
- Import a WSDL and generate Java classes.
- Examine the deployment descriptors.
- Implement the service.
- Create a Public WebApp to host a JavaServer Page (JSP).
- Create a JSP that invokes the service.
- Implement the bean.
- Consume the publicly-hosted service at XMethods.
- Use the Axis TCP Monitor to monitor SOAP message between the client and the service.
- Consume the locally-hosted service.

This tutorial is also available as a sample in the JBuilder directory: `<jbuilder>/samples/webservices/axis/BasicWebService`.

This tutorial assumes you are familiar with Java, JavaServer Pages (JSP) and the JBuilder IDE. For more information on Java, see *Getting Started with Java.* For more information on JavaServer Pages, see "Developing JavaServer Pages" in the *Web Application Developer's Guide.* For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder*.

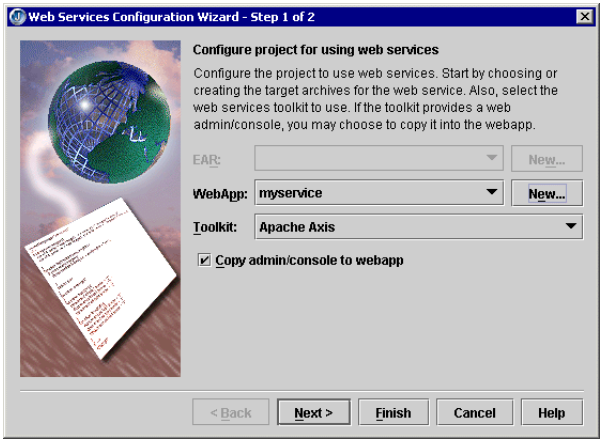
See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Configuring the project for web services

In this step you'll create a new project and then configure the project for web services using the Web Services Configuration wizard. The Web Services Configuration wizard creates a SOAP implementation for the project based upon the selected toolkit, in this case, Axis. It also creates a Web Services Server runtime configuration for deploying and running the service or an implementation of the service. Once a web service is deployed to the web services server, the web service can receive and send XML SOAP messages to and from client applications.

1. Select File | New Project to display the Project wizard.
2. Enter `wsdltutorial` in the Name field.
3. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard. A new project is created.
4. Check the Server page of Project Properties (Project | Project Properties) to make sure the server is set to Tomcat 4.0.
5. Choose File | New and choose the Web Services tab of the object gallery.
6. Double-click the Web Services Configuration icon. Since the new project doesn't contain a web application, you'll need to create one. A web services server requires a web application to host it.
7. Click the New button next to the WebApp field to open the Web Application wizard.
8. Type `myservice` for both the web application name and directory.
9. Click OK to close the Web Application wizard, create the myservice WebApp, and return to the Web Services Configuration wizard, which looks like this:



10 Accept myservice as the WebApp and Apache Axis as the toolkit and click Next to see that the wizard will create a runtime configuration named Web Services Server.

11 Click Finish to close the Web Services Configuration wizard and create the web services server.

Step 2: Importing the WSDL document

In this step, you'll add a WSDL document to the project, then generate Java classes from the WSDL document using the Import A Web Service wizard. The WSDL document describes the web service.

1 Click the Add Files/Packages/Classes button in the project pane toolbar.

2 Browse to `BabelFishService.wsdl` in `<jbuilder>/samples/webservices/axis/BasicWebService`. Select it and click OK to add the file to your project. This WSDL document will be used to generate Java classes which implement the web service.

3 Right-click `BabelFishService.wsdl` in the project pane and choose Import A Web Service from the context menu to open the Import A Web Service wizard.

4 Accept the WSDL URL on Step 1 and click Next. You don't need to enter a user name or password for this WSDL.

5 Accept the defaults on all the other pages and click Finish. A new package node, `net.xmethods.www.sd.BabelFishService_wsdl`, displays in the project pane. The package name is based on the WSDL target namespace by default.

6 Expand the `net.xmethods.www.sd.BabelFishService_wsdl` package node in the project pane to see the generated files.

- `BabelFishBindingImpl.java` is the service implementation. You write your code in this class. This class implements `BabelFishPortType.java`.
- `BabelFishBindingStub.java` is the client stub that serializes the Java call and parameters into SOAP.
- `BabelFishPortType.java` is an abstract interface for the service.
- `BabelFishService.java` is an abstract interface which defines a factory class to get a stub instance.
- `BabelFishServiceLocator.java` is the implementation of the abstract service interface.
- `BabelFishServiceTestCase.java` is the JUnit test case for the service. For more information about JUnit test cases, see "Unit Testing" in *Building Applications with JBuilder*.

For more information on the files created by the wizard, see "Importing a WSDL" on page 7-5.

7 Choose Project | Make Project to compile the generated classes.

Step 3: Looking at the deployment descriptors

The Import A Web Service wizard creates a deployment file: `deploy.wsdd`. The `deploy.wsdd` is a web services deployment descriptor XML file. Values in the `deploy.wsdd` file are derived from the information in the WSDL document. Another deployment file, `server-config.wsdd`, is created by the Axis toolkit when you make the project. This is a final deployment file that is comprised of multiple separate deploys. It lists the deployed services, as well as specifying handlers, transports, remote administration, and so on. In this step, you'll look at the deployment descriptors in the editor.

1 Expand the myservice WebApp node in the project pane and expand its child node, Web Service Components [Apache Axis toolkit].

2 Expand the Java-based Services node and double-click `[net.methods.www.sd.BabelFishService_wsdl]deploy.wsdd` to open the file in the editor.

Note the following elements:

- `<service name="BabelFishPort" provider="java:RPC" style="rpc" use="encoded">`

  The name of the published service.

- `<parameter name="className" value= "net.xmethods.www.sd.BabelFishService_wsdl.BabelFishBindingImpl"/>`

  The name of the implementation class for the `portType` interface.

- `<parameter name="allowedMethods" value="babelFish"/>`

  The published method.

3 Expand the Deployment Descriptors node and double-click `server-config.wsdd` to open the file in the editor. The `server-config.wsdd` file is created after you make the project.

Note the following elements in this final Axis deployment file:

- `<handler name="" type=""/>`

  A handler used for processing SOAP messages.

- `<parameter name="enableRemoteAdmin" value=""/>`

  A parameter that specifies if the Axis service can be accessed remotely. For more information on remote access, see "Accessing remote Axis servers" on page 10-17.

- `<transport name="">`

  Used for sending and listening for SOAP messages.

Step 4: Implementing the service

In this step, you'll replace the generated code in `BabelFishBindingImpl.java` with code that implements the web service. You write your code in this class, which implements `BabelFishPortType.java`. `BabelFishBindingImpl.java` is the service implementation, which implements the PortType interface, `BabelFishPortType.java`. This is where you enter your code to implement the web service.

1. Double-click `BabelFishBindingImpl.java` in the `net.xmethods.www.sd.BabelFishService_wsdl` package node in the project pane to open it in the editor.
2. Remove the generated body of the `BabelFishBindingImpl` class and replace it with the code indicated in bold that implements the service and returns the translation requested:

```
public class BabelFishBindingImpl implements
     net.xmethods.www.sd.BabelFishService_wsdl.BabelFishPortType {
  private String translatedOutput = "Sorry, I don't speak ";
  public java.lang.String babelFish(java.lang.String translationmode,
       java.lang.String sourcedata) throws java.rmi.RemoteException {
    if (translationmode.equals("en_fr"))
      translatedOutput = translatedOutput + " English";
    else
      translatedOutput = translatedOutput + " your language";
    return translatedOutput;
  }
}
```

   When you send a request for a translation to this local implementation of the service, you'll receive a standard reply. If you choose English to French, the standard reply is "Sorry, I don't speak English." If you choose any other language, the reply is "Sorry, I don't speak your language."

Later in this tutorial, you'll use the JSP to access the local service, `IndexBean.java`, using this class.

Step 5: Creating the Public web application

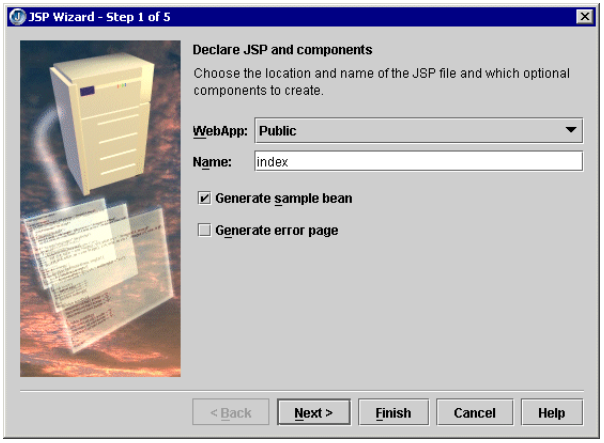
In this step, you'll create a WebApp to host the JavaServer Page (JSP) that you'll create in the following step.

1. Choose File | New.
2. Click the Web tab of the object gallery, select Web Application icon, and click OK. The Web Application wizard appears.
3. Enter `Public` for both the web application name and directory.
4. Click OK to close the wizard.

Step 6: Creating a JSP that invokes the web service

In this step you'll use the JSP wizard to create the skeleton of a JavaServer Page (JSP). Then you'll add JSP code that will generate a form for interacting with the web service. The form contains a text field for entering the text to be translated, a drop-down list of languages, a server drop-down list that allows you to choose to invoke the web service at XMethods or the local service, and a Submit and Reset button.

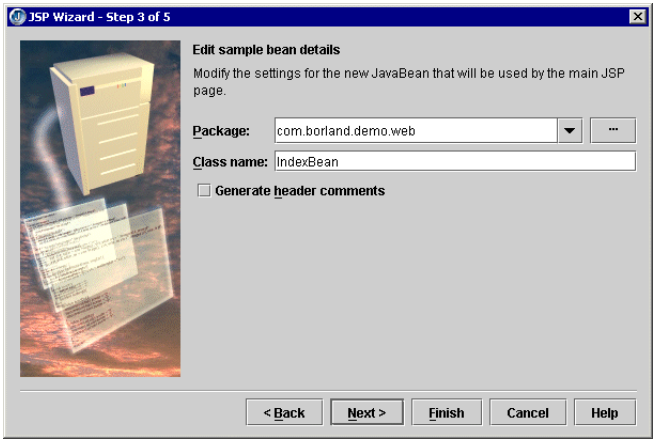
1. Choose File | New.
2. Click the Web tab of the object gallery and double-click the JavaServer Page icon to open the JSP wizard.
3. Make sure `Public` is selected as the WebApp.
4. Type `index` for the name of the JSP.
5. Make sure Generate Sample Bean is checked. The JSP wizard looks like this:



6. Click Next and uncheck Generate Submit Form on Step 2.

7 Click Next and enter `com.borland.demo.web` for the sample bean's package name.

8 Make sure the name of the bean is `IndexBean`. The JSP wizard looks like this:



9 Click Finish.

10 Make sure `index.jsp` is open in the editor.

11 Replace the contents of the `<body></body>` tags with the following JSP code to create the form for interacting with the web service:

```
<h1>A simple JSP using a Web Service</h1>
<form method="post">
<table>
  <tr>
    <td>Enter the text to translate</td>
    <td><input name="inputParam" value=
         "<jsp:getProperty name="indexBeanId"
         property="inputParam" />"></td>
  </tr>
  <tr>
    <td>Choose the language to translate text to</td>
    <td><select name="inputOption">
        <option value="en_fr">English to French
        <option value="en_de">English to German
        <option value="en_it">English to Italian
        <option value="en_pt">English to Portuguese
        <option value="en_es">English to Spanish
        <option value="fr_en">French to English
        <option value="de_en">German to English
        <option value="it_en">Italian to English
        <option value="pt_en">Portuguese to English
        <option value="ru_en">Russian to English
        <option value="es_en">Spanish to English
        </select>
    </td>
  </tr>
```

```
  <tr>
    <td>Select the server</td>
    <td><select name="serverOption">
        <option value="public">Publicly hosted service at XMethods
        <option value="local">Locally hosted service you built
        </select>
    </td>
  </tr>
</table>
<input type="submit" name="Submit" value="Submit">
     <input type="reset" value="Reset"></br>
</form>

<br>I translated your text</br>
<br><b>"<jsp:getProperty name="indexBeanId"
     property="inputParam"/>"</b></br>
<br>into your language as</br>
<br><b>"<jsp:getProperty name="indexBeanId"
     property="outputParam"/>"</b></br>
```

Step 7: Implementing the bean

In this step, you'll write code to implement the `IndexBean` class that was created by the JSP wizard in the previous step. `IndexBean.java` sets defaults for text fields in the form, invokes the service through the client stub (`BabelFishBindingStub.java`), and sends the user input information to the service.

1. Expand the `com.borland.demo.web` package node in the project pane.
2. Double-click `IndexBean.java` to open it in the editor.
3. Remove the body of the `IndexBean` class and add the code shown in **bold**:

```
package com.borland.demo.web;

import net.xmethods.www.sd.BabelFishService_wsdl.*;
import java.net.URL;

public class IndexBean {
  public static void main(String[] args){
      BabelFishPortType ws = null;
    try {

     ws = new BabelFishServiceLocator().getBabelFishPort
        (new URL("http://localhost:8080/myservice/services/BabelFishPort"));
     System.out.println(ws.babelFish("en_fr", "Hello"));
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

```
  private BabelFishPortType ws;
  private String outputParam;
  private String inputParam = "Hello";
  private String inputOption = "en_fr";
  private String serverOption = "public";


  public String getInputParam() {
    return inputParam;
  }
  public String getInputOption() {
    return inputOption;
  }
  public String getOutputParam() {
    try {
      if (serverOption.equals("public"))
         ws = new BabelFishServiceLocator().getBabelFishPort();
      else
         ws = new BabelFishServiceLocator().getBabelFishPort
            (new URL(
            "http://localhost:8080/myservice/services/BabelFishPort"));
      outputParam = ws.babelFish(inputOption, inputParam);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
    return outputParam;
  }
  public void setInputParam(String newInput) {
    if (newInput!=null) {
      inputParam = newInput;
    }
  }
  public void setInputOption(String newOption) {
    if (newOption!=null) {
      inputOption = newOption;
    }
  }

  public String getServerOption() {
    return serverOption;
  }

  public void setServerOption(String newOption) {
    if (newOption!=null) {
      serverOption = newOption;
    }
  }

}
```

Notice that the `getOutputParam()` method takes the server option input by the user consuming the service from the JSP form, and if it's equal to

"public" it makes a call to the `getBabelFishPort()` where the address is to the publicly hosted service at XMethods, `"http://services.xmethods.net:80/perl/soaplite.cgi"`. If the server option input by the user isn't "public", the local service is used instead at `"http://localhost:8080/myservice/services/BabelFishPort"`.

Step 8: Invoking the web service and monitoring SOAP messages

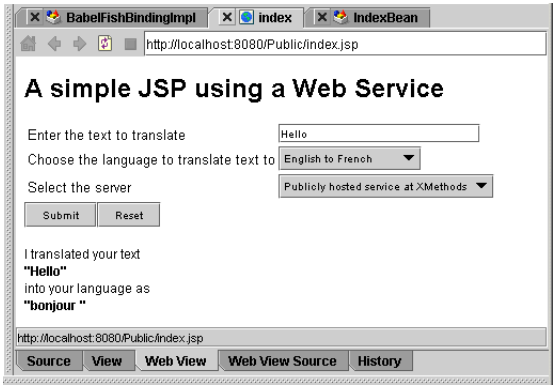
In this step, you'll invoke two different web services: the public web service at XMethods and the local web service you created. You'll also monitor the SOAP messages sent between the client and the server with the Axis TCP Monitor. First, you'll consume the publicly-hosted service at the XMethods website. Then, you'll consume the service you created locally and use the Axis TCP Monitor to monitor the SOAP messages as they travel between the client and the service.

Invoking the publicly-hosted web service

Next, you'll compile the project and invoke the web service at the XMethods web site.

1. Right-click the `index.jsp` file tab in the editor and choose Web Run Using "Web Services Server" from the context menu. This runtime configuration builds the project and then starts both web applications: `myservice`, which hosts the web service, and `Public`, which contains the JSP that's used to access the web service. Note that "Web Services Server" in the Web Run command of the context menu refers to a runtime configuration that was automatically created for you by the Web Services Configuration wizard. For more information on runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*.

   The JSP form looks like this:



2 Enter `good-bye` in the text field and select English to Spanish from the language drop-down list. Notice that the selected server is a Publicly Hosted Service At XMethods.

3 Click Submit to send the text to the service to be translated. The request is sent to XMethods and the translated response displays at the bottom of the form: `I translated your text "good-bye" into your language as "adios".`

Invoking the locally-hosted web service and monitoring SOAP messages

Next, you'll consume the local service that you created from the WSDL and use the Axis TCP Monitor to monitor the SOAP messages. The local service doesn't translate the language like the XMethods web service. If English to French is selected as the language, the response is: `Sorry, I don't speak English`. If any other language is selected, the response is: `Sorry, I don't speak your language`. This is the implementation code you added to `BabelFishBindingImpl.java` in "Step 4: Implementing the service" on page 11-12.

First, you'll set up the TCP Monitor, modify `IndexBean.java` to send requests and responses to the TCP Monitor on 8082, recompile the project, and then consume the locally-hosted service. The TCP Monitor sits between the SOAP client and server. The client sends its request to the TCP Monitor, which then forwards it on to the server. Responses from the server are in turn sent to the TCP Monitor and then to the client. For more information on the TCP Monitor, see Chapter 4, "Monitoring SOAP messages." When you use the TCP Monitor, you need to modify the client code's address and port number to send the messages to and receive messages from the TCP Monitor on the listen port.

1 Choose Tools | TCP Monitor to open the TCP Monitor. Note that the default port for the TCP Monitor to listen on is 8082. Accept the defaults and return to JBuilder, leaving the TCP Monitor open.

2 Return to `IndexBean.java` in the editor and change the port number from 8080 to 8082 in two places. You can use Search | Replace to change the port number. For example, change `http://localhost:8080/myservice/services/BabelFishPort` to the port the TCP Monitor is listening on: `http://localhost:8082/myservice/services/BabelFishPort`. `IndexBean.java`, which is the local service, needs to receive requests and send responses to the same port the TCP Monitor is listening on, port 8082.

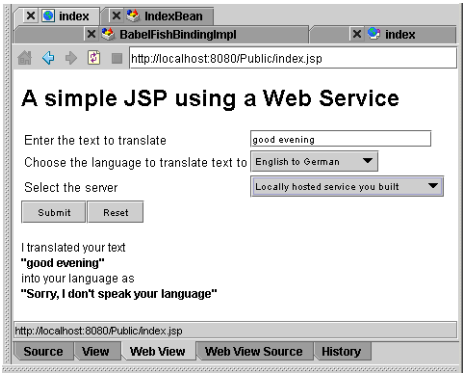
3 Choose the Reset Program button on the message pane toolbar to stop the server.

4 Choose Project | Rebuild Project to recompile and include the port change you just made to `IndexBean.java`.

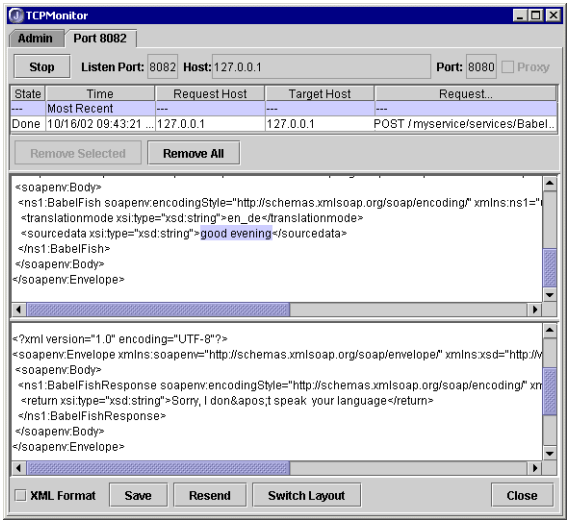
5 Run `index.jsp` again to reload the JSP form.

Now, you'll consume the local service that you created from the WSDL and look at the SOAP messages sent between the client and the service.

1. Return to the `index.jsp` form where you enter the text you want to translate.
2. Enter `good evening` in the text field.
3. Choose English To German from the language drop-down list.
4. Choose Locally Hosted Service You Built from the server drop-down list.
5. Click the Submit button to send the text to the locally-hosted service. The translation displays beneath the Submit button: `"Sorry, I don't speak your language."`



6. Return to the TCP Monitor and examine the messages sent between the client and the service by the TCP Monitor. You can use the TCP Monitor to edit and resend the SOAP message to the server.

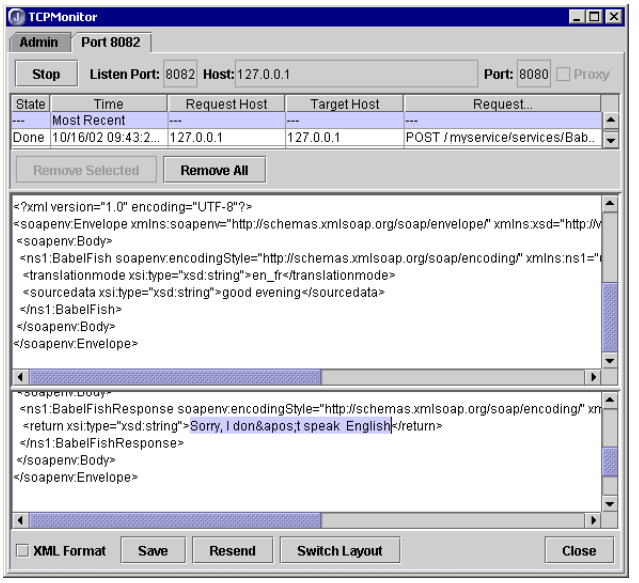


7 Edit the request in the TCP Monitor Request panel and resend it as follows:

   a Change the language in the <translationmode> element from `en_de` to **`en_fr`** to change the language to French:
   `<translationmode xsi:type="xsd:string">en_fr</translationmode>`.

   b Change the `good evening` in the <sourcedata> element to `good-bye`:
   `<sourcedata xsi:type="xsd:string">good-bye</sourcedata>`

   c Click Resend to send the revised message. The log at the top of the TCP Monitor shows that the message was resent and a new response is also logged and displayed in the Response panel. The response from the service is now: `Sorry, I don't speak English`. The TCP Monitor looks similar to this:



8 Stop and close the TCP Monitor.

Congratulations! You've completed the tutorial and created a web service that provides translation of strings using AltaVista's BabelFish. You've also consumed the service at the XMethod's site, consumed the local service you created, and monitored the SOAP messages between the client and the server with the TCP Monitor. For more information about WSDL, see Chapter 5, "Working with WSDL." For more information on configuring JBuilder projects for web services, see Chapter 3, "Configuring projects for web services."

Tutorial: Creating a web service from an EJB application with Borland Enterprise Server

This tutorial uses features in JBuilder Enterprise. Support for the Apache Axis toolkit and this application server is not provided with the JBuilder WebLogic Edition.

This tutorial, which assumes knowledge of Enterprise JavaBeans (EJBs), illustrates how to create a web service from an EJB application using the Borland Enterprise Server 5.1.x, 5.2, or 5.2 as the application server and Axis as the web services toolkit. The tutorial uses a sample in the JBuilder `samples` directory, `<jbuilder>/samples/tutorials/webservices/axis/Employee.jpx`. The sample contains an EJB application which accesses a database of employee records.

The tutorial includes the following steps:

- Setting up the sample project.
- Creating a web services server to host the service.
- Deploying the web service to the web services server and exporting an EJB application as a web service.
- Generating client and server code from the WSDL.
- Testing the service.
- Writing the client and consuming the service.

For more information on EJBs, see *Developing Enterprise JavaBeans*. For more information on web services and EJBs, see Chapter 6, "Developing EJBs as web services."

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Setting up the sample project

If you don't have your Borland Enterprise Server 5.1.x, 5.2, or 5.2 configured yet, see "Using JBuilder with Borland servers" in *Developing J2EE Applications* for instructions on how to configure it.

1. Open the sample, `Employee.jpx`, located in `<jbuilder>/samples/Tutorials/webservices/axis`.
2. Complete the instructions in the sample project file, `EmployeeProject.html`, for connecting to the database before continuing to the next step.

Step 2: Creating a web services server and deploying to the application server

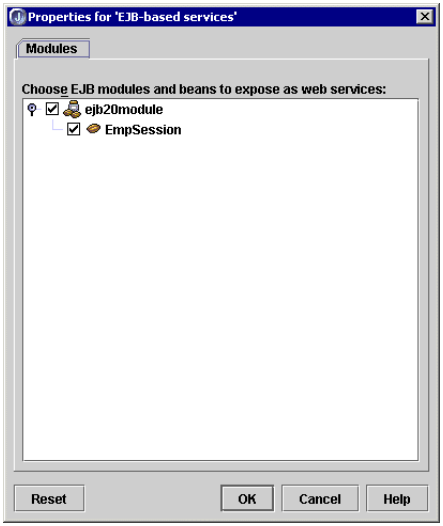
In this step, you'll use the Web Services Configuration wizard to create a web services configuration for a WebApp that hosts a web services server. Then you'll deploy the web service to an EAR file. When you deploy to the server, JBuilder automatically exposes any stateless session beans containing methods in the remote interface as web services.

1. Choose File | New to open the object gallery.
2. Click the Web Services tab and double-click the Web Services Configuration icon. The Web Services Configuration wizard configures a toolkit for the web service and generates a web services server to host the service. Before you can configure the project, you must create an EAR for the EJB deployment and a WebApp to host the web services server.
3. Create a new EAR archive as follows. You'll deploy everything to an EAR file, including the Axis web services server.
   a. Click the New button next to the EAR field to open the EAR wizard.
   b. Accept the default name for the EAR, `Employee`, and click Finish.
4. Create a new WebApp as follows:
   a. Click the New button next to the WebApp field to open the Web Application wizard.
   b. Enter `myservice` in the Name and the Directory fields.
   c. Click OK to create the WebApp and return to the Web Services Configuration wizard.
5. Choose Apache Axis as the toolkit and click Next to see the Web Services Server runtime configuration that the wizard will create for building the project and running the server.
6. Click Finish to close the wizard.
7. Expand the new myservice WebApp node to see the nodes and the deployment file generated by the Web Services Configuration wizard. When you run the Web Services Server runtime configuration created by the wizard, all of the EJB modules and any stateless session beans in the project are automatically deployed as web services. You can modify which modules are deployed on the Properties page. In this example,

you want to expose everything in the project, so you won't make any changes. To see what JBuilder automatically deploys, do the following:

a Expand the Web Service Components node.

b Right-click the EJB-based Services node and choose Properties. Here you'll see that the EJB module and bean are exposed as services. In this example, you want to expose everything in the project.



c Accept the defaults and click OK to close the dialog box.

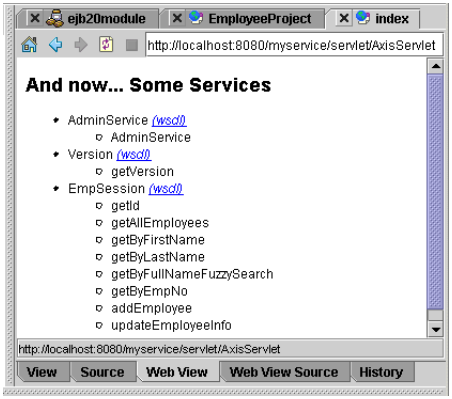
8 Save the project.

9 Choose Tools | Borland Enterprise Server Management Agent. This must be running to run the Borland Enterprise Server.

10 Start the server by clicking the drop-down arrow next to the Run button on the main toolbar and choosing Web Services Server. This runtime configuration builds the project, automatically deploys the EJBs, and starts the server. Once the server is running, the Axis Admin Console displays on the Web View tab of the content pane.

11 Click the View hyperlink on the Axis Admin Console to confirm that the service has deployed correctly. If you get a "500 No context" message, wait until the server is running and choose the Refresh button

on the toolbar to display the Axis page. Notice that all the methods in EmpSession are deployed.

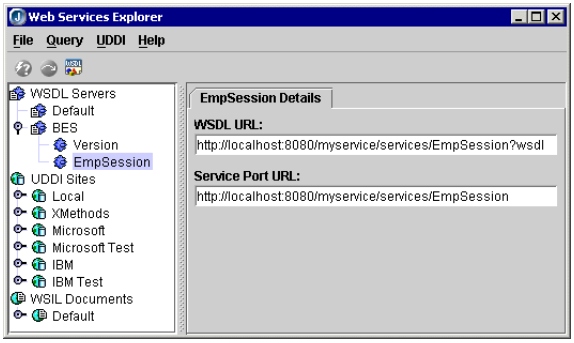


The WSDL generated by the toolkit is also available here. Click the EmpSession (WSDL) link to view the WSDL.

Step 3: Generating client and server code from the WSDL

Now that you've deployed the EJB application as a web service, you'll use the Web Services Explorer to browse to the Axis server hosting the service, display the EJB service, import the WSDL that describes it, and generate server, client, and test classes for the service.

1. Choose Tools | Web Services Explorer to open the Explorer.
2. Right-click the WSDL Servers node in the tree and choose New WSDL Server to create a new node.
3. Enter `BES` in the name field and click OK.
4. Change the URL to `http://localhost:8080/myservice/services` if you're running on port 8080 or modify the port number.
5. Click the Display Services button to display the new EJB service in the tree, EmpSession.



6 Select the EmpSession EJB service in the tree and click the Import A Web Service button to open the Import A Web Service wizard which imports the WSDL and generates the Java classes from the WSDL.

7 Choose these options in the wizard:

   a Accept the WSDL URL. In this example, you don't need to fill out the User Name and Password fields.

   b Click Next to continue to the next step.

   c Check Generate Skeleton Classes. This option creates a client to consume the service.

   d Click Next and change the package name to `com.borland.samples.emp.webservices`.

   e Check the Use This Package Globally For All Types option. Checking this option puts all of the generated classes in the `com.borland.samples.emp.webservices` package.

   f Accept all the other wizard defaults. Notice that the Generate JUnit Test Case option is checked. Later in the tutorial, you'll use this test case to test the service.

   g Click Finish.

Notice that a new `webservices`package node displays as a child node of `com.borland.samples.emp` in the project pane. Expand the `com.borland.samples.emp.webservices` in the project pane to see the files generated by the wizard. For information on the files generated by the Import A Web Service wizard, see "Importing a WSDL" on page 7-5.

Step 4: Testing the service

In this step, you'll add code to the JUnit test generated by the wizard and test the service. Each test will test each method in the service. First, you'll compile the project, then add new employee information to the `test7EmpSessionAddEmployee()` method. Then you'll add code to the other methods.

1 Choose Project | Make Project to make the project and compile the new classes.

2 Double-click `EmpSessionServiceTestCase.java` in the `com.borland.samples.emp.webservices` package node to open it in the editor.

3. Change the cast short in the `test6EmpSessionGetByEmpNo()` method from 0 to 2. There is an employee number 2 but not an employee number 0 in the database:

```
// Test operation
com.borland.samples.emp.webservices.EmployeeInfo value = null;
value = binding.getByEmpNo((short)2);
```

4. Add the new employee data indicated in bold to the `test7EmpSessionAddEmployee()` method. This code adds Jim Shorts as a new employee to the database.

```
// Test operation
boolean value = false;
com.borland.samples.emp.data.EmployeeInfo empInfo =
     new com.borland.samples.emp.data.EmployeeInfo();
empInfo.setFirstName("Jim");
empInfo.setLastName("Shorts");
empInfo.setFullName("Shorts, Jim");
value = binding.addEmployee(empInfo);
```

5. Add the following code indicated in bold to the `test8EmpSessionUpdateEmployeeInfo()` method. This code updates employee number 145 with the modified employee information.

```
// Test operation
boolean value = false;
com.borland.samples.emp.data.EmployeeInfo empInfo =
     new com.borland.samples.emp.data.EmployeeInfo();
empInfo.setEmpNo(new Short((short)145));
empInfo.setFirstName("Sally");
empInfo.setLastName("Shorts");
empInfo.setFullName("Shorts, Sally");
value = binding.updateEmployeeInfo(empInfo);
```

6. Add the following code in bold to the `test3EmpSessionGetByFirstName()` method. This code retrieves all employees with Robert as the first name. The string is case-sensitive and must be an exact match to the database.

```
// Test operation
com.borland.samples.emp.webservices.EmployeeInfo[] value = null;
value = binding.getByFirstName(new java.lang.String("Robert"));
```

7. Add the following code in bold to the `test4EmpSessionGetByLastName()` method. This code retrieves all employees with the last name of Nelson. The string is case-sensitive and must be an exact match to the database.

```
// Test operation
com.borland.samples.emp.webservices.EmployeeInfo[] value = null;
value = binding.getByLastName(new java.lang.String("Nelson"));
```

8 Add the following code in bold to the `test5EmpSessionGetByFullNameFuzzySearch()` method. This code retrieves all employees whose full name starts with N. It must be a regular expression search.

```
// Test operation
com.borland.samples.emp.webservices.EmployeeInfo[] value = null;
value = binding.getByFullNameFuzzySearch(
     new java.lang.String("Nelson, Robert%"));
```

9 Right-click `EmpSessionServiceTestCase.java` in the project pane and choose Run Test Using Defaults to run the service test. The project builds and then the tests run. Notice that the tests are successful. For more information about JUnit test cases, see "Unit Testing" in *Building Applications with JBuilder*.

10 Close the EmpSessionServiceTestCase tab in the message pane.

Step 5: Writing the client and consuming the service

Next, you'll write a simple client that will get all the employee records in the database and output them to the message pane. It'll only have one method, the `empSessionGetAllEmployees()` method.

1 Choose File | New Class to open the Class wizard or right-click the `.jpx` project node in the project pane and choose New | Class.

2 Enter `com.borland.samples.emp.client` in the Package field.

3 Enter `Client` in the Class Name field and click OK to close the wizard and create the class file.

4 Remove the contents of `Client.java` and replace it with this code:

```
package com.borland.samples.emp.client;

public class Client {
  public static void empSessionGetAllEmployees() {
      com.borland.samples.emp.webservices.EmpSession binding;
      try {
          binding = new com.borland.samples.emp.webservices.
               EmpSessionServiceLocator().getEmpSession();
      }
      catch (javax.xml.rpc.ServiceException jre) {
          jre.printStackTrace();
          return;
      }
```

```
            try {
                com.borland.samples.emp.data.EmployeeInfo[] value = null;
                value = binding.getAllEmployees();
                if (value != null) {
                  for (int i=0; i < value.length; i++) {
                    System.out.println(value[i]);
                  }
                }
            }
            catch (java.rmi.RemoteException re) {
              re.printStackTrace();
              return;
            }
        }
        public static void main(String[] args) {
           empSessionGetAllEmployees();
        }
    }
```

The client must look up the binding for the service, which is located in the Locator class. Notice that the first try statement specifies the binding for the service and calls the `Locator` class. The Locator class is the client-side server implementation of the service interface.

```
try {
          binding = new com.borland.samples.emp.webservices.
               EmpSessionServiceLocator().getEmpSession();
      }
```

5 Right-click `Client.java` in the project pane and choose Run Using Client. The employee records are printed to the message pane.

You could also add other methods to the client to get more employee information.

Once you've deployed the EJB application and created and tested the web service, you can publish it to a UDDI registry from an Axis server. For more information, see "Publishing web services from an Axis server" on page 10-24

Congratulations, you've completed the tutorial. You've configured the project for web services, automatically exported an EJB application as a web service, deployed the service to a web services server, hosted it on an Axis server, and tested it.

Tutorial: Importing a web service as an EJB application

This tutorial uses features in JBuilder Enterprise. Support for the Apache Axis toolkit and this application server is not provided with the JBuilder WebLogic Edition.

This tutorial, which assumes knowledge of Enterprise JavaBeans (EJBs), illustrates how to create an EJB application from an existing web service using the Borland Enterprise Server 5.1.x, 5.2, or 5.2 as the application server. The tutorial uses a sample Web Services Description Language (WSDL) document located in the JBuilder `samples` directory, `<jbuilder>/samples/tutorials/webservices/wsdl`.

The tutorial includes the following steps:

- Adding a WSDL document to the project.
- Creating an EJB application from the WSDL.
- Testing the EJB application.

For more information on EJBs, see *Developing Enterprise JavaBeans*. For more information on web services and EJBs, see Chapter 6, "Developing EJBs as web services."

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Adding a WSDL document to the project

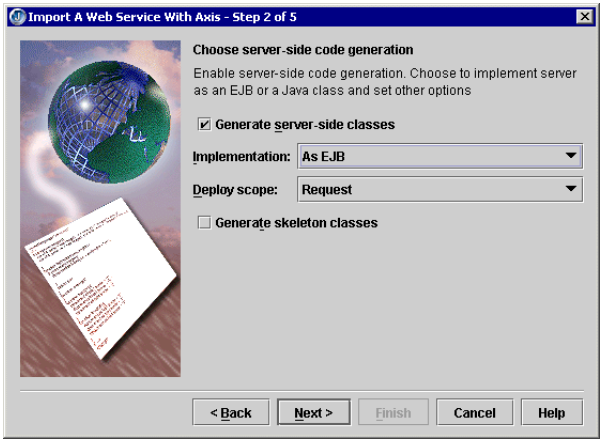
In this step, you'll create a new project, specify Borland Enterprise Server 5.1.x or 5.2 as the server, and add a WSDL to your project. If you don't have your Borland Enterprise Server configured yet, see "Using JBuilder with Borland servers" in *Developing J2EE Applications* for instructions on how to configure it.

1. Choose File | New Project to open the Project wizard.
2. Enter `interop` for the project name and click OK.
3. Configure the project to use Borland Enterprise Server. The server must be configured (Tools | Configure Servers) to appear in the servers list.
   a. Choose Project | Project Properties and click the Server page.
   b. Choose Borland Enterprise Server AppServer Edition 5.1.x, 5.2, or 5.2 from the single server drop-down list and click OK.
4. Choose Project | Add Files/Packages/Classes and browse to `InteropTest.wsdl` in `<jbuilder>/samples/webservices/axis/wsdl`.
5. Click OK to add the WSDL to your project.

Step 2: Creating an EJB application from the WSDL

Next, you'll import the WSDL and use the Import A Web Service wizard to generate an EJB application from the WSDL.

1. Right-click `InteropTest.wsdl` in the project pane and choose Import A Web Service to open the Import A Web Service wizard.
2. Choose Apache Axis as the toolkit and click OK.
3. Accept `InteropTest.wsdl` as the WSDL URL and click Next. A user name and password aren't required.
4. Choose As EJB from the Implementation drop-down list on Step 2 and click Next.



5. Check the Use This Package Globally For All Types options, accept all other defaults on Step 3, and click Next.
6. Click New on Step 4 to open the EJB Module wizard and create a new EJB module for the EJB application.
7. Accept the default name for the module, choose EJB 2.0 Compliant as the version, and click OK to close the EJB Module wizard.
8. Click Finish in the Import A Web Service wizard.

The Import A Web Service wizard creates an EJB application, as well as the server-side implementation of the web service from the WSDL. Expand the `org.soapinterop` package in the project pane to see the generated EJB files. Also notice that an EJB module was created.

Step 3: Testing the EJB application

In this step, you'll create an EJB test client and implement it to test the EJB application.

1. Choose File | New | Enterprise and double-click the EJB Test Client icon.
2. Choose Application as the test client type.
3. Click Next.
4. Change the package name to `org.soapinterop`, accept all other defaults, and click Finish. The test client, `InteropSessionPortTypeTestClient1.java`, is created in the `org.soapinterop` package.
5. Choose Project | Make Project to build the project.
6. Modify the `main()` method of `InteropSessionPortTypeTestClient1.java`, adding the code in bold:

```
public static void main(String[] args) {
  InteropTestPortTypeSessionTestClient1 client =
      new InteropTestPortTypeSessionTestClient1();
  client.create();
  client.echoString("Test");
}
```

7. Open `InteropTestPortTypeSessionBean.java` in the editor and implement the `echoString()` method to return a value.

```
public java.lang.String echoString(java.lang.String inputString)
    throws java.rmi.RemoteException  {
  return inputString;
 }
```

8. Run the server:
   a. Choose Tools | Borland Enterprise Server Management Agent.
   b. Right-click the EJB module and select Run Using Defaults.
9. Right-click `InteropTestPortTypeSessionTestClient1.java` in the project pane and choose Run Using InteropTestPortTypeSessionTestClient1 to run the test client. The return value displays in the message pane.

```
Return value from echoString(Test): Test.
```

Congratulations, you've completed the tutorial. You've created an EJB application from a WSDL, generated the server-side classes with the Import A Web Service wizard, and created and run an EJB test case.

Chapter

12

WebLogic tutorials

This is a feature of JBuilder Enterprise

The following tutorials are available for the WebLogic web services toolkit:

- "Tutorial: Creating a simple web service" on page 12-1 - Explains how to use the Export As A Web Service wizard to publish a JavaBean as a web service, exposing selected methods to the web service consumer.
- "Tutorial: Creating a web service from an EJB application with WebLogic Server" on page 12-9 - Explains how to create a web service from an EJB application using WebLogic Server.
- "Tutorial: Creating an asynchronous web service with WebLogic Server 8.1" on page 12-15 - Explains how to create an asynchronous web service using WebLogic Server 8.1.

Tutorial: Creating a simple web service

This tutorial uses features in JBuilder Enterprise

This tutorial teaches you how to use the Export As A Web Service wizard to export a JavaBean as a web service, create a WSDL to describe the service, and create a web services server using the WebLogic Server to host the service and the WebLogic toolkit to generate the service.

In this tutorial, you'll complete the following tasks:

- Create a sample JavaBean.
- Export the sample bean as a web service and configure the project for web services.
- Run the server and deploy the service.
- Test the service in the Web Services Console.
- Write a client to test the service.

For more information on WebLogic Server and web services, see the BEA documentation at `http://edocs.bea.com/wls/docs70/webserv/index.html` for version 7.0 or `http://edocs.bea.com/wls/docs81/webserv/index.html` for version 8.1.

This tutorial assumes you are familiar with Java and the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder*

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Creating a sample JavaBean

In this step, you'll create a new project using the Project wizard an specify WebLogic as the server. Then you'll create a JavaBean that you'll export as a web service.

1. Select File | New Project to display the Project wizard.
2. Enter `exportbean` in the Name field.

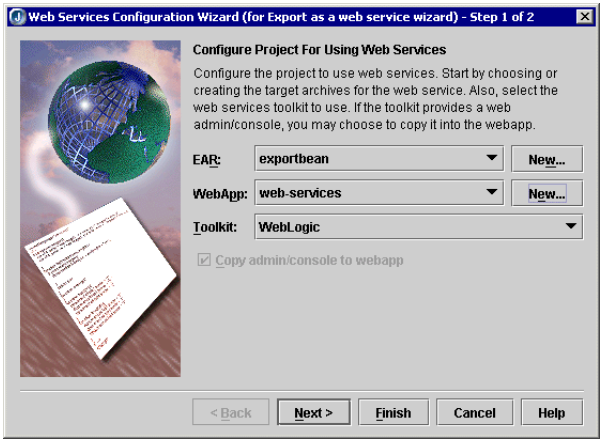
Important Be sure to create the project in a directory **without** any spaces in the name or WebLogic Server may generate errors.

3. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.
4. Choose Project | Project Properties, click the Server tab, and choose WebLogic Application Server 7.x or 8.1 as the single server for the project. If you don't see it in the list, you need to enable WebLogic 7.x or 8.1 in the Configure Servers dialog box (Tools | Configure Servers).
5. Select File | New to display the object gallery and choose the General tab.
6. Select JavaBean and click OK to launch the JavaBean wizard.
7. Choose `java.lang.Object` as the base class, check the Public and Generate Sample Property options, and uncheck the Generate Main Method option.
8. Click OK to close the wizard. A `Bean1.java` file is generated in the `exportbean` package.

Step 2: Exporting the sample bean as a web service

In this step, you'll configure the project for web services and use the Export As A Web Service wizard to export the bean as a web service. The wizard will expose all the bean methods in the service and create a web services deployment file.

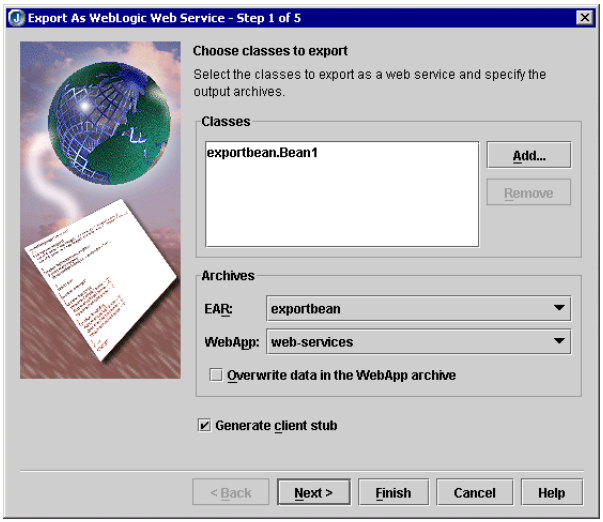
1. Right-click `Bean1.java` in the project pane and choose Export As A Web Service from the context menu. Because your project isn't configured for web services yet, the Web Services Configuration wizard displays so you can create a WebApp to host the web service and an EAR for the web service. After the project is configured, the Export As A Web Service wizard displays.
2. Configure the project for web services using the Web Services Configuration wizard as follows:
   a. Choose the New button next to the EAR field to create an EAR with the EAR wizard.
   b. Accept `exportbean` as the default name for the EAR and click Finish to return to the Web Services Configuration wizard.
   c. Choose WebLogic as the toolkit.
   d. Choose the New button next to the WebApp field to create a WebApp with the Web Application wizard. A WebApp is required to host the service.
   e. Accept `web-services` as the name and directory for the WebApp and click OK to return to the Web Services Configuration wizard. The Web Services Configuration wizard should look like this:



   f. Click Next in the Web Services Configuration wizard to see that the wizard creates a Web Services Server runtime configuration. You'll use this runtime configuration later to run the project.

g Click Finish to close the Web Services Configuration wizard and continue on with the Export As A Web Service wizard.

Now that you've configured the project for web services, you'll use the Export As A Web Service wizard to export the class as a web service. The Export As A Web Service wizard looks like this:



1 Accept the defaults on Step 1 of the wizard, such as class to export, EAR, and WebApp. Notice that the option, Generate Client Stub is checked. When this option is checked, the wizard generates a client JAR that contains the Java classes, stubs, and interfaces needed by a client application to invoke the service.

2 Click Next to go to Step 2.

3 Accept all other defaults and notice that the ServiceURI name is `exportbean`. The ServiceURI is used by the Web Services Console to locate the deployed service. You'll use the Web Services Console later to test the deployed service.

4 Continue on to Step 4 to see the default names for the package and the client JAR that's generated for the service: `exportbean.generated` and `Bean1_client.jar`.

5 Click Finish to close the wizard and generate the web service.

6 Expand the web-services WebApp node in the project pane and expand the Web Service Components node created by the Web Services Configuration wizard. Then expand the Java-based Services node to see the WLDU deployment file created by the wizard. This deployment file provides the web services deployment information for the WebLogic Server, such as the name of the EAR, WAR, and the class to be exported. When you deploy the service to the server, it uses the

deployment information to create the web service and the WSDL from the selected class.

7 Expand the GeneratedWebServicesClients node in the project pane and its child node, `Bean1_client.jar` to see its contents.

8 Expand the `exportbean` and `generated` package nodes in the project pane to see the classes and WSDL generated by the WebLogic toolkit.

Step 3: Running the server and deploying the service

In this step, you'll run the WebLogic Server using the Web Services Server runtime configuration and deploy the service to the Server.

1 Choose Run | Run Project to run the server. When you run the project, it uses the Web Services Server runtime configuration created by the Web Services Configuration wizard. This runtime configuration builds the project and runs the WebLogic Server as the web services server. For more information on runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*.

2 Right-click the exportbean.eargrp node after the server is running and choose Deploy Options | Deploy to deploy the service to the WebLogic Server.

Important If you make any changes to `Bean1.java` after you've exported it, you would need to recompile, export as a web service again, and redeploy for the changes to take effect.

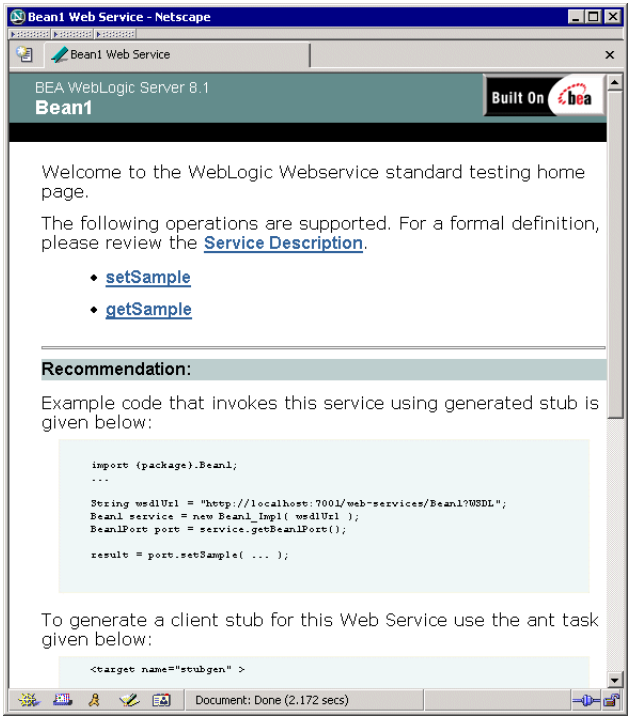
Step 4: Testing the deployed service

You can test your deployed service in the Web Services Console which displays the testing page for the deployed web service. Here you can test the methods exposed in the service, view SOAP messages between the client and the server, and view the WSDL.

1 Choose Tools | Web Services Console to open the Web Services Console. For this tool to be active, you must have a web browser specified for the WebLogic Admin Console (Tools | Configure Servers). The Web Services Console displays the testing home page for the deployed service.

Important To use the Web Services Console, a web browser must be configured for the WebLogic Server. For more information, see "Using JBuilder with BEA WebLogic servers" in *Developing J2EE Applications* for instructions on how to configure it.

The exposed methods in the class display on the WebLogic Web Services Home Page: `getSample()` and `setSample()` methods. Here you can also click the Services Description link to see the generated WSDL. The testing home page also has example code for invoking the service that you can copy, modify, and use to test the service you've just deployed.



2. Click the setSample link and notice that the setSample value is set to "sample string."
3. Choose the Invoke button to see the SOAP request and response to and from the server.
4. Return to the first page, click the getSample link, and click the Invoke button to get the value of sample. The server response returns a value of "sample string."
5. Return to the first page and click the Services Description link to see the generated WSDL for the service. The `getSample()` and `setSample()` methods are exposed as operations in the service description.

Step 5: Writing a client to test the service

Next you'll write a client to test the exported web service.

1. Return to the first page of the testing home page in the Web Services Console and copy the example code that invokes the service:

```
import {package}.Bean1;
    ...

    String wsdlUrl =
         "http://localhost:7001/web-services/exportbean?WSDL";
    Bean1 service = new Bean1_Impl( wsdlUrl );
    Bean1Port port = service.getBean1Port();

    result = port.getSample( ... );
```

2. Return to JBuilder and choose File | New Class or right-click the project file in the project pane and choose New | Class to create a new class.
3. Enter `test` in the Package field and `Test` in the Class Name field.
4. Check Generate Main Method and uncheck Generate Default Constructor.
5. Click OK to create the new test class.
6. Paste the example code into the class and modify it as follows. Note that there are two versions of code: one for WebLogic 8.1 and one for WebLogic 7.0. See the code example below and note the difference in the catch clause for WebLogic 7.0.
   a. Paste the code copied from the testing home page into the `main()` method.
   b. Move the import statement `import {package}.Bean1;` below the package statement at the top of the file and replace it with `import exportbean.generated.*;`.
   c. Add another import statement below it: `import java.io.*;`.
   d. Delete the ellipsis (…) from the copied code.
   e. Put a try/catch block around the code you copied from the Web Services Console.
   f. Replace the result statement with a print statement.
   g. Add the catch statement according to your version of WebLogic as shown in the following code examples.

Code for WebLogic 8.1

```
package test;

import exportbean.generated.*;
import java.io.*;

public class Test {
  public static void main(String[] args) {

    try {
          String wsdlUrl =
               "http://localhost:7001/web-services/exportbean?WSDL";
          Bean1 service = new Bean1_Impl(wsdlUrl);
          Bean1Port port = service.getBean1Port();

          System.out.println("Output is "+ port.getSample());
    } catch (Exception ex) {
      ex.printStackTrace();
        }
  }
}
```

Code for WebLogic 7.0

```
package test;

import exportbean.generated.*;
import java.io.*;

public class Test {
  public static void main(String[] args) {

    try {
          String wsdlUrl =
               "http://localhost:7001/web-services/exportbean?WSDL";
          Bean1 service = new Bean1_Impl(wsdlUrl);
          Bean1Port port = service.getBean1Port();

          System.out.println("Output is "+ port.getSample());
    } catch (IOException ex) {
      ex.printStackTrace();
        }
  }
}
```

7 Right-click `Test.java` in the project pane and choose Run Using Defaults. The output displays in the message pane: `Output is Sample string`.

Congratulations! You've completed the tutorial. You've exported a JavaBean as a web service, deployed it to the WebLogic Server, tested it using the Web Services Console, and created a client to test it. For more information about the WebLogic toolkit, see Chapter 8, "Using the WebLogic toolkit."

Tutorial: Creating a web service from an EJB application with WebLogic Server

This tutorial uses features in JBuilder Enterprise

This tutorial, which assumes knowledge of Enterprise JavaBeans (EJBs), illustrates how to create a web service from an EJB application using the WebLogic Application Server 7.x or 8.1 as the application server. The tutorial uses a sample in the JBuilder `samples` directory, `<jbuilder>/samples/tutorials/webservices/weblogic/Employee.jpx`. The sample contains an EJB application which accesses a database of employee records.

The tutorial includes the following steps:

- Configuring the project for web services.
- Deploying an EJB application as a web service.
- Generating client-side code from the WSDL
- Writing a client application to consume the service locally.

For more information on EJBs, see *Developing Enterprise JavaBeans*. For more information on web services and EJBs, see Chapter 6, "Developing EJBs as web services."

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Setting up the project

If you don't have the WebLogic Server 7.x or 8.1 configured yet, see "Using JBuilder with BEA WebLogic servers" in *Developing J2EE Applications* for instructions on how to configure it.

1. Open the sample, `Employee.jpx`, located in `<jbuilder>/samples/Tutorials/webservices/weblogic/`.
2. Complete the instructions in the sample project file, `EmployeeProject.html` before continuing to the next step.

Step 2: Configuring the project for web services

In this step, you'll use the Web Services Configuration wizard to create a web services configuration for a WebApp that hosts a web services server.

1. Choose File | New to open the object gallery.
2. Click the Web Services tab and double-click the Web Services Configuration icon. The Web Services Configuration wizard configures a toolkit for the web service and generates a web services server to host the service. Before you can configure the project, you must create an EAR for the EJB deployment and a WebApp to host the web services server.
3. Create a new EAR archive as follows. You'll deploy everything to an EAR file.
   a. Click the New button next to the EAR field to open the EAR wizard.
   b. Accept the default name for the EAR, `Employee`, and click Finish.
4. Choose WebLogic as the toolkit.
5. Create a new WebApp as follows:
   a. Click the New button next to the WebApp field to open the Web Application wizard.
   b. Accept `web-services` in the Name and the Directory fields and click OK to create the WebApp and return to the Web Services Configuration wizard.
6. Click Next to see that the wizard creates a Web Services Server runtime configuration for building the project and running the WebLogic Server.
7. Click Finish to close the wizard.

Step 3: Deploying the EJB as a web service

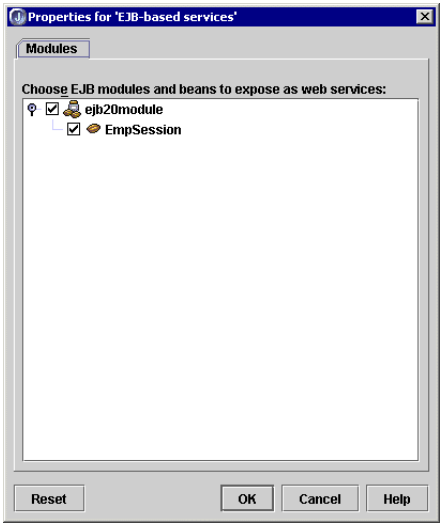
In this step, you'll deploy the EJB as a web service to an EAR file. When you deploy to the WebLogic Server, JBuilder automatically exposes any stateless session beans containing methods in the remote interface as web services.

1. Expand the new web-services WebApp node to see the nodes and the deployment file generated by the Web Services Configuration wizard. When you run the Web Services Server runtime configuration created by the wizard, all of the EJB modules and any stateless session beans in the project are automatically deployed as web services. You can modify which modules are deployed on the Properties page. In this example,

you want to expose everything in the project, so you won't make any changes.

a Expand the Web Service Components node.

b Right-click the EJB-based Services node and choose Properties. Here you'll see that the EJB module and bean are exposed as services. In this example, you want to expose everything in the project.



c Accept the defaults and click OK to close the dialog box.

2 Save the project.

3 Start the server by clicking the drop-down arrow next to the Run button on the main toolbar and choosing Web Services Server. This runtime configuration builds the project, automatically deploys the EJBs, and starts the WebLogic Server.

4 Right-click the Employee.eargrp node in the project pane and choose Deploy Options | Deploy to deploy the service to the WebLogic Server.

Step 4: Generating client-side code from the WSDL

Now that you've deployed the EJB application as a web service, you'll import the web service from the EAR and generate client classes and a WSDL to describe the service.

1 Right-click the Employee.eargrp and choose Import A Web Service to open the Import A Web Service wizard which generates the client-side Java classes and a WSDL.

2 Choose these options in the wizard:

   a Accept the EAR name, WebApp name, service to import, and click Next.

   b Change the package name to `com.borland.samples.emp.generated`.

   c Accept all the other wizard defaults.

   d Click Finish.

   e Expand the GeneratedWebServiceClients node in the project pane and double-click the `EmpSession_client.jar` to see the generated web service and the WSDL.

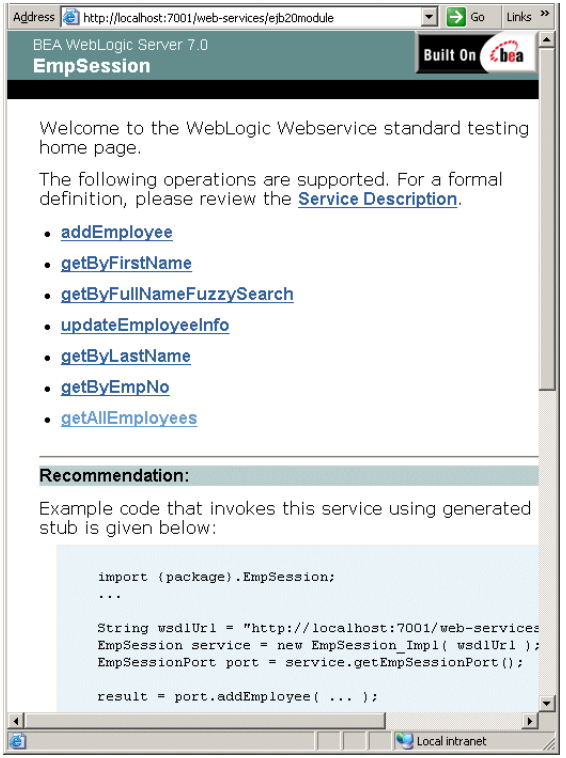
Step 5: Writing the client and consuming the service locally

Next, you'll write a simple client that will get all the employee records in the database and output them to the message pane. First, you'll use the Web Services Console to find sample code for writing the client that invokes the service. The Web Services Console displays the testing home page for the deployed service. Then, you'll add code to the client class that will get all the employee records and print them to the JBuilder message pane.

1 Choose Tools | Web Services Console to open the Web Services Console. For this tool to be active, you must have a web browser specified for the WebLogic Admin Console (Tools | Configure Servers). The Web Services Console displays the testing home page for the deployed service. For more information on configuring a web browser for the server, see "Using JBuilder with BEA WebLogic servers" in *Developing J2EE Applications*.

   If the deployment is successful, the exposed methods in EmpSession display on the testing home page in the Web Services Console. There's

also a link to the WSDL for the service and example code you'll copy and modify to invoke the service locally.



2 Copy these lines of code from the Home Page:

```
String wsdlUrl =
     "http://localhost:7001/web-services/ejb20module?WSDL";
EmpSession service = new EmpSession_Impl( wsdlUrl );
EmpSessionPort port = service.getEmpSessionPort();
```

3 Return to JBuilder and choose File | New Class to create a new class.

4 Make these modifications in the Class wizard:

   a Enter `com.borland.samples.emp.client` in the Package field.

   b Enter `Client` in the Class Name field.

   c Check Generate Main Method.

   d Uncheck Generate Default Constructor.

   e Click OK to close the wizard and create the new class.

5 Add these import statements below the package name:

```
import com.borland.samples.emp.generated.*;
import java.io.*;
```

6. Paste the code you copied from the WebLogic Web Services Home Page into the main method of the new Client class.
7. Select the code you just added and use *Shift+Ctrl+C* to put a try/catch statement around the code.
8. Add the code in bold to the catch statement.

```
catch (IOException ex) {
          ex.printStackTrace();
        }
```

9. Add a call to the `getAllEmployees()` method, statements to get information from the array, and a print statement to print out all the employee records in the database.

```
//call to getAllEmployees
EmployeeInfo [] employees = port.getAllEmployees();
//extract information from the array.
for (int i=0; i < employees.length; i++) {
  //check to see if any nulls are being returned from the database.
  if (employees[i] != null) {
    //if not null, print out the full name of the employee.
    System.out.println(employees[i].getFullName());
  }
```

The completed Client class should look like this:

```
public class Client {
  public static void main(String[] args) {
    try {
      String wsdlUrl =
           "http://localhost:7001/web-services/ejb20module?WSDL";
      EmpSession service = new EmpSession_Impl(wsdlUrl);
      EmpSessionPort port = service.getEmpSessionPort();
      //call to getAllEmployees
      EmployeeInfo [] employees = port.getAllEmployees();
      //extract information from the array.
      for (int i=0; i < employees.length; i++) {
      //check to see if any nulls are being returned from the database
        if (employees[i] != null) {
          //if not null, print out the full name of the employee.
          System.out.println(employees[i].getFullName());
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

10. Right-click the `Client.java` file tab in the editor and choose Make to compile the class.

11 Right-click the `Client.java` file tab in the editor and choose Run Using Client to run the class. The employee records are printed to the message pane.

Congratulations, you've completed the tutorial. You've configured the project for web services, deployed the EJB application as a web service, generated client-side classes, and tested it.

For more information about using the WebLogic toolkit, see Chapter 8, "Using the WebLogic toolkit."

Tutorial: Creating an asynchronous web service with WebLogic Server 8.1

This tutorial uses features in JBuilder Enterprise and WebLogic Server 8.1

This tutorial, which assumes knowledge of Enterprise JavaBeans (EJBs), illustrates how to create an asynchronous web service using WebLogic Application Server 8.1 as the application server. The tutorial uses a sample in the JBuilder `samples` directory, `<jbuilder>/samples/tutorials/webservices/weblogic/Employee.jpx`. The sample contains an EJB application which accesses a database of employee records.

The WebLogic toolkit uses a JMS-implementation for creating asynchronous web services. The JMS implementation requires a JMS server that queues the messages, a message-driven bean that processes JMS messages, and a JMS message type (data) that's sent between the client and the server.

In this example, you'll create a message-driven bean in the EJB module, create an asynchronous web service, modify the code in the message-driven bean and implement the business logic, and pass a request for employee data from the web service. The server will send the employee information in a return response through the MDB from the database back to the web service asynchronously.

The tutorial includes the following steps:

- Setting up the project.
- Configuring WebLogic Server 8.1 for JMS.
- Creating a message-driven bean.
- Creating an asynchronous web service.
- Implementing the business logic in the message-driven bean.
- Writing a client application to consume the service.
- Deploying and testing the web service.

See also

- "Creating asynchronous web services" on page 8-10
- Chapter 6, "Developing EJBs as web services"
- "Writing an Asynchronous Client Application" in "Programming WebLogic Web Services" in the WebLogic documentation at `http://edocs.bea.com/wls/docs81/webserv/index.html`

For more information on EJBs, see *Developing Enterprise JavaBeans*.

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Setting up the project

If you don't have WebLogic Server 8.1 configured yet, see "Using JBuilder with BEA WebLogic servers" in *Developing J2EE Applications* for instructions on how to configure it.

1. Open the sample, `Employee.jpx`, located in `<jbuilder>/samples/Tutorials/webservices/weblogic`.
2. Complete the instructions in the sample project file, `EmployeeProject.html`, for connecting to and refreshing the database.

Step 2: Configuring WebLogic Server 8.1 for JMS

In this step, you'll configure the WebLogic Server 8.1 for JMS and add a property to the DataSource, which JBuilder automatically sets up for you. For more detailed information, see the WebLogic documentation at `http://edocs.bea.com`.

1. Start WebLogic Server 8.1 in JBuilder by choosing the Server runtime configuration from the drop-down list next to the Run button on the main toolbar.
2. Open the WebLogic Admin Console (Tools | WebLogic Admin Console).
3. Set the property, Emulate Two-Phase Commit, for the JDBC DataSource that JBuilder added to the configuration. This allows you to connect to the data source in the project. If JBuilder didn't add the DataSource, create a TX data source named DataSource.

4 Set up the following for JMS:
   a Create a JMS connection factory with a JNDI name, `EmployeeJMSFactory`. Enable the property, XA-Enabled Transactions.
   b Create a JMS server and create two JMS queues with JNDI names, `EmployeeRequestQueue` and `EmployeeResponseQueue`.
5 Exit the WebLogic Admin Console and restart the server.

Step 3: Creating a message-driven bean

Next, you need to create a message-driven bean. Message-driven beans listen and respond to JMS messages that are sent to the server.

1 Double-click ejbmodule20 in the project pane to open it in the EJB Designer.
2 Right-click in the EJB Designer and choose Create EJB | Message-Driven Bean.
3 Click the new message-driven bean in the EJB Designer and enter `EmpMdb` in the Bean Name field.
4 Click the Classes And Packages button and enter `com.borland.samples.emp.ejb` in the Package Name field.
5 Enter `EmployeeRequestQueue` in the Destination Name field.
6 Enter `EmployeeJMSFactory` in the Connection Factory Name field.
7 Enter `1` in the Initial Pool Size field.
8 Enter `10` in the Maximum Pool Size field.
9 Choose File | Save to save the changes.

Step 4: Creating an asynchronous web service

In this step, you'll configure your project for web services and create the asynchronous web service with the Export As An Asynchronous Web Service wizard.

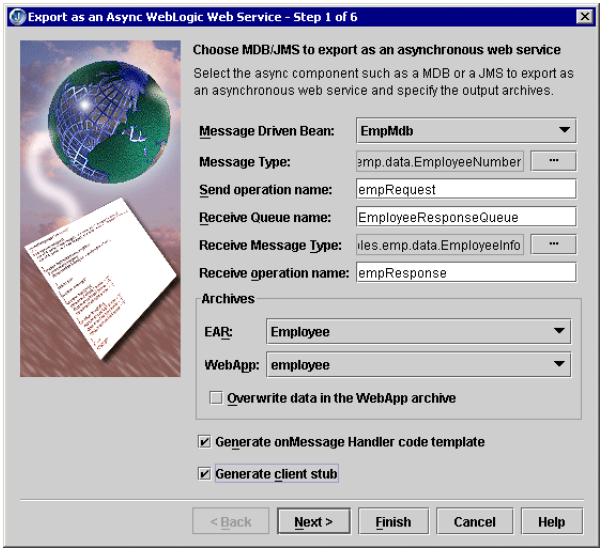
1 Expand the `com.borland.samples.emp.data` package node in the project pane.

2. Right-click `EmployeeNumber.java` and choose Export As An Async Web Service. First, the Web Services Configuration opens so you can configure the project. The Web Services Configuration wizard configures a toolkit for the web service and generates a web services server to host the service. Before you can configure the project, you must create an EAR for the EJB deployment and a WebApp to host the web services server. Then, the Export As An Asynchronous Web Service wizard will open.
3. Complete these steps in the Web Services Configuration wizard.
   a. Click the New button next to the EAR field to create an EAR.
   b. Accept the default name, `Employee`, in both Name fields and click Finish to close the EAR wizard and return to the Web Services Configuration wizard.
   c. Click the New button next to the WebApp field to open the Web Application wizard and create a WebApp to host the service.
   d. Enter `employee` in the Name and the Directory fields and click OK to create the WebApp and return to the Web Services Configuration wizard.
   e. Choose WebLogic as the toolkit.
   f. Click Finish to close the wizard.

Now, that the project is configured for web services, the Export As An Asynchronous Web Service wizard displays. Complete these steps in the wizard to create the asynchronous web service.

1. Enter `empRequest` for the Send Operation Name. The send operation sends the employee request to the web service.
2. Enter `EmployeeResponseQueue` for the Receive Queue Name. The receive queue is where the message-driven bean sends the response.
3. Click the ellipsis (…) button next to the Receive Message Type field and enter `emp` in the Search field of the dialog box. Choose `com.borland.samples.emp.data.EmployeeInfo` from the list of classes and press *Enter*. The message type is the piece of data sent back and forth.

4 Enter `empResponse` for the Receive Operation Name. Step 1 of the wizard should look like this:



5 Click Next twice to continue to Step 3.

6 Change the package name to `com.borland.samples.emp.async`. The JMS access helper class will be generated in this package.

7 Click Next and accept the defaults on Step 4.

8 Click Next again to continue to Step 5.

9 Change the package name to `com.borland.samples.emp.async.generated`. The client stub for accessing the web service will be generated in this package.

10 Click Finish to close the wizard and generate the Java classes for the asynchronous web service. The

11 Expand the generated WebServicesClient node and the `com.borland.samples.emp.async.generated` package in the client JAR to see the generated Java classes. Also notice that a JMS class is created in the `com.borland.samples.emp.async` package.

12 Expand the `employee` WebApp node. Then expand the Web Service Components node and the Async Component-based Services node to see the `servicegen.wldu` deployment file generated by the wizard.

13 Double-click `servicegen.wldu` to view it in the editor. Notice that all the fields you filled out in the wizard are specified in the deployment file. There are two JMS operations: one that sends the request to the database on the server and one that returns the employee information from the database.

Step 5: Implementing the business logic in the message-driven bean

Now, you'll implement the business logic, so the message-driven bean can process the JMS messages it receives. You'll do this in the `onMessage()` method. Before implementing the business logic, you need to define two variables, and modify the `setMessageDrivenContext()` method.

1. Return to the EJB Designer, right-click the EmpMdb message-driven bean, and open the DD Editor.
2. Click the EJB References tab and click the Add button to add a reference.
3. Enter `ejb/EmpSession` in the Name field and click OK.
4. Check the isLink check box and choose EmpSession from the Link drop-down box.
5. Return to the EJB designer, right-click MDBQueue, and choose View Bean Source to open the source code for the message-driven bean.
6. Declare the two variables indicated in bold below the `messageDrivenContext` variable:

```
public class EmpMdbBean implements MessageDrivenBean, MessageListener {
  MessageDrivenContext messageDrivenContext;
  private EmpSessionHome empSessionHome;
  private EmpSession empSession;

  . . .
```

7. Create a try/catch block in the `setMessageDrivenContext()` method as follows:
   a. Enter the three lines of code in bold after the messageDrivenContext declaration

```
public void setMessageDrivenContext(MessageDrivenContext
    messageDrivenContext) {
  this.messageDrivenContext = messageDrivenContext;
  Context context = new InitialContext();
  Object object = context.lookup("java:comp/env/ejb/EmpSession");
  empSessionHome = (EmpSessionHome) javax.rmi.PortableRemoteObject.narrow(
      object, EmpSessionHome.class);

    . . .
}
```

b Select the new code and use *Shift+Ctrl+C* to optimize imports and to surround it with a try/catch block. The try/catch block should now look like this:

```
try {
     Context context = new InitialContext();
     Object object = context.lookup("java:comp/env/ejb/EmpSession");
     //cast the home interface using an RMI cast
     empSessionHome = (EmpSessionHome) javax.rmi.PortableRemoteObject.narrow(
         object, EmpSessionHome.class);
   }
   catch (ClassCastException ex1) {
   }
   catch (NamingException ex1) {
   }
```

8 Add this line of code to each catch block to print out any exceptions:

```
ex.printStackTrace();
```

Your catch blocks should look like this:

```
catch (ClassCastException ex) {
      ex.printStackTrace();
   }
   catch (NamingException ex) {
     ex.printStackTrace();
   }
```

Now, you're ready to implement the business logic in the `onMessage()` method.

1 Enter this line of code below the comment, `//implement your business logic`. This code creates an instance of the session bean remote interface.

```
empSession = empSessionHome.create();
```

Use code templates to put a try/catch block around it. Your code should look like this:

```
//Implement your business logic using data in EmployeeNumber.
   try {
     empSession = empSessionHome.create();
   }
   catch (CreateException ex1) {
   }
```

2 Change the JMS exception in the catch block to Exception, so all exceptions are caught and add the printStackTrace() method:

```
catch (Exception ex) {
      ex.printStackTrace();
   }
```

3 Add this line of code below the comment, `//Populate employeeInfo with response data`. This code populates employeeInfo with response data before sending it to the client. First, an instance of the session bean remote interface is created. Then the empNo for the employee is retrieved and set it in the employeeInfo object, which is sent back to the receive queue (EmployeeResponseQueue).

```
employeeInfo = empSession.getByEmpNo(employeeNumber.getEmpNo());
```

4 Surround it with a try/catch block and add the printStackTrace() method to the catch block. Your code should look like this:

```
//Populate employeeInfo with response data before sending to client.
   try {
     employeeInfo = empSession.getByEmpNo(employeeNumber.getEmpNo());
   }
   catch (RemoteException ex2) {
     ex2.printStackTrace();
   }
```

The completed code for the message-driven bean, EmpMdbBean, should look like this:

```
package com.borland.samples.emp.ejb;

import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import com.borland.samples.emp.data.EmployeeNumber;
import com.borland.samples.emp.data.EmployeeInfo;
import com.borland.samples.emp.async.Jms1;
import java.rmi.*;

public class EmpMdbBean implements MessageDrivenBean, MessageListener {
  MessageDrivenContext messageDrivenContext;
  private EmpSessionHome empSessionHome;
  private EmpSession empSession;

  public void ejbCreate() throws CreateException {
    /**@todo Complete this method*/
  }
  public void ejbRemove() {
    /**@todo Complete this method*/
  }
  public void onMessage(Message msg) {
    /**
 * @todo
 * The OnMessage handler is called when data arrives in the Queue from a Web
 * Services client. The following template code retrieves the data from the Queue
 * and recreates an instance of a strongly typed data of type EmployeeNumber,
 * exactly as sent by the client.
 */
```

```
    EmployeeNumber employeeNumber = null;
    EmployeeInfo employeeInfo = null;
    try {
      if (msg instanceof ObjectMessage) {
        employeeNumber = (EmployeeNumber)((ObjectMessage) msg).getObject();
      }
    }
    catch(JMSException ex) {
      ex.printStackTrace();
    }
    //Implement your business logic using data in EmployeeNumber.
    try {
      empSession = empSessionHome.create();
    }
    catch (CreateException ex1) {
      ex1.printStackTrace();
    }
    catch (RemoteException ex1) {
      ex1.printStackTrace();
    }
    employeeInfo = new EmployeeInfo();
    //Populate employeeInfo with response data before sending to client.
    try {
      employeeInfo = empSession.getByEmpNo(employeeNumber.getEmpNo());
    }
    catch (RemoteException ex2) {
      ex2.printStackTrace();
    }

    try {
      Jms1 jms1 = new Jms1();
      jms1.sendObject(employeeInfo);
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
    /**@todo Complete this method*/
  }
  public void setMessageDrivenContext(MessageDrivenContext
      messageDrivenContext) {
    this.messageDrivenContext = messageDrivenContext;
    try {
      InitialContext context = new InitialContext();
      Object object = context.lookup("java:comp/env/ejb/EmpSession");
      empSessionHome =
          (EmpSessionHome)javax.rmi.PortableRemoteObject.narrow(object,
             EmpSessionHome.class);
    }
    catch (ClassCastException ex) {
      ex.printStackTrace();
    }
    catch (NamingException ex) {
      ex.printStackTrace();
    }
  }
}
```

Step 6: Writing a test client

Now that you've deployed the web service, you'll write a simple client that will get employee information from the database on the server and output it to the message pane.

1. Choose File | New Class to create a new test class.
2. Change the package name to `com.borland.samples.emp.async`.
3. Enter `Test` in the Name field.
4. Click OK to close the wizard.
5. Remove the contents of the new class.
6. Copy and paste this code into the new Test class:

```
package com.borland.samples.emp.async;

import weblogic.webservice.async.FutureResult;
import weblogic.webservice.async.AsyncInfo;
import weblogic.webservice.async.ResultListener;
import weblogic.webservice.async.InvokeCompletedEvent;
import javax.xml.rpc.*;
import java.io.*;
import java.rmi.RemoteException;
import com.borland.samples.emp.async.generated.*;

public class Test {

  public static void main(String[] args) {
    System.out.println("calling polling method");
    pollingMethod();
    System.out.println("calling callback method");
    callbackMethod();
  }

  private static void pollingMethod() {
    try {
      //Get the EmpMdb instance from the generated stub.
      EmpMdbPort port = new EmpMdb_Impl().getEmpMdbPort();

      //Make the request call. Since this is an async call, it is done with a
      //pair of methods - start<MethodName> and end<MethodName>.
      //The start method returns a future result data type, where a result
      //will be populated by the system at a later time
      FutureResult futureResult = port.startEmpRequest(new EmployeeNumber
          (new java.lang.Short((short)2)), null);
      //Keep polling FutureResult, if it isn't done
      while( !futureResult.isCompleted() ){
        Thread.sleep( 300 );
      }
```

```
    //Future result is done, so get result data. In this case, it's a void
    //so there isn't a return value ... just want to know it's done.
    port.endEmpRequest(futureResult);

    //The Server MDB, which is listening on the Queue, will be invoked
    //asynchronously. The MDB processes the request from the Queue. In this
    //example, it picks up employee number, looks it up in the database, and
    //posts the employee info in another response queue.
    //The following client code picks up that response data asynchronously.
    futureResult = port.startEmpResponse(null);
    while( !futureResult.isCompleted() ){
      Thread.sleep( 300 );
    }
    //Future result is done, so get result data. In this example, the result
    //data is an  EmployeeInfo data type. Get and print data out.
    EmployeeInfo EmployeeInfo = port.endEmpResponse(futureResult);
    System.out.println("EmployeeInfo: " + EmployeeInfo.getFullName());
    System.out.println("");
  }
  catch (Exception ex) {
  }
}

private static boolean done;
private static void callbackMethod() {
  try {

    //Get the EmpMdb instance from the generated stub.
    EmpMdbPort port = new EmpMdb_Impl().getEmpMdbPort();
    FutureResult futureResult = port.startEmpRequest(new EmployeeNumber
        (new java.lang.Short((short)2)), null);
    while( !futureResult.isCompleted() ){
      Thread.sleep( 300 );
    }
    port.endEmpRequest(futureResult);


    // In the callback method, instead of polling again and again, we just
    // register a callback and wait.
    //The callback is defined by the AsyncInfo class. The onCompletion()
    //method will be called by the system, when response is available.
    AsyncInfo asyncInfo = new AsyncInfo();
    asyncInfo.setResultListener( new ResultListener(){
    public void onCompletion( InvokeCompletedEvent event ){
      EmpMdbPort source = (EmpMdbPort) event.getSource();
      try {
        //Since we are in the oncomplete event, we know that the
        //response is available right now, so get response data.
        EmployeeInfo EmployeeInfo =
            source.endEmpResponse(event.getFutureResult());
        System.out.println("EmployeeInfo: " + EmployeeInfo.getFullName());
        System.out.println("");
      }
```

```
        catch (RemoteException e) {
          e.printStackTrace(System.out);
        }
        finally {
          done = true;
        }
      }
      });
      done = false;
      //Initialize the call by the start method.
      futureResult = port.startEmpResponse(asyncInfo);
      while(!done) {
        Thread.sleep( 300 );
      };
    }
    catch (Exception ex) {
    }
  }
}
```

This test has two methods: a `pollingMethod()` method and a `callbackMethod()` method. Both methods return the full name for employee number two. Read the comments in the code to understand in detail what the methods in the Test class do.

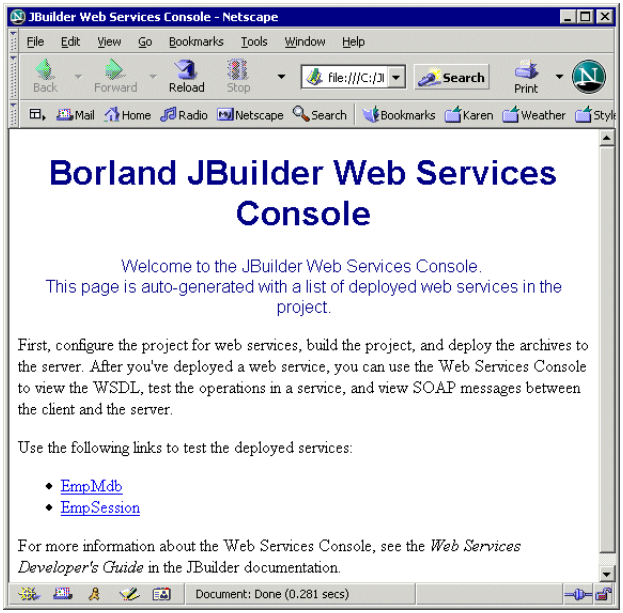
Step 7: Deploying and testing the web service

Once the business logic has been implemented, you're ready to deploy the web service and test it with the Test client application.

1. Choose Project | Make Project to build the project.
2.  Choose the Web Services Server runtime configuration from the drop-down list next to the Run button to run the project and the server.
3. Right-click the Employee.eargrp node in the project pane and choose Deploy Options | Deploy to deploy the service to the WebLogic Server.
4. Right-click `Test.java` in the `com.borland.samples.emp.async` package in the project pane and choose Run Using "Client" to run the client test application. The test returns the full name for employee number two and prints it to the message pane:

```
calling polling method
EmployeeInfo: Nelson, Robert
calling callback method
EmployeeInfo: Nelson, Robert
```

After you've deployed a web service with the WebLogic toolkit, you can also use the Web Services Console (Tools | Web Services Console) to view the WSDL, test the service, and view the SOAP messages between the client and the server. For more information on the Web Services Console, see "Testing deployed services" on page 8-31.



Congratulations, you've completed the tutorial. You've configured the WebLogic Server for JMS, created a message-driven bean, implemented the business logic in the message-driven bean, created an asynchronous web service, deployed the web service, and tested it with a test client.

For more information about using the WebLogic toolkit, see Chapter 8, "Using the WebLogic toolkit."

Chapter

13

General tutorials

This is a feature of JBuilder Enterprise

The following tutorial is available for all toolkits:

- "Tutorial: Browsing UDDI web services" - Explains how to use the Web Services Explorer to browse web services and generate Java classes from a WSDL document.

Tutorial: Browsing UDDI web services

This tutorial uses features in JBuilder Enterprise

JBuilder provides the Web Services Explorer for browsing and querying UDDI registries. UDDI registries are databases used by businesses to register their web services and to search for other available web services. These registries are based on the *Universal Description, Discovery and Integration* (UDDI) framework. UDDI uses the Extensible Markup Language (XML) and may also use the Web Services Description language (WSDL) for describing a service.

In this tutorial, you'll learn how to use the Web Services Explorer by executing the following tasks:

- Browse web services at the XMethods site.
- Browse tModels.
- Find software publishers at the Microsoft UDDI site.
- Generate Java classes.

Important The UDDI feature of the Web Services Explorer requires the use of Sun's Java Secure Socket Extension (JSSE). If you're using JDK 1.3 or earlier, you must download and install the JSSE Extension from Sun at `http://java.sun.com/products/jsse/index-103.html`.

See also

- "UDDI terms and definitions" on page 10-4
- UDDI specification at `http://www.uddi.org/specification.html`
- Chapter 10, "Browsing and publishing web services"

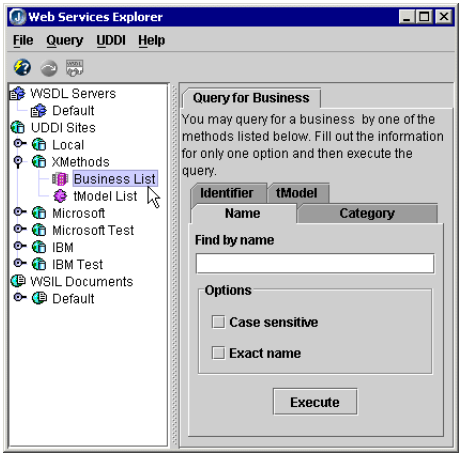
See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Browsing web services at the XMethods site

In this step, you'll use the Web Services Explorer to browse some of the web services available at XMethods, an organization that promotes the development, deployment, and use of web services. For more information about XMethods, visit `http://www.xmethods.net/`.

1. Choose Tools | Web Services Explorer to open the Web Services Explorer.
2. Expand the XMethods operator site node, a child node of the UDDI Sites node, in the tree on the left.
3. Select the Business List node to display the Query For Business page on the right.



4 Enter the letters `xm` in the Find By Name field to find all the businesses at the site beginning with "xm" and choose the Execute button on the Query For Business page to execute the query. You can also choose Query | Execute or click the Execute Query button on the toolbar.

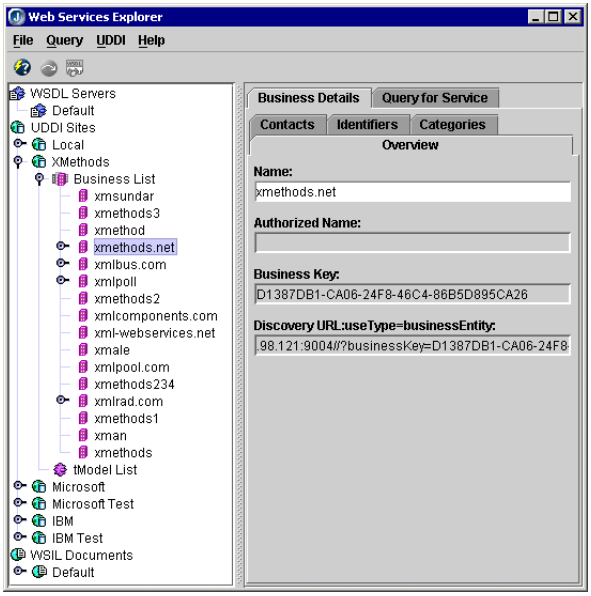
Tip If you want to find all the businesses at a site, leave the Find By Name field blank.

Important You can increase the timeout for the Web Services Explorer. Choose UDDI | Connection Settings and enter the new timeout in the Timeout field. The server may also truncate your query results if it determines that the results are too large.

5 Expand the XMethods Business List node to see all the businesses beginning with "xm" at the XMethods site.

6 Select the xmethods.net node in the tree.

Notice that the Business Details page displays on the right with overview information as well as contact, identifier, and category information, if available. Also, the Query For Service tab displays on the right. If you know the name of the service, you can narrow your search even further by entering the name of the service in the Find By Name field on the Query For Service page.



7 Expand the xmethods.net node and notice that there are several services available.

8 Choose the BabelFish service node. Now the Service Details page displays on the right with general information about the web service, such as service name, description, and service key.

9 Expand the BabelFish service node and choose the next node below it, SOAP binding. The Binding Details page displays on the right with a description of how to bind to the service, the access point to the service, and the access point URL type. Some of these fields may be blank if they aren't required.

10 Expand the SOAP binding node and choose the next node, <No Description>. The TModel Instance Details page displays on the right with details about the tModel, such as a description of the service, tModel key, instance description, overview document description and URL, and any instance parameters. Many of these fields are blank, because they are optional.

11 Navigate to the last xmethods.net node under the <No Description> node, the BabelFish node. This is the tModel node. Examine the TModel details displayed to the right on the Overview tab, such as name, description, TModel key, and overview document description and URL. Some of the information is the same as the TModel Instance Details page. The description field describes what the web service does. The overview document URL field contains the URL that points to a WSDL document, which describes the specification for the web service in the Web Services Description Language. The WSDL document, written in XML, describes the service interface and the XML messages that go in and out of the web service. By examining the WSDL document, you can write programs that invoke and interact with the web service.

Note 

At this point you could use the Import A Web Service button on the toolbar to open the Import A Web Service wizard. This wizard generates Java classes from the WSDL. See "Step 4: Generating Java classes" on page 13-7. This particular service requires two input parameters, `translationmode(xsd:string)` and `sourcedata (xsd:string)`, and has a return of type string. Visit the XMethods website at `http://www.xmethods.net/` to find out more about this web service.

Step 2: Browsing tModels

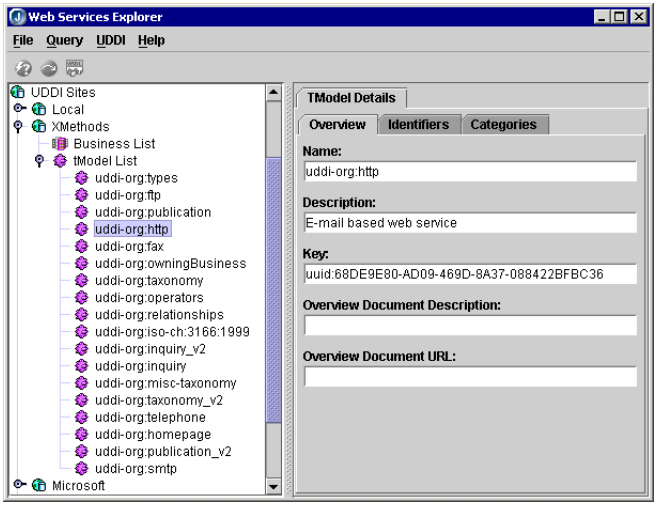
In this step, you'll find all the tModels that begin with the uddi-org name. TModels are references to a technical specification of a service. UDDI uses tModels to point to the specification, instead of holding all the data within the registry. Each tModel has a unique identifier called a Universal Unique Identifier (UUID). UDDI assigns these identifiers when a web service is first published. Programmers who develop applications that consume web services, use this UUID to access information about the specification for the web service.

1 Expand the Microsoft node in the UDDI Sites tree and choose the tModel List node in the tree to display the Query For tModel page.

2 Enter `uddi-org` in the Find By Name field to find all tModels of this type at the Microsoft site.

3 Click the Execute button to execute the query.

4 Expand the tModel List node and notice the different categories for uddi-org, such as http, types, publication, ftp, fax, taxonomy, and so on. Choose one of the nodes, such as http.



5 Click the Categories tab to see how this tModel is categorized.

Step 3: Finding software publishers at the Microsoft UDDI site

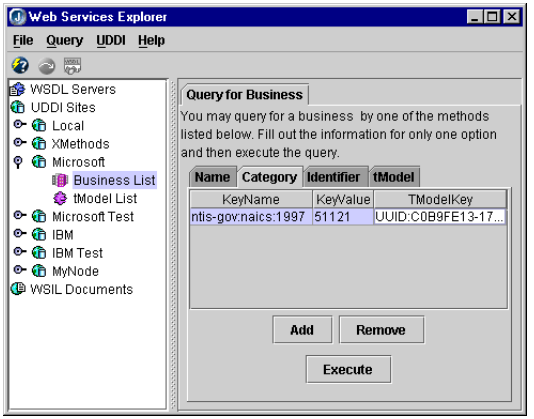
Next, you'll use a category search to find all the software publishers at the Microsoft UDDI site.

1 Select the Microsoft Business List node to display the Query For Business page.

2 Choose the Category tab.

3 Choose `ntis-gov:naics:1997` from the KeyName drop-down list. The North American Industry Classification System (NAICS) is an industrial classification system that has replaced the U.S. Standard Industrial Classification (SIC) system. Notice that the TModelKey field is filled out automatically.

4 Enter `51121` in the KeyValue field. This is the NAIC code for software publishers.



5 Press *Enter* to commit the value.

6 Click the Execute button to execute the query.

7 Expand the Business List node to see the results of the query.

8 Select a business in the list and choose the Categories tab on the Business Details page. One of the categories should be software publisher with a KeyValue of 51121. Notice that businesses often have multiple categories.

9 Select the IBM Corporation node under the Microsoft Business List node and click the Identifiers tab on the Business Details page. Each company also has identifiers. Here, you see that IBM has a D-U-N-S identifier, which is a Dun & Bradstreet Number Identifier System that uses a nine-digit code. The identifier keyName is D-U-N-S and the keyValue is the nine-digit code: 00-136-8083. Each identifier also has a unique UUID (Universal unique identifier) tModelKey assigned to it.

10 Expand the IBM Corporation node and select Buy From IBM.

11 Choose the Categories tab on the Service Details page to see how IBM is categorized. Because IBM is a large company with a wide range of products, it's registered under several categories or KeyNames. Notice that many of the KeyNames begin with NAICS (North American Industry Classification System) and UNSPSC (Universal Standard Products and Services Classification), which are different classification systems. Each KeyName has a corresponding unique tModelKey and KeyValue.

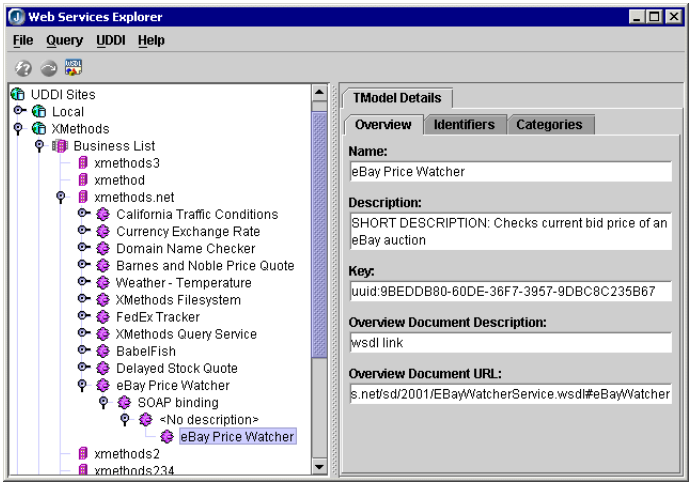
Step 4: Generating Java classes

JBuilder provides the Import A Web Service wizard for quickly generating Java files from an existing WSDL document. A WSDL document, Web Services Description Language, describes a web service interface and the XML messages that go in and out of the web service. TModels may point to a WSDL description, which is written in XML.

In this step, you'll create a project, configure it for web services, search for a web service with the Web Services Explorer, and use the Import A Web Service wizard to generate Java classes from the WSDL document for the web service.

1. Return to the JBuilder IDE and choose File | New Project to open the Project wizard and create a new project.
2. Enter `WsdlToJava` in the Name field and click Finish to create the project. To use the WebLogic toolkit to generate the Java classes, set the server for the project to WebLogic on the Server page of the Project Properties dialog box (Project | Project Properties).
3. Return to the Web Services Explorer and expand the XMethods node in the tree.
4. Select the Business List node and enter `xmethods` in the Find By Name Field on the Name tab of the Query For Business page. Click Execute or press *Enter* to execute the query.
5. Expand the xmethods.net node under the Business List node.
6. Choose the web service, eBay Price Watcher.
7. Continue navigating down the eBay Price Watcher node and select the last node, eBayPriceWatcher, which is the tModel node. When the tModel node is selected, the toolbar button and File menu for the Import A Web Service wizard are enabled. When you open the wizard, with the tModel node selected, the WSDL field is automatically filled out with the correct WSDL document for that service. The tModel node

must be selected to use the Import A Web Service wizard in the Web Services Explorer.



8 Click the Import A Web Service button on the toolbar to open the Import A Web Service wizard or choose File | Import A Web Service.

9 Choose the toolkit, if required, and continue through the steps of the wizard choosing the desired options. For more information on this wizard, see the appropriate JBuilder documentation for the toolkit or click the Help button on each step of the wizard.

For Axis, see "Importing a WSDL" on page 7-5. For WebLogic, see "Importing a WSDL as a web service" on page 8-7.

10 Click Finish to close the wizard and generate the Java classes from the WSDL.

Java classes are generated in a package node in the project pane. The Import A Web Service wizard generates Java classes from the WSDL. It generates a package name based on the WSDL's target namespace or one that you specify. It also adds the necessary libraries to the project.

11 Return to the Web Services Explorer and highlight the URL for the WSDL document in the Overview Document URL field.

12 Copy and paste the URL into an XML-compliant browser, such as Internet Explorer, to open the WSDL document. Here you'll see the methods used to access the web service. Methods are listed in the XML `<operation>` element. For example, one of the methods, `getCurrentPrice`, is represented in the WSDL document as: `<operation name="getCurrentPrice">`.

See also

- "Tutorial: Generating a web service from a WSDL document" on page 11-8

Congratulations! You've completed the tutorial. The Web Services Explorer also includes these additional features:

- Publishing web services
- Browsing web services on Axis servers, a feature of the Axis toolkit
- Publishing web services from Axis servers, a feature of the Axis toolkit
- Browsing web services with WSIL documents
- Monitoring UDDI messages

To learn more about the Web Services Explorer and its features, see Chapter 10, "Browsing and publishing web services," or choose Help | Contents in the Web Services Explorer.

Appendix

A

Modifying enterprise application server settings for web services

This is a feature of JBuilder Enterprise

The JBuilder web services features are supported on the following enterprise application servers:

- WebLogic toolkit: WebLogic Server 7.0 SP2 and 8.1
- Axis and SOAP toolkits:
  - Borland Enterprise Server 5.1.x, 5.2, and 5.2.1
  - WebLogic Server 7.0 SP2 and 8.1
  - WebSphere Application Server 4.0 AES/AE and 5.0
  - Sybase 4.1.3 and 4.2

Because enterprise application servers require different settings when using the web services features, you may need to make modifications when using the web services wizards and when deploying your application.

For information on configuring these servers in JBuilder, see "Configuring the target application server settings" in *Developing J2EE Applications* and "Release Notes" (Help | Release Notes).

Borland Enterprise Server 5.1.x, 5.2, and 5.2.1

Support for this application server is not provided with the JBuilder WebLogic Edition.

When running multiple partitions in JBuilder with archives deployed across these partitions, ensure that the naming service for only one of the partitions is enabled. To do this,

1. Click on Project | Project Properties.
2. Click the Run tab.
3. Edit the server runtime configuration.

Note If a server runtime configuration does not exist, create a new runtime configuration and click the Server tab.

4. Uncheck the Naming/Directory service in the Services list. Use this configuration to start up partitions without the naming service.
5. Create a new runtime configuration with the naming service turned on to launch one of the partitions with the naming service.
6. Ensure that the EJB and Naming/Directory services are enabled on the same partition and make sure that you start this partition first.

WebSphere Application Server 4.0 AE/AES

Support for this application server is not provided with the JBuilder WebLogic Edition.

After creating the web services server node with the Web Services Configuration wizard, move Axis to the top of the required libraries list in the Project Properties dialog box:

1. Choose Project | Project Properties.
2. Choose the Required Libraries tab on the Paths page.
3. Select Axis and choose the Move Up button to move it to the top of the list.

Ensure that you do not have the option Make Project Libraries Available On Run checked when you start the server or the server will not start up due to conflicting XML parser versions.

To add any libraries to the server's classpath at startup, add them on the Required libraries tab in the server configuration for WebSphere 4.0 AE/AES (Tools | Configure Servers).

Index

U

W