CHAPTER 19

Building Web Services with JAX-WS

Java API for XML Web Services (JAX-WS) is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as Remote Procedure Call-oriented (RPC-oriented) web services.

In JAX-WS, a web service operation invocation is represented by an XML-based protocol, such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-WS API hides this complexity from the application developer. On the server side, the developer specifies the web service operations by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: A JAX-WS client can access a web service that is not running on the Java platform, and vice versa. This flexibility is possible because JAX-WS uses technologies defined by the W3C: HTTP, SOAP, and WSDL. WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

Note – Several files in the JAX-WS examples depend on the port that you specified when you installed the GlassFish Server. These tutorial examples assume that the server runs on the default port, 8080. They do not run with a nondefault port setting.

The following topics are addressed here:

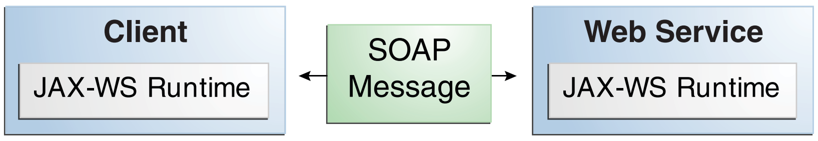
- "Creating a Simple Web Service and Clients with JAX-WS" on page 368
- "Types Supported by JAX-WS" on page 377
- "Web Services Interoperability and JAX-WS" on page 379
- "Further Information about JAX-WS" on page 379

Creating a Simple Web Service and Clients with JAX-WS

This section shows how to build and deploy a simple web service and two clients: an application client and a web client. The source code for the service is in the *tut-install*`/examples/jaxws/helloservice/` directory, and the clients are in the *tut-install*`/examples/jaxws/appclient/` and *tut-install*`/examples/jaxws/webclient/` directories.

Figure 19–1 illustrates how JAX-WS technology manages communication between a web service and a client.

FIGURE 19–1 Communication between a JAX-WS Web Service and a Client



The starting point for developing a JAX-WS web service is a Java class annotated with the `javax.jws.WebService` annotation. The `@WebService` annotation defines the class as a web service endpoint.

A *service endpoint interface* or *service endpoint implementation* (SEI) is a Java interface or class, respectively, that declares the methods that a client can invoke on the service. An interface is not required when building a JAX-WS endpoint. The web service implementation class implicitly defines an SEI.

You may specify an explicit interface by adding the `endpointInterface` element to the `@WebService` annotation in the implementation class. You must then provide an interface that defines the public methods made available in the endpoint implementation class.

The basic steps for creating a web service and client are as follows:

1. Code the implementation class.
2. Compile the implementation class.
3. Package the files into a WAR file.

4. Deploy the WAR file. The web service artifacts, which are used to communicate with clients, are generated by the GlassFish Server during deployment.
5. Code the client class.
6. Use a `wsimport` Ant task to generate and compile the web service artifacts needed to connect to the service.
7. Compile the client class.
8. Run the client.

If you use NetBeans IDE to create a service and client, the IDE performs the `wsimport` task for you.

The sections that follow cover these steps in greater detail.

Requirements of a JAX-WS Endpoint

JAX-WS endpoints must follow these requirements.

- The implementing class must be annotated with either the `javax.jws.WebService` or the `javax.jws.WebServiceProvider` annotation.
- The implementing class may explicitly reference an SEI through the `endpointInterface` element of the `@WebService` annotation but is not required to do so. If no `endpointInterface` is specified in `@WebService`, an SEI is implicitly defined for the implementing class.
- The business methods of the implementing class must be public and must not be declared `static` or `final`.
- Business methods that are exposed to web service clients must be annotated with `javax.jws.WebMethod`.
- Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types. See the list of JAXB default data type bindings at `http://docs.oracle.com/javaee/5/tutorial/doc/bnazq.html#bnazs`.
- The implementing class must not be declared `final` and must not be `abstract`.
- The implementing class must have a default public constructor.
- The implementing class must not define the `finalize` method.
- The implementing class may use the `javax.annotation.PostConstruct` or the `javax.annotation.PreDestroy` annotations on its methods for lifecycle event callbacks.

  The `@PostConstruct` method is called by the container before the implementing class begins responding to web service clients.

  The `@PreDestroy` method is called by the container before the endpoint is removed from operation.

Coding the Service Endpoint Implementation Class

In this example, the implementation class, `Hello`, is annotated as a web service endpoint using the `@WebService` annotation. `Hello` declares a single method named `sayHello`, annotated with the `@WebMethod` annotation, which exposes the annotated method to web service clients. The `sayHello` method returns a greeting to the client, using the name passed to it to compose the greeting. The implementation class also must define a default, public, no-argument constructor.

```
package helloservice.endpoint;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Hello {
    private String message = new String("Hello, ");

    public void Hello() {
    }

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Building, Packaging, and Deploying the Service

You can use either NetBeans IDE or Ant to build, package, and deploy the `helloservice` application.

▼ To Build, Package, and Deploy the Service Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

   tut-install`/examples/jaxws/`
3. **Select the `helloservice` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**

6 **In the Projects tab, right-click the `helloservice` project and select Deploy.**

This command builds and packages the application into `helloservice.war`, located in *tut-install*`/examples/jaxws/helloservice/dist/`, and deploys this WAR file to the GlassFish Server.

Next Steps You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/helloservice/HelloService?wsdl` in a web browser. Now you are ready to create a client that accesses this service.

▼ To Build, Package, and Deploy the Service Using Ant

1 **In a terminal window, go to:**

tut-install`/examples/jaxws/helloservice/`

2 **Type the following command:**

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `helloservice.war`, located in the `dist` directory.

3 **Make sure that the GlassFish Server is started.**

4 **Type the following:**

```
ant deploy
```

Next Steps You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/helloservice/HelloService?wsdl` in a web browser. Now you are ready to create a client that accesses this service.

Testing the Methods of a Web Service Endpoint

GlassFish Server allows you to test the methods of a web service endpoint.

▼ To Test the Service without a Client

To test the `sayHello` method of `HelloService`, follow these steps.

1 **Open the web service test interface by typing the following URL in a web browser:**

`http://localhost:8080/helloservice/HelloService?Tester`

2 **Under Methods, type a name as the parameter to the `sayHello` method.**

3 Click the `sayHello` button.

This takes you to the `sayHello` Method invocation page.

Under `Method returned`, you'll see the response from the endpoint.

A Simple JAX-WS Application Client

The `HelloAppClient` class is a stand-alone application client that accesses the `sayHello` method of `HelloService`. This call is made through a port, a local object that acts as a proxy for the remote service. The port is created at development time by the `wsimport` task, which generates JAX-WS portable artifacts based on a WSDL file.

Coding the Application Client

When invoking the remote methods on the port, the client performs these steps:

1. Uses the generated `helloservice.endpoint.HelloService` class, which represents the service at the URI of the deployed service's WSDL file:

   ```
   import helloservice.endpoint.HelloService;
   import javax.xml.ws.WebServiceRef;

   public class HelloAppClient {
       @WebServiceRef(wsdlLocation =
         "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
       private static HelloService service;
   ```
2. Retrieves a proxy to the service, also known as a port, by invoking `getHelloPort` on the service:

   ```
   helloservice.endpoint.Hello port = service.getHelloPort();
   ```

   The port implements the SEI defined by the service.
3. Invokes the port's `sayHello` method, passing a string to the service:

   ```
   return port.sayHello(arg0);
   ```

Here is the full source of `HelloAppClient`, which is located in the following directory:

tut-install`/examples/jaxws/appclient/src/appclient/`

```
package appclient;

import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
      "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private static HelloService service;
```

```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
       System.out.println(sayHello("world"));
    }

    private static String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

Running the Application Client

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `appclient` application. To build the client, you must first have deployed `helloservice`, as described in "Building, Packaging, and Deploying the Service" on page 370.

▼ To Run the Application Client Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

   tut-install`/examples/jaxws/`
3. **Select the `appclient` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `appclient` project and select Run.**

   You will see the output of the application client in the Output pane.

▼ To Run the Application Client Using Ant

1. **In a terminal window, go to:**

   tut-install`/examples/jaxws/appclient/`
2. **Type the following command:**

   ```
   ant
   ```

   This command calls the `default` target, which runs the `wsimport` task and builds and packages the application into a JAR file, `appclient.jar`, located in the `dist` directory.

3 Type the following command:

```
ant getclient
```

This command deploys the `appclient.jar` file and retrieves the client stubs.

4 To run the client, type the following command:

```
ant run
```

A Simple JAX-WS Web Client

`HelloServlet` is a servlet that, like the Java client, calls the `sayHello` method of the web service. Like the application client, it makes this call through a port.

Coding the Servlet

To invoke the method on the port, the client performs these steps:

1. Imports the `HelloService` endpoint and the `WebServiceRef` annotation:

   ```
   import helloservice.endpoint.HelloService;
   ...
   import javax.xml.ws.WebServiceRef;
   ```
2. Defines a reference to the web service by specifying the WSDL location:

   ```
   @WebServiceRef(wsdlLocation =
     "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
   ```
3. Declares the web service, then defines a private method that calls the `sayHello` method on the port:

   ```
   private HelloService service;
   ...
   private String sayHello(java.lang.String arg0) {
       helloservice.endpoint.Hello port = service.getHelloPort();
       return port.sayHello(arg0);
   }
   ```
4. In the servlet, calls this private method:

   ```
   out.println("<p>" + sayHello("world") + "</p>");
   ```

The significant parts of the `HelloServlet` code follow. The code is located in the *tut-install*/`examples/jaxws/src/java/webclient/` directory.

```
package webclient;

import helloservice.endpoint.HelloService;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;

@WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})
public class HelloServlet extends HttpServlet {
    @WebServiceRef(wsdlLocation =
      "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private HelloService service;

    /**
     * Processes requests for both HTTP <code>GET</code>
     *   and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {

            out.println("<html lang=\"en\">");
            out.println("<head>");
            out.println("<title>Servlet HelloServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HelloServlet at " +
                request.getContextPath () + "</h1>");
            out.println("<p>" + sayHello("world") + "</p>");
            out.println("</body>");
            out.println("</html>");

        } finally {
            out.close();
        }
    }

    // doGet and doPost methods, which call processRequest, and
    //   getServletInfo method

    private String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

Running the Web Client

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `webclient` application. To build the client, you must first have deployed `helloservice`, as described in "Building, Packaging, and Deploying the Service" on page 370.

▼ To Run the Web Client Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

   tut-install`/examples/jaxws/`
3. **Select the `webclient` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `webclient` project and select Deploy.**

   This task runs the `wsimport` tasks, builds and packages the application into a WAR file, `webclient.war`, located in the `dist` directory, and deploys it to the server.
7. **In a web browser, navigate to the following URL:**

   `http://localhost:8080/webclient/HelloServlet`

   The output of the `sayHello` method appears in the window.

▼ To Run the Web Client Using Ant

1. **In a terminal window, go to:**

   tut-install`/examples/jaxws/webclient/`
2. **Type the following command:**

   ```
   ant
   ```

   This command calls the `default` target, which runs the `wsimport` tasks, then builds and packages the application into a WAR file, `webclient.war`, located in the `dist` directory.
3. **Type the following command:**

   ```
   ant deploy
   ```

   This task deploys the WAR file to the server.
4. **In a web browser, navigate to the following URL:**

   `http://localhost:8080/webclient/HelloServlet`

   The output of the `sayHello` method appears in the window.

Types Supported by JAX-WS

JAX-WS delegates the mapping of Java programming language types to and from XML definitions to JAXB. Application developers don't need to know the details of these mappings but should be aware that not every class in the Java language can be used as a method parameter or return type in JAX-WS.

The following sections explain the default schema-to-Java and Java-to-schema data type bindings.

Schema-to-Java Mapping

The Java language provides a richer set of data type than XML schema. Table 19–1 lists the mapping of XML data types to Java data types in JAXB.

TABLE 19–1 JAXB Mapping of XML Schema Built-in Data Types

XML Schema Type	Java Data Type
`xsd:string`	`java.lang.String`
`xsd:integer`	`java.math.BigInteger`
`xsd:int`	`int`
`xsd.long`	`long`
`xsd:short`	`short`
`xsd:decimal`	`java.math.BigDecimal`
`xsd:float`	`float`
`xsd:double`	`double`
`xsd:boolean`	`boolean`
`xsd:byte`	`byte`
`xsd:QName`	`javax.xml.namespace.QName`
`xsd:dateTime`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:base64Binary`	`byte[]`
`xsd:hexBinary`	`byte[]`
`xsd:unsignedInt`	`long`
`xsd:unsignedShort`	`int`
`xsd:unsignedByte`	`short`

TABLE 19–1 JAXB Mapping of XML Schema Built-in Data Types *(Continued)*

XML Schema Type	Java Data Type
`xsd:time`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:date`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:g`	`javax.xml.datatype.XMLGregorianCalendar`
`xsd:anySimpleType`	`java.lang.Object`
`xsd:anySimpleType`	`java.lang.String`
`xsd:duration`	`javax.xml.datatype.Duration`
`xsd:NOTATION`	`javax.xml.namespace.QName`

Java-to-Schema Mapping

Table 19–2 shows the default mapping of Java classes to XML data types.

TABLE 19–2 JAXB Mapping of XML Data Types to Java Classes

Java Class	XML Data Type
`java.lang.String`	`xs:string`
`java.math.BigInteger`	`xs:integer`
`java.math.BigDecimal`	`xs:decimal`
`java.util.Calendar`	`xs:dateTime`
`java.util.Date`	`xs:dateTime`
`javax.xml.namespace.QName`	`xs:QName`
`java.net.URI`	`xs:string`
`javax.xml.datatype.XMLGregorianCalendar`	`xs:anySimpleType`
`javax.xml.datatype.Duration`	`xs:duration`
`java.lang.Object`	`xs:anyType`
`java.awt.Image`	`xs:base64Binary`
`javax.activation.DataHandler`	`xs:base64Binary`
`javax.xml.transform.Source`	`xs:base64Binary`
`java.util.UUID`	`xs:string`

Web Services Interoperability and JAX-WS

JAX-WS supports the Web Services Interoperability (WS-I) Basic Profile Version 1.1. The WS-I Basic Profile is a document that clarifies the SOAP 1.1 and WSDL 1.1 specifications to promote SOAP interoperability. For links related to WS-I, see "Further Information about JAX-WS" on page 379.

To support WS-I Basic Profile Version 1.1, the JAX-WS runtime supports doc/literal and rpc/literal encodings for services, static ports, dynamic proxies, and the Dynamic Invocation Interface (DII).

Further Information about JAX-WS

For more information about JAX-WS and related technologies, see

- Java API for XML Web Services 2.2 specification:

  `http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index4.html`
- JAX-WS home:

  `http://jax-ws.java.net/`
- Simple Object Access Protocol (SOAP) 1.2 W3C Note:

  `http://www.w3.org/TR/soap/`
- Web Services Description Language (WSDL) 1.1 W3C Note:

  `http://www.w3.org/TR/wsdl`
- WS-I Basic Profile 1.1:

  `http://www.ws-i.org`