

Building Web Services with JAX-WS

JAX-WS stands for 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 RPC-oriented web services. In JAX-WS, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing remote procedure calls 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 remote procedures 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 World Wide Web Consortium (W3C): HTTP, SOAP, and the Web Service Description Language (WSDL). WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

Setting the Port

Several files in the JAX-WS examples depend on the port that you specified when you installed the Application Server. The tutorial examples assume that the server runs on the default port, 8080. If you have changed the port, you must update the port number in the following files before building and running the JAX-WS examples:

- `<INSTALL>/javaeetutorial5/examples/jaxws/simpleclient/HelloClient.java`

Creating a Simple Web Service and Client with JAX-WS

This section shows how to build and deploy a simple web service and client. The source code for the service is in `<INSTALL>/javaeetutorial5/examples/jaxws/helloservice/` and the client is in `<INSTALL>/javaeetutorial5/examples/jaxws/simpleclient/`.

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

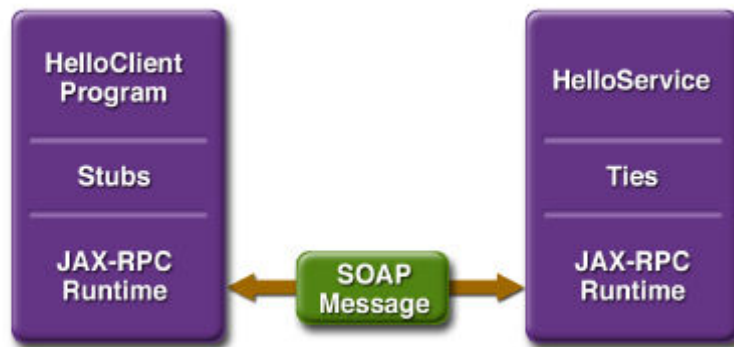


Figure 1-1 Communication Between a JAX-WS Web Service and a Client

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

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

You use the endpoint implementation class and the `wsgen` tool to generate the web service artifacts and the stubs that connect a web service client to the JAXWS runtime. For reference documentation on `wsgen`, see the Application Server man pages at <http://docs.sun.com/db/doc/817-6092>.

Together, the `wsgen` tool and the Application Server provide the Application Server's implementation of JAX-WS.

These are the basic steps for creating the web service and client:

1. Code the implementation class.
2. Compile the implementation class.
3. Use `wsgen` to generate the artifacts required to deploy the service.
4. Package the files into a WAR file.
5. Deploy the WAR file. The tie classes (which are used to communicate with clients) are generated by the Application Server during deployment.
6. Code the client class.
7. Use `wsimport` to generate and compile the stub files.
8. Compile the client class.
9. Run the client.

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 `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 not 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 Default Data Type Bindings (page 6).
- 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 `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. `@WebMethod` exposes the annotated method to web service clients. `sayHello` returns a greeting to the client, using the name passed to `sayHello` to compose the greeting. The implementation class also must define a default, public, no-argument constructor.

```
package helloservice.endpoint;
import javax.jws.WebService;
@WebService()
public class Hello {
    private String message = new String("Hello, ");
    public void Hello() {}
    @WebMethod()
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Building the Service

To build `HelloService`, in a terminal window go to the `<INSTALL>/javaee/tutorial5/examples/jaxws/helloservice/` directory and type the following:

asant build

The build task command executes these asant subtasks:

- `compile-service`

The compile-service Task

This asant task compiles Hello.java, writing the class files to the build subdirectory. It then calls the wsgen tool to generate JAX-WS portable artifacts used by the web service. The equivalent command-line command is as follows:

```
wsgen -d build -s build -classpath build  
helloservice.endpoint.Hello
```

The -d flag specifies the output location of generated class files. The -s flag specifies the output location of generated source files. The -classpath flag specifies the location of the input files, in this case the endpoint implementation class, helloservice.endpoint.Hello.

Packaging and Deploying the Service

You package and deploy the service using asant. Upon deployment, the Application Server and the JAX-WS runtime generate any additional artifacts required for web service invocation, including the WSDL file.

Packaging and Deploying the Service with asant

To package and deploy the helloservice example, follow these steps:

1. In a terminal window, go to
<INSTALL>/javaeetutorial5/examples/jaxws/helloservice/.
2. Run asant create-war.
3. Make sure the Application Server is started.
4. Set your admin username and password in
<INSTALL>/javaeetutorial5/examples/common/build.properties.
5. Run asant deploy.

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

Undeploying the Service

At this point in the tutorial, do not undeploy the service. When you are finished with this example, you can undeploy the service by typing this command: asant undeploy

Testing the Service Without a Client

The Application Server Admin Console allows you to test the methods of a web service endpoint. To test the sayHello method of HelloService, do the following:

1. Open the Admin Console by opening the following URL in a web browser:
<http://localhost:4848/>
2. Enter the admin username and password to log in to the Admin Console.
3. Click Web Services in the left pane of the Admin Console.
4. Click Hello.
5. Click Test.
6. Under Methods, enter a name as the parameter to the sayHello method.
7. Click the sayHello button. This will take you to the sayHello Method invocation page.
8. Under Method returned, you'll see the response from the endpoint.

A Simple JAX-WS Client

HelloClient is a stand-alone Java program that accesses the sayHello method of HelloService. It makes this call through a stub, a local object that acts as a proxy for the remote service. The stub is created at development time by the wsimport tool, which generates JAX-WS portable artifacts based on a WSDL file.

Coding the Client

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

1. Uses the `javax.xml.ws.WebServiceRef` annotation to declare a reference to a web service. `WebServiceRef` uses the `wsdlLocation` element to specify the URI of the deployed service's WSDL file.

```
@WebServiceRef(wsdlLocation="http://localhost:8080/hello/hello?wsdl")
static HelloService service;
```

2. Retrieves a proxy to the service, also known as a port, by invoking `getHelloPort` on the service.

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

The port implements the SEI defined by the service.

3. Invokes the port's `sayHello` method, passing to the service a name.

```
String response = port.sayHello(name);
```

Here's the full source of `HelloClient`, located in the

<INSTALL>/javaee/tutorial5/examples/jaxws/simpleclient/src/ directory.

```
package simpleclient;
import javax.xml.ws.WebServiceRef;
import hello.service.endpoint.HelloService;
import hello.service.endpoint.Hello;
public class HelloClient {
    @WebServiceRef(wsdlLocation="http://localhost:8080/hello/hello?wsdl")
    static HelloService service;
    public static void main(String[] args) {
        try {
            HelloClient client = new HelloClient();
            client.doTest(args);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void doTest(String[] args) {
        try {
            System.out.println("Retrieving the port from
the following service: " + service);
            Hello port = service.getHelloPort();
            System.out.println("Invoking the sayHello operation
on the port.");
            String name;
            if (args.length > 0) {
                name = args[0];
            } else {
                name = "No Name";
            }
            String response = port.sayHello(name);
            System.out.println(response);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Building and Running the Client

To build the client, you must first have deployed `HelloServiceApp`, as described in "Packaging and Deploying the Service with `asant` (page xx)." Then navigate to

<JAVA_EE_HOME>/examples/jaxws/simpleclient/ and do the following:

`asant build`

The run the client, do the following:

asant run

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 they should be aware that not every class in the Java language can be used as a method parameter or return type in JAX-WS. For information on which types are supported by JAXB, see Default Data Type Bindings (page 6).

Web Services Interoperability and JAXWS

JAX-WS 2.0 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 in order to promote SOAP interoperability. For links related to WS-I, see Further Information (page xxiv).

To support WS-I Basic Profile Version 1.1, JAX-WS has the following features:

The JAX-WS runtime supports doc/literal and rpc/literal encodings for services, static stubs, dynamic proxies, and DII.

Further Information

For more information about JAX-WS and related technologies, refer to the following:

- Java API for XML Web Services 2.0 specification <https://jax-ws.dev.java.net/spec-download.html>
- JAX-WS home <https://jax-ws.dev.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>