

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/hello-service-war/` directory, and the clients are in the `tut-install/examples/jaxws/hello-appclient/` and `tut-install/examples/jaxws/hello-webclient/` directories.

Figure 31-1 illustrates how JAX-WS technology manages communication between a web service and a client.

Figure 31-1 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.

Basic Steps for Creating a Web Service and Client

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 GlassFish Server during deployment.
5. Code the client class.
6. Use the `wsimport` Maven goal 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 in [Types Supported by JAX-WS](#).
- 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 javaeetutorial.helloservice;

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

@WebService
public class Hello {
    private final String message = "Hello, ";
```

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

Building, Packaging, and Deploying the Service

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

The following topics are addressed here:

- [To Build, Package, and Deploy the Service Using NetBeans IDE](#)
- [To Build, Package, and Deploy the Service Using Maven](#)

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

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](#)).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

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

4. Select the `helloservice-war` folder.
5. Click Open Project.
6. In the Projects tab, right-click the `helloservice-war` project and select Run.

This command builds and packages the application into a WAR file, `helloservice-war.war`, located in `tut-install/examples/jaxws/helloservice-war/target/`, and deploys this WAR file to your GlassFish Server instance. It also opens the web service test interface at the URL shown in [To Test the Service without a Client](#).

Next Steps

You can view the WSDL file of the deployed service by requesting the URL <http://localhost:8080/helloservice-war/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 Maven

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](#)).
2. In a terminal window, go to:

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

3. Enter the following command:

```
mvn install
```

This command builds and packages the application into a WAR file, `helloservice-war.war`, located in the `target` directory, and then deploys the WAR to GlassFish Server.

Next Steps

You can view the WSDL file of the deployed service by requesting the URL <http://localhost:8080/helloservice-war/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.

The following topics are addressed here:

- [To Test the Service without a Client](#)

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 entering the following URL in a web browser:

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

2. Under Methods, enter a name as the parameter to the `sayHello` method.
3. Click `sayHello`.

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` Maven goal, which generates JAX-WS portable artifacts based on a WSDL file.

The following topics are addressed here:

- [Coding the Application Client](#)
- [Running the Application Client](#)

Coding the Application Client

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

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

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

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/helloservice-war/HelloService?WSDL")
    private static HelloService service;
```

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

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

The port implements the SEI defined by the service.

3. It invokes the port's `sayHello` method, passing a string to the service:

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

Here is the full source of `HelloAppClient.java`, which is located in the `tut-install/examples/jaxws/hello-appclient/src/main/java/javaeetutorial/hello/appclient/` directory:

```
package javaeetutorial.hello.appclient;

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

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/helloservice-war/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) {
        javaeetutorial.helloservice.endpoint.Hello port =
            service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

```
}  
}
```

Running the Application Client

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `hello-appclient` application. To build the client, you must first have deployed `helloservice-war`, as described in [Building, Packaging, and Deploying the Service](#).

The following topics are addressed here:

- [To Run the Application Client Using NetBeans IDE](#)

To Run the Application Client Using NetBeans IDE

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

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

3. Select the `hello-appclient` folder.
4. Click Open Project.
5. In the Projects tab, right-click the `hello-appclient` project and select Build.

This command runs the `wsimport` goal, then builds, packages, and runs the client. You will see the output of the application client in the `hello-appclient` output tab:

```
--- exec-maven-plugin:1.2.1:exec (run-appclient) @ hello-appclient ---  
Hello, world.
```

To Run the Application Client Using Maven

1. In a terminal window, go to:

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

2. Enter the following command:

```
mvn install
```

This command runs the `wsimport` goal, then builds, packages, and runs the client. The application client output looks like this:

```
--- exec-maven-plugin:1.2.1:exec (run-appclient) @ hello-appclient ---  
Hello, world.
```

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.

The following topics are addressed here:

- [Coding the Servlet](#)
- [Running the Web Client](#)

Coding the Servlet

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

1. It imports the `HelloService` endpoint and the `WebServiceRef` annotation:

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

2. It defines a reference to the web service by specifying the WSDL location:

```
@WebServiceRef(wsdlLocation =
    "http://localhost:8080/helloservice-war/HelloService?WSDL")
```

3. It 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) {
    javaeetutorial.helloservice.endpoint.Hello port =
        service.getHelloPort();
    return port.sayHello(arg0);
}
```

4. In the servlet, it 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/hello-webclient/src/java/javaeetutorial/hello/webclient/` directory.

```
package javaeetutorial.hello.webclient;

import javaeetutorial.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 =
        "http://localhost:8080/hello-service-war/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");
        try (PrintWriter out = response.getWriter()) {

            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>");

        }
    }

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

    private String sayHello(java.lang.String arg0) {
        javaee tutorial.hello-service.endpoint.Hello port =
            service.getHelloPort();
    }
}

```

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

Running the Web Client

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `hello-webclient` application. To build the client, you must first have deployed `helloservice-war`, as described in [Building, Packaging, and Deploying the Service](#).

The following topics are addressed here:

- [To Run the Web Client Using NetBeans IDE](#)
- [To Run the Web Client Using Maven](#)

To Run the Web Client Using NetBeans IDE

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

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

3. Select the `hello-webclient` folder.
4. Click Open Project.
5. In the Projects tab, right-click the `hello-webclient` project and select Build.

This task runs the `wsimport` goal, builds and packages the application into a WAR file, `hello-webclient.war`, located in the `target` directory, and deploys it to GlassFish Server.

6. In a web browser, enter the following URL:

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

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

To Run the Web Client Using Maven

1. In a terminal window, go to:

```
tut-install/examples/jaxws/hello-webclient/
```

2. Enter the following command:

```
mvn install
```

This command runs the `wsimport` goal, then build and packages the application into a WAR file, `hello-webclient.war`, located in the `target` directory. The WAR file is then deployed to GlassFish Server.

3. In a web browser, enter the following URL:

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

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

[Previous](#)[Next](#)[Contents](#)

ORACLE Copyright © 2017, Oracle and/or its affiliates. All rights reserved.