Creating a Simple Web Service and Client with JAX-RPC

This section shows how to build and deploy a simple web service and client. A later section, Web Service Clients, provides examples of additional JAX-RPC clients that access the service. The source code for the service is in `<INSTALL>/j2eetutorial14/examples/jaxrpc/helloservice/` and the client is in `<INSTALL>/j2eetutorial14/examples/jaxrpc/staticstub/`.

Figure 8-1 illustrates how JAX-RPC technology manages communication between a web service and client.

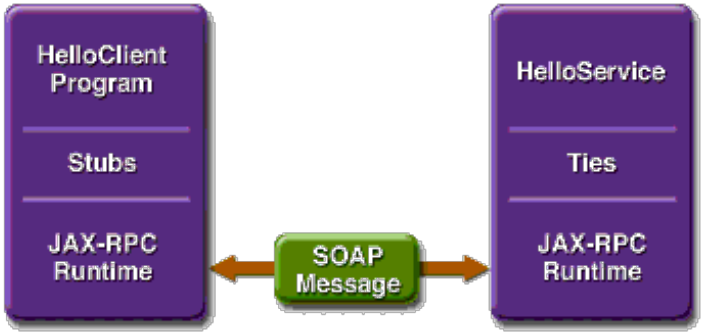

Figure 8-1 Communication Between a JAX-RPC Web Service and a Client

The starting point for developing a JAX-RPC web service is the service endpoint interface. A *service endpoint interface* (SEI) is a Java interface that declares the methods that a client can invoke on the service.

You use the SEI, the `wscompile` tool, and two configuration files to generate the WSDL specification of the web service and the stubs that connect a web service client to the JAX-RPC runtime. For reference documentation on `wscompile`, see the Application Server man pages at `http://docs.sun.com/db/doc/817-6092`.

Together, the `wscompile` tool, the `deploytool` utility, and the Application Server provide the Application Server's implementation of JAX-RPC.

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

1. Code the SEI and implementation class and interface configuration file.
2. Compile the SEI and implementation class.
3. Use `wscompile` to generate the files required to deploy the service.
4. Use `deploytool` to 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 and WSDL configuration file.
7. Use `wscompile` 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.

Coding the Service Endpoint Interface and Implementation Class

In this example, the service endpoint interface declares a single method named `sayHello`. This method returns a string that is the concatenation of the string `Hello` with the method parameter.

A service endpoint interface must conform to a few rules:

- It extends the `java.rmi.Remote` interface.
- It must not have constant declarations, such as `public final static`.
- The methods must throw the `java.rmi.RemoteException` or one of its subclasses. (The methods may also throw service-specific exceptions.)
- Method parameters and return types must be supported JAX-RPC types (see Types Supported by JAX-RPC).

In this example, the service endpoint interface is named `HelloIF`:

```
package helloservice;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloIF extends Remote {
    public String sayHello(String s) throws RemoteException;
}
```

In addition to the interface, you'll need the class that implements the interface. In this example, the implementation class is called `HelloImpl`:

```
package helloservice;

public class HelloImpl implements HelloIF {

    public String message ="Hello";

    public String sayHello(String s) {
        return message + s;
    }
}
```

Building the Service

To build `MyHelloService`, in a terminal window go to the `<INSTALL>/j2eetutorial14/examples/jaxrpc/helloservice/` directory and type the following:

```
asant build
```

The `build` task command executes these `asant` subtasks:

- `compile-service`
- `generate-wsdl`

The compile-service Task

This `asant` task compiles `HelloIF.java` and `HelloImpl.java`, writing the class files to the `build` subdirectory.

The generate-wsdl Task

The `generate-wsdl` task runs `wscompile`, which creates the WSDL and mapping files. The WSDL file describes the web service and is used to generate the client stubs in Static Stub Client. The mapping file contains information that correlates the mapping between the Java interfaces and the WSDL definition. It is meant to be portable so that any J2EE-compliant deployment tool can use this information, along with the WSDL file and the Java interfaces, to generate stubs and ties for the deployed web services.

The files created in this example are `MyHelloService.wsdl` and `mapping.xml`. The `generate-wsdl` task runs `wscompile` with the following arguments:

```
wscompile -define -mapping build/mapping.xml -d build -nd build
-classpath build config-interface.xml
```

The `-classpath` flag instructs `wscompile` to read the SEI in the `build` directory, and the `-define` flag instructs `wscompile` to create WSDL and mapping files. The `-mapping` flag specifies the mapping file name. The `-d` and `-nd` flags tell the tool to write class and WSDL files to the `build` subdirectory.

The `wscompile` tool reads an interface configuration file that specifies information about the SEI. In this example, the configuration file is named `config-interface.xml` and contains the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service
      name="MyHelloService"
      targetNamespace="urn:Foo"
      typeNamespace="urn:Foo"
      packageName="helloservice">
      <interface name="helloservice.HelloIF"/>
  </service>
</configuration>
```

This configuration file tells `wscompile` to create a WSDL file named `MyHello Service.wsdl` with the following information:

- The service name is `MyHelloService`.
- The WSDL target and type namespace is `urn:Foo`. The choice for what to use for the namespaces is up to you. The role of the namespaces is similar to the use of Java package names--to distinguish names that might otherwise conflict. For example, a company can decide that all its Java code should be in the package `com.wombat.*`. Similarly, it can also decide to use the namespace `http://wombat.com`.
- The SEI is `helloservice.HelloIF`.

The `packageName` attribute instructs `wscompile` to put the service classes into the `helloservice` package.

Packaging and Deploying the Service

You can package and deploy the service using either `deploytool` or `asant`.

Packaging and Deploying the Service with deploytool

Behind the scenes, a JAX-RPC web service is implemented as a servlet. Because a servlet is a web component, you run the New Web Component wizard of the `deploytool` utility to package the service. During this process the wizard performs the following tasks:

- Creates the web application deployment descriptor
- Creates a WAR file
- Adds the deployment descriptor and service files to the WAR file

To start the New Web Component wizard, select File → New → Web Component. The wizard displays the following dialog boxes.

1. Introduction dialog box
    a. Read the explanatory text for an overview of the wizard's features.
    b. Click Next.
2. WAR File dialog box
    a. Select the button labeled Create New Stand-Alone WAR Module.
    b. In the WAR File field, click Browse and navigate to `<INSTALL>/j2eetutorial14/examples/jaxrpc/helloservice/.`
    c. In the File Name field, enter `MyHelloService`.
    d. Click Create Module File.
    e. Click Edit Contents.
    f. In the tree under Available Files, locate the `<INSTALL>/j2eetutorial14/examples/jaxrpc/helloservice/` directory.
    g. Select the `build` subdirectory.
    h. Click Add.
    i. Click OK.
    j. In the Context Root field, enter `/hello-jaxrpc`.
    k. Click Next.
3. Choose Component Type dialog box
    a. Select the Web Services Endpoint button.
    b. Click Next.
4. Choose Service dialog box
    a. In the WSDL File combo box, select `WEB-INF/wsdl/MyHelloService.wsdl`.
    b. In the Mapping File combo box, select `build/mapping.xml`.
    c. Click Next.
5. Component General Properties dialog box

a. In the Service Endpoint Implementation combo box, select `helloservice.HelloImpl`.

b. Click Next.

6. Web Service Endpoint dialog box

a. In the Service Endpoint Interface combo box, select `helloservice.HelloIF`.

b. In the Namespace combo box, select `urn:Foo`.

c. In the Local Part combo box, select `HelloIFPort`.

d. The `deploytool` utility will enter a default Endpoint Address URI `HelloImpl` in this dialog. This endpoint address *must* be updated in the next section.

e. Click Next.

f. Click Finish.

Specifying the Endpoint Address

To access `MyHelloService`, the tutorial clients will specify this service endpoint address URI:

```
http://localhost:8080/hello-jaxrpc/hello
```

The `/hello-jaxrpc` string is the context root of the servlet that implements `MyHelloService`. The `/hello` string is the servlet alias. You already set the context root in Packaging and Deploying the Service with deploytool above. To specify the endpoint address, set the alias as follows:

1. In `deploytool`, select `MyHelloService` in the tree.
2. In the tree, select `HelloImpl`.
3. Select the Aliases tab.
4. In the Component Aliases table, add `/hello`.
5. In the Endpoint tab, select `hello` for the Endpoint Address in the Sun-specific Settings frame.
6. Select File → Save.

Deploying the Service

In `deploytool`, perform these steps:

1. In the tree, select `MyHelloService`.
2. Select Tools → Deploy.

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

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>/j2eetutorial14/examples/jaxrpc/helloservice/`.
2. Run `asant create-war`.
3. Make sure the Application Server is started.
4. Set your admin username and password in `<INSTALL>/j2eetutorial14/examples/common/build.properties`.
5. Run `asant deploy-war`.

You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/hello-jaxrpc/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
```

Static Stub Client

`HelloClient` is a stand-alone program that calls the `sayHello` method of the `MyHelloService`. It makes this call through a *stub*, a local object that acts as a proxy for the remote service. Because the stub is created by `wscompile` at development time (as opposed to runtime), it is usually called a *static stub*.

Coding the Static Stub Client

Before it can invoke the remote methods on the stub, the client performs these steps:

1. Creates a `Stub` object:

   ```
   (Stub)(new MyHelloService_Impl().getHelloIFPort())
   ```

   The code in this method is implementation-specific because it relies on a `MyHelloService_Impl` object, which is not defined in the specifications. The `MyHelloService_Impl` class will be generated by `wscompile` in the following section.

2. Sets the endpoint address that the stub uses to access the service:

   ```
   stub._setProperty
   (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
   ```

   At runtime, the endpoint address is passed to `HelloClient` in `args[0]` as a command-line parameter, which `asant` gets from the `endpoint.address` property in the `build.properties` file. This address must match the one you set for the service in Specifying the Endpoint Address.

3. Casts `stub` to the service endpoint interface, `HelloIF`:

   ```
   HelloIF hello = (HelloIF)stub;
   ```

Here is the full source code listing for the `HelloClient.java` file, which is located in the directory

<INSTALL>/j2eetutorial14/examples/jaxrpc/staticstub/src/:

```
package staticstub;

import javax.xml.rpc.Stub;

public class HelloClient {

    private String endpointAddress;

    public static void main(String[] args) {

        System.out.println("Endpoint address = " + args[0]);
        try {
            Stub stub = createProxy();
            stub._setProperty
              (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
               args[0]);
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello("Duke!"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private static Stub createProxy() {
        // Note: MyHelloService_Impl is implementation-specific.
        return
        (Stub) (new MyHelloService_Impl().getHelloIFPort());
    }
}
```

Building and Running the Static Stub Client

To build and package the client, go to the <INSTALL>/j2eetutorial14/examples/jaxrpc/staticstub/ directory and type the following:

```
asant build
```

The build task invokes three asant subtasks:

- generate-stubs
- compile-client
- package-client

The generate-stubs task runs the wscompile tool with the following arguments:

```
wscompile -gen:client -d build -classpath build config-wsdl.xml
```

This wscompile command reads the MyHelloService.wsdl file that was generated in Building the Service. The command generates files based on the information in the WSDL file and the command-line flags.

The -gen:client flag instructs wscompile to generate the stubs, other runtime files such as serializers, and

value types. The `-d` flag tells the tool to write the generated output to the `build/staticstub` subdirectory.

The `wscompile` tool reads a WSDL configuration file that specifies the location of the WSDL file. In this example, the configuration file is named `config-wsdl.xml`, and it contains the following:

```
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="http://localhost:8080/hello-
jaxrpc/hello?WSDL" packageName="staticstub"/>
</configuration>
```

The `packageName` attribute specifies the Java package for the generated stubs. Notice that the location of the WSDL file is specified as a URL. This causes the `wscompile` command to request the WSDL file from the web service, and this means that the web service must be correctly deployed and running in order for the command to succeed. If the web service is not running or if the port at which the service is deployed is different from the port in the configuration file, the command will fail.

The c`ompile-client` task compiles `src/HelloClient.java` and writes the class file to the `build` subdirectory.

The `package-client` task packages the files created by the ge`nerate-stubs` and `compile-client` tasks into the `dist/client.jar` file. Except for the `HelloClient.class`, all the files in `client.jar` were created by `wscompile`. Note that `wscompile` generated the `HelloIF.class` based on the information it read from the `MyHelloService.wsdl` file.

To run the client, type the following:

```
asant run
```

This task invokes the web service client, passing the string `Duke` for the web service method parameter. When you run this task, you should get the following output:

```
Hello Duke!
```