Invoking web services with Java clients

A look at web services clients in the J2SE and J2EE environment

Bertrand Portier

November 04, 2003

In this article, IBM developer Bertrand Portier describes the different types of Java web services clients and explains how to write portable, vendor independent code. There are two families of web services clients in the Java world: unmanaged and J2EE container-managed clients. The article starts by briefly describing the web services invocation process and the web services standards for Java environments. The two families of Java web services clients are then described, including their similarities and differences for the two steps they need to perform: service lookup and access.

The power of web services is interoperability. Thanks to industry collaboration on the web services technologies (SOAP, WSDL, UDDI) and, more specifically, thanks to the work of the Web Services Interoperability organization (WS-I.org), a web service can interact with any other web service, no matter which platform the web service is developed and run on (such as Microsoft .NET or IBM WebSphere). A web service client can be of many types, such as another web service, a client written in a scripting language, a C# client, a Java client, etc. This article focuses on Java clients, which can be used to access any web service conforming to web services specifications (not only Java web services). Throughout the article, you will see the web service client code required to invoke the same web service using different lookup and access methods. The example used is the "Hello" web service, which provides a "getGreeting" operation. This operation takes one String parameter (for example, Jane) and returns a greeting: "Hello Jane!".

Web services roles

This section will describe the process of invoking web services. A web service provider describes a web service in a Web Services Description Language (WSDL) document. The web service is typically published to a Universal Description, Discovery and Integration (UDDI) registry. A web service requester finds the web service in the UDDI registry, binds to the web service, and invokes it. The web services roles are shown in Figure 1. This article will focus on the horizontal arrow (bind) from the service requester to the service provider. This article will call the requester a client; it can also be referred to as a consumer.
**Figure 1. Web services roles**

![Web services roles diagram]

**Web services standards for Java**

The development of Java technology standards occurs through Java Specification Requests (JSRs) being submitted to the Java Community Process (JCP). Two JSRs cover the Java web services architecture:

1. JSR 101: Java API for XML based RPC (JAX-RPC)
2. JSR 109: Implementing Enterprise Web services.

Both specifications provide conformance and interoperability requirements for vendors' implementations.

**JAX-RPC**

JAX-RPC defines a simple and easy-to-use Java Application Programming Interface (API) for XML-based Remote Procedure Calls (RPC) and the Java to XML and XML to Java mapping:

- WSDL to Java and Java to WSDL mappings: For example, a WSDL port type is mapped to a Java Service Endpoint Interface (SEI).
- XML data type to Java data type and Java data type to XML data type mappings, including simple types, complex types, and arrays.

In addition to XML mappings, JAX-RPC also defines the client-side programming model and API, which I will cover in more details in later sections. JAX-RPC 1.1 adds interoperability requirements based on the Web Services Interoperability organization (WS-I) Basic Profile version 1.0.

**JSR 109**

JSR 109 specifies the web services programming model and architecture for the Java 2 Enterprise Edition (J2EE) environment. JSR 109 builds on SOAP 1.1 and WSDL 1.1 to cover the use of JAX-RPC in a J2EE environment (Figure 2). It also defines a deployment model to J2EE application servers. JSR 109's client-side programming model, which I will cover in sections below, is conformant to JAX-RPC.
JAX-RPC 1.1 and JSR 109 are part of J2EE 1.4.

**Service lookup**

There are two families of clients, which differ in the way the code is written, packaged, and invoked:

1. Unmanaged clients
2. J2EE container-managed clients.

Here, unmanaged means not J2EE container-managed. These are Java 2 Standard Edition (J2SE) clients and are invoked with a simple `java` command. For unmanaged clients, the service lookup is through the JAX-RPC `ServiceFactory`, a factory for the creation of instances of services access points. For J2EE container-managed clients, service lookup is through JNDI lookup.

**JAX-RPC ServiceFactory**

The JAX-RPC `ServiceFactory` is an abstract class which acts as a factory for instantiating JAX-RPC services. It is vendor independent and lets you write portable code. The `ServiceFactory` is instantiated and used as follows:

```java
javax.xml.rpc.Service service = ServiceFactory.newInstance().createService(...);
```

You need to pass the fully qualified name of the web service, that is namespace plus service name, to the `createService()` method and optionally the URL of the WSDL document describing the web service you want to look up. The steps are as follows:

1. Optionally, specify the WSDL URL.
2. Specify the web service's fully qualified name.
3. Invoke `ServiceFactory`'s `createService()` method.

The Service Interface obtained is then used to get a stub, dynamic proxy, or DII Call object, as described in the "Service access" section. Also covered in that section is the Dynamic Invocation Interceptor (DII) object.
Interface (DII). With this method, you do not need to know the WSDL URL of the web service you want to invoke, and you only specify the service name parameter to the createService() method. Listing 1 shows how to use the ServiceFactory to instantiate a JAX-RPC Service. QName is a javax.xml.namespace.QName.

**Listing 1. Using the JAX-RPC ServiceFactory to obtain a JAX-RPC Service**

```java
String wsdlURL = http://localhost:6080/HelloWebService/services/Hello?wsdl;";
String namespace = "http://Hello.com";
String serviceName = "HelloWebService";
QName serviceQN = new QName(namespace, serviceName);

ServiceFactory serviceFactory = ServiceFactory.newInstance();
/* The "new URL(wsdlURL)" parameter is optional */
Service service = serviceFactory.createService(new URL(wsdlURL), serviceQN);
```

There are vendor-specific alternatives to the JAX-RPC ServiceFactory. These are usually very easy to use (the client code is very simple to write) if you want to use a vendor's stub. However, such extensions are not standard and will probably not work on other vendor's J2EE implementations.

**JNDI lookup**

JSR 109: "JNDI lookup is the standard way to look up web services in a J2EE environment."

**JNDI service lookup**

J2EE container-managed clients are packaged into Enterprise Archives (.EAR) files and run from inside a J2EE container. In addition to the Java code, descriptors are also packaged into the archive. Several different types of J2EE container-managed clients are:

- Application client
- Container clients
- Web container clients: JavaBean or Servlet
- EJB container clients: EJB

JAX-RPC defines the programming model for unmanaged clients, whereas JSR 109, "Implementing Enterprise Web services", defines the programming model for J2EE container-managed clients. One of the goals of JSR 109 is that its client programming model is compatible with JAX-RPC. However, JSR 109 does not recommend the use of the JAX-RPC ServiceFactory. It recommends clients use Java Naming and Directory Interface (JNDI) instead to obtain a Service Interface. This is a two step process, also illustrated in Listing 2:

1. Instantiate a local JNDI Context.
2. Do a JNDI lookup for the web service name in this context.

**Listing 2. JNDI service lookup**

```java
Context ic = new InitialContext();
Service service = (Service) ctx.lookup("java:comp/env/service/HelloService");
```

The name of the web service, in this case java:comp/env/service/HelloService, is specified in the client application's deployment descriptor. JSR 109 recommends that all service reference
logical names be organized under the service sub-context. With the client environment context being Java:comp/env, you end up with:

```
  service name in context =
  client environment context + "service" subcontext + service name.
```

In this case, the service name in context is:

```
  java:comp/env/ + service/ + HelloService.
```

service subcontext + service name (for example, service/HelloService) is also called the logical service name and is declared in the web service client application's deployment descriptor.

The JNDI lookup returns a JAX-RPC Service Interface. The J2EE container makes sure an implementation of the generic JAX-RPC Service is bound at the location specified in the deployment descriptor. You can also cast the object returned by the lookup to the specific interface for your web service. This is shown on Listing 3, where the HelloService extends the generic JAX-RPC Service interface.

### Listing 3. Alternative JNDI lookup

```
  Context ic= new InitialContext();
  HelloServiceInterface service =
      (HelloServiceInterface) ic.lookup("java:comp/env/service/HelloService");
```

The Service Interface obtained is then used to get a static stub, dynamic proxy, or a DII Call object, as described in the "Service access" section below.

### Service access

In previous sections, you saw that a JAX-RPC ServiceFactory acts as a factory for JAX-RPC Services. Similarly, a JAX-RPC Service acts as a factory for proxies and stubs. Once you have instantiated a Service, there are three methods for accessing and invoking the web service:

1. Stub
2. Dynamic Proxy
3. Dynamic Invocation Interface (DII).

Stub and dynamic proxy methods use the Service Endpoint Interface (SEI). It is basically the Java representation of the web service operations described in the WSDL port type element. It is a Java interface defining methods used by the Java client to interact with the web service. The SEI is generated by a WSDL to Java mapping tool (such as Apache Axis' Java2WSDL or IBM WSDK's WSDL2Client).

### SEI

A Service Endpoint Interface (SEI) is the Java representation of a WSDL port type.
**Stub**

The stub method uses a platform-specific stub created before runtime during the WSDL to Java mapping stage. Because the stub is created before runtime, it is sometimes called a *static stub*. It is a Java class implementing the SEI. A WSDL to Java mapping tool generates the client-side artifacts needed; basically, the tool imports the WSDL service definition and creates the corresponding Java code. The artifacts include an SEI, a Stub, and optionally holders, serializers, deserializers, and utility classes. JAX-RPC recommends an instance of a stub to be bound to a specific protocol and transport, such as a SOAP binding stub. For the stub method, the steps to perform are:

1. Get a JAX-RPC Service.
2. Obtain a stub.
3. Invoke the web service's operations on the stub.

Steps 2 and 3 are shown in **Listing 4**. Note that it is also possible to use the JAX-RPC service's getPort method (described in the next section) to obtain a stub.

**Listing 4. Accessing a web service through a stub**

```java
Hello myStub = (Hello) service.getHello();
System.out.println(myStub.getGreeting("Jane"));
```

The advantage of this method is its simplicity. Basically, only two lines of code are required to access and invoke a web service's operation. However, you need to know the WSDL URL at development-time and run your WSDL to Java mapping tool. Also, these stubs are not portable because they depend on implementation classes and should not be packaged as part of an application. The design of portable stubs is out-of-scope for JAX-RPC 1.0 and 1.1.

**Dynamic proxy**

From a JAX-RPC service, you can use a proxy to invoke the web service's operations. The proxy is a Java class implementing the SEI. A proxy is obtained with the JAX-RPC service's getPort() method, which takes the name of the port for the web service you want to invoke (found in the WSDL document), as well as the SEI implemented by the proxy. It is called *dynamic* because the proxy is created at runtime. The steps for dynamic proxy clients are:

1. Get a JAX-RPC Service.
2. Obtain a proxy using the JAX-RPC service's getPort() method in order to invoke the web service's operations.

In step 1, for unmanaged clients, a JAX-RPC service is obtained from the JAX-RPC ServiceFactory by passing the WSDL URL as well as the web service name parameter to the createService() method. For J2EE container-managed clients, you get a JAX-RPC service from JNDI lookup. **Listing 5** shows the dynamic proxy method (step 2) to invoke the "getGreeting" operation on the web service.
Listing 5. Invoking a web Service's operation on a dynamic proxy

```java
String namespace = "http://Hello.com";
String portName = "Hello";
QName portQN = new QName(namespace, portName);
Hello myProxy = service.getPort(portQN, Hello.class);
System.out.println(myProxy.getGreeting("Jane"));
```

This is all the code you need to write to invoke a web service using the dynamic proxy method. The advantage of using this method is that you write portable, vendor-independent code. However, you need to know the WSDL URL at development-time and you need to run your WSDL to Java mapping tool against the WSDL document before runtime. If you do not have this information, or if the WSDL URL is likely to change, you should use the DII method instead.

**Dynamic Invocation Interface (DII)**

The JAX-RPC `Call` interface supports the dynamic invocation of a web services' operations. With this method, you do not need to know the WSDL URL at development-time. Instead of obtaining a proxy from the JAX-RPC `Service`, the JAX-RPC `Service` acts as a factory for instantiating JAX-RPC `Call`s. The steps for this method are:

1. Get a JAX-RPC `Service`.
2. Instantiate a JAX-RPC `Call` using JAX-RPC `Service`'s `createCall()` method.
3. Configure your `Call` instance with its setter methods.
4. Invoke the web service's operation using the JAX-RPC `Call`'s `invoke` method.

In step 1, for unmanaged clients, a JAX-RPC `Service` is obtained from the JAX-RPC `ServiceFactory` by passing only the name of the web service (not the WSDL URL) to the `createService()` method. For J2EE container-managed clients, you get a JAX-RPC `Service` from JNDI lookup. In step 3, configuration parameters are: name of the operation, port type, address of the target service endpoint, return type. Refer to section 8.2.4.2 of the JAX-RPC specification for the standard set of properties (see Related topics). Steps 2 to 4 are shown in Listing 6.

Listing 6. Invoking a web service using the DII method

```java
String namespace = "http://Hello.com";
String portName = "Hello";
QName portQN = new QName(namespace, portName);
String operationName = "getGreeting";
Call call = service.createCall();
call.setPortTypeName(portQN);
call.setOperationName(new QName(namespace, operationName));
call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY, "");
call.setProperty(Call.OPERATION_STYLE_PROPERTY, "wrapped");
call.addParameter("param1", <xsd:string>, ParameterMode.IN);
call.setReturnType(<xsd:string>);
Object[] inParams = new Object[] {"Jane"};
String ret = (String) call.invoke(inParams);
```

You can reuse the `Call` instance to invoke other operations on the web service.
Note: the createCall() and addParameter() methods have other signatures. What was just described is not the only way to invoke them. For example, it is possible to invoke createCall() with port type name and operation name parameters.

Making DII calls through a Call object is programmatically more complex than using a stub or dynamic proxy. However, the advantage of using a DII Call interface is that a client can call a remote procedure without development-time knowledge of the WSDL URI or the web service operations' signatures. This makes the code easy to modify if the web service details change. With DII clients, runtime classes generated by WSDL to Java mapping tools (emitters) are not required like the dynamic proxy or static stub cases. However, if you know the web service you want to invoke is unlikely to change, you should use dynamic proxy because configuring the Call instance can be complex.

**Dynamic Discovery and Invocation (DDI)**

Dynamic Discovery and Invocation (DDI) is the ultimate use of web services' flexibility where a web service client can dynamically discover and invoke a web service without any prior knowledge of it. Although DII clients, described in the previous section, do not require development-time knowledge of a web service’s details, they do not involve the process of discovering the web service. A DDI client performs three steps:

1. Discovers the web service's details from UDDI: finds the business providing the service and then the URL of the WSDL document describing the service
2. Reads the WSDL document to find information on the web service: namespace, service, ports, and parameters
3. Invokes the service.

In step 1, the UDDI Registry Enquiry API is used to browse the UDDI registry. In step 2, the UDDI4J API is used to parse the WSDL document. Finally, in step 3, the DII method (described in the previous section) is used. For information on DDI you are encouraged to read the developerWorks article "Dynamic Discovery and Invocation of web services," listed in the Related topics section.

**Conclusion**

This article described different ways to write Java web services client code. The source code is available in zipped format under "Related topics". As mentioned earlier, artifacts may be required in order to build and run the client code provided (for example, JAX-RPC or JSR 109 class libraries, Java2WSDL mapping tool-emitted stubs, Service Endpoint Interface SEI, deployment descriptors, etc.). Also, the "Hello" web service implementation needs to be up and running! Refer to the Readme file included in the zip archive for instructions on how to build and run the classes.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ws-javaclient.zip</td>
<td></td>
</tr>
</tbody>
</table>
Related topics

- Download the source code used in this article.
- Browse the Java Community Process, JCP site.
- Get the JSR 101 Java API for XML based RPC (JAX-RPC) downloads and specification.
- Check out the Web Services Interoperability organization, WS-I site.
- Find the WS-I Basic Profile, BP 1.0 specification.
- Download the IBM WebSphere SDK for web services, version 5.1.

© Copyright IBM Corporation 2003
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)