

## CHAPTER

## 10

# Building RPC Web Services with JAX-RPC

This chapter discusses the Java API for XML RPC (JAX-RPC), which enables the development of Web services incorporating XML-based remote procedural calls (RPCs).

As we discussed in Chapter 7, “Introduction to the Java Web Services Developer Pack (JWSDP),” JAX-RPC is an integral part of the JWSDP. In a JWSDP-based Web services environment, JAX-RPC provides XML-based RPC functionality between the service requestors and the service provider. JAX-RPC provides standard API mechanisms for creating RPC-based Web services and a runtime services environment for Web services applications. Through these applications, a service requestor client can invoke remote calls to access the services exposed by a service provider. JAX-RPC also defines mappings between WSDL-based service descriptions and Java classes and provides tools to generate WSDL from Java classes and vice-versa. This enables developers to create JAX-RPC clients for Web services that enable interoperability to occur between Web services applications written using different languages and/or running on heterogeneous platforms.

JAX-RPC is quite typical to Java Remote Method Invocation (RMI), which handles RPC mechanisms by passing serialized Java objects between Java applications in a distributed computing environment; whereas JAX-RPC uses SOAP-based RPC and WSDL mechanisms to invoke Web services

## 452 Chapter 10

---

running on heterogeneous environments. More importantly, JAX-RPC hides all of the complexities of the underlying SOAP packaging and messaging by providing the required mapping between Java and XML/WSDL.

Currently, the JAX-RPC 1.0 is fully compliant with the SOAP 1.1 protocol and WSDL specifications and supports HTTP as its primary transport protocol. This helps the Java Web services developers write JAX-RPC-based Web services applications with minimal effort and little understanding of SOAP RPC. In a Web services business context, using JAX-RPC the service requestor and service provider can execute SOAP-based requests and responses and exchange XML as parameters or return values.

This chapter provides in-depth coverage on the JAX-RPC-based API mechanisms and illustrates its usage scenarios for developing and deploying Web services applications. In particular, we will be focusing on the following:

- The role of JAX-RPC in Web services
- JAX-RPC application architecture
- JAX-RPC implementation model
- JAX-RPC deployment model
- Data type mappings between XML and Java
- Understanding Java to WSDL and WSDL to Java mappings
- Developing JAX-RPC-based Web services.
- JAX-RPC in J2EE 1.4

At the time of this book's writing, JAX-RPC 1.0 has been released as a final specification; its reference implementation and API libraries are available for downloading as part of JWSDP 1.0 at Sun's Web site: <http://java.sun.com/xml/download.html>.

JAX-RPC was developed by Sun Microsystems as part of its Java Community Process (JCP), backed by J2EE vendors and most Java-based Web services platforms. In this chapter, we will be using the JWSDP 1.0 Reference Implementation (RI) for discussion and also to illustrate the case study examples.

### The Role of JAX-RPC in Web Services

---

In a Web services environment, JAX-RPC defines an API framework and runtime environment for creating and executing XML-based remote procedural calls. The Web service requestors invoke the service provider's

methods and transmit parameters and then receive return values as XML-based requests and responses. Typically, the Web service endpoints and the participating application clients use JAX-RPC for defining and executing the RPC-based Web services.

The core features of JAX-RPC are as follows:

- Provides APIs for defining RPC-based Web services, hiding the underlying complexities of SOAP packaging and messaging
- Provides runtime APIs for invoking RPC-based Web services based on
  - Stub and ties
  - Dynamic proxy
  - Dynamic invocation
- Using WSDL, JAX-RPC enables interoperability with any SOAP 1.1-based Web services
- Provides data types mapping between Java and XML
- Supports standard Internet protocols such as HTTP
- Service endpoints and service clients are portable across JAX-RPC implementations

The JAX-RPC-based Web services can be deployed in the Java servlet 2.2- or the J2EE 1.3-compliant server providers. JAX-RPC-based services and clients are developed and deployed as J2EE components.

In a Web services scenario, using JAX-RPC, the service requestor and service provider can communicate and execute SOAP-based requests and responses and exchange XML as parameters or return values. For the service provider, JAX-RPC provides support for exposing business components as Web services; for the service requestor, JAX-RPC defines a mechanism to access and invoke JAX-RPC-based services or any SOAP 1.1-compliant RPC-based Web services. JAX-RPC provides APIs for data type mappings between Java and XML, which enables on-the-fly Java-to-XML and XML-to-Java mappings during communication; that is, when a client invokes a JAX-RPC service, its XML parameters are mapped to Java objects. Similarly, while returning a response, the Java objects are mapped to XML elements as return values. JAX-RPC also enables the development of pluggable serializers and deserializers to support any data type mapping between Java and XML, which can be packaged with a JAX-RPC service. In addition, JAX-RPC defines mappings between WSDL and Java, through which a WSDL document provided by Web services can be mapped to Java classes as client stubs and server-side ties.

## 454 Chapter 10

**Table 10.1** Comparison of JAX-RPC with JAXM

| JAX-RPC  | JAXM  |
|--|---|
| Synchronous RPC-based service interaction        | Asynchronous/synchronous messaging-based service interaction        |
| Message sent as XML-based requests and responses | Message sent as document-driven XML messages                        |
| Exposes internals to service requestors          | Loosely coupled and does not expose internals to service requestors |
| Provides a variety of client invocation models   | Does not provide a client-specific programming model                |
| No reliability mechanisms                        | Provides guaranteed message delivery and reliability mechanisms     |

### Comparing JAX-RPC with JAXM

While comparing JAX-RPC with JAXM, both are an integral part of JWSDP but provide different API-based mechanisms for developing Web services. Table 10.1 compares the features of JAX-RPC to JAXM.

JAX-RPC is also a best-fit solution over JAXM especially in request/response communications where high performance, limitations in memory, and maintaining client state are defined as the key requirements.

Now, let's take a closer look at a JAX-RPC-based Web services application and explore its features.

### JAX-RPC Application Architecture

Figure 10.1 represents a JAX-RPC-based application architectural model and its core elements.

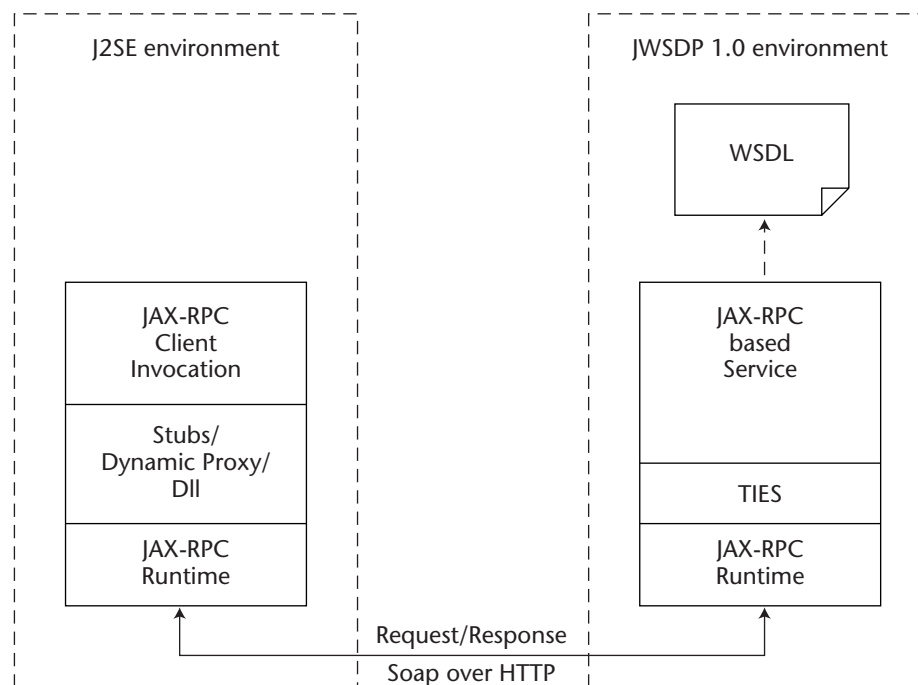
A typical JAX-RPC application architectural model consists of the following elements:

**JAX-RPC service.** Represents a business component that can be implemented in Java or generated from existing Java classes or from a WSDL document. In a J2EE environment, it can be implemented as a servlet, stateless session bean, or a message-driven bean. During deployment, the JAX-RPC service is assigned with one or more service endpoints and then is configured to a transport protocol binding. For instance, a JAX-RPC service can be bound to HTTP and all the messages are exchanged as HTTP-based requests and

responses using its assigned endpoint. The JAX-RPC services do not dictate that it has to be accessed by a JAX-RPC client and thus a non-Java client running on heterogeneous environments can access it.

**JAX-RPC service client.** Represents a JAX-RPC-based service client that can access a service. The service client is independent of the target implementation on the service provider. This means that the accessed service can be a service implemented using a Java platform or a SOAP 1.1-compliant service running on a non-Java platform. To support these client scenarios, JAX-RPC defines a variety of client invocation models using different mechanisms, such as stubs-based mechanisms, dynamic proxies, and dynamic invocation. The JAX-RPC service clients can import WSDL exposed by a service provider and can generate Java-based client classes to access the service.

**Serialization and deserialization.** During communication, JAX-RPC uses serializing and deserializing mechanisms to facilitate Java-to-XML and XML-to-Java mappings, which enables the conversion of the Java primitives and objects to XML-based representations and vice versa. It also is possible to create serializers and deserializers for custom data types.



**Figure 10.1** JAX-RPC-based Web services application architecture.

## 456 Chapter 10

---

**xrpcc tool.** The JAX-RPC provides the `xrpcc` tool, which enables the generation of the required client and server-side class files and WSDL to enable communication between a service provider and a requestor. In particular, `xrpcc` generates a WSDL document representing the service and the stubs and ties that define the lower-level Java classes. In a service client scenario, `xrpcc` also enables Stub classes to generate using a WSDL exposed by a service provider.

**JAX-RPC runtime environment.** Defines the runtime mechanisms and execution environment for the JAX-RPC services and service clients by providing the required runtime libraries and system resources. The JAX-RPC 1.0 specification mandates a servlet 2.3 or J2EE 1.3-based servlet container environment for its services and service clients. As per JAX-RPC 1.0, both the JAX-RPC services and service clients using JAX-RPC APIs are required to be implemented as servlets running a servlet 2.2-complaint container. Upcoming J2EE 1.4 specifications will likely provide support for JAX-RPC and enable the creation of JAX-RPC-based services using session beans and message-driven beans. The JAX-RPC 1.0 runtime services also provide support for HTTP-based basic authentication and session management.

Now, let's explore the previous JAX-RPC features and how they are represented using the JAX-RPC APIs and implementation.

### JAX-RPC APIs and Implementation Model

---

As per JAX-RPC 1.0 specifications, JAX-RPC defines both service and client implementation models intending to do RPC-based messaging using a JAX-RPC runtime environment or directly to a SOAP service. The core JAX-RPC APIs are packaged as `javax.xml.rpc`, which provides the runtime mechanisms package and `javax.xml.rpc.handler.*` as its SOAP message handler API package.

### JAX-RPC-Based Service Implementation

The JAX-RPC 1.0 specification does not define any APIs for implementing JAX-RPC-based services. JAX-RPC-based services can be implemented using Java classes (similar to writing an RMI application) or by using a WSDL document. In both cases, JAX-RPC does not specify any requirements for its service client implementation to access and use the deployed services.

The following two sections look at those two different ways of services implementation and walk through the programming steps required.

### ***Developing a JAX-RPC Service from Java Classes***

As we mentioned earlier, developing a JAX-RPC-based service (also referred to as a JAX-RPC service definition) is quite similar to developing an RMI application. The steps involved are as follows:

1. Define the remote interface (Service Definition).
2. Implement the remote interface (Service Implementation).
3. Configure the service.
4. Generate the stubs and ties.
5. Package and deploy the service.

Now, let's take a look at the previous steps by walking through a sample application.

#### **Defining the Remote Interface (Service Definition)**

The service interface defines the set of exposed methods that can be invoked by the service requestor clients. In JAX-RPC, it is referred to as Service Definition. The client stubs and server ties are generated based on this interface. From a programming model's standpoint, the programming rules involved in defining the remote interface are as follows:

- The remote interface of the service definition must be declared as public.
- The remote interface must extend the `java.rmi.Remote` interface and all of the methods must throw a `java.rmi.RemoteException`.
- The remote interface must not contain any declaration as static constants.
- All of the parameters and return values of the methods must be supported as part of JAX-RPC-supported data types. In case of unsupported data types, then it is required to use custom serializers and deserializers to facilitate Java-to-XML and XML-to-Java mappings.

For example, the following is a code listing that defines a remote interface of a service definition (`BookPriceIF.java`):

## 458 Chapter 10

---

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

public interface BookPriceIF extends Remote {
    public String getBookPrice(String bookName)
        throws RemoteException;
}
```

### Implementing the Remote Interface (Service Implementation)

The implementation of the remote interface is the actual class that implements all of the exposed methods defined in the remote interface. In JAX-RPC, it is referred to as Service Implementation. For example, a typical service implementation of a remote interface is as follows:

```
import java.rmi.Remote;
import java.rmi.Remote.*;

public class BookPriceImpl implements BookPriceIF {

    public String getBookPrice (String bookName) {
        System.out.println("The price of the book titled "+ bookName);
        return new String("13.25");
    }
}
```

The service implementation can also implement the `javax.xml.rpc.server.ServiceLifecycle` interface that allows handling the complete lifecycle of a JAX-RPC service. The `ServiceLifecycle` interface provides `init()` and `destroy()` methods, which are quite similar to the `init()` and `destroy()` methods in a Servlet lifecycle for initializing and releasing resources. In the case of a service implementation implementing a `ServiceLifecycle` interface, also allows to set `ServletEndpointContext`, which is quite similar to `SessionContext` (in EJBs) that enables to maintain state information. For example, a typical service implementation of a remote interface implementing `ServiceLifecycle` is as follows:

```
import java.rmi.Remote;
import java.rmi.Remote.*;
import javax.xml.rpc.server.ServiceLifecycle;
import javax.xml.rpc.server.ServletEndpointContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;

public class BookPriceImpl implements BookPriceIF, ServiceLifecycle {
    private ServletEndpointContext serviceContext;
```

```
public void init(java.lang.Object context){
    serviceContext=(ServletEndpointContext)context;
}

public String getBookPrice (String bookName) {
    SOAPMessageContext soapMsgContext=
        (SOAPMessageContext) (serviceContext.getMessageContext());
    HttpSession session = serviceContext.getHttpSession();
    //...Obtain state information here
    System.out.println("The price of the book titled "+ bookName);
    return new String("13.25");
}
}
```

### Configuring the Service

To configure the service, you first need to create a configuration file in an XML format that provides information such as

- The name of the service
- The name of the package containing the stubs and ties
- The target namespace for the generated WSDL and its XML schema and class names of the remote interface and its implementation

The `xrpscc` tool uses this configuration file to generate the stubs and ties of the service.

Listing 10.1 is a code listing of a sample configuration file (`service-config.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="WileyBookCatalog"
    targetNamespace="http://www.wiley.com/jws/wsdl"
    typeNamespace="http://www.wiley.com/jws/types"
    packageName="com.wiley.jws.ch10.jaxrpc">
    <interface name="jws.wiley.jaxrpc.BookPriceIF"
      servantName="jws.wiley.jaxrpc.BookPriceImpl"/>
    </service>
  </configuration>
```

**Listing 10.1** Sample configuration of a service definition.

To find out the other optional elements of the service configuration file, refer to the JAX-RPC API documents provided with the JWSDP 1.0 bundle.

## 460 Chapter 10

---

### Generating the Stubs and Ties

Before generating the stubs and ties, ensure that the source code of the remote interface and the implementation is compiled using `javac` and that it is available in `CLASSPATH`.

Use the `xrpcc` tool to generate the stubs and tie classes, the WSDL document associated with the service, and the property files required by the JAX-RPC runtime environment. In a typical scenario, the `xrpcc` tool can be executed as a command line utility, as follows:

In a Windows environment:

```
xrpcc -classpath %CLASSPATH% -keep
      -both -d build\classes serviceconfig.xml
```

In a UNIX environment:

```
xrpcc -classpath $CLASSPATH -keep
      -both -d build/classes serviceconfig.xml
```

In this command, the option `-classpath` refers to the `CLASSPATH`, including the service interface and implementation classes and JAX-RPC libraries; `-keep` refers to saving the generated source files (.java) and the WSDL documents; `-both` refers to the generating of both the stubs and tie classes; and `-d` refers to the destination directory. To find out more `xrpcc` options, refer to the JAX-RPC implementation documentation for syntax and usage information.

As a result, the preceding command generates the following:

- Client-side stubs and server-side tie classes
- Serialization and deserialization classes representing the data-type mappings between Java primitives and XML data types
- A WSDL document
- Property files associated with the service

### Packaging and Deployment

According to JWSDP 1.0, JAX-RPC-based services are specified with only servlet-based service endpoints and the JAX-RPC services are required to be deployed as a servlet in a Java servlet 2.2-based container. This mandates that the JAX-RPC-based services be packaged as a Web application (WAR).

To package a JAX-RPC service as a Web application, we need to create a WAR file that includes the following classes and other configuration files:

- Remote interface of the service
- Service implementation of the remote interface
- Serializer and deserializer classes
- Server-side (tie) classes created by `xrpcc`
- Property files created by `xrpcc`
- Other supporting classes required by the service implementation
- WSDL document describing the service
- Web application deployment descriptor (`web.xml`)

Except for the deployment descriptor, we have seen the process required for creating those classes and property files using `xrpcc`.

In a JAX-RPC environment, the deployment descriptor is similar to a servlet deployment descriptor (`web.xml`), which provides information about the class name of the JAX-RPC service, its associated property file created by the `xrpcc` tool, the servlet mappings and URL pattern, and so on. Listing 10.2 is a sample code listing of a deployment descriptor (`web.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
<display-name>WileyProductServices</display-name>
<description>Wiley Web Services Company</description>
<servlet>
  <servlet-name>JAXRPCEndpoint</servlet-name>
  <display-name>JAXRPCEndpoint</display-name>
  <description>Endpoint for Wiley Book Catalog
Service</description>
  <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet
    </servlet-class>

  <init-param>
    <param-name>configuration.file</param-name>
    <param-value>/WEB-INF/WileyBookCatalog_Config.properties
    </param-value>

  </init-param>
  <load-on-startup>0</load-on-startup>
</servlet>
```

**Listing 10.2** Sample deployment descriptor for a JAX-RPC service definition. (*continues*)

## 462 Chapter 10

```
<servlet-mapping>
  <servlet-name>JAXRPCEndpoint</servlet-name>
  <url-pattern>/jaxrpc/wiley/*</url-pattern>
</servlet-mapping>
<session-config>
  <session-timeout>60</session-timeout>
</session-config>
</web-app>
```

**Listing 10.2** Sample deployment descriptor for a JAX-RPC service definition. (*continued*)

To package the service as Web application, use the `jar` utility and create a Web application archive (WAR). For example,

```
jar cvf wileywebapp.war .
```

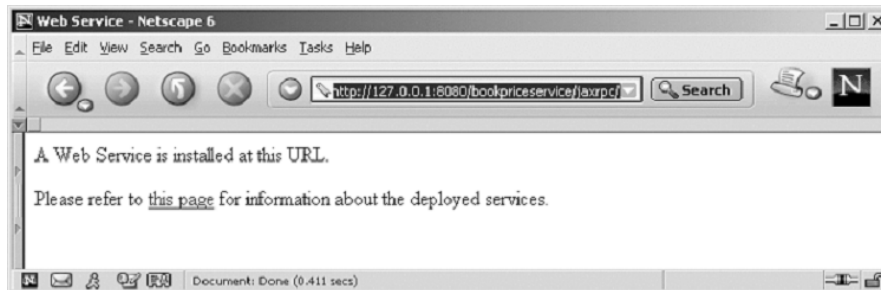
Finally, to deploy the service as a Web application running in a JWS DP 1.0/Tomcat environment, just copy the WAR file to the servlet engine / `webapps` directory. For example, to deploy in a Tomcat servlet container in a Windows environment, deploy the following:

```
copy wileywebapp.war %CATALINA%/webapps
```

Then restart the Tomcat server, which automatically deploys the service as an application. To verify the service deployment, use your Web browser and execute the following URL (using the URL pattern defined in the deployment descriptor):

```
http://localhost:8080/bookpriceservice/jaxrpc/wiley
```

If everything has been deployed successfully, the browser will display “A Web Service is installed at this URL” (see Figure 10.2).



**Figure 10.2** Browser showing successful installation of a JAX-RPC-based service.

### ***Developing a JAX-RPC-Based Service from a WSDL Document***

In this section, we will look at developing a JAX-RPC-based service using a WSDL document exposed by an existing Web services environment. In this case, by importing a WSDL document from an existing Web service, the `xrpcc` utility generates the JAX-RPC services classes. The key steps involved are as follows:

1. Create a service configuration referring to the WSDL.
2. Generate the client-side stubs and server-side ties using `xrpcc`.
3. Package and deploy the service.

To illustrate the previous steps, read the following sections. For the example, assume that a Web service and its WSDL location are available at the following URL:

```
http://nramesh:80/axis/AcmeProductCatalog.jws?WSDL
```

#### **Create a Service Configuration Using WSDL**

To configure the service, you first must create a configuration file in an XML format that provides information about the URL location of the WSDL, including the name of the service and the name of the package for the generated stubs and ties. The `xrpcc` tool uses this configuration file to generate the stubs and ties and the RMI interfaces of the service.

Listing 10.3 is a sample code listing of a sample configuration file (`serviceconfig.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location=" http://nramesh:80/axis/AcmeProductCatalog.jws?WSDL"
    packageName="com.wiley.jws.ch10.jaxrpc.wsdl">
  </wsdl>
</configuration>
```

**Listing 10.3** Sample service configuration using WSDL.

To find out other optional elements of the service configuration file, refer to the JAX-RPC API documents.

#### **Generating the Stubs and Ties**

Before generating the stubs and ties, ensure that the source code of the remote interface and the implementation is compiled using `javac`. Also ensure that the code is available in `CLASSPATH`.

## 464 Chapter 10

---

Use the `xrpsc` tool to generate the stubs and tie classes, the WSDL document associated with this service, and the property files required by the JAX-RPC runtime environment. In a typical scenario, the `xrpsc` tool can be executed as a command line utility shown as follows:

In a Windows environment:

```
xrpsc -classpath %CLASSPATH% -keep  
      -both -d build\classes serviceconfig.xml
```

In a UNIX environment:

```
xrpsc -classpath $CLASSPATH -keep  
      -both -d build/classes serviceconfig.xml
```

In the previous command, the option `-classpath` refers to the CLASSPATH including the service interface and implementation classes and JAX-RPC libraries, `-keep` refers to saving the generated source files (.java) and the WSDL documents, `-both` refers to the generating of both the stubs and tie classes, and `-d` refers to the destination directory. To find out more `xrpsc` options, refer to the JAX-RPC implementation documentation for syntax and usage information.

As a result, the previous command generates the following:

- Client-side stubs and server-side tie classes
- Serialization and deserialization classes representing the data-type mappings between Java primitives and XML data types
- A WSDL document for the service
- Property files for service configuration

### Packaging and Deployment

The packaging and deployment steps are quite similar to those we discussed in the earlier section titled *Developing a JAX-RPC from Service using Java Classes*. Because JWSDP 1.0 specifies only servlet-based service endpoints, the JAX-RPC-based services are packaged as a Web Application (WAR).

### JAX-RPC-Based Client Implementation

According to the JAX-RPC 1.0 specification, JAX-RPC-based service clients are independent of target service implementation, and the service client does not depend upon its service provider or its running platform using

Java or non-Java environments. The JAX-RPC provides client-side APIs and defines a client invocation model for accessing and invoking Web services.

Now, let's take a look at the JAX-RPC client-side APIs and the client invocation models.

### **JAX-RPC Client-Side APIs**

The JAX-RPC 1.0 client-side APIs are defined in a single package structure as `javax.xml.rpc`, which provides a set of interfaces and classes that support the JAX-RPC clients intending to invoke RPC-based services, and the JAX-RPC runtime implements them.

The `javax.xml.rpc` package provides a set of interfaces and classes for creating JAX-RPC clients, which support the different JAX-RPC client invocation models. The JAX-RPC API interfaces and classes are as follows:

#### **INTERFACES**

**`javax.xml.rpc.Stub`** Is the base interface for the JAX-RPC stub classes. All JAX-RPC stub classes are required to implement this interface. This interface represents the client-side proxy or an instance of the target endpoint.

```
private static Stub createMyProxy() {
    return (Stub)(new StockPrice_Impl().getStockPriceIFPort());
}
```

**`javax.xml.rpc.Service`** Acts as a factory class for creating a dynamic proxy of a target service and for creating `Call` instances for invoking the service.

```
Service service =
    serviceFactory.createService(new QName(qService));

Call call = service.createCall(target_port);
```

**`javax.xml.rpc.Call`** Provides support for the JAX-RPC client components to dynamically invoke a service. In this case, after a call method is created, we need to use the setter and getter methods to configure the call interface for the parameters and return values.

```
Call call = service.createCall(target_port);
call.setTargetEndpointAddress(target_endpoint);

call.setOperationName
    (new QName(BODY_NAMESPACE_VALUE, "getStockPrice"));
```

## 466 Chapter 10

### CLASSES

**javax.xml.rpc.ServiceFactory** Is an abstract class that provides a factory class for creating instances of `javax.xml.rpc.Service` services.

```
ServiceFactory sfactory =
    ServiceFactory.newInstance();
Service service =
    sfactory.createService(new QName(qService));
```

**javax.xml.rpc.ParameterMode** Provides a type-safe enumeration of the parameter mode.

```
call.addParameter( "stocksymbol", QNameType.STRING,
    ParameterMode.IN );
```

**javax.xml.rpc.NamespaceConstants** Defines the constants used in JAX-RPC for the XML schema namespace prefixes and URIs.

### EXCEPTIONS

**javax.xml.rpc.JAXRPCException** Throws an exception while a JAX-RPC exception is occurring. The exception details the reasons for the failure, which are related to JAX-RPC runtime-specific problems.

**javax.xml.rpc.SERVICEException** Throws an exception from the methods in the `JAVAX.XML.RPC.SERVICE` interface and `ServiceFactory` class.

### *JAX-RPC Client Invocation Programming Models*

The JAX-RPC 1.0 specification defines the implementation of a JAX-RPC-based client using any of the following client invocation programming models:

- Stub-based
- Dynamic proxy
- Dynamic invocation

Now, let's take a look at these different ways of client implementation and walk through the programming steps required for each.

#### **Stub-Based Client**

A stub-based model is the simplest client-programming model. This model uses the local stub classes generated by the `xrpsc` tool. To create the

stub-based client invocation, ensure that the stub classes are available in the CLASSPATH.

Listing 10.4 is a sample code listing of a client using stub-based client invocation.

```
// Import the Stub Interface
import javax.xml.rpc.Stub;

public class BookCatalogClient {

    // Main method
    public static void main(String[] args) {

        try {
            // Obtain the Instance of the Stub Interface
            BookCatalogIF_Stub stub = (BookCatalogIF_Stub)
                (new BookCatalog_Impl().getBookCatalogIFPort());

            // Configure the stub setting required properties
            // Setting the target service endpoint
            stub._setProperty(
                javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                "http://www.wiley.com/jws/jaxrpc/bookcatalog");
            // Execute the remote method
            System.out.println (stub.getBookPrice("JAX-RPC in 24 hours"));

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

**Listing 10.4** Sample code illustrating a stub-based client invocation.

### Dynamic Proxy-Based Client

A dynamic proxy client enables the invocation of a target service endpoint dynamically at runtime, without requiring a local stub class. This type of client uses the dynamic proxy APIs provided by the Java reflection API (`java.lang.reflect.Proxy` class and `java.lang.reflect.InvocationHandler` interface). Particularly, the `getPort` method on the `javax.xml.rpc.Service` interface enables a dynamic proxy to be created. Listing 10.5 is a sample code listing of a client using a dynamic proxy-based client invocation.

**468 Chapter 10**

```
// Import Service & ServiceFactory

import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;

import javax.xml.namespace.QName;
import java.net.URL;

public class BookCatalogProxyClient {

    // Main Method
    public static void main(String[] args) {

        try {
            // WSDL location URL
            String wsdlURL =
                "http://www.wiley.com/jws/jaxrpc/BookCatalog.wsdl";

            // WSDL namespace URI
            String namespaceURI = "http://www.wiley.com/jws/wsdl";

            // Service Name
            String serviceName = "BookCatalogService";

            // Service port Name
            String portName = "BookCatalogIFPort";

            URL bookCatalogWSDL = new URL(wsdlURL);

            // Obtain an Instance of Service factory
            ServiceFactory serviceFactory = ServiceFactory.newInstance();

            // Create a service from Instance using WSDL
            Service bookCatalogService =
                serviceFactory.createService(bookCatalogWSDL,
                    new QName(namespaceURI,
                        serviceName));

            // Get the proxy object
            BookCatalogIF bcProxy = (BookCatalogIF)
                bookCatalogService.getPort(
                    new
                    QName(namespaceURI, portName), proxy.BookCatalogIF.class);

            // Invoke the remote methods
            System.out.println(bcProxy.getBookPrice("JAX-RPC in 24 hours "));

        } catch (Exception ex) {
```

**Listing 10.5** Sample code illustrating a client using dynamic proxy.

```
        ex.printStackTrace();
    }
}
}
```

**Listing 10.5** Sample code illustrating a client using dynamic proxy. (*continued*)

### Dynamic Invocation Interface (DII) Client

Using the Dynamic Invocation Interface (DII) enables the client to discover target services dynamically on runtime and then to invoke methods. During runtime, the client uses a set of operations and parameters, establishes a search criterion to discover the target service, and then invokes its methods. This also enables a DII client to invoke a service and its methods without knowing its data types, objects, and its return types.

DII looks up a service, creates a `Call` object by setting the endpoint specific parameters and operations, and finally invokes the call object to execute the remote methods. Listing 10.6 is a sample code listing of a client using DII-based client invocation.

```
// imports
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;

public class BookCatalogDIIClient {

    // Service Name
    private static String qService = "BookCatalogService";

    // Port Name
    private static String qPort = "BookCatalogIF";

    // Name space URI
    private static String BODY_NAMESPACE_VALUE =
        "http://www.wiley.com/jws/wsdl";

    // Encoding style
    private static String ENCODING_STYLE_PROPERTY =
```

**Listing 10.6** Sample code illustrating a JAX-RPC client using DII. (*continues*)

**470 Chapter 10**

```
        "javax.xml.rpc.encodingstyle.namespace.uri";

// XML Schema
private static String NS_XSD =
    "http://www.w3.org/2001/XMLSchema";

// SOAP encoding URI
private static String URI_ENCODING =
    "http://schemas.xmlsoap.org/soap/encoding/";

// Main method
public static void main(String[] args) {

    try {
        String target_endpoint =
            "http://www.wiley.com/jws/jaxrpc/bookcatalog";

        // Obtain an Instance of Service factory
        ServiceFactory sFactory =
            ServiceFactory.newInstance();

        // Create a Service
        Service service =
            sFactory.createService(new QName(qService));

        // Define a port
        QName port = new QName(qPort);

        // Create a Call object using the service
        Call call = service.createCall(port);

        // Set the target service endpoint
        call.setTargetEndpointAddress(target_endpoint);

        // Set properties - SOAP URI, Encoding etc.
        call.setProperty(Call.SOAPACTION_USE_PROPERTY,
            new Boolean(true));

        call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");

        call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);

        // Set Parameter type and Return value type as String
        QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");

        call.setReturnType(QNAME_TYPE_STRING);

        // Set operations
        call.setOperationName(new QName(BODY_NAMESPACE_VALUE,
```

**Listing 10.6** Sample code illustrating a JAX-RPC client using DII.

```
        "getBookPrice"));

    // Set Parameters
    call.addParameter("BookName", QNameType.STRING,
                     ParameterMode.IN);

    String[] BookName = {"JAX-RPC in 24 hours"};

    // Invoke and obtain response object
    Object response = (Object)call.invoke(BookName);

    System.out.println(response.toString());

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
}
```

**Listing 10.6** Sample code illustrating a JAX-RPC client using DII. (*continued*)

A DII client also can be invoked using one-way RPC mechanisms. In that case, the client does not require setting the return value types, and the call can be invoked as follows:

```
call.invokeOneWay(parameter);
```

Now, let's take a look at the JAX-RPC-supported data types mapping between Java and XML.

## JAX-RPC-Supported Java/XML Mappings

JAX-RPC abstracts and hides the complexities of SOAP and XML data types by providing serialization and deserialization features and by performing automatic mapping between Java classes and XML data (and vice versa). To handle these chores, JAX-RPC 1.0 provides APIs and conventions for mappings between the Java data types and XML/WSDL data as per the XML schema 1.0 representation (XSD) and SOAP 1.1 encoding (SOAP-ENC) specifications. The online locations of those specifications are as follows:

```
http://www.w3.org/2001/XMLSchema
http://www.w3.org/2001/XMLSchema-instance
http://schemas.xmlsoap.org/soap/encoding/
```

## 472 Chapter 10

The `xrpcc` tool provides these features by automating the tasks of mapping XML to Java classes and also the mapping between WSDL definitions and their mapping Java representations.

In a JAX-RPC-based Web services scenario, when a JAX-RPC service is invoked, the JAX-RPC runtime transforms the XML-based RPC call to its corresponding Java object representation and then executes the required service using them; this process is referred to as *deserialization*. After execution, the service returns the call to its service client by transforming the returning Java objects as an XML-based data representation; this process is referred to as *serialization*.

Now, let's take a look at the standard mappings that are supported by JAX-RPC 1.0.

### Java/XML Data Type Mappings

JAX-RPC 1.0 provides support for the following mapping between Java classes and XML data types as defined in XML schema 1.0 (`xsd`) and SOAP 1.1 encoding (`SOAP-ENC`). Table 10.2 shows the JAX-RPC-supported Java primitives and their mapping XML data types.

In an example scenario, using a Java primitive, such as

```
float price;
```

is mapped to an XML schema representation as

```
<element name="price" type="xsd:float"/>
```

Table 10.3 shows the JAX-RPC-supported Java classes and their mapping XML data types.

**Table 10.2** JAX-RPC-Supported Java Primitives and Mapping XML Data Types

| JAVA PRIMITIVES      | XML SCHEMA DEFINITION    |
|----------------------|--------------------------|
| <code>int</code>     | <code>xsd:int</code>     |
| <code>long</code>    | <code>xsd:long</code>    |
| <code>float</code>   | <code>xsd:float</code>   |
| <code>double</code>  | <code>xsd:double</code>  |
| <code>short</code>   | <code>xsd:short</code>   |
| <code>boolean</code> | <code>xsd:boolean</code> |
| <code>byte</code>    | <code>xsd:byte</code>    |

**Table 10.3** JAX-RPC-Supported Java Classes and Mapping XML Data Types

| JAVA CLASSES | XML SCHEMA DEFINITION |
|--------------|-----------------------|
| String       | xsd:string            |
| BigDecimal   | xsd:decimal           |
| BigInteger   | xsd:integer           |
| Calendar     | xsd:dateTime          |
| Date         | xsd:dateTime          |

## Arrays

JAX-RPC supports the mapping of XML-based array types to a Java array. This enables mapping an XML array containing elements of XML data types to corresponding types of the Java array. For example, a Java array, such as

```
int [] employees;
```

is mapped to an XML schema representation as

```
<element name="employees" type="xsd:Array"/>

<employees arrayType="xsd:int[2]">
  <employeeID>1001</employeeID>
  <employeeID>1002</employeeID>
</employees>
```

## Java Classes to XML Structure and Complex Types

JAX-RPC provides support for mapping XML structure and complex value types as JavaBeans with the getter and setter methods. The bean property must be a JAX-RPC-supported Java type, as in the code in the following example.

The following XML schema represents information about a product:

```
<element name="Product"/>
<complexType>
  <all>
    <element name="productID" type="xsd:int"/>
    <element name="productDesc" type="xsd:string"/>
    <element name="price" type="xsd:float"/>
    <element name="color" type="xsd:string"/>
  <all>
</complexType>
```

## 474 Chapter 10

The preceding schema is mapped to a Java class representation as follows:

```
public class Product implements java.io.Serializable {
    // ...
    public String getProductID() { ... }
    public void setProductID(int productID) { ... }
    public String getProductDesc() { ... }
    public void setProductDesc(String productDesc) { ... }
    public float getPrice() { ... }
    public void setPrice(float price) { ... }
    public String getColor() { ... }
    public void setColor(String color) { ... }
}
```

### Java/WSDL Definition Mappings

JAX-RPC specifies the mappings for a JAX-RPC-based service endpoint definition to a WSDL service description and vice versa. The `xrpc` tool facilitates these mappings by mapping a WSDL document to a Java package, including Java interfaces and classes that provide bindings of a WSDL document in a Java representation.

The Java representation mapping to the abstract WSDL definitions is as follows:

**wSDL:types** Maps the WSDL message types to the Java method parameter types of the target service.

**wSDL:message** Maps the WSDL message to the Java method parameters of the target service.

**wSDL:operation** Maps the WSDL operation to the Java method of the service interface.

**wSDL:portType** Maps the WSDL port type to the service interface.

The following WSDL document represents a service for obtaining a book price from a book catalog service:

```
<message name="GetBookPriceInput">
    <part name="bookName" type="xsd:string"/>
</message>
<message name="GetBookPriceOutput"?
    <part name="price" type="xsd:float"/>
</message>
<portType name="BookCatalogServiceIF">
    <operation name="GetBookPrice" parameterOrder="bookName">
```

```

        <input message="tns:GetBookPriceInput"/>
        <output message="tns:GetBookPriceOutput"/>
    </operation>
</portType>

```

The preceding document is mapped to a Java class representation (BookCatalogServiceIF.java) as

```

public interface BookCatalogServiceIF extends java.rmi.Remote {
    public float getBookPrice(String bookName)
        throws java.rmi.RemoteException;
}

```

As we discussed earlier, the `xrpcc` tool facilitates the WSDL-to-Java and Java-to-WSDL mappings. In the future, it is expected that JAX-RPC will support Java APIs for XML binding (JAXB) for providing Java-to-XML and XML-to-Java mappings. To find out more information on JAXB, refer to Chapter 8, “XML Processing and Data Binding with Java APIs.”

### Handling SOAP Attachments in JAX-RPC

As per SOAP 1.1 specifications, a SOAP message may contain zero to many attachment parts using MIME encoding. JAX-RPC allows attaching SOAP attachment parts using JavaBeans Activation Framework (JAF). During runtime, JAX-RPC uses `javax.activation.DataHandler` and `javax.activation.DataContentHandler`, which provide access to the attachments using the `getContent` method of the `DataHandler` class.

Table 10.4 lists the standard Java data type mapping of the attachment parts for certain MIME types.

**Table 10.4** Mapping of MIME Types to Java Data Types

| MIME TYPE                      | JAVA DATA TYPE                    |
|--------------------------------|-----------------------------------|
| image/gif                      | java.awt.Image                    |
| image/jpeg                     | java.awt.Image                    |
| text/plain                     | java.lang.String                  |
| multipart/*                    | javax.mail.internet.MimeMultipart |
| text/xml or<br>application/xml | javax.xml.transform.Source        |

## Developing JAX-RPC-Based Web Services

In this section, we illustrate and discuss an example scenario of developing JAX-RPC-based Web services applications and JAX-RPC-based service client invocation models using the JWSDP 1.0 environment.

To demonstrate this, we use a fictitious example implementing a JAX-RPC-based request/response scenario showing how a service requestor (service client) obtains a book price from a JAX-RPC-based Web services provider. The service client makes a request using a book name (string) as a parameter, and the service provider returns a response to the service client with the book price (float).

### Creating a JAX-RPC-Based Service (`BookPriceService`)

The key steps for creating a JAX-RPC-based service (`BookPriceService`) using a JWSDP 1.0/Tomcat-based environment are as follows:

1. Develop the remote interface of the service (`BookPriceServiceIF.java`).
2. Create the implementation class of the remote interface (`BookPriceServiceImpl.java`).
3. Configure the service (`BookPriceService.xml`).
4. Set up the environment and compile the source code.
5. Generate the server-side artifacts (ties) and the WSDL document.
6. Package and deploy the service (`BookPriceService.war`).
7. Test the service deployment and the WSDL.
8. Generate the client stubs and package as a client JAR (`BookPriceServiceStubs.jar`).

The following sections will explore the preceding tasks involved in creating `BookPriceService` and will walk you through them.

#### ***Develop the Service's Remote Interface***

The service's remote interface defines the remote methods of the `BookPriceService` that are invoked by the service requestor clients. Listing 10.7 is a code listing that defines a service's remote interface.

```
(BookPriceServiceIF.java):
package jws.ch10.jaxrpc.bookprice;

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

public interface BookPriceServiceIF extends Remote {
    public String getBookPrice(String bookName)
        throws RemoteException;
}
```

**Listing 10.7** BookPriceServiceIF.java.

### ***Create the Service Implementation Class of the Interface***

It provides an implementation for all the methods declared in the remote interface. The code in Listing 10.8 implements the methods in a service's remote interface (BookPriceServiceImpl.java).

```
package jws.ch10.jaxrpc.bookprice;

public class BookPriceServiceImpl implements BookPriceServiceIF {

    float bookprice = 0;

    // Implementation of getBookPrice() method
    // creates a fictitious price !

    public float getBookPrice( String bookName) {

        for( int i = 0; i < bookName.length(); i++ ) {
            bookprice = bookprice + (int) bookName.charAt( i );
        }
        bookprice = bookprice/3;

        return bookprice;
    }
}
```

**Listing 10.8** BookPriceServiceImpl.java.

## 478 Chapter 10

### Configure the Service

To generate the client-side and server-side artifacts (stubs and ties), you must create a configuration file that provides information on the service name, target namespaces of the service, required package and class names, and so forth.

Listing 10.9 is a code listing of the configuration file (`BookPriceService.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="BookPriceService"
    targetNamespace="http://jws.wiley.com/wsdl"
    typeNamespace="http://jws.wiley.com/types"
    packageName="jws.ch10.jaxrpc.bookprice">
    <interface name="jws.ch10.jaxrpc.bookprice.BookPriceServiceIF"

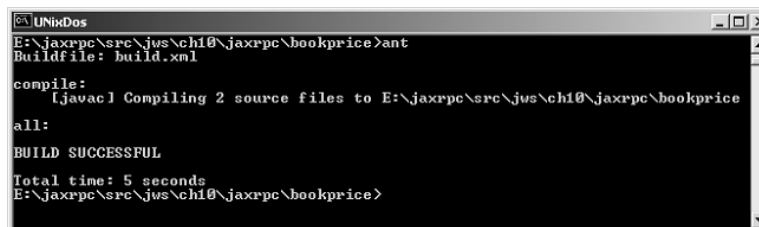
servantName="jws.ch10.jaxrpc.bookprice.BookPriceServiceImpl" />
  </service>
</configuration>
```

**Listing 10.9** `BookPriceService.xml`.

### Set Up the Environment and Compile the Source Code

Next you need to create a CLASSPATH environment that includes the JWS DP 1.0 class libraries for JAX-RPC and its supporting packages. To ensure this, make sure that the class libraries (\*.jar) in `%JWS DP_HOME%/common/lib/` and `%JWS DP_HOME%/common/endorsed/` are included.

Use `javac` and compile the source code of the remote interface (`BookPriceServiceIF.java`, in this example) and the implementation (`BookPriceServiceImpl.java`). You choose to use an Ant build script. After successful compilation, ensure that the compiled classes are available in the CLASSPATH. Figure 10.3 shows the compilation of the service definition classes.



**Figure 10.3** Building the services classes using Ant.

### ***Generate Server-Side Artifacts (Ties) and WSDL***

Using the `xrpcc` tool, generate the server-side artifacts and the WSDL document associated with the service. In a JWSDP 1.0 environment, the `xrpcc` tool requires `-server` and `-keep` as options for generating the server-side tie classes and the WSDL document.

Listing 10.10 is a code listing of the Ant script specific to creating server-side tie classes and the WSDL document associated with the service (`BookPriceService.xml`).

```
<!-- ===== Create Server Ties & WSDL ===== -->

<target name="server-ties">
  <property name="xrpcc" value="{JWSDP_HOME}/bin/xrpcc.bat"/>
  <exec executable="{xrpcc}">
    <arg line="-classpath ." />
    <arg line="-keep" />
    <arg line="-server" />
    <arg line="-d ." />
    <arg line="BookPriceServiceConfig.xml" />
  </exec>
</target>
```

**Listing 10.10** Ant script for using 'xrpcc' for generating server ties and WSDL.

To run `xrpcc` from the command line on Windows, use the following command:

```
xrpcc.bat -classpath %CLASSPATH% -keep -server
          -d . BookPriceServiceConfig.xml
```

To run `xrpcc` from the command line on UNIX, use the following command:

```
xrpcc.sh -classpath $CLASSPATH -keep -server
          -d . BookPriceServiceConfig.xml
```

The `xrpcc` tool generates the following tie classes, including the source files (not listed):

```
BookPriceServiceIF_GetBookPrice_RequestStruct.class
BookPriceServiceIF_GetBookPrice_ResponseStruct.class
BookPriceServiceIF_GetBookPrice_RequestStruct_SOAPSerializer.class
BookPriceServiceIF_GetBookPrice_ResponseStruct_SOAPSerializer.class
BookPriceService_SerializerRegistry.class
BookPriceServiceIF_Tie.class
```

## 480 Chapter 10

In addition to the previous tie classes, the tool also generates the WSDL document and the configuration properties file associated with the service as follows:

```
BookPriceService.wsdl
BookPriceService_Config.properties
```

The property file will look like the following:

```
# This file is generated by xrpscc.

port0.tie=jws.ch10.jaxrpc.bookprice.BookPriceServiceIF_Tie
port0.servant=jws.ch10.jaxrpc.bookprice.BookPriceServiceImpl
port0.name=BookPriceServiceIF
port0.wsdl.targetNamespace=http://jws.wiley.com/wsdl
port0.wsdl.serviceName=BookPriceService
port0.wsdl.portName=BookPriceServiceIFPort
portcount=1
```

In JWSDP1.0, in order to make the WSDL description accessible through a browser, one manual modification is required for the `BookPriceService_Config.properties` file to work. Although it is not recommended to modify the configuration properties file, it is possible to do so. Add the following line at the end of `BookPriceService_Config.properties`:

```
wsdl.location=/WEB-INF/BookPriceService.wsdl
```

With this line enabled, the WSDL now can be referenced by pointing the browser to `http://localhost:8080/bookpriceservice/jaxrpc?WSDL`.

### **Package and Deploy the Service**

Because JWSDP 1.0 currently enables deployment of the JAX-RPC-based service as a Web application (WAR), create a deployment descriptor (`web.xml`) and insert `BookPriceService_Config.properties` as a parameter value for the parameter name `configuration.file`.

Listing 10.11 is a code listing of the Web deployment descriptor (`web.xml`) for deploying `BookPriceService`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
```

**Listing 10.11** Deployment descriptor (`web.xml`) for deploying `BookPriceService`.

```
<web-app>
  <display-name>BookPriceService</display-name>
  <description>BookPriceService</description>
  <servlet>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <display-name>JAXRPCEndpoint</display-name>
    <description>Endpoint for Book Price Service</description>
    <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet
    </servlet-class>
    <init-param>
      <param-name>configuration.file</param-name>
      <param-value>/WEB-INF/BookPriceService_Config.properties
      </param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <url-pattern>/jaxrpc/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

**Listing 10.11** Deployment descriptor (web.xml) for deploying BookPriceService. (continued)

At this point, we are all set to compile and package the Web application. Create a Web application (WAR) file containing the classes including the deployment descriptors web.xml and configuration property file BookPriceService\_Config.properties.

Listing 10.12 is a sample code listing of an Ant script, which packages and deploys bookpriceservice.war in a JWSDP 1.0 environment (that is, a Tomcat /webapps directory).

```
<!-- ===== Package ===== -->
<target name="package">
  <delete dir="./WEB-INF" />
  <copy todir="./WEB-INF">
    <fileset dir=".">
      <include name="*.xml"/>
      <include name="*.wsdl"/>
      <include name="*.properties"/>
    </fileset>
  </copy>
</target>
```

**Listing 10.12** Ant script for packaging BookPriceService. (continues)

## 482 Chapter 10

```

    </fileset>
  </copy>
  <copy todir="./WEB-INF/classes">
    <fileset dir=".">
      <include name="**/*.class"/>
    </fileset>
  </copy>
  <jar jarfile="${JAXRPC_WORK_HOME}/bookpriceservice.war">
    <fileset dir="." includes="WEB-INF/**" />
  </jar>
  <copy todir="${JWSDP_HOME}/webapps">
    <fileset dir="${JAXRPC_WORK_HOME}">
      <include name="bookpriceservice.war"/>
    </fileset>
  </copy>
</target>

```

**Listing 10.12** Ant script for packaging BookPriceService. (continued)

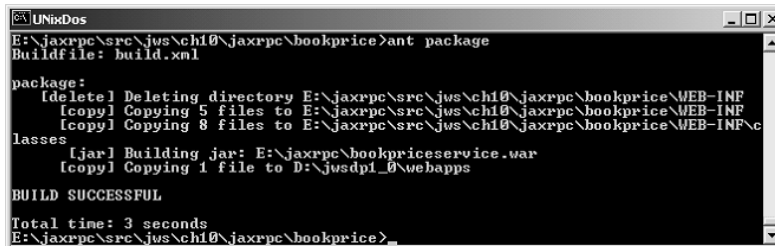
The successful running of the previous Ant script copies the `bookpriceservice.war` to the JWSDP1.0/Tomcat environment `/Webapps` directory and then restarts the Tomcat server. Figure 10.4 shows the successful packaging and deployment of `BookPriceService` in the Tomcat Web container.

### Test the Service Deployment and WSDL

To test the successful packaging and deployment of the service, run the following URL using a Web browser:

```
http://127.0.0.1:8080/bookpriceservice/jaxrpc/
```

If everything works successfully, the browser will display the page shown in Figure 10.5.



```

UNIXDos
E:\jaxrpc\src\jws\ch10\jaxrpc\bookprice>ant package
Buildfile: build.xml

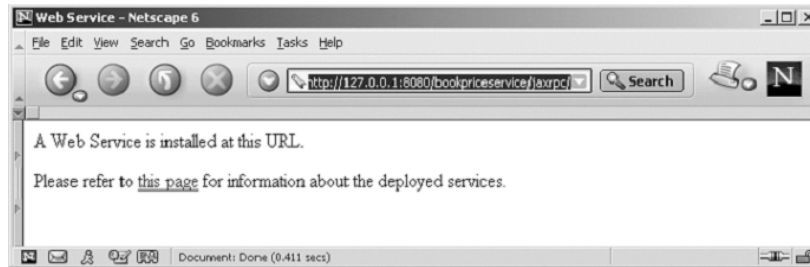
package:
[delete] Deleting directory E:\jaxrpc\src\jws\ch10\jaxrpc\bookprice\WEB-INF
[copy] Copying 5 files to E:\jaxrpc\src\jws\ch10\jaxrpc\bookprice\WEB-INF
[copy] Copying 8 files to E:\jaxrpc\src\jws\ch10\jaxrpc\bookprice\WEB-INF\c
lasses
[jar] Building jar: E:\jaxrpc\bookpriceservice.war
[copy] Copying 1 file to D:\jwsdp1_0\webapps

BUILD SUCCESSFUL

Total time: 3 seconds
E:\jaxrpc\src\jws\ch10\jaxrpc\bookprice>

```

**Figure 10.4** Packaging and deployment of `BookPriceService`.



**Figure 10.5** Browser displaying the installation of BookPriceService.

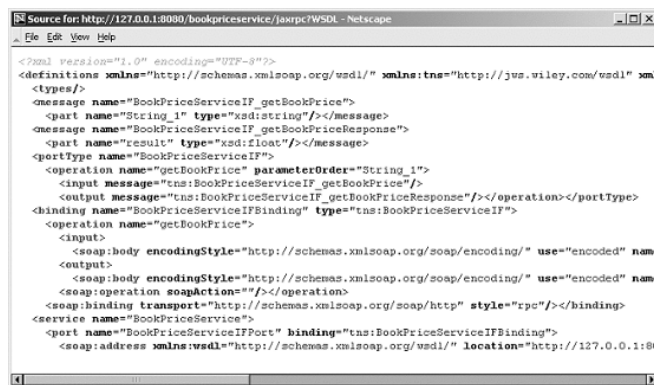
And, to display the WSDL document, run the following URL using a Web browser:

```
http://127.0.0.1:8080/bookpriceservice/jaxrpc?WSDL
```

If the WSDL generation is successful, the browser will display the page shown in Figure 10.6.

### Generating Client-Side Artifacts (Stubs)

Using the `xrpic` tool, generate the client-side stubs of the service. In a JWSDP 1.0 environment, the `xrpic` tool requires a `-client` option for generating the client-side stub classes. Ensure that the generated classes are copied to the directory `client-stubs`, which helps package the client stubs separately.



**Figure 10.6** Browser showing the WSDL of BookPriceService.

## 484 Chapter 10

---

The following is a code listing of the Ant script specific to creating client-side stub classes:

```
<!-- ===== Create Client Stubs ===== -->
<target name="client-stubs">
  <property name="xrpsc" value="{JWSDP_HOME}/bin/xrpsc.bat" />
  <exec executable="{xrpsc}">
    <arg line="-classpath ." />
    <arg line="-client" />
    <arg line="-d ./client-stubs" />
    <arg line="BookPriceServiceConfig.xml" />
  </exec>
</target>
```

This is the tool that generates the following stub classes in the `client-stubs` directory:

```
BookPriceService.class
BookPriceService_Impl.class
BookPriceService_SerializerRegistry.class
BookPriceServiceIF_GetBookPrice_RequestStruct.class
BookPriceServiceIF_GetBookPrice_ResponseStruct.class
BookPriceServiceIF_GetBookPrice_RequestStruct_SOAPSerializer.class
BookPriceServiceIF_GetBookPrice_ResponseStruct_SOAPSerializer.class
BookPriceServiceIF_Stub.class
```

Now, navigate to the `client-stubs` directory and, using the `jar` utility, create the `client-stubs.jar` file, including all the stub classes. Additionally, include the remote interface of the service `BookPriceServiceIF` class. Ensure that the `client-stubs.jar` file is in the `CLASSPATH` for developing service clients.

### Developing JAX-RPC Clients (BookPriceServiceClient)

As we discussed earlier, JAX-RPC 1.0 enables a JAX-RPC-based client to be implemented using three different client invocation models. To illustrate our `BookPriceService` client example, we will implement all three models of client implementation and walk through them in the following sections.

#### *Stub-Based Client*

In this client model, we will implement a standalone Java client that uses the stub classes (`client-stubs.jar`), which act as a proxy for invoking

remote methods. The client uses the command line argument service endpoint, which refers to the target service endpoint.

Listing 10.13 is a code listing of the stub-based service client (BookPriceServiceClient.java).

```
package jws.ch10.jaxrpc.bookprice.stubclient;

// Import the Stub Interface
import javax.xml.rpc.Stub;
import jws.ch10.jaxrpc.bookprice.BookPriceServiceIF;

public class BookPriceServiceClient {

// Main method
public static void main(String[] args) {

    try {

        if(args.length==0) {
            System.out.println("Usage:
            java jws.ch10.bookprice.stubclient.BookPriceServiceClient
                                SERVICE_ENDPOINT");
            return;
        }

// Obtain the Instance of the Stub Interface
BookPriceServiceIF_Stub stub = (BookPriceServiceIF_Stub)
    (new BookPriceService_Impl().getBookPriceServiceIFPort());

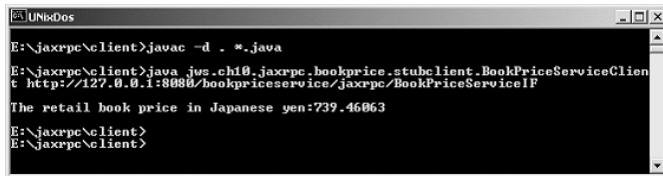
// Configure the stub setting required properties
// Setting the target service endpoint
stub._setProperty(
    javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
    args[0]);

// Execute the remote method
System.out.println ("\nThe retail book price in Japanese yen:"
    + stub.getBookPrice("JAX-RPC in 24 hours"));

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
}
```

**Listing 10.13** BookPriceServiceClient.java.

## 486 Chapter 10



```

E:\jaxrpc\client>javac -d . *.java
E:\jaxrpc\client>java jws.ch10.jaxrpc.bookprice.stubclient.BookPriceServiceClient
http://127.0.0.1:8080/bookpriceservice/jaxrpc/BookPriceServiceIF
The retail book price in Japanese yen:739.46063
E:\jaxrpc\client>
E:\jaxrpc\client>

```

**Figure 10.7** Output showing stub-based client invocation on `BookPriceService`.

Ensure that the `client-stubs.jar` and JAX-RPC API libraries are available in the CLASSPATH and that the JWSDP 1.0/Tomcat server is up and running. Then, compile the source code using `javac` and execute the client providing the service endpoint as the argument.

If everything works successfully, we see output like that shown in Figure 10.7.

### ***Dynamic Proxy-Based Client***

In the dynamic proxy client model, we will implement a standalone Java client that invokes a target service endpoint dynamically at runtime without using the local stub class. The `getPort` method on the `javax.xml.rpc.Service` interface enables a dynamic proxy to be created.

Listing 10.14 is a code listing of the dynamic proxy-based service client (`BookPriceServiceProxyClient.java`).

```

package jws.ch10.jaxrpc.bookprice.proxyclient;

// Import Service & ServiceFactory

import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
import jws.ch10.jaxrpc.bookprice.*;

import javax.xml.namespace.QName;
import java.net.URL;

public class BookPriceServiceProxyClient {

// Main method
public static void main(String[] args) {

try {

```

**Listing 10.14** `BookPriceServiceProxyClient.java`.

```
// WSDL location of the BookPriceService
String wsdlURL
    = "http://127.0.0.1:8080/bookpriceservice/jaxrpc?WSDL";

// WSDL namespace URI
String namespaceURI = "http://jws.wiley.com/wsdl";

// Service Name
String serviceName = "BookPriceService";

// Service port Name
String portName = "BookPriceServiceIFPort";

URL serviceWSDL = new URL(wsdlURL);

// Obtain an Instance of Service factory
ServiceFactory serviceFactory = ServiceFactory.newInstance();

// Create a service from Instance using WSDL
Service bookPriceService =
    serviceFactory.createService(serviceWSDL,
        new QName(namespaceURI, serviceName));

// Get the proxy object
BookPriceServiceIF bpProxy =
    (BookPriceServiceIF) bookPriceService.getPort
    (new QName(namespaceURI, portName), BookPriceServiceIF.class);

// Invoke the remote methods
System.out.println("\n\nThe retail book price
    in Japanese Yen : "
    + bpProxy.getBookPrice("JAX-RPC in 24 hours"));

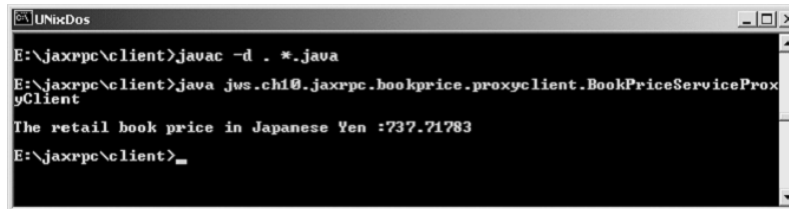
} catch (Exception ex) {
    ex.printStackTrace();
}
}
}
```

**Listing 10.14** BookPriceServiceProxyClient.java. (*continued*)

Ensure that the JAX-RPC API libraries are available in the CLASSPATH and that the JWSDP 1.0/Tomcat server is up and running and then compile the source code using javac.

If everything works successfully, you should see output like that shown in Figure 10.8.

## 488 Chapter 10



```

UNIXDos
E:\jaxrpc\client>javac -d . *.java
E:\jaxrpc\client>java jws.ch10.jaxrpc.bookprice.proxyclient.BookPriceServiceProxyClient
The retail book price in Japanese Yen :737.71783
E:\jaxrpc\client>_

```

**Figure 10.8** Output showing a dynamic proxy-based invocation on BookPriceService.

### *Dynamic Invocation Interface (DII) Client*

In the DII client model, the client discovers the target service dynamically at runtime and then invokes its methods. Listing 10.15 is a code listing of a DII client (BookPriceServiceDIIClient.java).

```

package jws.ch10.jaxrpc.bookprice.diiclient;

// Imports
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;
import java.net.URL;

public class BookPriceServiceDIIClient {

// Main method
public static void main(String[] args) {

// Service Name
String qService = "BookPriceService";

// Port Name
String qPort = "BookPriceServiceIF";

// Name space URI
String BODY_NAMESPACE_VALUE = "http://jws.wiley.com/wsdl";

// Encoding style

```

**Listing 10.15** BookPriceServiceDIIClient.java.

```
String ENCODING_STYLE_PROPERTY =
    "javax.xml.rpc.encodingstyle.namespace.uri";

// XML Schema
String NS_XSD = "http://www.w3.org/2001/XMLSchema";

// SOAP encoding URI
String URI_ENCODING = "http://schemas.xmlsoap.org/soap/encoding/";

try {
    String target_endpoint =
"http://127.0.0.1:8080/bookpriceservice/jaxrpc/BookPriceServiceIF";

    // Obtain an Instance of Service factory
    ServiceFactory sFactory =
        ServiceFactory.newInstance();

    // Create a Service
    Service service =
        sFactory.createService(new QName(qService));

    // Define a port
    QName port = new QName(qPort);

    // Create a Call object using the service
    Call call = service.createCall(port);

    // Set the target service endpoint
    call.setTargetEndpointAddress(target_endpoint);

    // Set properties - SOAP URI, Encoding etc.
    call.setProperty(Call.SOAPACTION_USE_PROPERTY,
        new Boolean(true));

    call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");

    call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);

    // Set Parameter In type as String
    QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");

    // Set Return value type as String
    QName QNAME_TYPE_FLOAT = new QName(NS_XSD, "float");
    call.setReturnType(QNAME_TYPE_FLOAT);
```

**Listing 10.15** BookPriceServiceDIIClient.java. (*continues*)

**490 Chapter 10**

```
// Set operations
call.setOperationName(new QName(BODY_NAMESPACE_VALUE,
                                "getBookPrice"));

// Set Parameters
call.addParameter("String_1", QName_TYPE_STRING,
                  ParameterMode.IN);

String[] BookName = {"JAX-RPC in 24 hours"};

// Invoke and obtain response
Object respObj = (Object) call.invoke(BookName);

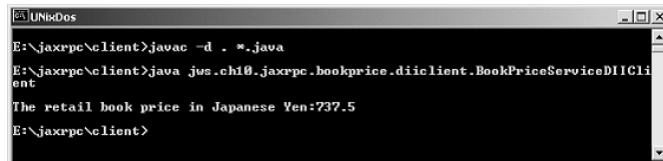
System.out.println("\nThe retail book price
                   in Japanese Yen:" + respObj.toString());

} catch (Exception ex) {
    ex.printStackTrace();
}
}
```

**Listing 10.15** BookPriceServiceDIIClient.java. (*continued*)

Ensure that the JAX-RPC API libraries are available in the CLASSPATH and that the JWS DP 1.0/Tomcat environment is up and running and then compile the source code using `javac`.

If everything works successfully, you should see output like that shown in Figure 10.9.



```
UNIKD0s
E:\jaxrpc\client>javac -d . *.java
E:\jaxrpc\client>java jws.ch10.jaxrpc.bookprice.diiclient.BookPriceServiceDIIClient
The retail book price in Japanese Yen:737.5
E:\jaxrpc\client>
```

**Figure 10.9** Output showing a DII-based invocation on `BookPriceService`.

## JAX-RPC in J2EE 1.4

---

The upcoming release of J2EE 1.4 platform specifications focuses on enabling J2EE components to participate in Web services. As a key requirement, it mandates the implementation of JAX-RPC 1.0 and EJB 2.1 specifications, which address the role of JAX-RPC in J2EE application components including EJBs. This means that all J2EE-compliant application servers will implement JAX-RPC, which allows exposing J2EE components as RPC-based Web services.

In EJB 2.1 specifications, it mandates the Stateless session EJBs to be exposed as Web services using a *Web Services Endpoint Interface*, which follows the same rules as a JAX-RPC service interface. This means the methods defined in the *Web Services Endpoint Interface* must be implemented in the bean implementation class. The EJB 2.1 deployment descriptor also introduces a new `<service-endpoint>` element, which contains the class name of the Web services endpoint interface. In the case of application exceptions, it is the responsibility of the container to map the exceptions to SOAP faults as per SOAP 1.1 specifications. At the time of this writing, the EJB 2.1 public draft specifies Web services endpoint interface for Stateless Session EJBs only.

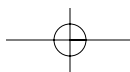
The introduction of JAX-RPC in J2EE environments enables J2EE components accessed as Web services using heterogeneous clients including both Java and non-Java applications. It also takes advantage of J2EE container services like transactions, application security, and so on.

## JAX-RPC Interoperability

---

A JAX-RPC Service provider can interoperate with any SOAP 1.1/WSDL 1.1-compliant service client and, similarly, a JAX-RPC Service client can interoperate with any SOAP 1.1/WSDL 1.1-compliant service provider.

To ensure JAX-RPC interoperability with other SOAP implementation providers, it is quite important to verify their compliance with specifications such as SOAP 1.1, WSDL 1.1, HTTP 1.1 transport, and XML Schema 1.0. To find out more information on JAX-RPC interoperability with other SOAP implementations, refer to Sun's SOAP Interoperability testing Web site at <http://soapinterop.java.sun.com>. (For more information on SOAP and WSDL interoperability and developing an interoperability scenario, refer to Chapter 6, "Creating .NET Interoperability.")



## 492 Chapter 10

---

### Summary

---

In this chapter, we have discussed how to develop JAX-RPC-based services and service clients for enabling Java Web services. We noted that JAX-RPC provides a RPC-based communication model between applications supporting industry standard messaging protocols. In general, we covered the role of JAX-RPC in Web services, JAX-RPC APIs and its programming model, JAX-RPC-supported mappings for Java and XML/WSDL, and the development of the JAX-RPC-based Web services applications.

In the next chapter, we will discuss how to describe, publish, and discover Web services using JAXR.

