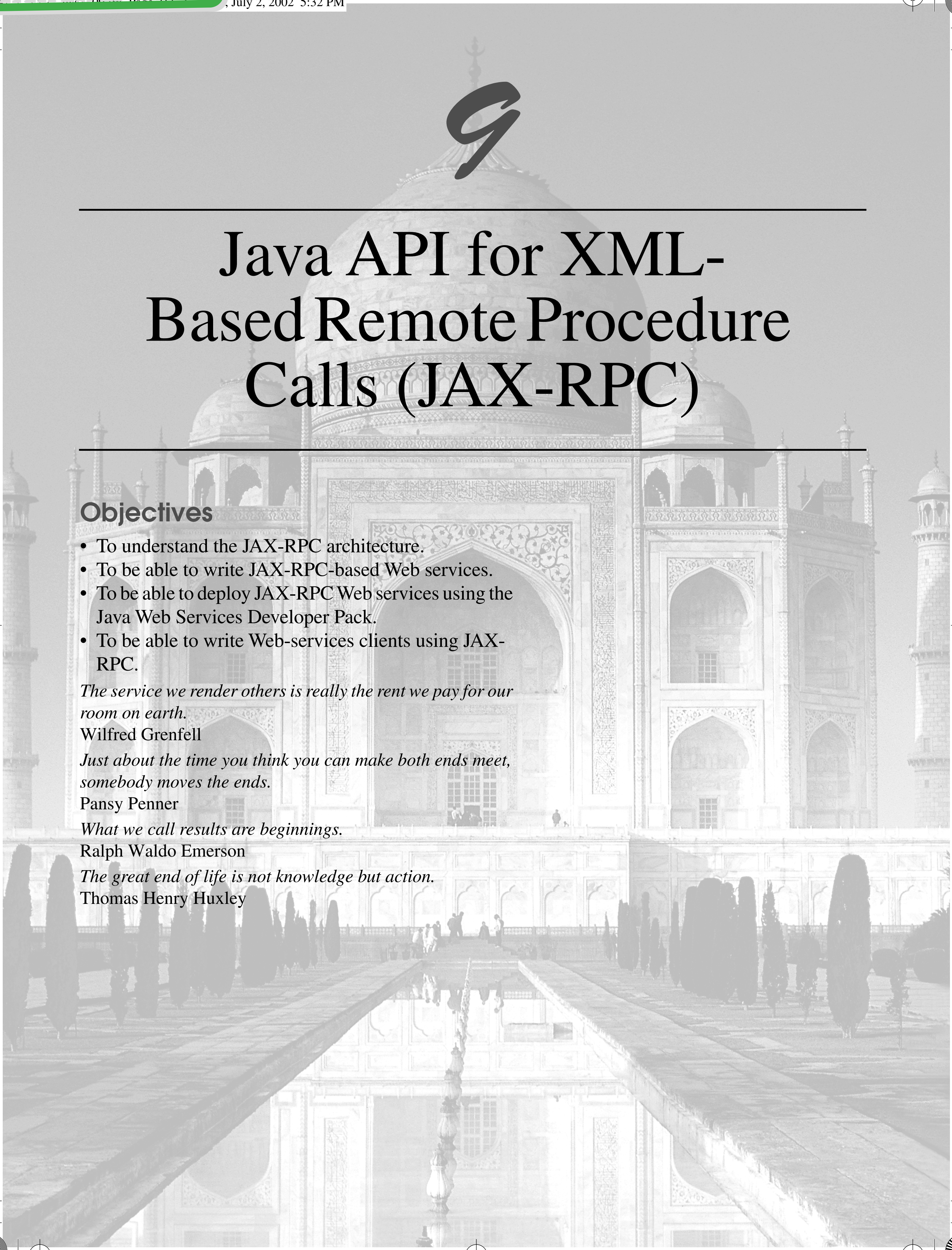
FOR PUBLIC RELEASE

, July 2, 2002 5:32 PM



9

Java API for XML-Based Remote Procedure Calls (JAX-RPC)

Objectives

- To understand the JAX-RPC architecture.
- To be able to write JAX-RPC-based Web services.
- To be able to deploy JAX-RPC Web services using the Java Web Services Developer Pack.
- To be able to write Web-services clients using JAX-RPC.

The service we render others is really the rent we pay for our room on earth.
Wilfred Grenfell

Just about the time you think you can make both ends meet, somebody moves the ends.
Pansy Penner

What we call results are beginnings.
Ralph Waldo Emerson

The great end of life is not knowledge but action.
Thomas Henry Huxley

jws1_09.fm Page 212 Tuesday, July 2, 2002 5:32 PM

Outline

9.1 Introduction

In Chapter 6, SOAP-Based Web Services Platforms, we introduced methods for creating Web services that use SOAP. This chapter and the next two discuss how to create Web services using the *Java Web Service Developer Pack* (*JWSDP*). JWSDP includes the *Java XML Pack*, *Tomcat Java Servlet and JavaServer Pages containers*, a registry server and the *Ant build tool*. The Java XML Pack provides the *Java API for XML-Based Remote Procedure Calls* (*JAX-RPC*). As of this writing, the current version of JWSDP is 1.0. Chapter 1, Introduction, discusses the installation of JWSDP.

RPC, which was developed in the 1980s, allows a procedural program (i.e., a program written in C or another procedural programming language) to call a function that resides on another computer as conveniently as if that function were part of the same program running on the same computer. A goal of RPC is to allow programmers to concentrate on the required tasks of an application; with RPC, the programmer can develop an application without concern for whether function calls are local or remote. RPC hides the network details that enable the application to communicate. It performs all the networking and *marshaling of data* (i.e., packaging of function arguments and return values for transmission over a network).

Web services evolved from the need to integrate applications written in many different languages with many different data formats. Whereas RPC requires the use of a single programming language and proprietary communications protocol, Web-services technology enables integration among many different languages and protocols. By

jws1_09.fm Page 213 Tuesday, July 2, 2002 5:32 PM

relying on XML—the de facto standard for marking up data—and HTTP—the de facto standard protocol for communication over the Web—SOAP provides such integration. JAX-RPC enables Java programmers to take advantage of these advances in distributed computing by providing a clean, simple API for creating and interacting with XML-based Web services.

JAX-RPC enables developers to build interoperable Web services and clients by providing a simple, RPC-oriented API that hides the details of the underlying SOAP communications and WSDL descriptions. A developer who builds JAX-RPC Web services does not need know in what programming language clients are written, because client requests are sent as XML messages that conform to the SOAP specification. Likewise, a developer who builds Web-services clients using JAX-RPC does not need to know the details of a Web service's underlying implementation, because the service's WSDL document specifies how to interact with the service.

9.2 JAX-RPC Overview

JAX-RPC provides a generic mechanism that enables developers to create and access Web services by using XML-based remote procedure calls. While such Web services can communicate by using any transport protocol, the current release of the JAX-RPC Reference Implementation (version 1.0) uses SOAP as the application protocol and HTTP as the transport protocol. Future versions likely will support other transport protocols as they become available.

When Web-service providers publish their Web services to XML registries (e.g., UDDI registries or ebXML registries), they may provide service interfaces or WSDL definitions for these services. The JAX-RPC specification defines a mapping of Java types (e.g., **`int`**, **`String`** and classes that adhere to the JavaBean design patterns) to WSDL definitions. When a client locates a service in an XML registry, the client retrieves the WSDL definition to get the service's interface definition. To access the service using Java, the client must transform the WSDL definitions to Java types. The ***`xrpcc`*** tool included in the JWSDP generates Java classes from the WSDL definitions.

Figure 9.1 shows the architecture of JAX-RPC. The service side contains a *JAX-RPC service runtime environment* and a *service end point*. The client side contains a *JAX-RPC client runtime environment* and a client application. The remote procedure calls use an XML-based protocol (e.g., SOAP) as the application protocol and an appropriate transport protocol (e.g., HTTP). The JAX-RPC client and service runtime environments send and process the remote procedure call and response, respectively. The JAX-RPC client runtime system creates a SOAP message to invoke the remote method, and the JAX-RPC service runtime system transforms the SOAP message to a Java method call, then dispatches the method call to the service end point. The JAX-RPC service runtime also translates the methods call's return value into a SOAP message and delivers that SOAP message to the client.

Before JAX-RPC was introduced, *Remote Method Invocation* (*RMI*) was the predominant RPC mechanism for Java. RMI allows Java programs to transfer complete Java objects over networks, using Java's object-serialization mechanism. Since RMI can be used to make remote procedure calls over the Internet, developers may wonder why they would need to use JAX-RPC, which seems to provide similar functionality.

jws1_09.fm Page 214 Tuesday, July 2, 2002 5:32 PM

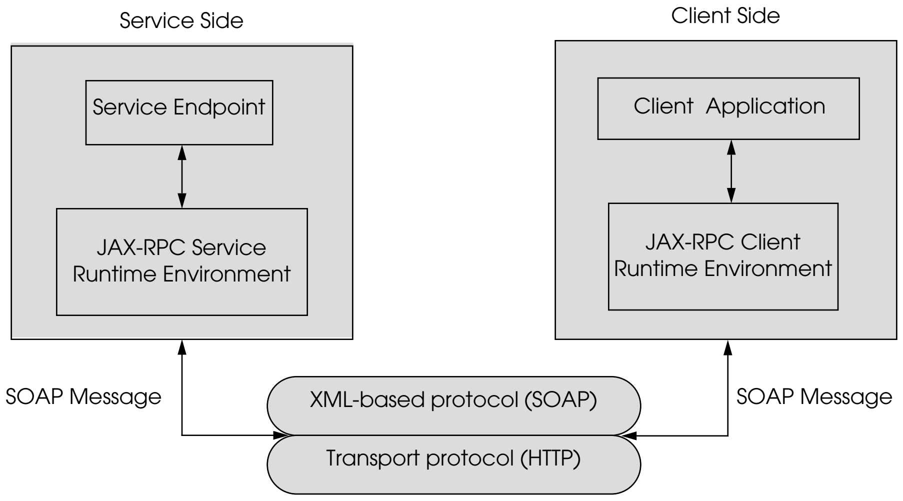

Service Side
Service Endpoint
JAX-RPC Service Runtime Environment
Client Side
Client Application
JAX-RPC Client Runtime Environment
SOAP Message
XML-based protocol (SOAP)
Transport protocol (HTTP)
SOAP Message

Fig. 9.1 JAX-RPC architecture.

As with RPC, both RMI and JAX-RPC handle the marshaling and unmarshaling of data across the network. RMI and JAX-RPC also provide APIs for transmitting and receiving data. The primary differences between RMI and JAX-RPC are as follows:

1. JAX-RPC enables applications to take advantage of SOAP's and WSDL's interoperability, which enables those applications to invoke Web services that execute on non-Java platforms, and non-Java applications to invoke Web services that execute on Java platforms. RMI supports only Java-to-Java distributed communication. The client needs only the WSDL to access the Web service. [*Note*: RMI/IIOP also provides interoperability with non-Java applications. However, JAX-RPC is easier to use. With several lines of code, the client can access Web services written in languages other than Java.]
2. RMI can transfer complete Java objects, while JAX-RPC is limited to a set of supported Java types, which we discuss in Section 9.3.1.

JAX-RPC hides the details of SOAP from the developer, because the JAX-RPC service/client runtime environments perform the mapping between remote method calls/return values and SOAP messages. The JAX-RPC runtime system also provides APIs for accessing Web services via static *stub*s (local objects that represent the remote services), dynamic proxies (objects that are generated during runtime) and for invoking Web services dynamically through the *Dynamic Invocation Interface* (*DII*). We discuss these APIs in detail in Section 9.3.5.

9.3 Simple Web Service: Vote Service

In this section, we present a simple JAX-RPC Web service that tallies votes for the users' favorite programming languages. The four major steps in this example are:

1. Defining a Web-service interface with methods that clients can invoke.

2. Writing a Java class that implements the interface [*Note*: By convention, the name of the service-implementation class is the same as that of the interface and ends with ***Impl***].
3. Deploying the service using the ***deploytool*** included with JWSDP 1.0.
4. Generating client-side stubs and writing the client application that interacts with the service.

Fig. 9.2 shows the structure of this example.

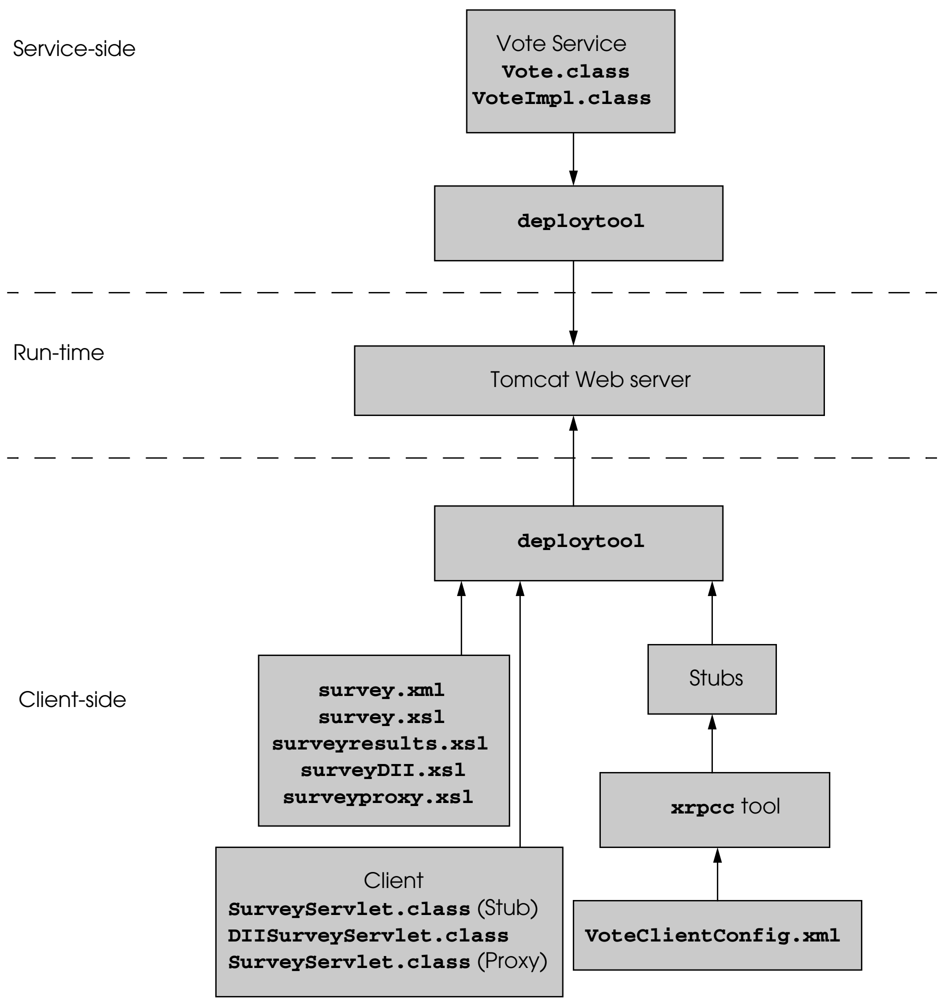


Fig. 9.2 Vote JAX-RPC Web service example structure.

jws1_09.fm Page 216 Tuesday, July 2, 2002 5:32 PM

Before providing the code for the example, we discuss the limited set of JAX-RPC-supported Java types.

9.3.1 JAX-RPC-Supported Java Types

JAX-RPC supports only a subset of Java types, because the data types transmitted by the remote procedure calls must map to SOAP XML data types. When a Web service receives a remote method call from a client, the JAX-RPC runtime service environment first transforms the XML representation of the call arguments to their corresponding Java types. (This process is known as *deserialization*.) The JAX-RPC runtime service environment then passes the Java representation of the call arguments to the service implementation for processing. After the call is processed, the JAX-RPC service runtime environment transforms the return object to an XML representation. (This process is known as *serialization*.) The XML representation of the return object (e.g., a SOAP message) is then sent back to the client. The serialization/deserialization process happens both at the client and at the service.

JAX-RPC supports Java primitive types and their corresponding wrapper classes. Figure 9.3 shows the mappings of Java primitive types and their wrapper classes to SOAP data types.

JAX-RPC supports a subset of standard Java classes as well. Figure 9.4 shows the mappings of this subset of standard Java classes to SOAP data types.

Java primitive types and their wrapper classes	XML elements
`boolean (Boolean)`	`xsd:boolean (soapenc:boolean)`
`byte (Byte)`	`xsd:byte (soapenc:byte)`
`double (Double)`	`xsd:double (soapenc:double)`
`float (Float)`	`xsd:float (soapenc:float)`
`int (Integer)`	`xsd:int (soapenc:int)`
`long (Long)`	`xsd:long (soapenc:long)`
`short (Short)`	`xsd:short (soapenc:short)`

Fig. 9.3 Mappings of Java primitive types and their wrapper classes to SOAP data types.

Standard Java classes	XML elements
`BigDecimal`	`xsd:decimal`
`BigInteger`	`xsd:integer`
`Calendar`	`xsd:dateTime`

Fig. 9.4 Mappings of standard Java classes to SOAP data types. (Part 1 of 2.)

Standard Java classes	XML elements
`Date`	`xsd:dateTime`
`String`	`xsd:string`

Fig. 9.4 Mappings of standard Java classes to SOAP data types. (Part 2 of 2.)

The JWSDP 1.0 final release also supports a set of classes that implement the **`java.util.Collection`** interface. These classes are: **`Vector`**, **`ArrayList`**, **`LinkedList`**, **`Stack`**, **`HashMap`**, **`HashTable`**, **`HashSet`**, **`Properties`**, **`TreeMap`** and **`TreeSet`**.

In addition to the aforementioned supported types, JAX-RPC supports objects of Java classes that satisfy the following conditions:

1. The class does not implement ***`java.rmi.Remote`***.
2. The class has a ***`public`*** default constructor.
3. The class's public fields are JAX-RPC-supported Java types.
4. Java classes may follow the JavaBean's *set* and *get* method-design patterns. Bean properties must be JAX-RPC-supported Java types.

Finally, Java arrays also can be used in JAX-RPC, provided that the member type of the array is one of the aforementioned JAX-RPC-supported Java types. JAX-RPC supports multidimensional Java arrays as well.

9.3.2 Defining the Vote Service Interface

The first step in the creation of a Web service with JAX-RPC is to define the remote interface that describes the *remote methods* through which the client interacts with the service. The following restrictions apply to the service interface definition:

1. The interface must extend **`java.rmi.Remote`**.
2. Each public method must include **`java.rmi.RemoteException`** in its **`throws`** clause. The **`throws`** clause may also include application-specific exceptions.
3. No declarations of constants are allowed.
4. All method parameters and return types must be JAX-RPC-supported Java types.

To create a remote interface, define an interface that extends interface **`java.rmi.Remote`**. Interface **`Remote`** is a *tagging interface*—it does not declare any methods, but identifies the interface as supporting remote method calls. Interface **`Vote`** (Fig. 9.5)—which extends interface **`Remote`** (line 9)—is the remote interface for our first JAX-RPC-based Web-service example. Line 12 declares method **`addVote`**, which clients can invoke to add votes for the users' favorite programming languages. Note that, although the **`Vote`** remote interface defines only one method, remote interfaces can declare multiple methods. A Web service must implement all methods declared in its remote interface.

jws1_09.fm Page 218 Tuesday, July 2, 2002 5:32 PM

```
// Fig. 9.5: Vote.java
// VoteService interface declares a method for adding a vote and
// for returning vote information.
package com.deitel.jws1.jaxrpc.service.vote;

// Java core packages
import java.rmi.*;

public interface Vote extends Remote {

    // obtain vote information from server
    public String addVote( String language ) throws RemoteException;
}
```

Fig. 9.5 **`Vote`** defines the service interface for the JAX-RPC Vote Web service.

9.3.3 Defining the Vote Service Implementation

After defining the remote interface, we define the service implementation. Class **`Vote-Impl`** (Fig. 9.6) is the Web-service end point that implements the **`Vote`** interface. The client interacts with an object of class **`VoteImpl`** by invoking method **`addVote`** of interface **`Vote`**. Method **`addVote`** enables the client to add a vote to the database and obtain tallies of previously recorded votes.

Class **`VoteImpl`** implements remote interface **`Vote`** and interface ***`ServiceLifecycle`*** (line 15). Interface **`ServiceLifecycle`** allows service implementations to perform initialization and termination processes, such as opening and closing database connections. We use a Cloudscape database[1] in this example to store the total number of votes for each programming language.

Lines 21–74 implement method **`init`** of interface **`ServiceLifecycle`** to set up access to a Cloudscape database. The JAX-RPC runtime system invokes method **`init`** when the service implementation class is instantiated. Lines 29–30 cast the parameter (**`context`**) of method **`init`** to **`ServletEndpointContext`**. Lines 33–34 invoke method **`getServletContext`** of class **`ServletEndpointContext`** to get the associated **`ServletContext`**. Lines 37–42 get the database driver and database name that are specified in the servlet's deployment descriptor (**`web.xml`**). Line 45 loads the class definition for the database driver. Line 48 declares and initializes **`Connection`** reference **`connection`** (package **`java.sql`**). The program initializes **`connection`** with the result of a call to **`static`** method **`getConnection`** of class **`DriverManager`**, which attempts to connect to the database specified by its URL argument. Lines 52–62 invoke **`Connection`** method **`prepareStatement`** to create SQL **`PreparedStatement`**s for updating the number of votes for the client's selected programming language and getting the vote count for each programming language.

Lines 77–111 implement method **`addVote`** of interface **`Vote`**. Line 83 sets the parameter of query **`sqlUpdate`** to the client-specified language. After setting the parameter for the **`PreparedStatement`**, the program calls method **`executeUpdate`** of interface **`PreparedStatement`** to execute the **`UPDATE`** operation. Line 89 calls method **`executeQuery`** of interface **`PreparedStatement`** to execute the **`SELECT`** operation.

1. We discuss the installation of the Cloudscape database in Chapter 1, Introduction.

jws1_09.fm Page 219 Tuesday, July 2, 2002 5:32 PM

`ResultSet results` stores the query results. Lines 93–98 iterate through the **`results`** and append them to a **`StringBuffer`**.

Lines 114–128 implement method **`destroy`** of interface **`ServiceLifecycle`**. Lines 118–120 close the prepared statements and the database connection.

```java
1  // VoteImpl.java
2  // VoteImpl implements the Vote remote interface to provide
3  // a VoteService remote object.
4  package com.deitel.jws1.jaxrpc.voteservice;
5
6  // Java core packages
7  import java.rmi.*;
8  import java.sql.*;
9
10 // Java XML packages
11 import javax.xml.rpc.server.*;
12 import javax.xml.rpc.JAXRPCException;
13 import javax.servlet.ServletContext;
14
15 public class VoteImpl implements ServiceLifecycle, Vote {
16
17    private Connection connection;
18    private PreparedStatement sqlUpdate, sqlSelect;
19
20    // set up database connection and prepare SQL statement
21    public void init( Object context )
22       throws JAXRPCException
23    {
24       // attempt database connection and
25       // create PreparedStatements
26       try {
27
28          // cast context to ServletEndpointContext
29          ServletEndpointContext endpointContext =
30             ( ServletEndpointContext ) context;
31
32          // get ServletContext
33          ServletContext servletContext =
34             endpointContext.getServletContext();
35
36          // get database driver from servlet context
37          String dbDriver =
38             servletContext.getInitParameter( "dbDriver" );
39
40          // get database name from servlet context
41          String voteDB =
42             servletContext.getInitParameter( "voteDB" );
43
44          // load database driver
45          Class.forName( dbDriver );
46
```

Fig. 9.6 **`VoteImpl`** defines the service implementation for the Vote JAX-RPC Web service. (Part 1 of 3.)

```java
         // connect to database
         connection = DriverManager.getConnection( voteDB );

         // PreparedStatement to increment vote total for a
         // specific language
         sqlUpdate =
            connection.prepareStatement(
               "UPDATE surveyresults SET vote = vote + 1 " +
               "WHERE name = ?"
            );

         // PreparedStatement to obtain surveyresults table's data
         sqlSelect =
            connection.prepareStatement( "SELECT name, vote " +
               "FROM surveyresults ORDER BY vote DESC"
            );

      } // end try

      // for any exception, throw an JAXRPCException to
      // indicate that the servlet is not currently available
      catch ( Exception exception ) {
         exception.printStackTrace();

         throw new JAXRPCException( exception.getMessage() );
      }

   } // end method init

   // implementation for interface Vote method addVote
   public String addVote( String name ) throws RemoteException
   {
      // get votes count from database and update it
      try {

         // set parameter in sqlUpdate
         sqlUpdate.setString( 1, name );

         // execute sqlUpdate statement
         sqlUpdate.executeUpdate();

         // execute sqlSelect statement
         ResultSet results = sqlSelect.executeQuery();
         StringBuffer voteInfo = new StringBuffer();

         // iterate ResultSet and prepare return string
         while ( results.next() ) {

            // append results to String voteInfo
            voteInfo.append( " " + results.getString( 1 ) );
            voteInfo.append( " " + results.getInt( 2 ) );
         }
```

Fig. 9.6 `VoteImpl` defines the service implementation for the Vote JAX-RPC Web service. (Part 2 of 3.)

jws1_09.fm Page 221 Tuesday, July 2, 2002 5:32 PM

```

         return voteInfo.toString();

      } // end try

      // handle database exceptions by returning error to client
      catch ( Exception exception ) {
         exception.printStackTrace();

         return exception.getMessage();
      }

   } // end method addVote

   // close SQL statements and database when servlet terminates
   public void destroy()
   {
      // attempt to close statements and database connection
      try {
         sqlUpdate.close();
         sqlSelect.close();
         connection.close();
      }

      // handle database exception
      catch ( Exception exception ) {
         exception.printStackTrace();
      }

   } // end method destroy

} // end class VoteImpl
```

Fig. 9.6 `VoteImpl` defines the service implementation for the Vote JAX-RPC Web service. (Part 3 of 3.)

9.3.4 Deploying the Vote Service

Once we have defined the service interface and implementation, the next step is to deploy the Web service. The JAX-RPC reference implementation provides the **deploytool** to deploy a JAX-RPC service onto Tomcat. The JAX-RPC reference implementation also provides the **xrpcc** tool to generate ties (server-side objects that represent the services), stubs and other service and client-side artifacts (such as a WSDL document). In this section, we discuss how to deploy the JAX-RPC service using the **deploytool**. Section 9.4.2 discuss how to deploy a JAX-RPC service via **xrpcc**.

The **deploytool** is a GUI utility that creates the WAR file and deploys the JAX-RPC service on Tomcat. The **deploytool** also generates the deployment descriptor (**web.xml**) and the service's WSDL document. Before starting the **deploytool**, we need to set two environment variables: **JAVA_HOME** and **JWSDP_HOME**. The **JAVA_HOME** points to the directory where the J2SE 1.3.1 or 1.4 is installed. The **JWSDP_HOME** points to the directory where the JWSDP 1.0 is installed. In our examples, the **JAVA_HOME** is set to **G:\j2sdk1.4.0** and the **JWSDP_HOME** is set to **G:\jwsdp-**

jws1_09.fm Page 222 Tuesday, July 2, 2002 5:32 PM

1_0. Then include the **bin** directories of the J2SE and the JWSDP to the front of the **PATH** environment variable. Before running the **deploytool**, start Tomcat. (In the command prompt terminal window, type **startup** to start Tomcat.) Type **deploytool** to start the deployment tool GUI. Once the **deploytool** is started, a **Set Tomcat Server** dialog asks for user name and password for using the Tomcat server, whose values were specified at installation time. Figure 9.7 is the **deploytool** initial window.

Click the **File** menu and select **New Web Application...** to create the Web application via the wizard. A **New Web Application Wizard - Introduction** dialog box appears. Read the instructions and click the **Next** button, which displays the **New Web Application Wizard - WAR File** dialog box. In the **WAR File Location** panel, enter the name and location where you would like to create the WAR file and enter a display name. In the **Contents** panel, click **Edit...** button to add **Vote.class**, **VoteImpl.class**, **RmiJdbc.jar** and **cloudclient.jar**. [*Note*: Compiling **Vote.java** and **VoteImpl.java** requires **jaxrpc-api.jar** and **servlet.jar**. Both JAR files are available at **%JWSDP_HOME%\common\lib** directory, where **JWSDP_HOME** is the JWSDP installation directory.] When adding the class files, be sure that the **Starting Directory** is set to the directory on your computer that contains the **com.deitel.jws1** package structure. The **deploytool** automatically places all class files into directory **classes** and all JAR files into directory **lib**. Figure 9.8 shows the result of editing the WAR file contents for this example.

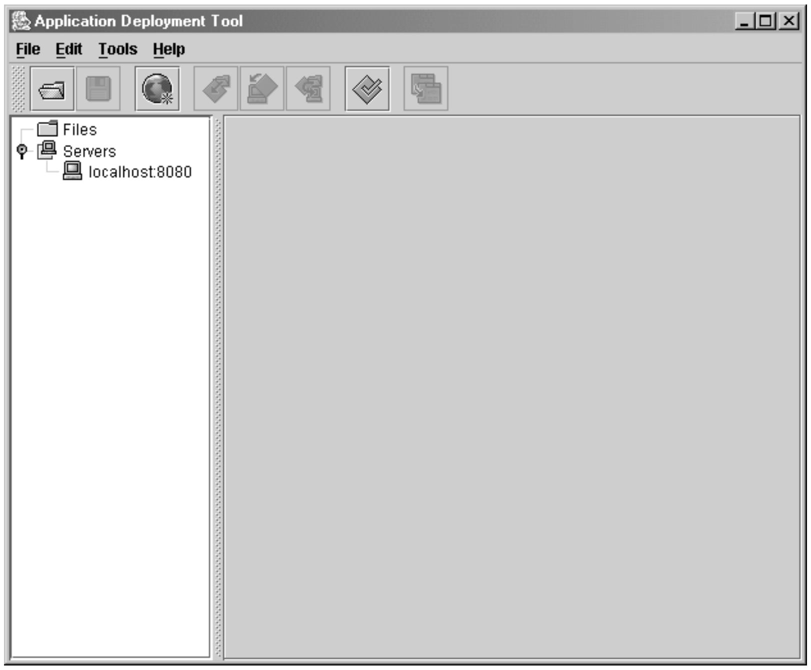
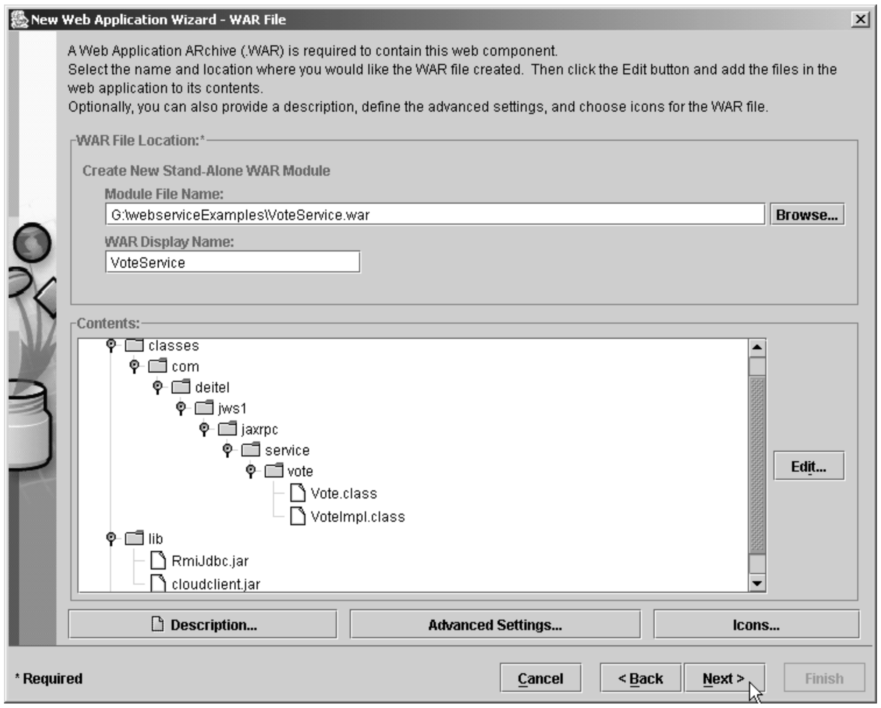


Fig. 9.7 The **deploytool** initial window.

jws1_09.fm Page 223 Tuesday, July 2, 2002 5:32 PM



A Web Application ARchive (.WAR) is required to contain this web component.
Select the name and location where you would like the WAR file created. Then click the Edit button and add the files in the web application to its contents.
Optionally, you can also provide a description, define the advanced settings, and choose icons for the WAR file.

WAR File Location:*
Create New Stand-Alone WAR Module
Module File Name: G:\webserviceExamples\VoteService.war
WAR Display Name: VoteService

Contents:
```
classes
  com
    deitel
      jws1
        jaxrpc
          service
            vote
              Vote.class
              VoteImpl.class
lib
  RmiJdbc.jar
  cloudclient.jar
```

Fig. 9.8 Creating WAR file.

In the next dialog box, **New Web Application Wizard - Choose Component Type**, select the **JAX-RPC Endpoint** radio button and check the **Context Parameters** check box, then click **Next** to continue. A dialog box **New Web Application Wizard - JAXRPC Default Settings** appears. In this dialog, enter the values as shown in Figure 9.9.

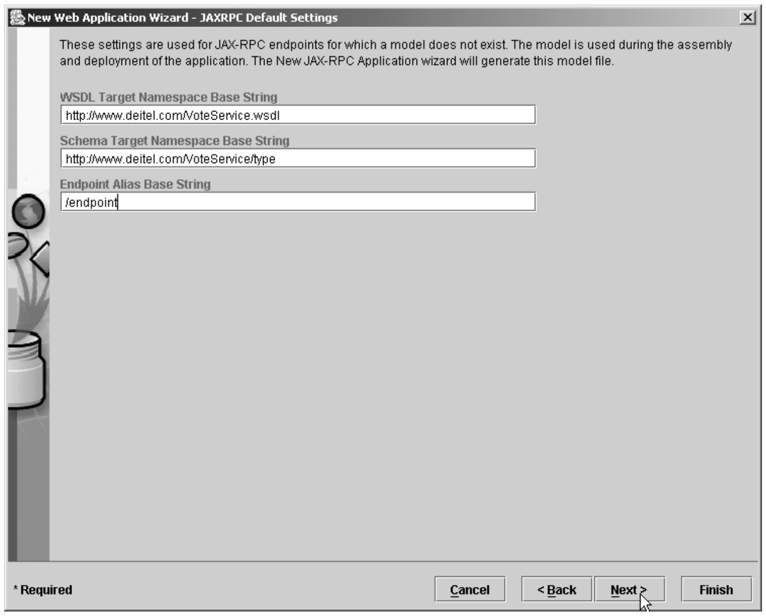
The next dialog box **New Web Application Wizard - JAX-RPC Endpoint** sets the service endpoint. Figure 9.10 shows the endpoint setting for the Vote service. Click **Next** to display the **New Web Application Wizard - JAX-RPC Model** dialog box. Select the **Use Default Model Setting** radio button and click **Next** to continue.

A **New Web Application Wizard - Context Parameters** dialog box appears next. In this dialog, we set the context parameters for accessing the database. Figure 9.11 shows the parameter names and values used in the Vote service.

Click **Next** to view the setting in dialog box **New Web Application Wizard - Review Settings**, then click **Finish** to confirm the settings. Figure 9.12 shows the Vote service Web application overview.

To deploy the Vote service, select the **Tools** menu and click **Deploy...**. Enter the Web context in the **Text Input Dialog**. Figure 9.13 shows the Web context for the Vote service. Click **OK** to deploy the service. A **Deployment Console** appears to show the deployment process. Click **Close** to close the console.

jws1_09.fm Page 224 Tuesday, July 2, 2002 5:32 PM



New Web Application Wizard - JAXRPC Default Settings

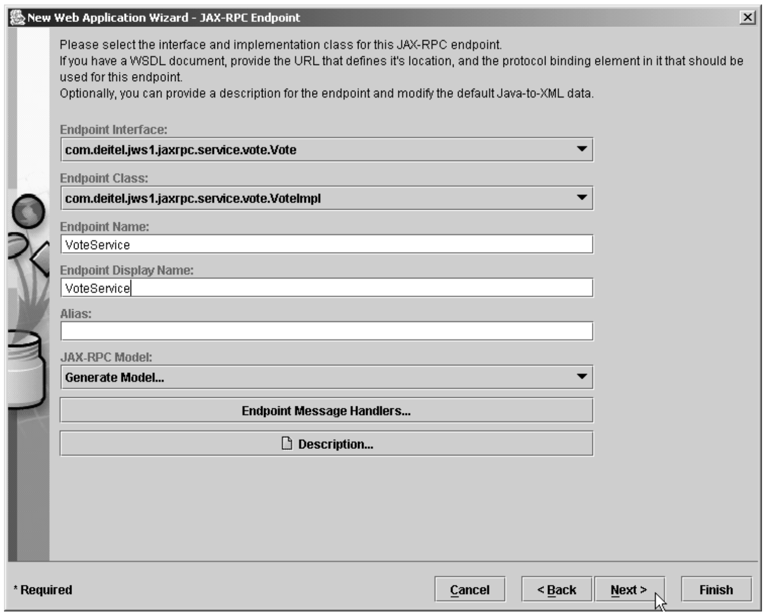
These settings are used for JAX-RPC endpoints for which a model does not exist. The model is used during the assembly and deployment of the application. The New JAX-RPC Application wizard will generate this model file.

WSDL Target Namespace Base String: http://www.deitel.com/VoteService.wsdl

Schema Target Namespace Base String: http://www.deitel.com/VoteService/type

Endpoint Alias Base String: /endpoint

Fig. 9.9 Default setting for the Vote service.



New Web Application Wizard - JAX-RPC Endpoint

Please select the interface and implementation class for this JAX-RPC endpoint.
If you have a WSDL document, provide the URL that defines it's location, and the protocol binding element in it that should be used for this endpoint.
Optionally, you can provide a description for the endpoint and modify the default Java-to-XML data.

Endpoint Interface: com.deitel.jws1.jaxrpc.service.vote.Vote

Endpoint Class: com.deitel.jws1.jaxrpc.service.vote.VoteImpl

Endpoint Name: VoteService

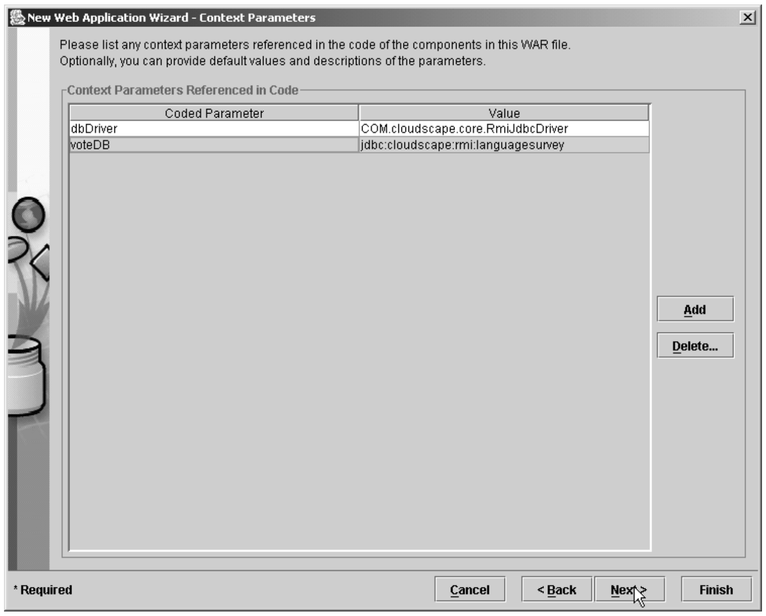
Endpoint Display Name: VoteService

Alias:

JAX-RPC Model: Generate Model...

Fig. 9.10 Endpoint setting for the Vote service.

jws1_09.fm Page 225 Tuesday, July 2, 2002 5:32 PM

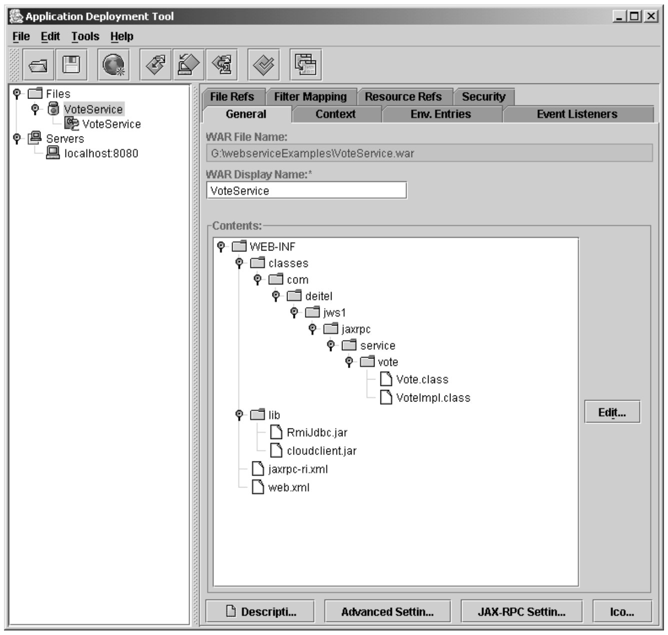


Please list any context parameters referenced in the code of the components in this WAR file.
Optionally, you can provide default values and descriptions of the parameters.

Context Parameters Referenced in Code

Coded Parameter	Value
dbDriver	COM.cloudscape.core.RmiJdbcDriver
voteDB	jdbc:cloudscape:rmi:languagesurvey

Fig. 9.11 Setting the servlet context parameters.



```
Files
  VoteService
    VoteService
Servers
  localhost:8080
```

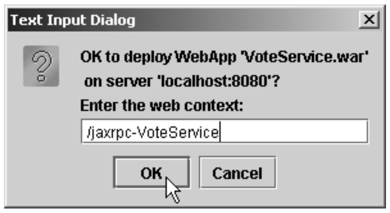
WAR File Name: G:\webserviceExamples\VoteService.war

WAR Display Name:* VoteService

Contents:

```
WEB-INF
  classes
    com
      deitel
        jws1
          jaxrpc
            service
              vote
                Vote.class
                VoteImpl.class
  lib
    RmiJdbc.jar
    cloudclient.jar
  jaxrpc-ri.xml
  web.xml
```

Fig. 9.12 Vote service Web application overview.



OK to deploy WebApp 'VoteService.war' on server 'localhost:8080'?
Enter the web context:
/jaxrpc-VoteService

Fig. 9.13 Specifying the Web context for the Vote Web service.

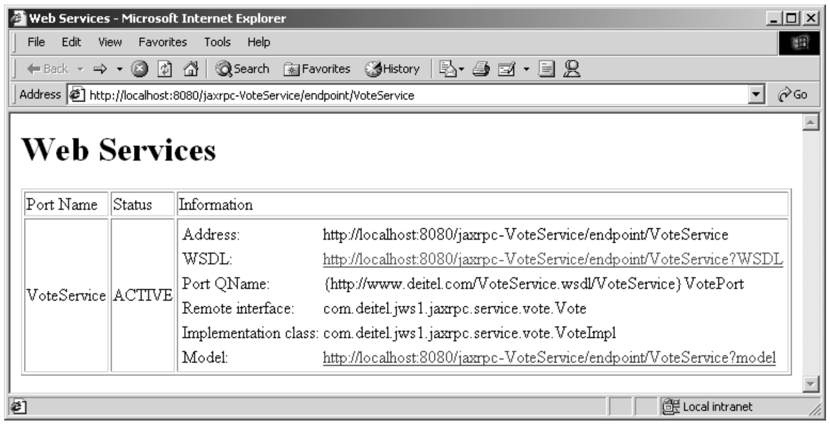
To verify whether the service has been installed correctly, open a Web browser and visit:

```
http://localhost:8080/jaxrpc-VoteService/endpoint/
   VoteService
```

Figure 9.14 shows that the Vote service has been deployed successfully.

9.3.5 Accessing the Vote Service

Once a Web service is deployed, a client can access that Web service via a static stub, the dynamic invocation interface (DII) or a dynamic proxy. We discuss each of these in detail in the following sections. Before we introduce service stubs, we demonstrate how to use the **`xrpcc`** tool for generating the stubs.


Web Services - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/jaxrpc-VoteService/endpoint/VoteService
Go
Local intranet

Web Services

Port Name	Status	Information	
VoteService	ACTIVE	Address:	http://localhost:8080/jaxrpc-VoteService/endpoint/VoteService
		WSDL:	http://localhost:8080/jaxrpc-VoteService/endpoint/VoteService?WSDL
		Port QName:	{http://www.deitel.com/VoteService.wsdl/VoteService} VotePort
		Remote interface:	com.deitel.jws1.jaxrpc.service.vote.Vote
		Implementation class:	com.deitel.jws1.jaxrpc.service.vote.VoteImpl
		Model:	http://localhost:8080/jaxrpc-VoteService/endpoint/VoteService?model

Fig. 9.14 Service endpoint verification.

jws1_09.fm Page 227 Tuesday, July 2, 2002 5:32 PM

***xrpcc* Tool**

The **`xrpcc`** tool generates a WSDL document or a remote-interface definition, depending on the command-line parameter. If we supply **`xrpcc`** with a remote-interface definition, it generates stubs, ties, a WSDL document and a server-configuration file used during deployment. If we supply **`xrpcc`** with a WSDL document, it generates stubs, ties, a server-configuration file and the remote-interface definition. Most users use an existing WSDL document to access a Web service. The stubs, ties, service and client-side artifacts are dictated by **`xrpcc`** options **`-client`**, **`-server`** and **`-both`**. We demonstrate the usage of all these options in the next several examples. In this example, we use **`xrpcc`** to generate the remote interface definition and client-side classes to access the Web service based on the WSDL document. In later examples, we use **`xrpcc`** to generate the WSDL document, based on the remote-interface definition.

Web service clients do not require access to the service interface and implementation classes. Clients need only the WSDL document that describes the service. The client specifies the location of the WSDL document in a configuration file and **`xrpcc`** generates the client-side classes that enable communication with the Web service. Figure 9.15 is the configuration file that is passed to the **`xrpcc`** tool to generate the client-side classes. Element **`wsdl`** (lines 4–5) specifies the WSDL-document URL and the fully qualified package name of the client-side classes. (The package name may differ from the package name of **`SurveyServlet`**.)

The command

```
xrpcc -d voteappclient -client -keep VoteClientConfig.xml
```

generates the client-side classes for the Vote service in directory **`voteappclient`**. **`VoteClientConfig.xml`** (Fig. 9.15) is the configuration file passed to **`xrpcc`**.

Using a Static Stub

In this section, we build a servlet-based client for the Vote Web service. Later sections introduce how to write Java applications to access Web services. Although **`SurveyServlet`** (Fig. 9.16) contains more than 200 lines of code, the core part that accesses the Vote service resides in lines 94–98. The remaining code uses DOM and JAXP to build and parse XML documents for interacting with the user.

`SurveyServlet` (Fig. 9.16) is a servlet that creates a survey form, invokes the Vote Web service to tally votes, and displays the survey results obtained from the Vote Web service.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <configuration
 3     xmlns = "http://java.sun.com/xml/ns/jax-rpc/ri/config">
 4     <wsdl location="http://localhost:8080/jaxrpc-VoteService/
endpoint/VoteService?WSDL"
 5        packageName = "vote">
 6     </wsdl>
 7  </configuration>
```

Fig. 9.15 Configuration file **`VoteClientConfig.xml`** to generate client-side classes from the WSDL document.

jws1_09.fm Page 228 Tuesday, July 2, 2002 5:32 PM

Method **doGet** (lines 28–75) displays the survey form. The **try** block (lines 37–68) performs the XML and XSL processing that results in an XHTML document containing a survey form. Creating a Document Object Model (DOM) tree from an XML document requires a **DocumentBuilder** parser object. Lines 41–42 obtain a **DocumentBuilderFactory**. Lines 45–46 obtain a **DocumentBuilder** parser object that enables the program to create a **Document** object tree in which the XML document elements are represented as **Element** objects. Line 54 invokes method **parse** of class **DocumentBuilder** to parse **survey.xml**. Lines 57–59 create an **InputStream** that will be used by the XSL transformation processor to read the XSL file. The response is created by the XSL transformation performed in method **transform** (lines 173–206).

```
1  // Fig. 9.16: SurveyServlet.java
2  // A Web-based survey that invokes a Web service
3  // from a servlet.
4  package com.deitel.jws1.jaxrpc.client.stub;
5
6  // Java core packages
7  import java.io.*;
8  import java.util.*;
9
10 // Java extension packages
11 import javax.servlet.*;
12 import javax.servlet.http.*;
13 import javax.xml.rpc.*;
14 import javax.xml.parsers.*;
15 import javax.xml.transform.*;
16 import javax.xml.transform.dom.*;
17 import javax.xml.transform.stream.*;
18
19 // third-party packages
20 import org.w3c.dom.*;
21
22 // xrpcc generated stub packages
23 import vote.*;
24
25 public class SurveyServlet extends HttpServlet {
26
27    // display survey form
28    protected void doGet( HttpServletRequest request,
29       HttpServletResponse response )
30          throws ServletException, IOException
31    {
32       // setup response to client
33       response.setContentType( "text/html" );
34       PrintWriter out = response.getWriter();
35
36       // get XML document and transform for browser client
37       try {
38
```

Fig. 9.16 **SurveyServlet** uses a static stub to access the Vote Web service. (Part 1 of 5.)

```
         // get DocumentBuilderFactory for creating
         // DocumentBuilder (i.e., an XML parser)
         DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

         // get DocumentBuilder for building DOM tree
         DocumentBuilder builder =
            factory.newDocumentBuilder();

         // open InputStream for XML document
         InputStream xmlStream =
            getServletContext().getResourceAsStream(
               "/survey.xml" );

         // create Document based on input XML file
         Document surveyDocument = builder.parse( xmlStream );

         // open InputStream for XSL document
         InputStream xslStream =
            getServletContext().getResourceAsStream(
               "/survey.xsl" );

         // transform XML document using XSLT
         transform( surveyDocument, xslStream, out );

         // flush and close PrinterWriter
         out.flush();
         out.close();

      } // end try

      // catch XML parser exceptions
      catch( Exception exception ) {
         exception.printStackTrace();
      }

   } // end method doGet

   // process survey response
   protected void doPost( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      // setup response to client
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // read current survey response
      String name = request.getParameter(
         "favoriteLanguage" );

```

Fig. 9.16 `SurveyServlet` uses a static stub to access the Vote Web service. (Part 2 of 5.)

jws1_09.fm Page 230 Tuesday, July 2, 2002 5:32 PM

```java
      // attempt to process vote and display current results
      try {

         // get stub and connect to Web service's endpoint
         Vote_Stub stub = ( Vote_Stub )
            ( new VoteService_Impl().getVotePort() );

         // get vote information from server
         String result = stub.addVote( name );

         StringTokenizer voteTokens =
            new StringTokenizer( result );

         // get DocumentBuilderFactory for creating
         // DocumentBuilder (i.e., an XML parser)
         DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

         // get DocumentBuilder for building DOM tree
         DocumentBuilder builder =
            factory.newDocumentBuilder();

         // create Document (empty DOM tree)
         Document resultDocument = builder.newDocument();

         // generate XML from result and append to Document
         Element resultElement = generateXML(
            voteTokens, resultDocument );
         resultDocument.appendChild( resultElement );

         // open InputStream for XSL document
         InputStream xslStream =
            getServletContext().getResourceAsStream(
               "/surveyresults.xsl" );

         // transform XML document using XSLT
         transform( resultDocument, xslStream, out );

         // flush and close PrintWriter
         out.flush();
         out.close();

      } // end try

      // catch connection and XML parser exceptions
      catch ( Exception exception ) {
         exception.printStackTrace();
      }

   }  // end method doPost

```

Fig. 9.16 **SurveyServlet** uses a static stub to access the Vote Web service. (Part 3 of 5.)

jws1_09.fm Page 231 Tuesday, July 2, 2002 5:32 PM

```
   // generate XML representation of vote information
   private Element generateXML( StringTokenizer voteTokens,
      Document document )
   {
      // create root element
      Element root = document.createElement( "surveyresults" );

      Element language;
      Element name;
      Element vote;

      // create language element for each language
      while ( voteTokens.hasMoreTokens() ) {
         language = document.createElement( "language" );
         name = document.createElement( "name" );
         name.appendChild( document.createTextNode(
            voteTokens.nextToken() ) );

         language.appendChild( name );
         vote = document.createElement( "vote" );
         vote.appendChild( document.createTextNode(
            voteTokens.nextToken() ) );
         language.appendChild( vote );
         root.appendChild( language );
      }

      return root; // return root element

   } // end method generateXML

   // transform XML document using XSLT InputStream
   // and write resulting document to PrintWriter
   private void transform( Document document,
      InputStream xslStream, PrintWriter output )
   {
      // transform XML to XHTML
      try {

         // create DOMSource for source XML document
         Source xmlSource = new DOMSource( document );

         // create StreamSource for XSLT document
         Source xslSource = new StreamSource( xslStream );

         // create StreamResult for transformation result
         Result result = new StreamResult( output );

         // create TransformerFactory to obtain Transformer
         TransformerFactory transformerFactory =
            TransformerFactory.newInstance();

```

Fig. 9.16 `SurveyServlet` uses a static stub to access the Vote Web service. (Part 4 of 5.)

jws1_09.fm Page 232 Tuesday, July 2, 2002 5:32 PM

```
192          // create Transformer for performing XSL transformation
193          Transformer transformer =
194             transformerFactory.newTransformer( xslSource );
195
196          // perform transformation and deliver content to client
197          transformer.transform( xmlSource, result );
198
199       } // end try
200
201       // handle exception when transforming XML document
202       catch( TransformerException exception ) {
203          exception.printStackTrace( System.err );
204       }
205
206    } // end method transform
207
208 } // end class SurveyServlet
```

Fig. 9.16 **`SurveyServlet`** uses a static stub to access the Vote Web service. (Part 5 of 5.)

Method **`transform`** takes three arguments—the XML **`Document`** to which the XSL transformation will be applied, the **`InputStream`** that reads the XSL file and the **`PrintWriter`** to which the results should be written. Line 180 creates a **`DOMSource`** that represents the XML document. This **`DOMSource`** serves as the source of the XML to transform. Line 183 creates a **`StreamSource`** for the XSL file. Line 186 creates a **`StreamResult`** for the **`PrintWriter`** to which the results of the XSL transformation are written. Lines 189–190 create a **`TransformerFactory`**, which enables the program to obtain the **`Transformer`** object that applies the XSL transformation. Lines 193–194 invoke **`TransformerFactory`** method **`newTransformer`** to create a **`Transformer`**. This method receives a **`StreamSource`** argument that represents the XSL (e.g., **`xslSource`**). Line 197 invokes **`Transformer`** method **`transform`** to perform the XSL transformation on the **`DOMSource`** object **`xmlSource`** and write the result to **`StreamResult`** object **`result`**. Lines 202–204 catch a **`TransformerException`** if a problem occurs when creating the **`TransformerFactory`**, creating the **`Transformer`** or performing the transformation.

Method **`doPost`** (lines 78–139) gets the client-specified programming language (lines 87–88). Lines 94–95 then connects to the Vote Web service. Once the connection is done, line 98 invokes the **`addVote`** method of the Vote Web service stub. To access a Web service via a stub, we need to obtain the stub object. The **`xrpcc`** tool generates the client-side stub of the Vote Web service—**`Vote_Stub`**. When **`xrpcc`** generates the stub, it uses the following convention: *serviceinterface*_**`Stub`**. Lines 94–95 get the **`Vote_Stub`** by invoking method **`getVotePort`** of class **`VoteService_Impl`**. The generated **`Vote_Stub`** class implements the **`javax.xml.rpc.Stub`** interface. Class **`VoteService_Impl`** is the **`xrpcc`**-generated implementation of the **`Vote`** service. When **`xrpcc`** generates the service implementation, it uses the naming convention *servicename*_**`Impl`**, where *servicename* is the service name specified in the **`xrpcc`** configuration file.

Line 98 invokes method **`addVote`** of the **`stub`** object to process the votes. Method **`addVote`** places a vote for the given language and returns a tally of previously recorded

jws1_09.fm Page 233 Tuesday, July 2, 2002 5:32 PM

votes for each language. Class **`SurveyServlet`** transforms the return value of method **`addVote`** to an XML representation. Method **`generateXML`** creates the XML representation of the vote information (lines 142–169). We pass two arguments to this method—the **`StringTokenizer`** that contains vote information and an empty **`Document`** tree. The complete vote information is placed in a **`surveyresults`** element (created at line 146). Lines 153–164 append elements for the individual votes for a programming language to the **`surveyresults`** element as children. Lines 154–155 use **`Document`** method **`createElement`** to create elements **`language`** and **`name`**. Lines 156–157 use **`Document`** method **`createTextNode`** to specify the language name in the **`name`** element, and **`Element`** method **`appendChild`** to append the text to element **`name`**. Line 159 appends element **`name`** as a child of element **`language`** with **`Element`** method **`appendChild`**. Similar operations are performed for the votes for a language. Line 164 appends each **`language`** element to the **`root`** element (**`surveyresults`**).

Once the program gets the XML representation of the vote information, line 118 appends **`Element resultElement`** (returned by method **`generateXML`**) to the **`Document`** object—**`resultDocument`**. Lines 121–123 create an **`InputStream`** that will be used by the XSL transformation processor to read the XSL file. Line 126 invokes method **`transform`** to create the response.

Class **`SurveyServlet`** requires one XML file that contains a list of programming languages for the survey, and two XSL files that transform XML files to XHTML responses. Method **`doGet`** requires **`survey.xml`** (Fig. 9.17) and **`survey.xsl`** (Fig. 9.18). Method **`doPost`** requires **`surveyresults.xsl`** (Fig. 9.19).

Figure 9.18 contains **`survey.xsl`**, which transforms **`survey.xml`** into an XHTML form. The names of the five elements in the XML document (**`survey.xml`**) are placed in the resulting XHTML form. Line 23 places the survey's **`title`** (line 1 in **`survey.xml`**) in the document's **`title`** element. Line 27 places the survey's **`question`** (line 2 in **`survey.xml`**) in a paragraph at the beginning of the document's **`body`** element. Lines 30–38 generate a form to be included in the document's **`body`** element. Line 31 uses XSLT element **`for-each`**, which applies the contents of the element to each of the nodes selected by attribute **`select`**, to include the name of each programming language in the form.

Line 32 uses XSLT element **`sort`** to sort the nodes by the field specified in attribute **`select`**, in the order specified in attribute **`order`** (either **`ascending`** or **`descending`**). In this example, we sort the nodes by attribute **`name`**, in **`ascending`** order. Lines 33–35 add each **`language`** element specified in the XML document (lines 3–7 in **`survey.xml`**) to the form.

```
1  <survey title = "Your Favorite Language"
2     question = "Choose your favorite programming language:">
3     <language name = "C"/>
4     <language name = "C++"/>
5     <language name = "Java"/>
6     <language name = "VB"/>
7     <language name = "Python"/>
8  </survey>
```

Fig. 9.17 **`survey.xml`** contains a list of programming languages for the survey.

jws1_09.fm Page 234 Tuesday, July 2, 2002 5:32 PM

```
<?xml version = "1.0"?>

<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
   version = "1.0">

<xsl:output method = "xml" omit-xml-declaration = "no"
   indent = "yes" doctype-system =
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtc"
   doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>

<!-- survey.xsl                                  -->
<!-- XSL document that rransforms XML into XHTML -->

<!-- specify the root of the XML document -->
<!-- that references this stylesheet      -->
<xsl:template match = "survey">

   <html xmlns = "http://www.w3.org/1999/xhtml">

   <head>

      <!-- obtain survey title from survey element -->
      <title><xsl:value-of select = "@title"/></title>
   </head>

   <body>
      <p><xsl:value-of select = "@question"/></p>

      <!-- create form -->
      <form method = "post" action = "/jaxrpc-VoteClient/Survey">
         <xsl:for-each select = "language">
            <xsl:sort select = "@name" order = "ascending"/>
            <input type = "radio" name = "favoriteLanguage"
               value = "{@name}"><xsl:value-of select = "@name"/>
            </input><br/>
         </xsl:for-each>
         <br/><input type = "Submit"/>
      </form>

   </body>

   </html>

</xsl:template>

</xsl:stylesheet>
```

Fig. 9.18 XSL style sheet (**survey.xsl**) that transforms **survey.xml** into an XHTML document.

Figure 9.19 contains **surveyresults.xsl**, which transforms the voting results to XHTML. Lines 30–34 use XSLT elements **for-each** to display the survey results in **ascending** order by **name**.

```
<?xml version = "1.0"?>

<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
   version = "1.0">

<xsl:output method = "xml" omit-xml-declaration = "no"
   indent = "yes" doctype-system =
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtc"
   doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>

<!-- survey.xsl                                   -->
<!-- XSL document that rransforms XML into XHTML -->

<!-- specify the root of the XML document -->
<!-- that references this stylesheet      -->
<xsl:template match = "surveyresults">

   <html xmlns = "http://www.w3.org/1999/xhtml">

   <head>

      <!-- obtain survey name from survey element -->
      <title>Survey Results</title>
   </head>

   <body>
      <h1>Vote Information:</h1>

      <!-- create result list -->
      <xsl:for-each select = "language">
         <p><xsl:value-of select = "name"/>:
            <xsl:value-of select = "vote"/>
         </p>
      </xsl:for-each>

   </body>

   </html>

</xsl:template>

</xsl:stylesheet>
```

Fig. 9.19 XSL style sheet (**`surveyresults.xsl`**) that transforms **`surveyresults.xml`** into an XHTML document.

To compile **`SurveyServlet`**, make sure that **`jaxrpc-api.jar`**; **`jaxrpc-ri.jar`**; **`servlet.jar`**; **`dom.jar`**; **`sax.jar`** and **`jaxp-api.jar`** are included in the classpath. These JAR files are located in directories **`%JWSDP_HOME%\common\lib`** and **`%JWSDP_HOME%\common\endorsed`**, where **`JWSDP_HOME`** is the home directory of the JWSDP installation.

To deploy the client using the **`deploytool`**, click the **File** menu and select **New Web Application...**. In the **New Web Application Wizard - WAR File** dialog box,

jws1_09.fm Page 236 Tuesday, July 2, 2002 5:32 PM

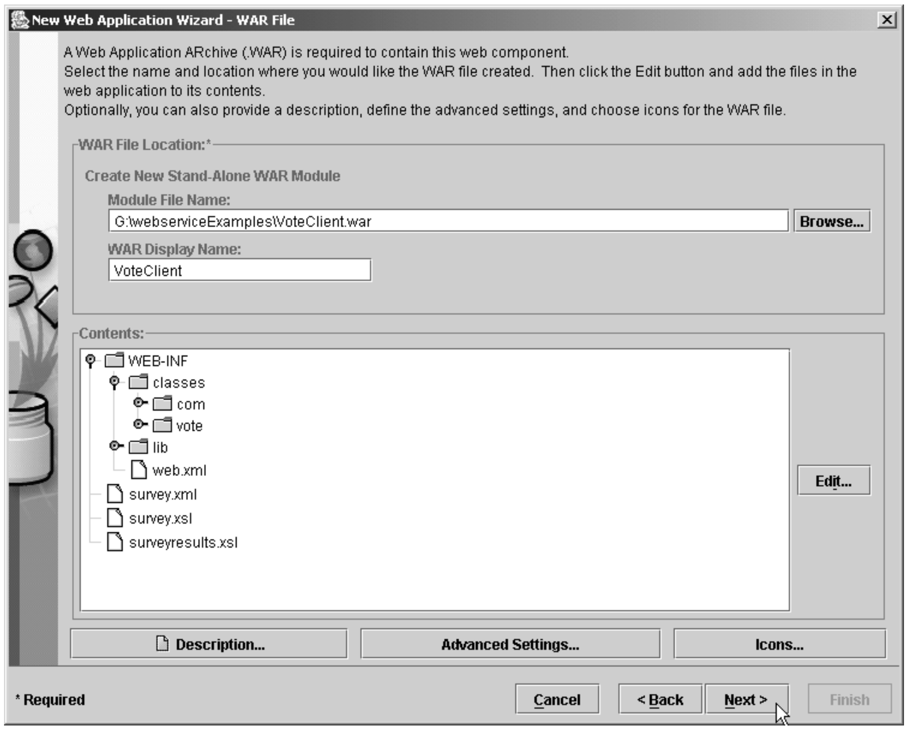
specify the **WAR File Location** and **Contents** as shown in Fig. 9.20. Directory **`classes`** contains **`SurveyServlet.class`** and client-side classes generated by **`xrpcc`**. The WAR contents also should include **`survey.xml`** (Fig. 9.17), **`survey.xsl`** (Fig. 9.18) and **`surveyresults.xsl`** (Fig. 9.19).

In the next dialog box, **New Web Application Wizard - Choose Component Type**, select the **Servlet** radio button and check the **Aliases** check box, then click **Next** to display the **New Web Application Wizard - Component General Properties** dialog box. Figure 9.21 specifies the client Web application component properties.

Click **Next** (Fig. 9.21) to display dialog box **New Web Application Wizard - Aliases**. Click **Add** to add aliases for **`SurveyServlet`**. Figure 9.22 shows the aliases for **`SurveyServlet`**.

Click **Next** to display the **New Web Application Wizard - Review Setting** window. Click **Finish** to close the review window. Figure 9.23 shoes the general information of the **`VoteClient`** Web application.

To deploy the client application, select the **Tools** menu and click **Deploy...**. The **Text Input Dialog** (Fig. 9.24) prompts for the Web context to create for the servlet.


New Web Application Wizard - WAR File

A Web Application ARchive (.WAR) is required to contain this web component.
Select the name and location where you would like the WAR file created. Then click the Edit button and add the files in the web application to its contents.
Optionally, you can also provide a description, define the advanced settings, and choose icons for the WAR file.

WAR File Location:*

Create New Stand-Alone WAR Module

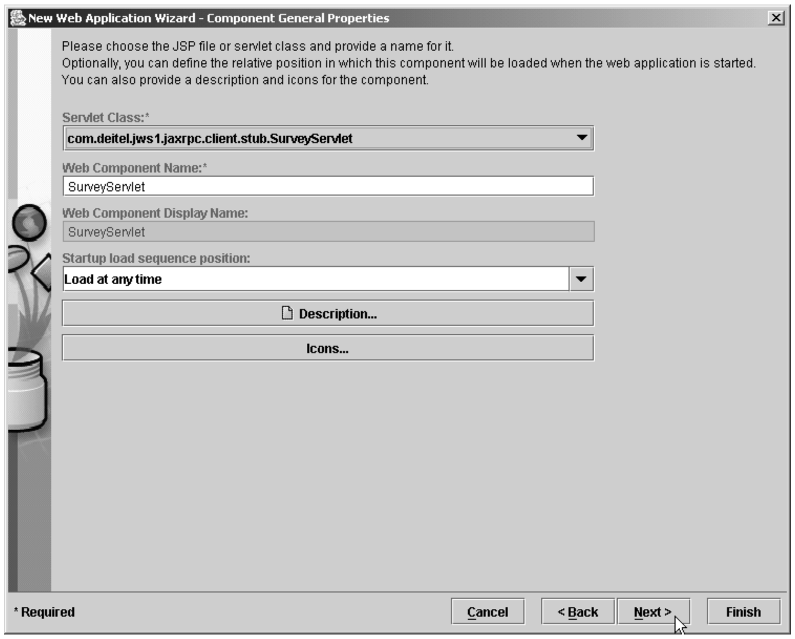
Module File Name: G:\webserviceExamples\VoteClient.war

WAR Display Name: VoteClient

Contents:

```
WEB-INF
  classes
    com
    vote
  lib
  web.xml
survey.xml
survey.xsl
surveyresults.xsl
```

Fig. 9.20 Creating the WAR file for the Vote client Web application.


New Web Application Wizard - Component General Properties

Please choose the JSP file or servlet class and provide a name for it.
Optionally, you can define the relative position in which this component will be loaded when the web application is started.
You can also provide a description and icons for the component.

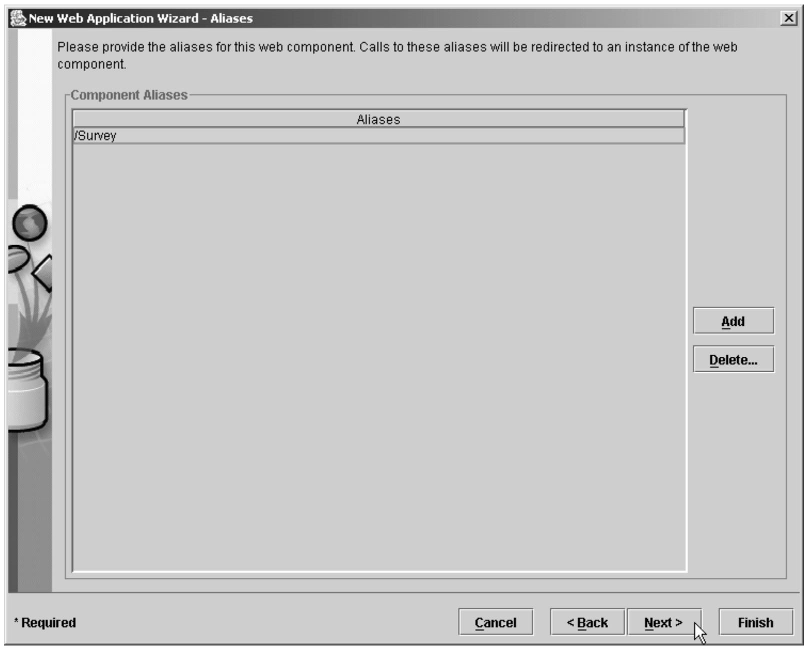
Servlet Class:* com.deitel.jws1.jaxrpc.client.stub.SurveyServlet

Web Component Name:* SurveyServlet

Web Component Display Name: SurveyServlet

Startup load sequence position: Load at any time

Fig. 9.21 Specifying the Vote client Web application component properties.


New Web Application Wizard - Aliases

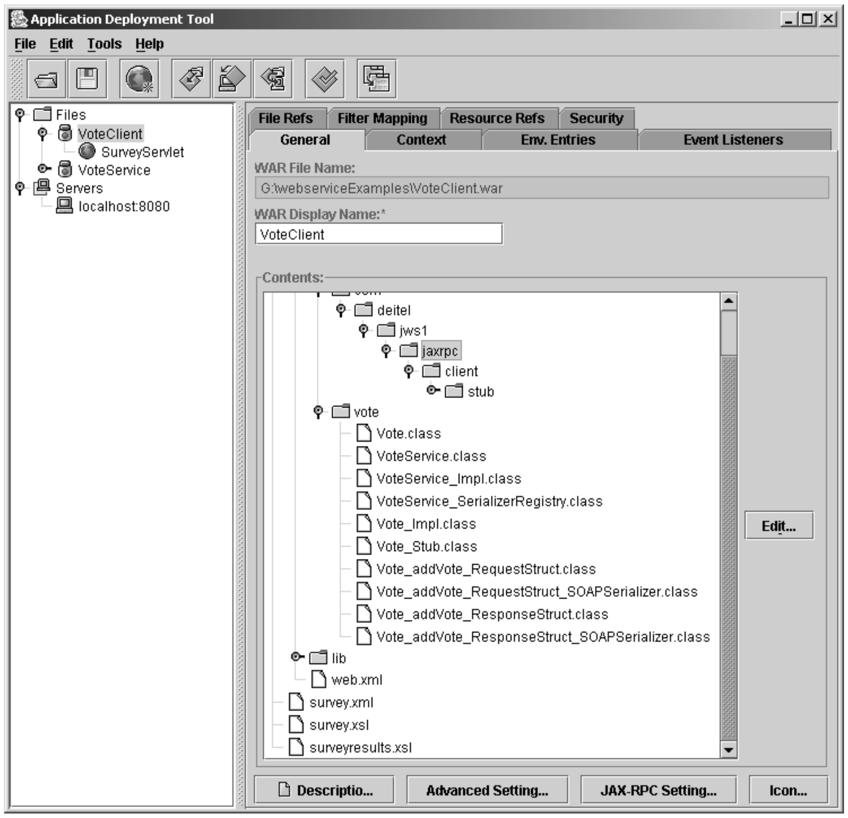
Please provide the aliases for this web component. Calls to these aliases will be redirected to an instance of the web component.

Component Aliases

Aliases
/Survey

Fig. 9.22 Setting aliases for `SurveyServlet`.

jws1_09.fm Page 238 Tuesday, July 2, 2002 5:32 PM



```
Files
  VoteClient
    SurveyServlet
  VoteService
Servers
  localhost:8080
```

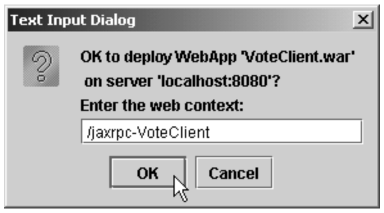
WAR File Name: G:\webserviceExamples\VoteClient.war

WAR Display Name:* VoteClient

Contents:

```
      deitel
        jws1
          jaxrpc
            client
              stub
    vote
      Vote.class
      VoteService.class
      VoteService_Impl.class
      VoteService_SerializerRegistry.class
      Vote_Impl.class
      Vote_Stub.class
      Vote_addVote_RequestStruct.class
      Vote_addVote_RequestStruct_SOAPSerializer.class
      Vote_addVote_ResponseStruct.class
      Vote_addVote_ResponseStruct_SOAPSerializer.class
  lib
  web.xml
survey.xml
survey.xsl
surveyresults.xsl
```

Fig. 9.23 **`VoteClient`** Web application general information.



OK to deploy WebApp 'VoteClient.war' on server 'localhost:8080'?

Enter the web context: /jaxrpc-VoteClient

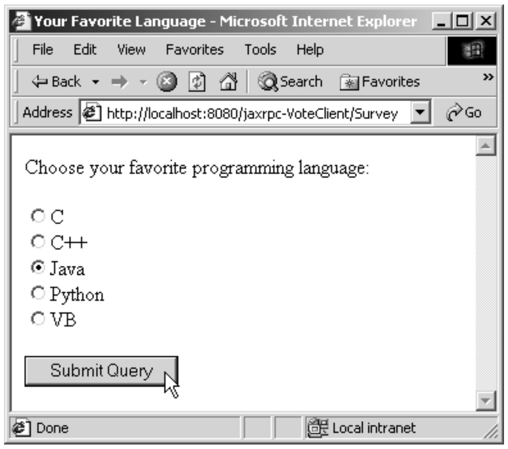
Fig. 9.24 Specifying the Web context for the Vote client.

To start the client, open a Web browser and visit:

```
http://localhost:8080/jaxrpc-VoteClient/Survey
```

Figure 9.25 shows the **`VoteClient`** Web application survey form and results.

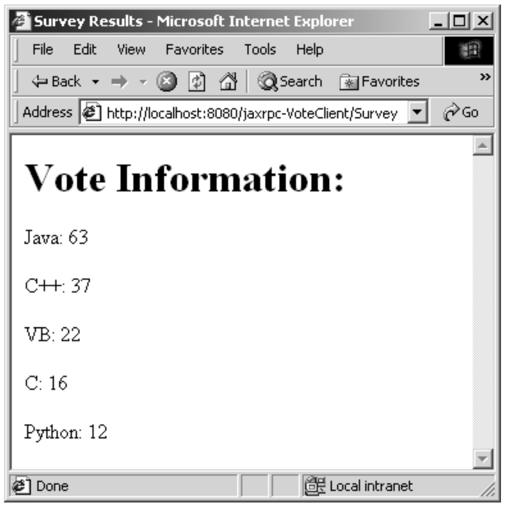
jws1_09.fm Page 239 Tuesday, July 2, 2002 5:32 PM


Your Favorite Language - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/jaxrpc-VoteClient/Survey
Go
Done
Local intranet

Choose your favorite programming language:

( ) C
( ) C++
(•) Java
( ) Python
( ) VB

Submit Query


Survey Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/jaxrpc-VoteClient/Survey
Go
Done
Local intranet

Vote Information:

Java: 63

C++: 37

VB: 22

C: 16

Python: 12

Fig. 9.25 `VoteClient` Web application survey form and results.

Using the Dynamic Invocation Interface

The *Dynamic Invocation Interface* (*DII*) enables Web-service clients to call methods on Web services without knowing the service's stub information. DII clients can build method calls dynamically, based on the WSDL document of the service. To use DII to invoke a Web service, clients must have the following information in advance:

1. Web-service endpoint.
2. Interface name of the Web service.
3. Name of the method call.
4. A list of the parameters.
5. Return type of the method call.

jws1_09.fm Page 240 Tuesday, July 2, 2002 5:32 PM

6. Service target namespace, which defines the namespaces used in the WSDL document.
7. Type namespace to register the serializers/deserializers.
8. SOAP action, which indicates the intent of the SOAP request (an empty-string value indicates that the HTTP request contains the SOAP request's intent).
9. Encoding style, which defines the serialization rules used to deserialize the SOAP message.

The Web service's WSDL document provides this information.

Next, we introduce how to invoke a Web service via DII. Using DII is more complicated than using static stubs. However, DII clients are more flexible than are static-stub clients, because DII clients can specify the remote procedure calls' properties (such as Web-service names, remote-method input parameters, and remote-method return types) at runtime.

`DIISurveyServlet` (Fig. 9.26) creates a survey form in method **`doGet`**, forwards the survey result to the Vote Web service and displays the vote information obtained from the Vote Web service in method **`doPost`**. Method **`doGet`** is almost identical to the method **`doGet`** of **`SurveyServlet`** (Fig. 9.16), except that lines 70–72 read **`surveyDII.xsl`** (Fig. 9.27) rather than **`survey.xsl`**.

```
 1  // Fig. 9.26: DIISurveyServlet.java
 2  // A Web-based survey that invokes the Web service
 3  // from a servlet using DII.
 4  package com.deitel.jws1.jaxrpc.client.dii;
 5
 6  // Java core packages
 7  import java.io.*;
 8  import java.util.*;
 9
10  // Java extension packages
11  import javax.servlet.*;
12  import javax.servlet.http.*;
13  import javax.xml.parsers.*;
14  import javax.xml.transform.*;
15  import javax.xml.transform.dom.*;
16  import javax.xml.transform.stream.*;
17
18  // Java XML RPC packages
19  import javax.xml.rpc.*;
20  import javax.xml.namespace.QName;
21  import javax.xml.rpc.encoding.*;
22
23  // JWSDP reference implementation
24  import com.sun.xml.rpc.client.*;
25  import com.sun.xml.rpc.client.dii.*;
26  import com.sun.xml.rpc.encoding.*;
27  import com.sun.xml.rpc.encoding.soap.*;
28  import com.sun.xml.rpc.soap.streaming.*;
29
```

Fig. 9.26 **`DIISurveyServlet`** invokes the Web service via DII. (Part 1 of 6.)

```
// third-party packages
import org.w3c.dom.*;

public class DIISurveyServlet extends HttpServlet {

   // servlet attributes
   private String surveyText;
   private String pageTitle;
   private String submissionIdentifier;

   // display survey form
   protected void doGet( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      // setup response to client
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // get XML document and transform for browser client
      try {

         // get DocumentBuilderFactory for creating
         // DocumentBuilder (i.e., an XML parser)
         DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

         // get DocumentBuilder for building DOM tree
         DocumentBuilder builder =
            factory.newDocumentBuilder();

         // open InputStream for XML document
         InputStream xmlStream =
            getServletContext().getResourceAsStream(
               "/survey.xml" );

         // create Document based on input XML file
         Document surveyDocument = builder.parse( xmlStream );

         // open InputStream for XSL document
         InputStream xslStream =
            getServletContext().getResourceAsStream(
               "/surveyDII.xsl" );

         // transform XML document using XSLT
         transform( surveyDocument, xslStream, out );

         // flush and close PrinterWriter
         out.flush();
         out.close();

      } // end try

```

Fig. 9.26 **DIISurveyServlet** invokes the Web service via DII. (Part 2 of 6.)

jws1_09.fm Page 242 Tuesday, July 2, 2002 5:32 PM

```
      // catch XML parser exceptions
      catch( Exception exception ) {
         exception.printStackTrace();
      }

   } // end method doGet

   // process survey response
   protected void doPost( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      // setup response to client
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // read current survey response
      String name = request.getParameter(
          "favoriteLanguage" );

      // read service endpoint
      String endpoint = request.getParameter( "endpoint" );

      // attempt to process vote and display current results
      try {

         // QName for "xsd:string"
         QName xmlString = new QName(
            "http://www.w3.org/2001/XMLSchema", "string" );

         // QName for service port
         QName servicePort = new QName(
            "http://www.deitel.com/VoteService.wsdl/VoteService",
            "VotePort" );

         // QName for service name
         QName serviceName = new QName( "VoteService" );

         // get Service object from SUN's Service implementation
         Service service =
            ServiceFactory.newInstance().createService(
               serviceName );

         // create Call object
         Call call = service.createCall();
         call.setPortTypeName( servicePort );
         call.setTargetEndpointAddress( endpoint );

         // set call properties
         call.setProperty( Call.SOAPACTION_USE_PROPERTY,
            new Boolean( true ) );
         call.setProperty( Call.SOAPACTION_URI_PROPERTY, "" );
```

Fig. 9.26 **`DIISurveyServlet`** invokes the Web service via DII. (Part 3 of 6.)

```
         String ENCODING_STYLE_PROPERTY =
            "javax.xml.rpc.encodingstyle.namespace.uri";
         call.setProperty( ENCODING_STYLE_PROPERTY,
            "http://schemas.xmlsoap.org/soap/encoding/" );

         // set call operation name and its input and output
         QName addVoteOperation = new QName(
            "http://www.deitel.com/VoteService.wsdl/VoteService",
            "addVote" );
         call.setOperationName( addVoteOperation );
         call.addParameter( "String_1", xmlString,
            ParameterMode.IN );
         call.setReturnType( xmlString );
         Object[] callInputs =
            new Object[] { name };

         // invoke addVote method
         String callOutput = ( String ) call.invoke(
            addVoteOperation, callInputs );

         StringTokenizer voteTokens =
            new StringTokenizer( callOutput );

         // get DocumentBuilderFactory for creating
         // DocumentBuilder (i.e., an XML parser)
         DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

         // get DocumentBuilder for building DOM tree
         DocumentBuilder builder =
            factory.newDocumentBuilder();

         // create Document (empty DOM tree)
         Document resultDocument = builder.newDocument();

         // generate XML from result and append to Document
         Element resultElement = generateXML(
            voteTokens, resultDocument );
         resultDocument.appendChild( resultElement );

         // open InputStream for XSL document
         InputStream xslStream =
            getServletContext().getResourceAsStream(
               "/surveyresults.xsl" );

         // transform XML document using XSLT
         transform( resultDocument, xslStream, out );

         // flush and close PrintWriter
         out.flush();
         out.close();

      } // end try
```

Fig. 9.26 **`DIISurveyServlet`** invokes the Web service via DII. (Part 4 of 6.)

```

      // return error page on exception
      catch ( Exception exception ) {
         exception.printStackTrace();
         out.println( "<title>Error</title>" );
         out.println( "</head>" );
         out.println( "<body><p>Error occurred: " );
         out.println( exception.getMessage() );
         out.println( "</p></body></html>" );
         out.close();
      }

   }  // end method doPost

    // generate vote information XML representation
   private Element generateXML( StringTokenizer voteTokens,
      Document document )
   {
      // create root element
      Element root = document.createElement( "surveyresults" );

      Element language;
      Element name;
      Element vote;

      // create language element for each language
      while ( voteTokens.hasMoreTokens() ) {
         language = document.createElement( "language" );
         name = document.createElement( "name" );
         name.appendChild( document.createTextNode(
            voteTokens.nextToken() ) );

         language.appendChild( name );
         vote = document.createElement( "vote" );
         vote.appendChild( document.createTextNode(
            voteTokens.nextToken() ) );
         language.appendChild( vote );
         root.appendChild( language );
      }

      return root; // return root element

   } // end method generateXML

   // transform XML document using XSLT InputStream
   // and write resulting document to PrintWriter
   private void transform( Document document,
      InputStream xslStream, PrintWriter output )
   {
      // transform XML to XHTML
      try {

```

Fig. 9.26 **`DIISurveyServlet`** invokes the Web service via DII. (Part 5 of 6.)

jws1_09.fm Page 245 Tuesday, July 2, 2002 5:32 PM

```
240          // create DOMSource for source XML document
241          Source xmlSource = new DOMSource( document );
242
243          // create StreamSource for XSLT document
244          Source xslSource = new StreamSource( xslStream );
245
246          // create StreamResult for transformation result
247          Result result = new StreamResult( output );
248
249          // create TransformerFactory to obtain Transformer
250          TransformerFactory transformerFactory =
251             TransformerFactory.newInstance();
252
253          // create Transformer for performing XSL transformation
254          Transformer transformer =
255             transformerFactory.newTransformer( xslSource );
256
257          // perform transformation and deliver content to client
258          transformer.transform( xmlSource, result );
259
260       } // end try
261
262       // handle exception when transforming XML document
263       catch( TransformerException exception ) {
264          exception.printStackTrace( System.err );
265       }
266
267    } // end method transform
268
269 } // end class DIISurveyServlet
```

Fig. 9.26 **`DIISurveyServlet`** invokes the Web service via DII. (Part 6 of 6.)

Method **`doPost`** (lines 91–200) implements support for DII clients. Lines 110–111 create a ***`QName`*** for the XML representation of Java **`String`** object. Class **`QName`** (in package ***`javax.xml.namespace`***) represents a data type with a qualified namespace in XML (e.g., **`xsd:string`**, where **`xsd`** is the namespace URI for the **`QName`** and **`string`** is the data type).

Lines 114–119 create two **`QName`**s that specify the service port and service name. Lines 122–124 create a service with the specified service name by invoking **`static`** method **`newInstance`** of class **`ServiceFactory`** to get an instance of **`ServiceFactory`** object, and by invoking method **`createService`** of class **`ServiceFactory`** to create a **`Service`** instance.

To invoke a method dynamically, line 127 creates a **`Call`** object, which enables dynamic invocation of a service port by invoking method **`createCall`** of interface **`Service`**. Line 128 invokes method **`setPortTypeName`** of interface **`Call`** to set the service port. Method **`setPortTypeName`** takes one argument that specifies the **`QName`** of the service port. Line 129 invokes method **`setTargetEndpointAddress`** of interface **`Call`** to set the URI of the service end point.

Lines 132–138 invoke method **`setProperty`** of interface **`Call`** to set the properties of the call. Lines 132–133 indicate that the **`SOAPAction`** HTTP header is used in

jws1_09.fm Page 246 Tuesday, July 2, 2002 5:32 PM

this method call. The **SOAPAction** HTTP header indicates the intent of the SOAP request. An empty-string value indicates that the HTTP request contains the SOAP request's intent. Line 134 sets the **SOAPAction** value to an empty string. Lines 137–138 set the encoding style as a namespace URI, which specifies serialization rules in the SOAP message.

Lines 141–149 define the call's operation name, input and output. Lines 141–143 create a **QName** for the operation. Line 144 invokes method **setOperationName** of interface **Call** to set the operation name. Lines 145–146 add a parameter for the operation by invoking method **addParameter** of interface **Call**. Method **addParameter** takes three arguments—a **String** that specifies the parameter's name (i.e., **"String_1"**, as in the WSDL document generated by **xrpcc**), a **QName** that specifies the XML data type of the parameter and a **ParameterMode** object that indicates the parameter's mode (**IN**, a **static** field of class **ParameterMode**). Line 147 invokes method **setReturnType** of interface **Call** to set the return type of the call operation. Method **setReturnType** takes a **QName** argument that specifies the XML representation of the return type.

To invoke the method call, lines 152–153 call method **invoke** of interface **Call** to make the remote procedure call, using the synchronous request-response interaction mode. Method **invoke** takes a **QName** that specifies the operation name and an array of **Object**s that contains all input parameters, and returns an **Object** that contains the output. The remaining part of method **doPost** (lines 155–185) is similar to that in **SurveyServlet** (Fig. 9.16).

Figure 9.27 contains **surveyDII.xsl**, which the XSL transformation uses. Most of the content is identical to that of **survey.xls** (Fig. 9.18) except that **surveyDII.xsl** changes the form action to **/jaxrpc-VoteDIIClient/Survey** and adds the text field (line 70) for the client to specify the service endpoint.

```
1   <?xml version = "1.0"?>
2
3   <xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
4      version = "1.0">
5
6   <xsl:output method = "xml" omit-xml-declaration = "no"
7      indent = "yes" doctype-system =
8      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtc"
9      doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
10
11  <!-- surveyDII.xsl                                -->
12  <!-- XSL document that transforms XML into XHTML -->
13
14  <!-- specify the root of the XML document -->
15  <!-- that references this stylesheet      -->
16  <xsl:template match = "survey">
17
18     <html xmlns = "http://www.w3.org/1999/xhtml">
19
```

Fig. 9.27 XSL style sheet (**surveyDII.xsl**) that transforms **survey.xml** into an XHTML document. (Part 1 of 2.)

```
   <head>

      <!-- obtain survey title from survey element -->
      <title><xsl:value-of select = "@title"/></title>
   </head>

   <body>
      <p><xsl:value-of select = "@question"/></p>

      <!-- create form -->
      <form method = "post" action = "/jaxrpc-VoteDIIClient/Survey">
         <xsl:for-each select = "language">
            <xsl:sort select = "@name" order = "ascending"/>
            <input type = "radio" name = "favoriteLanguage"
               value = "{@name}"><xsl:value-of select = "@name"/>
            </input><br/>
         </xsl:for-each>
         <br/><p>Type in the service endpoint:</p>
         <input type = "text" name = "endpoint" size = "70"/>
         <br/><br/><input type = "Submit"/>
      </form>

   </body>

   </html>

</xsl:template>

</xsl:stylesheet>
```

Fig. 9.27 XSL style sheet (**surveyDII.xsl**) that transforms **survey.xml** into an XHTML document. (Part 2 of 2.)

To deploy the DII client, we use the same procedure that we used to deploy the stub client. Figure 9.28 shows dialog box **New Web Application Wizard - WAR File** and Figure 9.29 shows the **VoteDIIClient** Web application. The Web contents for the DII clients include **DIISurveyServlet.class** (Fig. 9.26), **survey.xml** (Fig. 9.17), **surveyDII.xsl** (Fig. 9.27) and **surveyresults.xsl** (Fig. 9.19).

To deploy the **VoteDIIClient** Web application, we use the same approach as the one we used to deploy the **VoteClient**. In the **Text Input Dialog**, enter **/jaxrpc-VoteDIIClient** as the Web context. To run the DII client, open a Web browser and visit:

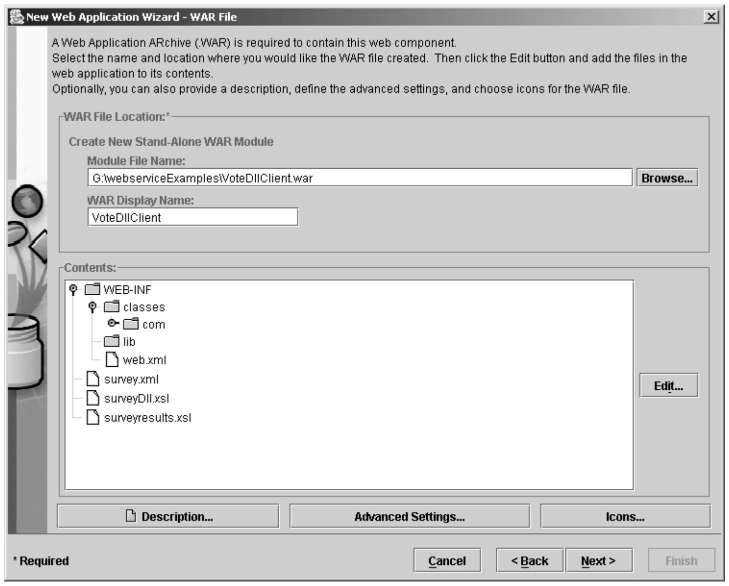
```
http://localhost:8080/jaxrpc-VoteDIIClient/Survey
```

Figure 9.30 shows the **VoteDIIClient** Web application survey form and results.

Using a Dynamic Proxy

In the previous examples, we used static stubs and DII to invoke the Web service. Using static stubs requires the programmer to generate the stubs using **xrpcc**. Using DII requires the programmer to do more coding. In this section, we introduce how to invoke a Web service using a dynamic proxy (a class that is generated at run time), which does not require the stubs and extra coding. To use the dynamic proxy, the clients must have access to the WSDL document and be able to extract service information from the WSDL document.

jws1_09.fm Page 248 Tuesday, July 2, 2002 5:32 PM



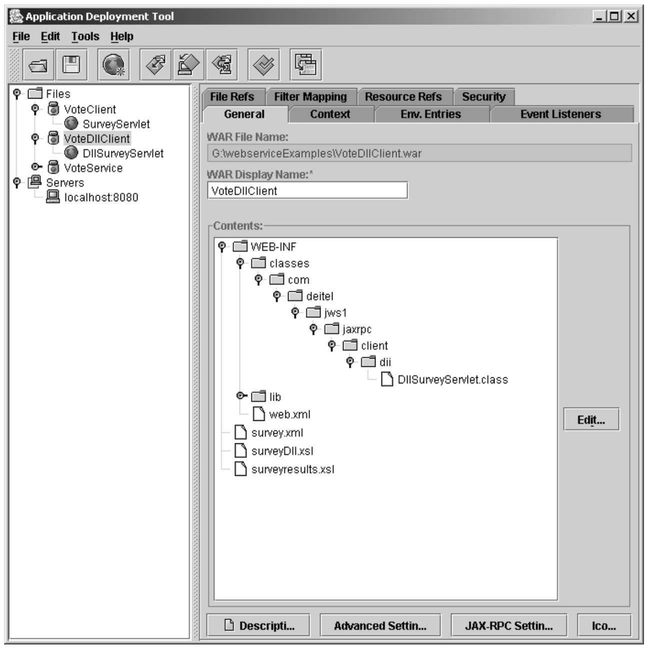
A Web Application ARchive (.WAR) is required to contain this web component.
Select the name and location where you would like the WAR file created. Then click the Edit button and add the files in the web application to its contents.
Optionally, you can also provide a description, define the advanced settings, and choose icons for the WAR file.

WAR File Location:*
Create New Stand-Alone WAR Module
Module File Name: G:\webserviceExamples\VoteDIIClient.war
WAR Display Name: VoteDIIClient

Contents:
```
WEB-INF
  classes
    com
  lib
  web.xml
survey.xml
surveyDII.xsl
surveyresults.xsl
```

* Required

Fig. 9.28 Creating the WAR file for the DII Vote client Web application.

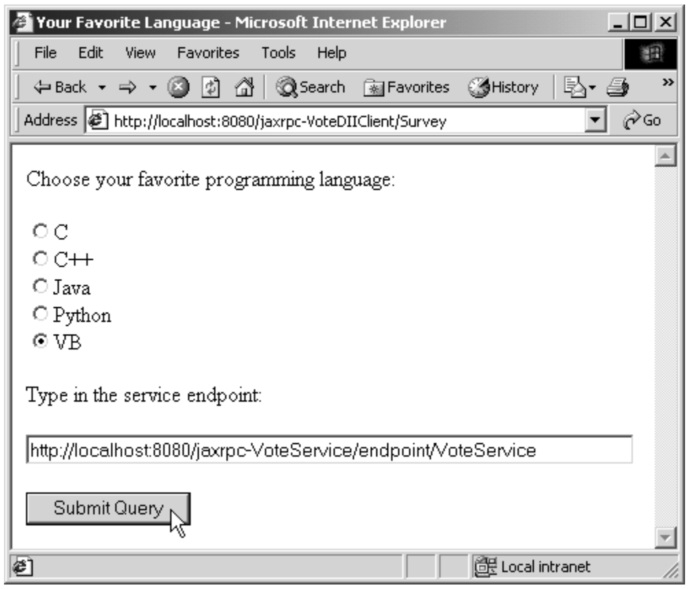


```
Files
  VoteClient
    SurveyServlet
  VoteDIIClient
    DIISurveyServlet
  VoteService
Servers
  localhost:8080
```

WAR File Name: G:\webserviceExamples\VoteDIIClient.war
WAR Display Name:* VoteDIIClient

Contents:
```
WEB-INF
  classes
    com
      deitel
        jws1
          jaxrpc
            client
              dii
                DIISurveyServlet.class
  lib
  web.xml
survey.xml
surveyDII.xsl
surveyresults.xsl
```

Fig. 9.29 `VoteDIIClient` Web application.


Your Favorite Language - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/jaxrpc-VoteDIIClient/Survey
Go
Local intranet

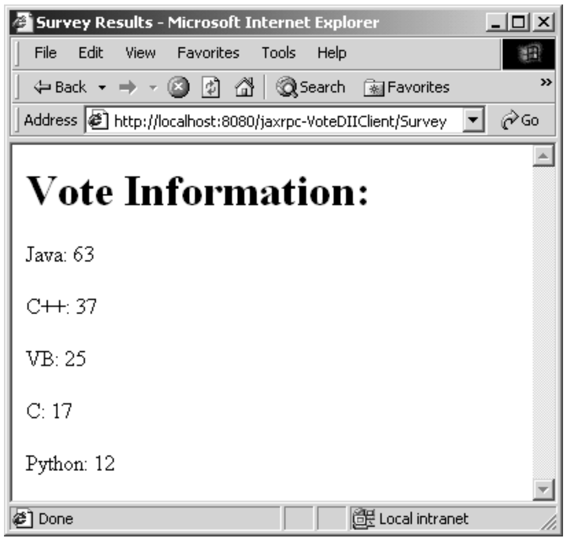
Choose your favorite programming language:

C
C++
Java
Python
[✓ VB]

Type in the service endpoint:

http://localhost:8080/jaxrpc-VoteService/endpoint/VoteService

Submit Query


Survey Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/jaxrpc-VoteDIIClient/Survey
Go
Done
Local intranet

Vote Information:

Java: 63

C++: 37

VB: 25

C: 17

Python: 12

Fig. 9.30 **`VoteDIIClient`** Web application survey form and results.

`SurveyServlet` (Fig. 9.31) creates a survey form in method **`doGet`**, forwards the survey result to the Vote Web service and displays the vote information obtained from the Vote Web service in method **`doPost`**. Method **`doGet`** is almost identical to the method **`doGet`** of **`DIISurveyServlet`** (Fig. 9.26), except that lines 59–61 read **`surveyProxy.xsl`** (Fig. 9.32) rather than read **`surveyDII.xsl`** (Fig. 9.27).

```java
// SurveyServlet.java
// A Web-based survey that invokes the Vote Web service
// from a servlet.
package com.deitel.jws1.jaxrpc.client.proxy;

// Java core packages
import java.io.*;
import java.util.*;
import java.net.URL;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.rpc.*;
import javax.xml.namespace.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// third-party packages
import org.w3c.dom.*;

// xrpcc generated stub packages
import vote.Vote;

public class SurveyServlet extends HttpServlet {

   // display survey form
   protected void doGet( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      // setup response to client
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // get XML document and transform for browser client
      try {

         // get DocumentBuilderFactory for creating
         // DocumentBuilder (i.e., an XML parser)
         DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

         // get DocumentBuilder for building DOM tree
         DocumentBuilder builder =
            factory.newDocumentBuilder();

```

Fig. 9.31 `SurveyServlet` invokes the Web service via a dynamic proxy. (Part 1 of 5.)

jws1_09.fm Page 251 Tuesday, July 2, 2002 5:32 PM

```
         // open InputStream for XML document
         InputStream xmlStream =
            getServletContext().getResourceAsStream(
               "/survey.xml" );

         // create Document based on input XML file
         Document surveyDocument = builder.parse( xmlStream );

         // open InputStream for XSL document
         InputStream xslStream =
            getServletContext().getResourceAsStream(
               "/surveyProxy.xsl" );

         // transform XML document using XSLT
         transform( surveyDocument, xslStream, out );

         // flush and close PrinterWriter
         out.flush();
         out.close();

      } // end try

      // catch XML parser exceptions
      catch( Exception exception ) {
         exception.printStackTrace();
      }

   } // end method doGet

   // process survey response
   protected void doPost( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      // setup response to client
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // read current survey response
      String name = request.getParameter(
         "favoriteLanguage" );

      // attempt to process vote and display current results
      try {

         // specify service WSDL URL
         URL serviceWSDLURL = new URL( "http://localhost:8080/" +
            "jaxrpc-VoteService/endpoint/VoteService?WSDL" );

```

Fig. 9.31 **SurveyServlet** invokes the Web service via a dynamic proxy. (Part 2 of 5.)

jws1_09.fm Page 252 Tuesday, July 2, 2002 5:32 PM

```
         // specify service QName
         QName serviceQName = new QName(
            "http://www.deitel.com/VoteService.wsdl/VoteService",
            "VoteService" );

         // get Service object from SUN's Service implementation
         Service service =
            ServiceFactory.newInstance().createService(
               serviceWSDLURL, serviceQName );

         // specify service port QName
         QName portQName = new QName(
            "http://www.deitel.com/VoteService.wsdl/VoteService",
            "VotePort" );

         // get dynamic proxy
         Vote proxy = ( Vote ) service.getPort(
            portQName, vote.Vote.class );

         // get vote information from server
         String result = proxy.addVote( name );

         StringTokenizer voteTokens =
            new StringTokenizer( result );

         // get DocumentBuilderFactory for creating
         // DocumentBuilder (i.e., an XML parser)
         DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

         // get DocumentBuilder for building DOM tree
         DocumentBuilder builder =
            factory.newDocumentBuilder();

         // create Document (empty DOM tree)
         Document resultDocument = builder.newDocument();

         // generate XML from result and append to Document
         Element resultElement = generateXML(
            voteTokens, resultDocument );
         resultDocument.appendChild( resultElement );

         // open InputStream for XSL document
         InputStream xslStream =
            getServletContext().getResourceAsStream(
               "/surveyresults.xsl" );

         // transform XML document using XSLT
         transform( resultDocument, xslStream, out );

         // flush and close PrintWriter
         out.flush();
```

Fig. 9.31 **SurveyServlet** invokes the Web service via a dynamic proxy. (Part 3 of 5.)

jws1_09.fm Page 253 Tuesday, July 2, 2002 5:32 PM

```
         out.close();

      } // end try

      // catch connection and XML parser exceptions
      catch ( Exception exception ) {
         exception.printStackTrace();
      }

   }  // end method doPost

   // generate vote information XML representation
   private Element generateXML( StringTokenizer voteTokens,
      Document document )
   {
      // create root element
      Element root = document.createElement( "surveyresults" );

      Element language;
      Element name;
      Element vote;

      // create language element for each language
      while ( voteTokens.hasMoreTokens() ) {
         language = document.createElement( "language" );
         name = document.createElement( "name" );
         name.appendChild( document.createTextNode(
            voteTokens.nextToken() ) );

         language.appendChild( name );
         vote = document.createElement( "vote" );
         vote.appendChild( document.createTextNode(
            voteTokens.nextToken() ) );
         language.appendChild( vote );
         root.appendChild( language );
      }

      return root; // return root element

   } // end method generate XML

   // transform XML document using XSLT InputStream
   // and write resulting document to PrintWriter
   private void transform( Document document,
      InputStream xslStream, PrintWriter output )
   {
      // transform XML to XHTML
      try {

         // create DOMSource for source XML document
         Source xmlSource = new DOMSource( document );

```

Fig. 9.31 **SurveyServlet** invokes the Web service via a dynamic proxy. (Part 4 of 5.)

jws1_09.fm Page 254 Tuesday, July 2, 2002 5:32 PM

```
203          // create StreamSource for XSLT document
204          Source xslSource = new StreamSource( xslStream );
205
206          // create StreamResult for transformation result
207          Result result = new StreamResult( output );
208
209          // create TransformerFactory to obtain Transformer
210          TransformerFactory transformerFactory =
211             TransformerFactory.newInstance();
212
213          // create Transformer for performing XSL transformation
214          Transformer transformer =
215             transformerFactory.newTransformer( xslSource );
216
217          // perform transformation and deliver content to client
218          transformer.transform( xmlSource, result );
219
220       } // end try
221
222       // handle exception when transforming XML document
223       catch( TransformerException exception ) {
224          exception.printStackTrace( System.err );
225       }
226
227    } // end method transform
228
229 } // end class SurveyServlet
```

Fig. 9.31 **`SurveyServlet`** invokes the Web service via a dynamic proxy. (Part 5 of 5.)

Method **`doPost`** (lines 80–160) implements support for dynamic proxy. Lines 96–97 specify the service WSDL's URL. Lines 100–102 specify the service name with a qualified namespace. The namespace and service name are obtained by examining the WSDL document. Lines 105–107 invoke method **`createService`** of class **`ServiceFactory`** with the WSDL URL and qualified service name to create a **`Service`** instance. Lines 110–112 specify the service port with a qualified namespace. Such information can be extracted from the WSDL document. Lines 115–116 invoke method **`getPort`** of class **`Service`** with the qualified service port and the service interface (**`vote.Vote.class`**) to obtain the dynamic proxy. You can define the service interface by examining the WSDL document. For simplicity, we just use the interface generated by **`xrpcc`** when we introduced the static stubs. Line 119 invokes method **`addVote`** of the Vote Web service to obtain the vote information. The remaining code in method **`doPost`** (lines 121–158) is similar to that of **`DIISurveyServlet`** (Fig. 9.26).

Figure 9.32 contains **`surveyProxy.xsl`**. Most of the content is identical to that of **`survey.xsl`** (Fig. 9.18) except that **`surveyProxy.xsl`** changes the form action to **`/jaxrpc-VoteProxyClient/Survey`** (line 31).

To deploy the dynamic proxy client, we use the same procedure that we used to deploy the stub client. Figure 9.33 shows dialog box **New Web Application Wizard - WAR File** and Figure 9.34 shows the **`VoteProxyClient`** Web application. Notice that directory **`vote`** contains only **`Vote.class`** rather than contains all client-side classes gener-

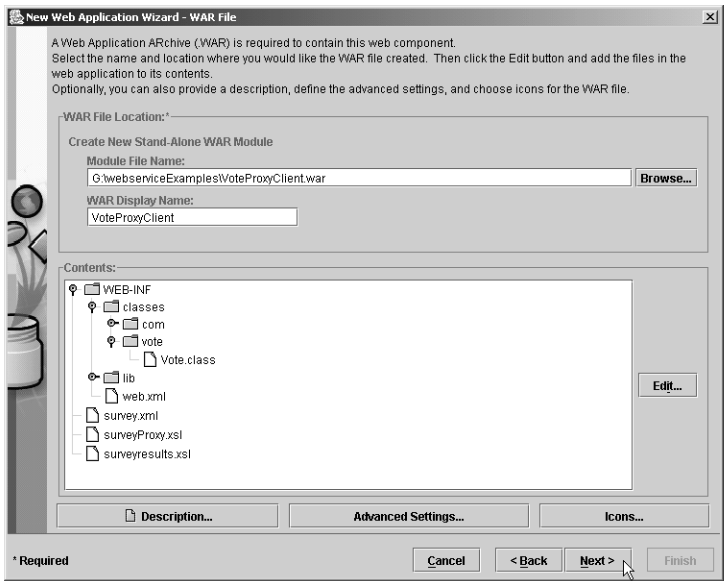
jws1_09.fm Page 255 Tuesday, July 2, 2002 5:32 PM

ated by **xrpcc**. The Web contents for the dynamic proxy clients include **SurveyServlet.class** (Fig. 9.31), **Vote.class** (generated by **xrpcc**), **survey.xml** (Fig. 9.17), **surveyProxy.xml** (Fig. 9.32) and **surveyre-sults.xsl** (Fig. 9.19).

```
<?xml version = "1.0"?>

<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
   version = "1.0">

<xsl:output method = "xml" omit-xml-declaration = "no"
   indent = "yes" doctype-system =
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtc"
   doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>

<!-- surveyProxy.xsl                                -->
<!-- XSL document that transforms XML into XHTML -->

<!-- specify the root of the XML document -->
<!-- that references this stylesheet      -->
<xsl:template match = "survey">

   <html xmlns = "http://www.w3.org/1999/xhtml">

   <head>

      <!-- obtain survey title from survey element -->
      <title><xsl:value-of select = "@title"/></title>
   </head>

   <body>
      <p><xsl:value-of select = "@question"/></p>

      <!-- create form -->
      <form method = "post"
         action = "/jaxrpc-VoteProxyClient/Survey">
         <xsl:for-each select = "language">
            <xsl:sort select = "@name" order = "ascending"/>
            <input type = "radio" name = "favoriteLanguage"
               value = "{@name}"><xsl:value-of select = "@name"/>
            </input><br/>
         </xsl:for-each>
         <br/><input type = "Submit"/>
      </form>

   </body>

   </html>

</xsl:template>

</xsl:stylesheet>
```

Fig. 9.32 XSL style sheet (**surveyProxy.xsl**) that transforms **survey.xml** into an XHTML document.

jws1_09.fm Page 256 Tuesday, July 2, 2002 5:32 PM

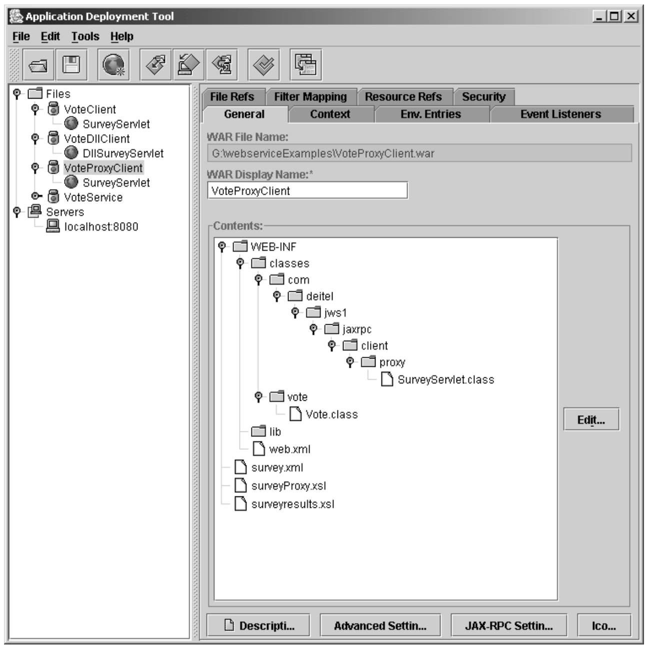

New Web Application Wizard - WAR File

A Web Application ARchive (.WAR) is required to contain this web component.
Select the name and location where you would like the WAR file created. Then click the Edit button and add the files in the web application to its contents.
Optionally, you can also provide a description, define the advanced settings, and choose icons for the WAR file.

WAR File Location:*
Create New Stand-Alone WAR Module
Module File Name: G:\webserviceExamples\VoteProxyClient.war
WAR Display Name: VoteProxyClient

Contents:
```
WEB-INF
  classes
    com
    vote
      Vote.class
  lib
  web.xml
survey.xml
surveyProxy.xsl
surveyresults.xsl
```

Fig. 9.33 Creating the WAR file for the dynamic proxy Vote client Web application.


Application Deployment Tool

```
Files
  VoteClient
    SurveyServlet
  VoteDIIClient
    DIISurveyServlet
  VoteProxyClient
    SurveyServlet
  VoteService
Servers
  localhost:8080
```

WAR File Name: G:\webserviceExamples\VoteProxyClient.war
WAR Display Name:* VoteProxyClient

Contents:
```
WEB-INF
  classes
    com
      deitel
        jws1
          jaxrpc
            client
              proxy
                SurveyServlet.class
    vote
      Vote.class
  lib
  web.xml
survey.xml
surveyProxy.xsl
surveyresults.xsl
```

Fig. 9.34 `VoteProxyClient` Web application.

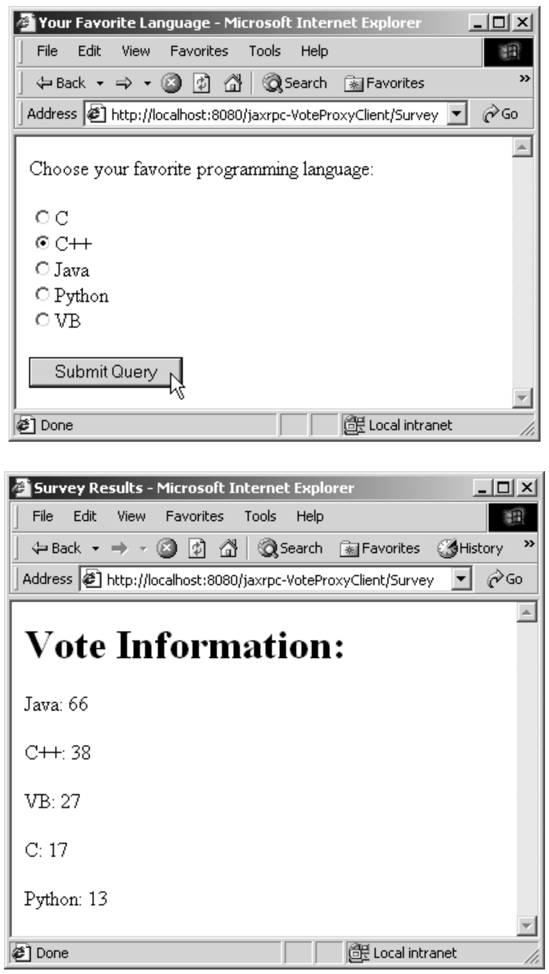
To deploy the **`VoteProxyClient`** Web application, we use the same approach we used to deploy the **`VoteClient`**. In the **Text Input Dialog**, enter **`/jaxrpc-VoteProxyClient`** as the Web context. To run the dynamic proxy client, open a Web browser and visit:

```
http://localhost:8080/jaxrpc-VoteProxyClient/Survey
```

Figure 9.35 shows the **`VoteProxyClient`** Web application survey form and results.

9.4 Improved Vote Service

The previous **`Vote`** service example takes a **`String`** that represents one vote for a particular programming language and returns a **`String`** that contains votes for each program-


Your Favorite Language - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/jaxrpc-VoteProxyClient/Survey Go
Done Local intranet
Survey Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/jaxrpc-VoteProxyClient/Survey Go
Done Local intranet

Choose your favorite programming language:

- ○ C
- ● C++
- ○ Java
- ○ Python
- ○ VB

Submit Query

Vote Information:

Java: 66

C++: 38

VB: 27

C: 17

Python: 13

Fig. 9.35 **`VoteProxyClient`** Web application survey form and results.

jws1_09.fm Page 258 Tuesday, July 2, 2002 5:32 PM

ming language. Recall that vote results are separated by **`" "`**. In this section, we represent the **`Vote`** service in a more realistic way: the service returns an array of JavaBean objects, in which each JavaBean represents the vote tally for one programming language.

The three major steps in this example are as follows:

1. Defining a service interface and implementation.
2. Deploying the service to the Web server (in this example, we use the version of Tomcat distributed with the JWSDP).
3. Generating the client-side classes and writing the client application that interacts with the service.

Fig. 9.36 shows the structure of this example.

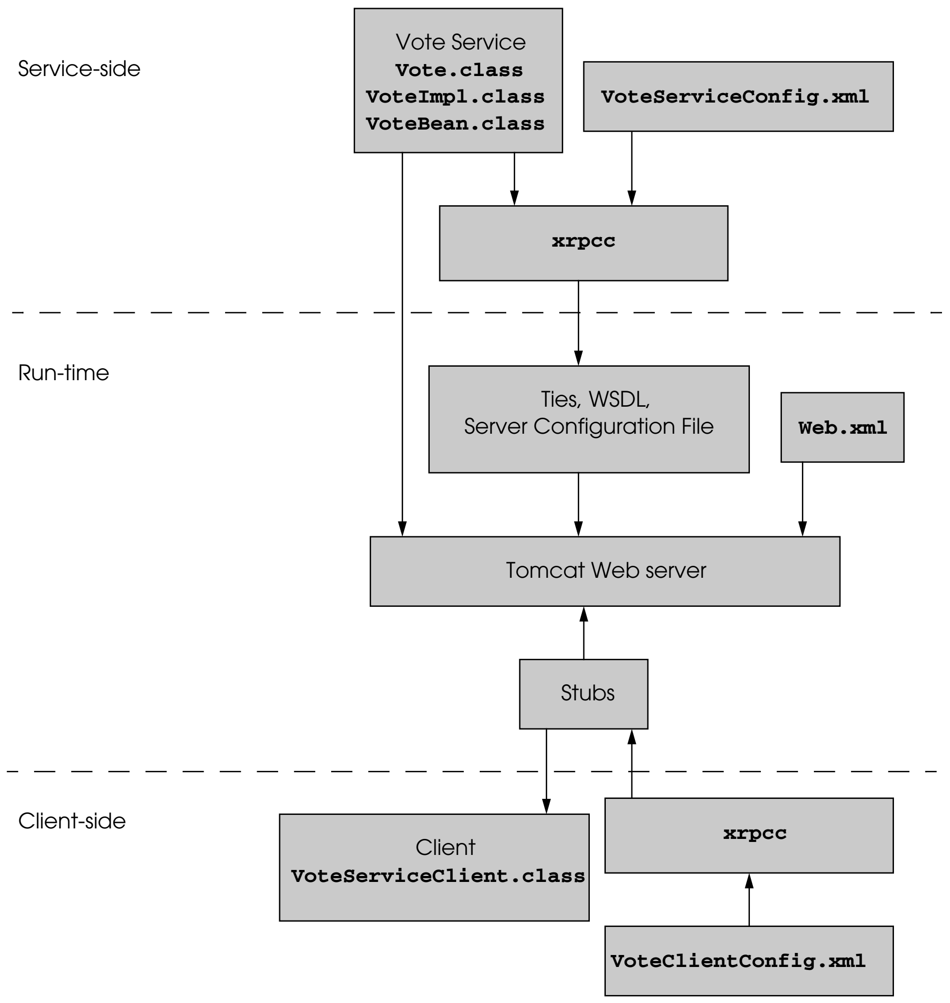


Fig. 9.36 Vote example structure.

jws1_09.fm Page 259 Tuesday, July 2, 2002 5:32 PM

9.4.1 Service Definition

As in the first example, the first step in the creation of a JAX-RPC Web service is to define the remote service endpoint interface. Interface **`Vote`** (Fig. 9.37)—which extends interface **`Remote`** (line 9)—is the remote interface for our second JAX-RPC Web service example. Lines 12–13 declare method **`addVote`**, which clients can invoke to add votes for the users' favorite programming languages. Recall that the input parameters and return values of the methods declared in a service interface must be JAX-RPC-supported types. The return value of method **`addVote`** is an array of **`VoteBean`** objects. When we implement the **`VoteBean`** class, we must follow the conditions listed in Section 9.3.1 to ensure compatibility with JAX-RPC.

Class **`VoteImpl`** implements remote interface **`Vote`** and interface **`ServiceLifecycle`** (line 18). Interface **`ServiceLifecycle`** allows service endpoint classes to setup access to external resources, such as databases.

Lines 24–74 implement method **`init`** of interface **`ServiceLifecycle`** to connect to a Cloudscape database. Lines 123–137 implement method **`destroy`** of interface **`ServiceLifecycle`**. These two methods are the same as in the previous example.

Lines 77–120 implement method **`addVote`** of interface **`Vote`**. Line 83 sets the parameter of **`sqlUpdate`** to the programming language that was selected by the user. After setting the parameter for the **`PreparedStatement`**, the program calls method **`executeUpdate`** of interface **`PreparedStatement`** to execute the **`UPDATE`** operation. Line 89 calls method **`executeQuery`** of interface **`PreparedStatement`** to execute the **`SELECT`** operation. **`ResultSet results`** stores the query results. Lines 91–107 process the **`ResultSet`** and store the results in an array of **`VoteBean`**s. Line 109 returns the **`VoteBean`**s to the client.

```
 1  // Vote.java
 2  // VoteService interface declares a method for adding votes and
 3  // returning vote information.
 4  package com.deitel.jws1.jaxrpc.voteservice;
 5
 6  // Java core packages
 7  import java.rmi.*;
 8
 9  public interface Vote extends Remote {
10
11     // obtain vote information from server
12     public VoteBean[] addVote( String languageName )
13        throws RemoteException;
14  }
```

Fig. 9.37 **`Vote`** interface defines the service interface for the Vote Web service.

```
 1  // VoteImpl.java
 2  // VoteImpl implements the Vote remote interface to provide
 3  // a VoteService remote object.
 4  package com.deitel.jws1.jaxrpc.voteservice;
```

Fig. 9.38 **`VoteImpl`** defines the service implementation for the Vote Web service. (Part 1 of 4.)

```java

// Java core packages
import java.rmi.*;
import java.sql.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;

// Java XML packages
import javax.xml.rpc.server.*;
import javax.xml.rpc.JAXRPCException;

public class VoteImpl implements ServiceLifecycle, Vote {

   private Connection connection;
   private PreparedStatement sqlUpdate, sqlSelect;

   // setup database connection and prepare SQL statement
   public void init( Object context )
      throws JAXRPCException
   {
      // attempt database connection and
      // create PreparedStatements
      try {

         // cast context to ServletEndpointContext
         ServletEndpointContext endpointContext =
            ( ServletEndpointContext ) context;

         // get ServletContext
         ServletContext servletContext =
            endpointContext.getServletContext();

         // get database driver from servlet context
         String dbDriver =
            servletContext.getInitParameter( "dbDriver" );

         // get database name from servlet context
         String voteDB =
            servletContext.getInitParameter( "voteDB" );

         Class.forName( dbDriver ); // load database driver

         // connect to database
         connection = DriverManager.getConnection( voteDB );

         // PreparedStatement to add one to vote total for a
         // specific language
         sqlUpdate =
            connection.prepareStatement(
```

Fig. 9.38 `VoteImpl` defines the service implementation for the Vote Web service. (Part 2 of 4.)

```
                "UPDATE surveyresults SET vote = vote + 1 " +
                "WHERE name = ?" );

         // PreparedStatement to obtain surveyresults table's data
         sqlSelect =
            connection.prepareStatement( "SELECT name, vote " +
                "FROM surveyresults ORDER BY vote DESC" );

      } // end try

      // for any exception throw a JAXRPCException to
      // indicate that the servlet is not currently available
      catch ( Exception exception ) {
         exception.printStackTrace();

         throw new JAXRPCException( exception.getMessage() );
      }

   } // end method init

   // implementation for interface Vote method addVote
   public VoteBean[] addVote( String name ) throws RemoteException
   {
      // obtain votes count from database then update database
      try {

         // set parameter in sqlUpdate
         sqlUpdate.setString( 1, name );

         // execute sqlUpdate statement
         sqlUpdate.executeUpdate();

         // execute sqlSelect statement
         ResultSet results = sqlSelect.executeQuery();

         List voteInformation = new ArrayList();

         // iterate ResultSet and prepare return string
         while ( results.next() ) {

            // store results to VoteBean List
            VoteBean vote = new VoteBean(
               results.getString( 1 ), results.getInt( 2 ) );
            voteInformation.add( vote );
         }

         // create array of VoteBeans
         VoteBean[] voteBeans =
            new VoteBean[ voteInformation.size() ];

         // get array from voteInformation List
         voteInformation.toArray( voteBeans );
```

Fig. 9.38 `VoteImpl` defines the service implementation for the Vote Web service. (Part 3 of 4.)

jws1_09.fm Page 262 Tuesday, July 2, 2002 5:32 PM

```java

         return voteBeans;

      } // end try

      // handle database exceptions by returning error to client
      catch ( Exception exception ) {

         //throw the exception back to the client
         throw new RemoteException( exception.getMessage() );
      }

   } // end method addVote

   // close SQL statements and database when servlet terminates
   public void destroy()
   {
      // attempt to close statements and database connection
      try {
         sqlUpdate.close();
         sqlSelect.close();
         connection.close();
      }

      // handle database exception
      catch ( Exception exception ) {
         exception.printStackTrace();
      }

   } // end method destroy

} // end class VoteImpl
```

Fig. 9.38 **`VoteImpl`** defines the service implementation for the Vote Web service. (Part 4 of 4.)

Class **`VoteBean`** stores data that represents the vote count for each programming language. Line 17 provides the public no-argument constructor. Lines 27–36 provide *get* methods for each piece of information. Lines 39–48 provide *set* methods for each piece of information.

```java
1  // VoteBean.java
2  // VoteBean maintains vote information for one programming language.
3  package com.deitel.jws1.jaxrpc.voteservice;
4
5  // Java core packages
6  import java.io.*;
7
8  // Java extension packages
9  import javax.swing.*;
```

Fig. 9.39 **`VoteBean`** stores the vote count for one programming language. (Part 1 of 2.).

```

public class VoteBean implements Serializable {

   private String languageName; // name of language
   private int count; // vote count

   // public no-argument constructor
   public VoteBean() {}

   // VoteBean constructor
   public VoteBean( String voteLanguage, int voteCount )
   {
      languageName = voteLanguage;
      count = voteCount;
   }

   // get language name
   public String getLanguageName()
   {
      return languageName;
   }

   // get vote count
   public int getCount()
   {
      return count;
   }

   // set language name
   public void setLanguageName( String voteLanguage )
   {
      languageName = voteLanguage;
   }

   // set vote count
   public void setCount( int voteCount )
   {
      count = voteCount;
   }

} // end class VoteBean
```

Fig. 9.39 **`VoteBean`** stores the vote count for one programming language. (Part 2 of 2.).

9.4.2 Service Deployment

In this section, we discuss how to generate the service-side artifacts (like ties, WSDL document) using **`xrpcc`** and how to deploy the service on Tomcat without using the **`deploytool`**. To generate a WSDL document, **`xrpcc`** reads an XML configuration file that lists remote interfaces. **`VoteServiceConfig.xml`** (Fig. 9.40) is the configuration for our **`Vote`** service example. **`VoteServiceConfig.xml`** follows the standard syntax provided by JWSDP to create the configuration file. The root element **`configuration`** contains one **`service`** element that corresponds to remote service. The **`name`** attribute of

jws1_09.fm Page 264 Tuesday, July 2, 2002 5:32 PM

element **service** (line 5) indicates the service name. The **targetNamespace** attribute specifies the target namespace for the generated WSDL document (line 6). The **type-Namespace** attribute (line 7) specifies the target namespace within the **types** section of the WSDL document. The **packageName** attribute specifies the fully qualified package name of the generated stubs, ties and other classes (line 8). The value of attribute **packa-geName** does not need to match the package name of any of the remote interfaces. Element **interface** (lines 10–13) defines the fully qualified name of the service interface via its attribute **name**, and the fully qualified name of the service implementation via its attribute **servantName**. Element **interface** defines a service port in the WSDL file.

Using **xrpcc** requires that we include the location of the service-interface definition and implementation in the classpath. Compile the source code and place the classes in directory **voteserviceoutput**. It is necessary to create directory **voteservice-output** before executing the following **xrpcc** command

```
xrpcc -classpath voteserviceoutput -d voteserviceoutput
   -server -keep VoteServiceConfig.xml
```

to create service-side classes and the WSDL document. Option **d** specifies the directory in which to place the generated files. Option **server** specifies that only server-side files should be generated. Option **keep** instructs **xrpcc** to keep all the generated resources, including the WSDL document and the Java source code.

The **xrpcc** tool also generates server configuration file **VoteService_Config.properties**, which is used by the JAX-RPC runtime environment. We may modify **VoteService_Config.properties** (in directory **voteserviceoutput**) to make the service WSDL document available from the service end point. Open **VoteService_Config.properties** with a text editor and append the following line:

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

to the end of the file. By doing so, the service WSDL document is accessible at

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <configuration
 3     xmlns = "http://java.sun.com/xml/ns/jax-rpc/ri/config">
 4
 5     <service name = "VoteService"
 6        targetNamespace = "http://www.deitel.com/VoteService.wsdl"
 7        typeNamespace = "http://www.deitel.com/VoteService/type"
 8        packageName = "com.deitel.jws1.jaxrpc.voteservice">
 9
10        <interface
11           name = "com.deitel.jws1.jaxrpc.voteservice.Vote"
12           servantName =
13              "com.deitel.jws1.jaxrpc.voteservice.VoteImpl"/>
14     </service>
15  </configuration>
```

Fig. 9.40 **VoteServiceConfig.xml** is the configuration file for generating the Vote Web service artifacts using **xrpcc**.

jws1_09.fm Page 265 Tuesday, July 2, 2002 5:32 PM

```
http://localhost:8080/jaxrpc-voteapp/vote/endpoint?WSDL
```

assuming the service is deployed into the **`jaxrpc-voteapp`** Web context.

To deploy the Vote Web service to Tomcat, we need to:

1. Write a deployment descriptor.
2. Create a Web context, which, for this example, is **`jaxrpc-voteapp`**.
3. Copy required classes to directory **`jaxrpc-voteapp\WEB-INFO\classes`** and required libraries to directory **`jaxrpc-voteapp\WEB-INFO\lib`**.

`Web.xml` (Fig. 9.41) is the deployment descriptor for the **`Vote`** service. Two **`context-param`** elements (lines 14–19 and 21–26) specify the database name and database driver as context parameters. The URL **`jdbc:cloudscape:rmi:languagesurvey`** specifies the protocol for communication (**`jdbc`**), the subprotocol for communication (**`cloudscape:rmi`**) and the name of the database (**`languagesurvey`**). Element **`servlet`** (lines 28–44) describes the ***`JAXRPCServlet`*** servlet that is distributed with the JWSDP 1.0 final release. Servlet **`JAXRPCServlet`** is a JAX-RPC implementation for dispatching the request to the Web-service implementation. In our case, the **`JAXRPCServlet`** dispatches the client request to the **`VoteImpl`** class. When the **`JAXRPCServlet`** receives an HTTP request that contains a SOAP message, the servlet retrieves the data that the SOAP message contains, then dispatches the method call to the service-implementation class via the tie. Element **`servlet-class`** (lines 34–36) specifies the compiled servlet's fully qualified class name—**`com.sun.xml.rpc.server.http.JAXRPCServlet`**. The **`JAXRPCServlet`** obtains information about the server-configuration file, which is passed to the servlet as an initialization parameter. Element **`init-param`** (lines 37–42) specifies the name and value of the initialization parameter needed by the **`JAXRPCServlet`**. Element **`param-name`** (line 38) indicates the name of the initialization parameter, which is **`configuration.file`**. Element **`param-value`** (lines 39–41) specifies the value of the initialization parameter, **`/WEB-INF/VoteService_Config.properties`** (generated by **`xrpcc`**), which is the location of the server-configuration file. Element **`servlet-mapping`** (lines 47–50) specifies **`servlet-name`** and **`url-pattern`** elements. The URL pattern enables the server to determine which requests should be sent to the **`JAXRPCServlet`**.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3    <!DOCTYPE web-app
4        PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5        "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
6
7   <web-app>
8      <display-name>
9         Java Web service JAX-RPC Vote service Example
10     </display-name>
11
12     <description>Vote service Application</description>
```

Fig. 9.41 **`web.xml`** for deploying the Vote service. (Part 1 of 2.)

jws1_09.fm Page 266 Tuesday, July 2, 2002 5:32 PM

```

   <context-param>
      <param-name>voteDB</param-name>
      <param-value>
         jdbc:cloudscape:rmi:languagesurvey
      </param-value>
   </context-param>

   <context-param>
      <param-name>dbDriver</param-name>
      <param-value>
         COM.cloudscape.core.RmiJdbcDriver
      </param-value>
   </context-param>

   <servlet>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <display-name>JAXRPCEndpoint</display-name>
      <description>
         Endpoint for Vote 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/VoteService_Config.properties
         </param-value>
      </init-param>
      <load-on-startup>0</load-on-startup>
   </servlet>

   <!-- Servlet mappings -->
   <servlet-mapping>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <url-pattern>/endpoint/*</url-pattern>
   </servlet-mapping>

   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>
</web-app>
```

Fig. 9.41 **web.xml** for deploying the Vote service. (Part 2 of 2.)

Figure 9.42 shows the resulting **jaxrpc-voteapp** Web-application directory structure. Because the Vote Web service implementation uses a Cloudscape database, we need to include both **cloudclient.jar** and **RmiJdbc.jar** in directory **jaxrpc-voteapp\WEB-INF\lib**. These JAR files are available from directory **%J2EE_HOME%\lib\cloudscape**, where **J2EE_HOME** is the J2EE installation directory. Directory **jaxrpc-voteapp\WEB-INF\classes** contains all classes in directory **voteserviceoutput**, including **Vote.class**, **VoteImpl.class**, **VoteBean.class** and other classes generated by **xrpcc**.

jws1_09.fm Page 267 Tuesday, July 2, 2002 5:32 PM

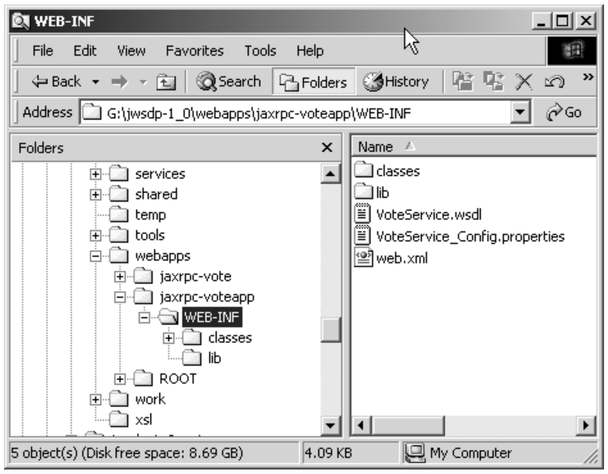

WEB-INF
File Edit View Favorites Tools Help
Back Search Folders History
Address G:\jwsdp-1_0\webapps\jaxrpc-voteapp\WEB-INF
Folders
services
shared
temp
tools
webapps
jaxrpc-vote
jaxrpc-voteapp
WEB-INF
classes
lib
ROOT
work
xsl
Name
classes
lib
VoteService.wsdl
VoteService_Config.properties
web.xml
5 object(s) (Disk free space: 8.69 GB)
4.09 KB
My Computer

Fig. 9.42 Vote Web service Web application directory and file structure.

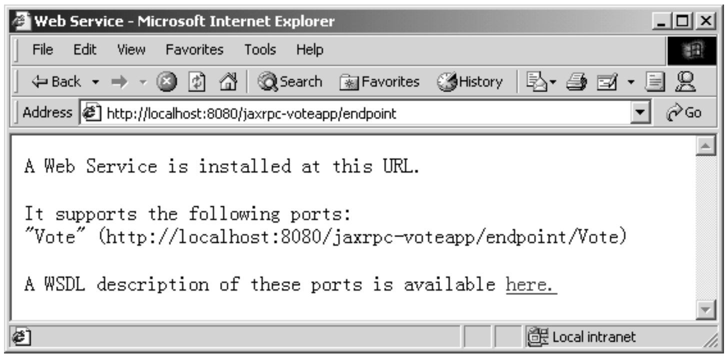
We may verify whether the Vote Web service is deployed successfully. To verify the deployment, start Tomcat and point your browser to

```
http://localhost:8080/jaxrpc-voteapp/endpoint
```

Figure 9.43 shows the result of this action.

9.4.3 Client Invocation

Next, we demonstrate how to write a Java client of the Vote Web service using JAX-RPC. Class **VoteServiceClient** (Fig. 9.44) is the client application that invokes remote method **addVote** to add votes and obtain voting information. The **vote** package name (line 16) is specified in the configuration file (Fig. 9.45) passed to **xrpcc**.


Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/jaxrpc-voteapp/endpoint
Go
Local intranet

```
A Web Service is installed at this URL.

It supports the following ports:
"Vote" (http://localhost:8080/jaxrpc-voteapp/endpoint/Vote)

A WSDL description of these ports is available here.
```

Fig. 9.43 Result of verification of the service's deployment.

jws1_09.fm Page 268 Tuesday, July 2, 2002 5:32 PM

In the **VoteServiceClient** constructor (lines 21–56), lines 26–49 create a **JButton voteButton** to invoke the Vote service. Lines 51–54 add the **voteButton** to the content pane. When users click **voteButton**, they must select their favorite programming language, which invokes method **showVotes** to get the vote result.

Method (lines 59–95) **showVotes** gets the service stub and casts the service stub to service interface (line 69). Line 72 invokes method **addVote** of the service interface to get an array of **VoteBean**s that contains vote information for each programming language. Lines 74–86 extract the vote information from the array and display the information in a **JOptionPane** message dialog.

Figure 9.45 is the configuration file that **xrpcc** uses to generate the client-side classes. Element **wsdl** (lines 4–6) specifies the location of the WSDL file and the fully qualified package name of the client-side classes. The package name may differ from the package name of **VoteServiceClient**.

```
1   // VoteServiceClient.java
2   // VoteServiceClient display the survey window.
3   package com.deitel.jws1.jaxrpc.voteclient;
4
5   // Java core packages
6   import java.awt.*;
7   import java.awt.event.*;
8
9   // Java extension packages
10  import javax.swing.*;
11
12  // Java XML packages
13  import javax.xml.rpc.*;
14
15  // client packages
16  import vote.*;
17
18  public class VoteServiceClient extends JFrame
19  {
20     private static String endpoint;
21
22     // VoteServiceClient constructor
23     public VoteServiceClient()
24     {
25        // create JButton for getting Vote service
26        JButton voteButton = new JButton( "Get Vote Service" );
27        voteButton.addActionListener(
28
29           new ActionListener() {
30
31              // action for the voteButton
32              public void actionPerformed( ActionEvent event )
33              {
34                 String[] languages =
35                    { "C", "C++", "Java", "VB", "Python" };
36
```

Fig. 9.44 **VoteServiceClient** is the client for the Vote Web service. (Part 1 of 3.).

jws1_09.fm Page 269 Tuesday, July 2, 2002 5:32 PM

```
                String selectedLanguage = ( String )
                   JOptionPane.showInputDialog(
                   VoteServiceClient.this,
                   "Select Language", "Language Selection",
                   JOptionPane.QUESTION_MESSAGE,
                   null, languages, "" );

                showVotes( selectedLanguage );

             } // end method actionPerformed

          } // end ActionListener

       ); // end call to addActionListener

       JPanel buttonPanel = new JPanel();
       buttonPanel.add( voteButton );
       getContentPane().add( buttonPanel, BorderLayout.CENTER );

    } // end VoteServiceClient constructor

    // connect to Vote Web service and get vote information
    public void showVotes( String languageName )
    {
       // connect to Web service and get vote information
       try {

          // get Web service stub
          Stub stub = ( Stub )
             ( new VoteService_Impl().getVotePort() );

          // cast stub to service interface
          Vote vote = ( Vote ) stub;

          // get vote information from Web service
          VoteBean[] voteBeans = vote.addVote( languageName );

          StringBuffer results = new StringBuffer();
          results.append( "Vote result: \n" );

          // get vote information from voteBeans
          for ( int i = 0; i < voteBeans.length ; i++) {
             results.append( "   "
                + voteBeans[ i ].getLanguageName() + ":" );
             results.append( voteBeans[ i ].getCount() );
             results.append( "\n" );
          }

          // display Vote information
          JOptionPane.showMessageDialog( this, results );

       } // end try
```

Fig. 9.44 `VoteServiceClient` is the client for the Vote Web service. (Part 2 of 3.).

```

         // handle exceptions communicating with remote object
         catch ( Exception exception ) {
            exception.printStackTrace();
         }

      } // end method showVotes

      // execute VoteServiceClient
      public static void main( String args[] )
      {
         // configure and display application window
         VoteServiceClient client = new VoteServiceClient();

         client.setDefaultCloseOperation( EXIT_ON_CLOSE );
         client.pack();
         client.setSize( 250, 65 );
         client.setVisible( true );

      } // end main

   } // end class VoteServiceClient
```

Fig. 9.44 **`VoteServiceClient`** is the client for the Vote Web service. (Part 3 of 3.).

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
   xmlns = "http://java.sun.com/xml/ns/jax-rpc/ri/config">
   <wsdl location =
      "http://localhost:8080/jaxrpc-voteapp/endpoint?WSDL"
      packageName = "vote">
   </wsdl>
</configuration>
```

Fig. 9.45 **`VoteClientConfig.xml`** that **`xrpcc`** uses to generate client-side classes from WSDL document.

The command

```
xrpcc -d voteclientoutput -client VoteClientConfig.xml
```

generates the client-side classes for the Vote service in directory **`voteclientoutput`**. **`VoteClientConfig.xml`** (Fig. 9.45) is the configuration file passed to **`xrpcc`**.

Figure 9.46 shows all the JAR files that are required to compile and execute client-side applications. Directories **`%JWSDP_HOME%\common\lib`** and **`%JWSDP_HOME%\common\endorsed`** contain these JAR files, where **`%JWSDP_HOME%`** is the installation directory of JWSDP.

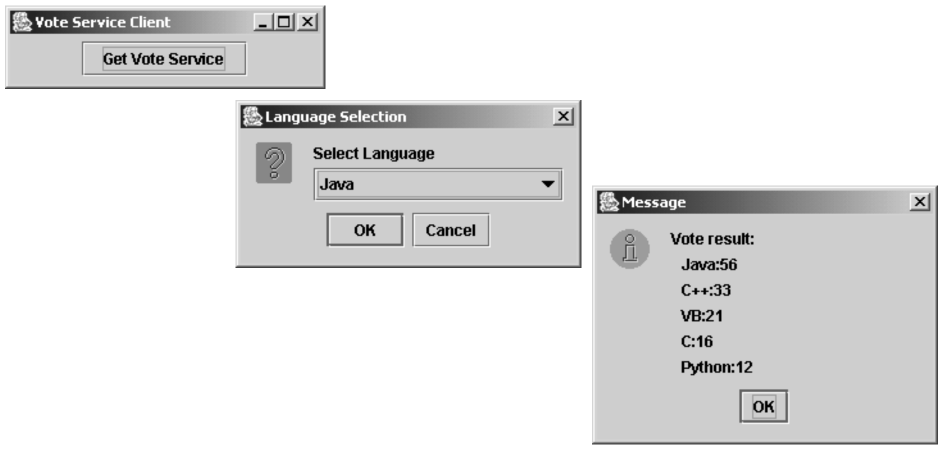
To run the client application, type:

```
java -classpath voteclientoutput;%CLASSPATH%
   com.deitel.jws1.jaxrpc.voteclient.VoteServiceClient
```

The **`%CLASSPATH%`** variable contains all the JAR files listed in Fig. 9.46. Fig. 9.47 shows the output of the client application.

JAR files required for compilation	JAR files required for execution
`jaxrpc-api.jar`	`jaxrpc-api.jar`
`jaxrpc-ri.jar`	`jaxrpc-ri.jar`
	`saaj-api.jar`
	`saaj-ri.jar`
	`mail.jar`
	`activation.jar`

Fig. 9.46 JAR files required to compile and execute client-side applications.



```
Vote result:
  Java:56
  C++:33
  VB:21
  C:16
  Python:12
```

Fig. 9.47 Output of the Vote service client.

9.5 Accessing Web Services Written by a Third Party

In most B2B or B2C applications, Web service providers and Web service consumers are separate organizations. Usually, Web service consumers search for a particular Web service from a Web service registry (such as public UDDI registry or ebXML registry) or from a well-known Web site that contains a list of Web services (such as **`www.xmethods.com`**). This section demonstrates how to connect to and access the Unisys Weather Web Service that is available from **`www.xmethods.com`**. You can get the Unisys Weather Web Service information from

```
www.xmethods.com/ve2/ViewListing.po?serviceid=47216
```

which contains information such as the location of the WSDL file and the description of the service. Although the Unisys Weather Web Service was written using Microsoft's .NET Framework, we can access it easily using JAX-RPC.

jws1_09.fm Page 272 Tuesday, July 2, 2002 5:32 PM

9.5.1 Generating Client Stubs Using `xrpcc`

To access a Web service, JAX-RPC clients usually obtain the WSDL document of the service and use **`xrpcc`** to generate client stubs based on the WSDL document. This is the approach we take for our Weather Web service client. Figure 9.48 is the configuration file that **`xrpcc`** uses to generate client stubs. The command

```
xrpcc -client -keep -d WeatherClient XmethodsUniSys.xml
```

generates the client-side classes for the Weather service in directory **`WeatherClient`**. Option **`-keep`** specifies that **`xrpcc`** should generate both **`.java`** and **`.class`** files, which helps the client-side programmer understand the input and return objects of the service's methods without referring to the WSDL document directly. **`XmethodsUniSys.xml`** is the configuration file shown in Fig. 9.48. Line 5 specifies the location of the service's WSDL document. Line 6 defines the client package **`weather`**.

9.5.2 Writing the Client

Once we have the client-side classes, writing a client is straightforward. Figure 9.49 is the client that connects to the Weather Web service to get the weather forecast. Line 19 imports the client package, which is defined in line 6 of Fig. 9.48. The constructor (lines 21–87) lays out the GUI components—a **`JButton`** that allows clients to get the service, a **`JPanel`** that displays general information (such as zip code, town), and a **`JPanel`** that displays a seven-day weather forecast.

Method **`getWeatherForecast`** (lines 90–114) takes a **`String`** that represents the zip code, which is the required input to the service's **`getWeather`** method. Lines 96–97 invoke method **`getWeatherServicesSoap`** of **`xrpcc`**-generated service-implementation class **`WeatherServices_Impl`** to get the service interface **`WeatherServicesSoap`**. Lines 100–101 invoke method **`getWeather`** of the service interface to get the weather forecast, which is contained in an object of class **`WeatherForecast`**. Class **`WeatherForecast`** is an **`xrpcc`**-generated class that corresponds to the return object of method **`getWeather`**.

Method **`createGeneralPanel`** (lines 117–144) adds five **`JTextField`**s to the first **`JPanel`** to display the general information, such as zip code and town name. Method **`createDailyPanel`** (lines 147–153) adds a **`JTextArea`** to the second **`JPanel`** to display the seven-day weather forecast. Methods **`updateGeneralContent`** (lines 156–169) and **`updateDailyContent`** (lines 172–201) are invoked each time the user clicks the **`JButton`** to update the display.

```
 1   <?xml version="1.0" encoding="UTF-8"?>
 2   <configuration
 3      xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
 4      <wsdl name="weather"
 5            location="http://hosting001.vs.k2unisys.net/Weather/
PDCWebService/WeatherServices.asmx?WSDL"
 6            packageName="weather">
```

Fig. 9.48 Configuration file used by **`xrpcc`** to generate client stubs. (Part 1 of 2.)

jws1_09.fm Page 273 Tuesday, July 2, 2002 5:32 PM

```
      <typeMappingRegistry>
      </typeMappingRegistry>
    </wsdl>
</configuration>
```

Fig. 9.48 Configuration file used by **xrpcc** to generate client stubs. (Part 2 of 2.)

```
// WeatherServiceClient.java
// WeatherServiceClient uses the Weather Web service
// to retrieve weather information.
package com.deitel.jws1.jaxrpc.weather;

// Java core packages
import java.rmi.*;
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;
import javax.swing.border.*;

// Java XML RPC packages
import javax.xml.rpc.*;

// Client packages
import weather.*;

public class WeatherServiceClient extends JFrame
{
   private JPanel generalPanel;
   private JPanel dailyPanel;

   private JTextField zipcodeField, cityField, timeField,
      sunriseField, sunsetField;

   private JTextArea dailyArea;

   // WeatherServiceClient constructor
   public WeatherServiceClient()
   {
      super( "Weather Forecast" );

      // create JButton for invoking Weather Web service
      JButton weatherButton =
         new JButton( "Get Weather Service" );
      weatherButton.addActionListener(

         new ActionListener() {

            public void actionPerformed( ActionEvent event )
            {
```

Fig. 9.49 Weather Web service client. (Part 1 of 5.)

jws1_09.fm Page 274 Tuesday, July 2, 2002 5:32 PM

```
               String zipcode = JOptionPane.showInputDialog(
                  WeatherServiceClient.this, "Enter zip code" );

               // if the user inputs zip code, get weather
               // forecast according to zip code
               if ( zipcode != null ) {

                  // get weather forecast
                  WeatherForecast weatherForecast =
                     getWeatherForecast( zipcode );

                  updateGeneralContent( weatherForecast );
                  updateDailyContent( weatherForecast );
               }

            } // end method ActionPerformed

         } // end ActionListener

      ); // end call to addActionListener

      // create JPanel for weatherButton
      JPanel buttonPanel = new JPanel();
      buttonPanel.add( weatherButton );

      // create JPanel for general weather forecast
      generalPanel = new JPanel();
      generalPanel.setBorder( new TitledBorder( "General" ) );
      createGeneralPanel();

      // create JPanel for daily weather forecast
      dailyPanel = new JPanel();
      dailyPanel.setBorder(
         new TitledBorder( "Daily Forecast" ) );
      createDailyPanel();

      // lay out components
      Container contentPane = getContentPane();
      contentPane.add( buttonPanel, BorderLayout.NORTH );
      contentPane.add( generalPanel, BorderLayout.CENTER );
      contentPane.add( dailyPanel, BorderLayout.SOUTH );

   } // end WeatherServiceClient constructor

   // get weather forecast from Unisys's WeatherService
   public WeatherForecast getWeatherForecast( String zipcode )
   {
      // connect to Web service and get weather information
      try {

         // get Web service
         WeatherServicesSoap weatherService = ( new
            WeatherServices_Impl() ).getWeatherServicesSoap();
```

Fig. 9.49 Weather Web service client. (Part 2 of 5.)

jws1_09.fm Page 275 Tuesday, July 2, 2002 5:32 PM

```

         // get weather information from server
         WeatherForecast result =
            weatherService.getWeather( zipcode );

         return result;

      } // end try

      // handle exceptions communicating with remote object
      catch ( Exception exception ) {
         exception.printStackTrace();

         return null;
      }

   } // end method getWeatherForecast

   // create general panel content
   public void createGeneralPanel()
   {
      generalPanel.setLayout( new GridLayout( 5, 2 ) );

      zipcodeField = new JTextField( 15 );
      cityField = new JTextField( 15 );
      timeField = new JTextField( 15 );
      sunriseField = new JTextField( 15 );
      sunsetField = new JTextField( 15 );

      zipcodeField.setEditable( false );
      cityField.setEditable( false );
      timeField.setEditable( false );
      sunriseField.setEditable( false );
      sunsetField.setEditable( false );

      generalPanel.add( new JLabel( "ZipCode: " ) );
      generalPanel.add( zipcodeField );
      generalPanel.add( new JLabel( "City: " ) );
      generalPanel.add( cityField );
      generalPanel.add( new JLabel( "Time: " ) );
      generalPanel.add( timeField );
      generalPanel.add( new JLabel( "Sunrise: " ) );
      generalPanel.add( sunriseField );
      generalPanel.add( new JLabel( "Sunset: " ) );
      generalPanel.add( sunsetField );

   } // end method createGeneralPanel

   // create daily panel content
   public void createDailyPanel()
   {
      dailyArea = new JTextArea( 20, 35 );

```

Fig. 9.49 Weather Web service client. (Part 3 of 5.)

```
      dailyPanel.add( new JScrollPane( dailyArea ) );

   } // end method createDailyPanel

   // update general forecast
   public void updateGeneralContent(
      WeatherForecast weatherForecast )
   {
      // parse WeatherForecast if not null
      if ( weatherForecast != null ) {
         zipcodeField.setText( weatherForecast.getZipCode() );
         cityField.setText(
            weatherForecast.getCityShortName() );
         timeField.setText( weatherForecast.getTime() );
         sunriseField.setText( weatherForecast.getSunrise() );
         sunsetField.setText( weatherForecast.getSunset() );
      }

   } // end method updateGeneralContent

   // update daily content
   public void updateDailyContent(
      WeatherForecast weatherForecast )
   {
      // get daily forecast if weatherForecast is not null
      if ( weatherForecast != null ) {
         ArrayOfDailyForecast dayForecast =
            weatherForecast.getDayForecast();

         // get DailyForecast array if dayForecast is not null
         if ( dayForecast != null ) {
            DailyForecast[] dailyForecast =
               dayForecast.getDailyForecast();

            StringBuffer results = new StringBuffer();

            // store daily forecast to results StringBuffer
            for ( int i = 0; i < dailyForecast.length ; i++ ) {
               results.append( dailyForecast[ i ].getDay() );
               results.append( ": \n   " );
               results.append( dailyForecast[ i ].getForecast() );
               results.append( "\n" );
            }

            dailyArea.setText( results.toString() );

         } // end inner if

      } // end outer if

   } // end method updateDailyContent

```

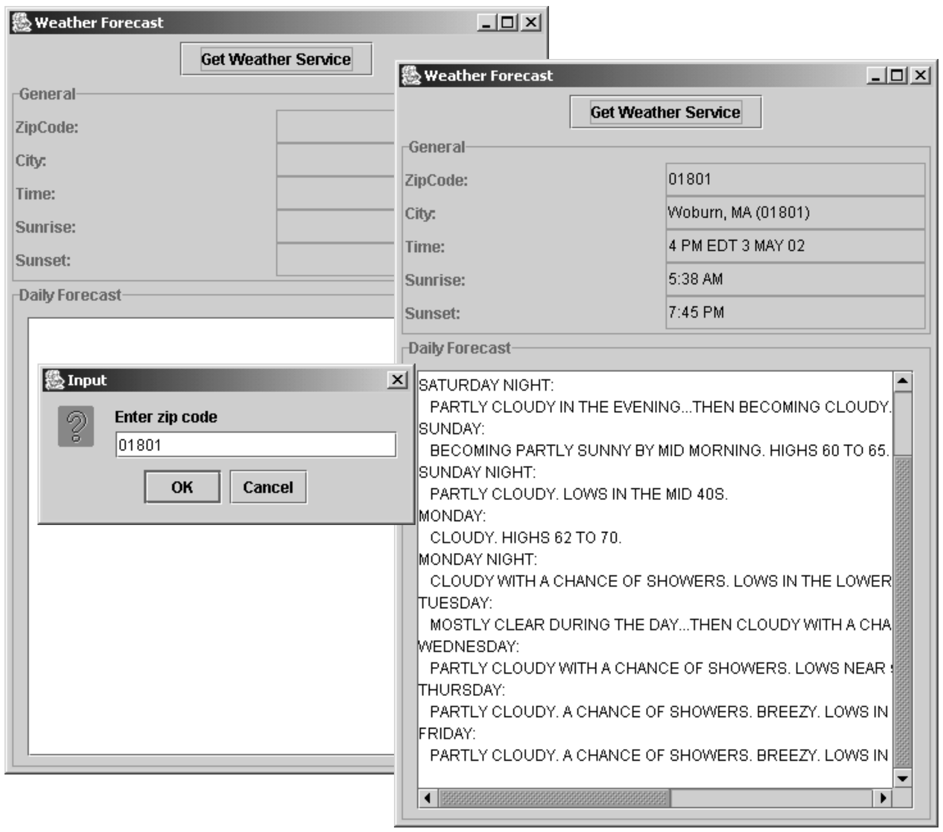
Fig. 9.49 Weather Web service client. (Part 4 of 5.)

jws1_09.fm Page 277 Tuesday, July 2, 2002 5:32 PM

```
   // execute WeatherServiceClient
   public static void main( String args[] )
   {
      WeatherServiceClient client =
         new WeatherServiceClient();
      client.setDefaultCloseOperation( EXIT_ON_CLOSE );
      client.setSize( 425, 600 );
      client.setVisible( true );

   } // end method main

} // end class WeatherServiceClient
```

Fig. 9.49 Weather Web service client. (Part 5 of 5.)

To compile and run the client, make sure your classpath includes all the JAR files mentioned in Fig. 9.46 and in directory **`unisys`**, which contains **`xrpcc`**-generated client classes. Figure 9.50 shows the client output.



01801

```
Weather Forecast
Get Weather Service
General
ZipCode:   01801
City:      Woburn, MA (01801)
Time:      4 PM EDT 3 MAY 02
Sunrise:   5:38 AM
Sunset:    7:45 PM
Daily Forecast
SATURDAY NIGHT:
  PARTLY CLOUDY IN THE EVENING...THEN BECOMING CLOUDY.
SUNDAY:
  BECOMING PARTLY SUNNY BY MID MORNING. HIGHS 60 TO 65.
SUNDAY NIGHT:
  PARTLY CLOUDY. LOWS IN THE MID 40S.
MONDAY:
  CLOUDY. HIGHS 62 TO 70.
MONDAY NIGHT:
  CLOUDY WITH A CHANCE OF SHOWERS. LOWS IN THE LOWER
TUESDAY:
  MOSTLY CLEAR DURING THE DAY...THEN CLOUDY WITH A CHA
WEDNESDAY:
  PARTLY CLOUDY WITH A CHANCE OF SHOWERS. LOWS NEAR
THURSDAY:
  PARTLY CLOUDY. A CHANCE OF SHOWERS. BREEZY. LOWS IN
FRIDAY:
  PARTLY CLOUDY. A CHANCE OF SHOWERS. BREEZY. LOWS IN
```

Fig. 9.50 Weather service output.

jws1_09.fm Page 278 Tuesday, July 2, 2002 5:32 PM

9.6 Summary

The Java Web Services Developer Pack (JWSDP) includes the Java XML Pack, Tomcat Java Servlet and JavaServer Pages containers, a Registry Server and the Ant build tool. As of this writing, the current version of JWSDP is 1.0.

RPC allows a procedural program to call a function that resides on another computer as conveniently as if that function were part of the same program running on the same computer. Whereas RPC requires the use of a single programming language and proprietary communications protocol, Web services enable integration among many different languages and protocols. As with RPC, both RMI and JAX-RPC handle the marshaling and unmarshaling of data across the network. Both RMI and JAX-RPC also provide APIs for transmitting and receiving data.

JAX-RPC provides a generic mechanism that enables developers to create Web services by using XML-based remote procedure calls. The JAX-RPC specification defines a mapping of Java types (e.g., **`int`**, **`String`** and classes that adhere to the JavaBean design pattern) to WSDL definitions. When a client locates a service in an XML registry, the client retrieves the WSDL definition to get the service-interface definition. To access the service using Java, the client must transform the WSDL definitions to Java types. The JAX-RPC client and service runtime environments send and process the remote method call and response, respectively.

JAX-RPC enables applications to take advantage of WSDL and SOAP's interoperability, which enables Java applications to invoke Web services that execute on non-Java platforms, and non-Java applications can invoke Web services that execute on Java platforms. JAX-RPC hides the details of SOAP from the developer, because the JAX-RPC service/client runtime environment performs the mapping between remote method calls and SOAP messages.

JAX-RPC supports only a subset of Java types, because the data types transmitted by the remote procedure calls must map to SOAP data types. JAX-RPC supports Java primitive types and their corresponding wrapper classes. JAX-RPC supports a subset of standard Java classes. Java arrays can be used in JAX-RPC as long as the member type of the array is one of the JAX-RPC-supported Java types.

There are some restrictions on the service-interface definition: It must implement interface **`java.rmi.Remote`**; all public methods must throw exception **`java.rmi.RemoteException`**; no constant declarations are allowed; all method parameters and return types must be JAX-RPC-supported Java types.

The JAX-RPC reference implementation provides the **`deploytool`** to deploy a JAX-RPC service onto Tomcat. The JAX-RPC reference implementation also provides the **`xrpcc`** tool to generate ties (server-side objects that represent the services), stubs and other service and client-side artifacts (such as a WSDL document).

If we supply **`xrpcc`** with a remote-interface definition, it generates stubs, ties, a WSDL document and a server-configuration file for use when deploying the Web service. If we supply **`xrpcc`** with a WSDL document, it generates stubs, ties, a server-configuration file and the remote-interface definition for use when developing a client. Most users use an existing WSDL document to access a Web service.

The stubs, ties, server and client-side artifacts are dictated by **`-client`**, **`-server`** and **`-both`** options of the **`xrpcc`** tool. The **`xrpcc`** tool's option **`server`** specifies that only server-side files should be generated. Option **`client`** specifies that only client-side files should be generated. Option **`both`** generates both of these artifacts.

jws1_09.fm Page 279 Tuesday, July 2, 2002 5:32 PM

Servlet **`JAXRPCServlet`** is a JAX-RPC implementation for dispatching the request to the Web service implementation. When the **`JAXRPCServlet`** receives an HTTP request that contains a SOAP message, the servlet retrieves the data that the SOAP message contains; it then dispatches the method call to the service-implementation class via the tie.

To access a Web service via a stub, we need to obtain the stub object. When **`xrpcc`** generates the stub, it uses the following convention: *serviceinterface*_**`Stub`**. When the **`xrpcc`** generates the service implementation, it uses the convention *servicename*_**`Impl`**, where *servicename* is the service name specified in the **`xrpcc`** configuration file.

The Dynamic Invocation Interface (DII) enables clients to call methods on Web services without knowing the service's stub information. Using DII to write clients is more complicated than using static stubs to write clients. However, DII clients are more flexible than are static-stub clients, because DII clients can specify the properties of remote procedure calls (such as Web-service names, remote-method input parameters, and remote-method return types) at runtime. To work with XML data types, DII clients must specify the mapping between Java types and XML data types. To dynamically invoke a method, DII clients need to create a **`Call`** object that enables dynamic invocation of a service port by invoking method **`createCall`** of interface **`Service`**.

Using static stubs requires the programmer to generate the stubs using **`xrpcc`**. Using DII requires the programmer to do more coding. Using dynamic proxies does not require stubs and extra coding. To use the dynamic proxy, the clients must have access to the WSDL document and be able to extract service-based information from the WSDL document.

9.7 Internet and World Wide Web Resources

`java.sun.com/webservices/`
Java technology and Web services at SUN.

`java.sun.com/xml/jaxrpc/`
This site contains links to JAX-RPC specifications, downloads, tutorials and lots more.

`archives.java.sun.com/archives/jaxrpc-interest.html`
Archives for people who are interested in JAX-RPC.

`developer.java.sun.com/developer/technicalArticles/WebServices/getstartjaxrpc/`
This site contains an article "Getting Started with JAX-RPC" that covers how to write JAX-RPC Web services and clients.

`developer.java.sun.com/developer/community/chat/JavaLive/2002/jl0402.html`
A chat session on JAX-RPC, from Java Developer Connection.

`forums.java.sun.com/forum.jsp?forum=331`
A discussion forum on Java technologies for Web Services, from the Java Developer Connection.

`www.fawcette.com/javapro/2002_05/magazine/features/shorrell/`
An article "Introducing JAX-RPC" from JavaPro.