67779

Галактионов В.В.
Б2-10-2003-141

ОБЪЕДИНЕННЫЙ ИНСТИТУТ ЯДЕРНЫХ ИССЛЕДОВАНИЙ

Дубна

Б2-10-2003-141

ДЕПОНИРОВАННАЯ ПУБЛИКАЦИЯ

У 340

Дубна ~~199~~ 2003

52-10-2003-141

Галактионов В.В.

Пакет JWSDP (Java Web Services Developer Pack) компании Sun для разработки и развертывания объектных распределенных систем по технологии Web Services

Описываются основные возможности JWSDP-пакета и их использование с точки зрения сугубо практического применения объектной технологии Web Services. Приводятся методика и примеры развертывания распределенных систем по этой технологии, подготовки и передачи данных в основных режимах – в SOAP-форматах и RPC (удаленный вызов процедур).

Работа выполнена в Лаборатории информационных технологий ОИЯИ.

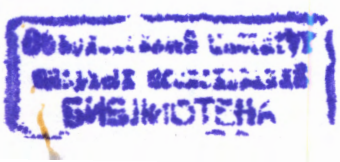
Galaktionov V.V.

Package JWSDP (Java Web Services Developer Pack) of Sun Company for Development and Deployment of the Object-Oriented Distributed Systems on Web Services Technology

The basic opportunities of a JWSDP package and technique of their application from the point of view of especially practical usage of object-oriented technology of Web Services are described. The distributed systems technique and deployment examples based on this technology along with data preparation and transfer in two main modes, namely, SOAP-format and RPC (remote procedures calls), are given in this work.

The investigation has been performed at the Laboratory of Information Technologies, JINR.

Рукопись поступила
в издательский отдел
05. 11. 2003



1. Состав и область применения пакета JWSDP

В августе 2002 г. компания Sun [1] предоставила для широкого пользования свободно распространяемый программный пакет **JWSDP 1.0_01** (Java Web Services Developer Pack) [2], реализующий концепции новой технологии **Web Services** (или Web-служб) [3,4] для объектно-ориентированных вычислительных распределенных систем согласно рекомендациям WWW-консорциума (W3C) [5].

Пакет представляет собой полностью укомплетованный набор программных средств, необходимых для разработки и развертывания Web-служб:

1. Пакет **Java XML Pack**, котрый содержит:
   - средства для передач SOAP/XML-сообщений - **JAXM** (Java API for XML Messaging);
   - средства для передач данных в режиме приложений – **SAAJ** (Soap with Attachments API for Java);
   - стандартные средства языка Java для обработки XML-документов – **JAXP** (Java API for XML Processing);
   - средства для регистрации Web-служб - **JAXR** (Java API for XML Registries);
   - средства реализации режима вызова удаленных процедур - **JAX-RPC** (Java API for XML-based RPC);
2. Web-сервер **Tomcat 4** для реализации протокола SOAP/HTTP (с поддержкой Java-сервлетов (спецификации Servlet 2.3) и JSP (JavaServer Pages 1.2);
3. Программную командную оболочку **ant** (Ant Build Tool 1.4.1) – для подготовки и развертывания клиентских и серверных программ Web-служб.

Для применения пакета JWSDP разработано учебное пособие "The Java Web Services Tutorial" [6], которым можно пользоваться в он-лайн режиме или скопировать для локального применения.

Пакет JWSDP подготовлен в двух вариантах – для операционных систем MS Windows NT/2000/XP и UNIX для всех аппаратных платформ, где возможно функционирование Java - (как минимум **SDK Java 2 SE v. 1.3**).

В данной публикации описываются основные возможности рассматриваемого пакета с точки зрения сугубо практического применения объектной технологии Web Services. Будут рассмотрены основные компоненты JWSDP и режимы обслуживания сервисов:

- Web-сервер **Tomcat**;
- командная оболочка **ant**;
- работа с Java-сервлетами и **JSP**;
- режим передач SOAP-сообщений, **JAXM**;
- режим вызова удаленных процедур, **JAX-RPC**;
- сервер регистрации Web-служб, **JAXR**.

Примечание. В состав JWSDP входит раздел **JAXP**, содержащий программные средства обработки XML-документов – одной из из важнейших технологических компонент Web-служб. Эта проблема не рассматривается в данной работе, поскольку основные элементы XML-технологии изложены в публикации автора [7] и, в большей части, являются стандартными средствами языка Java (в издании SDK Java 2 SE v. 1.3). Основные элементы JAXP, применяющиеся для обеспечения Web-служб в JWSDP:

- **SAX** (Simple API for XML) – для динамической обработки XML-документов, представлямых в виде файлов (синтаксический контроль (parsing) исходных

текстов, разбор структуры XML-контейнеров с применением *событийных хандлеров* (event handlers);
- **DOM** (Document Object Model) – представление XML-документа в виде иерархической объектной структуры элементов и атрибутов согласно принятой в W3C стандартной XML-модели [8];
- **XSLT** (The XML Stylesheet Language for Transformation) – конвертирование XML-документов на основании стилевых таблиц XSL [9].

Инсталляция пакета JWSDP

Инсталляция **JWSDP** в MS Windows выполняется стандартным способом:
- запуск исходного пакета в exe-формате и указание директории для инсталляции, например: **d:/jwsdp1.0/;**
- заполнение регистрационных форм для дальнейшего администрирования пакета и Web-сервера Tomcat;
- установка в переменной PATH пути доступа к **bin**-директории пакета (например: **d:/jwsdp1.0/bin**), в которой размещаются все исполнительные программы;

Примечание: Инсталляционная программа проверяет наличие в операционной системе Java-комплекта (JDK или JRE) не ниже версии SE 1.3.

2. Web-сервер Tomcat

Tomcat является Web-сервером [10], поддерживающим HTTP-протокол для обслуживания стандартных WWW-приложений, в частности HTML-документов, Java-сервлетов и JSP (JavaServer Pages). Инсталляция **Tomcat** выполняется одновременно с запуском пакета JWSDP. При этом:
- в исполнительной директории **/bin** будут записаны командные файлы (startup.bat, startup.sh) для запуска и (shutdown.bat, shutdown.sh) - для остановки сервера;
- в директории **/conf** записываются конфигурационные файлы server.xml и tomcat-users.xml;
- в директории **/logs** записывается статистика работы Tomcat;
- в директории **/webapps** записывается различного рода информация о приложениях, зарегистрированных для работы с Web-сервером. Так например, в **/webapps/root** помещаются HTML-документы, предназначенные для вызова через Web-сервер;
- порт Web-сервера по умолчанию принимает значение 8080 и задается в конфигурационном файле **server.xml**

Пример файла **tomcat-users.xml**:

```
<?xml version='1.0'?>
<tomcat-users>
  <role rolename="admin"/>
  <role rolename="manager"/>
  <role rolename="provider"/>
<user username="galaktion" password="asdfgh" fullName="Galaktionov Victor"
roles="admin,manager,provider"/>
</tomcat-users>
```

Пример обращения к приложениям Tomcat:
http://js.jinr.ru:8080/index.html

Примечание: При обращениях (явно или косвенно) к серверу Tomcat зарегистрированные данные пользователя должны находиться в **home**-директории пользователя (для Windows NT – в директории "Documents and Settings") в файле **build.properties** в простом формате, например:
username=galaktion
password=asdfgh

3. Командная оболочка ant (Ant Build Tool)

Языком программирования для Web-служб в JWSDP является Java, поэтому подготовка (компиляция, архивирование, подключение библиотек и исполнение (интерпретация) программ) как клиентских, так и серверных приложений выполняются стандартными средствами Java-комплекта (JDK). Кроме того, для развертывания Web-сервисов используются утилиты для генерации служебных конфигурационных файлов и вспомогательных программ (стабов). Например, для выполнения простейшего демонстрационного теста надо выполнить более десяти "ручных" запусков процедур. Для упрощения процесса подготовки программ в большинстве случаев предлагается использовать командную оболочку **ant** (или Ant Build Tool) [11]. Работа **ant** выполняется под управлением конфигурационного файла **build.xml**, в котором в XML-формате перечисляются типы операций и конкретизация каждой из них. При этом build.xml может содержать ссылки на другие конфигурационные файлы, например на файл targets.xml (см. ниже). Каждая из операций описывается в контейнере **<target>**, а комбинирование операций выполняется включением в оператор **depends** списка операций-контейнеров.
Пример:

Файл **build.xml:**

```
<!DOCTYPE project [
  <!ENTITY commonTargets SYSTEM "../common/targets.xml">
]>
<project name="JAX-RPC Client  Example" default="build" basedir=".">
 . . . . . . . . . . . . . . . . . .
  &commonTargets; <!-- The ant targets are in ../common/targets.xml -->
. . . . . . . . . . . . . . . . . . .
</project>
```

Фрагмент файла **targets.xml:**

```
<!-- targets.xml: Referenced by the build.xml files, this file
    contains targets common to all of the jaxrpc examples.  -->
. . . . . . . . . . . . . . . . . . .
<target name="prepare"
    description="Creates the build and dist directories" >
    <echo message="Creating the required directories...." />
    <mkdir dir="${build}/client/${example}" />
    <mkdir dir="${build}/server/${example}" />
    <mkdir dir="${build}/shared/${example}" />
    <mkdir dir="${build}/wsdeploy-generated" />
    <mkdir dir="dist" />
    <mkdir dir="${build}/WEB-INF/classes/${example}" />
```

```
</target>
<target name="compile-client" depends="prepare"
   description="Compiles the client-side source code" >
   <echo message="Compiling the client source code...."/>
   <javac
     srcdir="."
     destdir="${build}/client"
     classpath="${jwsdp-jars}:build/shared:build/client"
     includes="*Client.java"
   />
</target
```

Команда **ant compile-client** вызовет последовательность действий, сконфигурированных в файле build.xml и targets.xml:

- **prepare** – подготовку директорий,
- **compile-client** – вызов Java-компилятора **javac** с параметрами.

4. Сервлеты в JWSDP

В сервлетах реализуется Java-технология обеспечения CGI-режима (Common Gateway Interface) для динамического использования Web-приложений. Сервлеты в JWSDP предназначены в первую очередь для "внутреннего потребления", т. е. – использование Web-сервера для передач SOAP-сообщений Web-служб в так называемых *сервлет-контейнерах*. Но используя поддержку сервлетов в Tomcat, можно использовать этот механизм в явном виде. Более того, применяя комбинирование средств сервлетов и клиентских приложений Web-служб, можно обеспечивать доступ к услугам серверных Web-приложений (Web-сервисов) по HTTP-протоколу из HTML-документов под управлением стандартных Web-браузеров, используя цепочку, изображенную на рисунке 1.

Рис. 1. Диаграмма использования сервлетов для Web-сервиса.

4.1. Развертывание сервлетов в JWSDP

Рассмотрим простой пример сервлета вне всякой связи с JWSDP. В HTML-документе реализуется вызов CGI-программы (в данном случае это будет сервлет) при нажатии кнопки "Submit":

Файл servlet.html:

```
<html>
<body>
<form action="http://js.jinr.ru:8080/servlets/simpleServlet"
    method="get">
     <input type="submit" value="Submit">
</form>
</body>
</html>
```

Простейший вариант обработки GCI-запроса реализуется Java-программой в виде сервлета:

Файл **helloServlet.java:**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class helloServlet extends HttpServlet {

  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException, IOException {

    response.setContentType("text/html");
    response.setBufferSize(8192);

    PrintWriter out = response.getWriter();
    out.println("<html>" +
          "<head><title>Hello</title></head><body>");

    String username = request.getParameter("username");
    if ( username != null && username.length() > 0 )
        out.println("<H1>Your name is " + username + "</H1>")
    else
        out.println("<H1> Please, enter correct value!</H1>");

    out.println("</body></html>");
    out.close();
  }
}
```

Для подключения сервлета к Web-серверу Tomcat по JWSDP-технологии надо выполнить последовательность действий:

- выбрать директорию для развертывания окружения сервлета, например: **d:\jwsdp1.0\myExamples\simple;**
- создать рабочие директории **\src** и **\web-servlet\Web-inf;**
- подготовить файлы:
  - **build.xml** - конфигурационный файл для утилиты **ant** в текущей директории,
  - **web.xml** - конфигуратор сервлета в директории \web-servlet\Web-inf,
  - **helloServlet.java** – текст программы-сервлета в директории \src.

Значения параметров:

- имена директорий **src** и **webServlet** задаются в файле **build.xml**:
  <javac srcdir="**src**" destdir="${build}/WEB-INF/classes">
  <fileset dir="**webServlet**/WEB-INF" >
  <fileset dir="**webServlet**">
- сетевой адрес сервлета определяется параметром <url-pattern>/**simpleServlet**</url-pattern> в файле web.xml и значением value (<property name="example" value="**servlets**" />) в файле build.xml. Таким образом, URL сервлета при обращении через Tomcat будет иметь вид **http://js.jinr.ru:8080/servlets/simpleServlet**

Файл **web.xml** – конфигуратор сервлета:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
'http://java.sun.com/dtd/web-app_2_3.dtd'>
<web-app>
```

```
  <display-name>Hello</display-name>
  <description>no description</description>
  <servlet>
      <servlet-name>helloMyServlet</servlet-name>
      <display-name>helloMyServlet</display-name>
      <description>no description</description>
      <servlet-class>helloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>helloMyServlet</servlet-name>
      <url-pattern>/simpleServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Конфигурационный файл **build.xml**:

```
<project name="servlet-example" default="build" basedir=".">
  <target name="init">
    <tstamp/>
  </target>
  <property name="example" value="servlets" />
  <property name="path" value="/${example}"/>
  <property name="build"   value="${jwsdp.home}/myExamples/${example}/build" />
  <property name="url"     value="http://localhost:8080/manager"/>
  <property file="build.properties"/>
  <property file="${user.home}/build.properties"/>
  <path id="classpath">
   <fileset dir="${jwsdp.home}/common/lib">
    <include name="*.jar"/>
   </fileset>
  </path>
  <taskdef name="install"  classname="org.apache.catalina.ant.InstallTask"/>
  <target name="prepare" depends="init"
          description="Create build directories.">
   <mkdir dir="${build}" />
   <mkdir dir="${build}/WEB-INF" />
   <mkdir dir="${build}/WEB-INF/classes" />
  </target>
<target name="install" description="Install web application"
        depends="build">
   <install url="${url}" username="${username}" password="${password}"
      path="${path}" war="file:${build}"/>
  </target>
  <target name="build" depends="prepare"
    description="Compile app Java files and copy HTML and JSP pages" >
   <javac srcdir="src" destdir="${build}/WEB-INF/classes">
          <include name="**/*.java" />
          <classpath refid="classpath"/>
   </javac>
   <copy todir="${build}/WEB-INF">
    <fileset dir="webServlet/WEB-INF"    >
     <include name="web.xml" />
    </fileset>
   </copy>
</target>
</project>
```

Конфигурационные файлы **build.xml** и **web.xml** полностью определяют последовательность развертывания и способ обращения к сервлету:

- команда **ant install** согласно предписанию в build.xml выполнит процедуры **build, prepare, init**, т.е. все операции развертывания сервлета: подготовку директорий, компиляцию Java-программ, архивирование Java-классов, копировние программ и регистрацию сервлета в Web-сервере Tomcat.
- Обращение к сервлету будет выполнено при активизации вышеприведенного HTML-файла **servlet.html** каким-либо образом, например, загрузкой в Web-браузер, локально или через сеть.

Примечания:

1. Операция регистрации сервлета выполняется при запущенном Tomcat.
2. Пользователь, выполняющий эту процедуру, в свою очередь, должен быть авторизован, т.е. зарегистрирован администратором Web-сервера.

5. Применение JSP (JavaServer Pages) в JWSDP

JSP page – это документ составленный комбинацией HTML-фрагментов и JSP-включений. В простом варианте это могут быть просто Java-операторы. Web-сервер Tomcat может выполнять интерпретацию JSP-документов и готовить для передачи Web-браузеру данные в формате HTML. Это достаточно мощное средство генрации данных для Web-приложений, успешно используемое, в частности, для написания сервлетов. Можно привести пример сервлета из предыдущего раздела, написаныый по JSP-технологии:

Файл **greeting.jsp**:

```
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
<%
  String username = request.getParameter("username");
   if ( username != null && username.length() > 0 ) {
%>
<h2><font color="black">Hello, <%=username%> !</font></h2>
<%
   } else {
%>
 <h2><font color="black">Please, enter correct name <font></h2>
<% } %>
</body>
</html>
```

Способ применения JSP для Tomcat аналогичен применению сервлетов: меняются некоторые конфигурационные параметры в файлах **build.xml** и **web.xml**:

Файл **web.xml:**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
'http://java.sun.com/dtd/web-app_2_3.dtd'>
<web-app>
 <display-name>Hello2</display-name>
 <description>no description</description>
 <servlet>
  <servlet-name>greeting</servlet-name>
  <display-name>greeting</display-name>
  <description>no description</description>
```

```
    <jsp-file>/greeting.jsp</jsp-file>
  </servlet>
  <servlet-mapping>
    <servlet-name>greeting</servlet-name>
    <url-pattern>/helloJSP</url-pattern>
  </servlet-mapping>
</web-app>
```

Файл build.xml:

```
<project name="myJSP-example" default="build" basedir=".">
  <target name="init">
    <tstamp/>
  </target>
  <property name="example" value="myJSP" />
  <property name="path" value="/${example}"/>
  <property name="build"   value="${jwsdp.home}/docs/tutorial/examples/web/${example}/build" />

 <property name="url"      value="http://localhost:8080/manager"/>
  <property file="build.properties"/>
  <property file="${user.home}/build.properties"/>

  <path id="classpath">
    <fileset dir="${jwsdp.home}/common/lib">
      <include name="*.jar"/>
    </fileset>
  </path>
<taskdef name="install"  classname="org.apache.catalina.ant.InstallTask"/>
<taskdef name="remove" classname="org.apache.catalina.ant.RemoveTask"/>
  <target name="prepare" depends="init"
          description="Create build directories.">
    <mkdir dir="${build}" />
    <mkdir dir="${build}/WEB-INF" />
    <mkdir dir="${build}/WEB-INF/classes" />
  </target>
  <target name="install" description="Install web application"
        depends="build">
    <install url="${url}" username="${username}" password="${password}"
      path="${path}" war="file:${build}"/>
  </target>
  <target name="remove" description="Remove web application">
    <remove url="${url}" username="${username}" password="${password}"
      path="${path}"/>
  </target>
  <target name="build" depends="prepare"
    description="Copy DD, HTML, and JSP pages" >
    <copy todir="${build}/WEB-INF">
      <fileset dir="webJSP/WEB-INF"    >
        <include name="web.xml" />
      </fileset>
    </copy>
    <copy todir="${build}">
      <fileset dir="webJSP">
<include name="*.jsp" />
</fileset>
    </copy>
  </target>
</project>
```

Развертывние JSP-программ выполняется аналогично сервлетам:

- в текущей директории, например **d:\jwsdp1.0\myExamples\myJSP,** создаются подддиректории **\webJSP** и **\webJSP\web-inf**;
- записывается JSP-файл (сервлет) **greeting.jsp** в директорию **\webJSP;**
- записывается конфигурационный файл сервлета **web.xml** в директорию **\webJSP\web-inf**;
- в текущей директории записывается конфигурационный файл **build.xml** для утилиты **ant;**
- при работающем Web-сервере Tomcat выполняется процедура **ant install.** При этом выполняются инсталляционные действия, как и при развертывании Java-сервлетов (компиляция, регистрация и др.).

После успешной инсталляции JSP-сервлет становится доступным через сеть и Web-сервер, в соответствии с конфигурационными параметрами, по адресу **http://js.jinr.ru:8080//myJSP/helloJSP** из HTML-документа:

```
<html>
<body>
<form action="http://js.jinr.ru:8080/myJSP/helloJSP"  method="get">
    <input type="text" name="username">
    <input type="submit" value="Submit">
</form>
</body>
</html>
```

6. JAXM API - приложение пакета JWSDP для управления SOAP/XML-сообщениями

JAXM (Java API for XML Messaging) соответствует спецификациям протокола SOAP 1.1 (Simple Object Access Protocol). Полный вариант JAXM API представлен в двух пакетах:

- **javax.xml.soap** - пакет, определенный в спецификациях SOAP для Java-приложений (**SAAJ 1.1,** SOAP with Attachments API for Java). Это - основной пакет для управления SOAP-сообщениями, который содержит API для создания и заполнения SOAP/XML сообщения.
- **javax.xml.messaging** - пакет, определенный в спецификациях JAXM 1.1. Этот пакет содержит API, необходимые для посылок *односторонних* сообщений..

6.1. Краткий обзор JAXM

Краткий обзор JAXM содержит три части:

- Сообщения.
- Соединения.
- Провайдеры сообщений

Сообщения

JAXM-сообщения соответствуют SOAP-стандартам, которые предписывают формат XML-сообщений и также определяют состояния или действия трех типов: *обязательные* (required), *дополнительные* (optional) или *неразрещенные* (not allowed).

Структура XML-документа в SOAP-сообщении

XML-документ имеет иерархическую структуру с элементами, подэлементами, и так далее. Можно отметить, что многие из классов SAAJ и интерфейсов соответствуют XML-элементам в SOAP-сообщениях и в своих названиях содержат слова **element** или **SOAP,** или оба слова вместе.

Элементы также носят названия *узлов* (nodes). Таким образом, SAAJ API имеет интерфейс **Node**, который является родительским классом (суперклассом) для всех классов и интерфейсов, которые представляют XML-элементы в SOAP-сообщении. Можно привести названия некоторых методов этих классов: **SOAPElement(), AddTextNode(), Node.detachNode()** и **Node.getValue()**.

Структура SOAP-сообщения

Существуют два главных типа SOAP-сообщений - с приложениями (attachments) и без приложений.

Сообщения без приложений

Следующая схема показывает высший уровень структуры SOAP-сообщения без приложений. Все внесенные в список части являются обязательными, кроме заголовка (SOAP header).

SOAP message

A. SOAP part

1. SOAP envelope
   a. SOAP header
   b. SOAP body

Соответственно, SAAJ API содержит класс SOAPMessage, чтобы представить элемент **SOAP message** , класс SOAPPart - для представления элемента **SOAP part**, класс SOAPEnvelope – для элемента **SOAP envelope** и так далее.

Когда создается новый объект SOAPMessage, обязательно должны быть сформированы все необходимые составные части SOAP-сообщения. Другими словами, новый объект SOAPMessage будет содержать объект SOAPPart, который включает объект SOAPEnvelope. Объект SOAPEnvelope, в свою очередь, автоматически включает пустой объект SOAPHeader, сопровождаемый пустым объектом SOAPBody. Если нет необходимости включать в сообщение объект SOAPHeader, который является необязательным, его можно удалить.

Объект SOAPHeader может содержать один или несколько заголовков (headers) с информацией относительно посылающей и получающей сообщения сторон, а также данные о промежуточных передающих пунктах. Объект SOAPBody, который всегда следует за объектом SOAPHeader, если таковой имеется, обеспечивает простой способ передачи полезной (mandatory) информации конечному получателю. Если имеется объект SOAPFault (для регистрации ошибок), он должен находиться в объекте SOAPBody. На рисунке 1 показана структура SOAP-сообщения без приложений.

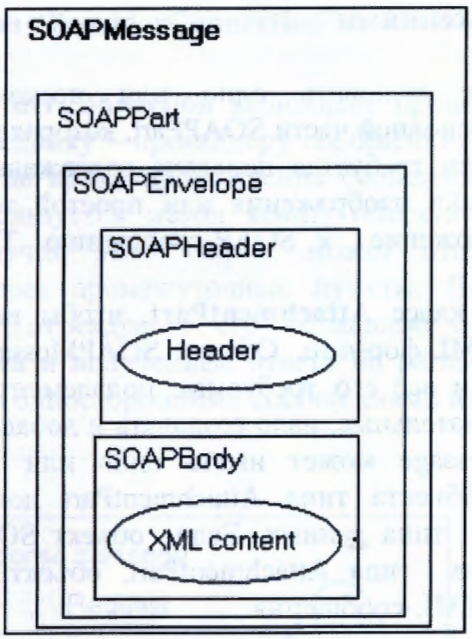


Рис 1. Объект **SOAPMessage** без приложений.

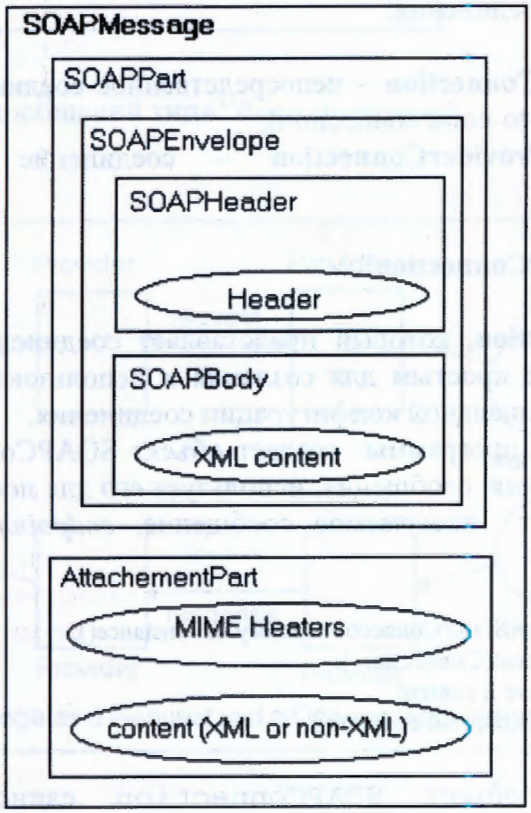


Рис 2. Объект **SOAPMessage** с одним attach-приложеним

SOAP-сообщения с приложениями

SOAP-сообщение может включать одно или несколько приложений (attachments) в дополнение к основной части SOAPPart, которая может содержать только XML-содержание. Если требуется передача содержания (content) не в формате XML, например, файл изображения или простой текст, оно должно передаваться как attach-приложение к SOAP-сообщению. Таких приложений может быть несколько.

SAAJ API обеспечивает класс **AttachmentPart**, чтобы включать в SOAP-сообщения данные не в XML-формате. Объект SOAPMessage автоматически включает объект SOAPPart и все его требуемые подэлементы. Но, поскольку объекты AttachmentPart необязательные, надо создавать и добавлять их отдельной операцией. Объект SOAPMessage может иметь одно или нескольеко таких приложений. Для каждого объекта типа AttachmentPart должен указываться MIME-заголовок для задания типа данных. Если объект SOAPMessage имеет один или несколько объектов типа AttachmentPart, объект SOAPPart может содержать или не содержать XML-сообщения.

Соединения

Все SOAP-сообщения передаются посредством *соединений.* Соединение может выполняться непосредственно к определенному потребителю или к *провайдеру сообщений.* JAXM API предоставляет классы и интерфейсы для выполнения двух типов соединения:

1. **javax.xml.soap.SOAPConnection** – непосредственное соединение отправителя с приемником (a point-to-point connection);
2. **javax.xml.messaging.ProviderConnection** – соединение с провайдером сообщений.

Соединение типа "SOAPConnection"

Объект **SOAPConnection**, который представляет соединение типа **point-to-point,** является наиболее простым для создания и использования. Для него не требуется проведения специальной конфигурации соединения.

Следующий фрагмент программы создает объект SOAPConnection и затем, после создания и заполнения сообщения, использует его для посылки сообщения. Параметр *request* задает посылаемое сообщение, *endpoint* - представляет направление передачи.

```
SOAPConnectionFactory factory = SOAPConnectionFactory.newInstance();
SOAPConnection con = factory.createConnection();
 // create a request message and give it content
SOAPMessage response = con.call(request, endpoint);
```

Когда используется объект `SOAPConnection`, единственный способ посылать SOAP-сообщение - использовать метод **call()**, который передает это сообщение и блокирует передающий процесс на время получения ответа. По этой причине этот метод передачи сообщений называется *request-response messaging.*

Реализация Web-служб для такого режима обязательно требует ответа провайдера – для *возврата результата* запроса, *подтверждения* запроса, либо любого ответа для *разблокировки* передающего процесса.

Соединение типа "ProviderConnection"

Объект **ProviderConnection** выполняет процедуру "соединение" и передачу сообщения посреднику – провайдеру сообщений. При этом передающий процесс не блокируется на время прохождения сообщений в сети. Для передачи такого сообщения используется метод **send()** объекта ProviderConnection. Провайдер сообщений получив этот запрос, может отправить его предназначенному потребителю через промежуточные пункты. Впоследствии, после получения ответа, провайдер переадресует его пославшему запрос процессу. Интервал между посылкой запроса и получением ответа не регламентируертся. Передачи такого типа называются односторонними сообщениями или *one-way* messaging.

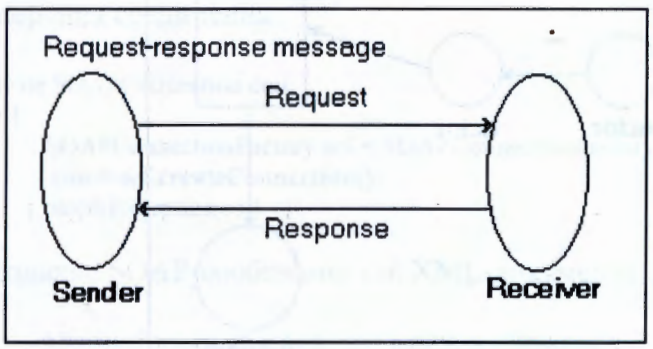


Рис 3. Передача сообщений типа "Request-response".

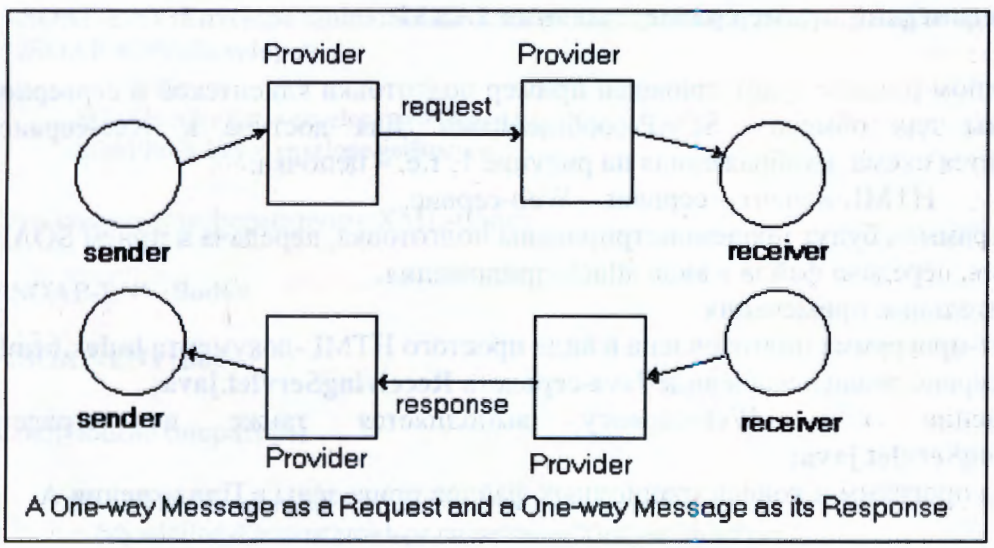


Рис 4. Передача сообщений типа "One-way Messaging".

Передача сообщений с промежуточными пунктами "Intermediate Destinations"

При использовании провайдера соединений может применяться режим передачи сообщений конечному потребитель с промежуточными станциями. Эти станции, называемые *actors*, определяются в объекте-заголовке SOAPHeader SOAP-сообщения. Промежуточные станции могут выполнять отметки о прохождении сообщении, используя метод addAttribute() объекта SOAPHeader.

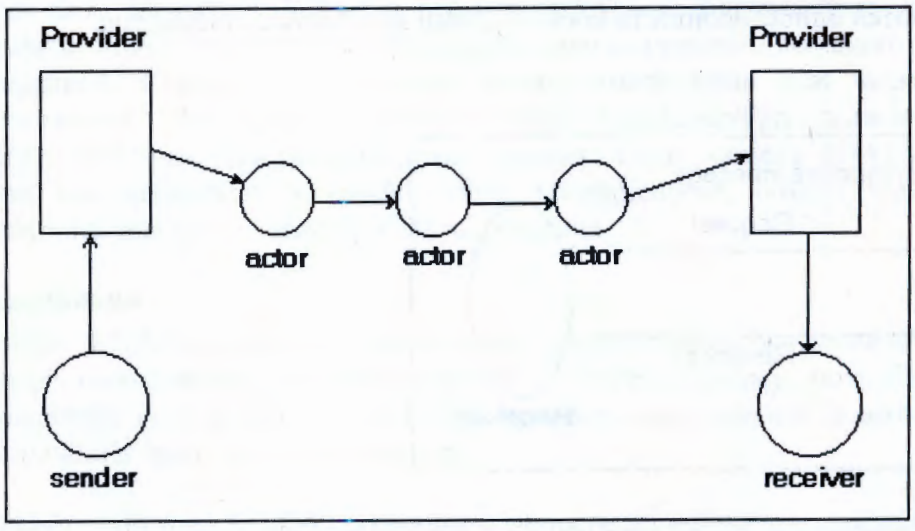


Рис 5. Односторонние SOAP-сообщения с промежуточными станциями.

6.2. Практический пример развертывания JAXM

В данном разделе будет приведен пример подготовки клиентской и серверной программы для обмена SOAP-сообщениями. Для доступа к Web-сервису используется схема, ихображенная на рисунке 1, т.е. – цепочка:

HTML-клиент – сервлет – Web-сервис.

В программах будут продемонстрированы подготовка, передача и прием SOAP-сообщения, передача файла в виде attach-приложения.

Предварительные примечания:

- клиент-программа подготовлена в виде простого HTML-документа **index.html**;
- Web-сервис реализован в виде Java-сервлета **ReceivingServlet.java;**
- обращение к Web-сервису выполняется также из сервлета **SendingServlet.java;**
- тексты программ и конфигурационных файлов приведены в Приложении А.

Подготовка сервлетов выполняется по схеме, описанной в разделе 3 (Сервлеты в JWSDP) с некоторыми модификациями:

- адрес директории для развертывания сервлетов: **d:\jwsdp1.0\myExamples\jaxm\simple**;
- тексты программ-сервлетов размещаются соответственно в поддиректориях **source\simple\sender** и **source\simple\receiver;**

- создаются конфигурационные файлы **build.xml** (для процедуры **ant**), **web.xml** (в поддиректории **web**) и **build.properties** (дополнительная информация для build.xml);
- инсталляция примера (компиляция Java-программ, подключение к Tomcat и др.) выполняется командой **ant install**.

Кодирование программ

Ниже будут приведены комментарии к программированию сервлетов для выполнения SOAP-протокола и включение файла для передачи в качестве attach-приложения.

Передача SOAP-сообщения из клиента-сервлета SendingServlet.java.

Операция соединения:

```
private SOAPConnection con
try {
      SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
       con = scf.createConnection();
   } catch(Exception e) {  }
```

Создание SOAP-сообщения как XML-документа:

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
SOAPPart sp = msg.getSOAPPart();
SOAPEnvelope envelope = sp.getEnvelope();
```

После выполнения этих операций формируется часть XML-документа:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
</SOAP-ENV:Envelope>
```

```
SOAPHeader hdr = envelope.getHeader();
SOAPBody bdy = envelope.getBody();
```

Эти операторы формируют XML-блок:

```
<SOAP-ENV:Body>
.....
</SOAP-ENV:Body>
```

Следующие операторы

```
SOAPBodyElement gltp
 = bdy.addBodyElement(envelope.createName("GetLastTradePrice",
"ztrade",
"http://wombat.ztrade.com"));
gltp.addChildElement(envelope.createName("symbol",
"ztrade",
"http://wombat.ztrade.com")).addTextNode("SUNW");
```

формируют XML-элемент

```
< ztrade:GetLastTradePrice xmlns: ztrade= "http://wombat.ztrade.com">
    < ztrade:symbol xmlns: ztrade= "http://wombat.ztrade.com">SUNW
    </ ztrade:symbol>
</ztrade :GetLastTradePrice>
```

Далее готовится информация для включения attach-приложения:

```
String data = "http://js.jinr.ru:8080/index.html";
URL url = new URL(data);
AttachmentPart ap = msg.createAttachmentPart(new DataHandler(url));
ap.setContentType("text/html");
msg.addAttachmentPart(ap);
```

Из контекста сервлета составляется сетевой адрес сервера **urlEndpoint**:

```
StringBuffer urlSB=new StringBuffer();
urlSB.append(req.getScheme()).append("://").append(req.getServerName());
urlSB.append( ":" ).append( req.getServerPort() ).append( req.getContextPath() );
String reqBase=urlSB.toString();
String to=reqBase + "/myReceiver";
URL urlEndpoint = new URL(to);
```

В файл **sent.msg** в директории Web-сервера записывается текст SOAP-сообщения:

```
FileOutputStream sentFile = new FileOutputStream("sent.msg");
msg.writeTo(sentFile);
sentFile.close();
```

И, наконец, передача SOAP-сообщения серверу и прием ответа **reply** в SOAP-формате:

```
SOAPMessage reply = con.call(msg, urlEndpoint);
```

Ответ reply записывается в файл **reply.msg**:

```
FileOutputStream replyFile = new FileOutputStream("reply.msg");
reply.writeTo(replyFile);
replyFile.close();
```

Прием SOAP-сообщения в сервере и возврат результата (сервлет ReceivingServlet.java)

Для приема SOAP-сообщения вызывается метод сервлета onMessage(). Входным параметром и возвращаемым результатом метода являются SOAP-сообщения:

SOAPMessage onMessage(SOAPMessage message)

Для ответа формируется SOAP-сообщение **msg**:

```
MessageFactory fac = MessageFactory.newInstance();
SOAPMessage msg = fac.createMessage();
SOAPEnvelope env = msg.getSOAPPart().getEnvelo
env.getBody().addChildElement(env.createName("Response"))
        .addTextNode("This is a response");
```

Содержательной частью ответа будет XML-элемент:

<Response> This is a response </Response>

Результат возвращается клиенту оператором **return msg;**

Расмотренный пример – очень простой, в нем не предусмотрена обработка принятого SOAP-сообщения. Это не представляет особых трудностей: используя структуру сообщения как объектное представление XML-документа, обработку сообщения можно выполнить следующим образом:

1. **Обработка XML-части (XML content) SOAP-сообщения**:

Формирование объекта SOAPPart и выделение XML-элементов сообщения:

```
SOAPPart sp = message.getSOAPPart();
 SOAPEnvelope env = sp.getEnvelope();
 SOAPBody sb = env.getBody();
 java.util.Iterator it = sb.getChildElements(bodyName);
SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
String res = bodyElement.getValue();
```

Если ответ содержит несколько XML-элементов, можно устраивать цикл:

```
while (it.hasNext()) {
    SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
    String res = bodyElement.getValue();
}
```

2. **Обработка принятых attach-приложений**

Выделение в цикле attach-приложений из SOAP-сообщения **message:**

```
java.util.Iterator it = message.getAttachments();
while (it.hasNext()) {
  AttachmentPart attachment = (AttachmentPart)it.next();
  Object content = attachment.getContent();
  String id = attachment.getContentId();
  System.out.print("Attachment " + id + " contains: " +
                   content);
  System.out.println("");
}
```

7. JAX RPC - приложение пакета JWSDP для управления удаленным вызовом процедур

Основы взаимодействия клиент-серверных приложений для режима RPC в Web-службах изложены в публикации автора [4].

Задачи серверного приложения:

- разработка интерфейса удаленного сервисного объекта, реализующего вычислительный сервис (Web Service);
- кодирование (реализация) методов сервисного объекта (implemention);
- создание описания методов объекта в WSDL-документе;
- декларация WSDL-документа;

Задачи клиент-приложения:

- обработка (поиск и анализ) WSDL-документа;
- связывание (binding) - подключение к серверу (провайдеру сервиса);
- формирование объектной ссылки для методов сервисного объекта;
- обращение к методам сервиса и получение результатов вычислений.

Надо отметить существенные моменты использования RPC:

- поскольку единственным форматом передач данных в технологии Web-служб являются SOAP-сообщения, поэтому все вызовы процедур и передача параметров *конвертируются* в стандартный формат, принятый в SOAP, т.е. в XML-структуру. Это взаимное конвертирование SOAP-сообщений и RPC-запросов выполняют специальные программы – представители противоположной стороны стабы (**stubs**) и **tie**-программы. Эти программы-посредники генерируются автоматически при подготовке и развертывании клиентских и серверных компонент Web-служб;
- в режиме RPC существенную роль играет WSDL-описания интерфейсов сервисных программ.

На рисунке 6 представлена диаграмма прохождения и конвертирования RPC-запросов.

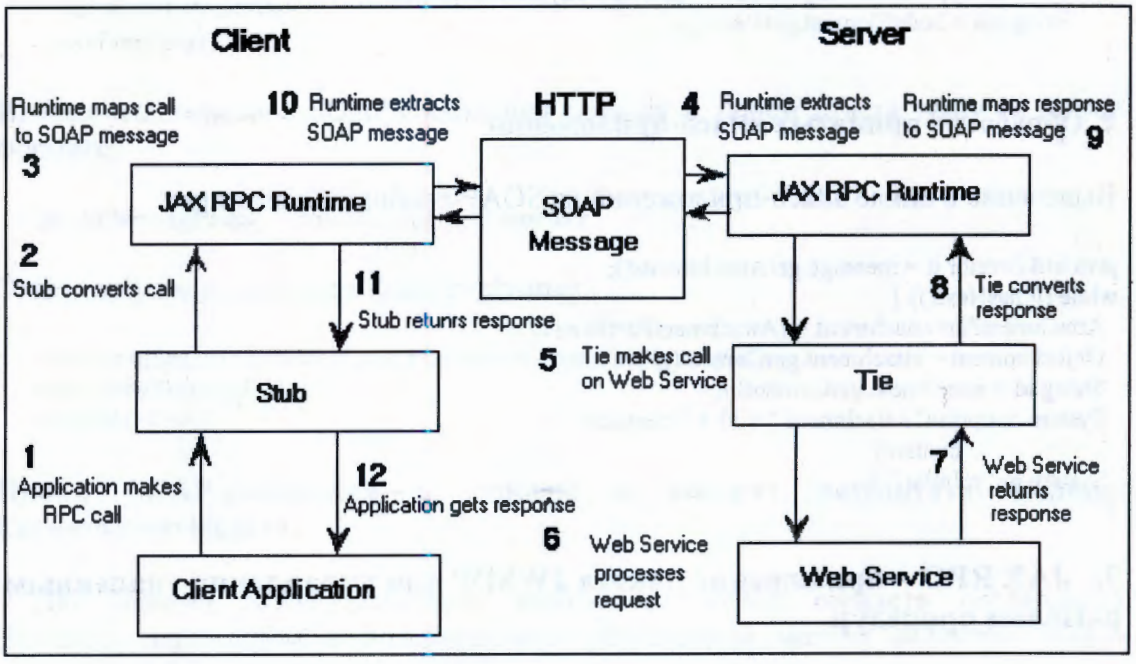


Рис. 6. Диаграмма преобразований RPC-вызовов в SOAP-сообщения.

7.1. Примеры практического развертывания клиентских и серверных приложений для режима RPC

Основные моменты развертывания Web-служб для RPC:

- все операции подготовки программ (компилирование, сборка, генерация программ-посредников) выполняются с применением командной оболочки **ant**, которая управляется конфигурационным файлом **build.xml**;
- для приближения к реальным ситуациям клиентские и серверные приложения готовятся раздельно (в разных директориях) со своими конфигурационными файлами;
- единственной основой подготовки клиент-приложения является доступ (локальный или сетевой) к wsdl-файлу описания интерфейса. Генерация Java-интерфейса из WSDL-описаний (mapping) выполняется утилитой **wscompile**;
- все операции развертывания Web-сервиса и доступа к нему выполняются при работающем Web-сервере Tomcat;
- для каждой RPC-задачи готовятся свои конфигурационные файлы **build.xml, web.xml, jaxrpc-ri.xml, build.properties**. Некторые из них содержат стандартные блоки, поэтому, при их создании можно использовать стандартные заготовки из других задач.

Подготовка серверного приложения

Выбор директории. Принципиального значения выбор директории не имееет, но полное ее название (все пути доступа) должно указываться при конфигурации сервиса. В примере это значение: **d:\jwsdp1.0\myExamples\rpc\Server.** В этой директории создается поддиректория **\stockCoffee** (по названию пакета **package stockCoffee**) и в ней готовятся все программы. В данном примере вычислительный сервис реализован в трех программах **CoffeeIF.java, CoffeeImpl.java, Coffee.java.**

Определение интерфейса Web-сервиса. Интерфейс сервиса задан Java-интерфейсом в файле **CoffeeIF.java:**

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

public interface CoffeeIF extends Remote {
  public boolean   insertCoffee(String coffeeName, float price)  throws RemoteException;
  public String    orderCoffee(String coffeeName, float quantity) throws RemoteException;
  public float[]   getPriceList() throws RemoteException;
  public String[]  getNameList()  throws RemoteException;
  public String    readIP(String s) throws RemoteException;
}
```

Реализация интерфейса. Реализация приведенного выше интерфейса (implementation) выполнена в программе **CoffeeImpl.java**:

```
package stockCoffee;
import java.net.*;

public class CoffeeImpl implements CoffeeIF {
   public int     max  = 10;
   public int     count = 0;
   public  Coffee[] cf   = new Coffee[max];

    public CoffeeImpl() {
      for(int i = 0; i < max; i++) {
        cf[i] = new Coffee();
```

```
        }
        cf[0].setCoffee("Arabica", (float)12.50);
        cf[1].setCoffee("Mocco", (float)9.30);
        count = 2;
    }
    public boolean insertCoffee(String nm, float pr)  {
        if(count < 9) {
            cf[count].setCoffee(nm, pr);
            count++;
            return true;
        } else return false;
    }
    public float[] getPriceList()  {
        if(count == 0) return null;
        float[] price = new float[count];
        for(int i=0; i < count; i++)
            price[i] = cf[i].getPrice();
     return price;
    }

    public String[] getNameList()  {
        if(count == 0) return null;
        String[] names = new String[count];
        for(int i=0; i < count; i++)
            names[i] = cf[i].getName();
        return names;
    }

    public String orderCoffee(String coffeeName, float quantity) {
        return "Order: (" + coffeeName + ", " + quantity + ") - OK";
    }

    public String readIP(String s) {
        InetAddress ip = null;
        try {
            ip = InetAddress.getLocalHost();
        } catch (Exception e)       {}
        String sip = ip.toString();
        return  s + sip;
    }
}
```

Дополнительная программа. В программе будет использоваться объект для передачи клиенту структурированных данных. Объект реализуется программой **Coffee.java**:

```
package stockCoffee;
public class Coffee {
  public String name;
  public float  price;

  public Coffee() {
    name = "foo";
    price = (float)1000.0;
  }
  public void setCoffee(String nm, float pr) {
      name = nm;
      price = pr;
  }
  public String getName()              { return name;      }
  public void   setName(String nm)  { this.name = nm; }
```

```
  public float  getPrice()          { return price;     }
  public void   setPrice(float pr)  { this.price = pr;  }
}
```

Конфигурационные файлы

Файл **build.properties**. Файл носит вспомогательный характер, в частности может использоваться для опеределения общих переменных.

```
root=${jwsdp.home}/myExamples
example=stockCoffee
context-path=stockCoffee-server
portable-war=${example}-portable.war
deployable-war=${context-path}.war
war-path=${root}/rpc/server/${example}/dist/${deployable-war}
```

В файле **build.xml** сконфигурированы операции для оболчки **ant.** В примере видно, что описания части операций вынесены в отдельный файл **targets.xml**, для совместного использования с другими задачами. Текст этого файла приводится в Приложении Б.

```
<!DOCTYPE project [
  <!ENTITY commonTargets SYSTEM "../common/targets.xml">
]>
<project name="JAX-RPC Server Example" default="build" basedir=".">
  <property file="${user.home}/build.properties"/>
  <property file="../common/build.properties"/>
  <property file="build.properties"/>
  &commonTargets; <!-- The ant targets are in ../common/targets.xml -->
  <target name="build" depends="build-service"
    description="Executes the targets needed to build the service.">
  </target>
</project>
```

Файлы **jaxrpc-ri.xml и web.xml** используются для генерации WSDL-описания сервиса.

Файл **jaxrpc-ri.xml:**

```
<?xml version="1.0" encoding="UTF-8"?>
<webServices
   xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
   version="1.0"
   targetNamespaceBase="http://com.test/wsdl"
   typeNamespaceBase="http://com.test/types"
   urlPatternBase="/ws">
   <endpoint
      name="StockCoffeeServer"
      displayName="Coffee Stock  Server"
      description="A simple web service"
      interface="stockCoffee.CoffeeIF"
      implementation="stockCoffee.CoffeeImpl"/>
   <endpointMapping
      endpointName="StockCoffeeServer"
      urlPattern="/stockCoffee"/>
</webServices>
```

Файл **web.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
 <display-name>Hello World Application</display-name>
 <description>A web application containing a simple IP endpoint</description>
 <session-config>
  <session-timeout>60</session-timeout>
 </session-config>
</web-app>
```

Процедура развертывания Web-сервиса. Процедура развертывания подготовленного серверного приложения выполняется в два этапа:

- подготовка и компилирование программ (создание директории **\build**, архивирование программ в формате **war**) выполняются операцией **ant build**;
- собственно развертывание сервиса (генерация WSDL-файла, подключение к Web-серверу, описание сервиса в HTML-формате) выполняется операцией **ant deploy**. Удаление декларированного в Tomcat сервиса выполняется операцией **ant undeploy**. Все операции **ant** выполняются в текущей директории сервиса (для обеспечения доступа к конфигурационым файлам).

Выполнение операции **ant build** эквивалентно выполнению последовательности шагов:

- **ant compile-server** – компиляция серверных Java-программ;
- **ant setup-web-inf** – копирование Java-классов и XML-файлов в директорию **`build/WEB-INF;`**
- **ant package** – создание архивного файла в WAR-формате и запись его в директорию **`dist/hello-portable.war`**;
- **ant process-war** – генерация **tie**-программы и интерфейсного WSDL-файла.

После успешного завершения процедуры развертывания сервиса, формализованное описание его в табличном виде должно быть доступно через Web-сервер по адресу: **http://js.jinr.ru:8080/stockCoffee-server/stockCoffee**

Port Name	Status	Information	
StockCoffeeServer	ACTIVE	Address:	http://js.jinr.ru:8080/stockCoffee-server/stockCoffee
		WSDL:	http://js.jinr.ru:8080/stockCoffee-server/stockCoffee?WSDL
		Port QName:	{http://com.test/wsdl/StockCoffeeServer} CoffeeIFPort
		Remote interface:	stockCoffee.CoffeeIF
		Implementation class:	stockCoffee.CoffeeImpl
		Model:	http://js.jinr.ru:8080/stockCoffee-server/stockCoffee?model

Рис. 7. Таблица описания Web-сервиса.

Из приведенной таблицы видно, что WSDL-файл доступен по адресу **http://js.jinr.ru:8080/stockCoffee-server/stockCoffee?WSDL**

Подготовка клиентского приложения

Выбор директории. Для подготовки клиентской программы и ее окружения создадим директорию **d:\jwsdp1.0\myExamples\rpc\Client** и в ней поддиректорию **\stock** (по названию пакета **package stock**), в которой будут подготовлены все необходимые файлы для развертывания клиентского приложения: конфигурационные файлы **config.xml**, **build.xml**, **build.properties** и программа **Client.java.**

Файл **config.xml** используется для указания расположения (локально или в сети) WSDL-файла. Этот файл используется утилитой **wscompile** для генерации Java-интерфейса – описания методов объекта Web-сервиса.

```
<?xml version="1.0" encoding="UTF-8"?>
 <configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
   <wsdl location="http://js.jinr.ru:8080/stockCoffee-server/stockCoffee?WSDL"
    packageName="stock"/>
 </configuration>
```

В файле **build.xml** сконфигурированы операции для оболочки **ant.** Также как и для серверного приложения описания части операций вынесены в отдельный файл **targets.xml** для совместного использования с другими задачами. Текст этого файла приводится в Приложении Б.

```
<!DOCTYPE project [
 <!ENTITY commonTargets SYSTEM "../common/targets.xml">
]>
<project name="JAX-RPC Pattern" default="build" basedir=".">
 <property file="${user.home}/build.properties"/>
 <property file="../common/build.properties"/>
 <property file="build.properties"/>
 &commonTargets;
 <target name="build" depends="build-service"
   description="Executes the targets needed to build the service.">
 </target>
</project>
```

Файл **build.properties** используется для опередения общих переменных:

```
example=stock
client-class=stock.Client
client-jar=${example}-client.jar
```

Таких файлов может быть несколько, все они перечисляются в **build.xml** в переменных **"property file"**.

Программа клиента. Для написания программы клиента необходимо проанализировать интерфейс методов сервисного объекта. Это можно сделать несколькими способами:

- изучить WSDL-описание сервиса;

- сгенерировать Java-интерфейс сервисного объекта, используя в явном виде утилиту wscompile, например **wscompile -import -keep config.xml**. Файл-параметр **config.xml** содержит адрес WSDL-интерфейса (см. выше).

Пример программы клиента **Client.java**:

```
package stock;
import javax.xml.rpc.Stub;
public class Client {
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            CoffeeIF objRef = (CoffeeIF)stub;
            objRef.insertCoffee("Mocca", (float)1.12);
            objRef.insertCoffee("Arabica", (float)2.24);

            String[] nameList =   objRef.getNameList();
            float[] priceList = objRef.getPriceList();

            for (int i = 0; i < priceList.length; i++) {
                    System.out.println(nameList[i]);
                    System.out.println(priceList[i]);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    private static Stub createProxy() {
        return (Stub)(new StockCoffeeServer_Impl().getCoffeeIFPort());
    }
}
```

Как и в предыдущих случаях, для развертывания клиентского приложения используется оболочка-утилита **ant**, управляемая конфигурационными файлами **build.xml** и **targets.xml.**

По команде **ant build-static** выполняются анализ и обработка WSDL-файла, генерация Java-интерфейса сервисного объекта (через утилиту wscompile) и программ-стабов, построение иерархической структуры директорий, компиляция и архивирование программ). Выполнение этой команды эквивалентно выполнению последовательности процедур:

- **ant generate-stubs** – генерация стабов и интерфейсных программ; Все действия сводятся к вызову утилиты **wscompile** с указанием адреса WSDL-файла: **wscompile -gen:client -d build/client -classpath build/shared config.xml;**
- **ant compile-client** – компиляция программы клиента;
- **ant jar-client** – создание архивного файла из откомпилированных Java-классов в JAR-формате и запись его в директорию **/dist**.

Команда **ant run** запускает на выполнение подготовленное клиентское приложение, видимой для пользователя вершиной которого является программа **Client.java**.

Кодирование программы-клиента

Первоочередной задачей клиентских программ во всех объектных распределенных системах, включая рассматриваемые Web-службы, является

создание *объектной ссылки* на серверный объект; в дальнейшем все обращения к методам этого объекта выполняются по правилам объектно-ориентированного программирования, т.е. – с использованием объектных ссылок. В JWSDP предлагаются три способа формирования объекных ссылок:

- Метод **статических стабов** (static stub). Для использования этого метода к моменту кодирования должны быть определены (сгенерированы) все необходимые атрибуты и программы-посредники (стабы) для создания объектной ссылки. Программисту необходимо изучение их текстов для выделения названий программ. В приведенном демонстрационном примере для получения объекта **stub** выполняется обращение к методу **getCoffeeIFPort()** объекта **StockCoffeeServer_Impl**, названия которых включают фрагменты программных компонент используемого сервиса.
- Метод **динамических посредников** (dynamic proxy). В этом методе часть необходимых данных для создания объектной ссылки получают из WSDL динамически после запуска программы: Некотрые данные, в том числе и адрес самого WSDL-файла, должны быть закодированы в программе предварительно, т.е. *статически*. Метод более универсален, но сложнее в кодировании программы клиента. Вариант Java-программы для демонстрационного теста с использованием метода *динамического прокси* приведен в Приложении В.
- Метод **динамически вызываемых интерфейсов** (DII, dynamic invocation interface). Метод абсолютно универсален, но для него данные должны храниться в регистрационных службах (агенствах) специальных серверов.

Как выше уже указывалось, в предлагаемом примере применен метод статических стабов, объектная ссылка **objRef** создается обращением к функции createProxy(), или в более простом виде операторами:

```
Stub stub = (Stub)(new StockCoffeeServer_Impl().getCoffeeIFPort());
CoffeeIF objRef = (CoffeeIF)stub;
```

В остальном, синтаксическое кодирование клиентской программы (формальное обращение к методам сервисного объекта) не вызывает особых трудностей.

Некоторые особенности JWSDP 1.0_01 при разработке Web-сервисов:

- В конфигурационном файле **targets.xml** для компиляции Java-программ (как серверных, так и клиентских) в разделах <target name="compile-server"> и <target name="compile-client"> при задании имен включаемых для компиляции Java-программ используются *метасимволы*, например: includes="*Client.java" или includes="*.java". Это надо иметь в виду при наличии нескольких файлов с Java-программами или личных библиотек программ.
- При передачах *объектов* в качестве параметров или значений результатов методов сервернрого объекта, надо выполнять ряд условий:
    1) объект должен содержать *стандартные методы* **get** и **set** для доступа к своим данным, например **setName()** или **getName();**
    2) должен присутствовать в явном виде *пустой конструктор* объекта с признаком доступа **public;**
    3) объект должен обладать свойством *сериализации*, т.е. наследовать интерфейс **Serializable;**

4) при *передаче* объекта передается его объектная ссылка, которая используется для доступа к его данным и методам;
5) передача структурированных данных (массивов и объектов) в версии JWSDP 1.0_01 нормально работает только в режиме *статических стабов*;
6) пример такого объекта приведен выше в разделе 6.1.

- При желании пользователя подключать к своим приложениям уже имеющиеся библиотеки откомпилированных Java-программ (классов) можно использовать библиотеки **clib** и **elib**, описанные в конфигурационном файле **build.properties**. Пример: в серверной программе импользуются обращения к базам данных; драйверы **JDBC** (Java-поддержка работы с базами данных) поставляются в упакованном формате, например **classes111.zip** для СУБД Oracle. Для правильного их использования в серверном приложении надо:
  1) переформатировать формат упаковки из ZIP-формата в JAR-формат;
  2) записать результат (например, файл **classes111.jar**) в директорию **/common/lib**. Эта директория описана в библиотеке **clib**.

Фрагмент файла **build.properties:**

clib=${jwsdp.home}/**common/lib**
elib=${jwsdp.home}/common/endorsed

8. JAXR (Java API for XML Registries) - приложение пакета JWSDP для регистрации Web-служб

В пакете JWSDP реализована поддержка регистрации Web-служб в соответствии с спецификациями издания 2 UDDI (Universal Description, Discovery and Integration). Это приложение имеет вспомогательный характер и в основном предназначено для тестирования разрабатывемых Web-служб. Для реального использования таких разработок существуют UDDI-сервера общего пользования (для тестрования и промышленного применения).

Обеспечение доступа программам клиента к регистрационным системам выполняет JAXR-провайдер, состоящий из двух Java-пакетов:

- **javax.xml.registry**, содержащий интерфейсы и классы, определяющие интерфейсы доступа к регистрации.
- **javax.xml.registry.infomodel**, содержащий интерфейсы, определяющие информационную модель JAXR. Эти интерфейсы задают типы регистрационных объектов и их взаимосвязи. Основной интерфейс этого пакета - RegistryObject-интерфейс. Наследующие его интерфейсы (подинтерфейсы) - Organization, Service, and ServiceBinding.

Главными интерфейсами пакета javax.xml.registry являются:

- **Connection.** Connection-интерфейс обеспечивает создание сессии клиента для доступа к регистрационному провайдеру.
- **RegistryService.** Этот интерфейс высшего уровня обеспечивает доступ клиентской программы к другим специализированным интерфейсам доступа к регистрационному провайдеру.

Первичными интерфейсами этого же пакета javax.xml.registry являются:

- **BusinessQueryManager,** обеспечивающий запросы пользователя для поиска регистрационных данных в соответствии с интерфейсом информационной модели javax.xml.registry.infomodel.
- **BusinessLifeCycleManager,** обеспечивающий клиенту модификацию регистрационных данных.

Примечания:

1. Знание перечисленных интерфейсов необходимо для программирования клиентских программ доступа к регистрационным данным.
2. При обнаружении сбоев или ошибок в выполнении запрошенных операций генерируется исключение **JAXRException.**

С точки зрения программиста-разработчика, применение JAXR заключается в использовании двух приложений:

- Сервер регистрации (Registry Server) – приложение, работающее в виде Java-сервлетов под управлением Web-сервера Tomcat, выполняет функции депозитария сервисов в UDDI-формате. Для хранения данных используется СУБД **Xindice**, которая является частью Apache XML-проекта;
- библиотек программ (в виде Java-классов) для кодирования доступа к регистрационным UDDI-серверам. В частности, программы расчитаны на работу с серверами IBM (http://uddi.ibm.com/testregistry/inquiryapi), Microsoft (http://uddi.microsoft.com/inquire) и JWSDP-сервером регистрации (http://localhost:8080/registry- server/RegistryServerServlet).

8.1. Сервер регистрации

Для работы сервера регистрации должны быть запущены Web-сервер **Tomcat** и программа управления базой данных **Xindice**. Запуск выполняется из директории **/bin** командными файлами:

- startup.bat (для Tomcat), xindice-start.bat (для Xindice) – для MS Windows;
- startup.sh (для Tomcat), xindice-start.sh (для Xindice) – для UNIX.

Остановка приложений выполняется:

- shutdown.bat (для Tomcat), xindice-stop.bat (для Xindice) – для MS Windows;
- shutdown.sh (для Tomcat), xindice-stop.sh (для Xindice) – для UNIX.

Конфигурационная информация (тип сервера, тип запроса, данные пользователя и др.) задается серверу стандартным образом для Java-программ - через объект Properties при подключении к серверу. Например:

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL", queryUrl);
ConnectionFactory factory = ConnectionFactory.newInstance();
factory.setProperties(props);
```

8.2. Программирование клиент-приложения для доступа к серверу регистрации

Кодирование программы для доступа к регистрационному серверу будет рассмотрено на конкретном примере. Для выполнения всего задания надо выполнить определенную последовательность действий.

Исходные условия:

- все операции выполняются локально;
- конфигурационные данные находятся в файле JAXRExamples.properties. Данные из файла считываются средствами объекта ResourceBundle, из них формируется объект Properties и в этом виде они передаются регистрационному серверу.

Фрагмент конфигурационного файла **JAXRExamples.properties**:

```
## Registry Server:
query.url=http://localhost:8080/registry-server/RegistryServerServlet
publish.url=http://localhost:8080/registry-server/RegistryServerServlet
## testuser/testuser are defaults for Registry Server
registry.username=testuser
registry.password=testuser
http.proxyHost=
http.proxyPort=8080
https.proxyHost=
https.proxyPort=8080
```

Установка соединения

1. Чтение конфигурационного файла:

```
ResourceBundle registryBundle = ResourceBundle.getBundle("JAXRExamples");
String queryURL = registryBundle.getString("query.url");
String httpProxyHost = registryBundle.getString("http.proxyHost");
String httpProxyPort = registryBundle.getString("http.proxyPort");
```

2. Установка режима поиска сервиса:

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL", queryUrl);
props.setProperty("com.sun.xml.registry.http.proxyHost", httpProxyHost);
props.setProperty("com.sun.xml.registry.http.proxyPort", httpProxyPort);
```

3. Установка соединения и передача параметров серверу:

```
try {
    // Create the connection, passing it the configuration properties
    ConnectionFactory factory = ConnectionFactory.newInstance();
    factory.setProperties(props);
    connection = factory.createConnection();
    System.out.println("Created connection to registry");
} catch (Exception e) { }
```

4. Установка типа запроса (поиск сервиса):

```
RegistryService rs = connection.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();
```

5. Задание поиска списка организаций по ключевому слову в переменной **qString**:

```
// Define find qualifiers and name patterns
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
Collection namePatterns = new ArrayList();
namePatterns.add("%" + qString + "%");
```

6. Обращение к серверу:

```
// Find using the name
BulkResponse response =  bqm.findOrganizations(findQualifiers,
        namePatterns, null, null, null, null);
```

7. Формирование объекта Collection из полученного ответа для итерационной обработки списка:

```
Collection orgs = response.getCollection();
Iterator orgIter = orgs.iterator();
Organization org = (Organization) orgIter.next();
Collection services = org.getServices();
User pc = org.getPrimaryContact();
PersonName pcName = pc.getPersonName();
Collection phNums =    pc.getTelephoneNumbers(null);
Collection eAddrs = pc.getEmailAddresses();
```

Примечания.

1. В рассмотренном примере приводится лишь один формат запроса регистрационных данных: поиск организаций (findOrganizations). Существуют еще два формата запросов: поиск сервиса (findServices) и поиск типов доступа к сервису (findServiceBindings). Формат поиска задается в примере с использованием доступа к серверу посредством объекта **BusinessQueryManager**.
2. Для модификации регистрационных данных (заведение, авторизация, редактирование) применяется обращение к серверу через объект **BusinessLifeCycleManager.**
3. Полный текст примера приведен в Приложении Г.

8.3. Развертывание клиентского приложения

Компиляция программы клиента выполняетя традиционным для Web-служб способом: командой **ant build** – командной оболочкой **ant** под управлением конфигурационного файла **build.xml**. Этот файл используется также и при запуске клиентской программы. Пример файла приведен в Приложении Г.

Для запуска программы клиента в текущей директории должен быть подготовлен файл **JAXRExamples.properties** с параметрами конфигурации регистрационного сервера. В этом же файле задан адрес **(query.url=http://localhost:8080/registry-server/RegistryServerServlet)** обращения к серверу регистрации, который выполнен в виде Java-сервлетов. Как выше уже указывалось, для работы сервера регистрации должны быть запущены программы Web-сервера **Tomcat** и СУБД **Xindice.** Запуск подготовленной программы клиента выполняется оболочкой **ant** с параметом **run-query;** программе передается ключевое слово **"coff"** для поиска сервиса:

ant run-query -Dquery-string=coff

Литература

1. http://java.sun.com/.
2. http://java.sun.com/webservices/.
3. http://www.w3c.org/2002/ws/.

4. Галактионов В.В. Web Services – новая сервис-ориентированная технология для распределенных объектных вычислительных систем. Основные концепции, протоколы и спецификации.Сообщение ОИЯИ, , Дубна, 2003.
5. http://www.w3c.org/.
6. The Java Web Services Tutorial. http://java.sun.com/webservices/tutorial.html.
7. Галактионов В.В. Расширяемый язык разметки XML (eXtensible Mark-up Language): промышленный стандарт, определяющий архитектуру программных средств Интернет следующего поколения. Сообщение ОИЯИ, Р10-2000-44, Дубна, 2000.
8. Document Object Model (DOM). http://www.w3.org/DOM/.
9. The Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/.
10. The Apache Jacarta Project. http://jacarta.apache.org/.
11. Apache Software Foundation. http://www.apache.org.

Приложение А. Тексты программ и конфигурационных файлов для JAXM.

Файл **index.html:**

```
<html>
<body>
This is a simple example of a roundtrip JAXM message exchange.
<p> Click <a href="http://js.jinr.ru:8080/jaxm-first/mySender">here</a> to send the message
</body>
</html>
```

Файл **SendingServlet.java:**

```
package simple.sender;
import java.net.*;
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
import javax.xml.soap.*;
import javax.activation.*;
import javax.naming.*;

public class SendingServlet extends HttpServlet {
  String to = null;
  String data = null;
  ServletContext servletContext;
  // Connection to send messages.
  private SOAPConnection con;

  public void init(ServletConfig servletConfig) throws ServletException {
    super.init( servletConfig );
    servletContext = servletConfig.getServletContext();
    try {
        SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
      con = scf.createConnection();
    } catch(Exception e) {
      logger.error("Unable to open a SOAPConnection", e);
    }
    InputStream in
    = servletContext.getResourceAsStream("/WEB-INF/address.properties");
    if (in != null) {
      Properties props = new Properties();
      try {
        props.load(in);
```

```
        to = props.getProperty("to");
        data = props.getProperty("data");
      } catch (IOException ex) {  }
    }
}

public void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException {
        String retval ="<html> <H4>";
   try {
      // Create a message factory.
      MessageFactory mf = MessageFactory.newInstance();
      // Create a message from the message factory.
      SOAPMessage msg = mf.createMessage();
     // Message creation takes care of creating the SOAPPart - a
      // required part of the message as per the SOAP 1.1 specification.
      SOAPPart sp = msg.getSOAPPart();
    // Retrieve the envelope from the soap part to start building
      // the soap message.
      SOAPEnvelope envelope = sp.getEnvelope();

      // Create a soap header from the envelope.
      SOAPHeader hdr = envelope.getHeader();

      // Create a soap body from the envelope.
      SOAPBody bdy = envelope.getBody();

      // Add a soap body element to the soap body
      SOAPBodyElement gltp
      = bdy.addBodyElement(envelope.createName("GetLastTradePrice",
      "ztrade",
      "http://wombat.ztrade.com"));

      gltp.addChildElement(envelope.createName("symbol",
      "ztrade",
      "http://wombat.ztrade.com")).addTextNode("SUNW");

      StringBuffer urlSB=new StringBuffer();
      urlSB.append(req.getScheme()).append("://").append(req.getServerName());
      urlSB.append( ":" ).append( req.getServerPort() ).append( req.getContextPath() );
      String reqBase=urlSB.toString();


      if(data==null) {
        data=reqBase + "/index.html";
      }
    data = "http://js.jinr.ru:8080/index.html";

      // Want to set an attachment from the following url.
     //Get context
      URL url = new URL(data);

      AttachmentPart ap = msg.createAttachmentPart(new DataHandler(url));

      ap.setContentType("text/html");

      // Add the attachment part to the message.
      msg.addAttachmentPart(ap);

      // Create an endpoint for the recipient of the message.
```

```
            if(to==null) {
               to=reqBase + "/myReceiver";
            }
            URL urlEndpoint = new URL(to);

            System.err.println("Sending message to URL: "+urlEndpoint);
            System.err.println("Sent message is logged in \"sent.msg\"");

            FileOutputStream sentFile = new FileOutputStream("sent.msg");
            msg.writeTo(sentFile);
            sentFile.close();

            retval += " Sent message (check \"sent.msg\") " + " / and ";

            // Send the message to the provider using the connection.
            SOAPMessage reply = con.call(msg, urlEndpoint);

            retval += " <br> After sending";

            if (reply != null) {
               FileOutputStream replyFile = new FileOutputStream("reply.msg");
               reply.writeTo(replyFile);
               replyFile.close();
               System.err.println("Reply logged in \"reply.msg\"");
               retval += " received reply (check \"reply.msg\"). </H4> </html>";

            } else {
               System.err.println("No reply");
               retval += " no reply was received. </H4> </html>";
            }

         } catch(Throwable e) {
            e.printStackTrace();
            retval += " There was an error " +
            "in constructing or sending message. </H4> </html>";
         }
         try {
            OutputStream os = resp.getOutputStream();
            os.write(retval.getBytes());
            os.flush();
            os.close();
         } catch (IOException e) {
            e.printStackTrace();
            logger.error( "Error in outputting servlet response "
            + e.getMessage());
         }
      }
   }
```

Файл ReceivingServlet.java:

```
package simple.receiver;
import javax.xml.soap.*;
import javax.xml.messaging.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.transform.*;
import javax.naming.*;
import org.apache.commons.logging.*;

public class ReceivingServlet
```

```
  extends JAXMServlet    implements ReqRespListener {
  static MessageFactory fac = null;

  static {
    try {
      fac = MessageFactory.newInstance();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  };
  public void init(ServletConfig servletConfig) throws ServletException {
    super.init(servletConfig);
    // Not much there to do here.
  }

  // This is the application code for handling the message.. Once the
  // message is received the application can retrieve the soap part, the
  // attachment part if there are any, or any other information from the
  // message.

  public SOAPMessage onMessage(SOAPMessage message) {
    System.out.println("On message called in receiving servlet");
    try {
      System.out.println("Here's the message: ");
      message.writeTo(System.out);

      SOAPMessage msg = fac.createMessage();
       SOAPEnvelope env = msg.getSOAPPart().getEnvelope();

      env.getBody()
        .addChildElement(env.createName("Response"))
        .addTextNode("This is a response");

      return msg;
    } catch(Exception e) {
        return null;
    }
  }
}
```

Файл **build.xml:**

```
<project name="JAXM Test" default="build" basedir=".">
 <property file="${user.home}/build.properties"/>
 <property file="build.properties"/>
 <taskdef name="reload" classname="org.apache.catalina.ant.ReloadTask" />
 <taskdef name="install" classname="org.apache.catalina.ant.InstallTask" />
 <taskdef name="remove" classname="org.apache.catalina.ant.RemoveTask" />
 <taskdef name="list" classname="org.apache.catalina.ant.ListTask" />
 <taskdef name="deploy" classname="org.apache.catalina.ant.DeployTask" />
 <taskdef name="undeploy" classname="org.apache.catalina.ant.UndeployTask" />

 <path id="classpath">
  <fileset dir="${jwsdp.home}/common/lib">
   <include name="*.jar"/>
  </fileset>
  <fileset dir="${jwsdp.home}/common/endorsed">
   <include name="*.jar"/>
  </fileset>
 </path>
```

```
<target name="prepare"
  description="Creates the build and dist directories" >
  <echo message="Creating the required directories...." />
  <mkdir dir="${build}/client" />
  <mkdir dir="${build}/server" />
  <mkdir dir="dist" />
</target>
<target name="compile-server" depends="prepare"
   description="Compiles the server-side source code">
   <echo message="Compiling the server-side source code...."/>
   <javac
     srcdir="${src}"
     destdir="${build}/server"
   >
   <classpath refid="classpath" />
   </javac>
</target>
<target name="compile-client"
   description="Compiles the client source code" >
   <echo message="Compiling the client source code...."/>
   <javac
     srcdir="${src}"
     destdir="${build}/client"
   >
   <classpath refid="classpath" />
   <classpath path="${common}" />
   </javac>
</target>
<target name="jar-client" depends="compile-client"
   description="Builds the JAR file that contains the JAX-M client routines">
   <echo message="Building the ${client-jar} file...."/>
   <delete file="dist/${client-jar}" />
   <jar jarfile="dist/${client-jar}" >
    <fileset dir="${build}/client" >
       <exclude name="**/*Test*" />
    </fileset>
   </jar>
</target>
<target name="setup-web-inf"
  description="Copies files to build/WEB-INF">
  <echo message="Setting up ${build}/WEB-INF...."/>
  <delete dir="${build}/WEB-INF" />
  <mkdir dir="${build}/WEB-INF/classes" />
  <copy todir="${build}/WEB-INF/classes">
       <fileset dir="${build}/server" />
  </copy>
  <copy file="web/web.xml" todir="${build}/WEB-INF" />
</target>
<target name="install" depends="build"
  description="Installs a Web application">
  <echo message="Installing the application...."/>
  <install
    url="${url}"
    username="${username}"
    password="${password}"
    path="/${context-path}"
    war="file:${build-path}"
  />
</target>
```

```
<target name="package" depends="build"
   description="Packages the WAR file">
   <echo message="Packaging the WAR...."/>
   <delete file="dist/${war-file}" />
   <jar jarfile="dist/${war-file}" >
    <fileset dir="${build-path}" includes="WEB-INF/**" />
   </jar>
</target>
<target name="deploy" depends="build, package"
  description="Deploys a Web application">
  <echo message="Deploying the application...."/>
  <deploy
    url="${url}"
    username="${username}"
    password="${password}"
    path="/${context-path}"
    war="file:${example-path}/dist/${war-file}"
  />
</target>
<target name="remove"
  description="Removes a Web application">
  <echo message="Removing the application...."/>
  <remove
    url="${url}"
    username="${username}"
    password="${password}"
    path="/${context-path}"
  />
</target>
<target name="build" depends="clean, compile-server, jar-client, setup-web-inf"
  description="Executes the targets needed to build this example.">
</target>
<target name="clean"
  description="Removes the build and dist directories">
  <delete dir="${build}" />
  <delete dir="dist" />
</target>
</project>
```

Файл web.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
    <servlet-name>
      sendingservlet
    </servlet-name>
    <servlet-class>
      simple.sender.SendingServlet
    </servlet-class>
      <load-on-startup>
        2
      </load-on-startup>
  </servlet>
  <servlet>
    <servlet-name>   receivingservlet    </servlet-name>
    <servlet-class>      simple.receiver.ReceivingServlet
    </servlet-class>
```

```
        <load-on-startup> 3
        </load-on-startup>
    </servlet>
    <servlet-mapping>
      <servlet-name>   sendingservlet     </servlet-name>
      <url-pattern>     /mySender        </url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>      receivingservlet     </servlet-name>
      <url-pattern>       /myReceiver      </url-pattern>
    </servlet-mapping>
</web-app>
```

Файл **build.properties:**

```
build=build
src=source
url=http://localhost:8080/manager
docs-path=${jwsdp.home}/myExamples
common=../common/build

context-path=jaxm-first
client-jar=jaxm-client.jar
war-file=${context-path}.war
example-path=${docs-path}/jaxm/simple
build-path=${example-path}/${build}
```

Приложение Б. Конфигурационный файл **targets.xml** для примера применения RPC:

```
<!-- targets.xml: Referenced by the build.xml files, this file
    contains targets common to all of the jaxrpc examples.  -->
  <taskdef name="deploy" classname="org.apache.catalina.ant.DeployTask" />
 <taskdef name="undeploy" classname="org.apache.catalina.ant.UndeployTask" />
 <taskdef name="list" classname="org.apache.catalina.ant.ListTask" />
 <taskdef name="start" classname="org.apache.catalina.ant.StartTask" />
 <taskdef name="stop" classname="org.apache.catalina.ant.StopTask" />
<target name="deploy"
   description="Deploys a Web application">
    <deploy url="${url}" username="${username}" password="${password}"
     path="/${context-path}" war="file:${war-path}"
    />
 </target>
 <target name="undeploy"
   description="Undeploys a Web application">
    <undeploy url="${url}" username="${username}" password="${password}"
     path="/${context-path}"
    />
 </target>
 <target name="redeploy"
    description="Undeploys and deploys a Web aplication">
    <antcall target="undeploy" />
    <antcall target="deploy" />
 </target>
 <target name="list"
   description="Lists Web applications">
   <echo message="Listing the applications...."/>
   <list
     url="${url}"
     username="${username}"
     password="${password}"
```

```
    />
  </target>
 <target name="start"
    description="Starts a Web application">
    <echo message="Starting the application...."/>
    <start
      url="${url}"
      username="${username}"
      password="${password}"
      path="/${context-path}"
    />
  </target>
 <target name="stop"
    description="Stops a Web application">
    <echo message="Stopping the application...."/>
    <stop
      url="${url}"
      username="${username}"
      password="${password}"
      path="/${context-path}"
    />
  </target>
 <target name="set-ws-scripts"
    description="Sets the value of the wscompile and wsdeploy properties for this build file" >
    <condition property="script-suffix" value="bat">
      <os family="windows"/>
    </condition>
    <condition property="script-suffix" value="sh">
      <not>
        <os family="windows"/>
      </not>
    </condition>
    <property name="wscompile" value="${jwsdp.home}/bin/wscompile.${script-suffix}"/>
    <property name="wsdeploy" value="${jwsdp.home}/bin/wsdeploy.${script-suffix}"/>
  </target>
 <target name="prepare"
    description="Creates the build and dist directories" >
    <echo message="Creating the required directories...." />
    <mkdir dir="${build}/client/${example}" />
    <mkdir dir="${build}/server/${example}" />
    <mkdir dir="${build}/shared/${example}" />
    <mkdir dir="${build}/wsdeploy-generated" />
    <mkdir dir="dist" />
    <mkdir dir="${build}/WEB-INF/classes/${example}" />
  </target>
 <target name="compile-server" depends="prepare"
    description="Compiles the server-side source code">
    <echo message="Compiling the server-side source code...."/>
    <javac
      srcdir="."
      destdir="${build}/shared"
      includes="*.java"
      excludes="*Client.java"
    />
  </target>
 <target name="setup-web-inf"
    description="Copies files to build/WEB-INF">
    <echo message="Setting up ${build}/WEB-INF...."/>
    <delete dir="${build}/WEB-INF" />
    <copy todir="${build}/WEB-INF/classes/${example}">
```

```
      <fileset dir="${build}/shared/${example}" />
      <fileset dir="${build}/server/${example}" />
    </copy>
    <copy file="web.xml" todir="${build}/WEB-INF" />
    <copy file="jaxrpc-ri.xml" todir="${build}/WEB-INF" />
  </target>
<target name="package"
    description="Packages the WAR file">
    <echo message="Packaging the WAR...."/>
    <delete file="dist/${portable-war}" />
    <jar jarfile="dist/${portable-war}" >
     <fileset dir="${build}" includes="WEB-INF/**" />
    </jar>
  </target>
  <target name="process-war" depends="set-ws-scripts"
    description="Runs wsdeploy to generate the ties and create a deployable WAR file">
    <echo message="Running wsdeploy...."/>
    <delete file="dist/${deployable-war}" />
   <exec executable="${wsdeploy}">
    <arg line="-tmpdir"/>
    <arg line="${build}/wsdeploy-generated"/>
    <arg line="-o"/>
    <arg line="dist/${deployable-war}"/>
    <arg line="dist/${portable-war}"/>
    <arg line="-verbose"/>
   </exec>
  </target>
  <target name="generate-stubs" depends="set-ws-scripts,prepare"
    description="Runs wscompile to generate the client stub classes">
    <echo message="Running wscompile...."/>
   <exec executable="${wscompile}">
     <arg line="-gen:client"/>
     <arg line="-d ${build}/client"/>
     <arg line="-classpath ${build}/shared"/>
     <arg line="config.xml"/>
   </exec>
  </target>
  <target name="compile-client" depends="prepare"
    description="Compiles the client-side source code" >
    <echo message="Compiling the client source code...."/>
    <javac
      srcdir="."
      destdir="${build}/client"
      classpath="${jwsdp-jars}:build/shared:build/client"
      includes="*Client.java"
    />
  </target>
  <target name="jar-client"
    description="Builds the JAR file that contains the client">
    <echo message="Building the client JAR  file...."/>
    <delete file="dist/${client-jar}" />
    <jar jarfile="dist/${client-jar}" >
     <fileset dir="${build}/client" />
     <fileset dir="${build}/shared" >
        <exclude name="**/*Impl*" />
      </fileset>
    </jar>
  </target>
  <target name="build-static" depends="clean-client,generate-stubs,
   compile-client,jar-client"
```

```
  description="Executes the targets needed to build a stub client.">
 </target>
 <target name="build-dynamic" depends="clean-client,compile-client,jar-client"
  description="Executes the targets needed to build a dynamic client.">
 </target>
 <target name="build-service" depends="clean,clean-wars,compile-server,
  setup-web-inf,package,process-war"
  description="Executes the targets needed to build the service.">
 </target>
 <target name="run"
  description="Runs the example client">
  <echo message="Running the ${client-class} program...." />
   <java
      fork="on"
      classpath="dist/${client-jar}"
      classname="${client-class}" >
   </java>
 </target>
 <target name="clean"
  description="Removes the build directory">
  <delete dir="${build}" />
 </target>
 <target name="clean-client"
  description="Removes the client classes and JAR">
  <delete>
   <fileset dir="${build}/client/${example}" includes="*.class"/>
  </delete>
  <delete file="dist/${client-jar}" />
 </target>
 <target name="clean-wars"
  description="Deletes the WAR files in the dist directory">
  <delete file="dist/${portable-war}" />
  <delete file="dist/${deployable-war}" />
 </target>
 <target name="debug" depends="set-ws-scripts"
  description="Displays values of some of the properties of this build file">
  <echo message="**Debug**" />
  <echo message="  ${user.home}" />
  <echo message="  ${username}" />
  <echo message="  ${password}" />
  <echo message="  ${url}" />
  <echo message="  ${wsdeploy}" />
  <echo message="  ${context-path}" />
  <echo message="  ${war-path}" />
 </target>
```

Приложение В. Пример клиентской Java-программы, использущей метод *динамического прокси.*

Файл **Client.java:**

```
package stock;
import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;

public class Client {
   public static void main(String[] args) {
      try {
         String UrlString ="http://js.jinr.ru:8080/stockCoffee-server/stockCoffee?WSDL";
```

```
        String nameSpaceUri = "http://com.test/wsdl/StockCoffeeServer";
        String serviceName = "StockCoffeeServer";
        String portName = "CoffeeIFPort";

        URL helloWsdlUrl = new URL(UrlString);

        ServiceFactory serviceFactory = ServiceFactory.newInstance();

        Service coffeeService = serviceFactory.createService(helloWsdlUrl,
                                new QName(nameSpaceUri, serviceName));

       CoffeeIF objRef = (CoffeeIF) coffeeService.getPort(
          new QName(nameSpaceUri, portName),    CoffeeIF.class);

        objRef.insertCoffee("Mocca", (float)1.12);
        objRef.insertCoffee("Arabica", (float)2.24);

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

Примечание: В директории, где разверывается программа клиента, должен находиться файл с описанием интерфейса сервиса **CoffeeIF.java:**

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

public interface CoffeeIF extends Remote {
 public boolean   insertCoffee(String coffeeName, float price)  throws RemoteException;
 public String    orderCoffee(String coffeeName, float quantity) throws RemoteException;
 public float[]  getPriceList() throws RemoteException;
 public String[] getNameList()  throws RemoteException;
 public String readIP(String s) throws RemoteException;
}
```

Развертывание и запуск клиентского приложения:

ant build-dynamic

ant run

Приложение Г. Пример программирования доступа к серверу регистрации в режиме поиска сервиса.

1. Файл **JAXRQuery.java**:

```
import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;
import java.net.*;
import java.util.*;

/**
 * The JAXRQuery class consists of a main method, a
 * makeConnection method, an executeQuery method, and some
 * private helper methods. It searches a registry for
 * information about organizations whose names contain a
 * user-supplied string.
 *
```

```
 * To run this program, use the command
 *
 *    ant run-query -Dquery-string=<value>
 */
public class JAXRQuery {
  Connection connection = null;

  public JAXRQuery() {}

  public static void main(String[] args) {
    ResourceBundle registryBundle =
      ResourceBundle.getBundle("JAXRExamples");

    String queryURL = registryBundle.getString("query.url");
    String publishURL = registryBundle.getString("publish.url");

    if (args.length < 1) {
      System.out.println("Usage: ant " +
        "run-query -Dquery-string=<value>");
      System.exit(1);
    }
    String queryString = new String(args[0]);
    System.out.println("Query string is " + queryString);

    JAXRQuery jq = new JAXRQuery();

    jq.makeConnection(queryURL, publishURL);

    jq.executeQuery(queryString);
  }

  /**
   * Establishes a connection to a registry.
   *
   * @param queryUrl     the URL of the query registry
   * @param publishUrl   the URL of the publish registry
   */
  public void makeConnection(String queryUrl,
    String publishUrl) {

    /*
     * Specify proxy information in case you
     *  are going beyond your firewall.
     */
    ResourceBundle registryBundle =
      ResourceBundle.getBundle("JAXRExamples");
    String httpProxyHost = registryBundle.getString("http.proxyHost");
    String httpProxyPort = registryBundle.getString("http.proxyPort");

    /*
     * Define connection configuration properties.
     * For simple queries, you need the query URL.
     */
    Properties props = new Properties();
    props.setProperty("javax.xml.registry.queryManagerURL",
      queryUrl);
    props.setProperty("com.sun.xml.registry.http.proxyHost",
      httpProxyHost);
    props.setProperty("com.sun.xml.registry.http.proxyPort",
      httpProxyPort);
```

```
    try {
      // Create the connection, passing it the
      // configuration properties
      ConnectionFactory factory =
        ConnectionFactory.newInstance();
      factory.setProperties(props);
      connection = factory.createConnection();
      System.out.println("Created connection to registry");
    } catch (Exception e) {
      e.printStackTrace();
      if (connection != null) {
        try {
          connection.close();
        } catch (JAXRException je) {}
      }
    }
  }

  /**
   * Searches for organizations containing a string and
   * displays data about them.
   *
   * @param qString        the string argument
   */
  public void executeQuery(String qString) {
    RegistryService rs = null;
    BusinessQueryManager bqm = null;

    try {
      // Get registry service and query manager
      rs = connection.getRegistryService();
      bqm = rs.getBusinessQueryManager();
      System.out.println("Got registry service and " +
        "query manager");

      // Define find qualifiers and name patterns
      Collection findQualifiers = new ArrayList();
      findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
      Collection namePatterns = new ArrayList();
      namePatterns.add("%" + qString + "%");

      // Find using the name
      BulkResponse response =
        bqm.findOrganizations(findQualifiers,
          namePatterns, null, null, null, null);
      Collection orgs = response.getCollection();

      // Display information about the organizations found
      Iterator orgIter = orgs.iterator();
      if (!(orgIter.hasNext())) {
        System.out.println("No organizations found");
      } else while (orgIter.hasNext()) {
        Organization org = (Organization) orgIter.next();
        System.out.println("Org name: " + getName(org));
        System.out.println("Org description: " +
          getDescription(org));
        System.out.println("Org key id: " + getKey(org));

        // Display primary contact information
```

```
            User pc = org.getPrimaryContact();
            if (pc != null) {
              PersonName pcName = pc.getPersonName();
              System.out.println(" Contact name: " +
                pcName.getFullName());
              Collection phNums =
                pc.getTelephoneNumbers(null);
              Iterator phIter = phNums.iterator();
              while (phIter.hasNext()) {
                TelephoneNumber num =
                  (TelephoneNumber) phIter.next();
                System.out.println("  Phone number: " +
                  num.getNumber());
              }
              Collection eAddrs = pc.getEmailAddresses();
              Iterator eaIter = eAddrs.iterator();
              while (eaIter.hasNext()) {
                EmailAddress eAd =
                  (EmailAddress) eaIter.next();
                System.out.println("  Email Address: " +
                  eAd.getAddress());
              }
            }
            // Display service and binding information
            Collection services = org.getServices();
            Iterator svcIter = services.iterator();
            while (svcIter.hasNext()) {
              Service svc = (Service) svcIter.next();
              System.out.println(" Service name: " +
                getName(svc));
              System.out.println(" Service description: " +
                getDescription(svc));
              Collection serviceBindings =
                svc.getServiceBindings();
              Iterator sbIter = serviceBindings.iterator();
              while (sbIter.hasNext()) {
                ServiceBinding sb =
                  (ServiceBinding) sbIter.next();
                System.out.println("  Binding " +
                  "Description: " +
                  getDescription(sb));
                System.out.println("  Access URI: " +
                  sb.getAccessURI());
              }
            }
            // Print spacer between organizations
            System.out.println(" --- ");
          }
        } catch (Exception e) {
          e.printStackTrace();
        } finally {
          // At end, close connection to registry
          if (connection != null) {
            try {
              connection.close();
            } catch (JAXRException je) {}
          }
        }
      }
```

```
    /**
     * Returns the name value for a registry object.
     *
     * @param ro   a RegistryObject
     * @return                the String value
     */
    private String getName(RegistryObject ro)
      throws JAXRException {

      try {
        return ro.getName().getValue();
      } catch (NullPointerException npe) {
        return "No Name";
      }
    }

    /**
     * Returns the description value for a registry object.
     *
     * @param ro   a RegistryObject
     * @return                the String value
     */
    private String getDescription(RegistryObject ro)
      throws JAXRException {

      try {
        return ro.getDescription().getValue();
      } catch (NullPointerException npe) {
        return "No Description";
      }
    }

    /**
     * Returns the key id value for a registry object.
     *
     * @param ro   a RegistryObject
     * @return                the String value
     */
    private String getKey(RegistryObject ro)
      throws JAXRException {

      try {
        return ro.getKey().getId();
      } catch (NullPointerException npe) {
        return "No Key";
      }
    }
}
```

2. Конфигурационный файл **build.xml**:

```
<project name="JAXR Tutorial" default="build" basedir=".">
 <target name="init">
   <tstamp/>
 </target>
<property name="build.home" value="build" />
<path id="classpath">
  <fileset dir="${jwsdp.home}/common/lib">
   <include name="*.jar" />
  </fileset>
  <fileset dir="${jwsdp.home}/common/endorsed">
```

```
    <include name="*.jar" />
  </fileset>
</path>
<target name="prepare" depends="init"
 description="Create build directory.">
  <mkdir dir="${build.home}" />
</target>
<target name="build" depends="prepare"
  description="Compile Java files.">
  <copy file="JAXRExamples.properties" todir="${build.home}" />
   <javac srcdir="." destdir="${build.home}">
        <include name="*.java" />
    <classpath refid="classpath"/>
   </javac>
</target>
<target name="run-publish" depends="build"
  description="Run JAXRPublish.">
  <java classname="JAXRPublish" fork="yes">
   <!--<sysproperty key="useSOAP" value="true"/>-->
   <classpath refid="classpath" />
   <classpath path="${build.home}" />
  </java>
</target>
<target name="run-query" depends="build"
  description="Run JAXRQuery. Argument: -Dquery-string=&lt;string&gt;">
  <java classname="JAXRQuery" fork="yes">
   <!--<sysproperty key="org.apache.commons.logging.simplelog.log.com.sun.xml.registry"
       value="debug"/>-->
   <arg line="${query-string}" />
   <classpath refid="classpath" />
   <classpath path="${build.home}" />
  </java>
</target>
<target name="run-delete" depends="build"
  description="Run JAXRDelete. Argument: -Dkey-string=&lt;key-from-run-publish&gt;">
  <java classname="JAXRDelete" fork="yes">
   <arg line="${key-string}" />
   <classpath refid="classpath" />
   <classpath path="${build.home}" />
  </java>
</target>
<target name="run-query-naics" depends="build"
  description="Run JAXRQueryByNAICSClassification.">
  <java classname="JAXRQueryByNAICSClassification" fork="yes">
   <classpath refid="classpath" />
   <classpath path="${build.home}" />
  </java>
</target>
<target name="run-query-wsdl" depends="build"
  description="Run JAXRQueryByWSDLClassification.">
  <java classname="JAXRQueryByWSDLClassification" fork="yes">
   <!--<sysproperty key="org.apache.commons.logging.simplelog.log.com.sun.xml.registry"
       value="debug"/>-->
   <classpath refid="classpath" />
   <classpath path="${build.home}" />
  </java>
</target>
<target name="run-save-scheme" depends="build"
  description="Run JAXRSaveClassificationScheme." >
  <java classname="JAXRSaveClassificationScheme" fork="yes">
```

```
      <classpath refid="classpath" />
      <classpath path="${build.home}" />
    </java>
  </target>
  <target name="run-publish-postal" depends="build"
    description="Run JAXRPublishPostal. Argument: -Duuid-string=&lt;key-from-run-save-scheme&gt;"
  >
    <java classname="JAXRPublishPostal" fork="yes">
      <!--<sysproperty key="org.apache.commons.logging.simplelog.log.com.sun.xml.registry"
          value="debug"/>-->
      <sysproperty key="com.sun.xml.registry.userTaxonomyFilenames"
        value="postalconcepts.xml"/>
      <arg line="${uuid-string}" />
      <classpath refid="classpath" />
      <classpath path="${build.home}" />
    </java>
  </target>
  <target name="run-query-postal" depends="build"
    description="Run JAXRQueryPostal. Arguments: -Dquery-string=&lt;string&gt; -Duuid-
string=&lt;key-from-run-save-scheme&gt;" >
    <java classname="JAXRQueryPostal" fork="yes">
      <!--<sysproperty key="org.apache.commons.logging.simplelog.log.com.sun.xml.registry"
          value="debug"/>-->
      <sysproperty key="com.sun.xml.registry.userTaxonomyFilenames"
        value="postalconcepts.xml"/>
      <arg line="${query-string}" />
      <arg line="${uuid-string}" />
      <classpath refid="classpath" />
      <classpath path="${build.home}" />
    </java>
  </target>
  <target name="run-delete-scheme" depends="build"
    description="Run JAXRDeleteScheme. Argument: -Duuid-string=&lt;uuid-from-run-save-
scheme&gt;">
    <java classname="JAXRDeleteScheme" fork="yes">
      <arg line="${uuid-string}" />
      <classpath refid="classpath" />
      <classpath path="${build.home}" />
    </java>
  </target>
  <target name="run-get-objects" depends="build"
    description="Run JAXRGetMyObjects." >
    <java classname="JAXRGetMyObjects" fork="yes">
      <classpath refid="classpath" />
      <classpath path="${build.home}" />
    </java>
  </target>
 <target name="clean" depends="prepare"
  description="Delete build directory.">
    <delete dir="${build.home}" />
  </target>
</project>
```

3. Файл конфигурации регистрационного сервера **JAXRExamples.properties:**

```
## Uncomment one pair of query and publish URLs.
## IBM:
#query.url=http://uddi.ibm.com/testregistry/inquiryapi
#publish.url=https://uddi.ibm.com/testregistry/protect/publishapi
## Microsoft:
#query.url=http://uddi.microsoft.com/inquire
```

```
#publish.url=https://uddi.microsoft.com/publish
## Registry Server:
query.url=http://localhost:8080/registry-server/RegistryServerServlet
publish.url=http://localhost:8080/registry-server/RegistryServerServlet
## Specify username and password if needed
## testuser/testuser are defaults for Registry Server
registry.username=testuser
registry.password=testuser
## HTTP and HTTPS proxy host and port;
##  ignored by Registry Server
http.proxyHost=
http.proxyPort=8080
https.proxyHost=
https.proxyPort=8080
## Values used by publish examples
org.name=The Coffee Break
org.description=Purveyor of the finest coffees. Established 1895
person.name=Jane Doe
phone.number=(800) 555-1212
email.address=jane.doe@TheCoffeeBreak.com
classification.scheme=ntis-gov:naics
classification.name=Snack and Nonalcoholic Beverage Bars
classification.value=722213
service.name=My Service Name
service.description=My Service Description
svcbinding.description=My Service Binding Description
svcbinding.accessURI=http://TheCoffeeBreak.com:8080/sb/
## Values used by postal address examples
postal.scheme.name=MyPostalAddressScheme
postal.scheme.description=A ClassificationScheme for My PostalAddressMappings
postal.classification.name=postalAddress
# value should be postalAddress, not categorization - IBM registry not
#  accepting this yet
postal.classification.value=categorization
postal.scheme.link=http://unrealcompany.com/PostalScheme.html
postal.scheme.linkdesc=My PostalAddress Scheme
postal.org.name=The Postal Coffee Break
postal.person.name=Jane Postal
postal.email.address=jane.postal@ThePostalCoffeeBreak.com
postal.streetNumber=99
postal.street=Imaginary Ave. Suite 33
postal.city=Imaginary City
postal.state=NY
postal.country=USA
postal.postalCode=00000
postal.type=
```