Лабораторная работа №2

Web-сервисы. Разработка web-сервиса, реализующего документоориентированное взаимодействие

Цель: научиться работать с технологией web-сервисов, используя документоориентированный подход.

В предыдущей лабораторной работе мы создали наш собственный web-сервис, используя модель RPC. В этой лабораторной работе документы XML будут являться основным инструментом взаимодействия между вызывающим объектом и web-сервисом. Этот тип SOAP-разработки называется документоориентированным(*document-style*) подходом.

В случае документоориентированного программирования клиент передает web-сервису документ XML, который обрабатывается на сервере. Здесь также (как и в первой лабораторной работе) на макроуровне клиент передает запрос SOAP и получает ответ SOAP (см. рис.1):

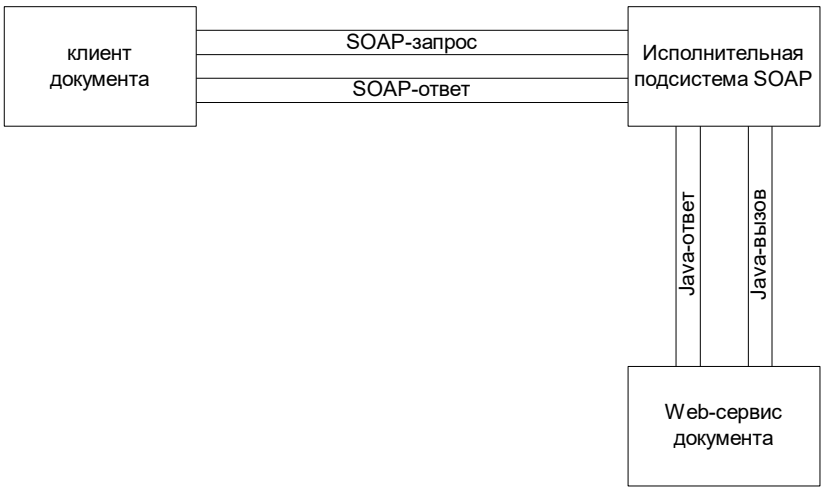

Рис.1. Схема документоориентированного программирования

Чтобы лучше увидеть различия между двумя подходами, рассмотрим их принцип действия подробнее (Рис.2 и 3):

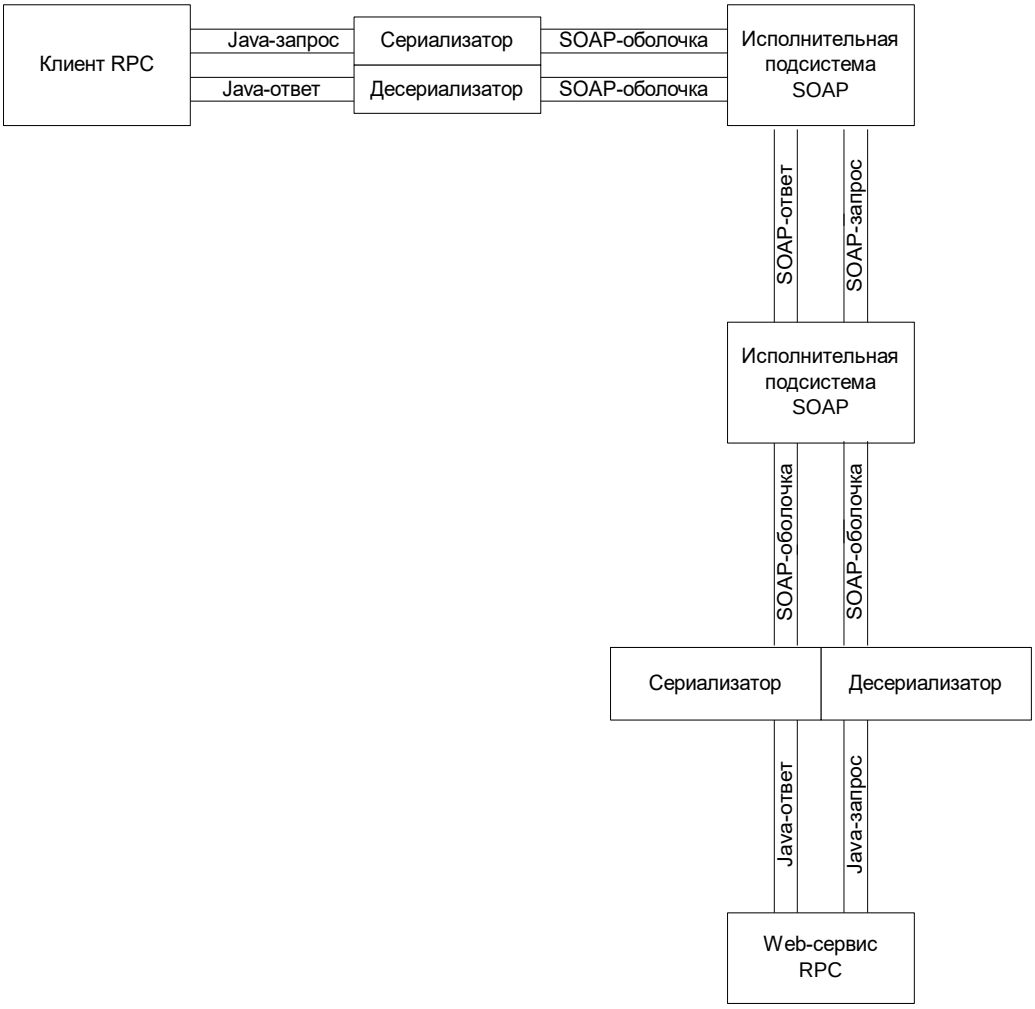

Рис. 2. RPC-подход. Подробная схема.

Рис.3. Документоориентированный подход. Подробная схема.

Основными различиями являются сериализация/десериализация и семантика.

В документоориентированном SOAP-программировании клиенту не нужно сериализовать вызов Java и его аргументы в документ XML. И обратно, серверу не нужно десериализовать документ XML в вызов Java и типы данных. В RPC-SOAP-программировании клиент и сервер должны придерживаться строго определенной программной модели – клиент вызывает метод с возможными аргументами, а сервер в ответ возвращает одно значение. В документоориентированном программировании в обмене участвует документ XML и значение каждого элемента интерпретируется участвующими во взаимодействии сторонами. Вдобавок к этому клиент и web-сервис могут объединить в запрос и ответ несколько документов.

Т.к. процесс сериализации/десериализации отсутствует, приложения, раработанные посредством использования документоориентирвоанного программирования, должны быть более быстрыми, чем их RPC-аналоги. Тем не менее это не обязательно будет так; это можно аргументировать тем, что при документоориентированном программировании сериализация зависит от разработчика, т.к. в некоторый момент нам может понадобиться преобразовать наши данные Java в XML и наоборот. А также задействованные доументы XML потенциально могут быть намного больше, чем простые типы запрос-ответ.

Документоориентированное SOAP-программирование подходит, когда мы хотим осуществлять обмен данными между двумя и более сторонами. Это особенно справедливо в случае, когда у нас уже есть документ XML, представляющий данные, т.к. мы можем обмениваться самим документом XML в исходном виде без необходимости преобразовывать его структуры данных Java. Отсутствие шагов сериализации/десериализации упрощает разработку и ускоряет обработку данных.

RPC-метод разработки SOAP срабатывает очень хорошо, если проблему можно легко смоделировать с помощью вызовов методов или если у нас уже есть функциональный API. Для приложений, которые манипулируют документами XML с помощью XSLT, документ XML более предпочтителен, Чем структуры данных Java, т.к. процессор XSLT может автоматически преобразовывать этот документ.

SAAJ (SOAP with Attachments API for Java) используется главным образом для обмена SOAP-сообщениями в рамках JAX-RPC и JAXR-реализаций. К тому же, это такой API, который разработчики могут использовать, если они хотят писать SOAP-приложения напрямую, нежели через JAX-RPC.

SAAJ API удовлетворяет спецификациям SOAP 1.1 и SOAP 1.2 и спецификации SOAP with Attachments.

Рассмотрим SAAJ применительно к двум аспектам: сообщения (*messages*) и соединения (*connections*).

SAAJ сообщения следуют стандартам SOAP. Существует два типа SOAP-сообщений: с прикреплениями (*with attachments*) и без. Мы использовали SOAP без прикреплений. Высокоуровневая структура таких сообщений показана на рис.1.

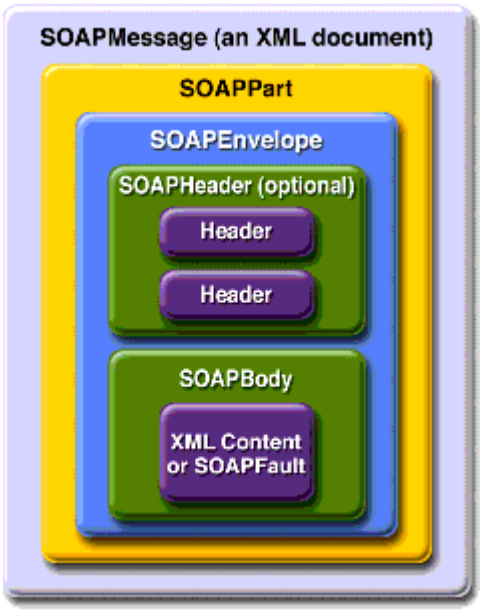

Рис.4. Объект SOAPMessage без прикреплений.

SAAJ API предоставляет класс *SOAPMessage* для представления SOAP-сообщения, *SOAPPart* для представления части SOAP, *SOAPEnvelope* – для SOAP-конверта и т.д.

Когда вы создаете объект *SOAPMessage*, он автоматически имеет разделы, которые требуются для SOAP-сообщения. Другими словами, в объект *SOAPMessage* входит объект *SOAPPart*, который содержит объект *SOAPEnvelope*. В свою очередь объект *SOAPEnvelope* автоматически содержит пустой *SOAPHeader* (заголовок), за которым следует пустой *SOAPBody* (тело).

Объект заголовка *SOAPHeader* может включать один или более заголовков, которые содержат метаданные о сообщении.

Все SOAP-сообщения отправляются и принимаются посредством соединения (*connection*). В SAAJ API соединение представлено объектом *SOAPConnection*, который связывает отправителя прямо с пунктом назначения. Такой тип соединения называется *point-to-point соединение*, потому что оно связывается две конечных точки. Сообщения, отправляемые с использованием SAAJ API, называются *request-response сообщения*. Они отправляются через объект *SOAPConnection* методом *call*, который отправляет сообщение (запрос), а затем блокируется в ожидании ответа.

Apache Axis вообще предусматривает 4 стиля сервисов. Кроме *RPC*, *Document*, это еще *Wrapped* и *Message*. Стили необходимо указывать в дескрипторе развертывания (в методических указаниях определим, как). **Мы будем использовать стиль *Message*** (он работает с XML-данными как они написаны, не превращая их в Java-объекты), а не с *Document*, как можно было подумать (*Document* привязывает Java-объекты к XML, так что разработчик имеет дело с Java-объектами, а не прямо с XML-конструкцией). Для объявления метода сервиса в стиле *message* существуют четыре действительные сигнатуры:

public Element [] method(Element [] bodies);
public SOAPBodyElement [] method (SOAPBodyElement [] bodies);
public Document method(Document body);
public void method(SOAPEnvelope req, SOAPEnvelope resp);

Теперь у нас достаточно теоретического основания для выполнения лабораторного задания. Предполагается, что студент имеет навыки составления XML и XSL-документов.

Методические указания

Задание.

На сервере (например, в папке D:\work\server) находится XML-документ, содержащий информацию о наличии товаров в магазине. Клиент отправляет следующий запрос к web-сервису:

```
<getfile>
        <file>ИМЯ_ФАЙЛА</file>
</getfile> ,
```

где ИМЯ_ФАЙЛА вводится пользователем. Web-сервис должен отправить клиенту файл с таким именем из сервера.

Задание выполним с учетом того, что сам файл и web-сервис находятся на одной машине. Клиент может быть удаленным.

1. Для начала, нам **необходимо составить XML-документ**. Например, такой:

```
<?xml version="1.0" encoding="windows-1251"?>
<?xml-stylesheet href="goods.xsl" type="text/xsl"?>

    <goods>
        <good ID="GSM">
            <type ID="Nokia">
                <code>100001</code>
                <model>3310</model>
                <price>40</price>
            </type>
            <type ID="Sony Ericsson">
                <code>100002</code>
                <model>T630</model>
                <price>150</price>
            </type>
        </good>
        <good ID="TV">
            <type ID="Horizont">
                <code>100003</code>
                <model>modellll</model>
                <price>170</price>
            </type>
            <type ID="Vityaz">
                <code>100004</code>
                <model>Europe</model>
                <price>185</price>
            </type>
        </good>
        <good ID="computer">
            <type ID="Dell">
                <code>100005</code>
                <model>Computers</model>
                <price>999</price>
            </type>
            <type ID="Apple">
                <code>100006</code>
                <model>Macintosh</model>
                <price>1999</price>
            </type>
        </good>
    </goods>
```

Сохраним его в файл D:\work\server\goods.xml.

Здесь приведем текст XSL-файла, который используется для визуализации XML-документа. Хотя для выполнения задания он не нужен, однако может пригодится для проверки правильности составления XML-документа, выполнения дополнительного задания преподавателя и т.д.

D:\work\server\goods.xsl имеет вид:

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
 <h1><b><center><font color="#ff0000">Товары:</font></center></b></h1>
      <table border="5">
      <xsl:for-each select="/goods/good">
       <tr bgcolor="cyan"><td colspan="3" width="600">
       <h2><font color="brown"><xsl:value-of select="@ID"/></font></h2>
       </td></tr>

       <xsl:for-each select="type">
        <tr>
            <td colspan="3" width="600"><center><h3><xsl:value-of
                                        select="@ID"/></h3></center></td>
        </tr>
        <tr>
            <td><xsl:value-of select="code"/></td>
            <td><xsl:value-of select="model"/></td>
            <td><xsl:value-of select="price"/></td>
        </tr>
      </xsl:for-each>
      </xsl:for-each>
      </table>
</xsl:template>
</xsl:stylesheet>
```

2. Теперь **напишем web-сервис StoreService.java**, выбрав четвертую сигнатуру (по желанию можно любую другую).

```
import org.apache.axis.message.SOAPBodyElement;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.AxisFault;
import org.apache.axis.MessageContext;
import org.w3c.dom.*;
import org.w3c.dom.Node;
```

```
import org.w3c.dom.Text;
import org.xml.sax.SAXException;

import javax.xml.transform.TransformerException;
import javax.xml.soap.*;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import java.io.IOException;
```

Пишем класс web-сервиса. Главный его метод – это *storeService()*. Он получает запрос в виде SOAP-сообщения и автоматически формирует ответное SOAP-сообщение. Однако мы должны ответ изменить в соответствии с заданием, то есть записать в него искомый XML-документ. Хотя сигнатура метода оперирует типами *SOAPEnvelope*, а не *SOAPMessage*, эти типы легко преобразуются друг к другу. Мы же отталкиваемся от структуры *SOAPMessage* в связи с реализуемым стилем *Message*.

Методу *storeService()* необходимо проанализировать запрос, выделив из него имя файла, а потом составить SOAP-ответ, который и будет автоматически послан отправителю. Поэтому нам удобно написать еще две функции, помимо главного метода, это *getFileName()* – она анализирует запрос и возвращает имя файла и *createSOAPresponse()* – она формирует SOAP-ответ на основе уже предусмотренного для ответа конверта и зная имя файла с XML-документом.

```
public class StoreService {
```

Итак, выделим имя файла:

```
public String getFileName(SOAPEnvelope req) throws AxisFault,
                                                    SOAPException {
```

Из входного конверта получим тело сообщения. Для этого нам нужен его элемент. То есть корневой элемент и для всех остальных элементов XML=запроса он будет являться предком.

```
SOAPBodyElement reqBody =  (SOAPBodyElement)
                                    req.getBodyElements().get(0);

String str= "";
```

Теперь в переменной *reqBody* находится структура <getfile><file>goods.xml</file></getfile>. Нам необходимо добраться до текстового узла, который в данном случае равен “goods.xml”. Будем использовать иерархическую модель DOM XML-документа. При этом надо иметь в виду, что не только тег <file> является дочерним по отношению к тегу <getfile>, но и символ конца строки, символ перевода каретки, поэтому мы

проверяем найденный узел, есть ли он экземпляр класса *Element* в строке *if (currentNode instanceof Element)*.

Возьмем все дочерние узлы для <getfile>:

```
NodeList nodes = reqBody.getChildNodes();
```

Циклом выделим узел, который является тегом <file>:

```
for (int i=0;i<nodes.getLength();i++){
    Node currentNode = nodes.item(i);
    if (currentNode instanceof Element){
```

теперь *currentNode* равен *<file>goods.xml</file>+*
Приведем этот узел к типу элемент:

```
Element element = (Element) currentNode;
```

Теперь доберемся до дочернего текстового узла, который и является именем нужного файла. Для обозначения текстовых узлов имеется класс *Text*, поэтому делать проход с помощью циклов теперь нет необходимости. Взять сам текст у текстового узла можно с помощью функции *getData()*.

```
        Text textNode = (Text) element.getFirstChild();
        str = textNode.getData().trim();
      }
    }
```

возвратим результат:

```
  return str;
}
```

Следующий метод создает SOAP-конверт с ответом:

```
public SOAPEnvelope createSOAPResponse(String fileName,
                    SOAPEnvelope resp) throws SOAPException,
                        ParserConfigurationException,
                                IOException, SAXException {
```

Java API for XML Processing (JAXP) предоставляет API для DOM-парсеров. Воспользуемся встроенными реализациями JAXP. Как альтернативу, можно было использовать и классы анализатора *Xerces*. В последнем случае, эту библиотеку надо скачивать отдельно или взять у преподавателя.

Класс *DocumentBuilderFactory* предоставляет фабричные методы для создания экземпляров *DocumentBuilder*. Класс *DocumentBuilder* предоставляет

методы для синтаксического анализа документов XML в деревья DOM, создания новых документов XML и так далее.

```
DocumentBuilderFactory dbFactory =
                        DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document document =
                    builder.parse("d:\\work\\server\\"+fileName);
```

Теперь в объекте *document* у нас находится документ, который и будет телом SOAP-ответа. Далее логика довольно проста. Мы получаем текущий контекст сообщения (не забываем, что при данной сигнатуре главного метода web-сервиса он формируется автоматически), получаем его SOAP-часть, конверт *envelope*, заголовочную часть, которую за ненадобностью отцепляем и наконец раздел тела SOAP:

```
MessageContext msgCntxt = MessageContext.getCurrentContext();
SOAPMessage respMess = msgCntxt.getMessage();

SOAPPart soapPart = respMess.getSOAPPart();
SOAPEnvelope envelope = (SOAPEnvelope) soapPart.getEnvelope();
SOAPHeader header = resp.getHeader();

SOAPBody body = resp.getBody();
header.detachNode();
```

Теперь назначим корневым элементом тела сообщения корневой элемент нашего XML-документа (а он уже отождествлен с объектом *document*):

```
    SOAPBodyElement docElement = (SOAPBodyElement)
body.addDocument(document);
```

Возвратим в метод web-сервиса полученный результативный конверт:

```
    return envelope;
  }
```

И наконец, главный метод web-сервиса, согласно четвертой сигнатуре:

```
  public void storeService(SOAPEnvelope req,SOAPEnvelope resp) throws
            Exception,TransformerException, IOException {
      String fileName = getFileName(req);
      resp = createSOAPResponse(fileName, resp);
  }
}
```

3. **Напишем клиентское приложение *StoreMessage.java*.**

Общий алгоритм работы клиентского приложения выглядит так:
Ввод имени файла, который находится по договоренности в D:\work\server\

- Формирование SOAP-запроса (*createSOAPRequest()*)
- Устанавление соединения (*createConnection()*)
- Отправка/получение SOAP-сообщения (*call()*)
- Вывод на экран тела ответа, т.е. искомого XML-документа (*displayMessage()*)

```
import org.apache.axis.client.Call;

import javax.xml.soap.*;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPBodyElement;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.transform.TransformerException;

import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.net.URL;
import java.net.MalformedURLException;
import java.util.Iterator;

public class StoreMessage {

  static String fileXml;
  static SOAPMessage reqMess,respMess;
```

составим запрос

```
  public void createSOAPRequest() throws SOAPException,
                                         TransformerException {
```

Создадим сообщение, используя объект *MessageFactory*. SAAJ API обеспечивает реализацию класса *MessageFatcory* так, чтобы упростить получение его экземпляра:

```
MessageFactory factory = MessageFactory.newInstance();
reqMess = factory.createMessage();
```

Доступ к элементам сообщения получаем аналогично тому, как объяснялось выше:

```
SOAPPart soapPart = reqMess.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
header.detachNode();
```

Когда мы создаем новый элемент, мы должны создать объект *Name*, которым элемент будет уникально определен. Объект *Name* связан с объектами *SOAPBodyElement* и *SOAPHeaderElement*. Он может объявляться с уточнениями префикса для используемого пространства имен и URI пространства имен, помимо самого имени. Мы используем краткую форму.

```
Name bodyName = envelope.createName("GetFile");
SOAPBodyElement bodyElement =
                        body.addBodyElement(bodyName);

Name name = envelope.createName("file");
SOAPElement fileName = bodyElement.addChildElement(name);
fileName.addTextNode(fileXml);

}
```

Функция для отображения тела документа на экран:

```
public static void displayMessage(SOAPMessage mess) throws
                                                SOAPException{
    SOAPBody body = mess.getSOAPBody();
    Iterator it = body.getChildElements();
    SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
    System.out.println(bodyElement);

}
```

Считываем с входного потока (с консоли) имя XML-файла, который должен находится в D:\work\server\ на той же машине, что и сервер с web-сервисом:

```
public void start(){

   BufferedReader br = new BufferedReader(new
                                        InputStreamReader(System.in));
   try {
         System.out.println("Enter the file XML:");
        fileXml = br.readLine();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Главная функция – *main()*:

```
    public static void main(String[] args) throws SOAPException,
                        MalformedURLException, TransformerException {

            StoreMessage client = new StoreMessage();
            client.start();
            client.createSOAPRequest();

            SOAPConnectionFactory soapConnectionFactory =
                                    SOAPConnectionFactory.newInstance();
            SOAPConnection connection =
                                    soapConnectionFactory.createConnection();

            URL endpoint = new
                        URL("http://localhost:8080/axis/services/StoreService");

            respMess = connection.call(reqMess, endpoint);

            displayMessage(respMess);
    }
}
```

Web-сервис, реализованный для обмена сообщениями по принципу запрос-ответ должен возвращать ответ на любое сообщение, которое он получает. Ответом является объект *SOAPMessage*, классу которого принадлежит и запрос. Некоторые сообщения могут не получать вообще никакого ответа. Сервис, который получает такое сообщение все еще должен отправить ответ, потому что необходимо разблокировать метод *call()*. В этом случае ответ не связан с содержимым сообщения; это просто сообщение для разблокировки метода *call()*.

4. **Компиляция классов клиента и сервиса.**

Компиляцию нужно производить, подключая все задействованные библиотеки (переменную CATALINA_HOME присвойте своему месторасположению Tomcat):

```
set CATALINA_HOME=f:\Tomcat 5.0

set classpath=.;%CATALINA_HOME%\webapps\axis\WEB-INF\ lib\
axis.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\ lib\
```

jaxrpc.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\lib\commons-logging-1.0.4.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\ lib\commons-discovery- 0.2.jar; %CATALINA_HOME%\ webapps\ axis\ WEB-INF\lib\saaj.jar; %CATALINA_HOME%\common\ endorsed\ xercesImpl.jar;%CATALINA_HOME%\common\endorsed\ xmlParserAPIs.jar

javac StoreMessage.java
javac StoreService.java

5. **Развертывание сервиса на Axis.**

Необходимо написать дескриптор развертывания (*StoreService.wsdd*):

```
<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <service name="StoreService" provider ="java:MSG">
      <parameter name="className" value = "StoreService"/>
      <parameter name="allowedMethods" value="storeService"/>
    </service>
</deployment>
```

Теперь

1) поместите класс web-сервиса (*StoreService.class*) в Tomcat 5.0\ webapps\axis\WEB-INF\classes\
2) в рабочей папке, в которой расположен класс web-сервиса, создайте командный файл с именем, например, *deploy.cmd* с таким содержимым (переменную CATALINA_HOME присвойте своему месторасположению Tomcat):

```
set  CATALINA_HOME=F:\Tomcat 5.0
java -cp "%CATALINA_HOME%\webapps\axis\WEB-INF\lib\axis.jar;
%CATALINA_HOME%\webapps\axis\WEB-INF\lib\jaxrpc.jar;
%CATALINA_HOME%\webapps\axis\WEB-INF\lib\commons-logging-
1.0.4.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\lib\commons-
discovery-0.2.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\lib\saaj.jar;
%CATALINA_HOME%\common\lib\activation.jar;%CATALINA_HOME%\
common\lib\mail.jar" org.apache.axis.client.AdminClient StoreService.wsdd
pause
```

3) запустите Tomcat. Он должен быть запущен по порту 8080.

4) теперь запустите *deploy.cmd*. При успешном выполнении развертывания консольное окно должно выглядеть примерно так (рис. 5):

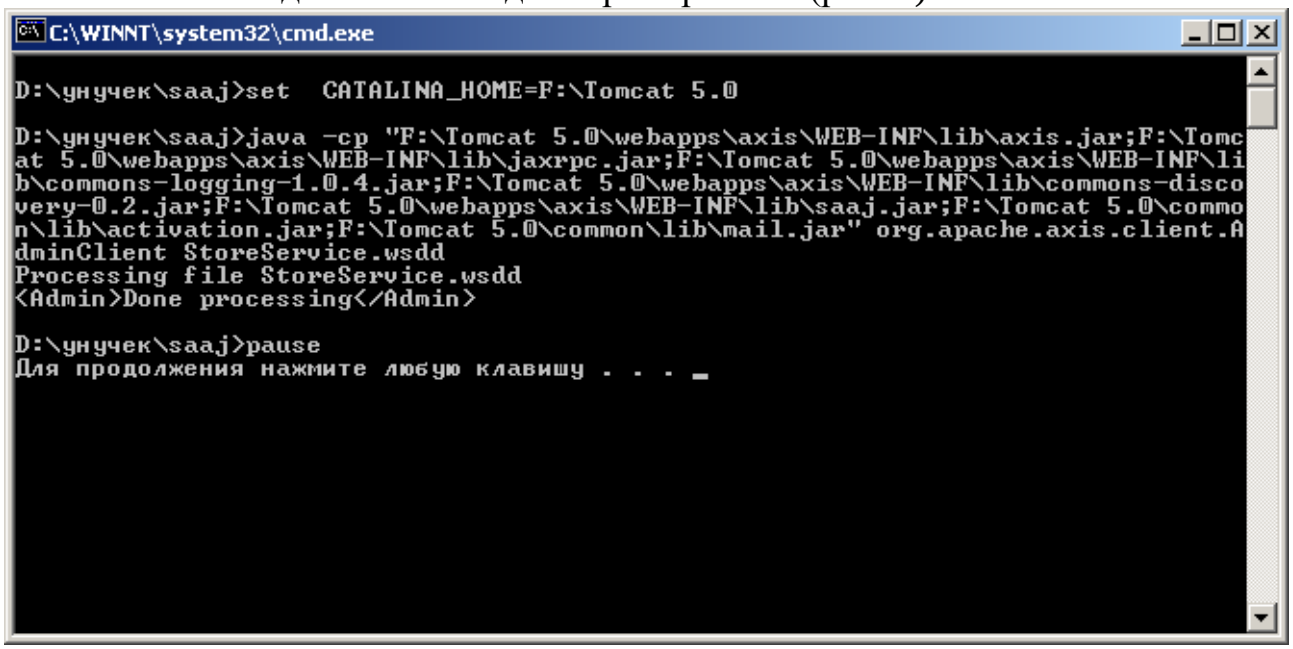

Рис.5. Успешное развертывание сервиса классом AdminClient

После этого мы можем найти наш сервис в списке развернутых сервисов в Axis (рис. 6):

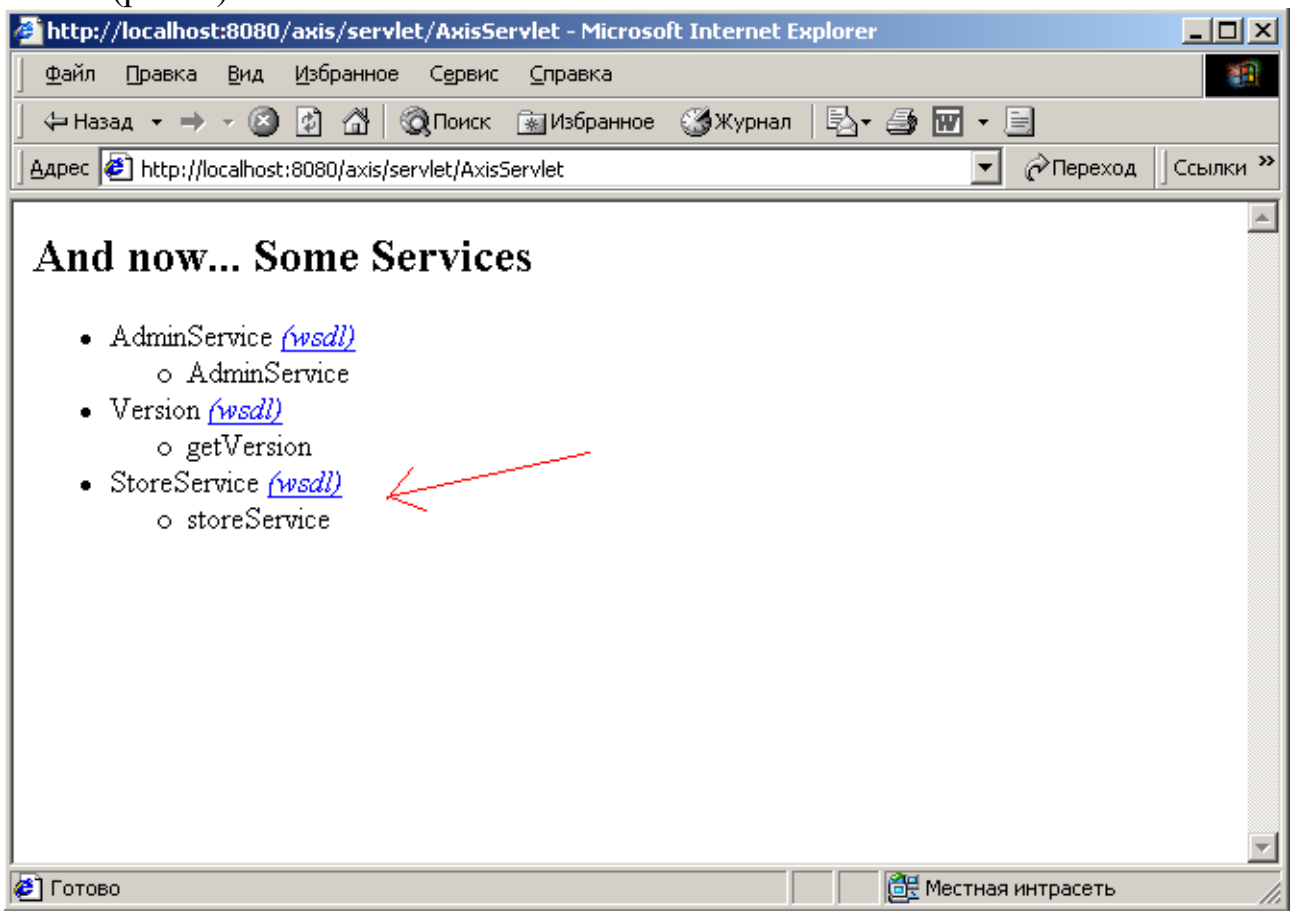

Рис.6. Пункт меню *List*

6) Таким образом сервер запущен, запускаем клиента (переменную CATALINA_HOME присвойте своему месторасположению Tomcat):

set CATALINA_HOME=F:\Tomcat 5.0

set classpath=.;%CATALINA_HOME%\webapps\axis\WEB-INF\ lib\ axis.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\lib\jaxrpc.jar; %CATALINA_HOME%\webapps\axis\WEB-INF\lib\commons-logging-1.0.4.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\lib\commons-discovery-0.2.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\lib\saaj.jar; %CATALINA_HOME%\common\endorsed\xercesImpl.jar;%CATALINA_HOME%\common\endorsed\xmlParserAPIs.jar;%CATALINA_HOME%\common\lib\activation.jar;%CATALINA_HOME%\common\lib\mail.jar;%CATALINA_HOME%\webapps\axis\WEB-INF\lib\wsdl4j-1.5.1.jar

java StoreMessage

После запуска появится приглашение ввести название XML-файла – вводим *goods.xml*. Вывод правильно работающей программы на рис.7.

Рис.7. Результат работы программы

Контрольные вопросы:

1. Что является основным инструментом взаимодействия при использовании документоориентированного подхода?
2. Как работает документоориентированное взаимодействие?
3. В чем его различие с RPC-взаимодействием?
4. Почему при документториентирвоанном программировании не требуется сериализация/десериализация объектов?
5. Когда лучше применять RPC-модель, а когда документную модель?
6. Объясните структуру SOAP-сообщения
7. Каким образом происходит соединения клиента с web-сервисом (при документоориентированном взаимодействии)?
8. Какие стили предусматривает Apache Axis? Почему message-style кажется наиболее подходящим для выполнения этой лабораторной работы?
9. Перечислите все допустимые сигнатуры методов на стороне сервиса при реализации стиля сообщений?
10. Как работает DOM-парсер? Почему мы используем цикл даже при доступе к единственному дочернему элементу?

Варианты индивидуального задания

1. Информация о телевизорах хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
2. Информация о музыкальных форматах хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
3. Информация о компьютерах хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
4. Информация о мобильных телефонах хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
5. Информация о сотрудниках хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
6. Информация о книгах хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
7. Информация о гостиницах хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
8. Информация о курсах валют хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
9. Информация о туристических направлениях хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
10. Информация о предприятии хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
11. Информация об оценках студентов хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
12. Информация о вокзале хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
13. Информация о больнице хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.
14. Информация о зоопарке хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.

15. Информация об автомобилях хранится на сервере в виде XML-файла. Написать web-сервис, возвращающий содержимое XML-файла клиентскому приложению.