Лекция 22

SOAP

Интернет объединяет в себе много различных платформ, а информация содержится в разнообразных источниках данных.

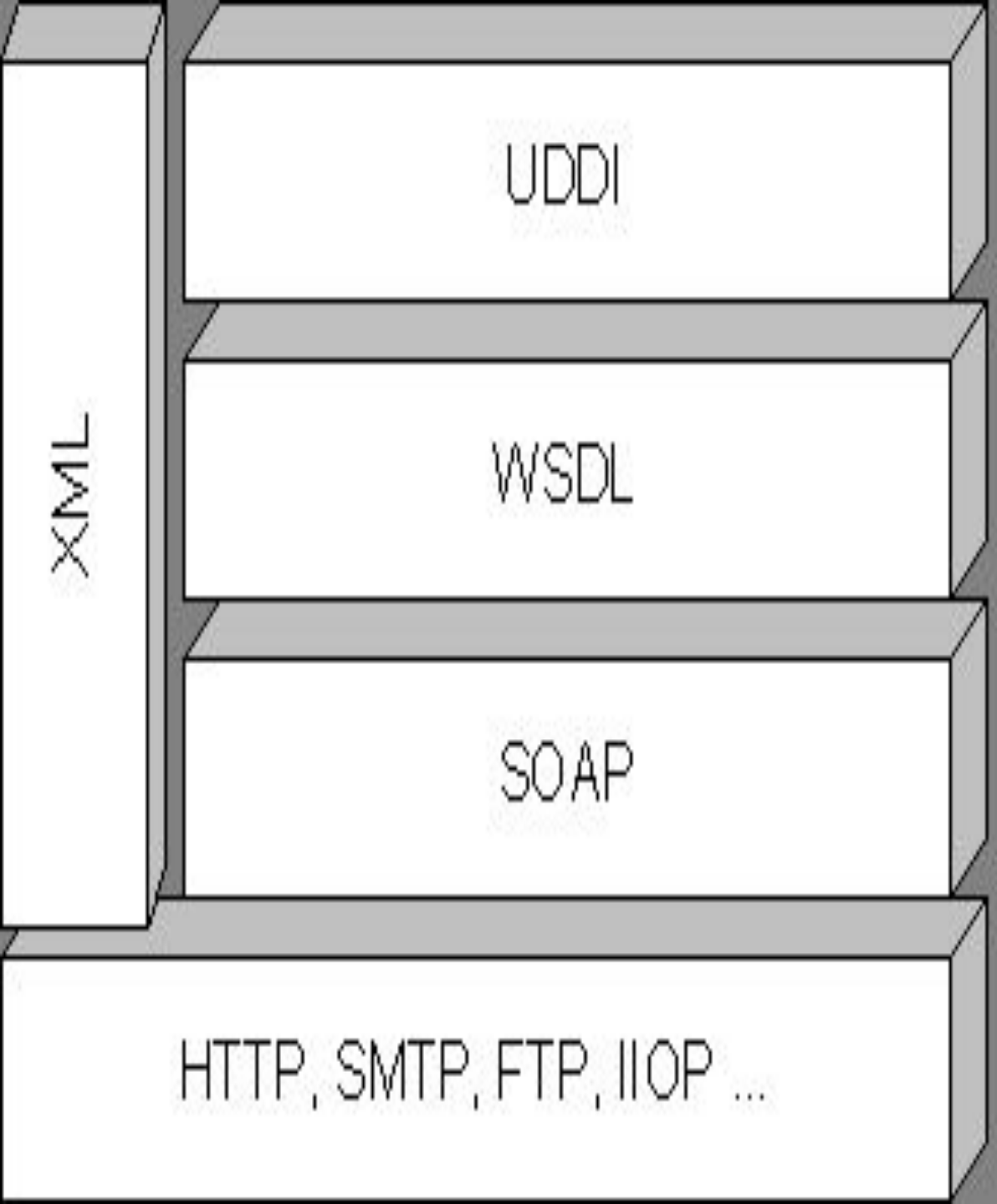
Концепция веб-сервисов (Web Services) призвана решить эту задачу объединения, интеграции разнородных систем на основе открытых стандартов.

Основные положения модели веб-сервисов

Веб-сервисы являются концепцией создания таких приложений, функции которых можно использовать при помощи стандартных протоколов Интернет.

Концепция веб-сервисов реализуется при помощи ряда технологий, которые стандартизованы World Wide Web Consortium (W3C).

Взаимосвязь этих технологий можно условно представить следующим образом.



Публикация и поиск сервисов

Описание интерфейсов сервисов

Обмен сообщениями

Транспортная инфраструктура

XML является фундаментом для создания большинства технологий, связанных с веб-сервисами.

Для удаленного взаимодействия с веб-сервисами используется Simple Object Access Protocol (SOAP).

SOAP обеспечивает взаимодействие распределенных систем, независимо от объектной модели, операционной системы или языка программирования.

Данные передаются в виде особых XML документов особого формата.

Согласно определению W3C, веб-сервисы это приложения, которые доступны по протоколам, которые являются стандартными для Интернет.

Нет требования, чтобы веб-сервисы использовали какой-то определенный транспортный протокол.

Спецификация SOAP определяет, каким образом связываются сообщения SOAP и транспортный протокол

Технология Universal Description, Discovery and Integration (UDDI) предполагает ведения реестра веб-сервисов.

Подключившись к этому реестру, потребитель сможет найти веб-сервисы, которые наилучшим образом удовлетворяют его потребностям

Веб-сервисы позиционируются как программное обеспечение промежуточного слоя.

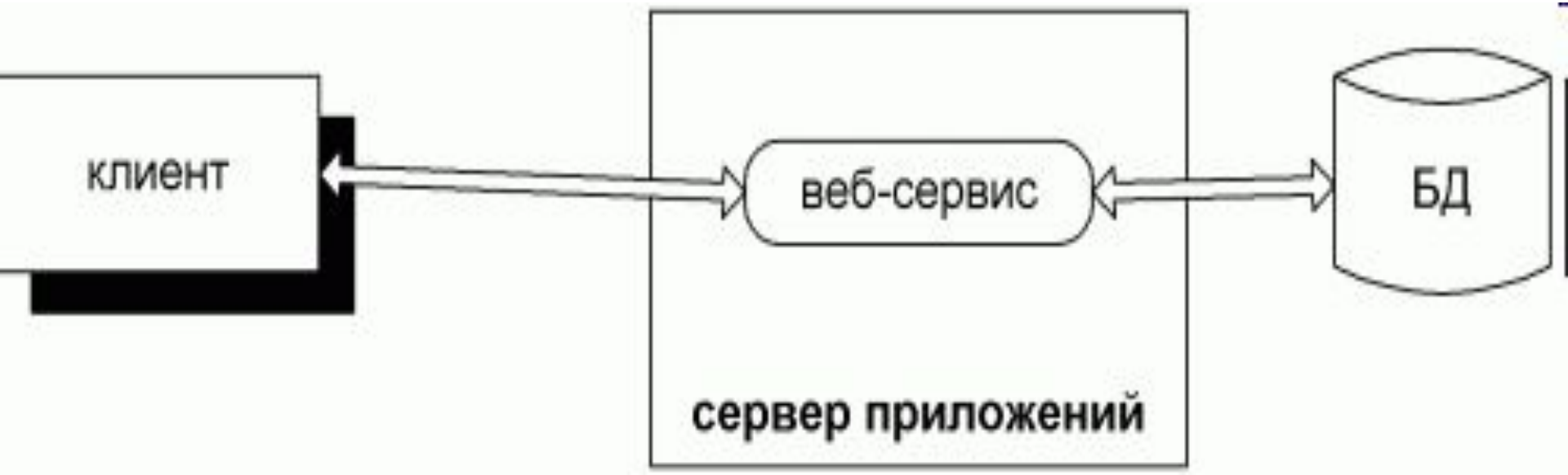
Использовать веб-сервисы могут как клиентские приложения, непосредственно работающие с пользователем, так и другие приложения.

Веб-сервисы размещаются на серверах приложений.

Существует несколько концепций применения Веб-сервисов.

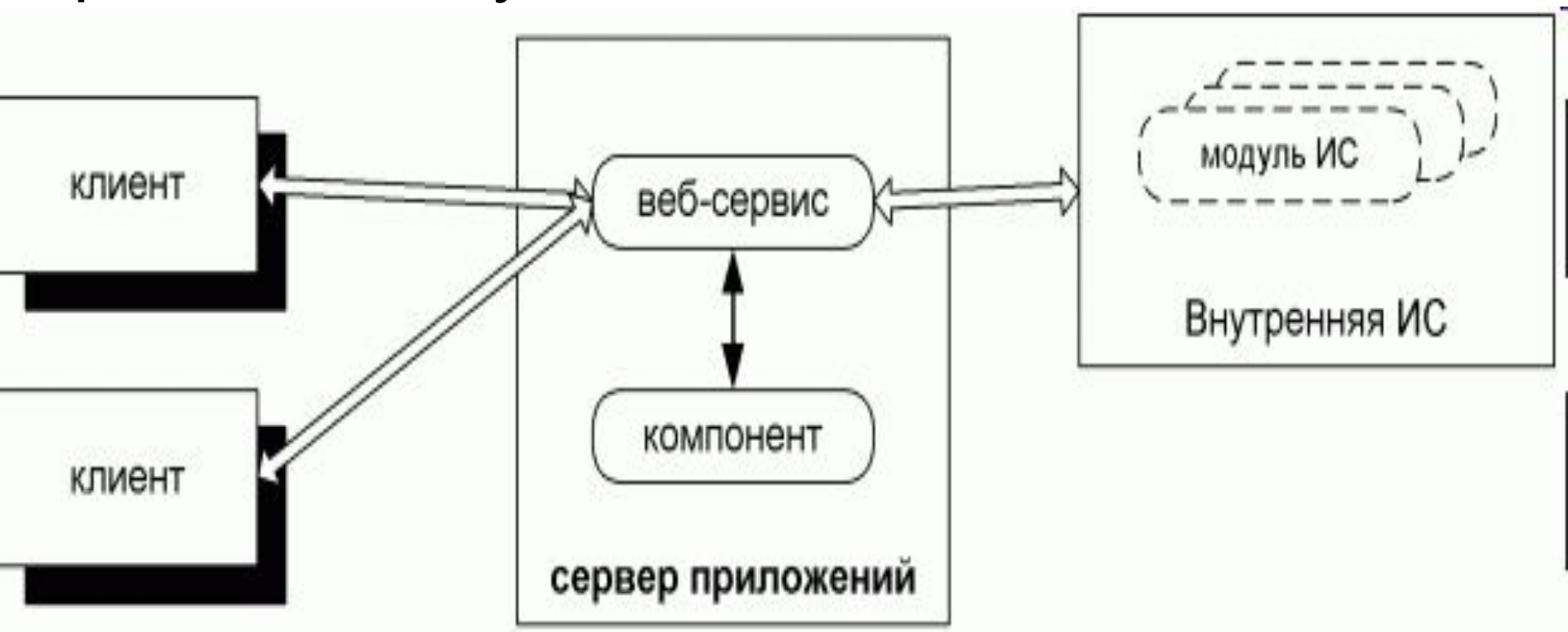
1. Веб-сервисы как реализация логики приложения (бизнес-логики).

   То есть, создание нового приложения бизнес-логика, которого реализуется в веб-сервисе.


клиент
веб-сервис
БД
сервер приложений

2. Веб-сервисы как средство интеграции.

То есть, использование веб-сервиса как способа доступа удаленных клиентов к внутренней ИС компании, или для организации взаимодействия компонента (например, EJB, COM-компонента) с различными удаленными клиентами.



Рассмотрим пример создания и развертывания Веб – сервиса.

Пусть Веб-сервис возвращает строку клиенту.

Тогда класс, реализующий данный Веб-сервис будет иметь вид (HelloService.java):

```
package samples.mysimple;
public class HelloService{
  private static int a=25;
  public HelloService(){
      a++;
   }
  public String myMethod(String s){
      return s+Integer.toString(a);
  }
}
```

Таким образом, логика Веб-сервиса заключена в методе myMethod.

Клиент, вызывающий данный Веб-сервис будет иметь следующий вид (Client.java):

```
package samples.mysimple;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;
 public class Client{
   public static void main(String [] args) throws Exception {
    String endpoint ="http://localhost:8080/axis/services/HelloService";
    String method = "myMethod";
    String s1 ="Hello";
```

```
//Создаем клиента для использование Web-сервиса
Service  service = new Service();

//Создаем класс динамического связывания клиента с Веб-сервисом
Call call    = (Call) service.createCall();

//Указываем где находится Веб -Сервис
call.setTargetEndpointAddress(new java.net.URL(endpoint));

//Указываем метод, который будем вызывать.
call.setOperationName( new QName("http://mysimple.samples",
                                                    method));

// Определяем тип параметра, передаваемого в метод, в XML формате
call.addParameter("op1",
                org.apache.axis.encoding.XMLType.XSD_STRING,
                                                    ParameterMode.IN);
```

```
  //Определяем тип возвращаемого методом
    значения,в XML формате
  call.setReturnType(org.apache.axis.encoding.
                                XMLType.XSD_STRING);
  //Вызываем метод
  String ret = (String)call.invoke(
                                    new Object [] { s1 });
  System.out.println("Got result : " + ret);
 }
}
```

Рассмотрим процедуру развертывания данного Веб-сервиса.

Для использования данного Веб-сервиса необходимы:

1. Apache Tomcat
2. Apache axis(axis2)
3. Пакет Xerces-J-bin.2.11.0.zip(сайт Apache)
4. xml-security-bin-1_4_4.zip(сайт Apache)

(Далее предполагается, что Apache Tomcat уже установлен в директории C:\Tomcat)

1. Разворачиваем пакет axis-bin-1_4.zip в директорию C:\axis

2. Копируем папку axis из директории c:\axis\webapp в директорию c:\tomcat\webapp
3. Копируем все jar файлы из директории c:\axis\lib в директорию c:\tomcat\webapps\axis\WEB-INF\lib\, а также их архивов Xerces-J-bin.2.11.0.zip и xml-security-bin-1_4_4.zip
4. Создаем переменные окружения

```
set AXIS_HOME=/usr/axis
set AXIS_LIB=$AXIS_HOME/lib
set AXISCLASSPATH
        =$AXIS_LIB/axis.jar:$AXIS_LIB/commons-discovery.jar:
$AXIS_LIB/commons- logging.jar:
$AXIS_LIB/jaxrpc.jar:$AXIS_LIB/saaj.jar:
$AXIS_LIB/log4j-1.2.8.jar:$AXIS_LIB/xml-apis.jar:
$AXIS_LIB/xercesImpl.jar


export AXIS_HOME;
export AXIS_LIB;
export AXISCLASSPATH
```

5. Создаем папку c:\axis\samples\mysimple\ и копируем туда файлы Client.java и HelloService.java

6. Копилируем данные файлы

>javac –cp “все необходимые jar файлы из папки c:\tomcat\webapps\axis\WEB-INF\lib\” *.java

7. Создаем дескрипторы развертывания в папке c:\axis\samples\mysimple\

deploy.wsdd

undeploy.wsdd

Файл deploy.wsdd имеет вид:

```
<deployment
   xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/
                                        providers/java">
  <service name="HelloService“ provider="java:RPC">
    <parameter name="className“
               value="samples.mysimple.HelloService"/>
    <parameter name="allowedMethods“ value="*"/>
  </service>
</deployment>
```

Файл undeploy.wsdd имеет вид:

```
<undeployment
        xmlns="http://xml.apache.org/axis/wsdd/">
   <service name="HelloService"/>
</undeployment>
```

8. Создаем папку mysimple в каталоге c:\tomcat\webapps\axis\WEB-INF\classes\samples\

9. Копируем в эту папку файл HelloService.class

10. Запускаем Tomcat

c:\tomcat\bin\>startup

11. Развертываем Веб-сервис

c:\axis\samples\mysimple\>java –cp “все необходимые jar файлы из папки c:\tomcat\webapps\axis\WEB-INF\lib\” org.apache.axis.client.AdminClient deploy.wsdd

На экране должно появиться

<Admin>Done processing</Admin>,

А также в файле c:\tomcat\webapps\axis\WEB-INF\server-config.wsdd

запись типа:

```
<service name="HelloService" provider="java:RPC">
  <parameter name="allowedMethods" value="*"/>
  <parameter name="className“
             value="samples.mysimple.HelloService"/>
</service>
```

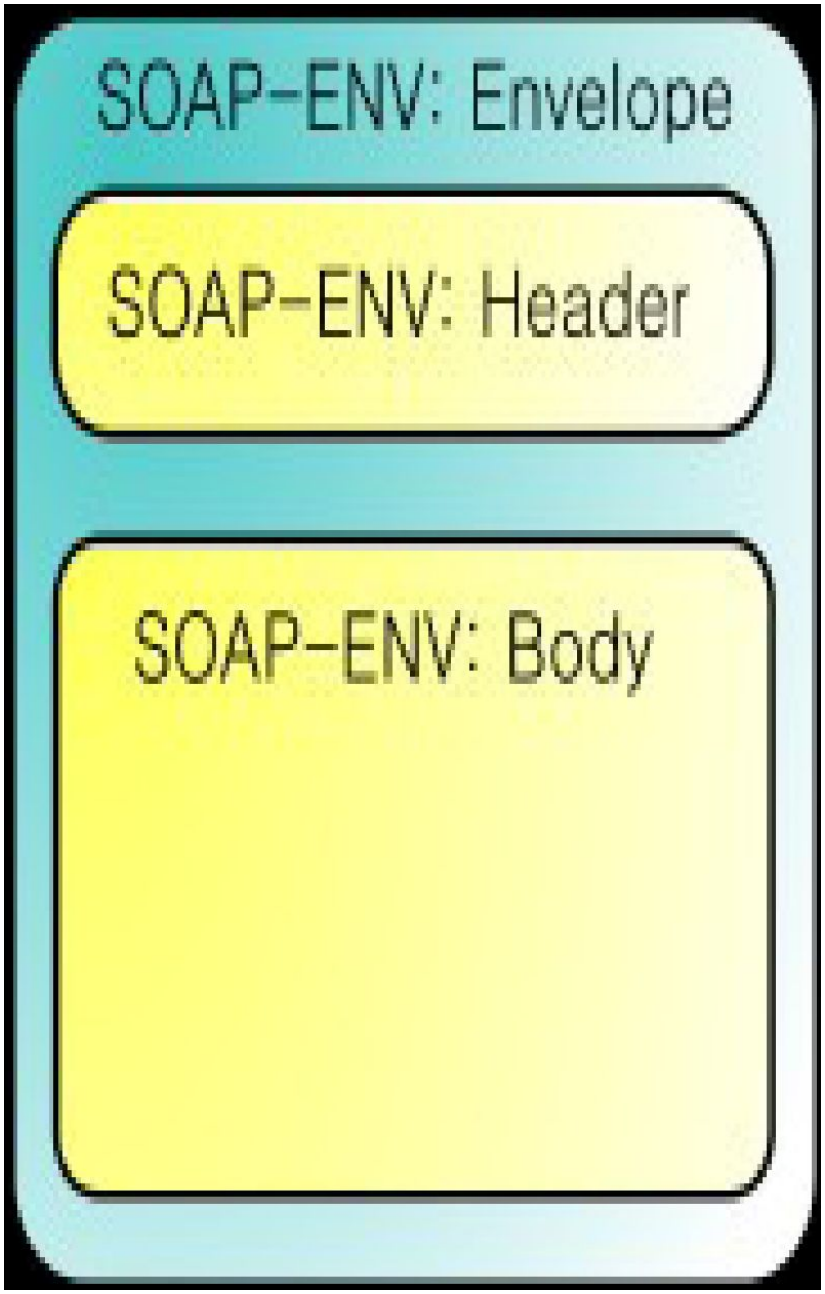
12. Запускаем клиент

>java –cp “все необходимые jar файлы из папки c:\tomcat\webapps\axis\WEB-INF\lib\” samples.mysimple.Client

Общая структура SOAP сообщения

SOAP-сообщение представляет собой XML-документ; сообщение состоит из трех основных элементов:

- конверт (SOAP Envelope)
- заголовок (SOAP Header)
- тело (SOAP Body)


SOAP-ENV: Envelope
SOAP-ENV: Header
SOAP-ENV: Body

Пример SOAP сообщения:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=http://www.w3.org/2003/05/soap
   -envelope
                                xmlns:t="www.example.com">
  <SOAP-ENV:Header> </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <t:CurrentDate>
      <Year>2011</Year>
      <Month>February</Month>
      <Day>12</Day>
      <Time>18:02:00</Time>
    </t:CurrentDate>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Конверт (SOAP Envelope)

SOAP Envelope является самым «верхним» элементом SOAP сообщения.

Содержит корневой элемент XML-документа. Описывается с помощью элемента Envelope с обязательным пространством имен http://www.w3.org/2003/05/soap-envelope для версии 1.2 и http://schemas.xmlsoap.org/soap/ для версии 1.1.

У элемента Envelope могут быть атрибуты xmlns, определяющие пространства имен, и другие атрибуты, снабженные префиксами.

Envelope может иметь необязательный дочерний элемент Header с тем же пространством имен — заголовок.

Если этот элемент присутствует, то он должен быть первым прямым дочерним элементом конверта.

Следующий дочерний элемент конверта должен иметь имя Body и то же самое пространство имен - тело.

Это обязательный элемент и он должен быть вторым прямым дочерним элементом конверта, если есть заголовок, или первым — если заголовка нет.

Элементы Header и Body могут содержать элементы из различных пространств имен

Заголовок SOAP (SOAP Header)

Первый прямой дочерний элемент конверта.

Не обязательный.

Заголовок кроме атрибутов xmlns может содержать 0 или более стандартных атрибутов:

- **encodingStyle**
- **actor (или role для версии 1.2)**
- **mustUnderstand**
- **relay**

Атрибут encodingStyle

В SOAP-сообщениях могут передаваться данные различных типов (числа, даты, массивы, строки и т.п.).

Определение этих типов данных выполняется в схемах XML (обычно — XSD).

Типы, определенные в схеме, заносятся в пространство имен, идентификатор которого служит значением атрибута encodingStyle.

Атрибут encodingStyle может появиться в любом элементе SOAP-сообщения, но версия SOAP 1.2 запрещает его появление в корневом элементе Envelope.

Атрибут actor

Тип данных URI.

Задает адрес конкретного SOAP-сервера, которому предназначено сообщение.

SOAP-сообщение может пройти через несколько SOAP-серверов или через несколько приложений на одном сервере.

Эти приложения выполняют предварительную обработку блоков заголовка послания и передают его друг другу.

Все эти серверы и/или приложения называются SOAP-узлами (SOAP nodes).

Спецификация SOAP не определяет правила прохождения послания по цепочке серверов.

Для этого разрабатываются другие протоколы, например, Microsoft WS-Routing.

В версии 1.2 атрибут actor заменен атрибутом role, потому что в этой версии SOAP каждый узел играет одну или несколько ролей.

Спецификация пока определяет три роли SOAP-узла:

Роль http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver играет конечный, целевой узел, который будет обрабатывать заголовок.

Роль http://www.w3.org/2003/05/soap-envelope/role/next играет промежуточный или целевой узел.

Такой узел может играть и другие, дополнительные роли.

Роль
http://www.w3.org/2003/05/soap-envelope/role/none
не должен играть ни один SOAP-узел.

Атрибут mustUnderstand

Тип данных — boolean.

По умолчанию 0.

Если значение равно 1, то SOAP-узел при обработке элемента обязательно должен учитывать его синтаксис, определенный в схеме документа, или совсем не обрабатывать сообщение.

Это повышает точность обработки сообщения.

В версии SOAP 1.2 вместо цифр нужно писать true или false.

Атрибут relay

Тип данных — boolean.

Показывает, что заголовочный блок, адресованный SOAP- посреднику, должен быть передан дальше, если он не был обработан.

Необходимо отметить, что если заголовочный блок обработан, правила обработки SOAP требуют, чтобы он был удален из уходящего сообщения.

В блоках заголовка могут быть атрибуты role, actor и mustUnderstand.

Действие этих атрибутов относится только к данному блоку.

Рассмотрим пример:

```
<env:Header>
  <t:Transaction
   xmlns:t=http://example.com/transaction
   env:role="http://www.w3.org/2003/05/
                soapenvelope/role/ultimateReceiver"
                              env:mustUnderstand="true">
   5
</t:Transaction>
</env:Header>
```

SOAP-сообщения с вложениями

Существуют ситуации, когда клиент и сервер должны обмениваться данными в формате, отличном от текстового.

С точки зрения обмена данными все нетекстовые данные рассматриваются как данные в двоичных кодах.

Двоичные данные включаются в сообщение в виде «вложения».

Структура SOAP-сообщения с вложениями имеет вид:


SOAP-ENV: Envelope
SOAP-ENV: Header
SOAP-ENV: Body
Attachments

Этот протокол определяет пересылку SOAP-сообщения внутри MIME-сообщения, состоящего из нескольких частей.

Первая часть MIME-сообщения- часть SOAP - содержит XML: конверт SOAP с вложенными в него заголовком и телом сообщения.

Остальные части - вложения - содержат данные в любом формате, двоичном или текстовом.

Каждая часть предваряется MIME-заголовком, описывающим формат данных части и содержащим идентификатор части

Рассмотрим пример:

```
MIME-Version: 1.0
Content-Type: Multipart/Related;
                                boundary=MIME_boundary;
type="application/xop+xml";
start="<soapmsg.xml@leonardo.com>";
start-info="text/xml"
Content-Description: An XML document with
                                        binary data in it
--MIME_boundary
```

Content-Type: application/xop+xml;
charset=UTF-8;
type="text/xml"
Content-Transfer-Encoding: 8bit
Content-ID: soapmsg.xml@daily-moon.com
<env:Envelope . . . >
  <env:Header>
  . . .
  </env:Header>
<env:Body>
  . . .
<xop:include
  xmlns:xop='http://www.w3.org/2004/08/xop/include‘
  href='cid:http://leonardo.com/mona_liza.jpg'/>
  . . .
 </env:Body>
</env:Envelope>

--MIME_boundary

Content-Type: image/jpg

Content-Transfer-Encoding: binary

Content-ID:
<http://leonardo.com/mona_liza.jpg>

// двоичное представление файла изображения

--MIME_boundary--

Axis2

В основе инфраструктуры Web-сервисов Apache Axis2 лежит новая модель XML документов AXIOM, обеспечивающая эффективную обработку сообщений SOAP, в отличии от Axis.

AXIOM - одна из основных инноваций в Axis2, и одна из причин того, что Axis2 показывает гораздо лучшие рабочие характеристики, чем оригинальный Axis.

API модели AXIOM ближе всего к DOM по общим свойствам, но имеет собственные характерные черты.

Например, методы доступа построены на основе java.util.Iterator экземпляров класса для доступа к компонентам.

Вместо индексации компонентов в списке навигатор использует методы **getNextOMSibling()** и **getPreviousOMSibling()** из класса org.apache.axiom.om.OMNode для последовательного продвижения по узлам на уровне дерева документа (сходные с DOM в этом случае).

Такое структурирование методов доступа и навигации отвечает работе по построению дерева по требованию, поскольку это значит, что AXIOM может позволить перейти к первому дочернему элементу стартового элемента без необходимости сперва обработать все родительские элементы для стартового.

AXIOM построен на основе интерфейса StAX анализатора.

Таким образом, AXIOM использует интерфейсы StAX Reader и Writer для обмена с внешним миром, как показано на схеме:

Естественно, можно использовать SAX и DOM для взаимодействия с AXIOM.

AXIOM использует “builder”, чтобы построить XML модель в памяти, но не всю модель, а только часть.

Рассмотрим пример использования AXIOM модели.

Пусть имеется XML фрагмент:

```
<?xml version="1.0" encoding="UTF-8"?>
<s:rottag xmlns:s="http://www.a.org/" >
<Employees>
  <Employee>
    <Name>Eran Chinthaka</Name>
    <Project>Axis2</Project>
    <WorkPlace>
     Ambalangoda, Sri Lanka
    </WorkPlace>
  </Employee>
  <Employee>
    <Name>Ajith Harshana</Name>
    <Project>Axis2</Project>
    <WorkPlace>Kuliyapitiya, Sri Lanka</WorkPlace>
 </Employee>
</Employees>
</s:rottag>
```

Предположим пользователь хочет получить проект первого служащего.

Таким образом, в памяти достаточно построить модель представляющую только первые четыре строки.

Остальные строки остаются нетронутыми в потоке.

Тогда, если понадобится проект второго сотрудника builder построит первые 9 строк.

Стоит заметит, что AXIOM архитектура не зависит от языка программирования.

Код использующий AXIOM будет иметь вид:

```
import org.apache.axiom.om.*;
import org.apache.axiom.om.impl.builder
                                        .StAXOMBuilder;
import java.io.*;
import javax.xml.stream.*;
import java.util.*;
public class Main {

   public static void main(String[] args) {
      try{
         FileReader soapFileReader = new FileReader("my.xml");
         XMLStreamReader parser =
                  XMLInputFactory.newInstance().createXMLStreamReader(

    soapFileReader);

        OMFactory omFactory = OMAbstractFactory.getOMFactory();
        StAXOMBuilder builder = new StAXOMBuilder(omFactory, parser);
         OMElement documentElement = builder.getDocumentElement();

       //получаем елементы дочерние к rottag
        OMElement childElement=documentElement.getFirstElement();
```

```
//Получаем дочерние элементы у первого Employees
OMElement employeesChild=childElement.getFirstElement();
int i=0;
for (Iterator iter = employeesChild.getChildElements();
                                                    iter.hasNext();) {

        OMNode child = (OMNode)iter.next();
         if(i==1){
           //На экране элемент <Project>Axis2</Project>
             child.serialize(System.out);
         }
         i++;
 }
```

Одним из наиболее интересных свойств AXIOM является его встроенная поддержка стандартов W3C XOP и MTOM, используемых в последних версиях приложений SOAP.

Эти два стандарта работают вместе:

оптимизированная двоичная компоновка XML - XML-binary Optimized Packaging (XOP) обеспечивается логическое включение в XML любых двоичных данных;

механизм оптимизации передачи сообщений MTOM (SOAP Message Transmission Optimization Mechanism) применяет технику XOP к сообщениям SOAP.

XOP и MTOM являются принципиальными характеристиками нового поколения инфраструктур Web-серверов, поскольку они обеспечивают поддержку интероперабельных приложений (т.е. взаимодействие приложений) и тем самым устраняют текущие проблемы в данной области.

XOP работает с данными, представленными в системе кодировки символов с основанием 64.

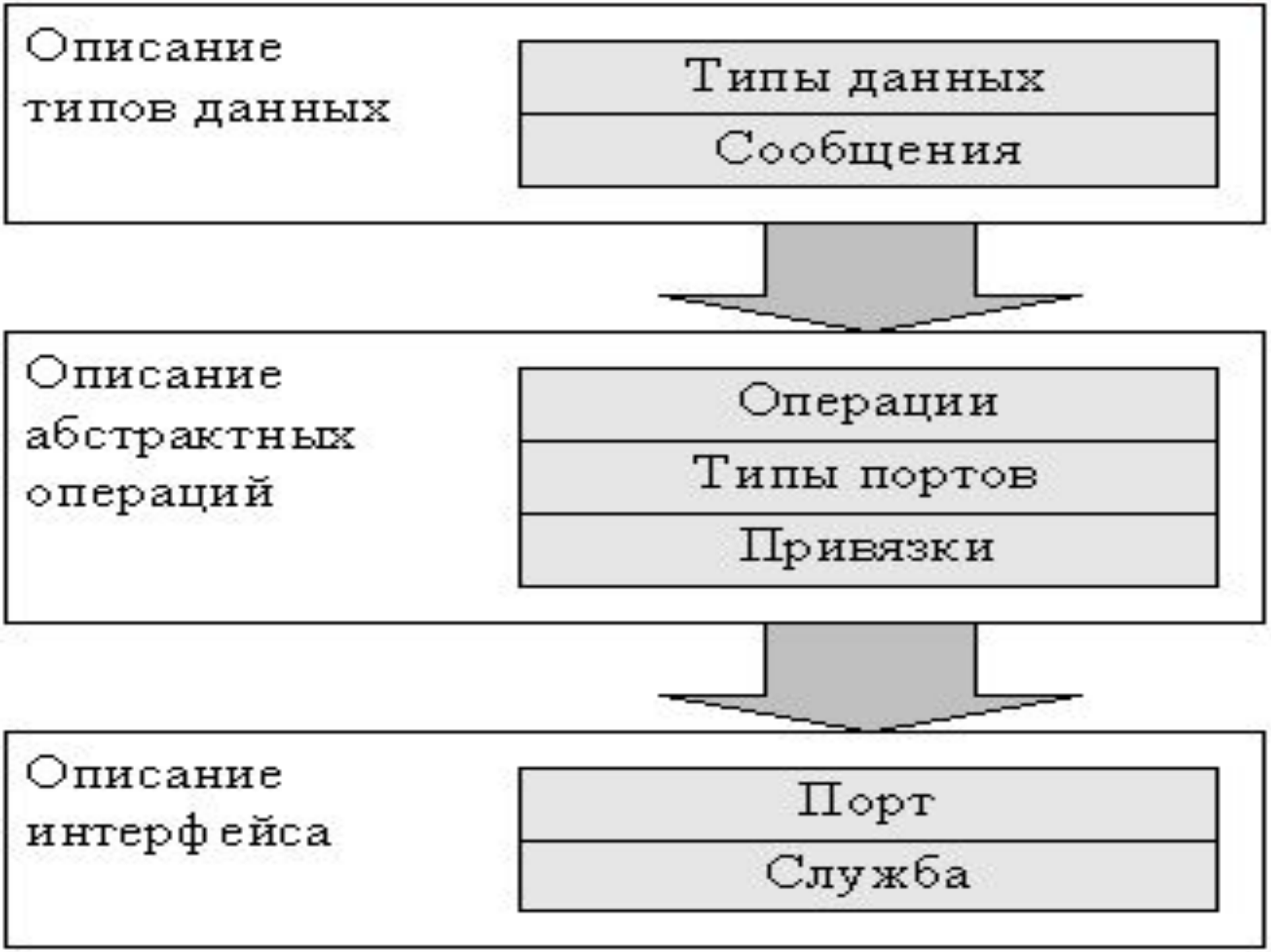
Base64 кодировка преобразует любые значения данных в ASCII символы, пригодные для печати, используя один символ ASCII для представления каждых 6 битов информации исходных данных.

Поскольку двоичные данные не могут быть размещены в XML (XML работает только с символами, а не непосредственно с байтами; даже использование номера кода символа недопустимо в XML), кодировка base64 незаменима для размещения двоичных данных в сообщениях XML.

Язык WSDL

Для описания интерфейса программной компоненты, включая спецификацию корректных сообщений, был разработан язык WSDL (Web Service Definition Language).

Описание на языке WSDL включает в себя следующие семь составляющих:


Описание типов данных
Типы данных
Сообщения
Описание абстрактных операций
Операции
Типы портов
Привязки
Описание интерфейса
Порт
Служба

- Описание типов передаваемых данных.
  При использовании кодирования SOAP Document оно состоит из схемы XML, определяющей корректные сообщения, получаемые программной компонентой в теле пакета SOAP.
- Описание входящих и исходящих сообщений, которые связываются с описанными типами данных.
- Описание операций (сервисов программной компоненты), с каждой из которых связывается входящее и исходящее сообщение.

- Описание типов портов (идентификаторов программных компонент), с каждым из которых связывается некоторый набор операций.
- Описание привязок (binding), связывающие типы портов и их сообщений с определенным типом кодирования тела пакета, а также с версией протокола SOAP.
- Описание портов, связывающие типы портов и соответствующие им привязки с конкретными URL.
- Общее описание службы (интерфейса программной компоненты) как совокупности портов.

Рассмотрим описание на языке WSDL интерфейса компоненты, которое содержит два сервиса – сложение двух чисел и сложение последовательности чисел.

В корневом элементе указаны все используемые пространства имен, включая пространство протокола SOAP 1.2

```
<?xml version="1.0" encoding="utf-8"?>
  <wsdl:definitions
 xmlns:tns="http://summa.test/webservices"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:soap12="http://schemas.xmlsoap.org/
                                    wsdl/soap12/"
 targetNamespace="http://summa.test/webservices"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

В элементе wsdl:types описываются все типы данных.

Тип Add будет связан со входящим сообщением сервиса сложения двух чисел, а тип AddResponse – с его исходящим сообщением.

```
<wsdl:types>
  <s:schema elementFormDefault="qualified"
     targetNamespace="http://summa.test/webservices">
    <s:element name="Add">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1"
              name="message" type="tns:AddMessage" />
        </s:sequence>
      </s:complexType>
    </s:element>
```

```
<s:complexType name="AddMessage">
  <s:sequence>
    <s:element minOccurs="1" maxOccurs="1"
                                                name="a" type="s:int" />
    <s:element minOccurs="1" maxOccurs="1"
                                                 name="b" type="s:int" />
  </s:sequence>
</s:complexType>

<s:element name="AddResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1"
                                        name="AddResult" type="s:int" />
    </s:sequence>
  </s:complexType>
</s:element>
```

Типы SumList и SumListResponse предназначены для сообщений сервиса сложения списка чисел.

```
<s:element name="SumList">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1"
                 name="list" type="tns:ArrayOfInt" />
    </s:sequence>
  </s:complexType>
</s:element>
```

```
<s:complexType name="ArrayOfInt">
 <s:sequence>
   <s:element minOccurs="0" maxOccurs="unbounded"
                                          name="int" type="s:int" />
 </s:sequence>
</s:complexType>
<s:element name="SumListResponse">
 <s:complexType>
   <s:sequence>
     <s:element minOccurs="1" maxOccurs="1"
                          name="SumListResult" type="s:int" />
   </s:sequence>
 </s:complexType>
</s:element>
</s:schema>
</wsdl:types>
```

В элементах wsdl:message типы данных связываются с идентификаторами сообщений.

```
<wsdl:message name="AddSoapIn">
  <wsdl:part name="parameters" element="tns:Add" />
</wsdl:message>
<wsdl:message name="AddSoapOut">
  <wsdl:part name="parameters"  element="tns:AddResponse" />
</wsdl:message>
<wsdl:message name="SumListSoapIn">
   <wsdl:part name="parameters"  element="tns:SumList" />
</wsdl:message>
<wsdl:message name="SumListSoapOut">
   <wsdl:part name="parameters"
                                  element="tns:SumListResponse" />
</wsdl:message>
```

В элементе wsdl:portType описываются абстрактные операции и используемые ими сообщения.

```
<wsdl:portType name="MathServiceSoap">
 <wsdl:operation name="Add">
  <wsdl:documentation
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
     Операция Add складывает два числа
  </wsdl:documentation>
 <wsdl:input message="tns:AddSoapIn" />
 <wsdl:output message="tns:AddSoapOut" />
</wsdl:operation>
```

```
<wsdl:operation name="SumList">
 <wsdl:documentation
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Операция SumList складывает несколько
    чисел
 </wsdl:documentation>
 <wsdl:input message="tns:SumListSoapIn" />
 <wsdl:output message="tns:SumListSoapOut" />
</wsdl:operation>
</wsdl:portType>
```

В элементе wsdl:binding операции связываются с транспортным протоколом (HTTP), версией протокола SOAP (1.2) и типом кодирования тела пакета (SOAP-Document).

```
<wsdl:binding name="MathServiceSoap12"
                        type="tns:MathServiceSoap">
 <soap12:binding
      transport="http://schemas.xmlsoap.org/soap/http" />
 <wsdl:operation name="Add">
 <soap12:operation
        soapAction="http://summa.test/webservices/Add"
                                        style="document" />
<wsdl:input>
```

Данный тег определяет как части сообщения будут располагаться в теле SOAP body, атрибут use оперделяет как сообщение буде кодироваться. Значение literal означает, что части сообщения определяются схемой типов.

```
   <soap12:body use="literal" />
</wsdl:input>
```

```
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="SumList">
    <soap12:operation
      soapAction="http://summa.test/webservices/SumList"
                                                  style="document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

В элементе wsdl:service интерфейс программной компоненты связывается с типом порта, с некоторой привязкой, а также с конкретным URL, используемым в дальнейшем для вызова веб службы.

```
<wsdl:service name="MathService">
  <wsdl:port name="MathServiceSoap12"
                        binding="tns:MathServiceSoap12">
    <soap12:address
                location="http://summa.test/webservices/
                                                summa.asmx" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Стоит заметить, что в настоящий момент существует два различных способа представления информации в теле пакета SOAP – кодирование SOAP RPC (в двух вариантах) и кодирование SOAP Document.

Кодирование SOAP RPC предназначено исключительно для передачи параметров удаленного вызова и определяет сообщение как имя метода и список параметров.

При использовании кодирования SOAP Document, которое является фактическим стандартом в настоящий момент, сообщение представляет собой XML документ со схемой и пространством имен, заданными в описании сервиса на языке WSDL.

Хотя обычно сообщение и состоит из имени метода удаленного объекта и списка его параметров, но сама спецификация кодирования не фиксирует как либо его содержание.

Получение кода из WSDL

В состав Axis2 входит инструмент WSDL2Java, служащий для формирования кода из описания сервиса WSDL.

Можно использовать этот инструмент напрямую, запуская класс org.apache.axis2.wsdl.WSDL2Java из приложения Java, или посредством задачи Ant, подключаемого модуля Maven или подключаемых модулей Eclipse или IDEA.

В инструменте WSDL2Java реализовано множество параметров командной строки, и их число постоянно растет.

В документации по Axis2 содержится полный список этих параметров, поэтому здесь мы рассмотрим только наиболее важные из их числа:

- **-o path —** Определяет директорию, в которой будут создаваться классы и файлы (по умолчанию используется рабочая директория)
- **-p package-name —** Определяет пакет, в который будут записываться формируемые классы (по умолчанию используется пространство имен WSDL)
- **-d name —** Определяет среду связывания данных (adb для ADB, xmlbeans для XMLBeans, none для отключения связывания данных; по умолчанию используется adb)
- **-uw —** Распаковывает упакованные документально-литеральные сообщения, только для поддерживаемых сред (на сегодняшний день это ADB)
- **-s —** Формирует только синхронный клиентский интерфейс
- **-ss —** Формирует серверный код
- **-sd —** Формирует файлы для установки на сервере
- **-uri path —** Задает путь к WSDL для формируемого сервиса

Формирование WSDL из кода

В состав Axis2 также входит инструмент Java2WSDL, который вы можете использовать для создания определения сервиса WSDL на основе существующего кода сервиса.

Однако полезность этого инструмента страдает от множества ограничений, в том числе невозможности работы с классами коллекций Java и отсутствия гибкости при структурировании XML, формируемого классами Java.

В Axis2 (начиная с версии 1.2) реализована полная поддержка нескольких вариантов связывания и готовится поддержка еще нескольких.

Рассмотрим некоторые из них.

Axis2 Data Binding

ADB (Связывание данных Axis2) - это расширение Axis2 для связывания данных.

В отличие от других сред связывания данных, код ADB можно использовать только вместе с Web-сервисами Axis2.

Это обстоятельство значительно ограничивает использование ADB, однако оно также даёт определенные преимущества.

Поскольку ADB интегрировано с Axis2, код может быть оптимизирован под требования Axis2

В WSDL2Java реализована полная поддержка формирования кода ADB, в том числе формирование классов модели данных, соответствующих компонентам схемы XML.

В базовом варианте формирования кода ADB используется прямая модель, в которой каждому входящему и исходящему сообщению, используемым в каждой операции, назначается отдельный класс.

Рассмотрим пример подобного связывания.

Пусть необходимо создать Web сервис, который производит сложение двух целых чисел.

1. Создадим абстрактный класс, в котором определена операция сложения

```
package hello;
 public abstract class Hello{
    public abstract Result add(Input in); }
```

2. Классы Result и Input имеют вид:

```
package hello;
  public class Input{
     private int a;
     private int b;
     public void setA(int a){  this.a=a;    }
     public void setB(int b){  this.b=b;     }
     public int getA(){ return a;  }
     public int getB(){ return b; } }
```

```
package hello;
public class Result{
  private int res;
  public void setRes(int r){ res=r;   }
  public int getRes(){  return res; } }
```

3. Данные классы Hello.java,Result.java и Input.java необходимо скомпилировать, поместив их в директорию hello

   javac hello/Hello.java

4. Необходимо сгенерировать описание сервиса WSDL(или написать вручную, тогда пункты 1-3 не нужны)

   java2wsdl -cn hello.Hello -cp . -sn Hello

Как результат получим следующий wsdl файл:

```
<wsdl:definitions ……..>
<wsdl:types>
  <xs:schema attributeFormDefault="qualified"
                                                    elementFormDefault="qualified"
                                                     targetNamespace="http://hello/xsd">
    <xs:complexType name="Input">
       <xs:sequence>
         <xs:element minOccurs="0" name="a"  type="xs:int"/>
         <xs:element minOccurs="0" name="b"  type="xs:int"/>
       </xs:sequence>
    </xs:complexTyp>
   <xs:complexType name="Result">
      <xs:sequence>
          <xs:element minOccurs="0" name="res" type="xs:int"/>
       </xs:sequence>
   </xs:complexType>
  </xs:schema>
```

```
<xs:schema xmlns:ax22="http://hello/xsd" attributeFormDefault="qualified"
            elementFormDefault="qualified"   targetNamespace="http://hello">
  <xs:import namespace="http://hello/xsd"/>
    <xs:element name="add">
        <xs:complexType>
           <xs:sequence>
            <xs:element minOccurs="0" name="args0" nillable="true"
                                                          type="ax21:Input"/>
         </xs:sequence>
      </xs:complexType>
     </xs:element>
    <xs:element name="addResponse">
      <xs:complexType>
        <xs:sequence>
         <xs:element minOccurs="0" name="return" nillable="true"
                                                        type="ax21:Result"/>
       </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
</wsdl:types>
```

```
<wsdl:message name="addRequest">
      <wsdl:part name="parameters" element="ns:add"/>
</wsdl:message>
<wsdl:message name="addResponse">
      <wsdl:part name="parameters"
                                  element="ns:addResponse"/>
</wsdl:message>
<wsdl:portType name="HelloPortType">
       <wsdl:operation name="add">
       <wsdl:input message="ns:addRequest"
                                        wsaw:Action="urn:add"/>
       <wsdl:output message="ns:addResponse"
                           wsaw:Action="urn:addResponse"/>
</wsdl:operation>
</wsdl:portType>
```

```
<wsdl:binding name="HelloSoap11Binding"
                                        type="ns:HelloPortType">
 <soap:binding
       transport="http://schemas.xmlsoap.org/soap/http"
                                              style="document"/>
  <wsdl:operation name="add">
       <soap:operation soapAction="urn:add"
                                              style="document"/>
       <wsdl:input>
            <soap:body use="literal"/>
       </wsdl:input>
        <wsdl:output>
             <soap:body use="literal"/>
         </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

```
<wsdl:binding name="HelloSoap12Binding"
                                        type="ns:HelloPortType">
  <soap12:binding
      transport="http://schemas.xmlsoap.org/soap/http"
                                              style="document"/>
   <wsdl:operation name="add">
      <soap12:operation soapAction="urn:add"
                                               style="document"/>
        <wsdl:input>
           <soap12:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
           <soap12:body use="literal"/>
        </wsdl:output>
   </wsdl:operation>
</wsdl:binding>
```

```
<wsdl:binding name="HelloHttpBinding"
                                        type="ns:HelloPortType">
  <http:binding verb="POST"/>
    <wsdl:operation name="add">
        <http:operation location="add"/>
         <wsdl:input>
           <mime:content type="text/xml"
                                          part="parameters"/>
         </wsdl:input>
         <wsdl:output>
            <mime:content type="text/xml"
                                          part="parameters"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

```
<wsdl:service name="Hello">
      <wsdl:port name="HelloHttpSoap11Endpoint"
                                        binding="ns:HelloSoap11Binding">
        <soap:address
              location="http://localhost:8080/axis2/services/Hello"/>
      </wsdl:port>
      <wsdl:port name="HelloHttpSoap12Endpoint"
                                        binding="ns:HelloSoap12Binding">
        <soap12:address
              location="http://localhost:8080/axis2/services/Hello"/>
       </wsdl:port>
       <wsdl:port name="HelloHttpEndpoint"
                                          binding="ns:HelloHttpBinding">
        <http:address
              location="http://localhost:8080/axis2/services/Hello"/>
       </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

5. Создаем скелет Web сервиса на основе wsdl файла Hello.wsdl, созданного на предыдущем шаге.

wsdl2java -uri Hello.wsdl -ss -sd

6. Логика Web сервиса должна быть заключена в классе **HelloSkeleton.java.**

Данный класс имеет вид:

```
package hello;
 public class HelloSkeleton{
    public hello.AddResponse add ( hello.Add add ) {
//Данный код был добавлен в ручную, это логика Web сервиса
            hello.AddResponse resp=new hello.AddResponse();
            hello.xsd.Input inp=add.getArgs0();
            hello.xsd.Result res=new hello.xsd.Result();
            res.setRes(inp.getA()+inp.getB());
            resp.set_return(res);
            return resp;
    }
}
```

7. Компилируем и создаем пакет **Hello.aar**, используя ant. build.xml создается автоматически.
8. Генерируем клиентский stub

wsdl2java -uri Hello.wsdl -o client

В результате в папке client появятся два файла

HelloStub.java

HelloCallbackHandler.java

9. Развертываем axis2 под Tomcat. Для этого в каталоге webappscоздаем поддиректорию axis2 и копируем туда директорию webapps в axis2. А затем копируем lib из axis2 в lib Tomcat
10. Запускаем Tomcat и заходим по ссылке

localhost:8086/axis2/axis2-admin/

Вводим

login: admin

password: axis2

11.Разворачиваем Web сервис, заходя по ссылке
http://localhost:8086/axis2/axis2-admin/upload
и выбирая файл Hello.aar.
12. Клиент, обращающийся к Web сервису будет иметь вид:

```
import hello.*;
    public class Main {
       public static void main(String[] args) {
             try {
                HelloStub stub = new HelloStub(null,
                            "http://localhost:8080/axis2/services/Hello");
                  HelloStub.Add add=new HelloStub.Add();
                  HelloStub.Input inp=new HelloStub.Input();
                  inp.setA(30);
                  inp.setB(20);
                  add.setArgs0(inp);
```

```
    HelloStub.AddResponse resp=stub.add(add);
     System.out.println(resp.get_return().getRes());
   } catch (org.apache.axis2.AxisFault e) {
      e.printStackTrace
   }catch(java.rmi.RemoteException e){
      e.printStackTrace();
   }
 }
}
```

После запуска на экране

поучим 50.

XMLBeans

XMLBeans - это общая среда обработки XML, в состав которой входит слой связывания данных.

Она создавалась как проект BEA Systems, а впоследствии была передана организации Apache Foundation.

XMLBeans была первой формой связывания данных, поддерживаемой Axis2, и продолжает быть наиболее популярным вариантом работы с Axis2, особенно со сложными определениями схем.

XMLBeans отличается от ADB тем, что в ней добавляется класс для документа, который содержит класс ввода или вывода.

Совокупный эффект при использовании XMLBeans по сравнению с ADB состоит в добавлении уровня создания объектов.

В Axis2 нет поддержки распаковки XMLBeans, поэтому и нет способа избежать этого дополнительного уровня объектов.

В результате классы, сформированные XMLBeans, получаются более сложными для использования, чем их эквиваленты в других средах связывания данных.

Рассмотрим тот же пример с сложением двух целых чисел, что и в случае ADB.

Шаги 1-4 остаются теми же самыми, что и в случае ADB.

Шаг 5. Создаем скелет Web сервиса на основе wsdl файла Hello.wsdl, созданного на предыдущем шаге.

wsdl2java -uri Hello.wsdl -ss -sd -d xmlbeans

Шаг 6. Логика Web сервиса должна быть заключена в классе **HelloSkeleton.java.**

Данный класс имеет вид:

```
package hello;
    public class HelloSkeleton{
        public hello.AddResponseDocument add(
                                                    hello.AddDocument add ) {
            hello.AddDocument.Add addd=add.getAdd();
            hello.xsd.Input inp=addd.getArgs0();
            int sum=inp.getA()+inp.getB();
            hello.xsd.Result res=hello.xsd.Result.Factory.newInstance();
            res.setRes(sum);

            hello.AddResponseDocument  respDoc=
                        hello.AddResponseDocument.Factory.newInstance();
            hello.AddResponseDocument.AddResponse addResp =
    hello.AddResponseDocument.AddResponse.Factory.newInstance();
            addResp.setReturn(res);
            respDoc.setAddResponse(addResp);
            return respDoc;
        }
      }
```

Шаг 7. Компилируем и создаем пакет **Hello.aar**, используя ant. build.xml создается автоматически.

Шаг 8. Генерируем клиентский stub

wsdl2java -uri Hello.wsdl -d xmlbeans –o client

В результате в папке client появятся папки

resources

src

с набором java файлов и xsd файлов.

Шаг 9-11 аналогичны ADB.

Шаг 12. Клиент, обращающийся к Web сервису будет иметь вид:

```
public static void main(String[] args) {
        try {
            HelloStub stub =
    new  HelloStub(null,"http://localhost:8086/axis2/services/Hello");
            hello.AddDocument adds=
                            hello.AddDocument.Factory.newInstance();
            hello.xsd.Input inp=hello.xsd.Input.Factory.newInstance();
            inp.setA(20);
            inp.setB(30);
            hello.AddDocument.Add addd=
                        hello.AddDocument.Add.Factory.newInstance();
            addd.setArgs0(inp);
            adds.setAdd(addd);
```

```
 hello.AddResponseDocument resp= stub.add(adds);
     System.out.println(resp.getAddResponse()
                                    .getReturn().getRes());
         }catch (org.apache.axis2.AxisFault e) {
                 e.printStackTrace();
          }catch(java.rmi.RemoteException e){
                 e.printStackTrace();
          }
}
```

На экране получим 50.

Стоит отметить, что клиент не запустится без папки **schemaorg_apache_xmlbeans,** находящейся в директории resources(в папке client).

Перед запуском клиента необходимо прописать путь к этой папке, либо скопировать ее к другим пакетам клиентского приложения в соответствующей среде разработки.

Restful services

Архитектура REST отличается своей простотой, требуя от приложений обеспечить только возможность приема сообщений с HTTP-заголовками.

Эта функция легко реализуется простыми Web-контейнерами для Java-приложений.

REST-приложения часто создаются на основе сервлетов. Сервлеты не предписывают какие-либо конкретные походы к разработке.

Как правило, сервлеты получают на обработку запросы, анализируют их заголовки, в том числе URI, чтобы определить, к какому ресурсу выполняется обращение.

Стандарт Rest описан в документе JSR-311.

Также спецификация JAX-RS 1.0, описывающая подход к созданию REST-сервисов на основе аннотаций.

В отличие от модели на основе сервлетов, аннотации JAX-RS позволяют разработчикам сосредоточиться на прикладных ресурсах и данных, не отвлекаясь на вопросы, связанные с обменом информацией

JAX-RS задает унифицированный способ описания ресурсов на основе своей модели программирования.

Он включает пять основных компонентов:

корневые ресурсы

дочерние ресурсы

методы ресурсов

методы дочерних ресурсов

локаторы дочерних ресурсов

Корневые ресурсы

Корневыми ресурсами являются Java-классы, отмеченные аннотацией @Path.

Эта аннотация включает атрибут value, задающий путь к ресурсу.

Его значением могут быть строка символов, переменные, а также переменные в сочетании с регулярным выражением

Пример корневого ресурса в JAX-RS

```
import javax.ws.rs.Path;
@Path(value="/contacts")
public class ContactsResource { ... }
```

Дочерние ресурсы

Дочерними ресурсами (subresources) являются Java-классы, полученные в результате вызова локатора.

Они аналогичны корневым ресурсам за тем исключением, что они не помечаются аннотацией @Path, поскольку путь к ним описывается в локаторе.

Дочерние ресурсы, как правило, содержат методы с аннотациями, задающими тип обрабатываемых HTTP-запросов.

Если они не содержат таких методов, то делегируют обработку запросов подходящему локатору дочерних ресурсов.

Пример дочернего ресурса в JAX-RS

```
import javax.ws.rs.GET;
public class Department {
 @GET
 public String getDepartmentName() { ... }
}
```

Методы ресурсов

Методами ресурсов называются методы Java-классов, представляющих собой корневые или дочерние ресурсы.

Эти методы привязаны к типам HTTP-запросов при помощи аннотаций

Пример метода ресурса в JAX-RS

```
import java.util.List;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
@Path(value="/contacts")
public class ContactsResource {
    @GET public List<ContactInfo> getContacts() { ... }
}
```

Методы дочерних ресурсов

Методы дочерних ресурсов аналогичны методам ресурсов за тем исключением, что они дополнительно отмечены аннотацией @Path, уточняющей, в каких случаях их следует вызывать.

Пример метода дочернего ресурса в JAX-RS

```
import java.util.List;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
@Path(value="/contacts")
public class ContactsResource {
   @GET
    public List<ContactInfo> getContacts() { ... }
  //дочерний ресурс
   @GET
   @Path(value="/ids")
   public List<String> getContactIds() { ... }
}
```

Локаторы дочерних ресурсов

Локаторы дочерних ресурсов представляют собой методы, служащие для уточнения того, какой ресурс должен обрабатывать входящий запрос.

Подобно методам дочерних ресурсов они отмечаются аннотацией @Path, однако в отличие от тех, они не привязаны к типам HTTP-запросов, в частности, у них аннотации @GET.

Пример локатора дочернего ресурса в JAX-RS

```
@Path(value="/contacts")
public class ContactsResource {
@GET
public List<ContactInfo> getContactss() { ... } @GET
@Path(value="/ids")
public List<String> getContactIds() { ... }
//локатор дочернего ресурса
@Path(value="/contact/{contactName}/department")
public Department getContactDepartment(
    @PathParam(value="contactName") String contactName) { ... }
}
```

Любой HTTP-запрос, отправленный по адресу /contact/{contactName}/department, будет обработан локатором getContactDepartment.

Фрагмент {contactName} относительного пути запроса означает, что вслед за префиксом contact может следовать любая последовательность символов, удовлетворяющая правилам URL.

Аннотации

Рассмотрим основные аннотации и варианты их применения.

Аннотация @Path

Аннотация @Path используется для описания пути к корневому ресурсу, дочернему ресурсу или описания метода дочернего ресурса.

Атрибут value может содержать символы, простые переменные, а также переменные, включающие регулярные выражения.

```
@Path(value="/contacts")
public class ContactsResource {
  @GET @Path(value="/{emailAddress:.+@.+\\.[a-z]+}")
   public ContactInfo getByEmailAddress(
      @PathParam(value="emailAddress") String emailAddress) { ... }
  @GET @Path(value="/{lastName}")
   public ContactInfo getByLastName(
         @PathParam(value="lastName") String lastName) { ... }
}
```

Аннотации @GET, @POST, @PUT, @DELETE, @HEAD

Аннотации @GET, @POST, @PUT, @DELETE и @HEAD соответствуют типам HTTP-запросов.

Их можно использовать для привязки методов корневых и дочерних ресурсов к запросам соответствующих типов.

Запросы типа GET передаются на обработку методам, аннотированным @GET, запросы типа @POST – методам с аннотацией @POST и т.д.

При этом существует возможность определения дополнительных аннотаций для обработки нестандартных HTTP-запросов.

Для этого служит аннотация @HttpMethod.

Объявление новой аннотации, соответствующей HTTP-запросам типа GET

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy; import
java.lang.annotation.Target;
import javax.ws.rs.HttpMethod;
  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.METHOD)
  @HttpMethod("GET")
  public @interface CustomGET { }
```

Аннотации @Consumes и @Produces

Аннотация @Consumes задает типы содержимого MIME, принимаемые ресурсом, а @Produces – типы MIME, возвращаемые ресурсом.

Этими аннотациями могут отмечаться ресурсы, дочерние ресурсы, методы ресурсов и дочерних ресурсов, а также локаторы дочерних ресурсов.

Примеры использования аннотаций @Consumes/@Produces

```
@Path(value="/contacts")
public class ContactsResource {
  @GET
   @Path(value="/{emailAddress:.+@.+\\.[a-z]+}")
   @Produces(value={"text/xml", "application/json"})
   public ContactInfo getByEmailAddress(
      @PathParam(value="emailAddress") String emailAddress) { ... }
   @GET
   @Path(value="/{lastName}")
   @Produces(value="text/xml")
    public ContactInfo getByLastName(
       @PathParam(value="lastName") String lastName) { ... }
    @POST
    @Consumes(value={"text/xml", "application/json"})
     public void addContactInfo(ContactInfo contactInfo) { ... }
}
```

Методы **getByEmailAddress** и **addContactInfo**, могут обрабатывать запросы с типом содержимого text/xml и application/json.

Представление входящего или возвращаемого ресурса зависит от параметров в заголовке HTTP-запроса, устанавливаемых клиентом.

Значение аннотации @Consumes сверяется с параметром Content-Type для того, чтобы определить, способен ли метод обработать содержимое полученного запроса.

Провайдеры

Провайдерами в JAX-RS называются компоненты, определяющие три ключевых аспекта в поведении приложения: связывание с данными, сопоставление исключений и разрешение контекста (примером последнего может служить предоставление экземпляров JAXBContext среде выполнения).

Каждый провайдер JAX-RS должен быть отмечен аннотацией @Provider.

Ниже приведен пример провайдеров MessageBodyWriter и MessageBodyReader, реализующих операции связывания с данными.

Провайдер MessageBodyWriter

Экземпляры этого класса используются средой выполнения JAX-RS для сериализации представления возвращаемого клиенту ресурса.

Реализации сред выполнения, удовлетворяющие JSR-311, должны самостоятельно поддерживать общеупотребительные типы данных, в том числе java.lang.String, java.io.InputStream, объекты JAXB и т.д., однако пользователь может указать среде, что следует использовать предоставленный ей провайдер (класс-наследник MessageBodyWriter).

```
@Provider
@Produces("text/xml")
public class ContactInfoWriter implements MessageBodyWriter<ContactInfo> {
   public long getSize(java.lang.Class<ContactInfo> type,
                       java.lang.reflect.Type genericType,
                       java.lang.annotation.Annotation[] annotations,
                       MediaType mediaType) { ... }
   public boolean isWriteable(java.lang.Class<ContactInfo> type,
                        java.lang.reflect.Type genericType,
                        java.lang.annotation.Annotation[] annotations,
                        MediaType mediaType) { return true; }
    public void writeTo(ContactInfo contactInfo,
                        java.lang.Class<ContactInfo> type,
                        java.lang.reflect.Type genericType,
                        java.lang.annotation.Annotation[] annotations,
                        MediaType mediaType,
                        MultivaluedMap< java.lang.String,
                        java.lang.Object> httpHeaders,
                        java.io.OutputStream entityStream) {
                   contactInfo.serialize(entityStream);
   }}
```

Класс ContactInfoWriter будет вызываться средой JAX-RS перед сериализацией возвращаемых клиенту ресурсов.

Если метод isWriteable провайдера возвращает true, а значение его аннотации @Produces наиболее точно соответствует значению той же аннотации метода ресурса, то будет вызван метод writeTo.

В этом случае класс ContactInfoWriter будет отвечать за сериализацию содержимого экземпляра ContactInfo в нижележащий поток вывода (OutputStream).

Провайдер MessageBodyReader

Провайдеры MessageBodyReaders выполняют функцию, обратную по отношению к MessageBodyWriter.

Реализации JAX-RS самостоятельно поддерживают восстановление из потока ввода (эта операция называется десериализацией) экземпляров тех же типов, что и для сериализации.

При этом пользователи могут предоставлять среде собственные классы-наследники MessageBodyReader.

Основная функция такого провайдера заключается в чтении данных из потока ввода (InputStream) и восстановлении содержимого Java-объектов, которые ожидает получить на вход метод ресурса, из неструктурированного набора байтов.

```
@Provider
@Consumes("text/xml")
  public class ContactInfoReader implements
                                        MessageBodyReader<ContactInfo> {
   public boolean isReadable(java.lang.Class<ContactInfo> type,
                             java.lang.reflect.Type genericType,
                             java.lang.annotation.Annotation[] annotations,
                             MediaType mediaType) { return true; }

   public ContactInfo readFrom(java.lang.Class<ContactInfo> type,
         java.lang.reflect.Type genericType,
         java.lang.annotation.Annotation[] annotations,
         MediaType mediaType,
         MultivaluedMap< java.lang.String,java.lang.String>httpHeaders,
          java.io.InputStream entityStream) {
            return ContactInfo.parse(entityStream);
    }
}
```

Подобно методу MessageBodyWriter.isWriteable, метод isReadable класса ContactInfoReader будет вызываться средой JAX-RS чтобы определить, способен ли провайдер десериализовать поступающие на вход данные.

Если этот метод возвращает true, а значение аннотации @Consumes наиболее точно соответствует значению той же аннотации метода ресурса, то провайдер будет выбран для десериализации.

Метод readFrom должен возвращать экземпляры класса ContactInfo, содержимое которых было восстановлено из потока ввода (InputStream).

Классы, описывающие конфигурацию

Выше были рассмотрены классы ресурсов JAX-RS, а также некоторые из классов-провайдеров (MessageBodyReader и MessageBodyWriter).

Далее мы перейдем к вопросам конфигурирования этих классов внутри среды выполнения JAX-RS, которое производится при помощи расширения класса javax.ws.rs.core.Application.

Этот класс предоставляет список классов (или объектов-синглетонов), включающих набор всех корневых ресурсов и провайдеров приложения JAX-RS.

```
public class ContactInfoApplicaiton extends Application {
   public Set<Class<?>> getClasses() {
     Set<Class<?>> classes = new HashSetSet<Class<?>>();
     classes.add(ContactsResource.class);
     classes.add(ContactInfoWriter.class);
     classes.add(ContactInfoReader.class);
    }
    public SetSet<Object<?>> getSingletons() {
       // Пустой метод, поскольку синглетоны не
           используются
     }
}
```

Среда выполнения JAX-RS вызывает метод getClasses для получения списка классов, предоставляющих все необходимые метаданные.

Применение синглетонов для ресурсов JAX-RS должно тщательно продумываться и в общем случае не рекомендуется.

Рассмотрим пример простейшего Rest сервиса(используется jersey 2.0)

```
package ua.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/service/rest")
public class RestService {
    @GET
   @Produces(MediaType.TEXT_HTML)
   public String getTitle(){
      return "<p>Hello,Alex</p>";
   }
}
```

Web.xml для этого сервиса имеет вид

```
<web-app>
   <display-name>FirstRestFulApplication</display-name>
   <servlet>
      <servlet-name>Rest</servlet-name>
      <servlet-class>org.glassfish.jersey.servlet.ServletContainer
                                                          </servlet-class>
      <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>ua.rest</param-value>
                                  // где искать классы с сервисами
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>Rest</servlet-name>
      <url-pattern>/api/*</url-pattern>
   </servlet-mapping>
</web-app>
```

WebSockets

Метод WebSockets, который появился в HTML5.

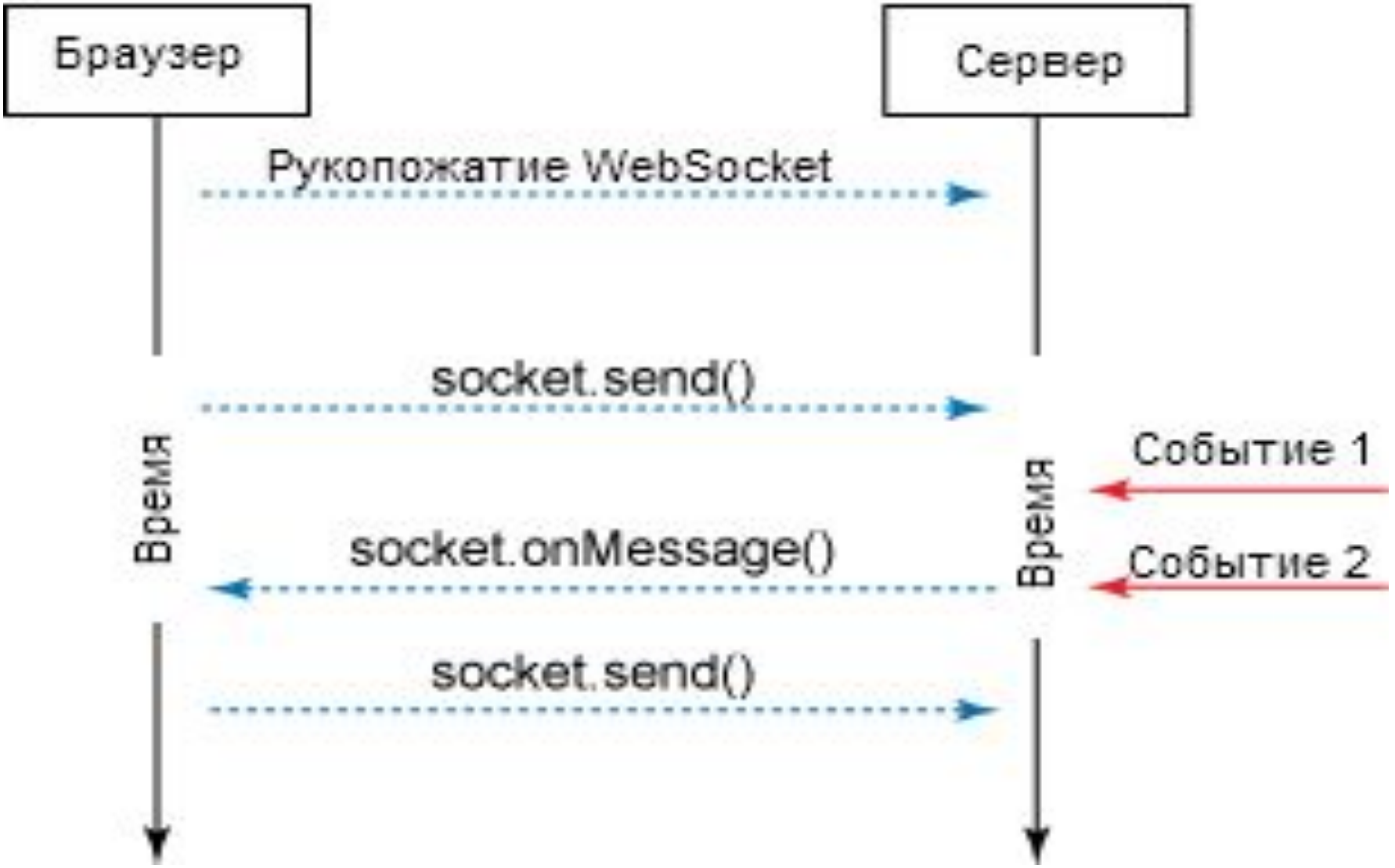
WebSockets создает двунаправленные, дуплексные каналы связи.

Соединение открывается посредством HTTP-запроса со специальными заголовками, который называется рукопожатием WebSockets.

Это соединение сохраняется постоянно, и через него можно записывать и получать данные посредством JavaScript, как при использовании стандартного сокета TCP.

URL WebSocket начинается с ws:// или wss:// (по SSL).

Временная диаграмма на иллюстрирует связь с использованием WebSockets



JavaScript позволяет использовать WebSockets, если эту возможность поддерживает браузер.

```
var ws = new WebSocket('ws://127.0.0.1:8080/async');
ws.onopen = function() {
  // вызывается при открытии соединения
 };
ws.onerror = function(e) {
  // вызывается в случае ошибки, например, при обрыве связи
 };
ws.onclose = function() {
  // вызывается при закрытии соединения
};
ws.onmessage = function(msg) {
 // вызывается, когда сервер посылает сообщение клиенту.
 // сообщение содержится в msg.data.
};
ws.send('some data');
ws.close();
```

Отправляемые и получаемые данные могут быть любого типа. WebSockets можно рассматривать как TCP сокеты, так что решение о том, какой тип данных использовать, остается за клиентом и сервером.

При создании объекта JavaScript WebSocket в HTTP-запросах в консоли браузера видны заголовки рукопожатия WebSocket.

Request URL:ws://127.0.0.1:8080/async

Request Method:GET

Status Code:101 WebSocket Protocol Handshake

Request Headers

Connection:Upgrade

Host:127.0.0.1:8080

Origin:http://localhost:8080

Sec-WebSocket-Key1:1 &1~ 33188Yd]r8dp W75q

Sec-WebSocket-Key2:1 7; 229 *043M 8

Upgrade:WebSocket

(Key3):B4:BB:20:37:45:3F:BC:C7

Response Headers

Connection:Upgrade

Sec-WebSocket-Location:ws://127.0.0.1:8080/async

Sec-WebSocket-Origin:http://localhost:8080

Upgrade:WebSocket

(Challenge

Response):AC:23:A5:7E:5D:E5:04:6A:B5:F8:CC:E7:AB:6D:1A:39

Все заголовки используются процессом рукопожатия WebSocket для установки и настройки долгоживущего соединения.

Объект WebSocket JavaScript также содержит два полезных свойства:

ws.url

Возвращает URL сервера WebSocket.

ws.readyState

Возвращает значение текущего состояния соединения:

CONNECTING = 0

OPEN = 1

CLOSED = 2

На стороне сервера получим(используются WebSockets Tomcat):

```
@ServerEndpoint("/ws")
public class WebSocket {
    @OnOpen
    public void onOpen(Session session){
        System.out.println(session.getId() + " has opened a connection");
        try {
            session.getBasicRemote().sendText("Connection Established");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```

```
@OnMessage
    public void onMessage(String message, Session session, boolean
last){
        System.out.println("Message from " + session.getId() + ": " +
message);
        try {
            session.getBasicRemote().sendText(message);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

@OnClose
    public void onClose(Session session){
        System.out.println("Session " +session.getId()+" has ended");
    }
}
```

Клиент может иметь следующий вид:

```
<html>
   <head>
<script type="text/javascript">
        var socket = new WebSocket("ws://localhost:8084/WebApplication13/ws");
        socket.onopen = function() {
           alert("Соединение установлено.");
        };
         socket.onclose = function(event) {
           if (event.wasClean) {
              alert('Соединение закрыто');
           } else {
              alert('Обрыв соединения');
           }
           alert('Код: ' + event.code + ' причина: ' + event.reason);
        };
        socket.onmessage = function(event) {
           var logarea = document.getElementByld("log");
           logarea.value = event.data+"n"+logarea.value;
        };
```

```
    socket.onerror = function(error) {
        alert("Ошибка " + error.message);
      };

      function send() {
        var s = document.getElementById("in").value;
        socket.send(s);
      };

    </script>
  </head>
  <body>
    <form>
      <input type="text" id="in" />
      <input type="button" onclick="send()" value="send" />
      <textarea id="log" rows="8" cols="20"></textarea>
    </form>
  </body>
</html>
```

Стоит заметить, что клиентом не обязательно может выступать Web browser. Клиентом может біть обічное java приложение.

```
@ClientEndpoint
public class JavaApplication103 {
 public static void main(String[] args) throws URISyntaxException,
                              DeploymentException, IOException, InterruptedException {
   WebSocketContainer wsContainer =
                                                    ContainerProvider.getWebSocketContainer();
    Session session = wsContainer.connectToServer(JavaApplication103.class,
                                    new URI("ws://localhost:8084/WebApplication13/ws"));
        session.getBasicRemote().sendText("Here is a message!");
        Thread.sleep(1000);
        session.close();
    }
    @OnMessage
    public void processMessage(String message){
        System.out.println(message);
    }

}
```