МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра информационных систем и технологий

УТВЕРЖДАЮ
Проректор по учебной работе
_____ О. Г. Локтионова
«____»_____ 2013 г.

Программирование Web-сервисов на языке Java

Методические указания по выполнению лабораторной работы по курсу «Интернет-технологии» для студентов, обучающихся по направлению подготовки 230400.62 «Информационные системы и технологии»

Курск 2013

УДК 004.42, 004.75

Составители Е. А. Титенко, М. В. Бородин

Рецензент
Доктор технических наук, профессор О. И. Атакищев

Программирование Web-сервисов на языке Java : методические указания по выполнению лабораторной работы / Юго-Зап. гос. ун-т; сост. Е. А. Титенко, М. В. Бородин. Курск, 2013. 21 с., библиогр.: 3.

В методических указаниях описывается технология разработки Web-сервисов на языке Java. Описание сопровождается примерами. Приводится список контрольных вопросов и вариантов заданий. Предназначены для студентов направления подготовки 230400.62.

Текст печатается в авторской редакции

Подписано в печать . Формат 60 × 84 1/16.
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз. Заказ . Бесплатно.
Юго-Западный государственный университет.
305040, г. Курск, ул. 50 лет Октября, 94

1. Цель

Целью настоящей лабораторной работы является освоение студентами навыков разработки Web-сервисов на языке Java.

2. Задание

В соответствии с индивидуальным вариантом задания, полученным от преподавателя, требуется:

- определить интерфейс, реализуемый сервисом;
- реализовать сервис на языке программирования Java;
- реализовать клиентское приложение (для тестирования сервиса) на любом языке программирования.

3. Содержание отчета

Отчет о выполнении работы должен включать в себя:

- титульный лист;
- вариант задания;
- описание интерфейса, реализуемого сервисом;
- исходный текст сервиса;
- исходный текст клиентской программы (для тестирования);
- тесты и описание процедуры тестирования.

4. Теоретические сведения

Web-сервис — это серверный компонент, который обменивается со своими клиентами сообщениями в формате XML, используя для их передачи стандартные коммуникационные протоколы (обычно HTTP). Отказ от использования собственных коммуникационных протоколов позволяет использовать Web-сервисы в глобальных сетях (что выгодно отличает их от таких технологий как CORBA или DCOM).

Для обеспечения независимости от языка программирования каждый Web-сервис снабжается описанием на языке WSDL (Web Service Description Language), являющемся диалектом XML. Язык WSDL позволяет описывать следующие объекты:

- сервис (service) содержит один или несколько портов (port), каждый из которых задается привязкой порта (binding);

• каждая привязка порта определяется типом порта (port type) с некоторым набором параметров, задающим соглашения, используемые при передаче данных;

• каждый тип порта содержит описание одной или нескольких операций (operation);

• каждая операция состоит в передаче одного или двух сообщений (message), причем, если сообщений два, то одно из сообщений должно быть входящим, а другое исходящим. Также в описании операции могут быть перечислены сообщения об ошибках;

• каждое сообщение может состоять из одной или нескольких частей (part);

• каждая часть характеризуется некоторым типом данных.

Каждый из вышеперечисленных объектов имеет имя, причем для предотвращения конфликтов имен используются пространства имен, устроенные аналогично пространствам имен XML.

Язык WSDL используется только для описания Web-сервисов. При обмене сообщениями между Web-сервисом и клиентом он не используется. Собственно сообщения представляет собой произвольные данные в формате XML, отражающие соответствующую предметную область.

Приведем пример простейшего Web-сервиса:

```
// на сервере
import javax.jws.*;
@WebService(targetNamespace = "http://samples/hello")
public class Hello {
  public String getHello(String name) {
    return String.format("Hello, %s!", name);
  }
}
```

В простейшем случае Web-сервис представляет собой обычный класс, аннотированный WebService, причем параметр targetNamespace задает пространство имен (параметр targetNamespace является необязательным; по умолчанию используется имя пакета, в котором находится Web-сервис). Для Web-сервиса автоматически создается описание на языке WSDL, доступное по адресу http://localhost/hello?wsdl[1] и включающее в дан-

[1] Мы не будем приводить здесь это описание, поскольку оно весьма объемно.

ном случае описание единственного сервиса HelloService, содержащего единственный порт HelloPort, который в свою очередь содержит единственную операцию getHello. По умолчанию имя сервиса и имя порта определяются именем класса, а множество операций и имя каждой из них — множеством общедоступных нестатических методов. Если некоторый метод определяет операцию, то типы данных его параметров, а также тип данных значения, возвращаемого из него, может быть:

- примитивным (т. е. int, double и т. д.);
- оберткой (т. е. Integer, Double и т. д.);
- строкой (String);
- датой (Date) или календарем (Calendar);
- любым классом, аннотированным XmlRootElement;
- массивом, коллекцией, списком или множеством элементов вышеперечисленных типов (массив байтов обрабатывается особым образом; коллекции, списки и множества не могут быть вложенными; массивы вложенными быть могут, однако делать так не рекомендуется по соображениям переносимости).

Предоставить клиентам доступ к Web-сервису можно следующим образом:

```
// на сервере
Endpoint.publish("http://localhost/hello", new Hello());
```

Чтобы клиент мог обратиться к Web-сервису, сначала необходимо определить интерфейс, представляющий операции Web-сервиса:

```
// на клиенте
@WebService(targetNamespace = "http://samples/hello")
public interface Hello {
  String getHello(String name);
}
```

В простейшем случае требуется точное совпадение, как имени интерфейса, так и сигнатур операций (благодаря тому, что пространство имен указано явно, совпадение имени пакета не требуется). Заметим, что описывать все операции необязательно.

Располагая интерфейсом, представляющим операции, обратиться к Web-сервису можно так:

```
// на клиенте
```

```
Service helloService = Service.create(
  new URL("http://localhost/hello?wsdl"),
    new QName("http://samples/hello", "HelloService"));
Hello helloPort = helloService.getPort(Hello.class);
String hello = helloPort.getHello("World");
System.out.println(hello);
```

Обращения к Web-сервису начинается с создания ссылки на него (представленной классом Service); для этого используется адрес описания Web-сервиса, а также квалифицированное (т. е. включающее пространство имен) имя сервиса. Имея ссылку на сервис, можно получить ссылку на порт, для чего достаточно указать описывающий его интерфейс (ссылка на порт представлена proxy-классом). Используя ссылку на порт, можно вызывать операции, так как если бы они были обычными методами.

По умолчанию сообщения передаются по протоколу HTTP, причем при передаче они упаковываются в конверты SOAP (Simple Object Access Protocol). Так в примере входящее сообщение, представленное элементом getHello, в конверте выглядит так:

```
<!-- от клиента к серверу -->
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:getHello xmlns:ns2="http://samples/hello">
   <arg0>World</arg0>
  </ns2:getHello>
 </S:Body>
</S:Envelope>
```

Совершенно аналогично исходящее сообщение, представленное элементов getHelloResponse, в конверте выглядит так:

```
<!-- от сервера клиенту -->
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Header/>
 <S:Body>
  <ns2:getHelloResponse xmlns:ns2="http://samples/hello">
   <return>Hello, World!</return>
  </ns2:getHelloResponse>
 </S:Body>
</S:Envelope>
```

Как видно, конверт (Envelope) состоит из необязательного заголовка (Header) и обязательного тела (Body). Основное назначение кон-

верта — передача вместе с самим сообщением служебных данных, помещаемых в заголовок.

Элементы getHello и getHelloResponse определены не непосредственно в описании Web-сервиса, но в прилагающейся к нему схеме данных в формате XSD (XML Schema Definition). Эта схема в данном примере доступна по адресу http://localhost/hello?xsd=1.

Поскольку в конверте не упоминается в явном виде ни имя порта, ни имя операции, при получении входящего сообщения Web-сервис выбирает выполняемую операцию только на основании тега элемента, содержащегося в теле конверта (т. е. фактически все, что надо знать для вызова Web-сервиса — это тип принимаемых им данных).[2]

Есть возможность явно задать имена элементов, используемых во входящем и исходящем сообщении. Например:

```
@RequestWrapper(localName = "hello-rq")
@ResponseWrapper(localName = "hello-rs")
@WebResult(name = "hello")
String getHello(@WebParam(name = "name") String name);
```

Аннотации RequestWrapper и ResponseWrapper задают имена элементов, представляющих соответственно входное и выходное сообщение. Кроме параметра localName можно указать пространство имен (параметр targetNamespace), а также имя автоматически генерируемого Java-класса (параметр className).

Аннотации WebResult и WebParam задают имена элементов, представляющих соответственно возвращаемое значение и параметры метода.

С учетом перечисленных аннотаций входящее сообщение в конверте будет выглядеть так:

```
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:hello-rq xmlns:ns2="http://samples/hello">
```

[2] Здесь можно проследить известную аналогию с языками программирования — массивы, структуры, функции и подпрограммы существуют лишь в исходном коде: после компиляции остается лишь последовательность байтов; точно так же порты и операции существуют лишь в описании Web-сервиса: в самих данных, которыми обменивается Web-сервис, никакого упоминания о чем-либо подобном нет.

```
   <name>World</name>
  </ns2:hello-rq>
 </S:Body>
</S:Envelope>
```

а исходящее так:

```
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Header/>
 <S:Body>
  <ns2:hello-rs xmlns:ns2="http://samples/hello">
   <hello>Hello, World!</hello>
  </ns2:hello-rs>
 </S:Body>
</S:Envelope>
```

Рассмотрим более сложный пример. Начнем с определения интерфейса сервиса

```
// на сервере и на клиенте
@WebService
public interface RecordManager {
  Record[] takeSnapshot(Date start, Date end,
      int maximumRecordCount,
      @WebParam(mode = WebParam.Mode.OUT)
      Holder<Integer> totalRecordCount,
      @WebParam(mode = WebParam.Mode.OUT)
      Holder<Integer> snapshotId);
  Record[] getSnapshotData(Integer snapshotId,
      int recordIndex, int maximumRecordCount)
      throws InvalidRecordSnapshotIdException;
  @Oneway
  void freeSnapshot(Integer snapshotId);
}
```

Хорошим стилем программирования является отделение интерфейса (контракта) от реализации.

Аннотация WebParam позволяет, в частности, задать вид параметра; возможные варианты: IN — входной параметр (по умолчанию), OUT — выходной параметр, и INOUT — входной и выходной параметр одновременно. Входные параметры включаются во входящее сообщение, а выходные — в исходящее. Если параметр является только входным, то его типом должен быть Holder.[3]

[3] Holder эмулирует передачу параметра по ссылке.

Аннотация Oneway показывает, что данная операция не предусматривает исходящего сообщения. На клиенте возврат из соответствующего метода произойдет сразу после отправки сообщения сервису. Из-за отсутствия сообщения от сервиса узнать, выполнилась ли соответствующая операция успешно невозможно. Разумеется, метод, аннотированный Oneway, не может возвращать значения и не может иметь параметров, кроме входных параметров.

```
// не сервере
@WebService(
  endpointInterface = "samples.soap.records.RecordManager",
  targetNamespace = "http://samples/records",
  serviceName = "RecordManager")
public class RecordManagerImpl implements RecordManager {
  private final AtomicInteger nextSnapshotId
      = new AtomicInteger();
  private final ConcurrentMap<Integer, Record[]> snapshots
      = new ConcurrentHashMap<Integer, Record[]>();
  public Record[] takeSnapshot(Date start, Date end,
      int maximumRecordCount,
      Holder<Integer> totalRecordCount,
      Holder<Integer> snapshotId) {
    Record[] snapshot = Records.getSnapshot(start, end);
    totalRecordCount.value = snapshot.length;
    if (totalRecordCount.value <= maximumRecordCount) {
      snapshotId.value = 0;
      return snapshot;
    } else {
      snapshotId.value = nextSnapshotId
          .incrementAndGet();
      snapshots.put(snapshotId.value, snapshot);
      return Arrays.copyOf(snapshot,
          maximumRecordCount);
    }
  }
  public Record[] getSnapshotData(Integer snapshotId,
      int recordIndex, int maximumRecordCount)
      throws InvalidRecordSnapshotIdException {
    Record[] snapshot = snapshots.get(snapshotId);
    if (null != snapshot) {
      return Arrays.copyOfRange(snapshot, recordIndex,
          Math.min(snapshot.length,
            recordIndex + maximumRecordCount));
    } else {
      throw new InvalidRecordSnapshotIdException(
```

```
            "Specified snapshot id was invalid");
      }
    }
    public void freeSnapshot(Integer snapshotId) {
      snapshots.remove(snapshotId);
    }
}
```

Параметр endpointInterface показывает, что операции Web-сервиса определяются не методами самого класса, реализующего сервис, а методами указанного интерфейса. Параметры serviceName позволяет явно задать имя сервиса. Также аннотация WebService поддерживает параметр portName, позволяющий задать имя порта. Кроме имени сервиса и имени порта можно явно задать имя операции — для этого используется аннотация WebMethod и ее параметр operationName.

```
Service recordManagerService = Service.create(
    new URL("http://localhost/records"),
    new QName("http://samples/records",
      "RecordManager"));
RecordManager recordManager = recordManagerService.getPort(
    RecordManager.class);
Holder<Integer> snapshotId = new Holder<Integer>();
Holder<Integer> totalRecordCount = new Holder<Integer>();
Record[] records = recordManager.takeSnapshot(
    start, end, 100, totalRecordCount, snapshotId);
displayRecords(records);
if (totalRecordCount.value > records.length) {
  for (int recordIndex = records.length;
      recordIndex < totalRecordCount.value;
      recordIndex += records.length) {
    records =
        recordManager.getSnapshotData(snapshotId.value,
        recordIndex, 100);
    displayRecords(records);
  }
  recordManager.freeSnapshot(snapshotId.value);
}
```

При отладке Web-сервисов или их клиентов может оказаться полезным вывод входящих и исходящих сообщений в консоль. Чтобы включить эту возможность на стороне Web-сервиса надо установить системное свойство

com.sun.xml.ws.transport.http.HttpAdapter.dump равным true.[4] С той же целью, на стороне клиента Web-сервиса надо использовать свойство com.sun.xml.ws.transport.http.client.HttpTransportPipe.dump. Заметим, что вывод сообщений в консоль будет работать, только если используется библиотека METRO; по умолчанию она не используется.

По умолчанию двоичные данные при передаче кодируются в соответствии со схемой Base64; это может привести к увеличению размера сообщения на треть. Чтобы избежать этого, можно использовать механизм оптимизации передачи сообщений (MTOM — Message Transmission Optimization Mechanism). Суть этого механизма в том, что при передаче вместо действительных бинарных данных в XML помещается заместитель (placeholder); сами же бинарные данные передаются отдельно. Обычно при использовании MTOM сообщение представляет собой архив в формате multipart/related.[5] Содержимым этого архива является как исходное сообщение в формате XML (и в конверте), так и бинарные данные. Рассмотрим пример:

```
// на сервере
@WebService(targetNamespace = "http://samples/files")
@MTOM(threshold = 1024)
public class FileStorage {
  private static final File ROOT_DIRECTORY =
      new File("C:\\Temp");
  public byte[] getContent(String[] path) {
    File file = ROOT_DIRECTORY;
    for (int i = 0; i < path.length; ++i) {
      if ('.' != path[i].charAt(0)) {
        file = new File(file, path[i]);
      } else {
        throw new IllegalArgumentException(
          "The path is invalid");
      }
    }
```

[4] Чтобы установить системное свойство <name> равным <value> можно запустить виртуальную машину с параметром -D<name>=<value> либо во время исполнения вызывать System.setProperty("<name>", "<value>").

[5] Также называется MIME HTML или, кратко, MHTML. Именно этот формат используют браузеры для сохранения Web-страницы вместе с картинками в одном файле.

```
    return doGetContent(file);
  }
  private byte[] doGetContent(File file) {
    ByteArrayOutputStream stream
        = new ByteArrayOutputStream();
    try {
      FileChannel channel
        = new FileInputStream(file).getChannel();
      try {
        channel.transferTo(0, Long.MAX_VALUE,
            Channels.newChannel(stream));
      } finally {
        channel.close();
      }
    } catch (IOException ex) {
      throw new WebServiceException(ex);
    }
    return stream.toByteArray();
  }
}
```

Аннотация MTOM включает поддержку MTOM на стороне Web-сервиса; параметр threshold задает пороговый размер в байтах, начиная с которого бинарные данные посылаются отдельно от XML (по умолчанию порог равен 0). Информация о том, что Web-сервис поддерживает MTOM, включается в WSDL-файл.

На стороне клиента Web-сервиса поддержка MTOM по умолчанию включена. Однако ее можно настроить, задав пороговый размер, или же вовсе выключить. Приведем пример:

```
FileStorage fileStorage =
fileStorageService.getPort(FileStorage.class, new
MTOMFeature(2048));
```

Здесь объект класса MTOMFeature используется для задания порогового размера (на Web-сервисе и на его клиентах могут использоваться разные пороговые размеры). Отметим, что клиент сообщает Web-сервису о своей поддержке MTOM тем, что отправляет входное сообщение в виде архива, причем даже в том случае, когда это сообщение не содержит бинарных данных.

Приведем пример сообщения без конверта, в котором бинарные данные заменены заместителями:

```
<f:getContentResponse xmlns:f="http://samples/files">
 <return>
```

```
  <xop:Include
   xmlns:xop="http://www.w3.org/2004/08/xop/include"
   href="cid:c6a446b8-b1f3-4ce5-8cbc-
d0102e9bbfb0@example.jaxws.sun.com"/>
 </return>
</f:getContentResponse>
```

Заместитель представлен элементом `Include`, причем `href` задает идентификатор части архива, в которой находятся соответствующие бинарные данные.

Существует возможность вызывать Web-сервис даже в том случае, если когда набор его операций не определен ни в одном интерфейсе; эта возможность может оказаться полезной в том случае, когда набор операций становится известен только на этапе исполнения. Следующий пример показывает, как сделать это:

```
// на клиенте
Service helloService = Service.create(
    new URL("http://localhost/hello?wsdl"),
    new QName("http://samples/hello", "HelloService"));
Dispatch<Source> helloDispatch =
helloService.createDispatch(
    new QName("http://samples/hello", "HelloPort"),
    Source.class, Service.Mode.PAYLOAD);
String request = "<h:getHello
      xmlns:h='http://samples/hello'>"
    + "<arg0>World</arg0>"
    + "</h:getHello>";
Source requestSource = new StreamSource(
    new StringReader(request));
Source responseSource =
helloDispatch.invoke(requestSource);
Result responseResult = new StreamResult(System.out);
Transformer transformer
    = TransformerFactory.newInstance().newTransformer();
transformer.transform(responseSource, responseResult);
```

Интерфейс Dispatch представляет диспетчер, позволяющий вызывать операции Web-сервиса динамически, указывая только входящее сообщение. При создании диспетчера надо указать имя порта, способ представления данных (обычно это Source) и режим. Режим определяет, кто должен помещать входное сообщение в конверт и извлекать выходное сообщение из конверта; возможные значения:

• PAYLOAD — все делает диспетчер;

• MESSAGE — все должна делать прикладная программа.

Диспетчер позволяет вызывать операции асинхронно; например:

```
Response<Source> response
    = helloDispatch.invokeAsync(requestSource);
transformer.transform(response.get(), responseResult);
```

Благодаря тому, что блокировка происходит не в методе invokeAsync, а в методе get, клиент может, не дожидаясь получения отклика, выполнять какие-либо действия параллельно с передачей сообщений по сети и их обработкой Web-сервисом; в частности клиент может начать следующий вызов Web-сервиса, не дождавшись окончания предыдущего.

Если клиент желает вовсе предотвратить блокировку, то он может использовать другую форму асинхронного вызова операции; например:

```
helloDispatch.invokeAsync(requestSource,
    new AsyncHandler<Source>() {
  public void handleResponse(Response<Source> response) {
    try {
      transformer.transform(response.get(),
          responseResult);
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
  }
});
```

В этом примере методу invokeAsync передается обработчик завершения асинхронной операции, представленный интерфейсом AsyncHandler. Метод handleResponse вызывается только после поступления отклика так, что метод get никогда не приводит к блокировке. Важно помнить, что метод handleResponse вызывается в нити, отличной от той, в которой был вызван метод invokeAsync.

С помощью диспетчера можно вызывать даже те операции, которые не определены в WSDL (хотя, они, разумеется, должны быть реализованы в самом Web-сервисе); например:

```
// на клиенте
Service helloService = Service.create(
    new QName("http://samples/hello", "HelloService"));
```

```
helloService.addPort(
    new QName("http://samples/hello", "HelloPort"),
    SOAPBinding.SOAP11HTTP_BINDING,
    "http://localhost/hello");
```

Метод addPort добавляет порт, определение которого отсутствует в WSDL.

Аналогом диспетчера на стороне Web-сервиса является провайдер. Приведем пример:

```
// на сервере
@WebServiceProvider
@ServiceMode(Mode.PAYLOAD)
public class HelloProvider implements Provider<Source> {
  public static final String NAMESPACE
      = "http://samples/hello";
  public Source invoke(Source requestSource) {
    try {
      DOMResult requestResult = new DOMResult();
      getTransformer().transform(requestSource,
          requestResult);
      Node getHelloNode
          = ((Document) requestResult.getNode())
            .getDocumentElement();
      if (NAMESPACE.equals(
          getHelloNode.getNamespaceURI())
          && "getHello".equals(
          getHelloNode.getLocalName())) {
        Node arg0Node = getHelloNode.getFirstChild();
        if ("arg0".equals(arg0Node.getNodeName())) {
          String name = arg0Node.getTextContent();
          String hello = String.format(
              "Hello, %s!", name);
          Document document
              = getDocumentBuilder()
                .newDocument();
          Element getHelloResponseElement
              = document.createElementNS(
              NAMESPACE, "getHelloResponse");
          Element returnElement
              = document.createElement("return");
          returnElement.appendChild(
              document.createTextNode(hello));
          getHelloResponseElement.appendChild(
              returnElement);
          document.appendChild(getHelloResponseElement);
          return new DOMSource(document);
```

```
        }
      }
      throw new WebServiceException("Invalid request");
    } catch (TransformerException ex) {
      throw new RuntimeException(ex);
    }
  }
  private static Transformer getTransformer() {
    try {
      return TransformerFactory.newInstance()
          .newTransformer();
    } catch (TransformerConfigurationException ex) {
      throw new RuntimeException(ex);
    }
  }
  private static DocumentBuilder getDocumentBuilder() {
    try {
      return DocumentBuilderFactory.newInstance()
          .newDocumentBuilder();
    } catch (ParserConfigurationException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

Провайдер Web-сервиса должен реализовывать интерфейс Provider (интерфейс Dispatcher является потомком интерфейса Provider); также он должен быть аннотирован WebServiceProvider. Для провайдера, как и для диспетчера, может быть задан режим. Публикация провайдера ничем не отличается от публикации собственно Web-сервиса.

Как Web-сервисы, так и их клиенты могут устанавливать обработчики, которые будут автоматически вызываться для каждого входного и выходного сообщения. Рассмотрим пример:

```
// на сервере или на клиенте
public class DumpHandler
    implements LogicalHandler<LogicalMessageContext> {
  private static final Logger LOGGER
      = Logger.getLogger(DumpHandler.class.getName());
  public boolean handleMessage(
      LogicalMessageContext messageContext) {
    try {
      if (!(Boolean) messageContext.get(
          MessageContext.MESSAGE_OUTBOUND_PROPERTY)) {
        LogicalMessage message
```

```
            = messageContext.getMessage();
        Document requestDocument
            = getDocumentBuilder().newDocument();
        getTransformer().transform(message.getPayload(),
            new DOMResult(requestDocument));
        Element requestElement
            = requestDocument.getDocumentElement();
        LOGGER.log(Level.INFO, "Got request: {0}",
            new QName(
              requestElement.getNamespaceURI(),
              requestElement.getLocalName()));
      }
      return true;
    } catch (TransformerException ex) {
      throw new RuntimeException(ex);
    }
  }
  public boolean handleFault(
      LogicalMessageContext messageContext) {
    return true;
  }
  public void close(MessageContext context) {
  }
  private static Transformer getTransformer() {
    try {
      return TransformerFactory.newInstance()
          .newTransformer();
    } catch (TransformerConfigurationException ex) {
      throw new RuntimeException(ex);
    }
  }
  private static DocumentBuilder getDocumentBuilder() {
    try {
      return DocumentBuilderFactory.newInstance()
          .newDocumentBuilder();
    } catch (ParserConfigurationException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

Обработчики бывают двух видов; один из них — логические обработчики. Логический обработчик должен реализовывать интерфейс LogicalHandler. Метод handleMessage вызывается для обработки обычных сообщений, а метод handleFault — для обработки сообщений, содержащих информацию об ошибке. Оба метода при-

нимают в качестве параметра контекст, представленный интерфейсом LogicalMessageContext. Контекст позволяет получить как само сообщение (с помощью метода getMessage), так и значения свойств, представляющих служебную информацию, связанную с сообщением. Свойство MESSAGE_OUTBOUND_PROPERTY показывает, является ли сообщение входным (false) или выходным (true). Другие свойства позволяют получить информацию о сообщении протокола HTTP, инкапсулирующем обрабатываемое сообщение Web-сервиса, а также об относящихся к обрабатываемому сообщению метаданных, заданных в описании Web-сервиса.

Для логического обработчика сообщение представлено интерфейсом LogicalMessage. Этот интерфейс позволяет не только получать содержащиеся в сообщении данные с помощью метода getPayload но и устанавливать их с помощью метода setPayload.

Подключение обработчиков на стороне Web-сервиса, и на стороне его клиентов осуществляется по-разному. На стороне Web-сервиса это выглядит так:

```
// на сервере
@WebService(targetNamespace = "http://samples/hello")
@HandlerChain(file = "HelloHandlers.xml")
public class Hello {
  public String getHello(String name) {
    return String.format("Hello, %s!", name);
  }
}
```

Аннотация HandlerChain задает файл, в котором должны быть перечислены обработчики. Обычно этот файл располагается там же, где и файлы классов. Приведем пример файла, перечисляющего обработчики.

```
<!-- на сервере -->
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
      <handler-name>DumpHandler</handler-name>
      <handler-class>
       samples.soap.handlers.DumpHandler
      </handler-class>
    </handler>
  </handler-chain>
```

```
</handler-chains>
```

Отметим, что порядок перечисления обработчиков в упомянутом файле определяет порядок их вызова.

На стороне клиента Web-сервиса можно подключить обработчики к порту можно так:

```
((BindingProvider) helloPort).getBinding().setHandlerChain(
    Collections.<Handler>singletonList(new DumpHandler()));
```

Метод setHandlerChain принимает в качестве параметра список обработчиков.

Кроме логических обработчиков поддерживаются SOAP-обработчики. Они отличаются тем, что должны реализовывать интерфейс SOAPHandler, а контекст и сообщение представлены для них интерфейсам SOAPMessageContext и SOAPMessage соответственно. В отличие от логических обработчиков SOAP-обработчики имеют доступ не только к самому сообщению, но и к конверту. Поэтому SOAP-обработчики обычно используются для обработки служебных данных, помещаемых в заголовок конверта. Подключение SOAP-обработчиков ничем не отличается от подключения логических обработчиков.

У клиента Web-сервиса есть возможность задавать для каждой ссылки на порт служебную информацию, которая будет передаваться Web-сервису при каждом вызове операции этого порта. Обычно таким способом задают имя пользователя и пароль, что делает возможным аутентификацию средствами используемого коммуникационного протокола (от коммуникационного протокола зависит как то, какой конкретный механизм аутентификации будет использоваться, так и само наличие этого механизма). Приведем пример:

```
// на клиенте
Hello helloPort = helloService.getHelloPort();
BindingProvider bindingProvider
    = (BindingProvider) helloPort;
Map<String, Object> requestContext
    = bindingProvider.getRequestContext();
requestContext.put(BindingProvider.USERNAME_PROPERTY,
    "fred");
requestContext.put(BindingProvider.PASSWORD_PROPERTY,
    "rockbed");
```

Помимо свойств USERNAME_PROPERTY и PASSWORD_PROPERTY можно отметить свойство ENDPOINT_ADDRESS_PROPERTY, позволяющее явно задать адрес Web-сервиса (по умолчанию используется адрес, указанный в WSDL-файле).

На стороне Web-сервиса параметры аутентификации определяются конфигурацией Web-контейнера либо сервера приложений, в котором выполняется Web-сервис. Если Web-сервис выполняется в обычном приложении и публикуется с помощью класса Endpoint, то простого способа, позволяющего задать параметры аутентификации, нет.

5. Контрольные вопросы

1. Что такое web-сервис?
2. Какие коммуникационные протоколы используются web-сервисами?
3. Для чего используется язык WSDL?
4. Какова структура простейшего web-сервиса?
5. Какие типы данных можно использовать для описания параметров и возвращаемого значения операций web-сервиса?
6. Как опубликовать web-сервис?
7. Какова структура простейшего клиента web-сервиса?
8. Как вызвать операцию опубликованного web-сервиса?
9. Что такое SOAP?
10. Из каких частей состоит пакет SOAP?
11. Каким образом web-сервис определяет, какая операция вызвана клиентом?
12. Для чего используется аннотация WebParam и класс Holder?
13. Для чего используется аннотация Oneway?
14. Что такое MTOM?
15. Что называют динамическим вызовом операции и как он осуществляется?
16. В чем разница между режимами PAYLOAD и MESSAGE?
17. Как вызвать операцию асинхронно?
18. Как вызвать операцию, не определенную в WSDL?
19. В каких случаях может понадобиться реализовать интерфейс Provider?

20. Что называют обработчиков вызова операции?
21. О чего зависит порядок вызова обработчиков?
22. Какие виды обработчиков бывают?
23. Как задать обработчик на стороне web-сервиса?
24. Как задать обработчик на стороне клиента web-сервиса?
25. Как передать вместе с вызовом операции дополнительные данные?

6. Варианты заданий

1. Чтение файлов на удаленном компьютере
2. Запись файлов на удаленном компьютере
3. Просмотр дерева каталогов на удаленном компьютере
4. Рекурсивный поиск файла на удаленном компьютере
5. Выполнение команды ОС на удаленном компьютере
6. Показ на удаленном компьютере графического изображения
7. Получение снимка экрана удаленного компьютера
8. Имитация нажатия клавиши на клавиатуре удаленного компьютера
9. Игра с удаленным компьютером в крестики-нолики
10. Отгадывание «задуманного» удаленным компьютером целого числа
11. Удаленная база данных отдела кадров
12. Удаленная база данных деканата
13. Удаленная база данных склада
14. Удаленное тестирование знаний
15. Электронная почта

7. Список литературы

1. Хабибуллин И. Создание распределенных приложений на Java 2. — СПб.: БХВ-Петербург, 2002. — 704 с.
2. Дейтел Х. М., Дейтел П. Дж., Сантри С. И. Технологии программирования на Java 2. Книга 3. Корпоративные системы, сервлеты, JSP, Web-сервисы — М.: ООО «Бином-Пресс», 2003. — 672 с.
3. Daigneau R.Service Design Patterns : Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services — Addison-Wesley, 2012 — 354 p.