WebService in Java SE und EE

Wolfgang Nast
MT AG
Ratingen

Schlüsselworte
Java, JAX-WS, JAX-RS, JAXB, XML.

Einleitung

Es werden die Mölichkeiten von WebServices in Java SE und EE, mit SOAP und REST gezeigt. Die Verwendung von JAX WS in SE mit dem internen WebServer wird gezeigt. Es werden auch die Grenzen beschrieben gegenüber der EE Umsetzung. In EE werden Beispiele mit GlasFish gebaut für die Verwendung von Annotations mit SOAP und REST. Auch die Verwendung von SOAP WS die mit wsdl beschrieben ist wird Vorgestellt. Hierfür wird WSImport verwendet. Der Tomcat wird mit den RI Metro und Jersy auch für WebServices nutzbar.

Webservices in JAVA SE und EE

Es wir unterteilt in Fähigkeiten von Java SE mit SOAP und Java EE mit SOAP und REST.

Webservice in JAVA SE

In Java SE können Webservices nur in SOAP realisiert werden. Dafür verwendet man JAX-WS. Als weiter Einschränkung ist zu beachten, dass der mitgelieferte http-Server von Java SE7 keine verschlüsselte Verbindung unterstützt.

Webservice in Java EE

Es wird unterschieden in SOAP und REST. Hier wird auch die verschlüsselte Verbindung unterstützt. Hier geben die unterschiedlichen Applicationserver vor, welche Verschlüsselungen möglich sind.

SOAP Webservice

Beim SOAP Webservice gibt es zwei Wege, wie man den Service erstellt, „Top Down“ und „Bottom Up“.

Bei „Top Down“ beginnt man mit der Beschreibung des Interfaces mit WSDL. Hierfür gibt es passende Editoren oder direkte Unterstützung in der IDE. Mit dem Programm WSimport generiert man dann die Java-Klassen für den Aufruf. Jetzt sind die Implementierungsklassen noch mit der Realisierung des Webservice zu füllen.

Bei „Bottom Up“ beginnt man mit der Implementierung der Webservices. Bei Java EE kann man sehr leicht aus einem Sessionbean durch hinzufügen der Webservice-Annotation einen Webservice machen. Bei Java SE werden auch die WebService-Annotations verwendet. Hier muss dann noch die Implementierung vorgenommen werden und es müssen noch die EndPoints eingetragen werden. Das WSDL wird hier von Java SE oder dem Applicationserver generiert.

REST WebService

Beim REST WebService gibt es keine Interface Beschreibung. Hier wird die Implementierung direkt mit den Annotations umgesetzt.

Realisierung

Angefangen wird mit dem Beispiel für SOAP in Java SE.

Eine einfache wsdl-Datei wird genommen, aus dem die Klassen für den WebService generiert werden.

Die Datei heißt „Hello.wsdl“ und mit folgendem Aufruf wird der Java-Teil daraus generiert:

```
wsimport -d gen -Xnocompile Hello.wsdl
```

Dabei ist das Verzeichnis zum Generieren „gen“ und es wird nicht direkt übersetzt.

```
wsimport -clientjar HelloWS.jar Hello.wsdl
```

Hier wird der Zugriff direkt in ein Jar verpackt.

Das Generieren geht auch mit ANT:

```
...
    <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
        <classpath>
            <fileset dir="wstools" includes="*.jar" />
        </classpath>
    </taskdef>
...
    <target name="wsimp" depends="xjc">
        <mkdir dir="srcwsimp"/>
        <wsimport destdir="srcwsimp2" sourcedestdir="srcwsimp"
xnocompile="true" wsdl="wsdl/Hello.wsdl"
wsdllocation="http://localhost:8442/Beispiel/Hallo?wsdl">
            <depends file="wsdl/Hello.wsdl"/>
            <produces dir="srcwsimp"/>
        </wsimport>
    </target>
...
```

Dabei werden die Dateien nach „srcwsimp“ generiert.

Das Generieren geht auch mit Maven:

```
...
  <build>
    <plugins>
      <plugin>
        <groupId>org.jvnet.jax-ws-commons</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <version>2.2</version>
        <executions>
          <execution>
            <goals>
              <goal>wsimport</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <wsdlDirectory>${basedir}/src/wsdl</wsdlDirectory>
```

```
<wsdlLocation>http://localhost:8442/Beispiel/Hallo?wsdl</wsdlLocation>
        </configuration>
      </plugin>
    </plugins>
  </build>
...
```

Jetzt kann man sich die generierten Dateien ansehen. Dabei wird der WebService als Interface zur Verfügung gestellt.

```
...

@WebService(name = "Hallo", targetNamespace =
"http://www.beispiel.de/Beispiel")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
@XmlSeeAlso({
    ObjectFactory.class
})
public interface Hallo {
    @WebMethod(action = "http://www.beispiel.de/Beispiel/Hallo")
    public void hallo();
...
```

Für den Zugriff wurde folgender Client generiert:

```
...

@WebServiceClient(name = "HalloInterfaceService", targetNamespace =
"http://www.beispiel.de/Beispiel/Hallo", wsdlLocation =
"http://localhost:8442/Beispiel/Hallo?wsdl")
public class HalloInterfaceService
    extends Service
{
...
```

Wie der Name Client schon sagt, ist es die Implementierung für den Client. Damit hat man mit dem generierten Jar auch schon die Implementierung für den Client.

Uns geht’s jetzt um die Implementierung des Servers. Dafür kann man eine Klasse realisieren:

```
...

@MTOM
@WebService(endpointInterface = "de.beispiel.beispiel.Hallo",
         targetNamespace = "http://www.beispiel.de/Beispiel/Hallo",
         serviceName = "HalloInterfaceService")
public class ServerHallo implements Hallo
...
   @Override
   public String hallo() {
      return "Hallo SOAP WS";
   }
...
```

Damit ist das generierte Interface als Basis für die Implementierung genommen worden.

Zum Starten des WebServices braucht man in Java SE nur den Endpoint zu veröffentlichen:

```
...
   public static void main(String[] args) {
      Endpoint.publish("http://localhost:8442/Beispiel/Hallo", new
ServerHallo());
   }
...
```

Und schon läuft der WebService als eigener Thread.

Für den AppServer GlassFish reicht es die War-Datei zu erstellen und zu deployen.

Es sind keine Einträge in „web.xml“ notwendig.

WebService mit Annotation

Zum Realisieren des WebService kann man auch mit dem Java-Code beginnen . Dafür wird die Realisierung um die Annotations erweitert.

```
...
@MTOM
@WebService(targetNamespace = "http://www.beispiel.de/Beispiel/Hallo",
         serviceName = "HalloInterfaceService",
         name = "Hallo",
         portName = "HalloPort")
public class ServerHallo {
   @WebMethod(action = "http://www.beispiel.de/Beispiel/Hallo")
   public String hallo() {
      return "Hallo SOAP WS";
   }
...
```

Jetzt kann man sich noch das WSDL generieren lassen. Es ist nicht nötig in Java 7 SE oder bei GlassFish.

```
wsgen -keep -s srcgen -r srcgen -wsdl -d bingen -inlineSchemas
de.beispiel.beispiel.hallo.ServerHallo
```

Hierbei ist –wsdl und –inlineSchemas sinnvoll, damit man nur eine wsdl-Datei hat, die nicht noch von externen xsd-Dateien abhängig ist, die auch zur Verfügung gestellt werden müssten. Leider ist die wsdl-Datei nicht vollständig, sondern nur eingeschränkt nutzbar. Für WebService-Aufrufe ist sie ausreichend, aber zum Generieren des Client-Jar mit wsimport ist nicht alles vorhanden.

Der Rest geht genauso wie bei dem WebService aus der WSDL.

REST WebService

Für die Implementierung von REST Webservices werden die Annotations eingetragen.

Hier ein Beispiel mit Daten:

```
...
@Path("/RS")
public class DatenService
...
   @GET
   @Path("/Status")
```

```
   @Produces(MediaType.APPLICATION_XML)
   public String getStatus() {
      int anzahl = buf.getIdMap().size();
      return "<?xml version=\"1.0\" encoding=\"UTF-
8\"?><Status><Server>Status Online!</Server></Status>";
   }
...
   @PUT
   @Path("/Daten/{name}")
   @Consumes(MediaType.TEXT_PLAIN)
   @Produces(MediaType.APPLICATION_XML)
   public String setDaten(@PathParam("name") String name, String daten)
...
```

Hier ist der Pfad für das Abholen des Status „/RS/Status“ und die Antwort ist in XML.

In der Datei web.xml ist folgender Eintrag vorzunehmen:

```
...
    <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-
class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey REST Service</servlet-name>
        <url-pattern>/Daten/*</url-pattern>
    </servlet-mapping>
...
```

Tomcat erweitern für die WebServices

Man kann die WebServices SOAP und REST auch mit dem Tomcat verwenden. Dafür müssen die RferenzImplementierungen(RI) in Tomcat installiert werden.

Für SOAP ist die RI Metro mit der Beschreibung im Netz:

`http://metro.java.net/1.2/docs/install.html`

Für REST ist die RI Jersey mit der Beschreibung im Netz:

`http://jersey.java.net/`

Kontaktadresse:
Wolfgang Nast
MT AG
Balcke-Dürr-Allee 9
D-40882 Ratingen

Telefon:	+49 (0) 2102 30961-0
Fax:	+49 (0) 2102 30961- 101
E-Mail	Wolfgang.Nast@mt-ag.com
Internet:	www.mt-ag.com