In diesem Kapitel

- Um was geht's?
- Installation von Axis
- Nutzen von Services mit Axis
- Veröffentlichen von Web Services mit Axis
- XML – Java Data Mapping in Axis

Apache AXIS

1.1. Um was geht's?

Sie wollen Apache SOAP kennen lernen, ohne über alle möglichen Probleme zu stolpern. In der Starthilfe wurde gezeigt, wie Sie Axis aufsetzen, testen und einfache Dienste deployen können. Nun gehen wir eine Ebene tiefer und befassen uns mit mehr Details. Teils sind die Themen überlappend mit der Starthilfe. Hier gehen wir aber genauer auf die Themen ein.

1.1.1. Was ist SOAP?

SOAP ist ein XML-basiertes Kommunikations-Protokoll und Kodierschema für Inter-Applikations-Kommunikation. Der Vorgänger von SOAP, XML-RPC, stammte von Dave Winer von der Firma Userland. SOAP wurde von Winer zusammen mit Microsoft in einer ersten Version definiert und anschliessend auch von andern Firmen unterstützt bis es schliesslich von W3C akzeptiert wurde. W3C hatte eine (nun eingeschlafene) Working Group zum Thema „XML Protokolle". Diese Gruppe definierte typische Szenarien und versuchte ein allgemeines Framework für XML Protokolle zu definieren. In Wahrheit orientierte sich dieses aber sehr an SOAP. Daher wurden die Aktivitäten auch eingestellt. SOAP aber überlebte und ist ab Version 1.2 sogar verständlich.

SOAP wird allgemein als Cross-Plattform, Cross-Programmiersprache Backbone Technologie für verteilte Applikationen angesehen – Web Services.

1.1.2. Was ist Axis?

Axis ist eine SOAP ENgine – ein Framework für die Konstruktion von SOAP Prozessoren, Clients, Servern, Gateways usw. Die aktuelle Version ist in Java geschrieben; die C++ Version ist (fast) brauchbar, zumindest zum Testen, für die Client-Seite.

Axis ist mehr als SOAP. Axis umfasst auch:

- einen standalone Server
- einen Server, der in Servlet Engines eingebaut werden kann, beispielsweise in Tomcat
- Unterstützung für die *Web Service Description Language* (WSDL)
- Werkzeuge, mit deren Hilfe Java Klassen aus WSDL generiert werden können
- Beispielprogramme
- Tools – beispielsweise den TCP Monitor

Axis gilt als Dritt-Generation SOAP (bei IBM hiess Axis SOAP4J). Gegen Ende 2000 wurde in den SOAP v2 Kommittees diskutiert, wie die SOAP Engines felxibler, leichter zu konfigurieren und einfacher zu handhaben gemacht werden könnten. Zudem sollte die Architektur so gewählt werden, dass neben SOAP auch allfällig weitere XML Protokolle von W3C unterstützt werden könnten.

Da die ersten Versionen von SOAP Engines kaum wartbar waren, sollte von Grund auf neu designed und entwickelt werden.

Nach vielen Diskussionen in verschiedenen Gremien einigte man sich auf folgende Key-Features:

- **Speed**:
  Axis verwendet SAX, einen ereignisgesteuerten XML Parser. Dieser ist schneller als die in älteren Apache Versionen verwendeten DOM Parser.
- **Flexibilität**:
  Die Axis Architektur gestattet es dem Entwickler, Erweiterungen in den SOAP Engine einzubauen (Header Verarbeitung, System-Management, u.v.m.)
- **Stabilität**:
  Axis definiert publizierte Interfaces, welche sich kaum verändern, so dass alle Erweiterungen, die diese Interfaces beachten, langlebig sein sollten.
- **Komponenten-orientiertes Deployment:**
  Axis gestattet die Definition von wieder verwendbaren Handlern, mit deren Hilfe unterschiedliche Kommunikations-Szenarien („Patterns“) zwischen geschäftspartnen oder Anwendungen realisiert werden können.
- **Transport Framework:**
  Der Kern von Axis ist völlig Transport-unbhängig, nicht zueltzt weil SOAP Sender und Listener über unterschiedliche Protokolle unterstützt (SMTP, FTP, Message-orientierte Middleware usw.).
- **WSDL Unterstützung:**
  Axis unterstützt die Web Service Description Language (WSDL). Mit deren Hilfe können Sie einfach Stubs für remote Services bauen und automatisch maschinenlesbare Beschreibungen (WSDL) der bereits vorhandenen Services generieren.

AXIS = Apache Extensible Interaction System.

1.1.3. Was steckt in der aktuellen Version von Axis?

Die aktuelle Version von SOAP ist bereits einigermassen stabil, im Gegensatz zu Version 1.0. In Axis sind enthalten:

- Eine SOAP 1.1/1.2 Engine
- Flexible Konfihurations- und Deployment-Möglichkeiten
- „drop-in“ Deployment der SOAP Services (JWS): Java statt Class Dateien
- Unterstützung aller Basisdatentypen und ein Type Mapping System für die Definition neuer Serializers/Deserializers
- Automatische Serialisierung / Deserialisierung der Java Beans, inklusive dem Mapping der Felder auf XML Elemente und Attribute.
- Automatische Konversion zwischen Java Collections und SOAP Arrays und umgekehrt.
- Providers für RPC und Message basierte SOAP Services
- Automatische WSDL Generierung von bereits deployed Services
- WSDL2Java Tool, um Java Proxies und Skeletons aus WSDL Dokumenten zu generieren.
- Java2WSDL Tool, um WSDL aus Java Klassen zu generieren.

- Einfache Securiry Erweiterungen, welche mit den Servlet 2.2 Security / Roles zusammenarbeiten.
- Unterstützung von Session-orientierten Services, via http Cookies oder Transport-unabhängigen SOAP Headern.
- Unterstützung einfacher „SOAP with Attachment“ Features.
- Ein EKB Provider, um auf EJB’s als Web Services zugreifen zu können.
- http Servlet-basierter Transport
- JMS basierter Transport
- Einen einfachen standalone Server (http Support)

1.2. Installation von Axis

Die Installation von Axis wird in der Starthilfe beschrieben. Beachten Sie den Classpath, den Axis Applikationen benötigen:

- falls Sie Axis selber aus einem CVS System auschecken und selber bauen, dann stehen die Bibliotheken im Verzeichnis AXIS_HOME\java\build\lib, sonst (beim Extrahieren des Axis ZIP Archivs) AXIS_HOME\lib.
- Die Libraries:
  axis-1_1/lib/axis.jar
  axis-1_1/lib/jaxrpc.jar
  axis-1_1/saaj.jar
  axis-1_1/commons-logging.jar
  axis-1_1/commons-discovery.jar
  axis-1_1/lib/wsdl4j.jar
  axis-1_1/... ein JAXP-1.1 konformer XML Parser (Xerxes oder Crimson)

1.3. Web Services mit Axis

Nun geht’s richtig los. Zuerst benutzen wir einen Echo Service auf dem Apache Tomcat Server.

1.3.1. Grundlagen – Der Start

Wir wollen eine Zeichenkette an einen vorhandenen Service senden und erhalten das Echo, also denselben String zurück.

```
package axis;

package axis;


/**
 * @author jjoller
 *
 */
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

import javax.xml.namespace.QName;

public class EchoClient2 {
   public static void main(String [] args) {
         try {
            String endpoint =
```

```
                  "http://localhost:8080/axis/services/urn:echo-service";
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress( new java.net.URL(endpoint) );
            call.setOperationName(new QName("http://soapinterop.org/",
                        "getEcho") );
            String ret = (String) call.invoke( new Object[]
                  { "Hallo, wie geht's!" } );

            System.out.println("Gesendet 'Hallo, wie geht's';Antwort: '"
                              + ret + "'");
         } catch (Exception e) {
            System.err.println(e.toString());
         }
   }
}
```

Erklärung:

1) Service und Call Objekte sind Standard JAX-RPC Objekte, welche Metadaten über den aufzurufenden Service speichern.
2) Endpoint URL: Ziel unserer SOAP Message
3) Operation : Methodenname des Web Services
4) `Call.invoke()` : Aufruf des gewünschten Services, mit Parameter (`String`)

Und hier die XML Message:
a) vom Client zum Server

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <SOAP-ENV:Body>
            <ns1:getEcho soapenv:encodingStyle=
                        ”http://schemas.xmlsoap.org/soap/encoding/”
            xmlns:ns1=  ”http://soapinterorp.org”>
            <ns1:arg0 xsi:type="xsd:string">Hallo, wie geht&aposs
            </ns1:arg0>
            </ns1:getEcho>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

b) vom Server zum Client

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <soapenv:Body>
            <ns1:getEchoResponse soapenv:encodingStyle=
                        ”http://schemas.xmlsoap.org/soap/encoding/”
            xmlns:ns1=  ”http://soapinterorp.org”>
            <ns1:getEchoReturn xsi:type="xsd:string">Echo vom Host...
            </ns1:getEchoReturn>
            </ns1:getEchoResponse>
      </soapenv:Body>
</soapenv:Envelope:Envelope>
```

Das Argument der remote Methode wird automatisch in XML serialisiert und der Server antwortet mit demselben String, welcher deserialisiert und ausgegeben wird.

1.3.2. Parameter-Namen

Im obigen Beispiel haben wir dem Parameter keinen eigenen Namen gegeben. Axis nennt somit die Parameter einfach „arg0“, „arg1“, ..., in unserem Fall also „arg0“.

Das können wir auch ändern, indem wir die Methode `addParameter()` vor dem Methodenaufruf ausführen.

Zudem müssen wir auch noch den Datentyp der Rückgabe angeben. Dies geschieht mithilfe der Methode `setReturnType(org.apache.axis.Constants...)`.

```
call.addParameter("testParam",
                  org.apache.axis.Constants.XSD_STRING,
                  javax.xml.rpc.ParameterMode.IN);
call.setReturnType(org.apache.axis.Constants.XSD_STRING);
```

Der Name des Parameters wurde frei gewählt; er hat nichts mit dem Namen des formalin Parameters im Service Methodenaufruf zu tun.

Die Monitor Nachricht sieht entsprechend anders aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <SOAP-ENV:Body>
            <ns1:getEcho soapenv:encodingStyle=
                  ”http://schemas.xmlsoap.org/soap/encoding/”
            xmlns:ns1=  ”http://soapinterorp.org”>
            <ns1:testParam xsi:type="xsd:string">Hallo, wie geht&aposs
            </ns1:testParam>
            </ns1:getEcho>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Die Meldung vom Server bleibt unverändert.

1.3.3. Datentypen und XML in SOAP

Im obigen Beispiel haben wir gewusst, dass der Rückgabetyp des Methodenaufrufes (`invole()`) explizit in eine Zeichenkette gecastet. Wir wissen also, welcher Objekttyp passend ist. Aber was, wenn wir den Datentyp der Antwort noch nicht kennen?

Schauen wir uns die obige Antwort des Servers nochmals genauer an:

```
      <SOAP-ENV:Body>
            <ns1:getEcho soapenv:encodingStyle=
                  ”http://schemas.xmlsoap.org/soap/encoding/”
            xmlns:ns1=  ”http://soapinterorp.org”>
            <ns1:testParam xsi:type="xsd:string">Hallo, wie geht&aposs
            </ns1:testParam>
            </ns1:getEcho>
      </SOAP-ENV:Body>
```

Offensichtlich übermittelt Axis den Datentyp, als XML verschlüsselt. Wir können aufgrund der Datentypinformation im XML den passenden Zieltyp beschreiben.

Andere Toolkits enthalten keine Typeninformationen, wie beispielsweise in folgendem Auszug:

```
<SOAP-ENV:Body>
      <ns1:getEcho soapenv:encodingStyle=
                  ”http://schemas.xmlsoap.org/soap/encoding/”
      xmlns:ns1=  ”http://soapinterorp.org”>
      <ns1:result>Hallo, wie geht&aposs
      </ns1:result>
      </ns1:getEcho>
</SOAP-ENV:Body>
```

In diesem Fall wird das Mapping auf die Programmiersprachen-Datentypen schwieriger. In diesem Fall müssen wir auf die Metadaten ausweichen:

- wir benötigen eine Beschreibung des Services, aus der wir herauslesen können, welchen Rückgabetypus wir zu erwarten haben.
- Beispiel:
  `call.setReturnType(org.apache.axis.Constants.XSD_STRING)`
  Diese Methode besagt, dass der Client, falls er keine Datentyp-Information vom Server erhält, so agieren soll, als ob der Rückgabewert ein `xsi:type` Attribut sei.

Axis geht noch einen Schritt weiter:

- Sie können auch eine Java Klasse als Datentyp angeben
- Beispiel: `call.setReturnClass(String.class);`

Ein vollständiges Beispiel sehen Sie im Echo Client, mit erweiterter Parameterübergabe.

1.4. Publizieren von Web Services mit Axis

Als nächstes schauen wir uns einen Taschenrechner an. Dieser kennt nur die zwei Operationen add und subtract und dies erst noch nur für Integer Variablen.

Der Dienst ist denkbar einfach:

```
public class Taschenrechner {
      /**
       * Addition
       * @param i1
       * @param i2
       * @return Summe
       */
      public int add(int i1, int i2) {
            return i1 + i2;
      }
      /**
       * Subtraktion
       * @param i1
       * @param i2
       * @return Differenz
       */
      public int subtract(int i1, int i2) {
            return i1 - i2;
      }

}
```

Wie kann man diesen Web Service deployen? Natürlich musste ich alle bisher benutzenDienste auch deployen; aber jetzt sind Sie dran!

Zuerst benutzten wir den *instant deployment* Service von Axis.

1.4.1. JWS (Java Web Service) Dateien – Axis Instant Deployment

Dieses Verfahren ist denkbar einfach:

1) erstellen Sie den Service als Java Datei: <Service>.java
2) benennen Sie die Datei um: <Service>.jws
3) kopieren Sie diese Datei ins Verzeichnis <webapp-root>/axis/Taschenrechner.jws
4) ... das war's!
   Nutzen Sie den Taschenrechner über das Web:
   http://localhost:8080/axis/Taschenrechner.jws?method=add&i1=12&i2=34

   Als Ergebnis erhalten Sie (nachdem Axis die Java Datei übersetzt hat):

```
<?xml version="1.0" encoding="UTF-8" ?>
- <soapenv:Envelope
mlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
mlns:xsd="http://www.w3.org/2001/XMLSchema"
mlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <soapenv:Body>
- <addResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<addReturn xsi:type="xsd:int">46</addReturn>
</addResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Die Namen der Parameter sind dabei unwichtig, die Methode natürlich schon.

Bemerkung
JWS sind lediglich für einfache Web Services sinnvoll und machbar:

- JWS dürfen keine Packages enthalten
- Fehler sind erst nach dem deployen erkennbar, beim Methodenaufruf.
- Professionelle Web Services verwenden immer Web Service Deployment Descriptoren, weil sich damit der Service besser beschreiben lässt.
- Sie benötigen den Source Code des Services

Beachten Sie auch folgenden Punkt:

- obschon der SOAPMonitor eingeschaltet ist, werden Sie keine SOAP Meldungen sehen
- der Grund ist der, dass der SOAP Monitor die urn verwendet, also eine Grösse, die Sie im JWS Verfahren nicht angeben können!

1.4.2. Custom Deployment – WSDD Starthilfe

JWS Dateien erlauben Ihnen, Web Services schnell zu entwickeln und zu testen. Aber JWS Dienste haben alle Nachteile, die wir eben geschildert haben.

Eventuell kann in einem späteren Release zusätzliche Metainformation im Service spezifiziert werden. Warten wir mal die Entwicklung ab.

1.4.2.1. Deployen mithilfe von Descriptoren

Wenn man die Möglichkeiten von Axis nutzen möchte, dann kommt man nicht darum herum, sich mit dem Web Service Deployment Descriptor (WSDD) zu beschäftigen.

Schauen wir unbs zuerst ein einfaches Beispiel an:

```
- <deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
- <service name="urn:beispiel3" provider="java:RPC">
  <parameter name="className" value="Beispiel3" />
  <parameter name="allowedMethods" value="*" />
    </service>
    </deployment>
```

Die einzelnen Zeilen und ihre Bedeutung

1) zuerst definieren wir den Namensraum für WSDD: wir wollen einen WSDD beschreiben
2) dann folgt der Namensraum für Java, unsere Implementierungssprache
3) der Service wurd als urn:beispiel3 beim Server angemeldet;
   es handelt sich um einen RPC (keinen Messaging) Web Service in Java. Es handelt sich in der Axis Architektur um einen Provider, „java:RPC“
4) der Web Service wird mithilfe der Klasse Beispiel 3 implementiert
5) Sie können alle Methoden des Dienstes („*“) nutzen.
   Alternativ könnten wir auch eine Liste der erlaubten Methoden (durch Leerzeichen getrennte Methodennamen) angeben.

1.4.2.2. WSDD mit Optionen

Nun ehen wir genauer auf zusätzliche Informationen, die Sie über einen Web Service angeben können. Auf die Axis Handler gehen wir später ein.

1.4.2.2.1. Scoped Services

Bei den Scoped Services geht es darum festzulegen, wie ein Service Objekt gehandhabt wird:

1) „**Request**“ Scope (Default):
   für jeden SOAP Request im Service wird ein neues Objekt kreiert
2) „**Application**“ Scope:
   In diesem Fall wird das Singleton Pattern benutzt d.h. ein einzelnes Objekt bedient alle Requests.
3) „**Session**“ Scope:
   kreiert für jeden Session-enabled Client ein Objekt.

Die Scope Option geschieht mithilfe eines Parameters für Ihren Service:

```
<service name="BestellService"...>
      <parameter name="scope" value="application">
      ....
```

1.4.2.3. Einsatz des Admin Clients

In unseren Beispielen haben wir mithilfe eines Apache Ant Skripts aus Eclipse die Dienste deployed. Im Skript selber haben wir jedoch die Ant Axis Dienste mithilfe einer `<taskdef>` festgelegt.

Der Apache Admin Client ist in folgender Klasse:

```
org.apache.axis.client.AdminClient
```

Falls Ihr Container nicht Tomcat ist, müssen Sie den Port mit `-p<port>` angeben.

Skript Beispiele (jeweils eine Zeile!)

```
%JAVA_HOME%\bin\java -cp %AXIS_PATH%
org.apache.axis.client.AdminClient -
lhttp://localhost:8080/axis/services/AdminService deploy.wsdd
```

```
%JAVA_HOME%\bin\java -cp %AXIS_PATH%
org.apache.axis.client.AdminClient deploy.wsdd
```

mit

```
<deployment name="test" xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="urn:xmltoday-delayed-quotes" provider="java:RPC">
    <parameter name="className"
               value="samples.stock.StockQuoteService"/>
    <parameter name="allowedMethods" value="getQuote test"/>
    <parameter name="allowedRoles" value="user1,user2"/>
    <parameter name="wsdlServicePort" value="GetQuote"/>
    <requestFlow name="checks">
      <handler
    type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>
      <handler
    type="java:org.apache.axis.handlers.SimpleAuthorizationHandler"/>
    </requestFlow>
  </service>
  <service name="urn:cominfo" provider="java:RPC">
    <parameter name="className" value="samples.stock.ComInfoService" >
    <parameter name="allowedMethods" value="getInfo" />
    <parameter name="allowedRoles" value="user3"/>
    <requestFlow type="checks"/>
  </service>
</deployment>
```

und folgendem Axis Pfad:

```
set AXIS_PATH=c:\axis-1_1\lib\axis.jar;c:\axis-1_1\lib\axis-
ant.jar;c:\axis-1_1\lib\commons-discovery.jar;c:\axis-1_1\lib\commons-
logging.jar;c:\axis-1_1\lib\jaxrpc.jar;c:\axis-1_1\lib\log4j-
1.2.8.jar;c:\axis-1_1\lib\saaj.jar;c:\axis-1_1\lib\wsdl4j.jar;
```

Wir können den Web Service mit folgendem Skript testen:

```
%JAVA_HOME%\bin\java -cp %AXIS_PATH%;.;.. OPService -uuser3 -wpass3
eBay gold
```

Sie können den Dienst *undeployen*, also vom Applikations Server entfernen / deaktivieren, indem Sie folgendes Skript ausführen:

```
%JAVA_HOME%\bin\java -cp %AXIS_PATH%
org.apache.axis.client.AdminClient undeploy.wsdd %*
```

mit

```
<undeployment name="test" xmlns="http://xml.apache.org/axis/wsdd/">
  <service name="urn:xmltoday-delayed-quotes"/>
  <service name="urn:cominfo"/>
</undeployment>
```

Falls Sie eine *Liste der vorhandenen Dienste* möchten, dann können Sie folgendes Skript verwenden:

```
%JAVA_HOME%\bin\java -cp %AXIS_PATH%
org.apache.axis.client.AdminClient list
```

mit folgender Ausgabe:

```
- <deployment xmlns="http://xml.apache.org/axis/wsdd/" xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  + <globalConfiguration>
  + <handler name="soapmonitor" type="java:org.apache.axis.handlers.SOAPMonitorHandler">
    <handler name="LocalResponder" type="java:org.apache.axis.transport.local.LocalResponder" />
    <handler name="URLMapper" type="java:org.apache.axis.handlers.http.URLMapper" />
    <handler name="Authenticate" type="java:org.apache.axis.handlers.SimpleAuthenticationHandler" />
  - <service name="urn:beispiel3" provider="java:RPC">
      <parameter name="allowedMethods" value="*" />
      <parameter name="className" value="Beispiel3" />
    </service>
  + <service name="AdminService" provider="java:MSG">
  + <service name="Version" provider="java:RPC">
  + <service name="SOAPMonitorService" provider="java:RPC">
  + <transport name="http">
  + <transport name="local">
  </deployment>
```

Dieses Listing ist identisch mit der Datei server-config.wsdd bei Tomcat.

Die entsprechenden Skripte zu den Beispielen finden Sie auf dem Server.

1.4.2.4. Handlers und Chains

Nun möchten wir etwas tiefer in die Architektur von Axis eindringen. Als Beispiel möchten wir feststellen, wieoft ein bestimmter Service aufgerufen wurde. Diese Aufgabe kann mithilfe einer Handler Klasse bestimmt werden, wobei man den Handler selbst auch deployen muss.

Handlers sind Erweiterungen des BasicHandlers. Die Einbindung des Handlers in den Ablauf eines Web Services geschieht, indem beim deployen des Web Services mitgeteilt wird, dass der Service als <requestFlow> den spezifischen Handler benötigt:

```
<service name="TestService" provider="java:RPC">
  <requestFlow>
   <handler type="tracker"/>
  </requestFlow>

  <parameter name="className" value="LogService"/>
  <parameter name="allowedMethods" value="*"/>
 </service>
```

wobei der Handler unter dem Namen “tracker” ebenfalls deployed werden muss:

```
<handler name="tracker" type="java:MeinHandler">
  <parameter name="filename" value="Tracking.log"/>
 </handler>
```

Beispiel:
In diesem Beispiel definieren wir einen einfachen Handler für's Loggen der Zugriffe auf einen bestimmten Dienst. Dazu definieren wir einen Handler, welcher die Anzahl Zugriffe, mit Timestamp, in einer Log-Datei festhält.

Damit der Handler benutzt / aufgerufen wird, muss beim Deployen des Services der Handler angegeben werden:

```
<deployment name="LogHandler"
      xmlns="http://xml.apache.org/axis/wsdd/"
      xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
 <!-- logging handler Konfiguration -->
 <handler name="tracker" type="java:LogHandler">
  <parameter name="filename" value="Tracking.log"/>
 </handler>
 <!-- Service, der den log handler oben benutzt -->
 <service name="LogTestService" provider="java:RPC">
  <requestFlow>
   <handler type="tracker"/>
  </requestFlow>
  <parameter name="className" value="LogService"/>
  <parameter name="allowedMethods" value="*"/>
 </service>
</deployment>
```

Zuerst definieren wir den Handler:

```
public class LogHandler extends BasicHandler {
      public void invoke(MessageContext msgContext) throws AxisFault   {
            /** Log : jede Aktivierung wird gelogged. */
            try {
                  Handler serviceHandler = msgContext.getService();
                  String filename = (String)getOption("filename");
                  if ((filename == null) || (filename.equals("")))
                        throw new AxisFault("Server.NoLogFile",
                   "Es wurde kein Logfile für den LogHandler angegeben!",
                                                      null, null);
            FileOutputStream fos = new FileOutputStream(filename, true);
                  PrintWriter writer = new PrintWriter(fos);
                  Integer numAccesses =
                   (Integer)serviceHandler.getOption("zugriffe");
                  if (numAccesses == null)
                        numAccesses = new Integer(0);
                  numAccesses = new Integer(numAccesses.intValue() + 1);
                  Date date = new Date();
                  String result = date + ": service " +
                                    msgContext.getTargetService() +
                                    " Zugriff " + numAccesses + " Mal(e).";
                  serviceHandler.setOption("zugriffe", numAccesses);
                  writer.println(result);
                  writer.close();
            } catch (Exception e) {
                  throw AxisFault.makeFault(e);
            }
      }
}
```

Wen wir nun den Logging Service deployen und direkt im Browser testen:

http://localhost:8080/axis/services/LogTestService?method=testMethode

erhalten wir folgende XML Meldung

```
<?xml version="1.0" encoding="UTF-8" ?>
- <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <soapenv:Body>
- <testMethodeResponse
    soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <testMethodeReturn
    xsi:type="xsd:string">[LogService]testMethode()</testMethodeReturn>
   </testMethodeResponse>
   </soapenv:Body>
   </soapenv:Envelope>
```

Aufgrund des Deployment Descriptors muss nun beim Ausführen des `LogService` jeweils auch der `LogHandler` ausgeführt werden.

Der `LogHandler` legt nun eine Datei an, deren Namen im Deployment Descriptor für den Handler angegeben wurde:

```
<handler name="tracker" type="java:LogHandler">
  <parameter name="filename" value="Tracking.log"/>
 </handler>
```

Allerdings müssen Sie diese Datei etwas suchen:

- die Datei befindet sich im Verzeichnis `AXIS_HOME\bin`

Der Inhalt der Datei `Tracking.log` siehtfolgendermassen aus:

```
Tue Jan 13 21:18:10 CET 2004: service LogTestService Zugriff 1 Mal(e)
Tue Jan 13 21:40:46 CET 2004: service LogTestService Zugriff 2 Mal(e)
Tue Jan 13 21:40:48 CET 2004: service LogTestService Zugriff 3 Mal(e).
Tue Jan 13 21:40:49 CET 2004: service LogTestService Zugriff 4 Mal(e).
Tue Jan 13 21:40:50 CET 2004: service LogTestService Zugriff 5 Mal(e).
Tue Jan 13 21:40:50 CET 2004: service LogTestService Zugriff 6 Mal(e).
Tue Jan 13 21:40:51 CET 2004: service LogTestService Zugriff 7 Mal(e).
```

1.4.2.5. Remote Administration von Axis

Standardmässig muss Axis lokal administriert werden. Allerdings besteht die Möglichkeit Axis remote zu administrieren, wobei Sie dies aus Sicherheitsgründen kaum tun sollten.

Dazu müssen Sie lediglich die „`enableRemoteAdmin`“ Property des `AdminServices` auf `true` setzen. Diese Änderung können Sie direkt in der Konfigurationsdatei `server-config.wsdd` vornehmen. Sie finden diese Datei im `WEB-INF` Verzeichnis.

```
...
<service name="AdminService" provider="java:MSG">
  <parameter name="allowedMethods" value="AdminService"/>
  <parameter name="enableRemoteAdmin" value="false"/>
  <parameter name="className" value="org.apache.axis.utils.Admin"/>
  <parameter name="name="enableRemoteAdmin" value="true"/>
  <namespace>http://xml.apache.org/axis/wsdd/</namespace>
 </service>
...
```

1.4.3. Service Styles

In Axis unterscheidet man folgende Service Styles:

- RPC – der default Style in Axis
- Document
- Wrapped
- Message

1.4.3.1. RPC Services

RPC ist der Standard Servicetyp in Axis. Der Diensttyp wird im Attribut `provider="java:RPC"` festgehalten, im `<service...>` Tag

```
<deployment xmlns=http://xml.apache.org/axis/wsdd/
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
      <service name="urn:beispiel3" provider="java:RPC">
            <parameter name="className" value="Beispiel3" />
            <parameter name="allowedMethods" value="*" />
      </service>
</deployment>
```

Der Aufbau des RPC Services ist denkbar einfach:

```
      <service name=".. " provider="java:RPC">
            <parameter name=".." value=".." />
            ...
      </service>
```

Die Verarbeitung geschieht folgendermassen:

- Axis deserialisiert diese XML Darstellung in Java Objekt(e),
- Diese werden an den Service übergeben
- Der Rückgabewert wird von Axis wieder serialisiert.

1.4.3.2. Kommunikationsstile - Document und Wrapped

Die beiden Servicetypes „Document“ und „Wrapped“ sind ähnlich aufgebaut. Im Prinzip übermitteln Sie in diesen beiden Fällen reines XML. Allerdings stellt Ihnen Axis eine Java Darstellung (ein Java „Binding“) zur Verfügung.

Wrapped Style

In diesem Falle werden alle Datenfelder eines Aufrufes einzeln an die remote Methode übergeben.

Document Style

Die selbe Information, die im Wrapped Style quasi in Stücken übergeben wird, kann in diesem Style mit einem einzigen XML Dokument übermittelt werden.

Beispiele

```
<soap:Envelope    xmlns="http://xml.apache.org/axis/wsdd"
                  xmlns="http://xml.apache.org/axis/wsdd/providers/java">
            <soap:Body>
                  <OP:Bestellung xmlns:OP="http://intertrade.com/PO"/>
                        <BestellPosition>001</BestellPosition>
                        <Menge>12</Menge>
                        <Bezeichnung>ToblerOne</Bezeichnung>
            </soap:Body>
</soap:Envelope>
```

mit folgendem XML Schema:

```
<schema taregtNamespace="http://intertrade.com/PO">
      <complexType name="BestellungTyp">
            <sequence>
                  <element name="BestellPosition" type="xsd:string"/>
                  <element name="Menge" type="xsd:int">
                  <element name="Bezeichnung" type="xsd:string"/>
            </sequence>
      </complexType>
      <element name="Bestellung" type="BestellType"/>
</schema>
```

Falls der Dienst „*Document Style*“ verwendet, dann muss remote eine Methode mit folgender Signatur vorhanden sein:

```
public void method(Bestellung bs)
```

Die Bestellung wird also als Ganzes, in einem, an die Methode übergeben. Bestellung ist eine Java Bean, welche die oben im XML erkennbaren Datenfelder (Bestellposition, Menge und Bezeichnung) enthält.

Falls der Dienst „*Wrapped Style*“ verwendet, dann muss remote eine Methode mit folgender Signatur vorhanden sein:

```
public void method(String bestellPosition, int menge, String beschr)
```

Die Bestellung wird also im Datenfeldformat, zerlegt, an die Methode übergeben. `<BestellTyp>` ist ein Wrapper, welche die oben im XML erkennbaren Datenfelder (Bestellposition, Menge und Bezeichnung) umschliesst (wrapped).

In der WSDD werden die zwei Stile folgendermassen angegeben / unterschieden:

```
<service ... style="document">
```

beziehungsweise

```
<service ... style="wrapped">
```

1.4.3.3. Message Services

Im Falle des Message „*Service Styles*“ müssen Sie die XML verarbeitung übernehmen. Axis überlässt Ihnen die Verarbeitung vom XML.

Dieser Service Stil kennt folgende vier Service-Methoden Typen:

```
1) public Element[] method(Element[] bodies);
2) public SOAPBodyElement[] method(SOAPBodyElement[] bodies);
3) public Document method(Document body);
4) public void method(SOAPEnvelope request, SOAPEnvelope response);
```

Bei den ersten zwei Methoden wird, wie Sie sehen, ein Array von DOM oder SOAPBody Elementen übergeben (je ein Element pro XML Element im `<soap:body>`).

Die dritte Methode akzeptiert eine DOM Repräsentation des `<soap:body>` und erwartet genau so ein DOM Dokument als Rückgabe.

Die vierte Methode enthält je ein SOAP Envelope Objekt für die Anfrage und die Antwort. Diese Methode ist ideal geeignet, falls Sie Header-Informationen lesen und verändern müssen.

Message Beispiel

1) Der Message Service mit einer Signatur gemäss obigem Template 3):

```
public class MessageService {
```

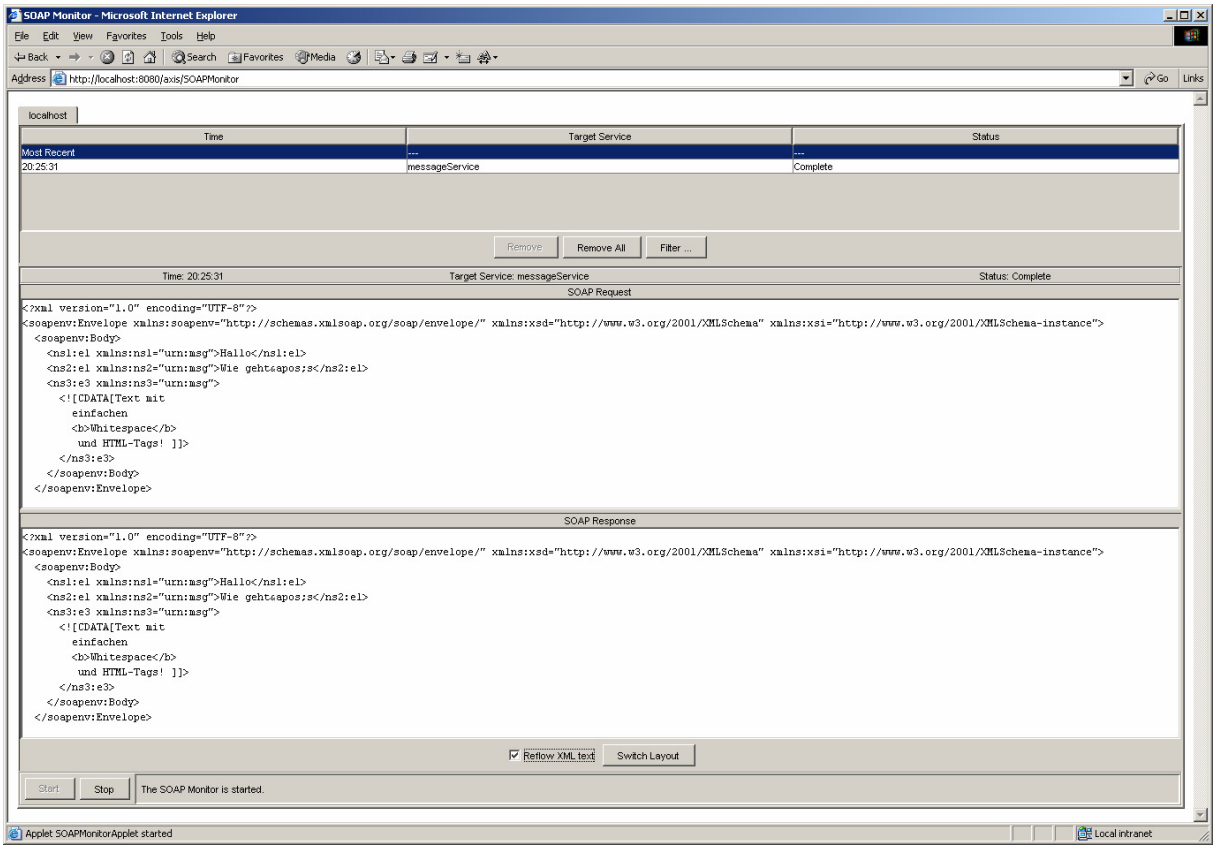
```
            /**
             * Echo Service Methode:
             *
             * @param elems : ein Array von DOM Elementen, je eines pro
                              SOAPBody Eintrag
             * @return ein DOM Elemente Array (für den Body der Response)
             */
            public Element[] echoElemente(Element[] elemsnte) {
                  return elemsnte;
            }
      }
```

2) Der Message Client

```
public class MessageClient {
      public String doit(String[] args) throws Exception {
            Options opts = new Options(args);
            opts.setDefaultURL(
                  "http://localhost:8080/axis/services/MessageService");

            Service service = new Service();
            Call call = (Call) service.createCall();

            call.setTargetEndpointAddress(new URL(opts.getURL()));
            SOAPBodyElement[] input = new SOAPBodyElement[3];

            input[0] =
                  new SOAPBodyElement(
                        XMLUtils.StringToElement("urn:msg", "e1", "Hallo"));
            input[1] =
                  new SOAPBodyElement(
                  XMLUtils.StringToElement("urn:msg", "e1", "Wie geht's"));

            DocumentBuilder builder =
            DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document doc = builder.newDocument();
            Element cdataElem = doc.createElementNS("urn:msg", "e3");
            CDATASection cdata =
                  doc.createCDATASection(
            "Text mit\n\teinfachen  <b>  Whitespace </b> und HTML-Tags!");
            cdataElem.appendChild(cdata);

            input[2] = new SOAPBodyElement(cdataElem);

            Vector elems = (Vector) call.invoke(input);//SOAPBodyElement
            SOAPBodyElement elem = null;
            Element e = null;

            elem = (SOAPBodyElement) elems.get(0);
            e = elem.getAsDOM();

            String str = "Res elem[0]=" + XMLUtils.ElementToString(e);

            elem = (SOAPBodyElement) elems.get(1);
            e = elem.getAsDOM();
            str = str + "Res elem[1]=" + XMLUtils.ElementToString(e);

            elem = (SOAPBodyElement) elems.get(2);
            e = elem.getAsDOM();
            str = str + "Res elem[2]=" + XMLUtils.ElementToString(e);
```

```
            return (str);
      }

      public static void main(String[] args) throws Exception {
            String res = (new MessageClient()).doit(args);
            System.out.println(res);
      }
}
```

3) und schliesslich der Traffic



Wie funktioniert das Ganze?
Axis besitzt einen Message Handler (`MsgHandler`). Dieser ruft die Methode

```
public Element[] echoElemente(Element[] elemsnte) {..}
```

auf, mit drei `org.w3c.dom.Element` Objekten, welche zum SOAP Body der eintreffenden Message gehören. Die Methode liefert dann die drei Elemente, die wir oben serialisiert als XML erkennen können
(MonitorApplet: http://localhost:8080/axis/SOAPMonitor).

Damit die Ausgabe im Applet angezeigt werden kann, muss der Request Flow wie üblich im WSDD angepasst werden.

Die WSDD sieht folgendermassen aus:

```
<deployment name="MessageService" xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
            xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
  <!-- zwei Varianten: style="message" OR provider="java:MSG" -->
  <service name="messageService" style="message">
      <requestFlow>
      <handler type="soapmonitor"/>
```

```
    </requestFlow>
    <responseFlow>
      <handler type="soapmonitor"/>
    </responseFlow>
    <parameter name="className" value="MessageService" />
    <parameter name="allowedMethods" value="echoElemente" />
  </service>
</deployment>
```

Wie Sie sehen, ist der Service vom Style Message.
Die (etwas abgeänderte) Ausgabe des Clients:

```
Elemente :
[0]<ns1:e1 xmlns:ns1="urn:msg">Hallo</ns1:e1>
[1]<ns2:e1 xmlns:ns2="urn:msg">Wie geht&apos;s</ns2:e1>
[2]<ns3:e3 xmlns:ns3="urn:msg"><![CDATA[Text mit
      einfachen  <b>  Whitespace </b> und HTML-Tags! ]]></ns3:e3>
```

mit

```
System.out.println("Elemente : ");
for (int i=0; i<elems.size(); i++) {
      elem = (SOAPBodyElement) elems.get(i);
      System.out.println("["+i+"]"+elem);
}
```

1.5. XML zu Java Data Mapping in Axis

Das Dilemma von SOAP sind die Datentypen:

- falls Sie die Möglichkeiten einer Programmiersprache voll ausnutzen, dann werden Sie vermutlich Probleme mit der Interoperabilität haben: Ihre SOAP Lösung wird nur noch für Server / Clients einsetzbar sein, welche diese Programmiersprache verwenden.
- In der Spezifikation JAX-RPC werden die grundsätzlichen Mappings aufgelistet, wobei aber einige Punkte offen sind.

Eine der wichtigsten Punkte der JAX-RPC Spezifikation ist die Definition von Standard Mappings für WSDL Dokumente (also die Service Beschreibung) auf Java Konstrukte (Service Endpoints, Stubs, Ties, Java Typen) und umgekehrt. Vor JAX-RPC definierte jeder Tool Anbieter sein eigenes Mapping, das Ganze war also äussers inoperable.

Die Standardisierung basiert auf WSDL 1.1 und SOAP 1.1, kann/muss also bei neuen Releases unter Umständen überarbeitet werden.

JAX-RPC befasst sich mit:

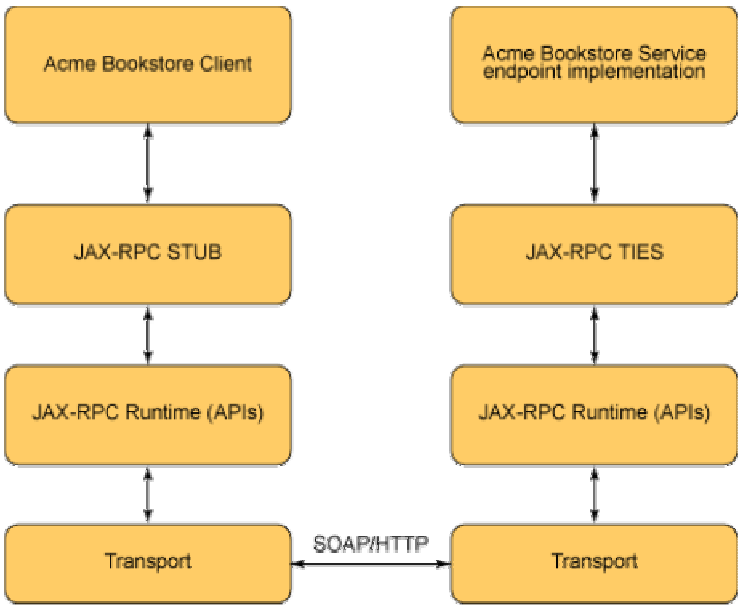
- dem Mapping von XML Datentypen auf Java Datentypen
- dem Mapping der abstrakten WSDL Definitionen (Port Types, Operations und Messages) auf Java Interfaces und Klassen
- Dem Mapping konkreter WSDL Definitionen (Ports, Bindings, Services) auf Java Klassen.

1.5.1.1. Java zu WSDL Standard-Mappings

xsd:base64Binary	`byte[]`
xsd:boolean	`boolean`
xsd:byte	`byte`
xsd:dateTime	`java.util.Calendar`
xsd:decimal	`java.math.BigDecimal`
xsd:double	`double`
xsd:float	`float`
xsd:hexBinary	`byte[]`
xsd:int	`int`
xsd:integer	`java.math.BigInteger`
xsd:long	`long`
xsd:QName	**`javax.xml.namespace.QName`**
xsd:short	`short`
xsd:string	`java.lang.String`

Das Beispiel xsd:QName zeigt, dass Java sich auch den Anforderungen von XML basierten Systemen anpassen muss.

JAX-RPC verwendet ein Kommunikationsmodell, welches aus folgender Skizze erkennbar ist:



JAX-RPC verwendet also explizit Stubs und Ties.
Die Spezifikation ist recht umfangreich (ca 160 Seiten) und definiert auch neue Entwicklungsmuster (Patterns) für Java Entwickler.

Unter anderem geht es in JAX-RPC um folgende Konzepte:

- Type-Mapping System
- Service Endpoints
- Exception Handling
- Service Endpoint Context
- Message Handlers
- Service Clients und Service Contexts
- SOAP mit Attachments
- Runtime Services
- JAX-RPC Client Invocation Modelle

1.5.1.1.1. Basisdatentypen

Die meisten einfachen XML Datentypen werden mithilfe von XML Schema und SOAP 1.1 definiert. Das Abbild auf Java Datentypen ist in Tabelle 4-1 (siehe Tabelle weiter oben) in der JAX-RPC Spezifikation festgehalten.

Es gibt aber einige Spezialfälle:

1) Nicht festgehalten ist die Abbildung von XML `xsd:anytype` auf Java.
2) Falls in einer Elemenr-Deklaration das `nillable` Attribut `true` gesetzt wird, dann wird der XML Datentyp auf die Java Wrapper Klasse zum Basisdatentyp abgebildet (`int` wird auf `Integer`, ... abgebildet)
   Beispiel:
   ```
   <xsd:element name="beispiel" type="xsd:int"
   nillable="true"/>
   ```

1.5.1.1.2. Komplexe Datentypen

XML Schema komplexe Datentypen werden auf Java Beans abgebildet, mit get/set Methoden, um auf die einzelnen Elemente / Komponenten des Datentyps zugreifen zu können.

Einige Punkte müssen dabei beachtet werden:

1) das Konstrukt `xsd:sequence` wird im Bean nicht abgebildet.
2) Das Attribut `maxOccurs` wird auf ein Java Array abgebildet.
   Aber es können nicht alle denkbaren Kombinationen von `minOccurs`, `maxOccurs` realisiert werden.
3) Die Konversion der Bean in eine WSDL Beschreibung ist nicht immer 1:1 möglich: JAX-RPC legt das Mapping von `xsd:attribute` nicht fest.

Beispiele

XML komplexer Datentyp

```
<xsd:complexType name="Artikel">
<sequence>
      <element name="Author" type="xsd:string" maxOccurs="10"/>
      <element name="AnzahlSeiten" type="xsd:int"/>
<sequence>
<xsd:attribute name="Reviewer" type="xsd:string"/>
</xsd:complexType>
```

und das dazugehörende Java Fragment:

```
public class Artikel {
```

```
      private float AnzahlSeiten;
      private String[] Author;
      private String Reviewer;

      // getter/setter
      public String[] getAuthor() {..}
      public setAuthor(String[] Author) {..}
      public float getAnzahlSeiten() {..}
      public void setAnzahlSeiten(int AnzahlSeiten) {..}
      public String getReviewer() {...}
      public void setReviewer(String Reviewer) {...}
}
```

1.5.1.1.3. Arrays

JAX-RPC bildet XML Arrays auf Java Arrays ab. Allerdings gibt es in einer WSDL Service Deklaration unterschiedliche Array Deklarationen.

Der erste Typ von Arrays wird aus `soapenc:Array` bzw. `wsdl:ArrayType` abgeleitet.
Beispiele

```
<complexType name="ArrayOfString">
<complexContent>
      <restriction base="soapenc:Array">
      <attribute ref="soapenc:ArrayType" wsdlArrayType="xsd:string[]">
      </restriction>
</complexContent>
</complexType>
```

bzw.

```
<complexType name="ArrayOfString">
<complexContent>
      <restriction base="soapenc:Array">
      <sequence>
        <element name="stringArray" type="xsd:string" maxOccurs="unbound">
      </sequence>
      </restriction>
</complexContent>
</complexType>
```

Beide Strukturen werden auf ein Java String-Array abgebildet.

Ein weiteres Beispiel zeigt ein Fragment eines WSDL Dokuments und eine Instanz dazu:

WSDL:

```
<element name="messDaten" type="soapenc:Array"/>
```

Eine Instanz davon könnte folgendermassen aussehen:

```
<messDaten soapenc:arrayType="xsd:int[2]">
      <number>12</number>
      <number>89</number>
</messDaten>
```

Diese deklaration wird auf ein Java `Object` Array abgebildet, wobei jedes Array Element einem XML Schema Typ (hier `integer`) entspricht.

1.5.1.1.4. Abbildung abstrakter WSDL Typen auf Java

Neben der Abbildung der XML Datentypen auf Java muss auch die Abbildung der WSDL Typen definiert werden. Auch dies ist Teil der JAX-RPC Spezifikation.

- **`wsdl:porttype`** : dieser Typ wird auf das Endpoint Interface abgebildet. Dieses erweitert das `java.rmi.Remote` Interface.
- **`wsdl:operation`** : der Name der Operation / Methode muss seit WSDL 1.1 nicht eindeutig sein (er kann wie in Java überladen sein: die Signatur muss eindeutig sein).

Parameter können in, out oder inout sein. Das Mapping ist entsprechend komplex und wird zum Teil über Holder / Hilfs-Klassen realisiert:

Parameter	Parameter Style	Mögliche JAX-RPC Mapping Klassen	JAX-RPC Mapping Class Beispiel
`wsdl:input`	`in`	Wrapper Klassen oder JavaBeans	`java.lang.Integer`
`wsdl:output`	`out`	`Holder` Class	`IntHolder`, `StringHolder`, etc.
`wsdl:input` und `wsdl:output`	`inout`	`Holder` Class	`IntHolder`, `StringHolder`, etc.

Der Inhalt der Holder Klassen wird durch remote Methodenaufrufe aktualisiert. Der Client kann auf diese Variablen zugreifen.

1.5.2. Exceptions

Die Behandlung von Ausnahmen auf eine interoperable Art und Weise, ist äusserst schwierig und zurzeit noch unvollständig.

1.5.2.1. `RMIRemoteExceptions` werden zu SOAP Faults

Gemäss JAX-RPC werden serverseitige RMI Exceptions auf SOAP Faults abgebildet. Der `faultcode` enthält den Klassennamen des Fehlers.

Falls Client und Server diese Klasse nicht gemeinsam haben, können Sie diese Meldung nicht mehr analysieren. Der kleinste gemeinsame Nenner ist in diesem Fall `java.rmi.RemoteException`.

Aber diese Möglichkeit haben Sie nur, falls Client und Server in Java geschrieben sind.

1.5.2.2. `Exception` werden zu `wsdl:fault`

Generelle Java `Exceptions` werden auf `wsdl:fault` abgebildet. JAX-RPC verlangt, dass eigene Exception Klassen Zugriffsmethoden besitzt, um auf die Objekt-Datenfelder zugreifen zu können. Zudem muss das Objekt einen Konstruktor besitzen, welcher alle Datenfelder als Parameter enthält. Die Parameter müssen Datentypen entsprechen, welche auf WSDL Datentypen abgebildet werden können.

1.5.3. Axis, SOAP und Interoperabilität

1.5.3.1. unsigned Datentypen

In C/C++/C# haben Sie die Möglichkeit „unsigned“ Datentypen einzusetzen (das Vorzeichenbit fehlt, der Datenbereich beschränkt sich auf positive Daten, doppelt so viele wie

bei „signed“ Datentypen). Diese Möglichkeit haben Sie in Java nicht. Falls Sie solche Datentypen einsetzen, können Java Clients nicht kontrolliert auf diese Services zugreifen!

1.5.3.2. Java Collections

Für einige Java Collections, beispielsweise `Hashtable`, existieren Serialisierer. Aber die Interoperabilität ist nicht gewährleistet. Wenn Sie Interoperabilität gewährleisten müssen, sollten Sie einfache Arrays einsetzen.

1.5.4. Übermitteln beliebiger Java Bean Klassen über SOAP

Damit Sie ein Java Objekt mit SOAP übertragen können, müssen Client und Server dieses analysieren können. Axis kennt das Konzept des Serializers / Deserializers. Sie haben die Möglichkeit eigene Serializer/Deserializer zu schreiben und in Axis zu registrieren.

Der Axis Bean Serializer gestattet das (de-)serialisieren von Java Bean Klassen. Sie teilen Axis (in XML) mit, welches Mapping (Java-XML Schema Typ) besteht.

Beispiel

```
<beanMapping qname="ns:local" xmlns:ns="meinNamensraum"
languageSpecificType="java:meinJavaTyp"/>
```

Das folgende Beispiel zeigt, wie eine Java Bean als Dienstklasse verwendet werden kann, sofern sie im Deployment Descriptor bekannt gegeben wird.

1) Die Bestell Service Klasse

```
public class BestellBeanService {
	public String bestellen(Bestellung bestellung) {
		String sep = System.getProperty("line.separator");
		String response =
			"[BestellBeanService]Hallo "
				+ bestellung.getKundenName()
				+ "!"
				+ sep;
		response += sep + "[BestellBeanService]Ihre Bestellung:" + sep;
		String[] items = bestellung.getBestellCodes();
		int[] quantities = bestellung.getAnzahl();
		for (int i = 0; i < items.length; i++) {
			response += sep
				+ " Anzahl : "
				+ quantities[i]
				+ " Artikel : "
				+ items[i];
		}
		response += sep
			+ sep
			+ "Ihre Kreditkarteninfo "
			+ "... fehlt (leider)!";
		return response;
	}
}
```

2) Die Bestellungsklasse

```
public class Bestellung {
	/** Kunde  */
	private String kundenName;
	/** Lieferadresse  */
	private String lieferAdresse;
```

```
	/** Artikel */
	private String bestellCodes[];
	/** Menge */
	private int anzahl[];

	// Bean Zugriffsmethoden
	public int[] getAnzahl() {
		return anzahl;
	}
	public String[] getBestellCodes() {
		return bestellCodes;
	}
	public String getKundenName() {
		return kundenName;
	}
	public String getLieferAdresse() {
		return lieferAdresse;
	}
	public void setAnzahl(int[] is) {
		anzahl = is;
	}
	public void setBestellCodes(String[] strings) {
		bestellCodes = strings;
	}
	public void setKundenName(String string) {
		kundenName = string;
	}
	public void setLieferAdresse(String string) {
		lieferAdresse = string;
	}
}
```

3) Der Client

```
public class BestellClient {
	public static void main(String[] args) throws Exception {
		Options options = new Options(args);
		Bestellung bestellung = new Bestellung();
		bestellung.setKundenName("Jim Knopf");
		bestellung.setLieferAdresse("Bahnhofstrasse 1, Zugwil, ZG");
		String[] items = new String[] { "Märklin", "ICE" };
		int[] quantities = new int[] { 1, 4 };
		bestellung.setBestellCodes(items);
		bestellung.setAnzahl(quantities);
		Service service = new Service();
		Call call = (Call) service.createCall();
		QName qn = new QName("urn:BestellBeanService", "Bestellung");
		call.registerTypeMapping(
			Bestellung.class,
			qn,
			new org.apache.axis.encoding.ser.BeanSerializerFactory(
				Bestellung.class,
				qn),
			new org.apache.axis.encoding.ser.BeanDeserializerFactory(
				Bestellung.class,
				qn));
		String result;
		try {
			call.setTargetEndpointAddress(new
					java.net.URL(options.getURL()));
			call.setOperationName(
				new QName("BestellAnnahme", "bestellen"));
			call.addParameter("arg1", qn, ParameterMode.IN);
```

```
                call.setReturnType(
                      org.apache.axis.encoding.XMLType.XSD_STRING);
                result = (String) call.invoke(new
                                        Object[] { bestellung });
            } catch (AxisFault fault) {
                result = "Error : " + fault.toString();
            }
            System.out.println(result);
      }
}
```

Der Deployment Descriptor beschreibt die Klasse und die Klassenmethoden:

```
      <deployment
            xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
          <service name="BestellAnnahme" provider="java:RPC">
              <parameter name="className" value="BestellBeanService"/>
              <parameter name="allowedMethods" value="bestellen"/>
              <beanMapping qname="meinNS:Bestellung"
                        xmlns:meinNS="urn:BestellBeanService"
              languageSpecificType="java:Bestellung"/>
          </service>
      </deployment>
```

Als Ausgabe erhalten wir:

```
[BestellBeanService]Hallo Jim Knopf!
[BestellBeanService]Ihre Bestellung:
 Anzahl : 1 Artikel : Märklin
 Anzahl : 4 Artikel : ICE
Ihre Kreditkarteninfo ... fehlt (leider)!
```

1.5.5. Custom Serialisierung – falls Beans nicht ausreichen

Die Bean Serialisierung, die wir gerade angeschaut haben, reicht in vielen Fällen nicht aus, speziell wenn es darum geht, bestehende Klassen einzubinden oder spezielle XML Schema Typen auf Java abzubinden.

In diesem Fall müssen Sie eigene Serializer / Deserializer bauen und anschliessend deployen.

1.5.5.1. Der <typeMapping> Tag

Falls Sie einen Serializer / Deserializer haben, zusammen mit den dazugehörigen Factories, dann müssen Sie Axis das Type Mapping mitteilen.

Dazu verwenden Sie den `<typeMapping>` Tag im WSDD:

```
<typeMapping qname="ns:local"
      xmlns:ns="meinNamespace"
      languageSpecificType="java:package.meineJavaKlasse"
      serializer="package.MeineSerializerFactory"
      deserializer="package.MmeineDeserilizerFactory"
      encodingStyle=http://schemas.xmlsoap.org/soap/encoding
/>
```

Wichtig

In den zwei Attributen serializer/deserializer wird die *Factory*, nicht die Serializer/ Deserializer Klasse angegeben.

Der Bean Mapping Tag ist ein Spezialfall:

```
serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
```

Die können Ihre Factory nach dem Muster dieser Klasse konstruieren.
Auf dem Server finden Sie ein Beispiel für einen Serializer/Deserializer und deren Factories (aus dem Apache Axis User Guide) inklusive Skripts zum Testen, Deployen und UnDeployen.

Die Serializer Factory ist denkbar einfach aufgebaut:

```
public class DataSerializerFactory implements SerializerFactory {
	private Vector mechanisms;

	public DataSerializerFactory() {
	}
	public javax.xml.rpc.encoding.Serializer
		getSerializerAs(String mechanismType) {
			return new DataSerializer();
	}
	public Iterator getSupportedMechanismTypes() {
		if (mechanisms == null) {
			mechanisms = new Vector();
			mechanisms.add(Constants.AXIS_SAX);
		}
		return mechanisms.iterator();
	}
}
```

Der Serializer ist der eigentliche Arbeiter:

```
public class DataSerializer implements Serializer {
	public static final String STRINGMEMBER = "stringMember";
	public static final String FLOATMEMBER = "floatMember";
	public static final String DATAMEMBER = "dataMember";
	public static final QName myTypeQName = new QName("typeNS", "Data");

	/**
	 *
	 * Serialisiert ein Element "name", mit Attributen und Werten
	 *
	 * @param name          Name des Elements
	 * @param attributes    Attribute
	 * @param value         Wert
	 * @param context       SerializationContext
	 */
	public void serialize(
		QName name,
		Attributes attributes,
		Object value,
		SerializationContext context)
		throws IOException {
		if (!(value instanceof Data))
			throw new IOException(
				"Die Klasse "
					+ value.getClass().getName()
		+ " kann mit dem DataSerializer nicht serialisiert werden.");
		Data data = (Data) value;
		context.startElement(name, attributes);
		context.serialize(new QName("", STRINGMEMBER), null,
                                                    data.stringMember);
		context.serialize(new QName("", FLOATMEMBER), null,
                                                   data.floatMember);
```

```
            context.serialize(new QName("", DATAMEMBER), null,
                                                    data.dataMember);
            context.endElement();
      }
      public String getMechanismType() {
            return Constants.AXIS_SAX;
      }
      public Element writeSchema(Class javaType, Types types)
                                                    throws Exception {
            return null;
      }
}
```

Der eigentliche Service zum Testen dieser Hilfsklassen ist der Element Service; die Testclients sind Element Client und Serializer Client.