8 Anwendungsentwicklung mit Axis

8.1 Einführung

In den vorangegangenen Kapiteln wurden bereits erste Möglichkeiten aufgezeigt, wie man die Programmierung von Web-Service-Anwendungen mit Axis angehen könnte. Auf den folgenden Seiten soll dies nun intensiviert werden. Zunächst wird es, aufbauend auf dem in Kapitel 5 vorgestellten Web Service `SimpleDVDStore`, um die Integration eigener komplexer Datentypen in Web-Service-Anwendungen gehen. In diesem Zusammenhang werden *Serializer* und *Deserializer* vorgestellt, mit denen Anwendungsentwickler über einen Mechanismus verfügen, der Einfluss auf den Konvertierungsvorgang von Java-Datentypen nach XML und zurück gewährt. Weiterhin wird detaillierter auf die Entwicklung von Clients eingegangen. In Kapitel 5 wurde neben dem Web Service bereits ein einfacher Client vorgestellt, der hier nochmals aufgegriffen wird, um die Klasse `Call`, die Axis zum Aufruf von Web Services zur Verfügung stellt, genauer unter die Lupe zu nehmen. Ein weiteres Thema wird die Behandlung von Fehlern im Web Service und im Client sein. In diesem Zusammenhang werden die in der Axis-Distribution enthaltenen Tools TCPMonitor und SOAPMonitor vorgestellt, mit denen sich die von einer Web-Service-Anwendung versendeten SOAP-Nachrichten, zur Laufzeit anzeigen und analysieren lassen. Daran anschließend wird erläutert, wie Session-Informationen verwaltet werden können, was es mit den so genannten Parameter Modes auf sich hat und wie in der Axis-Distribution enthaltene Tools zur Codegenerierung verwendet werden können, um den Entwicklungsaufwand zu reduzieren.

8.2 Der komplexe Datentyp Movie

Jeder Anwendungsentwickler wird in ernsthaften Anwendungen irgendwann an einen Punkt kommen, an dem die Basisdatentypen nicht mehr ausreichen. Die Programmiersprache Java kennt neben den Basisdatentypen daher auch komplexere Datentypen und Datenstrukturen. Hierzu zählen beispielsweise

Arrays, Klassen der Java Collection API und eigene, anwendungsspezifische Datentypen. Mit diesen Konstrukten ist es möglich, Daten effizienter zu speichern, zu verarbeiten und weiterzugeben. Der DVDStore zum Beispiel könnte zu einem Film neben dem eigentlichen Titel noch weitere Informationen wie Beschreibung, Produktionsjahr und Verfügbarkeit speichern. Sollen diese Daten nun per Web Service abrufbar sein und würde man diesen mit den Basisdatentypen realisieren, die Java zur Verfügung stellt, so bräuchte man im einfachsten Falle für jede abzurufende Information eine eigene Methode:

```
public String getMovieTitle(int  movieId)
public String getMovieDescription (int movieId)
public int  getMovieYear (int movieId)
public boolean isReserved (int movieId)
...
```

Neben der Tatsache, dass diese Vorgehensweise schnell unübersichtlich werden kann, liegt der größte Nachteil hier mit Sicherheit in der Anzahl an Web-Service-Aufrufen, die nötig wären, um einen kompletten Datensatz mit allen Filminformationen zu erhalten. In einem solchen Fall bietet sich daher der Einsatz einer speziellen, anwendungsspezifischen Datenstruktur an. In Datenstrukturen können Daten, die in einer Beziehung zueinander stehen, zusammengefasst, bearbeitet und zwischen Methoden ausgetauscht werden. Auf diese Art ließe sich für den DVDStore eine Funktion

```
public Movie getMovieDetails(int movieId)
```

codieren, die alle oben aufgelisteten Funktionen in einem Aufruf in sich vereint und eine Datenstruktur `Movie` zurückgibt, in der alle Filminformationen zu einem handlichen Paket zusammengeschnürt sind.

Diese Datenstruktur `Movie` wird in Java in einer Klasse abgebildet – üblicherweise in einem JavaBean, das im einfachsten Fall aus einer bestimmten Anzahl Properties besteht, die nach außen über entsprechende `get`- und `set`-Methoden ansprechbar sind. So lässt sich auf einfache Weise eine Klasse konstruieren, in der alle Filminformationen abgelegt und abgerufen werden können. Das folgende Listing zeigt, wie eine solche `Movie`-Klasse aussehen könnte. Dabei ist `Movie` genau genommen gar kein JavaBean, da es die Schnittstelle `Serializable` nicht implementiert. Axis drückt bei diesem

Kriterium jedoch ein Auge zu. Wir werden diese Klasse daher im weiteren Verlauf ebenfalls als Bean ansehen.

```
package de.javamagazin.store;
public class Movie {
  private String title;
  private float pricePerDay;
  private int year;
  private String description;
  private String actors;
  private boolean reserved;

  public Movie() {
    title = "not yet set";
    pricePerDay = 0.0f;
  }

  public Movie(String title, float pricePerDay) {
    this.title = title;
    this.pricePerDay = pricePerDay;
  }
  public Movie(String title, int year,
               float pricePerDay) {
    this.title = title;
    this.year = year;
    this.pricePerDay = pricePerDay;
  }

  public Movie(String title, float pricePerDay, int year,
               String actors, String description) {
    this.title = title;
    this.pricePerDay = pricePerDay;
    this.year = year;
    this.description = description;
    this.actors = actors;
  }
```

```
public String getTitle() {
  return title;
}
public void setTitle(String title) {
  this.title = title;
}

public String getDescription() {
  return description;
}

public void setDescription(String description) {
  this.description = description;
}

public String getActors() {
  return actors;
}

public void setActors(String actors) {
  this.actors = actors;
}
public int getYear() {
  return year;
}

public void setYear(int year) {
  this.year = year;
}

public float getPricePerDay() {
  return pricePerDay;
}
```

```
  public void setPricePerDay(float pricePerDay) {
    this.pricePerDay = pricePerDay;
  }

  public boolean isReserved() {
    return reserved;
  }

  public void setReserved(boolean reserved) {
    this.reserved = reserved;
  }
}
```

Dieses Implementierungsbeispiel bietet durch mehrfach überladene Konstruktoren die Möglichkeit, ein `Movie`-Objekt schon beim Instanzieren mit Daten vorzubelegen.

8.3 Komplexe Datentypen einsetzen

In Kapitel 5 wurde mit dem `SimpleDVDStore` bereits ein erster, sehr einfacher Web Service vorgestellt. Dieser speicherte lediglich Filmtitel in einem Array und war mit der Funktion `getMovieTitle` in der Lage, einen dieser Titel zurückzugeben. Als Weiterentwicklung dieses zugegebenermaßen noch recht schlanken Web Service soll nun die `Movie`-Klasse zur Speicherung von Filminformationen zur Anwendung kommen. Ferner sollen zusätzliche Methoden entstehen, in denen dieser komplexe Datentyp verwendet wird.

```
package de.javamagazin.store;

public class DVDStore {

  private Movie[] movies =
    { new Movie("The Lord of the Rings",2.00f, 2001,
                "Elijah Wood, Ian McKellen", ""),
```

```
      new Movie("Cube", 1.75f, 1997,
                "Maurice Dean Wint, Andrew Miller", ""),
      new Movie("Star Wars Episode II", 1.50f, 2002,
                "Ewan McGregor, Natalie Portman", ""),
      new Movie("Lola rennt", 2.00f, 1998,
                "Franka Potente, Moritz Bleibtreu", "")
    };

  public Movie getMovieDetails(int movieId) {
    if (movieId < movies.length) {
      return movies[movieId];
    } else {
      return null;
    }
}
  public String[] getMovieTitles() {
    String[] titles = new String[movies.length];

    for(int i=0; i<movies.length; i++) {
      titles[i] = movies[i].getTitle();
    }
    return titles;
  }
}
```

Zunächst wurde das ursprünglich mit Strings gefüllte Array durch ein Array des Datentyps `Movie` ersetzt. Zum Zeitpunkt der Instanzierung des `DVDStore` wird das Array mit einer Reihe von `Movie`-Objekten befüllt und diese mit einigen Filminformationen vorbelegt. In einer realen Anwendung würde man diese Informationen zum Beispiel aus einer Datenbank laden.

Die ursprüngliche Methode `getMovieTitle` wurde durch die neue Methode `getMovieDetails` ersetzt, die angeforderte `Movie`-Objekte aus dem Array entnimmt und zurückgibt. Die neue Methode `getMovieTitles` dient hingegen der Rückgabe eines String-Arrays, das eine Liste aller Filmtitel enthält.

8.3.1 Notwendige Konfigurationen auf Seiten des Service

Beim Aufruf einer Web-Service-Methode müssen bestimmte Mechanismen dafür Sorge tragen, dass beispielsweise die Rückgabewerte einer aufgerufenen Methode in entsprechende XML-Datentypen umgewandelt werden, um sie im Body der SOAP-Antwort zum Client transportieren zu können. Auf Client-Seite muss ebenfalls ein spezieller Mechanismus dafür sorgen, dass diese XML-kodierten Rückgabewerte in die entsprechenden Datentypen der verwendeten Programmiersprache zurückgewandelt werden. Die Umwandlung von Objekten nach XML wird Serialisierung genannt, der umgekehrte Weg Deserialisierung (alternative Begriffe sind Marshalling und Unmarshalling). Diese Aufgabe übernehmen in Axis spezielle Klassen: die so genannten *Serializer* und *Deserializer*. Für gewisse Basisdatentypen enthält Axis bereits vordefinierte Standard-Serializer und -Deserializer. Auch für die etwas komplexeren Konstrukte wie zum Beispiel Arrays bringt es entsprechende Unterstützung mit. Dabei befolgt es die Abbildungsregeln der JAX-RPC-Spezifikation, die in Kapitel 7.2 im Detail erläutert wurden. Solange bei der SOAP-Kommunikation nur diese Basisdatentypen als Methodenparameter und Rückgabewerte verwendet werden, sorgt Axis mit Hilfe dieser Regeln und der mitgelieferten Serializer und Deserializer ganz automatisch für deren Umwandlung zwischen XML und Java.

Für die Serialisierung und Deserialisierung eines Datentyps wird ein Quadrupel an Informationen benötigt: der Name der Java-Klasse, der qualifizierte Name des zugehörigen XML-Datentyps sowie die Klassennamen des Serializers und des Deserializers, die für die Umwandlung verantwortlich sind. Ein solches Quadrupel wird Type Mapping genannt; alle ihm bekannten Type Mappings verwaltet Axis intern in einer Type Mapping Registry. Tiefergehende Informationen zu diesen Konzepten befinden sich im separaten Kapitel 12 sowie in Kapitel 9.5.

Während der `SimpleDVDStore` noch Basisdatentypen verwendete, die Axis automatisch umwandeln konnte, wird es in der erweiterten Version des Web Service mit dem dort verwendeten `Movie`-Objekt schon etwas komplizierter. Die Serialisierung des String-Arrays, das von `getMovieTitles` zurückgegeben wird, stellt noch kein Problem dar, da sich hier die in der Axis-Distribution mitgelieferten `ArraySerializer` und `SimpleSerializer` automatisch um die Serialisierung kümmern. Im Falle der Methode `getMovieDe-`

tails, die ein Movie-Objekt zurückgibt, ist eine Serialisierung dagegen nicht ohne weiteres möglich, weil Axis den anwendungsspezifischen Datentyp Movie nicht kennt. Er muss Axis daher mit Hilfe eines neuen Type Mappings bekannt gemacht werden – sowohl auf dem Client als auch auf Seiten des Service. Außerdem müssen geeignete Serializer und Deserializer für die Klasse Movie gefunden werden. Notfalls sind diese zu programmieren.

Glücklicherweise enthält Axis jedoch auch einen Standard-Serializer und zugehörigen Deserializer für JavaBeans. Es handelt sich dabei um die Klassen BeanSerializer und BeanDeserializer. Sie sind in der Lage, beliebige Java-Klassen, die den JavaBean-Konventionen folgen, zu (de-) serialisieren. Da das Movie-Objekt diesen Konventionen entspricht (abgesehen von der oben erwähnten Ausnahme), können beide für diesen Datentyp verwendet werden. Axis muss allerdings explizit mitgeteilt werden, dass Instanzen der Klasse Movie mit dem BeanSerializer verarbeitet werden sollen. Beim Deployment des Web Service ist daher im WSDD mit dem Element <typeMapping> eine zusätzliche Konfiguration erforderlich.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/
                                             providers/java">

 <service name="DVDStore" provider="java:RPC">
   <parameter name="className"
              value="de.javamagazin.store.DVDStore"/>
   <parameter name="allowedMethods"
              value="getMovieDetails getMovieTitles"/>

   <typeMapping qname="myNS:Movie"
                xmlns:myNS="http://store.javamagazin.de"
       languageSpecificType=
                "java:de.javamagazin.store.Movie"
       serializer="org.apache.axis.encoding.ser.
                                  BeanSerializerFactory"
       deserializer="org.apache.axis.encoding.ser.
                                BeanDeserializerFactory"
```

```
        encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/"
    />
 </service>
</deployment>
```

Der in diesem Zusammenhang interessante Teil, nämlich die Anwendung des `<typeMapping>`-Elementes, ist fett dargestellt. Dieses Element dient dazu, Axis das bereits erwähnte Quadrupel aus Java-Klasse (Attribut `languageSpecificType`), dem qualifizierten Namen des zugehörigen XML-Datentyps (Attribut `qname`) sowie Serializer und Deserializer bekannt zu machen. Da die Klasse `Movie` ein JavaBean ist, konnten die Klassen `BeanSerializer` und `BeanDeserializer` ausgewählt werden. Bei genauem Blick fällt jedoch auf, dass im WSDD nicht diese Klassen selbst, sondern stattdessen zwei Factory-Klassen aufgeführt sind. Dies liegt darin begründet, dass je nach eingesetzter Kodierungsvorschrift (Encoding Style) unterschiedliche Serializer-Klassen oder unterschiedlich parametrisierte Instanzen zum Einsatz kommen können. Die zu verwendende Kodierungsvorschrift wird mit dem Attribut `encodingStyle` festgelegt, nähere Informationen zum Encoding befinden sich in Kapitel 2.2.5.

Besondere Sorgfalt sollte bei der Auswahl des qualifizierten Namens für die XML-Darstellung komplexer Datentypen walten. Theoretisch kann der qualifizierte Name beliebig gewählt werden. Dennoch gibt es gewisse Regeln, die man zur Vermeidung von Komplikationen einhalten sollte. So ist es empfehlenswert, den Local Part des qualifizierten Namens mit dem Namen der Java-Klasse gleichzusetzen. Im vorliegenden Beispiel lautet der Local Part also *Movie*. Der zugehörige Namensraum sollte gebildet werden, indem der Zeichenkette *http://* der vollständige Package-Name der Klasse in umgekehrter Reihenfolge angehängt wird. Das Package der Klasse `Movie` hat den Namen `de.javamagazin.store`, somit ergibt sich der Namensraum *http://store.javamagazin.de*. Weiterhin ist grundsätzlich darauf zu achten, dass bei der Bekanntmachung eines Datentyps in Client-Anwendungen genau der gleiche qualifizierte Name verwendet wird, der auch im WSDD definiert wurde.

Im Falle von Klassen, die keine JavaBeans sind und für die Axis auch keine automatische Umwandlung durch andere mitgelieferte Serializer und Deseri-

alizer bietet, müssen andere gefunden und deren jeweils zugehörige Factory-Klassen in die Attribute `serializer` und `deserializer` eingetragen werden. Lassen sich für eine Java-Klasse jedoch keine Serializer/Deserializer finden, weder unter den in der Axis-Distribution mitgelieferten noch in den Weiten des Internets, so müssen diese vom Anwendungsentwickler erstellt werden. Man spricht dann von so genannten Custom-Serializern. Auf die Programmierung von eigenen Serializern und deren Einbindung in Web Services wird in Kapitel 12 „Type Mapping Framework & benutzerdefinierte Serializer" im Detail eingegangen.

Ausschließlich für den speziellen Fall, dass für die Umwandlung eines Datentyps `BeanSerializer` und `BeanDeserializer` zum Einsatz kommen sollen, gibt es die Möglichkeit einer vereinfachten Schreibweise der obigen Konfiguration. Anstelle des Elementes `<typeMapping>` kann dann das Element `<beanMapping>` verwendet werden. Es handelt sich dabei im Grunde lediglich um eine Kurzform, bei der die Angabe der Factory-Klassen entfällt. Stattdessen zeigt der Elementname `beanMapping` an, dass `BeanSerializer` und `BeanDeserializer` gemeint sind. Der Vorteil dieses Elements liegt also einzig in dessen Kürze.

```
<beanMapping qname="myNS:Movie"
           xmlns:myNS="http://store.javamagazin.de"
 languageSpecificType="java:de.javamagazin.store.Movie"/>
```

Somit wurde der komplexe Datentyp `Movie` auf Seiten des Service bekannt gemacht und festgelegt, welche Serializer/Deserializer für dessen Umwandlung in/aus XML zuständig sind. Es ist jedoch unbedingt notwendig, die gleichen Informationen auch in der Client-Anwendung zu hinterlegen. Wie dies bewerkstelligt werden kann und welche Einstellungen dort außerdem noch möglich sind, wird im folgenden Abschnitt beschrieben.

8.3.2 Notwendige Programmierschritte auf dem Client

Im Falle von Basisdatentypen sorgt Axis auch auf dem Client automatisch dafür, dass die richtigen Serializer und Deserializer verwendet werden. Da dies bei anwendungsspezifischen komplexen Datentypen wie `Movie` aller-

dings nicht zutrifft, müssen für diese – ähnlich wie auf der Serverseite – entsprechende Type Mappings registriert werden. Zur Registrierung eines Type Mappings auf dem Client dienen mehrere Methoden der Klasse `Call`, die alle den Namen `registerTypeMapping` tragen, jedoch unterschiedliche Methodenparameter erwarten. Das folgende Listing demonstriert, welche Erweiterungen für den Einsatz komplexer Datentypen vorzunehmen sind.

```
package de.javamagazin.store;

import org.apache.axis.AxisFault;
import org.apache.axis.Handler;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
// BeanDeserializerFactory and BeanSerializerFactory
import org.apache.axis.encoding.ser.*;

import java.net.URL;
import javax.xml.rpc.encoding.XMLType;
import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;

public class DVDStoreClient {

  public static void main(String [] args) {
    try {
      String endpoint =
      "http://localhost:8080/axis/services/DVDStore";

      Service service = new Service();
      Call call = (Call) service.createCall();
      call.setTargetEndpointAddress(new URL(endpoint));

      // get ALL movie titles
      // --> demonstration of using SOAP arrays
      call.setOperationName("getMovieTitles");
```

```
String[] titles=
   (String[]) call.invoke(new Object[] {} );

for (int i=0; i<titles.length; i++) {
  System.out.println("Movie " + i + ": " +
                     titles[i]);
}

// get movie details for movie #2
// --> demonstration of using complex data types
call.removeAllParameters();
call.setOperationName("getMovieDetails");
call.addParameter("movieId", XMLType.XSD_INT,
                             ParameterMode.IN);
QName qnMovie = new QName
       ("http://store.javamagazin.de", "Movie");
call.setReturnType(qnMovie);

call.registerTypeMapping(Movie.class, qnMovie,
new BeanSerializerFactory(Movie.class, qnMovie),
new BeanDeserializerFactory(Movie.class, qnMovie));

Movie movieDetails =
(Movie) call.invoke(new Object[] {new Integer(1)});


System.out.println("\nDetails for movie #2:");
System.out.println("----------------------------");
System.out.println("Title: " +
                   movieDetails.getTitle());
System.out.println("Year: " +
                    movieDetails.getYear());
System.out.println("Actors: " +
                    movieDetails.getActors());
```

```
      System.out.println("Description: " +
                          movieDetails.getDescription());
      System.out.println("Price/Day: " +
                          movieDetails.getPricePerDay());

    } catch (Exception e) {
      System.out.println("An exception occurred.");
      e.printStackTrace();
    }
  }
}
```

Zunächst wird, wie im ursprünglichen Client auch, ein `Call`-Objekt erzeugt, über dessen Methoden die weitere Verarbeitung abgewickelt wird. Als Erstes wird diesem der Service-Endpoint mitgeteilt, dann mit `setOperationName` die Web-Service-Methode angegeben, die aufgerufen werden soll – im Beispiel also zunächst `getMovieTitles`. Da `getMovieTitles` keinerlei Parameter erwartet, wird der Methode `invoke` ein leeres Object-Array übergeben. Der Rückgabewert des Web Service muss in ein String-Array umgewandelt werden. Anschließend kann dessen Inhalt in einer `for`-Schleife ausgegeben werden.

Bis hier war noch keine Angabe von Serializern nötig, da sowohl für die Serialisierung von Strings als auch von Arrays entsprechende Standard-Serializer zur Verfügung stehen und sich Axis automatisch um deren (De-)Serialisierung kümmert. Dies ändert sich beim Aufruf der Methode `getMovieDetails`. Das bereits existierende Call-Objekt kann erneut verwendet werden, jedoch sollten mit Hilfe von `removeAllParameters` eventuelle Methodenparameter und Rückgabewerte entfernt werden, die darin noch vom ersten Aufruf gespeichert sind. Anschließend kann mit `setOperationName` der Name der nun aufzurufenden Operation gesetzt werden.

Die Methode `addParameter` dient dazu, die für den Web Service bestimmten Methodenparameter näher zu spezifizieren. So kann festgelegt werden, welcher Elementname verwendet werden soll, um einen Parameter in der Request-Nachricht zu repräsentieren (im Beispiel `movieID`), um welchen XML-Datentyp es sich dabei handelt und ob es ein IN-, OUT- oder INOUT-Parameter ist (diese so genannten Parameter Modes werden in Abschnitt 8.9

näher erläutert). Für die Angabe des XML-Datentyps wird im Beispiel die Klasse XMLType verwendet. In ihr sind für alle Basisdatentypen die zugehörigen qualifizierten Namen als Konstanten zusammengefasst.

Die Verwendung von addParameter ist in diesem Beispiel optional. Lässt man den Aufruf weg, besorgt sich Axis den qualifizierten Namen des XML-Datentyps aus der Type Mapping Registry, nimmt für alle Parameter den Parameter Mode IN an und verwendet als Elementnamen für die Parameter im SOAP-Request die Bezeichnungen <arg0>, <arg1>, <arg2> usw. Verwendet werden sollte addParameter also insbesondere immer dann, wenn ein oder mehrere Parameter nicht IN-Parameter, sondern OUT- bzw. IN-OUT-Parameter sind oder wenn die explizite Vorgabe von Elementnamen gewünscht wird. Zusätzlich gilt folgende Regel: Wenn addParameter aufgerufen wurde, muss auch die ansonsten ebenfalls optionale Methode setReturnType aufgerufen werden, mit der Axis der qualifizierte Name des Methodenrückgabewertes bekannt gemacht wird.

Bei diesem Rückgabewert handelt es sich im Beispiel also um den komplexen Datentyp Movie. Für diesen Datentyp wird zunächst ein QName-Objekt erzeugt, welches den qualifizierten Namen des zugehörigen XML-Datentyps repräsentiert. Dabei ist unbedingt darauf zu achten, dass Namensraum und Name des Datentyps ganz genau der serverseitig vorgenommenen Konfiguration im WSDD entsprechen. Andernfalls wird die Umwandlung scheitern, da Axis dann nicht erkennen kann, dass es sich um die gleichen Datentypen handelt und sie stattdessen für verschieden hält. Dieses QName-Objekt ist setReturnType zu übergeben.

Mit registerTypeMapping wird zum Schluss schließlich der Type Mapping Registry von Axis bekannt gemacht, wie der anwendungsspezifische Datentyp Movie zu (de-)serialisieren ist. Hierzu werden der Methode genau die gleichen vier Informationen übergeben, die auch im <typeMapping>-Element des WSDD spezifiziert wurden: die Klasse Movie, das soeben erzeugte QName-Objekt mit dem qualifizierten Namen des zugehörigen XML-Datentyps sowie die zu verwendenden Factoryklassen für die zuständigen Serializer und Deserializer.

Durch Aufruf der Methode invoke wird der SOAP-Request auf die Reise geschickt. Dieses Mal wird invoke kein leeres Array übergeben, sondern eines, das mit einem Integer-Objekt gefüllt ist, da ja der Index des Films

übergeben werden muss, dessen Detailinformationen angefordert werden. Die Web-Service-Methode ist zwar so definiert, dass der Datentyp `int` erwartet wird; da `invoke` aber andererseits ein `Object`-Array erwartet, muss die Wrapper-Klasse verwendet werden.

8.4 Mehr über die Klasse Call

Wie in den vorausgegangenen Beispielen und auch aus Kapitel 7 deutlich wurde, ist `Call` die zentrale Klasse für das Aufrufen von Web Services mit JAX-RPC-kompatiblen Frameworks wie Axis. Nach seiner Erzeugung wird ein Objekt dieser Klasse mit den nötigen Informationen wie beispielsweise Zieladresse, Methodenname und Methodenparameter befüllt. Neben den bisher demonstrierten Einstellungen kann das `Call`-Objekt noch mit einer Reihe weiterer `set`-Methoden konfiguriert werden. So kann beispielsweise mit `setOperationStyle` oder `setOperationUse` Einfluss darauf genommen werden, welcher Nachrichtenstil und welches Encoding für den SOAP-Request verwendet werden sollen (nähere Informationen über Nachrichtenstile befinden sich in Kapitel 13). Werden diese Einstellungen – wie in den bisherigen Beispielen – nicht vorgenommen, so verwendet Axis die jeweilige Standardvoreinstellung. Im Falle des Nachrichtenstils und des Encodings ist dies RPC/Encoded. Tabelle 8.1 enthält die wichtigsten `set`-Methoden der Klasse `Call`. Alternativ zum manuellen Setzen der jeweiligen Werte kann das `Service`-Objekt, welches das `Call`-Objekt erzeugt, mit Hilfe der Klasse `ServiceFactory` und der WSDL-Beschreibung des aufzurufenden Web Service initialisiert werden, sodass das `Call`-Objekt automatisch mit den darin enthaltenen Informationen befüllt wird. In Kapitel 7.3.2 befindet sich ein Code-Beispiel, das diese Vorgehensweise verdeutlicht.

Nachdem das `Call`-Objekt also befüllt wurde, kann schließlich durch Aufruf der Methode `invoke()` die Erzeugung und der Versand der SOAP-Nachricht zum Web Service initiiert werden. Der Aufruf von `invoke()` bewirkt allerdings nicht den unmittelbaren Versand der SOAP-Nachricht. Stattdessen legt Axis die SOAP-Nachricht nach deren Erzeugung zunächst in einem Objekt der Klasse `MessageContext` ab, welches außerdem mit einer Vielzahl zusätzlicher Informationen über den bevorstehenden SOAP-Request befüllt wird. In diesem zusätzlichen Objekt sind alle für den Web-Service-Aufruf

relevanten Informationen zusammengefasst. Der `MessageContext` wird dann intern an eine Reihe weiterer Verarbeitungsschritte übergeben. Welche Verarbeitungsschritte dies genau sind, kann anwendungsspezifisch konfiguriert werden. Erst am Ende dieser Reihe von Verarbeitungsschritten erfolgt der eigentliche Versand der SOAP-Nachricht. Nähere Informationen zum `MessageContext` und der internen Verarbeitung von Web-Service-Aufrufen befinden sich im Kapitel 9 „Architektur und Konfiguration“.

Methode	**Beschreibung**
`setTargetEndpoint-Address`	Definiert die Adresse des Service-Endpoints.
`setOperationName`	Legt den Namen der aufzurufenden Web-Service-Methode fest.
`addParameter`	Konfiguriert Methodenparameter für den Web-Service-Aufruf durch Angabe von Name, XML-Datentyp und Parametermodus (IN, OUT, INOUT). Der Parameter wird im SOAP-Body eingefügt. Diese Methode ist optional; wird sie verwendet, muss laut JAX-RPC-Spezifikation auch die Methode `setReturnType` aufgerufen werden.
`removeAllParameters`	Entfernt alle konfigurierten Parameter aus einer `Call`-Instanz und initialisiert sie damit für einen weiteren Aufruf.
`addParameterAsHeader`	Wie `addParameter`, fügt den Parameter jedoch in den SOAP-Header ein.
`setProperty`	Ermöglicht das Setzen von Properties, die bei der Verarbeitung der SOAP-Nachricht von clientseitigen Handlern ausgelesen werden können (Handler werden in den Kapiteln 9 und 10 ausführlich beschrieben).

Methode	Beschreibung
`removeProperty`	Entfernt die übergebene Property aus dem Call-Objekt.
`setUsername` `setPassword`	Übergibt Benutzername bzw. Passwort an das `Call`-Objekt, mit denen sich Axis beispielsweise beim Einsatz von HTTP Basic Authentication am Web-Server authentifizieren kann.

`setSOAPVersion`	Ermöglicht das Umschalten der SOAP-Version durch Übergabe einer Klasse mit entsprechenden Konstanten. Gültige Werte:
	`SOAPConstants.SOAP11_CONSTANTS`
	`SOAPConstants.SOAP12_CONSTANTS`
	Die Klasse SOAPConstants stammt aus dem Package `org.apache.axis.soap`.

`setOperationUse` `setOperationStyle`	Mit diesen beiden Methoden kann der gewünschte Nachrichtenstil für den Web-Service-Aufruf eingestellt werden. Werden diese Methoden nicht explizit verwendet, so kommt die Standardeinstellung RPC/Encoded zum Einsatz. Nachrichtenstile werden im Kapitel 13 ausführlich behandelt.
`setEncoding`	Diese optional aufzurufende Methode setzt das Encoding; die unterschiedlichen Encodings werden durch URLs identifiziert.
`invoke`	Ruft den Web Service synchron mit Request-Response-Verfahren auf.

Tabelle 8.1: Die wichtigsten Methoden des Call-Objekts

8.5 Fehlerbehandlung

Die Behandlung von Fehlern ist auch bei der Entwicklung von Web-Service-Anwendungen von großer Wichtigkeit. Im folgenden Abschnitt soll daher gezeigt werden, wie gezielt auf das Erzeugen von SOAP-Faults Einfluss genommen werden kann.

8.5.1 Axis Faults verwenden

Mit der Methode `getMovieDetails` können im `DVDStore` Filminformationen abgerufen werden. Wird ein Film abgerufen, den es nicht gibt, so sollte die Methode nicht `null` zurückgeben wie in den bisherigen Beispielen, sondern eine anwendungsspezifische Exception werfen, wie zum Beispiel `NoSuchMovieException`.

```
public Movie getMovieDetails(int movieId)
    throws NoSuchMovieException {

    if (movieId < movies.length) {
      return movies[movieId];
    } else {
    throw new NoSuchMovieException("Film gibt es nicht");
    }

}
```

Die Klasse `NoSuchMovieException` erbt von `RemoteException`.

```
public class NoSuchMovieException
 extends RemoteException {
    public NoSuchMovieException( String message) {
        super(message);
    }
}
```

Nachdem die Methode `getMovieDetails` die Exception geworfen hat, wandelt Axis diese in einen SOAP-Fault um, wie nachfolgender Ausschnitt aus einer SOAP-Response zeigt:

```
<soapenv:Fault>
  <faultcode>soapenv:Server.userException</faultcode>
  <faultstring>de.javamagazin.store.NoSuchMovieException:
   Film gibt es nicht
  </faultstring>
  <detail/>
</soapenv:Fault>
```

Das Element `faultcode` wird dabei immer mit der Fehlerkategorie `Server` und der Unterkategorie `userException` gefüllt, `faultstring` enthält den Klassennamen der Exception, die in der Web-Service-Methode aufgetreten ist. Das Element `detail` bleibt leer, sofern der Axis-Server nicht im „Development-Modus" betrieben wird (siehe Kapitel 9.9).

Der Client sollte Exceptions, die während eines Web-Service-Aufrufs auftreten können, natürlich fangen und entsprechend reagieren. Sie werden in Form eines `AxisFault` gemeldet; dabei handelt es sich um eine Klasse, die von `RemoteException` abgeleitet ist. Folgender Code-Ausschnitt zeigt eine Erweiterung des obigen Clients, in der die Klasse der gefangenen Exception und die darin enthaltene Nachricht ausgegeben wird.

```
...
} catch (AxisFault af) {
  System.out.println("Message: " + af.getMessage());
} catch (Exception e) {
  // catch all other exceptions here…
  System.out.println("Exception: " + e.getClass());
  System.out.println("Message: " + e.getMessage());
}
```

In der Nachricht des `AxisFault` ist als Text jeweils der Klassenname der tatsächlich aufgetretenen Exception sowie deren Nachricht enthalten.

```
Message: de.javamagazin.store.NoSuchMovieException: Film
gibt es nicht
```

Alternativ dazu können anwendungsspezifische Exceptions auch in der Clientanwendung gefangen werden, anstatt diese in einen `AxisFault` kap-

seln zu lassen. Hierzu ist die Exception-Klasse genau wie ein anwendungsspezifischer Datentyp auch auf Client- und Serverseite mit Hilfe eines Type Mapping bekannt zu machen.

8.6 Werkzeuge zur Anzeige von SOAP-Nachrichten

Oftmals benötigen Entwickler Hilfsmittel zur Anzeige und Analyse von SOAP-Nachrichten, die eigene Web Services erzeugen und verarbeiten. Um beobachten zu können, welche SOAP-Nachrichten zwischen einer Client-Anwendung und einem Web Service oder auch zwischen mehreren Web Services ausgetauscht werden, bringt die Axis-Distribution zu diesem Zweck zwei Werkzeuge mit: TCPMonitor und SOAPMonitor. Sie können sehr hilfreich beim Debugging sein. Die nachfolgenden Abschnitte erläutern die Funktionsweise und die praktische Anwendung dieser hilfreichen Tools.

8.6.1 TCPMonitor

Bei TCPMonitor handelt es sich um eine sehr einfache Anwendung, die in die Mitte eines Kommunikationskanals eingeklinkt wird, sodass sämtliche Nachrichten hindurchgeleitet werden. Abbildung 8.1 illustriert diesen Ablauf. Die Client-Anwendung schickt ihre Nachrichten normalerweise direkt an einen Web Service. Nach dem Start von TCPMonitor ist als Erstes die Client-Anwendung dahingehend zu ändern, dass sie ihre Nachrichten fortan nicht mehr an den Web Service direkt, sondern stattdessen an TCPMonitor schickt. Weiterhin muss TCPMonitor mitgeteilt werden, wohin es die empfangenen SOAP-Nachrichten weiterleiten soll: natürlich an den eigentlichen Empfänger – den Web-Container, auf dem der Service installiert ist.

Während der Entwicklung wird es häufig vorkommen, dass Client-Anwendung, Web Service und TCPMonitor auf ein und demselben Rechner laufen. In diesem Fall muss TCPMonitor auf einem anderen Port gestartet werden als der Web-Container und seine Nachrichten jeweils an `localhost` weiterleiten. Ebenso kann TCPMonitor natürlich auch verwendet werden, wenn Client-Anwendung und Web Service auf verschiedenen Rechnern laufen. In diesem Fall kann TCPMonitor entweder auf dem Client-Rechner oder auf

dem Server laufen. Daneben ist es auch immer möglich, TCPMonitor auf einem dritten Rechner zu starten.

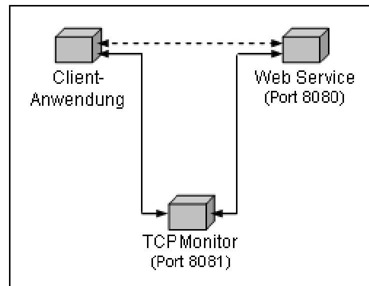

Abb. 8.1: Der TCPMonitor in der Mitte einer SOAP-Kommunikation

Der Start von TCPMonitor erfolgt auf der Kommandozeile mit dem Befehl

```
java org.apache.axis.utils.tcpmon
```

Daraufhin startet eine grafische Benutzeroberfläche, in der zunächst eingestellt werden muss, auf welchem Netzwerkport TCPMonitor lauschen und an welchen Empfänger es die eingehenden Nachrichten weiterleiten soll. Diese Angaben können alternativ auch über die Kommandozeile mitgegeben werden. So startet beispielsweise der Befehl

```
java org.apache.axis.utils.tcpmon 8081 localhost 8080
```

ein TCPMonitor, das auf Port 8081 lauscht und alle Nachrichten auf den Port 8080 desselben Rechners weiterleitet. TCPMonitor kann auch als Proxy agieren und eine langsame Verbindung simulieren, indem es empfangene Nachrichten nur verzögert weiterleitet.

Sind die nötigen Einstellungen vorgenommen, wird der Button ADD betätigt. Es erscheint ein neuer Reiter in TCPMonitor, dessen Titel den Port anzeigt, auf dem TCPMonitor von nun an lauscht. Dieser Vorgang kann beliebig oft wiederholt werden, wenn auf mehreren Ports gleichzeitig gelauscht werden soll.

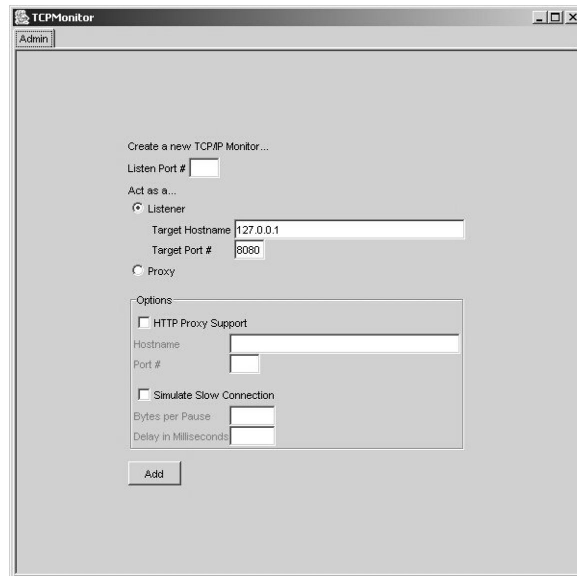

Abb. 8.2: Einstellungsmöglichkeiten im TCPMonitor

Nun können Clientanwendungen ausgeführt werden, die entsprechend den obigen Ausführungen geändert wurden. Jeden einzelnen abgefangenen Nachrichtenumlauf stellt TCPMonitor in einer Liste am oberen Rand des Panels für den jeweiligen Port dar. Darunter wird sowohl der jeweils zugehörige SOAP-Request als auch die vom Web Service zurückgeschickte SOAP-Response angezeigt. Mit dem Button SAVE können abgefangene Nachrichten gespeichert werden, SWITCH LAYOUT zeigt Request- und Response-Nachrichten nebeneinander statt untereinander an. Ein Aktivieren der Checkbox XML FORMAT bewirkt, dass TCPMonitor alle abgefangenen Nachrichten formatiert bzw. einrückt. Dies verbessert deren Lesbarkeit stellenweise erheblich.

Von besonderem Nutzen ist schließlich der Button RESEND. Er erlaubt es, abgefangene Nachrichten erneut zu versenden – gegebenenfalls können sie zuvor noch editiert werden. Somit ist es möglich, manuelle Tests auf sehr einfache Weise durchzuführen, indem man wiederholt Kleinigkeiten an ei-

nem SOAP-Request ändert und deren Auswirkung auf den Web Service und die SOAP-Response testet. Dies lässt sich mit Hilfe des RESEND-Buttons oft deutlich schneller und einfacher erledigen als mit der Client-Anwendung selbst. Nicht zuletzt dient TCPMonitor somit neben dem Debugging auch dazu, eigene Erfahrungen mit SOAP zu machen – also zum Lernen.

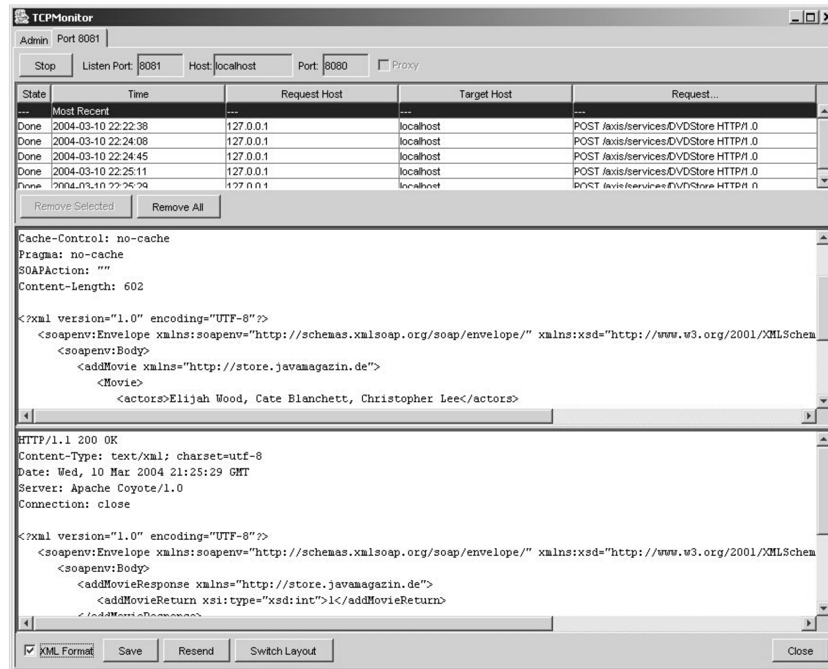

Abbildung 8.3: Beobachtung der SOAP-Kommunikation mit dem TCPMonitor

8.6.2 SOAPMonitor

Während sich TCPMonitor direkt zwischen Web Service und Client hängt und neben der eigentlichen SOAP-Nachricht auch den HTTP-Header ausgibt, geht SOAPMonitor einen gänzlich anderen Weg. Das erklärte Ziel seiner

Entwickler war es, eine Möglichkeit zu schaffen, SOAP-Nachrichten anzuzeigen, ohne dass hierdurch eine spezielle Konfiguration externer Tools, wie beispielsweise bei TCPMonitor, erforderlich ist. Zu diesem Zweck wurde ein spezieller Handler entwickelt, der in die globale Handlerkette eingeklinkt werden kann (mehr zu Handlern in Kapitel 10 „Handler und Chains“). Dieser Handler leitet SOAP-Nachrichten an den SOAPMonitor-Service weiter, der schließlich als Schnittstelle zur Anzeige dient. Die Anzeige erfolgt über ein Applet, das im Browser über eine URL wie beispielsweise

```
http://localhost:8080/SOAPMonitor
```

gestartet werden kann. Dieses Applet kommuniziert über eine Socketverbindung mit dem SOAPMonitor-Service und zeigt die Nachrichten schließlich an. Zur einwandfreien Ausführung des Applets im Webbrowser muss das Java-Plug-In 1.3 oder höher installiert sein.

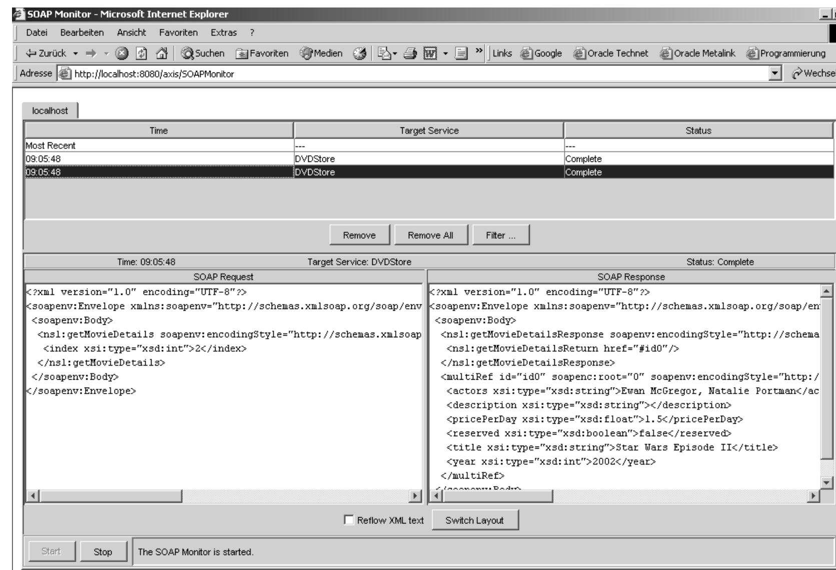

Abb. 8.4: Das SOAPMonitor-Applet in Aktion

Die Socketverbindung zum SOAPMonitor Service wird standardmäßig über Port 5001 hergestellt. In der für die Axis-Webanwendung zuständigen Konfigurationsdatei *web.xml* lässt sich dies bei Bedarf umstellen. SOAPMonitor kann nicht für Web Services verwendet werden, die über JWS installiert wurden, und ist aus Sicherheitsgründen nach einer Axis-Installation standardmäßig deaktiviert.

Um SOAPMonitor zu benutzen, sind ein paar Vorarbeiten nötig. Axis liefert das Applet zur Anzeige der Nachrichten ausschließlich im Quelltext mit, welcher unter `$AXIS_HOME/SOAPMonitorApplet.java` zu finden ist. Nachdem aus diesem Verzeichnis der Java-Compiler wie folgt aufgerufen wurde, sollten sich einige *.class*-Dateien im Wurzelverzeichnis der Axis-Installation befinden:

```
javac –classpath WEB-INF/lib/axis.jar
SOAPMonitorApplet.java
```

Im nächsten Schritt müssen der bereits angesprochene Handler und der SOAP-Monitor-Service in Axis eingebunden werden. Dies bewerkstelligt man am einfachsten mit dem AdminClient und folgendem WSDD:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/
                                           providers/java">
 <handler name="soapmonitor"
          type="java:org.apache.axis.handlers.
                SOAPMonitorHandler">
  <parameter name="wsdlURL"
             value="/axis/SOAPMonitorService-impl.wsdl"/>
  <parameter name="namespace"
             value="http://tempuri.org/wsdl/2001/12/
                          SOAPMonitorService-impl.wsdl"/>
  <parameter name="serviceName"
             value="SOAPMonitorService"/>
  <parameter name="portName" value="Demo"/>
 </handler>
```

```
 <service name="SOAPMonitorService" provider="java:RPC">
   <parameter name="allowedMethods"
     value="publishMessage"/>
   <parameter name="className"
     value="org.apache.axis.monitor.SOAPMonitorService"/>
    <parameter name="scope" value="Application"/>
  </service>
</deployment>
```

Damit wurde das SOAPMonitor-Utility aktiviert. Um aber nun tatsächlich SOAP-Nachrichten anzeigen zu können, muss der Web Service, der beobachtet werden soll, angewiesen werden, den SOAPMonitor-Handler zu durchlaufen. Durch Verwendung von `<requestFlow>` und `<responseFlow>` direkt nach dem `<service>`-Tag kann im Deployment Descriptor des gewünschten Web Service der SOAPMonitor-Handler eingebunden werden:

```
...
  <service name="DVDStore" provider="java:RPC">
    <requestFlow>
      <handler type="soapmonitor"/>
    </requestFlow>
    <responseFlow>
      <handler type="soapmonitor"/>
    </responseFlow>

    <parameter name="className"
      value="de.javamagazin.store.DVDStore"/>
    ...
```

Nachdem der modifizierte Deployment Descriptor über AdminClient eingespielt wurde, wird in diesem Beispiel der DVDStore sämtliche SOAP-Nachrichten an das SOAPMonitor-Applet weiterleiten.

Das Applet listet jetzt jede Kommunikation auf; mit Klick auf einen Listeneintrag werden die Nachrichten, die in dieser Kommunikation ausgetauscht wurden, angezeigt (Request und Response). Die Checkbox REFLOW XML

TEXT sorgt bei Aktivierung für eine lesbare Anordnung der Tags in den angezeigten SOAP-Nachrichten.

8.7 Verwaltung von Sessions

Nicht selten kommt es vor, dass in einer Anwendung die Verwaltung von Sessions benötigt wird. Damit ist gemeint, dass Zustände oder Informationen über mehrere Aufrufe hinweg gespeichert und vor allen Dingen dem jeweiligen Client wieder zugeordnet werden können. Axis verfügt auch zur Lösung dieser Problematik über einen Mechanismus, der zudem offen und erweiterbar gestaltet wurde.

Im Package `org.apache.axis.session` befindet sich die Schnittstelle `Session`, die definiert, welche Methoden eine Klasse unterstützen muss, wenn sie zur Speicherung von Session-Informationen dienen soll. Dies sind insbesondere Methoden zum Speichern und späteren Auslesen von Objekten einer Session (unter Zuhilfenahme eines Schlüssels). Weiterhin kann ein Timeout gesetzt und mit Hilfe der Methode `touch` die Session „berührt“ werden, um einen baldigen Timeout zu verhindern. Mit `invalidate` kann eine Session schließlich explizit ungültig gemacht werden.

Für diese Schnittstelle existieren in Axis zwei Implementierungen: die Klasse `SimpleSession` befindet sich ebenfalls im Package `org.apache.axis.session` und stellt eine Implementierung auf der Basis einer einfachen Hash Table zur Verfügung; `AxisHttpSession` ist dagegen im Package `org.apache.axis.transport.http` zu finden und legt stattdessen ein `HTTPSession`-Objekt aus dem Servlet API zugrunde. Der Einsatz des letzteren bietet sich damit an, wenn HTTP als Transportprotokoll verwendet wird, da hiermit Servlet-Session und Axis-Session zusammengeführt werden.

...