Asynchrone Webservices mit Axis 1.x in Java

1. Übersicht

Architektur

Da Webservices nach relativ kurzen Timeouts Anfragen abgearbeitet haben müsse, sind komplexe Anfragen – wie sie in der Bioinformatik üblich sind – praktisch nicht möglich. Dieses Java-Paket stellt ein Framework für Dokument-basierte, asynchrone Webservices bereit. Es basiert auf den HOBIT Empfehlungen und teilt die eigentlich asynchone Anfrage für den Benutzer unsichtbar in drei synchrone Ausrufe auf. Dazu wird der eigentlich Request, an Endpoint übermittelt und eine Id zugeteilt. Anschließend wird von dieser Id regelmäßig der Status abgefragt, wobei die HOBIT Statuscodes verwendet werden. Ist die Anfrage abgearbeitet oder liegt ein Fehler vor wird die letztendlich als Ergebnis an den Benutzer weitergereicht. Dazu ist sowohl eine Client- als auch eine Server-Komponente erforderlich. Die Client-Komponente "verwandelt" die asynchrone Anfrage wie beschrieben in synchrone Aufrufe. Die Server-Komponente hat die Aufgabe, diese synchronen Anfragen engegenzunehmen, in einer Datenbank zwischenzuspeichern und abzuarbeiten. Sie stellt zusätzlich ein einfaches Interface für die Erweiterung um eigene Nutz-methoden zur Verfügung.

Anforderungen

Für den Betrieb ist Server-seitig Axis (getestet mit Version 1.1 und 1.3) und eine Datenbank (getestet mit MySQL 4.0) erforderlich.

Pakete

Für den Client sind folgende externen Pakete erforderlich und müssen vor Betrieb heruntergeladen werden: log4j-1.2.8+, xml-apis (ab JDK 1.5 enthalten), jdom-1.0+, xercesImpl, Axis-Bibliotheken. Für den Server zusätzlich mysql-connector-java-3.0.9+ (für MySQL, sonst entsprechend) und Tools (Paket enthalten) erforderlich und müssen im Classpath des Webservers liegen.

2. Server

Implementieren eines Webserices

Um einen (bestehenden) Webservice asynchron zu machen, ist es in erster Linie nur notwendig, die Klasse AsynchronousCallHandler zu erweitern. Alle öffentlichen Methoden dieser Klasse mit org.w3.dom.Document als Ein- und Ausgabeparameter werden dann bei entsprechendem Aufruf durch den Client asynchron behandelt:

```
public class Example extends AsynchronousCallHandler {
        public synchronized org.w3c.dom.Document wait(org.w3.dom.Document rqst) throws Exception {
                int time = Integer.parseInt(rqst.getDocumentElement().getAttribute("time"));
                wait(time);
                return rqst;
        }
}
```

Zur Konfiguration sind folgende Methoden verfügbar: setLogger(Class, Logger) setzt für die Klasse

Class einen org.apache.log4j.Logger, auf dem Logging-Informationen ausgegben werden. Per default wird hier der mit der Klasse AsynchronousCallHandler asoziierte Logger gesetzt. Mit setValidation(Class, boolean) kann beeinflusst werden, ob die Anfragen eines Webservices einer bestimmten Klasse validiert werden. Dies ist aktiviert, sofern es nicht geändert wird. Dabei wird nur der Teil der Anfragen validiert, der an den Namespace http://hobit.sourceforge.net/xsds/hobitAsyncWS.xsd gebunden ist, nicht aber der Inhalt des mitgeschickten Dokuments. Mit setAsynchronous(Class, List) kann man für eine Klasse eine Liste von Methodennamen übergeben, die als asynchron behandelt werden. Alle anderen Methoden stehen nicht asynchron zur Verfügung. Wird diese Methode nicht aufgerufen werden die geeigneten Methoden (also public, mit org.w3c.dom.Document als Ein- und Ausgabeparameter) mit Reflection ermittelt.

Die beschriebenen Methoden sollten in einem static-Konstruktor des Webservices entsprechend aufgerufen werden:

```
static {
        setLogger(Example.class, Logger.getLogger(Example.class);
        setValidation(Example.class, true);
        List methods = new ArrayList();
        methods.add("wait");
        setAsynchronous(Example.class, methods);
}
```

Konfiguration der Umgebung

Im Hintergrund läuft ein Server, der die eingegangenen Anfragen aus der Datenbank lädt und abarbeitet. Der Server wird statisch beim ersten Zugriff auf den Webservice (z.B. bei Abfrage des wsdl). Der Server loggt in einen org.apache.log4j.Logger, der – wie üblich – über das erste im Classpath gefundene File log4j.properties konfiguriert wird (Beispiel im Paket enthalten). Die Einstellungen des Servers können über die Datei server.properties konfiguriert werden, die im Classpath liegen muss. Folgende Einstellungen sind möglich:

```
SERVER_POLLING_INTERVAL=10000  // alle ~ ms in DB neue Jobs suchen
THREAD_COUNT=10                // Anzahl der Jobs, die gleichzeitig laufen
```

Der Server kann auch auf *einem* anderen Rechner gestartet (und gestoppt) werden. Dazu muss man auf diesem Rechner "java -jar AsyncWS.jar (start|stop)" ausgeführt werden. Dazu müssen die oben angegebenen Pakete schon im Verzeichnis lib/ liegen.

In der Datei DbManager.xml (die auch im Classpath liegen muss) wird der Zugriff auf die benötigte Datenbank eingestellt. Folgende Zeilen sind dafür wichtig und müssen an die eigene Umgebung angepasst werden. Über den $dbdriver$ und $dbtype$ kann die verwendete Datenbank eingestellt werden (die Klasse muss im Classpath liegen, für MySQL ist das mysql-connector-java-3.0.9+.jar). Die angegebene Datenbank muss für den Benutzer mit Schreib und Lock Grants angelegt werden, Tabellen werden automatisch erzeugt.

```
<property><name>($dbdriver$)</name><value>com.mysql.jdbc.Driver</value></property>
<property><name>($dbtype$)</name><value>mysql</value></property>
```

```
<property><name>($user$)</name><value>asyncws</value></property>
<property><name>($password$)</name><value>mypassword</value></property>
<property><name>($host$)</name><value>localhost</value></property>
<property><name>($port$)</name><value>3306</value></property>
<property><name>($database$)</name><value>profi</value></property>
```

Die Schemas hobitStatuscode.xsd und hobitAsyncWS.xsd sind im Paket enthalten und müssen im Classpath des Webservice liegen. Diese werden für die Validierung der eingehenden Requests an den asychronen Webservice benötigt.

Konfiguration des Webservice

Zu einem Webservice gehört als Beschreibung ein wsdl. Für alle abgeleiteten asynchronen Webservices ist im Paket ein Template “AsynchronousCallHandlerTemplate.wsdl” enthalten, in dem die geerbten und zur asynchronen Betrieb notwendigen Typen, Messages und Operationen definiert sind. Hier muss in erster Linie der Target Namespace sowie eigene Namespaces gesetzt werden (xmlns:tns="http://ExampleNS", xmlns:example=”http://myExample”), eigene Typen definiert und/oder importiert werden (<xsd:import namespace="http://myExample" schemaLocation="myexample.xsd"/>) sowie für Request und Response Elemente für die neu definierten Methoden erstellt werden

```
<message name="waitRequest">
	<part name="request" type="example:waitType"/>
</message>
<message name="waitResponse">
	<part name="request" element="example:waitType"/>
</message>
```

und die Operationen in Port und Binding definiert werden:

```
<portType name="ExampleType">
	<operation name="wait">
		<input message="tns:waitRequest"/>
		<output message="tns:waitResponse"/>
	</operation>
	...
<binding name="ExampleBinding" type="tns:ExampleType">
	<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
	<operation name="wait">
		<input>
			<soap:body use="literal"/>
		</input>
		<output>
			<soap:body use="literal"/>
		</output>
	</operation>
	...
```

Am Schluss muss noch der Port des Service auf die engültige URL gesetzt werden. Hört sich anstrengend an, ist aber in fünf Minuten erledigt!

Neben den üblichen Einstellungen für den Webservice müssen die Methoden des Webservice nun

im Webservice Deployment Descriptor (server-config.wsdd) entsprechend eingestellt werden:

```
<!-- Web Service Administration Service -->
<service name="ProMLResolver" provider="java:MSG" use="literal" style="document">
        <parameter name="allowedMethods" value="*"/>
        <wsdlFile>/Example.wsdl</wsdlFile>
        <parameter name="className" value="de.lmu.ifi.bio.hobit.profi.Webservice.Example"/>
</service>
```

Hier ist wichtig, dass in allowedMethods auf jeden Fall die Methoden isAsynchronous, getAsynchronous, getStatus und getResult enthalten ist, da diese vom AsynchronousCallHandler geerbt und vom Client aufgerufen werden. Für die Konfiguration als Message basierten Webservice müssen unbedingt provider auf “java:MSG” und style auf “document” gesetzt werden. Außerdem sollte das wie oben beschrieben erzeugte wsdl-File statisch erzeugt und angegeben werden. Wie üblich muss der Klassenname gesetzt werden.

3. Client

Für die Abfrage eines asynchronen Webservices ist die Klasse `AsynchronousCall` vorgesehen. Sie erweitert die Klasse org.apache.axis.Call von Axis, sodass die gewohnten Funktionen zur Verfügung stehen. So kann man einen Asynchron-Call mit einem Endpoint konstruieren, und dann einen Request an den Webservice schicken:

```
org.w3c.dom.Document request = ...
SOAPBodyElement input[] = new SOAPBodyElement[1];
input[0] = new SOAPBodyElement(request.getDocumentElement());
String endpoint = “http://services.bio.ifi.lmu.de/localhost:8084/ProMLResolver/services/ProMLResolver”;
AsynchronousCall ac = new AsynchronousCall(endpoint);
Vector response = ac.invoke(input[]);
org.w3c.dom.Document result = ((SOAPBodyElement)response.get(0)).getAsDocument();
```

Dabei pollt die Methode invoke() intern solange, bis der Aufruf vom Server abgearbeitet ist und kehrt erst dann wieder zurück! Wie häufig gepollt wird kann über setPollingInterval(int interval) in Millisekunden gesetzt werden.

Alternativ dazu ist es auch möglich, den Request “zu Fuß” abzusetzen, den Status zu pollen und das Ergebnis abzuholen. Dafür sind folgende Methoden vorhanden:

```
public boolean isAsynchronous(String methodname);                  // ist eine bestimmte Method asynchron?
public String getAsynchronous(org.w3c.dom.Element request);        // setzt Request ab und gibt Id zurück
public int getStatus(String id);                                   // fragt den Status einer Id ab
public org.w3c.dom.Element getResult(String id);                   // ruft das Ergebnis zu einer Id ab
```

Bei den Werten des Typs org.w3c.dom.Element in den Methoden getAsynchronous() / getResult() handelt es sich um die Root-Elemente der Anfrage/des Ergebnisses. Alle Methoden werfen eine Exception wenn der Aufruf nicht korrekt abgearbeitet werden konnte. Die Methode getResult() wirft eine InvocationTargetException, wenn der eigentliche Methodenaufruf ein Fehler zurücklieferte. Der zurückgegebene Status der Methode getStatus() richtet sich nach der Hobit Spezifikation für asynchrone Webservices (http://hobit.sourceforge.net/statuscodes_list.html).