

Das Web ist zu Diensten – aber wie?

WebServices beschreiben

Von Thomas Wieland

WebServices sind derzeit eines der am heißesten diskutierten Themen für Anwendungsentwickler, versprechen sie doch, mit den alten Kommunikationsproblemen verteilter Anwendungen endlich Schluss zu machen. Was in der Theorie bestechend klingt, davor stellen sich in der Praxis natürlich noch einige Hürden. Eine der ersten ist die Frage, wie ein Client eigentlich erkennt, welche Methoden ein Server anbietet und welche Parameter dabei nötig sind. Um dies zu spezifizieren, bildet sich mit WSDL gerade ein herstellerübergreifendes Format heraus. Um dieses und seine Anwendung in der Java-Programmierung soll es in diesem Beitrag gehen.

Das Problem ist bekannt: Es gibt Millionen von Web-Servern weltweit, die so gut wie das gesamte Wissen der Menschheit jedermann zugänglich machen. Die übliche Form, in der Informationen geliefert werden, ist jedoch HTML, welches – von einem Browser dargestellt – den Inhalt dem menschlichen Betrachter präsentiert. Man kann also sagen, dass die Web-Server vorwiegend auf einen menschlichen Benutzer zugeschnitten sind. Inhalt und Layout sind beim Eintreffen beim Client miteinander so vermengt, dass eine Trennung durch ein Programm nur mühsam möglich ist. Sobald das Layout sich ändert (was ja mindestens jedes Vierteljahr passiert), ist das Programm wieder hilflos.

Plattformübergreifende Kommunikation mit SOAP

So liegt der Wunsch der Entwickler nahe, einen Zugriffsweg zu schaffen, auf dem Anwendungen direkt die auf einem Web-Server verfügbaren Informationen nutzen können. Die erste Schwierigkeit auf dem Weg zu diesem Ziel ist das Protokoll.

Setzt man sich mit einem Server in Verbindung, der aktive Inhalte über Formulare o.ä. anbietet, so wird auf diesem eine Funktion aufgerufen. Je nach eingesetzter Technologie drückt sich das als PHP-Funktion, ASP-Aufruf an eine COM-Komponente oder JSP-Aufruf an ein Bean bzw. ein Servlet aus. Obwohl dabei auch Methodenaufrufe dem Client angeboten werden, ist das Ergebnis für ihn doch i.a. schwer zu handhaben, denn er erhält lediglich eine HTML-Seite, die die Informationen zwischen einer Menge Layout-Angaben versteckt.

Mit CORBA/IIOP, RMI und DCOM gibt es andererseits bereits verschiedene Protokolle für den Zugriff auf entfernte Dienste. Diese sind aber leider nicht interoperabel, so dass Client und Server dieselbe Plattform verwenden müssen, um sinnvoll

miteinander zu kommunizieren – denn die Formate sind u.a. eng mit den darunter liegenden Komponentenmodellen verbunden. Außerdem sind diese Protokolle für Aufrufe über das Internet ungeeignet, da sie aufgrund variabler Portzuordnungen und Ähnlichem keine Verbindungen durch Firewalls aufbauen können.

Die Idee ist nun, für die Kommunikation das übliche Web-Protokoll HTTP einzusetzen, das durch alle Firewalls durchgelassen wird, und für die Serialisierung der Aufrufe und die Darstellung der Parameter ein Format, das selbstbeschreibend und daher *per se* interoperabel ist, nämlich XML. Beides zusammen ergibt das *Simple Object Access Protocol* SOAP [1]. Dieses wird mittlerweile von einem breiten Spektrum führender IT-Firmen unterstützt, angeführt von Microsoft und IBM; mit von der Partie sind aber auch Compaq, Hewlett-Packard und SAP. SOAP wurde ursprünglich bei der IETF (Internet Engineering Task Force) eingereicht und erhielt dort den Status einer „official recommendation“. Mittlerweile kümmert sich die „XML Protocol Working Group“ des WorldWide Web Consortiums (W3C) um die Standardisierung. Aktuell ist noch SOAP Version 1.1 gültig, so wie sie dem W3C übergeben wurde.

Die breite Unterstützung durch die Industrie führte auch dazu, dass es mittlerweile fast vierzig verschiedene SOAP-Implementierungen gibt. Für die Java-Welt ist dabei vor allem diejenige von IBM von Bedeutung, die nun von der Apache-Gruppe weiterentwickelt wird [2]. Die Installation ist dabei etwas aufwendig, da Apache-SOAP die Pakete Xerces als XML-Parser, das JavaBeans Activation Framework sowie JavaMail voraussetzt. Da der Beitrag [1] bereits den Umgang mit SOAP

ausführlich dargestellt hat, kann ich mich hier auf einige allgemeine Bemerkungen beschränken.

Das Ziel von SOAP ist, einen plattformunabhängigen Kommunikationsmechanismus durch Serialisierung in XML zu schaffen. Sogar das Transportprotokoll ist variabel und kann neben HTTP auch ein asynchrones Protokoll wie SMTP oder MQSeries sein. Für unsere Zwecke wollen wir uns im Folgenden jedoch auf HTTP beschränken. Eine Maxime bei der Definition von SOAP war die Einfachheit: Man wollte überall, wo es möglich ist, auf bestehende Technologien aufsetzen und keine neuen schaffen. Aus diesem Grund enthält die SOAP-Spezifikation auch keine Aussagen zu Fragen wie Objektaktivierung, Verbreitung von Ereignissen oder Speicherverwaltung, wie man sie von echten Objektmodellen für verteilte Systeme wie etwa CORBA kennt.

Außerdem enthält SOAP derzeit keine Mechanismen, um Daten verschlüsselt zu übertragen. Hier kann man jedoch auf HTTPS oder SSL zurückgreifen. Schwieriger sieht es da mit Transaktionen aus. Da SOAP genauso wie HTTP ein verbindungsloses Protokoll ist, bringt es derzeit keine Unterstützung für Transaktionen mit. Auch Rückrufe an den Client, beispielsweise bei asynchronen Rückmeldungen oder beim Eintreffen von Ereignissen, auf die sich der Client abonniert hat, sind im aktuellen Entwurf nicht spezifiziert. In dem Maße, in dem die Akzeptanz von SOAP jedoch breiter wird, ist auch eine Lösung für diese Fragen zu erwarten.

WebServices

Eigentlich sollte man meinen, dass es bereits genügend Kommunikationstechnologien im Internet gibt. Das eingangs beschriebene Problem, eine Funktion oder Methode über eine Web-Schnittstelle aufzurufen, lösen diese jedoch alle nicht. Erst seit SOAP besteht die Chance, dass dafür wirklich ein plattformunabhängiger Mechanismus geschaffen wird. Dies eröffnet völlig neue Perspektiven für die Anwendungsentwicklung, denn damit wird es möglich, dass eine einzige Applikation sich die riesige Funktionalität zunutze macht, die die Server des Internet jederzeit bereitstellen.

Ganz allgemein gesehen ist ein Webservice eine Anwendung, die ihre Methoden über eine Web-Schnittstelle zur Verfügung stellt. Informationen und Dienste, die ein Server anbietet, können so von anderen Anwendungen über eine Internet-

Verbindung abgefragt, aufgefunden und benutzt werden. Damit lassen sich verteilte Software-Architekturen erstellen, die Service-Komponenten aus dem Internet zur Laufzeit integrieren, um hochgradig dynamische Software-Systeme zu realisieren. Der auf den ersten Blick schlichte Aufruf einer Methode auf einem Webserver bietet also die Chance, den Traum von der völligen Plattformtransparenz und Interoperabilität in verteilten Anwendungen Wirklichkeit werden zu lassen.

Der Begriff des Webservice ist noch sehr neu. Daher sollte es uns nicht wundern, wenn es über seine Bedeutung noch unterschiedliche Auffassungen gibt. Nicht einmal die Bezeichnung selbst ist einheitlich. Hewlett-Packard spricht von „E-Services“, Sun nennt sie „Smart Services“, bei Microsoft und IBM wird „WebServices“ verwendet. Aufgrund der Marktmacht der letzten beiden dürfen wir jedoch annehmen, dass sich „WebService“ letztlich durchsetzen wird (wobei man wieder hierzulande rätselt, wie man es richtig schreiben soll).

Auch wenn in den White Papers und sonstigen Broschüren oft von „Komponenten“ und „Methoden“ gesprochen wird und die Syntax mancher Implementierungen dies suggeriert, sollten wir festhalten, dass Webservices zunächst einmal mit Objektorientierung nichts zu tun haben. Es gibt kein zugrunde liegendes Objektmodell auf Implementierungsebene wie etwa bei CORBA oder COM. Da zudem das SOAP-Protokoll verbindungslos ist, hat der Client keinerlei Anhaltspunkt, ob er bei zwei Aufrufen jedes Mal dasselbe Objekt erreicht – sollte die Implementierung tatsächlich mit Objekten arbeiten. Selbst die Vorstellung, bei einem Webservice als Ganzem handle es sich um ein Objekt, ist nicht korrekt; genauso wie Web-Server kann auch ein Service verschiedene Clients ohne Zusatzaufwand nicht auseinanderhalten. Zwei Aufrufer können also in exakt demselben Kontext arbeiten und haben es nicht etwa mit separaten „Objekten“ zu tun.

Wir sollten uns daher einen Webservice eher wie ein Modul vorstellen, das verschiedene Funktionen in einem gemeinsamen Container zusammenfasst oder – wenn wir beim Denken in Java bleiben wollen – als eine Klasse mit nur statischen Methoden. Das wirkt sich auch auf die Implementierung aus. Ein Webservice sollte stets zustandslos programmiert werden; denn das „Hinüberretten“ von

Daten von einer Methode zur anderen mittels Member-Variablen macht keinen Sinn. Man kann jedoch sich mit einem Schwenk auf ein dokumentenorientiertes Programmiermodell behelfen, bei dem die Identität eines Objekts in Form einer ID und der Status in einer mit dieser ID versehenen Datenbankzeile gespeichert werden.

WSDL

Als Protokoll für den Aufruf einer Methode eines `WebService` scheint sich SOAP durchzusetzen (obwohl es auch darum noch genügend Diskussion in den einschlägigen Gremien gibt). Wie wir aber von anderen Komponententechnologien wissen, genügt ein Aufrufprotokoll allein nicht. Es muss auch eine Terminologie bereitgestellt werden, mit der sich die zur Verfügung stehenden Dienste beschreiben lassen. Der Client muss ja wissen, wie die Methoden des jeweiligen `WebService` heißen, welche Parameter sie verlangen und welche Rückgaben sie liefern.

Zu diesem Zweck hat man die *Web Service Description Language* WSDL entwickelt [3]. Damit lässt sich ein `WebService` in allgemeiner Form beschreiben. Alle wesentlichen Aspekte, d.h.

- die Syntax des Service,
- die Bindung an ein bestimmtes Netzprotokoll,
- die Festlegung des Übertragungsformats sowie
- die Zuordnung zu bestimmten Internetadressen (URIs)

erscheinen getrennt voneinander und können daher auch unabhängig verändert werden. Die WSDL-Syntax wirkt daher auf den ersten Blick recht umfangreich. Sehen wir uns also die einzelnen Bestandteile an:

Ein *Port* spezifiziert eine konkrete Netzadresse eines `WebService`, also einschließlich URL und Kommunikationsport. Dabei wird gleichzeitig die Bindung an einen bestimmtes Übertragungsformat festgelegt – dazu später mehr. Hier erst einmal ein Beispiel für einen Port (der `WebService` ist dabei eine Schnittstelle zu Übersetzungsdienst Babelfish aus [4]):

```
<port name = "BabelFishPort" binding =
"tns:BabelFishBinding">
  <soap:address location
=
"http://services.xmethods.net:80/perl/soaplite.cgi"/>
</port>
```

Mehrere Ports, die inhaltlich zusammengehören, aber nicht untereinander kommunizieren, werden zu einem *Service* zusammengefasst. Auf diese

Weise erhält der Benutzer eine einheitliche Sicht auf die verschiedenen Zugangswege zum `WebService`. Im Beispiel der Kürze halber nur ein Port:

```
<service name = "BabelFish">
  <documentation>Translates text of up
to 5k in length, between a
  variety of languages.
</documentation>
  <port name = "BabelFishPort" binding
= "tns:BabelFishBinding">
    <soap:address location
=
"http://services.xmethods.net:80/perl/soaplite.cgi"/>
  </port>
</service>
```

Im Port mussten wir bereits die Bindung an das Übertragungsprotokoll angeben. Dieses müssen wir nun noch im Abschnitt *Binding* genauer festlegen. Derzeit sind für WSDL Bindungen an SOAP 1.1, HTTP Get/Post sowie MIME spezifiziert; da SOAP wohl das wichtigste Format werden dürfte, will ich mich hier darauf beschränken. Bei der SOAP-Bindung geht es nicht mehr nur um Deployment-Aspekte wie bei Service und Ports, sondern ganz konkret um die Schnittstelle des `WebService`. Wir müssen hier angeben, welche Methoden unser Service unterstützt und wie diese durch SOAP gehandhabt werden sollen. Insbesondere sind die Definitionen von Transportprotokoll, Headern und Namensräumen von Bedeutung. Zum Beispiel:

```
<binding name = "BabelFishBinding" type =
"tns:BabelFishPortType">
  <soap:binding style = "rpc" transport =
"http://schemas.xmlsoap.org/soap/http"/>
  <operation name = "BabelFish">
    <soap:operation soapAction
=
"urn:xmethodsBabelFish#BabelFish"/>
    <input>
      <soap:body use = "encoded" name-
space = "urn:xmethodsBabelFish"
encodingStyle
=
"http://schemas.xmlsoap.org/soap/encoding/"
/>
    </input>
    <output>
      <soap:body use = "encoded" name-
space = "urn:xmethodsBabelFish"
encodingStyle
=
"http://schemas.xmlsoap.org/soap/encoding/"
/>
    </output>
  </operation>
</binding>
```

Die eigentlichen Nachrichten werden im XML-Tag *Message* eingeschlossen. Hierbei werden der Name der Nachricht sowie ihre Parameter angegeben. Für die Parameter werden einige Standarddatentypen vom XML-Schema der WSDL unterstützt. Für komplexere kann man auch eigene Schemata angeben bzw. darauf verweisen. Beachten Sie, dass eine *Message* nur eine Nachricht in eine Richtung ausdrückt. Hierbei bleibt noch offen,

ob es sich um eine eingehende oder ausgehende Nachricht handelt. Hier zwei einfache Beispiele:

```
<message name = "BabelFishRequest">
  <part name = "translationmode" type =
"xsd:string"/>
  <part name = "sourcedata" type =
"xsd:string"/>
</message>
<message name = "BabelFishResponse">
  <part name = "return" type =
"xsd:string"/>
</message>
```

In *Part* geben Sie die Bestandteile der Nachricht und ihre Typen an, also die Parameter, die bei einem Methodenaufruf übergeben bzw. zurückgegeben werden. Hier geben Sie etwa in *translationmode* die Sprachen an, zwischen den übersetzt werden soll – für Deutsch -> Englisch beispielsweise „de_en“ – und in *sourcedata* den Text.

Unter *PortType* werden die Nachrichten dann zu Operationen (*operations*) zusammengefasst, die jeweils unterschiedliche Kommunikationsparadigmen folgen können. Je nach Art der Operation kann diese setzt sich aus einer Eingabe und/oder einer Ausgabenachricht zusammensetzen. Im folgenden Beispiel handelt es sich um eine Anfrage/Antwort-Operation:

```
<portType name = "BabelFishPortType">
  <operation name = "BabelFish">
    <input message
"tns:BabelFishRequest" name = "BabelFish"/>
    <output message
"tns:BabelFishResponse" name = "BabelFishResponse"/>
  </operation>
</portType>
```

Auch wenn die zugrunde liegende Infrastruktur, d.h. vor allem die Übertragungsprotokolle, noch nicht alle Möglichkeiten unterstützt, sind in WSDL schon einmal vier verschiedene Kommunikationsszenarien spezifiziert:

- *Oneway*: Hier findet der Nachrichtenaustausch nur in eine Richtung statt, d.h. der Service bekommt nur eine Eingabe, ohne eine Ausgabe liefern zu müssen. Daher besteht dabei der *portType* auch nur aus einer *Input-Message*.
- *Request-response*: Dies ist das gebräuchlichste Szenario, das man auch von HTML-Seiten kennt. Der Server erhält eine Nachricht, verarbeitet diese und schickt eine Antwort an den Aufrufer zurück, etwa wie in obigem Babelfish-Beispiel.
- *Solicit-Response*: Das ist genau das umgekehrte Verhalten, bei dem der Server erst eine Nachricht verschickt und dann eine Antwort vom Client erhält. Mit SOAP ist das heute noch nicht ohne Weiteres möglich.

- *Notification*: Dies drückt die Einbahnstraße in die andere Richtung aus, d.h. der Server schickt lediglich eine Nachricht los, ohne auf eine Rückmeldung zu warten. Typisches Beispiel dafür sind Notifizierungen bei Ereignissen, für die sich Clients angemeldet haben.

WSDL-Dateien sollte ein Entwickler von WebServices (oder darauf zugreifender Clients) zwar einigermaßen lesen können, muss sie aber nicht restlos verstehen. Im Gegensatz zu anderen Schnittstellenbeschreibungssprachen wie IDL bei CORBA oder COM sollte man sich bei WSDL auf das Funktionieren der verwendeten Entwicklungswerkzeuge verlassen und WSDL entweder aufgrund der programmierten Schnittstelle der jeweiligen Klasse automatisch erzeugen lassen oder es zur Nutzung lediglich importieren.

WebServices mit Java

Für Java-Programmierer bietet IBM ein mittlerweile sehr umfangreiches WebServices Toolkit zum Download an [5]. Es enthält verschiedene Tools zum Erstellen, Konfigurieren und Publizieren von WebServices, einen Embedded WebSphere als Test-Servlet-Engine, die JAR-Archive für Xalan, Xerces, SOAP und weitere nützliche Software; dazu auch ausführliche Dokumentation wie etwa die WSDL-Spezifikation. Die meisten Beispiele von XMethods (www.xmethods.com), einer guten Adresse für interessante WebServices, sind auf das IBM Toolkit abgestimmt.

Um von Java aus mit einem Webservice in Verbindung zu treten, benötigen wir eine Proxy-Klasse, die den Service kapselt. Ein solche lässt sich sehr leicht aus der WSDL-Beschreibung mittels

```
java com.ibm.wsdl.Main -in BabelFishService.wsdl
```

erstellen. Der Client dazu ist sehr einfach zu bauen, wie Listing 2 zeigt. Es genügt, eine Instanz der Proxy-Klasse anzulegen und schon ist der Webservice nutzbar.

Natürlich unterstützt das Toolkit auch die serverseitige Programmierung, also die Erstellung eines Rumpfes für die Implementierung des Webservice selbst auf der Basis der Definitionen in der WSDL-Datei. Umgekehrt können Sie sich zudem für eine bestehende Server-Klasse eine Datei mit der Schnittstelle im WSDL-Format erzeugen lassen und so aus Ihren Klassen WebServices machen. Das Deployment ist dabei durch den automatisch

erzeugten Deployment Descriptor bereits auf der Kommandozeile sehr einfach; richtig komfortabel geht es mit dem webbasierten Admin-Werkzeug von Apache SOAP.

Fazit

WebServices haben das Potenzial, unser Verständnis von Anwendungsentwicklung nachhaltig zu verändern. Wenn Bibliotheken oder Komponenten mit den benötigten Funktionen nicht mehr lokal installiert werden müssen, sondern dynamisch über das Netz eingebunden werden, kann sich der Programmierer vielleicht einiger Infrastrukturentwicklung entledigen und hat so mehr Zeit für seine eigentlichen Aufgaben. Auf der anderen Seite begibt man sich dabei natürlich in eine neue Form der Abhängigkeit, da die so gebaute Anwendung zur Laufzeit stets auf die Verfügbarkeit der Dienste angewiesen ist, die sie aufruft.

Zudem sind momentan WebServices noch ein sehr neues Thema, bei dem noch vieles im Fluss ist. Die WSDL-Spezifikation ist noch nicht letztgültig verabschiedet, die Entwicklungswerkzeuge der beiden Protagonisten Microsoft und IBM können noch nicht mit allen Feinheiten, die spezifiziert

sind, wirklich umgehen – und sich miteinander auch nur sehr rudimentär austauschen. Die Interoperabilität, die WebServices versprechen, ist daher momentan noch nicht gewährleistet. Bei dem Tempo, in dem sich Ankündigungen und Releases derzeit überschlagen, kann das bis zum Erscheinen dieses Heftes schon wieder anders sein.

Literatur

- [1] T. Frotscher: „Kommunikation mit dem Simple Object Access Protocol“, *Java-Spektrum*, 2/2001, S. 60
- [2] <http://xml.apache.org/soap>
- [3] Web Services Description Language (WSDL) 1.0, <http://www-4.ibm.com/software/developer/library/w-wsdl.html?dwzone=web>
- [4] <http://www.xmethods.net/detail.html?id=14>
- [5] <http://www.alphaworks.ibm.com/tech/webservicestoolkit>

Listing 1

```
<?xml version = "1.0"?>
<definitions name = "BabelFishService"
xmlns:tns="http://www.xmethods.net/sd/BabelFishService.wsdl" targetNamespace =
"http://www.xmethods.net/sd/BabelFishService.wsdl" xmlns:xsd =
"http://www.w3.org/1999/XMLSchema" xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/" xmlns
= "http://schemas.xmlsoap.org/wsdl/">
  <message name = "BabelFishRequest">
    <part name = "translationmode" type = "xsd:string"/>
    <part name = "sourcedata" type = "xsd:string"/>
  </message>
  <message name = "BabelFishResponse">
    <part name = "return" type = "xsd:string"/>
  </message>
  <portType name = "BabelFishPortType">
    <operation name = "BabelFish">
      <input message = "tns:BabelFishRequest" name = "BabelFish"/>
      <output message = "tns:BabelFishResponse" name = "BabelFishResponse"/>
    </operation>
  </portType>
  <binding name = "BabelFishBinding" type = "tns:BabelFishPortType">
    <soap:binding style = "rpc" transport = "http://schemas.xmlsoap.org/soap/http"/>
    <operation name = "BabelFish">
      <soap:operation soapAction = "urn:xmethodsBabelFish#BabelFish"/>
      <input>
        <soap:body use = "encoded" namespace = "urn:xmethodsBabelFish" encodingStyle =
"http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
      <output>
        <soap:body use = "encoded" namespace = "urn:xmethodsBabelFish" encodingStyle =
"http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
</service name = "BabelFish">
```

```
<documentation>Translates text of up to 5k in length, between a variety of lan-
guages.</documentation>
<port name = "BabelFishPort" binding = "tns:BabelFishBinding">
  <soap:address location = "http://services.xmethods.net:80/perl/soaplite.cgi"/>
</port>
</service>
</definitions>
```

Listing 2

```
import java.net.*;
import org.apache.soap.*;

public class BabelFishTranslator
{
  public static void main(String[] argv) throws MalformedURLException, SOAPException
  {
    BabelFishServicePortTypeProxy proxyObject = new BabelFishServicePortTypeProxy();
    System.out.println("Expression in English: " + proxyObject.BabelFish("de_en", argv[1]));
  }
}
```

Über den Autor

Dr. Thomas Wieland ist Software-Architekt im Bereich Middleware and Application Integration der Corporate Technology der Siemens AG. Er ist erreichbar unter thomas@drwieland.de.