Orientation in Objects

Apache Axis

Architektur und Erweiterbarkeit

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Thomas Bayer
Orientation in Objects GmbH

Veröffentlicht am: 23.9.2003

ABSTRACT

Die SOAP Implementierung Apache Axis hat sich zu einem wichtigen Tool und Framework für Web Services in der Java-Welt entwickelt. Sein Design wurde von Grund auf neu gestaltet und basiert auf Handlern und dem SAX API. Dieser Artikel führt den Web Service und SOAP kundigen Leser in die Architektur von Axis ein und beschreibt, wie Axis mit eigener Funktionalität am Beispiel eines Handlers für signierte Nachrichten und eines Providers für Stored Procedures erweitert werden kann.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

ERWEITERN VON AXIS MIT HANDLERN UND PROVIDERN

Axis stellt eine typische Web Anwendung dar, die in Form eines Web-Archives in einen Servlet 2.2 konformen Web Container wie z.B. Tomcat oder Jetty installiert werden kann. Das Web-Archiv enthält drei Servlets für die Laufzeit, die Administration und für das Verfolgen von SOAP Aufrufen wie in Abbildung 1 dargestellt.

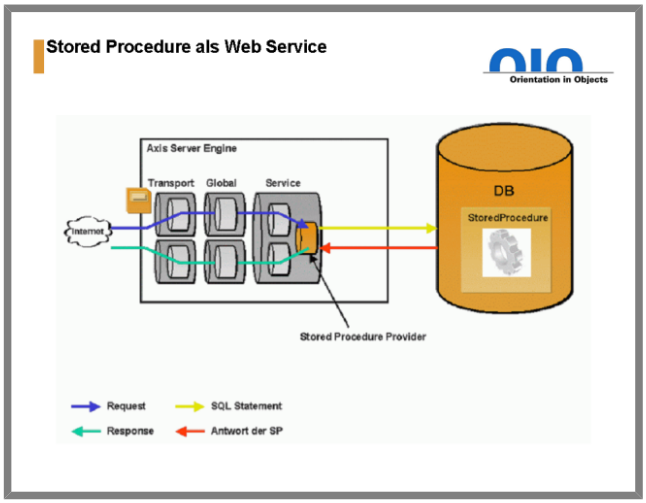

Abbildung 1: Axis im Web Container

Die benötigten Klassen befinden sich in Jar-Dateien im lib-Verzeichnis. Das classes- sowie das lib-Verzeichnis können später die Klassen der Web Service Implementierungen und die Axis Erweiterungen aufnehmen.

INTEGRATION IN EIGENE WEB ANWENDUNGEN

Der Entwickler kann zwischen zwei Alternativen bei der Verbindung seiner eigenen Implementierung mit Axis wählen. Das Axis Web-Archiv kann mit eigenem Code erweitert, oder Bestandteile von Axis können in ein eigenes Web Archiv aufgenommen werden.

ARCHITEKTUR

Apache Axis kann über die Entwicklung von Clients und Servern hinaus auch für die Entwicklung von Infrastruktur wie z.B. SOAP Gateways verwendet werden.

Axis ist keine starre Software aus einem Guss, sondern ein erweiterbares Framework. Module gliedern Axis in Teilsysteme, die verschiedene Aufgaben übernehmen. Abbildung 2 zeigt u.a. Subsysteme für den Transport von Nachrichten, die Administration und den Aufruf von Serviceimplementierungen. Höhere Schichten bauen auf den Diensten der niedrigeren Schichten auf. Die Aufteilung der Aufgaben in Module in Verbindung mit Konfigurationsdateien ermöglicht eine Erweiterung ohne Änderungen an Axis selbst vorzunehmen.

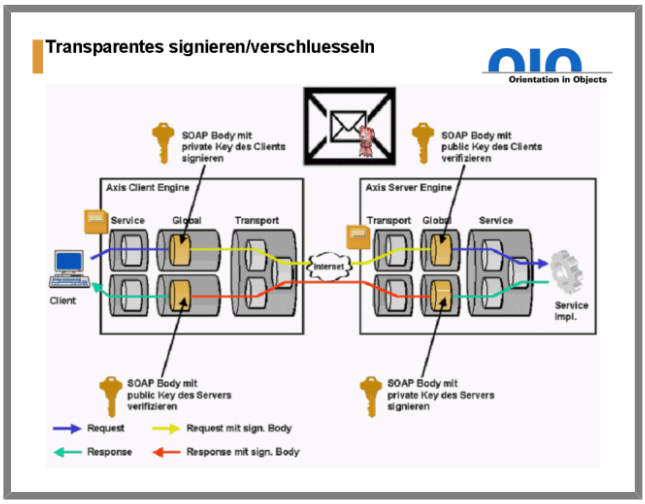

Abbildung 2: Subsysteme

Von grundlegender Bedeutung sind Nachrichten und Handler. Nachrichten durchlaufen in Form eines *MessageContext* die einzelnen Subsysteme.

MESSAGECONTEXT

Der *MessageContext* ist, wie aus Abbildung 3 ersichtlich, eine Struktur, die Referenzen auf die Request- und Response Nachricht sowie auf eine Reihe von Attributen beinhaltet. Der Zugriff auf den Request, die Response und weitere Eigenschaften wie z.B. die Art des Transports, erfolgt in den Handlern über den *MessageContext*.

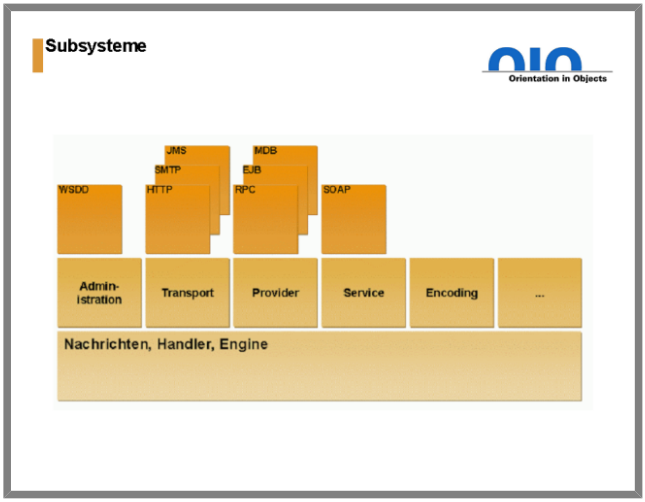

Abbildung 3: Message Context

HANDLER UND KETTEN

Die Verarbeitung der Request-Nachricht und das Erzeugen der Response-Nachricht wird von Handlern durchgeführt. Über die *invoke()* Methode bekommt ein Handler einen *MessageContext* übergeben, auf den er in seinem Verarbeitungsschritt zugreifen kann.

Mehrere Handler können in einer Kette zusammengefaßt werden. Eine Handlerkette bietet die gleiche Schnittstelle wie ein Handler und kann eine geordnete Menge von Handlern enthalten. Abbildung 4 zeigt eine Kette aus drei Handlern.

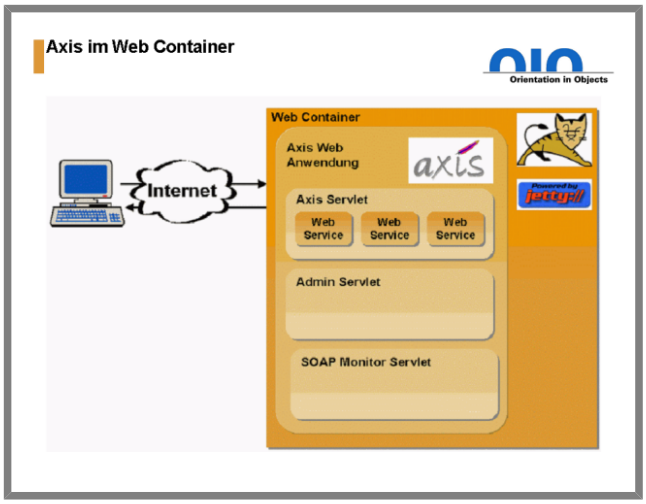

Abbildung 4: Handler Kette

Die Verarbeitung von Anfragen erfolgt über Filterketten, die sich aus Handlern zusammensetzen. Client und Server bestehen immer aus einer Axis Engine, welche die Handlerketten Service, Transport und Global umfaßt.

Die Verkettung der Handler ist flexibel und kann über Konfigurationsdateien angepaßt werden. Über eigene Handler kann in die Verarbeitung von SOAP Nachrichten eingegriffen werden. Typische Aufgaben für Handler sind Logging oder Überprüfungen für die Sicherheit.

Bei der alltäglichen Anwendungsentwicklung ist es nicht notwendig, selbst Handler zu erstellen. Der Entwickler kann fertige Handler für Logging, Authentifizierung oder das SOAP Monitortool einbinden. Wird Funktionalität benötigt, die über die mitgelieferten Handler hinausgeht, ist Axis offen für eigene Erweiterungen, die sich recht einfach realisieren lassen.

Bevor die Erstellung von eigenen Handlern beschrieben wird, betrachten wird den Fluß von Nachrichten durch die Handler genauer.

IT'S A LONG WAY

Die Infrastruktur auf Server- und Clientseite unterscheidet sich nur geringfügig. Abbildung 5 skizziert den Weg einer Nachricht von der Client Engine zur Server Engine. Im Folgenden wird der Weg, den ein synchroner Aufruf durchläuft, skizziert. Zu Beginn nimmt die Axis Engine auf Clientseite einen Aufruf der Anwendung entgegen, erzeugt eine Nachricht, die dann die Ketten und Handler auf Clientseite durchläuft. Begonnen wird bei der Servicekette, gefolgt von der globalen Kette. Die Transportkette ist anschließend für den Versand der Nachricht zum Server zuständig. Axis unterstützt Protokolle für einen Transport über HTTP, Mail und den Java Message Service JMS.

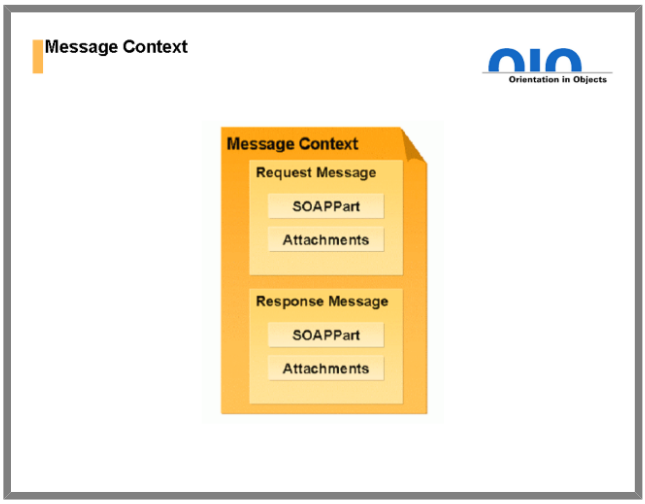

Abbildung 5: Nachrichten Pfad

Auf Serverseite nimmt ein Transport Listener die Nachricht entgegen und packt diese wieder in einen *MessageContext*, damit die Nachricht die Handler des Servers durchlaufen kann. Der Listener für den Transport von SOAP über HTTP ist mit einem Servlet realisiert.

In der Servicekette läuft der Nachrichtenpfad auf einen sogenannten "Pivot-Handler", der den Verlauf der Nachricht umkehrt. Bevor die Nachricht wieder zurück zum Client läuft, ruft der Pivot-Handler, der auf Serverseite Provider genannt wird, den Zieldienst auf. Es gibt Provider für zahlreiche Serviceimplementierungen. Der Service kann mit normalen Java Objekten oder mit Enterprise JavaBeans realisiert sein. Skriptsprachen können über das Bean Scripting Framework angesprochen werden. Für entfernte Objekte stehen nicht nur RMI und CORBA, sondern auch Microsofts COM zur Verfügung.

Nachdem die Implementierung des Service ihren Dienst vollbracht hat, kehrt sich der Nachrichtenfluss um. Im *MessageContext*, der zuvor nur den Request beinhaltet hat, wird jetzt zusätzlich eine Response referenziert. Die Ketten werden in umgekehrter Reihenfolge durchlaufen. Der Transport auf der Seite des Servers schickt die Antwort zum Client, bei dem ebenfalls der Transport die Nachricht entgegennimmt. Verpackt als *MessageContext* geht es über die globale Kette zurück zur Servicekette und danach zum Client Code, von dem der Aufruf ursprünglich initiiert wurde.

KONFIGURATION DER HANDLER

Nachdem der Fluss der Nachrichten skizziert wurde, wird im Folgenden die Konfiguration der Handler und Ketten beschrieben. Neben Web Services können über einen Deployment Descriptor auch Ketten und Handler in eine Axis Engine installiert und deinstalliert werden.

Wie **Listing 1** zeigt, finden wir im Deployment Descriptor Abschnitte für die Konfiguration der Handlerketten Global, Service und Transport. Ein Deployment Descriptor muß nicht alle diese Element enthalten.

```
<?xml version="1.0" encoding="UTF-8"?>
  <deployment ...>
    <globalConfiguration> ... </globalConfiguration>
    <service ...> ... </service>
  <transport ...> ... </transport>
</deployment>
```

Beispiel 1: Struktur eines Deployment Descriptors

Im **Listing 2** wird eine Servicekette für einen Web Service konfiguriert. In den Request Flow wird ein Handler für die Authentifizierung über Benutzername und Passwort eingeklinkt. Der Handler wird nur dann im Nachrichtenfluss aufgerufen, wenn der Order Web Service angesprochen wird. Damit ein Handler beim Aufruf aller Web Services ausgeführt wird, kann er in die globale Kette aufgenommen werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment ...>
  <handler
    name="auth"
    type="java:org.apache.axis.
       handlers.SimpleAuthenticationHandler" />
  <service name="OrderService"
         provider="java:RPC">
    <requestFlow>
      <handler type="auth" />
    </requestFlow>
    <parameter name="allowedMethods"
           value="*" />
    <parameter name="className"
           value="de.oio.soap.OrderService"/>
  </service>
</deployment>
```

Beispiel 2: Konfiguration einer Service Kette

SZENARIO "SICHERHEIT"

Mit Handlern in der globalen Kette lassen sich für den Programmierer transparent neue Eigenschaften und Funktionalitäten hinzufügen. Wie es geht, verdeutlicht ein Beispiel zur SOAP Sicherheit.

Alle Nachrichten sollen transparent signiert und verifiziert werden. Die Signaturen garantieren, daß die Nachrichten unverfälscht übermittelt wurden und daß sie tatsächlich vom Client oder Server stammen.

Abbildung 6 zeigt, an welchen Stellen im Nachrichtenfluss eingegriffen wird. Ein globaler Handler im Request Flow des Clients kann den SOAP Envelope durch einen signierten ersetzen. Auf Serverseite wird der signierte SOAP Umschlag in einem globalen Handler verifiziert. Bei einer ungültigen Signatur wird der Vorgang abgebrochen und ein SOAP Fault erzeugt. Das Signieren und Verifizieren der Response kann analog erfolgen. Für die Realisierung sind nur zwei verschiedene Handler notwendig, einer zum Signieren und einer zum Verifizieren.

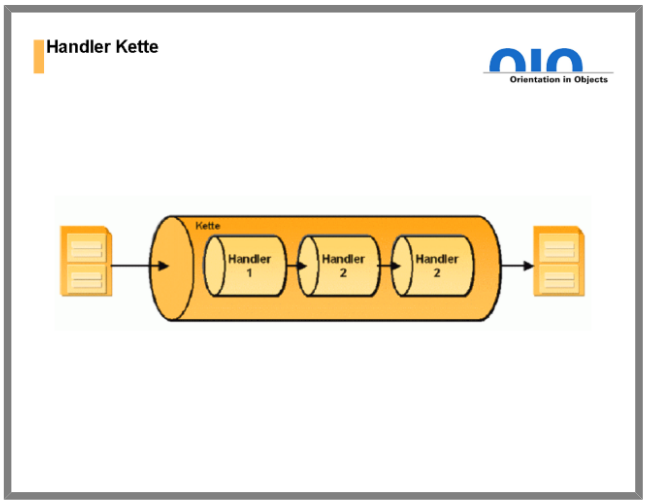

Abbildung 6: Transparentes signieren/verschluesseln

Das Signieren und Verifizieren einer Nachricht erfordert ein asynchrones Verfahren mit öffentlichen und privaten Schlüsseln. Die W3C Standards XML Signature und XML Encryption berücksichtigen die Besonderheiten von XML und beschreiben die Einbettung von Signaturen, Zertifikaten und Metainformationen. Signierte oder verschlüsselte Nachrichten werden deutlich größer und die erforderlichen Algorithmen benötigen einiges an Rechenzeit. Ein Roundtrip auf PC Hardware mit der XML Security Implementierung von Apache und dem JCE Provider der "Legion of the Bouncy Castle" nimmt ca. 500 ms in Anspruch (auf P III 600 MHz, 384 MB).

ERSTELLEN VON HANDLERN

Ein Handler wird mit einer Klasse erstellt, die das Handler-Interface implementiert. Die abstrakte Klasse *BasicHandler* erleichtert die Erstellung von Handlern über Vererbung. Die Callback Methoden *invoke()* und *onFault()* können mit eigener Funktionalität überschrieben werden.

Listing 3 zeigt einen Handler, der je nach Stand des Message Flows den SOAP Umschlag des Requests oder den der Response in einen signierten Umschlag packt. In Axis enthalten ist ein Beispiel für das Signieren über einen Handler, das als Vorlage für eigene Projekte dienen kann. Zukünftig ist sicher auch mit Handlern zu rechnen, die beispielsweise den "Standard" WS-Security implementieren.

```
package de.oio.soap.security;
import org.apache.axis.*;
public class SigningHandler
                    extends BasicHandler {
  static {
    org.apache.xml.security.Init.init();
  }
  public void invoke(MessageContext ctx)
                       throws AxisFault {
    try {
      ctx.setCurrentMessage( new Message(
         new SignedSOAPEnvelope( ctx)
       )
      );
    } catch ( Exception e) {
       throw AxisFault.makeFault( e);
    }
  }
}
```

Beispiel 3: SigningHandler

PROVIDER

Mit Provider ist hier kein Internet Dienstleister, sondern der Anbieter eines Service gemeint. Ein Provider ist ein spezieller Handler, der einen Aufruf an eine Implementierung weiterleitet und dessen Rückgabe wieder dem Nachrichtenfluss übergibt. Axis enthält Provider für folgende Implementierungen:

- Java Klassen
- Enterprise JavaBeans
- RMI Server
- CORBA Servants
- Beans Scripting Framework
- Microsoft COM Objekte

Über den Provider für Java Klassen kann prinzipiell alles angesprochen werden, was von Java aus erreichbar ist. Bei bestimmten Aufgaben, wie z.B. dem Aufruf von Methoden eines stateless SessionBean ist es komfortabler, einen spezialisierten Provider zu verwenden. Um neue Beans verfügbar zu machen, genügt die Erstellung eines Deployment Descriptors.

Axis kann um eigene Provider erweitert werden. Der zweite Teil dieses Artikels beschreibt die Erweiterung am Beispiel eines Providers für den Zugriff auf gespeicherte Prozeduren einer Datenbank. Über den Provider können Stored Procedures, wie in Abbildung 7 dargestellt, als Web Service angeboten werden.

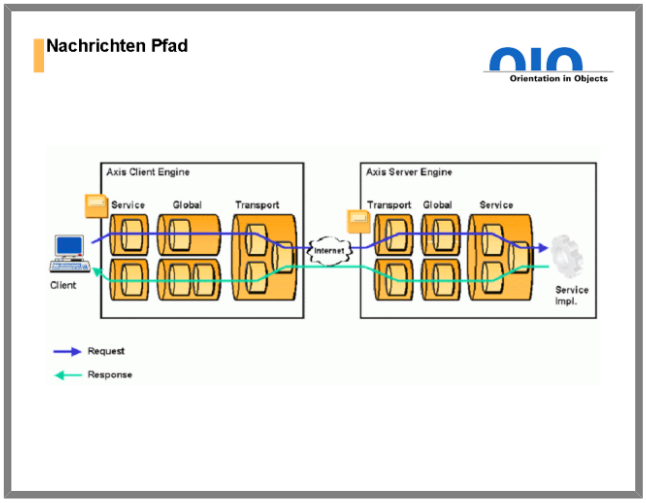

Abbildung 7: Stored Procedure als Web Service

Listing 4 zeigt einen Deployment Descriptor für einen Service "sp", der mit User, Passwort und URL für eine Datenquelle konfiguriert wird. Der dafür notwendige Provider "java:SP" muss in Axis eingeklinkt werden.

```
<?xml version="1.0"?>
<deployment xmlns="http://xml.apache.org/
                              axis/wsdd/"
        xmlns:java="http://xml.apache.org/
                axis/wsdd/providers/java">
  <service name="sp" provider="java:SP">
    <parameter name="driver"
      value="oracle.jdbc.driver.OracleDriver"/>
    <parameter name="dburl"
            value="jdbc:oracle:thin:
                  @localhost:1521:ORCL"/>
    <parameter name="user"
            value="scott" />
    <parameter name="password"
            value="tiger" />
  </service>
</deployment>
```

Beispiel 4: Deployment Descriptor für Stored Procedures

DAS ADMINISTRATION SUBSYSTEM

Über das Teilsystem für Administration können Axis Engines konfiguriert werden. Die Konfiguration kann aus einer XML Datei, dem *Web Service Deployment Descriptor* ausgelesen werden. Für die XML Elemente der Datei gibt es eine Klassenhierarchie von Fabriken. Über die Fabriken werden zur Laufzeit benötigte Handler, Ketten, Provider und andere Artefakte erzeugt.

Für den Stored Procedures Provider wird ebenfalls eine Fabrik benötigt, die die Implementierung des Providers erzeugen kann. Die Klasse *WSDDStoredProcedureProvider* in **Listing 5** übernimmt diese Aufgabe. Die Methode *newProviderInstance()* liefert eine neue Instanz des Providers. Den Namen, über den im Deployment Descriptor, der Provider referenziert werden kann, liefert die Methode *getName()*.

```
package de.oio.soap.axis;
import org.apache.axis.*;
import org.apache.axis.deployment.wsdd.*;
public class WSDDStoredProcedureProvider
               extends WSDDProvider {
  public Handler newProviderInstance(
    WSDDService service,
    EngineConfiguration config)
                        throws Exception {
      return new StoredProcedureProvider();
  }
  public String getName() {
    return "SP";
  }
}
```

Beispiel 5: WSDDProvider für die StoredProcedureProvider-Implementierung

Die Fabrik wird Axis über eine Datei im Klassenpfad bekannt gemacht. Axis sucht über den Klassenpfad in einem lokalen Unterverzeichnis META-INF/services eine Datei mit dem recht ungewöhnlichen Namen:

`org.apache.axis.deployment.wsdd.Provider`

In dieser Datei kann sich ein Liste mit Klassen befinden, die als zusätzliche Fabriken für Provider dienen. In diesem Fall ist das die Klasse:

`de.oio.soap.axis.WSDDStoredProcedureProvider`

Die Implementierung des Providers stellt die Klasse *StoredProcedureProvider* in **Listing 6** dar. Als Basisklasse dient *BasicProvider*, die selbst die Handler Schnittstelle implementiert. Die *invoke()* Methode ist in diesem speziellen Handler dafür zuständig, aus dem Request die Response zu erzeugen. Zunächst werden Parameter für die Datenbankverbindung, die im Deployment Descriptor spezifiziert wurden, ausgelesen und eine Verbindung hergestellt. Danach wird über den *MessageContext* navigiert und das SOAP Body Element mit der SOAP Operation ermittelt.

```
package de.oio.soap.axis;
import java.sql.*;
import java.util.*;
import org.apache.axis.*;
import org.apache.axis.handlers.soap.*;
import org.apache.axis.message.*;
import org.apache.axis.providers.*;
import org.xml.sax.*;
public class StoredProcedureProvider
                  extends BasicProvider {
  private static final String NS_URI =
                "http://oio.de/soap/sp/";
  private Connection connection;
  public void invoke(
    MessageContext ctx) throws AxisFault{
     try{
      // DB Verbindung herstellen
      connection = getConnection(
        (String) ctx.getService().
                    getOption("driver"),
        (String) ctx.getService().
                     getOption("dburl"),
        (String) ctx.getService().
                     getOption("user"),
        (String) ctx.getService().
                    getOption("password")
      );
      // Element mit SOAP Operation holen
      RPCElement rpcElement = (RPCElement)
                  ctx.getRequestMessage()
                     .getSOAPEnvelope()
                     .getBodyElements()
                     .iterator()
                     .next();
      // Parameter der SOAP Operation holen
      Vector sqlParams =
             getCallParameter(rpcElement);
      // Callable Statement mit
      // Parameterliste erzeugen
      CallableStatement spCall =
        createCallableStatement(
                rpcElement.getMethodName(),
                           sqlParams );
      ResultSet resultSet =
                    spCall.executeQuery();
      // BodyElement mit Rückgabewert erzeugen
      SOAPBodyElement bodyElement =
                  new SOAPBodyElement();
      bodyElement.setName(
               rpcElement.getMethodName() +
                           "Result");
      bodyElement.setNamespaceURI(NS_URI);
      bodyElement.setPrefix("sp");
      ResultSetMetaData metaData =
                resultSet.getMetaData();
      // Rückgabe in Form von
      // canonischem XML erzeugen
      while (resultSet.next()) {
        RPCElement rowElement
               = new RPCElement(NS_URI,
                           "rowSet",
                           null);
        for (int i = 1;
            i <= metaData.getColumnCount();
            i++) {
          rowElement.addParam(
            new RPCParam( NS_URI,
                 metaData.getColumnName(i),
                   resultSet.getString(i))
          );
        }
        bodyElement.addChild(rowElement);
      }
      connection.close();
      // SOAP Umschlag erzeugen und
      // BodyElement aufnehmen
      SOAPEnvelope resEnv =
        new SOAPEnvelope(
                  ctx.getSOAPConstants());
      resEnv.addBodyElement(bodyElement);
      // Response in Context stellen
      ctx.setResponseMessage(
                   new Message(resEnv));
    } catch (Exception e) {
      log.error(e);
      throw new AxisFault(
            "Fehler im SP-Provider: " + e);
    }
  }
  ...
}
```

Beispiel 6: Implementierung des StoredProcedureProvider

```
RPCElement rpcElement = (RPCElement)
                 ctx.getRequestMessage()
                    .getSOAPEnvelope()
                    .getBodyElements()
                    .iterator()
                    .next();
```

Beispiel 7: Element mit SOAP Operation holen (Detail)

Die Parameter der SOAP Operation werden in einen *Vector* übertragen, aus dem dann ein *CallableStatement* für den Aufruf einer Stored Procedure erzeugt wird. Für den Aufruf wird das Statement mit einem String wie dem Folgenden vorbereitet und anschließend mit den Parameterwerten gefüllt.

```
{call CustOrdersOrders(? )}
```

Nach der Ausführung wird aus dem *ResultSet* der Abfrage kanonisches XML erzeugt und in den Umschlag der Response eingepackt. Eine Referenz auf die Response-Nachricht wird dem *MessageContext* zugewiesen und die Methode beendet.

```
<?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:sp="http://oio.de/soap/sp/"
   xmlns:xsi="http://www.w3.org/2001/
                     XMLSchema-instance">
  <soapenv:Body>
    <sp:CustOrdersOrdersResult>
      <sp:rowSet>
        <sp:OrderID xsi:type="xsd:string">
           10301
        </sp:OrderID>
        <sp:OrderDate xsi:type="xsd:string">
           1996-09-09 00:00:00.0
        </sp:OrderDate>
        <sp:RequiredDate xsi:type="xsd:string">
           1996-10-07 00:00:00.0
        </sp:RequiredDate>
        <sp:ShippedDate xsi:type="xsd:string">
           1996-09-17 00:00:00.0
        </sp:ShippedDate>
      </sp:rowSet>
      ...
</soapenv:Envelope>
```

Beispiel 8: Response Nachricht einer Stored Procedure (Auszug)

EINSCHRÄNKUNGEN UND MÖGLICHKEITEN

Der Provider für die Stored Procedures ist einfach aufgebaut und unterstützt nur die Rückgabe in Form einer Tabelle. Für die Datentypen werden ausschließlich Strings verwendet. OUTPUT-Parameter und eine WSDL-Generierung werden nicht unterstützt. Für das Erzeugen von WSDL steht in Axis eine komfortable Infrastruktur in Form von Basisklassen und einer *Emitter* Klasse zur Verfügung. OUTPUT-Parameter lassen sich über Metadaten der Datenbank realisieren.

Das Beispiel für den Stored Procedure Provider zeigt, wie sich Provider realisieren lassen und kann als Ausgangsbasis für eigene Implementierungen dienen. Mit einem eigenen Provider besteht die Möglichkeit eine Infrastruktur zu schaffen, die es erleichtert Web Services für bestehende Systeme anzubieten.

FAZIT

SOAP ist nicht wirklich ein Protokoll, sondern vielmehr ein Baukasten für die Erstellung von SOAP Protokollen. Dienste für Transaktionen oder Sicherheit können über Contexte, Token und Schlüssel im SOAP Header realisiert werden. Die Möglichkeiten von XML wie Namespaces und Schemas bieten zusätzliche Möglichkeiten. Mit Axis kann die Erweiterbarkeit von SOAP genutzt werden, indem eigene Handler sich um die Verarbeitung der zusätzlichen Daten kümmern. Selbst wenn der einzelne Entwickler nicht die Erweiterungsmöglichkeiten von Axis selbst nutzt, kann er von zukünftigen Erweiterungen durch Drittanbieter profitieren.

Thomas Bayer ist einer der Gründer von Orientation in Objects, einem Anbieter für Entwicklung und Schulung im Bereich Java Enterprise, XML und Web Services.

REFERENZEN

- Quellcode der Beispiele
  www.oio.de/axis-architecture.htm (http://www.oio.de/axis-architecture.htm)
- Apache Axis
  ws.apache.org/axis/ (http://ws.apache.org/axis/)
- XML Signature
  www.w3.org/Signature/ (http://www.w3.org/Signature/)
- Apache XML Security
  hxml.apache.org/security/ (http://xml.apache.org/security/)
- Bouncy Castle Crypto APIs
  www.bouncycastle.org/ (http://www.bouncycastle.org/)