Martin Kalin

Java Web Services

Up and Running

1. Dezember 2010

α-Version

Deutsche Übersetzung von Ralf Wahner

Inhalt

Vorwort

Inhaltsübersicht

[0] Dieses Buch richtet sich an Programmierer, die sich für die Entwicklung von Java-Webservices, das heißt in Java geschriebener Webservices und Java-Clients für in einer beliebigen Sprache geschriebene Webservices interessieren. Das Buch ist eine beispielorientierte Einführung in die Java API for XML-Web Services (JAX-WS), das Framework der Wahl für Java-Webservices, gleichsam SOAP-basiert oder im REST-Stil. Die JAX-WS wird in ihrer gesamten Bandbreite erklärt. Daher werden auch führende Entwicklungen wie das Jersey-Projekt für REST-basierte Webservices berücksichtigt. Jersey heißt offiziell „Java API for RESTful Web Services" (JAX-RS).

[1] Die JAX-WS gehört zum Metro Web Services Stack, kurz Metro. Metro gehört seit Version 6 der Java Standard Edition (SE 6) zum Grundumfang der Java-Distribution, wobei die Metro-Releases dem Stand der SE 6 voraus eilen. Das aktuelle Metro-Release kann separat unter der Adresse *https://wsit.dev.java.net* heruntergeladen werden. Metro ist außerdem in den Applikationsserver GlassFish von Sun integriert. Im Hinblick auf diese Möglichkeiten, werden die Beispiele in diesem Buch auf vier unterschiedliche Arten in Betrieb genommen („deployt"):

- *Nur unter Version 6 der Java Standard Edition (SE 6)*: Dieser Ansatz gestattet Ihnen, Webservices und ihre Clients ohne viel Aufhebens zu schreiben und in Betrieb zu nehmen. Die einzige benötigte Software ist das Java Software Development Kit (SDK) ab SE 6. Webservices können mühelos mit Hilfe der Klassen `javax.xml.ws.Endpoint` („`Endpoint`-Publisher") und `HttpServlet` deployt werden. Die ersten Beispiele werden per `Endpoint`-Publisher in Betrieb genommen.
- *Unter der SE 6 mit dem aktuellen Metro-Release*: Dieser Ansatz nutzt die Vorteile durch Eigenschaften und Fähigkeiten der Metro-Distribution, die noch nicht in der SE 6 verfügbar sind. In der Regel wird das Schreiben von Webservices und Clients mit jedem Metro-Release einfacher. Das aktuelle Metro-Release gibt insbesondere die Richtung an, in die sich die JAX-WS entwickelt. Das Metro-Release kann außerdem auch mit der SE 5 kombiniert werden, falls SE 6 nicht in Frage kommt.
- *Unter einem separat betriebsfähigen Tomcat-Webcontainer*: Dieser Ansatz baut auf der Vertrautheit der Java-Entwickler mit separat betriebsfähigen Webcontainern wie Apache Tomcat auf, der freien Referenzimplementierung. Webservices können mit Hilfe eines Webcontainers

im wesentlichen auf dieselbe Weise deployt werden, wie Servlets, JavaServer Pages (JSP) und JavaServer Faces (JSF). Ein separat betriebsbereiter Webcontainer wie Tomcat ist außerdem eine gutes Umfeld, um die containergesteuerte Sicherheit bei Webservices zu diskutieren.

- *Unter dem Applikationsserver GlassFish*: Dieser Ansatz gestattet Webservices in natürlicher Weise die Interaktion mit anderen Komponenten von Enterpriseapplikationen wie ~~Java Message Service (JMS) topics und -queues~~, JNDI-Anbietern (Java Naming and Directory Interface), Datenbanksystemen und den ~~@Entity-Instanzen~~, die zwischen Applikation und Datenbank vermitteln sowie EJB-Containern (Enterprise JavaBean). Der EJB-Container ist wesentlich, da ein Webservice als zustandslose Session-EJB deployt werden kann, so daß Vorteile wie containergesteuerte Threadsicherheit genutzt werden können. GlassFish arbeitet nahtlos mit Metro (auch hinsichtlich der fortgeschrittenen Eigenschaften und Fähigkeiten) sowie mit den gängigen integrierten Entwicklungsumgebungen wie NetBeans und Eclipse zusammen.

[2] Eine attraktive Eigenschaft von JAX-WS besteht darin, daß die API klar vom Deployment getrennt ist. Es gibt mehrere Alternativen, ein und denselben Webservice in Betrieb zu nehmen, um verschiedene Anforderungen zu erfüllen. Die Mittel der SE 6 eignen sich zum Lernen, Entwickeln und sogar für leichtgewichtiges Deployment. Ein separat betriebsbereiter Webcontainer wie Tomcat bietet darüber hinausgehende Unterstützung. Ein Java-Applikationserver wie GlassFish fördert die mühelose Integration von Webservices und anderen Enterprisetechnologien.

Beispielorientierter Ansatz

[3] Die Beispiele sind kurz genug, um die wesentlichen Eigenschaften und Fähigkeiten von JAX-WS sichtbar hervortreten zu lassen, aber auch realistisch genug, um die Fähigkeiten des JAX-WS-Frameworks im produktiven Betrieb vorzuführen. Alle Beispiele sind komplett abgedruckt, inklusive sämtlicher `import`-Anweisungen. Ich beginne stets mit einem relativ kleinen Beispiel und füge anschließend Eigenschaften hinzu oder verändere sie. Die Länge der Beispiele variiert zwischen wenigen Zeilen und mehreren Seiten. Die Beispiele sind bewußt modular. Hatte ich die Wahl zwischen Kürze und Klarheit, so habe ich versucht, mich für die Klarheit zu entscheiden.

[4] Zu jedem Beispiel gehört eine Anleitung zum Übersetzen und zur Inbetriebnahme des Webservice' (auch „Dienstes“) sowie zu dessen Test anhand eines oder mehrerer Clients. Diese Vorgehensweise stellt einerseits die Wahlmöglichkeiten vor, die JAX-WS dem Programmierer zur Verfügung anbietet, unterstützt aber auch die klare und sorgfältige Untersuchung der JAX-WS-Bibliotheken und -Hilfsmittel. Es war mein Ziel, die Beispiele so zu wählen, daß sie als Vorlage für kommerzielle Applikationen verwendet werden können.

[5] Die JAX-WS ist eine umfangreiche API und läßt sich am besten durch eine Mischung aus Überblick und Beispielen beschreiben. Ich möchte die wesentlichen Eigenschaften der Architektur von Webservices erklären, vor allem aber solche Eigenschaften anhand von Beispielen veranschaulichen, die so funktionieren wie im Text beschrieben. Architektur ohne Quelltext ist inhaltslos. Quelltext ohne Architektur ist blind. Mein Ansatz besteht darin, Architektur und Quelltext weitestmöglich zu integrieren.

[6] Webservices sind ein moderner, leichtgewichtiger Ansatz für verteilte Softwaresysteme, also Systeme wie E-Mail oder das WWW, deren Funktionsprinzip die Ausführung unterschiedlicher Softwarekomponenten auf verschiedenen, physikalisch voneinander getrennten Geräten bedingt. Beispiele für solche Geräte rangieren von großen Servern über Desktop-PCs bis hin zu verschiedenen Typen von Organizern. Verteilte Systeme sind kompliziert, da sie aus vernetzten Komponenten bestehen. Nichts ist frustrierender als ein Beispiel im Kontext verteilter Systeme, das nicht so funktioniert, wie es soll und die Suche nach dem Fehler ist oft mühsam. Mein Ansatz besteht aus diesem Grund

darin, vollständige, funktionstüchtige Beispiele mit kurzen präzisen Anleitungen zu verwenden, wie Sie die Applikationen konfigurieren und in Betrieb nehmen können. Der gesamte Quelltext der Beispiele steht auf der Webseite zu diesem Buch zum Herunterladen zur Verfügung: *http://www.oreilly.com/catalog/9780596521127*. Meine E-Mailadresse ist *kalin@cdm.depaul.edu*. Wenn Sie Fehler entdecken, teilen Sie sie mir bitte mit.

Übersicht über die einzelnen Kapitel

[7] Die folgende Liste vermittelt einen Überblick über die sieben Kapitel dieses Buches:

- Kapitel 1 „Einführung in Java-Webservices“: Dieses Kapitel beginnt mit einer Definition des Begriffs „Webservice“, die insbesondere die Unterscheidung zwischen SOAP-basierten Diensten und solchen im REST-Stil berücksichtigt. Das Kapitel konzentiert sich anschließend auf die Grundzüge des Schreibens, Deployments und Gebrauchs SOAP-basierter Webservices mit den Mitteln der Java Standard Edition. Es werden Webservice-Clients in Perl, Ruby und Java vorgestellt, um die Sprachtransparenz von Webservices zu veranschaulichen. Darüberhinaus stellt das Kapitel die SOAP-API von Java vor und gibt verschiedene Möglichkeiten an, um den Datenverkehr von Webservices auf der Übertragungsebene abzuhören. Die Beziehung zwischen der SE 6 und Metro wird ausgearbeitet.

- Kapitel 2 „WSDL-Dokumente“: Dieses Kapitel konzentiert sich auf den Dienstkontrakt (*service contract*), bei SOAP-basierten Webservices ein WSDL-Dokument (Web Services Description Language). Standardthemen wie der Stil eines Webservice' (Dokument- beziehungsweise RPC-Stil) und die Zeichenkodierung (`literal` beziehungsweise `encoded`) werden behandelt. Das Kapitel widmet sich auch der verbreiteten aber inoffiziellen Unterscheidung zwischen der verpackten und der unverpackten Variante des Dokumentstils. Alle Gegenstände werden anhand von Beispielen erklärt, darunter auch Java-Clients für den E-Commerce-Dienst von Amazon. Das Kapitel erläutert, wie das Kommando `wsimport` die Entwicklung eines Clients für einen kommerziellen Webservice erleichtert und welche Rolle das Kommando `wsgen` bei der Unterscheidung zwischen Dokument- und RPC-Stil spielt. Auch die Grundzüge der Java Architecture for XML Binding (JAX-B) werden beschrieben.

- Kapitel 3 „Verarbeitung von SOAP-Nachrichten“: Dieses Kapitel führt in das Thema SOAP-Behandler und logische Behandler ein, die dem service- oder clientseitigen Programmierer direkten Zugriff auf die gesamte SOAP-Nachricht beziehungsweise nur die Nutzdaten gewähren. Der Aufbau von SOAP-Nachrichten wird behandelt, ebenso die Unterschiede zwischen den Protokollversionen 1.1 und 1.2 sowie die messaging architecture. Verschiedene Beispiele veranschaulichen, wie SOAP-Nachrichten zugunsten der Applikationslogik verarbeitet werden können. Das Kapitel erklärt darüberhinaus die Zugriffs- und Manipulationsmöglichkeiten der JAX-WS auf Nachrichten in der Transportebene (etwa eine typische HTTP-Nachricht mit SOAP-Envelope als Nutzlast bei einem SOAP-basierten Dienst). Das Kapitel endet mit einem Abschnitt über die JAX-WS-Unterstützung für den Transport byteorientierter Daten unter Hervorhebung des MTOM (SOAP Message Transmission Optimization Mechanism).

- Kapitel 4 „Dienste im REST-Stil“: Dieses Kapitel beginnt mit einer technischen Betrachtung der Bestandteile von Diensten im REST-Stil und nähert sich zügig den ersten Beispielen. Das Kapitel beschreibt verschiedene Ansätze für die Inbetriebnahme in Java geschriebener Dienste im REST-Stil, darunter mit `@WebServiceProvider` annotierte Klassen, `HttpServlets`, Jersey und traditionelle Java-Objekte (POJOs) sowie Restlets. WADL-Dokumente (Web Application Description Language) als Dienstkontrakte werden mit Hilfe von Beispielen behandelt. Auch die JAX-P-Bibliothek (Java API for XML Processing) zur vereinfachten XML-Verarbeitung

wird behandelt. Das Kapitel zeigt diverse Beispiele für Java-Clients zu realen Diensten im REST-Stil, darunter Dienste von Yahoo!, Amazon und Tumblr.

- Kapitel 5 „Sicherheit“: Das Kapitel beginnt mit einem Überblick über Sicherheitsanforderungen an reale Webservices, gleichsam SOAP-basiert oder im REST-Stil. Die Übersicht deckt zentrale Themen wie die gegenseitige Authentifizierung und den Datenschutz bei Nachrichten, rollenbasierte Sicherheit und WS-Security ab. Die Beispiel erklären die Sicherheitseigenschaften der Übertragungsebene, insbesondere bei HTTPS. Die containergesteuerte Sicherheit wird anhand von Beispiel eingeführt, die mit Hilfe des separat betriebsfähigen Tomcat-Webcontainers in Betrieb genommen werden. Die in diesem Kapitel eingeführten Sicherheitsgrundlagen werden im nächsten Kapitel erweitert.

- Kapitel 6 „Java-Webservices in Java-Applikationsservern“: Das Kapitel beginnt mit einem Rundgang um die Ausstattung eines Java-Applikationsservers (JAS): ein EJB-Container, ein Nachrichtenübermittlungssystem, ein Namensdienst, ein integriertes Datenbanksystem und so weiter. Das Kapitel bietet eine Vielzahl von Beispielen, darunter ein SOAP-basierter Dienst, der als zustandslose Sitzungs-EJB implementiert wird, ein Webservice und ein Webservice-Provider, die mit Hilfe des eingebetteten Tomcat deployt werden, ein mit einer traditionellen website application zusammen deployter Webservice, ein mit dem JMS (Java Message Service) verknüpfter Webservice, ein Webservice that uses an @Entity to read and write über das in GlassFish integrierte Java-Datenbanksystem sowie eine WS-Security Applikation unter GlassFish.

- Kapitel 7 „*Beyond the Flame Wars*“: Ein sehr kurzes Kapitel, welches sich dem Streit zum die SOAP- beziehungsweise REST-Orientierung bei Webservice annimmt. Mein Ziel besteht darin, beide Ansätze gut zu heißen. Beide sind dem überlegen, was zuvor da war. Das Kapitel skizziert die Entwicklung moderner Webservices beginnend bei DCE/RPC in den frühen 1990er Jahren über CORBA und DCOM bis hin zu den Frameworks Java Enterprise Edition und .NET. Das Kapitel erklärt, warum beide Ansätze für Webservice besser sind als die Architektur der verteilten Objekte, die einst bei verteilten Softwaresystemen vorherrschend war.

Freie Wahl der Entwicklungsumgebung und sonstiger Hilfsmittel

[8] Java-Entwickler haben eine breite Auswahl an produktivitätsunterstützenden Werkzeugen, wie Ant oder Maven für Build-Skripte und integrierte Entwicklungsumgebungen wie Eclipse, NetBeans oder IntelliJ Idea zur Verfügung. Build-Skripte und integrierte Entwicklungsumgebungen steigern die Produktivität, indem sie „schmutzige“ Details verbergen. Im produktiven Betrieb sind solche Werkzeuge und integrierte Entwicklungsumgebungen sinnvoll. Beim Lernen dagegen, besteht das Ziel *gerade darin*, diese „schmutzigen“ Details zu verstehen, so daß das erworbene Verständnis während der unvermeidlichen Perioden der Fehlersuche und Applikationspflege vorteilhaft eingebracht werden kann. Dementsprechend verhält sich mein Buch Build-Werkzeugen und integrierten Entwicklungsumgebungen gegenüber neutral. Fühlen Sie sich frei, Ihre Hilfsmittel so zu wählen, wie es Ihren Bedürfnissen am besten entgegenkommt. Meine Anleitungen beschreiben das Übersetzen des Quelltextes, das Deployment und den Aufruf per Kommandozeile, so daß Details wie etwa Ergänzungen des Klassenpfades und Kommandozeilenschalter/-optionen klar werden. Kein Aspekt eines Beispiels hängt von einem bestimmten Build-Werkzeug oder einer bestimmten integrierten Entwicklungsumgebung ab.

Konventionen in diesem Buch

[9] Folgende typographische Konventionen treten in diesem Buch auf:

- *Geneigte Schrift* zeigt neue Begriffe, URLs, Dateinamen, Dateiendungen und Hervorhebungen an.
- *Fette Schrift* zeigt das Vorkommen eines Eintrages im Stichwortverzeichnis im Fließtext an.
- `Sperrschrift` wird sowohl für Beispiele, als auch bei Quelltextbausteinen im Fließtext benutzt, etwa bei Feldern, lokalen Variablen und Schlüsselworten.
- *`Unterstrichene Sperrschrift`* wird in Beispielen benutzt, um besonders wichtige Abschnitte hervorzuheben und im Fließtext, um Akronyme einzuführen.

Rahmen mit oder ohne Überschrift

kennzeichnen Tipps, Anregungen oder allgemeine Bemerkungen.

Verwendung der Codebeispiele

[10] Dieses Buch soll Ihnen vor allem bei der Arbeit helfen. Den Code, den Sie hier finden, dürfen Sie generell in Ihren Programmen und Dokumentationen verwenden. Sie brauchen uns deswegen nicht um Erlaubnis zu fragen, es sei denn, Sie reproduzieren einen großen Teil des Codes. Einige Beispiele: Wenn Sie ein Programm schreiben, in dem mehrere Codestücke aus diesem Buch genutzt werden, ist dafür keine Erlaubnis erforderlich; wenn Sie hingegen eine CD-ROM mit Beispielen aus O'Reilly-Büchern verkaufen oder verteilen, müssen Sie eine Erlaubnis einholen. Wenn Sie eine Frage beantworten, indem Sie dieses Buch und den Beispielcode daraus zitieren, benötigen Sie keine Erlaubnis; wenn Sie einen größeren Teil unseres Beispielcodes in Ihre Produktdokumentation aufnehmen, müssen Sie eine Erlaubnis haben.

[11] Wie freuen uns, wenn Zitate aus unserem Buch mit Quellenangaben versehen werden, aber wir verlangen es nicht. Zu einer Quellenangabe gehören normalerweise der Titel, Autor, Verlag und die ISBN-Nummer. Zum Beispiel: „Java Web Services: Up and Running, by Martin Kalin. Copyright 2009 Martin Kalin, 978-0-596-52112-7".

[12] Wenn Sie nach eigener Einschätzung mehr Codebeispiele verwenden als erlaubt, kontaktieren Sie uns bitte unter *permissions@oreilly.com*.

Kontakt mit O'Reilly

[15] Richten Sie Bemerkungen und Fragen zu diesem Buch an den Verlag:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

Der Verlag unterhält eine Webseite zur englischen Originalausgabe dieses Buches, auf der Errata, Beispiele und zusätzliche Informationen zu finden sind. Sie finden die Webseite unter der Adresse *http://www.oreilly.com/catalog/9780596521127*. Bemerkungen und technische Fragen können auch per E-Mail gesendet werden: *bookquestions@oreilly.com*. Informationen über unsere Bücher, Konferenzen, das Resource Center und das O'Reilly-Network finden auf der Webseite des Verlags: *http://www.oreilly.com*.

Danksagungen

[16] Christian A. Kenyeres, Greg Ostravich, Igor Polevoy und Ken Yu waren so freundlich, dieses Buch durchzusehen und aufschlußreiche Anregungen zu seiner Verbesserung zu geben. Sie haben das Buch besser gemacht, als es andernfalls geworden wäre. Ich danken ihnen herzlich für die Zeit und Mühe, die sie in dieses Projekt investiert haben. Die verbliebenen Mängel gehen selbstverständlich alleine auf mein Konto.

[17] Mein Dank gilt außerdem Mike Loukides, meinem ersten Ansprechpartner bei O'Reilly Media, für seine Rolle auf dem Weg meines anfänglichen Vorschlages durch den Prozeß, der zu seinem Einverständnis geführt hat. Meine Lektorin, Julie Steele, hat mich unschätzbar unterstützt und das Buch wäre ohne ihre Hilfe nicht zustande gekommen. Ich danke darüberhinaus allen Mitarbeitern von O'Reilly Media die hinter den Kulissen an diesem Projekt gearbeitet haben.

[18] Dieses Buch ist für Janet.

Kapitel 1

Einführung in Java-Webservices

Inhaltsübersicht

1.1 Was sind Webservices?

[0] Obwohl der Begriff „*Webservice*“ verschiedene, unpräzise und sich kontinuierlich weiterentwickelnde Bedeutungen hat, genügt es, einige für Webservices typische Eigenschaften zu betrachten, um einen Webservice sowie einen Client (in der englischsprachigen Literatur auch „Consumer“ oder „Requester“) zu schreiben. Wie der Terminus andeutet, ist ein Webservice eine Art *webfähige Applikation*, das heißt eine typischerweise über *HTTP (Hypertext Transfer Protocol)* betriebene Applikation. Ein Webservice ist somit eine verteilte Anwendung, deren Komponenten auf verschiedenen Geräten deployt und ausgeführt werden können. Beispielsweise könnte ein Webservice für Stock Picking (Zielkauf von Aktien) aus mehreren Komponenten bestehen, die auf separaten industriellen Servern laufen. Der Webservice könnte von PCs, Organizern oder sonstigen Geräten in Anspruch genommen werden.

[1] Webservices lassen sich grob in zwei Gruppen einteilen: *SOAP*-basierte Dienste und solche im REST-Stil. Die Unterscheidung zwischen beiden Gruppen ist unscharf, da SOAP-basierte Dienste ein Spezialfall der Dienste im REST-Stil sind, wie eines der späteren Beispiele zeigen wird [(Seite ??)]. Die Abkürzung SOAP stand ursprünglich für „Simple Object Access Protocol“, repräsentiert durch einen glücklichen Zufall heute aber das „Service Oriented Architecture (SOA) Protocol“. Die Deutung der Abkürzung SOA ist keine einfache Aufgabe. Unbestreitbar ist dagegen: Was auch immer sich hinter der Abkürzung SOA verbirgt; Webservices erfüllen beim SOA-Ansatz eine zentrale Funktion, sowohl beim Softwareentwurf als auch bei der Softwareentwicklung. (Dies ist nur zum Teil ironisch gemeint. Die Abkürzung SOAP ist offiziell kein Akronym mehr und SOAP und SOA können nicht länger getrennt voneinander existieren.) SOAP ist zunächst nur ein *XML-Dialekt (Extensible Markup Language)*, dessen Dokumente Nachrichten sind. Bei SOAP-basierten Webservices ist SOAP ein zumeist unbemerkter Teil der Infrastruktur. In der typischen, durch das *Nachrichtenaustauschmuster* Request/Response beschriebenen Situation, sendet beispielsweise die dem Client unterliegende SOAP-Bibliothek eine SOAP-Nachricht als Anfrage und die dem Webservice unterliegende SOAP-Bibliothek sendet eine weitere SOAP-Nachricht als korrespondierende Antwort zurück. Die Quelltexte von Client und Webservice können, wenn überhaupt, einige wenige Hinweise zum unterliegenden SOAP-Mechanismus enthalten (Abbildung 1.1).

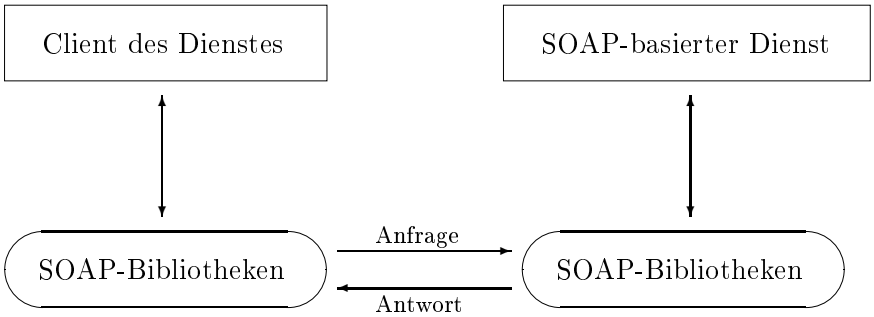

Abbildung 1.1: Architektur eines typischen SOAP-basierten Dienstes.

[2] Die Abkürzung *REST* steht für „Representational State Transfer“. *Roy Fielding*, einer der Hauptautoren der HTTP-Spezifikation, rief dieses Akronym in seiner Dissertation ins Leben, um eine architektonische Erscheinungsform beim Entwurf von Webservices zu beschreiben. Der SOAP-Ansatz umfaßt Standards des *W3C (World Wide Web Consortium)*, Werkzeuge und großzügige Softwarebibliotheken. Der REST-Stil beinhaltet dagegen keine Standards, nur wenige Werkzeuge und spärliche Bibliotheken. Der REST-Ansatz wird häufig als Gegenmittel zur schleichenden Komplexität SOAP-basierter Webservices angesehen. Dieses Buch behandelt den SOAP-basierte Webservices und Dienste im REST-Stil gleichermaßen und beginnt mit dem SOAP-Ansatz.

[3] Abgesehen vom Testbetrieb ist der Client eines Webservice', gleichsam SOAP-basiert oder im REST-Stil, nur selten ein Webbrowser, sondern vielmehr eine Applikation ohne graphische Benutzerschnittstelle. Der Client kann in jeder Sprache geschrieben werden, die entsprechende unterstützende Bibliotheken zur Verfügung stellt. Diese *Sprachtransparenz* (auch Sprachneutralität) ist sogar einer der hauptsächlichen Anreize für Webservices: der Dienst und seine Clients müssen nicht in derselben Sprache geschrieben sein. Die Sprachtransparenz ist der Schlüssel zur *Interoperabilität* von Webservices, das heißt der Fähigkeit von Webservices und deren Clients, nahtlos über verschiedene Programmiersprachen, unterstützende Bibliotheken und Plattformen hinweg zu interagieren. Die Clients der Java-Webservices in diesem Buch sind in verschiedenen Sprachen wie *C#*, *Perl* oder *Ruby* geschrieben, um diese Sprachtransparenz hervorzuheben. Umgekehrt werden auch Java-Clients Anfragen an Webservices senden, die in anderen Sprachen geschrieben sind und es gibt Fälle, in denen die Sprache des Dienstes unbekannt ist.

[4] Hinter der Sprachtransparenz verbirgt sich natürlich keinerlei Magie. Ist ein SOAP-basierter, in Java geschriebener Webservice in der Lage, mit Perl- oder Ruby-Clients kommunizieren, so muß ein Mittler vorhanden sein, der sich um die Unterschiede in den Datentypen zwischen der Webservice- und der Clientsprache kümmert. Diese Aufgabe fällt der XML-Technologie zu, die das Austauschen und die Verarbeitung strukturierter Dokumente unterstützt. Bei einem typischen SOAP-basierten Webservice, sendet der Client transparent ein SOAP-Dokument als Anfrage an den Webservice, der transparent ein anderes SOAP-Dokument als Antwort zurücksendet. Bei einem Dienst im REST-Stil würde der Client eine gewöhnliche HTTP-Anfrage an den Webservice senden und ein entsprechendes XML-Dokument zurückerhalten.

[5] Webservices unterscheiden sich in einigen Eigenschaften von anderen verteilten Softwaresystemen. Drei dieser Eigenschaften sind:

- *Offene Infrastruktur*: Webservices werden mit Hilfe omnipräsenter, allgemein verstandener und anbieterunabhängiger Industriestandardprotokolle wie HTTP oder XML deployt. Webservices bauen auf Netzwerk, Datenformatierung, Sicherheit und anderen bereits vorhandenen Infrastrukturkomponenten auf. Das senkt die Einstiegskosten und begünstigt die Interoperabilität zwischen Diensten.
- *Sprachtransparenz*: Webservices und ihre Clients können zusammenarbeiten, selbst wenn sie in verschiedenen Programmiersprachen geschrieben sind. Viele Sprachen, darunter *C/C++*, *C#*, *Java*, *Perl*, *Python* und *Ruby*, stellen Bibliotheken, Hilfsmittel und sogar Frameworks zur Verfügung, um Webservices zu unterstützen.
- *Modulares Design*: Webservices sind dazu bestimmt, modulare Entwürfe zu implementieren, damit neue Webservices durch Integration und Schichtung existierender Webservices konstruiert werden können. Stellen Sie sich zum Beispiel einen Bestandsverfolgungsdienst vor, der mit einem Online-Bestelldienst zusammenarbeitet, in dem ein (kombinierter neuer) Dienst automatisch die entsprechenden Produkte je nach Lagerbestand nachbestellt.

1.1.1 Wozu sind Webservices gut?

[6] Es gibt keine einfache und alleinige Antwort auf diese offensichtliche Frage. Dennoch sind die Vorteile und Verheißungen des Webservice-Konzeptes im wesentlichen klar umrissen. Moderne Softwaresysteme sind in vielen verschiedenen Programmiersprachen geschrieben, laufen auf zahlreichen Plattformen und es liegt nahe, daß diese Vielfalt noch zunehmen wird. Große wie kleine Institutionen haben erhebliche Mittel in ältere Softwaresysteme investiert, deren Funktion nützlich und eventuell sogar für den Betrieb wesentlich ist. Nur wenige dieser Institutionen haben den Wunsch und die Resourcen, sowohl finanziell als auch an Mitarbeitern, um ihre älteren Softwaresysteme umzuschreiben.

[7] Es geschieht nur selten, daß ein Softwaresystem in der Isolation reift. Ein typisches Softwaresystem muß mit anderen Systemen zusammenarbeiten, die auf verschiedenen Hosts laufen und in unterschiedlichen Sprachen geschrieben sind. Interoperabilität ist nicht nur eine langfristige Aufgabe, sondern auch eine aktuelle Anforderung an Software im produktiven Betrieb.

[8] Webservices erfüllen diese Anforderungen unmittelbar, da sie an erster Stelle sprach- und plattformneutral sind. Wird ein älteres, in COBOL entwickeltes Softwaresystem mit Unterstützung eines Webservice' erreichbar, so wird dieses System befähigt, mit Clients zusammenzuarbeiten, die in anderen Programmiersprachen geschrieben sind.

[10] Webservices sind in natürlicher Weise *verteilte Systeme*, die zumeist über HTTP kommunizieren, sich aber auch auf andere gängige Transportprotokolle stützen können. Die „Nutzlast“ bei der

Kommunikation von Webservices sind strukturierte Texte, das heißt XML-Dokumente, die ausgewertet, transformiert, dauerhaft gespeichert oder auf andere Weise mit weit verbreiteten und teilweise sogar kostenlosen Werkzeugen verarbeitet werden können. Sofern aus Effizienzgründen erforderlich, können Webservices auch byteorientierte Nutzdaten transportieren. Letztendlich sind Webservices a work in progress mit realen verteilten Systemen als Testumgebung. Aus allen diesen Gründen sind Webservices ein wesentliches Hilfsmittel im Werkzeugkasten jedes modernen Programmierers.

[11] Die folgenden Beispiele, in diesem und den folgenden Kapiteln, sind einfach genug, um wesentliche Eigenschaften von Webservices isoliert darzustellen, aber auch realistisch genug, um die durch Webservices in der Softwareentwicklung verfügbare Kraft und Flexibilität vorzuführen.

1.2 Erstes Beispiel: Der TimeServer-Dienst

[12] Das erste Beispiel ist ein SOAP-basierter, in Java geschriebener Webservice mit Clients in Perl, Ruby und Java. Der Webservice besteht aus einem Interface und einer Implementierung.

1.2.1 Service Endpoint Interface und Service Implementation Bean

[13] Der erste, in Java geschriebene Webservice läßt sich, wie die meisten übrigen in diesem Buch, mit den Mitteln der *SE 6 (Version 6 der Java Standard Edition)* oder höher ohne zusätzliche Software übersetzen und deployen. Alle für die Übersetzung, Ausführung und Nutzung von Webservices erforderlichen Bibliotheken sind in der SE 6 enthalten, insbesondere die *JAX-WS (Java API for XML-Web Services)*. JAX-WS wiederum, unterstützt sowohl SOAP-basierte Dienste als auch Dienste im REST-Stil. Anstelle der Abkürzung JAX-WS wird gelegentlich auch die Abkürzung *JWS (Java Web Service)* gebraucht. Die aktuelle Versionsbezeichnung von JAX-WS lautet 2.x. Dies ist ein wenig irreführend, da die 1.x Versionen als JAX-RPC bezeichnet wurden. JAX-WS erhält die Fähigkeiten von JAX-RPC, erweitert sie aber in erheblichem Ausmaß.

[14] Ein SOAP-basierter Webservice kann zwar in Form einer einzigen Klasse implementiert werden, aber es hat sich bewährt, die Methoden (oder auch „Operationen“) des Webservice' in einem Interface zu *deklarieren* und eine separate Implementierung vorzunehmen, in der die deklarierten Methoden *definiert* werden. Das Interface wird als Service Endpoint Interface (SEI) bezeichnet, die Implementierung als Service Implementation Bean (SIB). Die SIB kann entweder ein *POJO (Plain Old Java Object)* oder eine zustandslose *Sitzungs-EJB (Enterprise JavaBean)* sein. Kapitel 6 behandelt den Applikationsserver GlassFish und zeigt, wie sich ein Webservice als EJB implementieren läßt. Bis dorthin werden alle SOAP-basierten Webservices als POJOs implementiert, das heißt als Objekte gewöhnlicher Java-Klassen. Diese Webservices werden zunächst mit Hilfe von Klassen aus der Standardbibliothek der SE 6 in Betrieb genommen, später dann mit Hilfe eines separaten Tomcat (Webcontainer) oder GlassFish.

Die Version 6 der Java Standard Edition, die JAX-WS-Schnittstelle und das Metro-Release

Die SE 6 beinhaltet JAX-WS. JAX-WS „lebt“ aber außerhalb der SE 6 und hat ein separates Entwicklerteam. Die Speerspitze von JAX-WS ist der *Metro Web Service Stack* (*https://wsit.dev.java.net*), der das Tango-Projekt beinhaltet, welches die Interoperabilität zwischen der Java-Plattform und der *Windows Communication Foundation* (WCF), auch unter der Bezeichnung „Indigo“ bekannt, verbessert. Die Interoperabilitätsintiative nennt sich *WSIT* (Web Services Interoperability Technologies). Die aktuelle Metro-Version von JAX-WS, von nun als „Metro-Release“ bezeichnet,

ist der Version von JAX-WS in der SE 6 typischerweise voraus. Mit Update 4 änderte sich die Versionsbezeichnung der JAX-WS in der SE 6 von 2.0 auf 2.1, deutet also erhebliche Verbesserungen an.

Die häufigen Metro-Releases beheben Fehler, ergänzen Eigenschaften und Fähigkeiten, erleichtern dem Programmierer die Arbeit und bauen JAX-WS aus. Für den Anfang ist es mein Ziel, JAX-WS mit so wenig Aufhebens wie möglich einzuführen. Im Augenblick genügt der durch die SE 6 verfügbare Umfang von JAX-WS. Gelegentlich ist für ein Beispiel etwas mehr Arbeit notwendig, als unter dem aktuellen Metro-Release erforderlich wäre. In diesen Fällen besteht meine Vorgehensweise darin, zu erläutern was tatsächlich geschieht, bevor die kürzere Fassung mit Hilfe von Metro vorgestellt wird.

Die Metro-Homepage gestattet einfaches Herunterladen. Einmal installiert, befindet sich das Metro-Release in einem Verzeichnis namens *jaxws-ri*. Diejenigen unter den folgenden Beispielen, die vom Metro-Release Gebrauch machen, erwarten eine Umgebungsvariable namens `$METRO_HOME`, die auf das Installationsverzeichnis *jaxws-ri* verweist. Die Buchstaben *ri* sind übrigens die Abkürzung für "*Referenzimplementierung*".

Schließlich bietet das heruntergeladene Metro-Release eine Möglichkeit, JAX-WS auch unter Version 5 der Java Standard Edition (JSE 5) zu nutzen. JAX-WS benötigt mindestens die SE 5, da Annotation erst ab dieser Version unterstützt werden.

---

[15] Das erste Beispiel zeigt das SEI eines Webservice', der das aktuelle Datum und die aktuelle Uhrzeit entweder als Zeichenkette oder als Anzahl der verstrichenen Millisekunden seit Beginn der Unix-Epoche (1. Januar 1970, 0:00 Uhr) zurückgibt:

Beispiel 1.1: Service Endpoint Interface (SEI) des TimeServer-Dienstes.

```
package ch01.ts;  // time server

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

/**
 *  The annotation @WebService signals that this is the
 *  SEI (Service Endpoint Interface). @WebMethod signals
 *  that each method is a service operation.
  *
 *  The @SOAPBinding annotation impacts the under-the-hood
 *  construction of the service contract, the WSDL
 *  (Web Services Definition Language) document. Style.RPC
 *  simplifies the contract and makes deployment easier.
 */
@WebService
@SOAPBinding(style = Style.RPC) // more on this later
public interface TimeServer {

    @WebMethod String getTimeAsString();
    @WebMethod long getTimeAsElapsed();

}
```

Das zweite Beispiel zeigt die SIB, die dieses SEI implementiert:

Beispiel 1.2: Service Implementation Bean (SIB) des TimeServer-Dienstes.

```
package ch01.ts;
```

```
import java.util.Date;
import javax.jws.WebService;

/**
 *  The @WebService property endpointInterface links the
 *  SIB (this class) to the SEI (ch01.ts.TimeServer).
 *  Note that the method implementations are not annotated
 *  as @WebMethods.
 */
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl implements TimeServer {

    public String getTimeAsString() { return new Date().toString(); }
    public long getTimeAsElapsed() { return new Date().getTime(); }

}
```

Beide Dateien werden, wie gewohnt, im aktuellen Arbeitsverzeichnis übersetzt, hier in dem Verzeichnis, welches das Unterverzeichnis *ch01* enthält. Das Prozentzeichen (%) repräsentiert die Eingabeaufforderung:

```
% javac ch01/ts/*.java
```

1.2.2 Inbetriebnahme des Dienstes per Endpoint-Publisher

[16] Nachdem das SEI und die SIB übersetzt sind, kann der Webservice in Betrieb genommen werden. Im produktiven Betrieb würde ein Java-Appliktionsserver wie *BEA-Weblogic*, *GlassFish*, *JBoss* oder *WebSphere* zum Einsatz kommen. Für die Entwicklungsphase und im kleineren produktiven Betrieb genügt eine einfache Java-Applikation. Das folgende Beispiel zeigt die Applikation zur Inbetriebnahme des *TimeServer*-Dienstes mit Hilfe der statischen `Endpoint`-Methode `publish()` (einen sogenannten „*Endpoint-Publisher*“):

Beispiel 1.3: Inbetriebnahme des TimeServer-Dienstes per Endpoint.publish().

```
package ch01.ts;

import javax.xml.ws.Endpoint;

/**
 * This application publishes the Web service whose
 * SIB is ch01.ts.TimeServerImpl. For now, the
 * service is published at network address 127.0.0.1.,
 * which is localhost, and at port number 9876, as this
 * port is likely available on any desktop machine. The
 * publication path is /ts, an arbitrary name.
 *
 * The Endpoint class has an overloaded publish method.
 * In its two-argument version, the first argument is the
 * publication URL as a string and the second argument is
 * an instance of the service SIB, in this case
 * ch01.ts.TimeServerImpl.
 *
 * The application runs indefinitely, awaiting service requests.
 * It needs to be terminated at the command prompt with control-C
 * or the equivalent.
 *
 * Once the applicatation is started, open a browser to the URL
 *
 *     http://127.0.0.1:9876/ts?wsdl
```

```
 *
 * to view the service contract, the WSDL document. This is an
 * easy test to determine whether the service has deployed
 * successfully. If the test succeeds, a client then can be
 * executed against the service.
 */
public class TimeServerPublisher {
    public static void main(String[] args) {
      // 1st argument is the publication URL
      // 2nd argument is an SIB instance
      Endpoint.publish("http://127.0.0.1:9876/ts", new TimeServerImpl());
    }
}
```

Nach dem Übersetzen läßt sich die Applikation wie folgt aufrufen:

```
% java ch01.ts.TimeServerPublisher
```

Verarbeitung von Anfragen durch den `Endpoint`-Publisher

Ein `Endpoint`-Publisher verarbeitet „in seiner Grundeinstellung“ stets höchstens eine Anfrage. Das ist in Ordnung, um Webservices während der Entwicklungsphase an den Start zu bringen. Bleibt allerdings die Verarbeitung einer Anfrage „hängen“, werden die Anfragen aller übrigen Clients effektiv blockiert. Das Beispiel in Abschnitt 1.10 zeigt, wie ein `Endpoint`-Publisher mehrere Anfragen gleichzeitig verarbeiten kann, so daß eine hängengebliebene Anfrage die anderen nicht blockiert.

1.2.3 Testen des Dienstes mit Hilfe eines Webbrowsers

[17] Der deployte Webservice läßt sich beispielsweise testen, in dem sein WSDL-Dokument (Web Services Description Language), ein automatisch generierter „Dienstkontrakt“, in einem Webbrowser geöffnet wird. (WSDL wird übrigens „whiz dull“ ausgesprochen.) In die Adreßzeile des Browserfensters wird eine zweiteilige URL eingetragen. Der erste Teil ist die in der Java-Applikation `TimeServerPublisher` festgelegte URL (*http://127.0.0.1:9876/ts*). Daran schließt sich der zweite Teil an, nämlich die Zeichenfolge *?wsdl*, in großen oder kleinen Buchstaben oder auch in gemischter Schreibweise. Die gesamte URL lautet *http://127.0.0.1:9876/ts?wsdl*. Das folgende Beispiel zeigt das im Webbrowser wiedergegebene WSDL-Dokument:

Beispiel 1.4: WSDL-Dokument des TimeServer-Dienstes.

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://ts.ch01/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    targetNamespace="http://ts.ch01/"
    name="TimeServerImplService">
  <types></types>

  <message name="getTimeAsString"></message>
  <message name="getTimeAsStringResponse">
    <part name="return" type="xsd:string"></part>
  </message>
  <message name="getTimeAsElapsed"></message>
  <message name="getTimeAsElapsedResponse">
```

```
    <part name="return" type="xsd:long"></part>
  </message>

  <portType name="TimeServer">
    <operation name="getTimeAsString" parameterOrder="">
      <input message="tns:getTimeAsString"></input>
      <output message="tns:getTimeAsStringResponse"></output>
    </operation>
    <operation name="getTimeAsElapsed" parameterOrder="">
      <input message="tns:getTimeAsElapsed"></input>
      <output message="tns:getTimeAsElapsedResponse"></output>
    </operation>
  </portType>

  <binding name="TimeServerImplPortBinding" type="tns:TimeServer">
    <soap:binding
        style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http">
    </soap:binding>
    <operation name="getTimeAsString">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </input>
      <output>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </output>
    </operation>
    <operation name="getTimeAsElapsed">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </input>
      <output>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </output>
    </operation>
  </binding>

  <service name="TimeServerImplService">
    <port name="TimeServerImplPort" binding="tns:TimeServerImplPortBinding">
      <soap:address location="http://127.0.0.1:9876/ts"></soap:address>
    </port>
  </service>
</definitions>
```

[18] Kapitel 2 untersucht die WSDL im Detail und stellt die Java-Hilfsmittel im Zusammenhang mit dem Dienstkontrakt vor. Im Augenblick verdienen die beiden hervorgehobenen Abschnitte des obigen WSDL-Dokuments unsere Aufmerksamkeit. Der `<portType>`-Abschnitt in der oberen Hälfte faßt die angebotenen Operationen des Webservice' zusammen, hier die Operationen `getTimeAsString()` und `getTimeAsElapsed()`, zwei im SEI deklarierte und in der SIB implementierte Java-Methoden. Das WSDL-Element `<portType>` verhält sich wie ein Java-Interface, indem es die Operationen des Dienstes abstrakt deklariert, aber keine detaillierte Implementierung liefert. Jede Operation des Webservice' besteht aus einer Ein- und einer Ausgabenachricht, Eingabe im Sinne einer Übergabe an den Webservice. Im Betrieb ist jede Nachricht ein SOAP-Dokument. Der `<service>`-Abschnitt in der unteren Hälfte ist der zweite interessante Teil des obigen WSDL-Dokumentes, insbesondere das `location`-Attribut, hier mit dem Wert `http://localhost:9876/ts`. Die URL ist der sogenannte

Endpunkt des Dienstes (*service endpoint*) und informiert Clients darüber, wo der Webservice zu erreichen ist.

[19] Das WSDL-Dokument ist sowohl beim Schreiben als auch bei der Ausführung von Clients eines Webservice' nützlich. Viele Sprachen verfügen über Hilfsmittel, um aus einem WSDL-Dokument Quelltext zur Unterstützung von Clients generieren zu können. Die Version 6 der Java Standard Edition beinhaltet zu diesem Zweck das *`wsimport`-Kommando*. Die früheren Namen des `wsimport`-Kommandos waren aussagekräftiger, nämlich *`wsdl2java`* und *`java2wsdl`*. Im Betrieb kann ein Client das dem Webservice zugehörige WSDL-Dokument auswerten, um wichtige Informationen über die Datentypen der Operationen des Webservice' zu erhalten. Ein Client könnte beispielsweise dem obigen WSDL-Dokument entnehmen, daß die Operation `getTimeAsElapsed()` einen ganzzahligen Wert zurückgibt und keine Argumente erwartet.

[20] Das WSDL-Dokument kann auch über verschiedene andere Kommandos angefordert werden, etwa per *`curl`*. Das Kommando

```
% curl http://localhost:9876/ts?wsdl
```

liefert ebenfalls das WSDL-Dokument.

Vermeiden eines subtilen Problems bei der Implementierung eines Webservice'.

Dieses Beispiel unterscheidet sich von der sehr verbreiteten Vorgehensweise, die SIB des Webservice' (hier die Klasse `TimeServerImpl`) nur über das `endpointInterface`-Attribut der Annotation `@WebService` mit dem SEI (hier das Interface *`TimeServer`*) zu verbinden. Folgendes ist häufig anzutreffen:

```
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl { // implements TimeServer removed
```

Dieser Programmierstil ist beliebt aber nicht sicher. Es ist besser, die `implements`-Klausel stehen zu lassen, damit der Compiler prüfen kann, ob die SIB die im SEI deklarierten Methoden implementiert. Wird die `implements`-Klausel entfernt und die beiden Definitionen der Operationen des Webservice' auskommentiert:

```
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl {

    // public String getTimeAsString() { return new Date().toString(); }
    // public long gettimeAsElapsed() { return new Date().getTime(); }

}
```

läßt sich der Quelltext dennoch übersetzen. Wäre die `implements`-Klausel vorhanden, so würde der Compiler einen groben Fehler melden, da die SIB die im SEI deklarierten Methoden nicht definiert.

1.3 Ein Perl- und ein Ruby-Client für den TimeServer-Dienst

[21] Der erste Client des in Java entwickelten *`TimeServer`*-Dienstes ist in Perl statt in Java geschrieben, um die *Sprachtransparenz* des Webservice-Konzeptes zu demonstrieren. Der zweite Client ist in Ruby geschrieben. Zunächst Perl:

Beispiel 1.5: Perl-Client für den TimeServer-Dienst.

```
#!/usr/bin/perl -w
```

```
use SOAP::Lite;
my $url = 'http://127.0.0.1:9876/ts?wsdl';
my $service = SOAP::Lite->service($url);

print "\nCurrent time is: ", $service->getTimeAsString();
print "\nElapsed milliseconds from the epoch: ", $service->getTimeAsElapsed();
```

Die Ausgabe lautet bei mir:

```
Current time is: Thu Oct 16 21:37:35 CDT 2008
Elapsed milliseconds from the epoch: 1224211055700
```

Das Perlmodul *`SOAP::Lite`* liefert die erforderliche Funktionalität, so daß der Client entsprechende SOAP-Anfragen senden und die resultierenden SOAP-Antworten verarbeiten kann. Die Anfrage-URL, dieselbe URL wie beim Testen des Dienstes im Webbrowser, endet mit einem „Query-String" (einer Argumentliste), der das WSDL-Dokument anfordert. Der Perl-Client erhält das WSDL-Dokument, anhand dessen die `SOAP::Lite`-Bibliothek das entsprechende Dienstobjekt erzeugt (in Perlsyntax referenziert von der skalaren Variablen `$service`). Die `SOAP::Lite`-Bibliothek erhält die benötigten Informationen durch Auswertung des WSDL-Dokuments, insbesondere die Namen der Operationen des Dienstes sowie die beteiligten Datentypen. Abbildung 1.2 zeigt die Architektur.

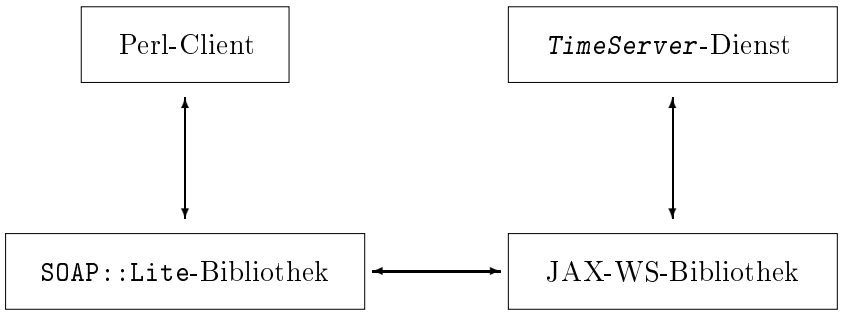

Abbildung 1.2: Architektur des Perl-Clients und des in Java geschriebenen *`TimeServer`*-Dienstes.

[22] Nach der Initialisierung der skalaren Variablen `$url` und `$service` ruft der Perl-Client die Operationen des Dienstes auf. Die SOAP-Nachrichten bleiben unsichtbar. Das nächste Beispiel zeigt einen zum obigen Perl-Client gleichwertigen Ruby-Client:

Beispiel 1.6: Ruby-Client für den TimeServer-Dienst.

```
#!/usr/bin/ruby

# one Ruby package for SOAP-based services
require 'soap/wsdlDriver'
wsdl_url = 'http://127.0.0.1:9876/ts?wsdl'

service = SOAP::WSDLDriverFactory.new(wsdl_url).create_rpc_driver

# Save request/response messages in files named '...soapmsgs...'
service.wiredump_file_base = 'soapmsgs'

# Invoke service operations.
result1 = service.getTimeAsString
result2 = service.getTimeAsElapsed

# Output results.
puts "Current time is: #{result1}"
puts "Elapsed milliseconds from the epoch: #{result2}"
```

1.4 Die unsichtbaren SOAP-Nachrichten

[23] Bei einem SOAP-basierten Webservice veranlaßt der Client typischerweise einen Methodenfernaufruf beim Dienst, indem er eine von dessen Operationen aufruft. Wie bereits in Abschnitt 1.1 beschrieben, folgt der Austausch von Nachrichten zwischen Client und Webservice dem Nachrichtenaustauschmuster Request/Response. Die bei diesem Muster ausgetauschten SOAP-Nachrichten gestatten das Entwickeln von Dienst und Clients in verschiedenen Sprachen. Wir wollen nun die Vorgänge hinter den Kulissen des ersten Beispiels genauer betrachten. Der Perl-Client erzeugt eine *HTTP-Anfrage*, genauer eine formatierte Nachricht, deren Körper aus einer SOAP-Nachricht besteht. Das folgende Beispiel zeigt diese HTTP-Anfrage:

Beispiel 1.7: HTTP-Anfrage an den TimeServer-Dienst.

```
POST http://127.0.0.1:9876/ts HTTP/ 1.1
Accept: text/xml
Accept: multipart/*
Accept: application/soap
User-Agent: SOAP::Lite/Perl/0.69
Content-Length: 434
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
     soap:encodingStyle="http:// schemas.xmlsoap.org/soap/encoding/"
     xmlns:soap="http://schemas.xmlsoap.org/soap/ envelope/"
     xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:tns="http://ts.ch01/"
     xmlns:xsd ="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <tns:getTimeAsString xsi:nil="true" />
  </soap:Body>
</soap:Envelope>
```

[24] Die HTTP-Anfrage ist eine Nachricht mit einer eigenen Struktur. Im Einzelnen:

- Die erste Zeile der HTTP-Anfrage ist die *HTTP-Kopfzeile* und definiert die HTTP-Methode, hier `POST`. Bei Anfragen nach dynamischen Resourcen wie Webservices oder Operationen von Webapplikationen (zum Beispiel einem Servlet) wird typischerweise die HTTP-Methode `POST` verwendet, im Gegensatz zu Anfragen nach statischen HTML-Seiten. Im vorliegenden Fall ist eine `POST`- anstelle einer `GET`-Anfrage erforderlich, da nur die erstere über einen Körper verfügt und somit eine SOAP-Nachricht beinhalten kann. An die HTTP-Methode schließen sich die Anfrage-URL und die Version des HTTP-Protokolls an, die der Client versteht, hier die aktuelle Protokollversion 1.1.

- Nach der HTTP-Kopfzeile folgen die *HTTP-Header*. Diese sind Schlüssel/Wert-Paare, wobei Schlüssel und Wert durch einen Doppelpunkt (`:`) voneinander getrennt sind. Die HTTP-Header können in beliebiger Reihenfolge angeordnet werden. Der Headerschlüssel *`Accept`* (kurz `Accept`-Header) tritt dreimal auf, jeweils mit einem *MIME*-Typ oder MIME-Untertyp (Multipurpose Internet Mail Extensions) als Header-Wert (kurz Wert): `text/xml`, `multipart/*` und `application/soap`. Diese drei Schlüssel/Wert-Paare geben an, daß der Client eine Antwort in einem beliebigen XML-Dialekt, eine Antwort mit beliebig vielen Anhängen beliebigen Typs (eine SOAP-Nachricht kann über beliebig viele Anhänge verfügen) oder ein SOAP-Dokument akzeptiert. Der `SOAPAction`-Header tritt bei HTTP-Anfragen an Webservices häufig auf, wo-

bei die leere Zeichenkette als Wert gewählt wird, wie in diesem Beispiel. Der Wert kann aber auch der Name der angeforderten Operation des Webservice' sein.

- Zwei Zeilenvorschübe (Carriage Return, Line Feed, CRLF), entsprechend zwei „\n"-Zeichen in Java, trennen die HTTP-Header vom Körper der HTTP-Anfrage, der bei `POST`-Anfragen verlangt wird, aber leer sein darf. Im vorliegenden Fall enthält der Körper der HTTP-Anfrage das *SOAP-Dokument*, häufig auch als *„SOAP-Envelope"* bezeichnet, da das *Wurzel- oder Dokumentelement* (äußerstes Element) `<Envelope>` heißt. Der SOAP-Body in diesem SOAP-Envelope enthält ein einzelnes Element mit dem *lokalen Namen* `getTimeAsString`, dem Namen der Operation des Dienstes, den der Client aufzurufen wünscht. Der SOAP-Envelope der Anfrage ist in diesem Beispiel einfach, weil die angeforderte Operation keine Argumente erwartet.

[25] Auf der Seite des Dienstes verarbeiten die unterliegenden Java-Bibliotheken die HTTP-Anfrage, extrahieren den SOAP-Envelope, ermitteln die angeforderte Operation, rufen die zugehörige Methode `getTimeAsString()` auf und generieren die entsprechende SOAP-Nachricht, um den Rückgabewert der Methode an den Client zurückzusenden. Das nächste Beispiel zeigt die HTTP-Antwort auf die Anfrage an den `TimeServerImpl`-Webservice im vorigen Beispiel:

Beispiel 1.8: HTTP-Antwort des TimeServer-Dienstes'.

```
HTTP/1.1 200 OK
Content-Length: 323
Content-Type: text/xml; charset=utf-8
Client-Date: Mon, 28 Apr 2008 02:12:54 GMT
Client-Peer: 127.0.0.1:9876
Client-Response-Num: 1

<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <ans:getTimeAsStringResponse xmlns:ans="http://ts.ch01/">
      <return>Mon Apr 28 14:12:54 CST 2008</return>
    </ans:getTimeAsStringResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Auch hier befindet sich der SOAP-Envelope im Körper einer HTTP-Nachricht, hier der HTTP-Antwort an den Client. Die HTTP-Kopfzeile der Antwort beinhaltet den ganzzahligen Statuswert `200` sowie die Textmeldung `OK`, um anzuzeigen, daß die Anfrage des Clients erfolgreich verarbeitet wurde. Der SOAP-Envelope im Körper der HTTP-Antwort beinhaltet das aktuelle Datum und die aktuelle Uhrzeit als Zeichenkette zwischen den Tags `<return>` und `</return>`. Die SOAP-Bibliothek von Perl extrahiert den SOAP-Envelope aus der HTTP-Antwort und erwartet den Rückgabewert der angeforderten Operation des Dienstes aufgrund der Struktur des SOAP-Dokumentes zwischen den `<return>`-Tags.

1.5 Ein Java-Client für den TimeServer-Dienst

[26] Das folgende Beispiel zeigt einen zu den Perl- und Ruby-Clients in den Beispielen 1-5 und 1-6 gleichwertigen Java-Client:

Beispiel 1.9: Java-Client für den TimeServer-Dienst.

```
package ch01.ts;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.URL;

class TimeClient {

    public static void main(String args[]) throws Exception {

        URL url = new URL("http://localhost:9876/ts?wsdl");

        // Qualified name of the service:
        //   1st arg is the service URI
        //   2nd is the service name published in the WSDL
        QName qname = new QName("http://ts.ch01/", "TimeServerImplService");

        // Create, in effect, a factory for the service.
        Service service = Service.create(url, qname);

        // Extract the endpoint interface, the service "port".
        TimeServer eif = service.getPort(TimeServer.class);

        System.out.println(eif.getTimeAsString());
        System.out.println(eif.getTimeAsElapsed());

    }

}
```

Der Java-Client verwendet dieselbe URL mit Query-String, wie der Perl- und der Ruby-Client, erzeugt aber explizit einen *qualifizierten XML-Namen* mit der Syntax `namespace-URI:local-name`. Ein *URI (Uniform Resource Identifier)* unterscheidet sich dadurch von der häufigeren *URL (Uniform Resource Locator)*, daß eine URL einen ~~Ort~~ angibt, während ein URI keinen ~~Ort~~ anzugeben braucht. Kurz: Ein URI ist nicht zwingend eine URL. Im Augenblick genügt es, hervorzuheben, daß die Java-Klasse `javax.xml.namespace.QName` qualifizierte XML-Namen repräsentiert. In diesem Beispiel liefert das WSDL-Dokument den Namenraum-URI. Der lokale Name ist der Name der SIB-Klasse `TimeServerImpl`, erweitert um die „Endung“ `Service`. Der lokale Name ist im `<service>`-Abschnitt angegeben, dem letzten Abschnitt des WSDL-Dokumentes.

[27] Sind das `URL`- und das `QName`-Objekt erzeugt sowie die statische `Service`-Methode `create()` aufgerufen, dann wird die eigentliche interessante Anweisung

```
TimeServer eif = service.getPort(TimeServer.class);
```

aufgerufen. Erinnern Sie sich daran, daß das der `<portType>`-Abschnitt des WSDL-Dokumentes im Stil eines Interface' die Operationen des Dienstes beschreibt. Die `getPort()`-Methode gibt eine Referenz auf ein Java-Objekt zurück, das die unter `<portType>` deklarierten Operationen aufrufen kann. Das von der lokalen Variablen `eif` referenzierte Objekt ist vom Typ *`ch01.ts.TimeServer`*, also vom Typ des SEI. Der Java-Client ruft, wie zuvor der Perl- und der Ruby-Client die beiden Operationen des Webservice' auf. Die Java-Bibliotheken generieren und verarbeiten wie zuvor die Perl- und Ruby-Bibliotheken, die SOAP-Nachrichten, welche transparent ausgetauscht werden, um das erfolgreiche Aufrufen der Methoden zu ermöglichen.

1.6 Abhören von HTTP- und SOAP-Nachrichten

[28] Die Beispiele 1-7 und 1-8 zeigen eine HTTP-Anfrage- beziehungsweise HTTP-Antwortnachricht. Jede dieser Nachrichten beinhaltet einen SOAP-Envelope. Die Aufzeichnung dieser Nachrichten

wurde mit dem Perl-Client aus Abschnitt 1.3 bewerkstelligt, indem die `use`-Anweisung von Perl aus Beispiel 1.5

```
use SOAP::Lite;
```

in

```
use SOAP::Lite +trace;
```

geändert wurde. Der Ruby-Client enthält zu diesem Zweck die Zeile

```
service.wiredump_file_base = 'soapmsgs'
```

Diese Zeile bewirkt, daß der SOAP-Envelope in einer Datei auf der lokalen Festplatte gespeichert wird. [Spätere Beispiele zeigen], daß sich die gesendeten Nachrichten auch mit Java direkt auf der Übertragungsebene abfangen lassen. Es gibt verschiedene Möglichkeiten, um SOAP- und HTTP-Nachrichten auf der Übertragungsebene abzuhören. Dieser Abschnitt liefert eine kurze Einführung in einige dieser Möglichkeiten.

[29] Das Hilfsprogramm *`tcpmon`* (*https://tcpmon.dev.java.net*), ist kostenlos in Form einer ausführbaren JAR-Datei erhältlich. Es handelt sich dabei um ein Programm mit einfach bedienbarer graphischer Benutzeroberfläche. Das Programm benötigt nur drei Angaben, nämlich den Namen des Servers (Voreinstellung ist `localhost`), den Port, an dem der Server horcht (beim *`TimeServer`*-Beispiel Port 9876, an dem der Dienst mit Hilfe der statischen `Endpoint`-Methode `publish()` in Betrieb genommen wird) und den lokalen Port (Voreinstellung ist Port 8080), an dem `tcpmon` lauscht. Bei Verwendung von `tcpmon` sendet der `TimeClient` seine Anfragen also nicht an den Port 9876, sondern an Port 8080. Das Hilfsprogramm `tcpmon` hört den HTTP-Verkehr zwischen Client und Webservice ab und zeigt die Nachrichten komplett in seiner graphischen Benutzeroberfläche an.

[30] Das *Metro*-Release verfügt über Hilfsklassen, um HTTP- und SOAP-Verkehr verfolgen zu können. Dieser Ansatz erfordert keine Änderungen am Quelltext des Clients oder des Dienstes. Allerdings muß ein weiteres Package in den Klassenpfad eingetragen und eine Systemeigenschaft entweder programmatisch oder per Kommandozeile konfiguriert werden. Das benötigte Package ist in der Datei *jaxws-ri/jaxws-rt.jar* enthalten. Verweist die Umgebungsvariable *`$METRO_HOME`* auf das Verzeichnis *jaxws-ri*, dann hört das folgende Kommando den HTTP- und SOAP-Verkehr zwischen dem Client `TimeClient` (verbindet sich mit dem Dienst an Port 9876) und dem *`TimeServer`*-Dienst ab. (Bei Windows wählen Sie `%METRO_HOME%` anstelle von `$METRO_HOME`.) Das Kommando ist zur besseren Lesbarkeit auf drei Zeilen verteilt:

```
% java -cp ".":$METRO_HOME/lib/jaxws-rt.jar \
  -Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true \
  ch01.ts.TimeClient
```

Die resultierende Ausgabe zeigt zwar den gesamten SOAP-Verkehr, aber nicht alle HTTP-Header. Das Abhören der Nachrichten ist auch auf der Seite des Dienstes möglich.

[31] Es gibt zahlreiche weitere, sowohl quelloffene als auch kommerzielle Hilfsmittel, um den SOAP-Verkehr zu verfolgen. Unter den Produkten auf die es sich lohnt, einen Blick zu werfen, sind *SOAPscope* (*http://home.mindreef.com*), *NetSniffer* (*http://www.miray.de*) und *Wireshark* (*http://www.wireshark.org*). Das *`tcpdump`-Kommando* ist Bestandteil der meisten unixartigen Systeme, darunter Linux und OS X, und ist in Form von *WinDump* (*http://www.winpcap.org*) auch für Windows verfügbar. Abgesehen davon, daß `tcpdump` kostenlos zur Verfügung steht, verhält sich das Programm auch unaufdringlich in dem Sinne, daß keine Modifikationen am Dienst oder Client erforderlich sind. `tcpdump` gibt den Nachrichtenverkehr über die Standardausgabe aus. Das begleitende *`tcptrace`-Kommando* (*http://www.tcptrace.org*) dient zur Auswertung der Ausgabe. Der Rest dieses Abschnittes liefert eine kurze Beschreibung des `tcpdump`-Kommandos, eines flexiblen und mächtigen Abhörwerkzeuges.

[32] Bei unixartigen Systemen muß *tcpdump* typischerweise vom Administrator ausgeführt werden. Diverse Kommandozeilenschalter und -optionen bestimmen die Funktionsweise des Kommandos, zum Beispiel:

```
% tcpdump -i lo -A -s 1024 -l 'dst host localhost and port 9876' | tee dump.log
```

Das Programm ist in der Lage, Pakete abzufangen, die über jede beliebige Netzwerkschnittstelle versendet werden. Der Aufruf `tcpdump -D` liefert eine Liste der verfügbaren Schnittstellen (bei Windows ~~WinDump -D~~) und ist gleichbedeutend mit dem Kommando *ifconfig* `-a` bei unixartigen Systemen. Beim obigen Beispiel bedeutet die Option `-i lo`, daß die über die Schnittstelle `lo` gesendeten Pakete abgefangen werden sollen, wobei `lo` eine Abkürzung für die bei vielen unixartigen Systemen vorhandene, sogenannte *Loopback-Schnittstelle* ist. Der Schalter `-A` bedeutet, daß die abgefangenen Pakete als ASCII-Zeichen dargestellt werden sollen. Diese Einstellung ist bei ~~web packets~~ nützlich, die typischerweise zeichenorientierte Daten enthalten. Die Option `-s 1024` stellt die sogenannte ~~snap length~~ ein, das heißt die Anzahl von Bytes, die von jedem Paket abgehört werden sollen. Der Schalter `-l` erzwingt zeilenweise gepufferte und somit besser lesbare Ausgabe. Die Konstruktion `| tee dump.log` am Ende des Kommandos bewirkt, daß die auf dem Bildschirm (Standardausgabe) angezeigte Ausgabe in identischer Form in einer lokalen Datei namens *dump.log* gespeichert wird. Schließlich wirkt der Ausdruck

```
'dst host localhost and port 9876'
```

als Filter, durch den nur Pakete mit dem Ziel `localhost` und Port 9876 abgefangen werden, dem Port, auf dem die Klasse `TimeServerPublisher` aus Beispiel 1.3 den *`TimeServer`*-Dienst in Betrieb nimmt.

[33] Es ist gleichgültig, ob `tcpdump` oder `TimeServerPublisher` zuerst aufgerufen wird. Wenn beide Programme laufen, kann der Java-Client `TimeClient` oder einer der anderen Clients aufgerufen werden. Der obige Beispielaufruf des `tcpdump`-Kommandos bewirkt, daß die über das Netzwerk transportierten Pakete in der Datei *dump.log* gespeichert werden. Es ist etwas Arbeit erforderlich, um den Dateiinhalt leichter lesbar zu machen. Auf jeden Fall aber, fängt die Datei *dump.log* die in den Beispielen 1-7 und 1-8 gezeigten SOAP-Envelopes ab.

1.7 Was haben wir bis jetzt gelernt?

[34] Unser erstes Beispiel ist ein Webservice mit zwei Operationen, die in verschiedenen Darstellungen jeweils das aktuelle Datum und die aktuelle Uhrzeit zurückgeben: Einmal als menschenlesbare Zeichenkette und einmal als die Anzahl verstrichener Millisekunden seit Beginn der Unix-Epoche. Die beiden Operationen sind als voneinander unabhängige, in sich geschlossene (*self-contained*) Methoden implementiert, das heißt, aus der Perspektive des Clients kann jede Methode unabhängig von der anderen aufgerufen werden und das Aufrufen einer Methode hat keine Auswirkungen auf die folgenden Aufrufe derselben Methode. Die beiden Methoden hängen weder voneinander, noch von einem dynamischen Feld (Objektfeld, im Gegensatz zum statischen Klassenfeld) ab, auf welches beide Methoden Zugriff haben. Die SIB-Klasse `TimeServerImpl` hat keine Felder. Kurz, die beiden Methoden sind zustandslos.

[35] Keine der beiden Methoden im ersten Beispiel erwartet Argumente. Im Allgemeinen können die Operationen eines Webservice' parametrisiert sein, damit sie bei Anfragen mit entsprechenden Informationen aufgerufen werden können. Gleichgültig, ob die Operationen eines Webservice' parametrisiert sind oder nicht, sollten sie aus der Perpektive des Clients stets unabhängig und in sich geschlossen sein. Diese Richtlinie wurde beim Entwurf aller Beispiele berücksichtigt, die wir betrachten werden, insbesondere für solche, die aufwändiger sind als das erste Beispiel.

1.7.1 Wesentliche Eigenschaften des ersten Beispiels

[36] Die Klasse `TimeServerImpl` implementiert einen Webservice nach dem unverkennbaren *Nachrichtenaustauschmuster* Request/Response. Der Dienst gestattet Clients sprachneutrale Methodenfernaufrufe, durch welche die Methoden `getTimeAsString()` und `getTimeAsElapsed()` aufgerufen werden. Es gibt noch weitere Nachrichtenaustauschmuster. Stellen Sie sich beispielsweise einen Webservice vor, der den Schneefall in einem Skigebiet beobachtet. Ein Teil der Clients, die mit Hilfe elektrischer Geräte die Höhe der Schneedecke an strategischen Position einer Loipe bestimmen, könnten das One-Way-Muster (unidirektionale Nachrichtenübertragung) implementieren, das heißt, die Höhe der Schneedecke an einer bestimmten Stelle übertragen, ohne eine Antwort des Dienstes zu erwarten. Der Dienst könnte das Notification-Muster (etwa „Benachrichtigungsmuster") darstellen, um registrierte Clients wie Reisebüros per Multicast über die aktuellen Schneeverhältnisse zu informieren. Schließlich könnte der Dienst periodisch das Solicit/Response-Muster (etwa „Auskunft erbitten") anwenden, um bei den registrierten Clients anzufragen, ob sie weiterhin benachrichtigt werden möchten. Kurz, SOAP-basierte Webservices unterstützen verschiedene Nachrichtenaustauschmuster. Das Request/Response-Muster von RPC ist vorherrschend. Es lohnt sich, die zur Unterstützung dieses Nachrichtenaustauschmusters benötigte Infrastruktur zusammenzufassen:

- *Transport der Nachrichten*: SOAP ist entwurfsbedingt neutral bezüglich des unterliegenden Transportprotokolls. Diese Eigenschaft verkompliziert die Dinge, da sich SOAP-Nachrichten nicht auf protokollspezifische, in der Transportinfrastruktur enthaltene Informationen verlassen können. SOAP-Nachrichten, die über das HTTP-Protokoll abgewickelt werden, dürfen sich nicht von SOAP-Nachrichten unterscheiden, die über ein anderes Protokoll versendet werden, zum Beispiel das *SMTP* (Simple Mail Transfer Protocol), das *FTP* (File Transfer Protocol) oder den *JMS* (Java Message Service). In der Praxis ist HTTP das übliche Transportprotokoll für SOAP-basierte Webservices, wie aus der üblichen Bezeichnung „SOAP-basierter *Web*service" hervorgeht.

- *Dienstkontrakt* (*service contract*): Der Client eines Webservice' benötigt Informationen über dessen Operationen, um sie aufrufen zu können. Insbesondere braucht der Client Informationen über die Aufrufsyntax, das heißt den Namen der Operation, Reihenfolge und Typ der Argumente, die der Operation übergeben werden sowie den Typ des Rückgabewertes. Außerdem braucht der Client den Endpunkt des Webservice', typischerweise dessen URL. Das WSDL-Dokument liefert diese und weitere Informationen. Ein Client kann einen Webservice auch ohne vorherigen Zugriff auf das WSDL-Dokument aufrufen, die Dinge würden dadurch aber unnötig kompliziert.

- *Datentypsystem*: Der Schlüssel zur Sprachtransparenz und somit zur Interoperabilität zwischen Dienst und Client ist ein gemeinsames Typsystem, so daß die Datentypen beim clientseitigen Aufruf mit den Typen auf der Dienstseite koordiniert werden können. Ein einfaches Beispiel: Angenommen, ein in Java geschriebener Webservice hat folgende Operation:

  ```
  boolean bytes_ok(byte[] some_bytes)
  ```

  Die Operation `bytes_ok` validiert die als Array übergebenen Bytes anhand irgendeines Tests und gibt entweder `true` oder `false` zurück. Ferner angenommen, daß ein in C geschriebener Client die `bytes_ok`-Operation aufrufen muß. C kennt die Typen `boolean` und `byte` nicht, sondern stellt boolesche Werte mit Hilfe ganzer Zahlen dar. Die 0 entspricht `false` und jeder von 0 verschiedene Wert `true`. Der vorzeichenbehaftete C-Typ `char` entspricht dem Java-Typ `byte`. Nun wäre ein Webservice sperrig zu benutzen, wenn der Client die Datentypen seiner Sprache auf die Datentypen der Sprache des Dienstes abbilden müßte. Bei SOAP-basierten Webservices ist das System der XML-Schematypen der Standardvermittler zwischen den Datentypen auf der Client- und der Dienstseite. Beim obigen Beispiel vermittelt der XML-

Schematyp `xsd:byte` zwischen dem vorzeichenbehafteten C-Typ `char` und dem Java-Typ `byte`. In derselben Weise vermittelt der XML-Schematyp `xsd:boolean` zwischen ganzzahligen von 0 verschiedenen Werten/dem Wert 0 bei C und den booleschen Java-Werten `true`/`false`. Der Präfix *xsd* (für „XML Schema Definition) in der Notation `xsd:byte` deutet an, daß es sich um einen XML-Schematyp handelt. (Die Endung einer Datei die eine XML-Schemadefinition enthält, ist üblicherweise *.xsd*, zum Beispiel *purchaseOrder.xsd*.)

1.8 Die SOAP-Schnittstelle der SE 6

[37] Der Reiz SOAP-basierter Webservices besteht zu einem nicht unwesentlichen Teil darin, daß die SOAP-Nachrichten in der Regel unsichtbar bleiben. Es kann sich dennoch als nützlich erweisen, einen Blick auf die bei Java vorhandene Unterstützung zum Erzeugen und Verarbeiten von SOAP-Nachrichten zu werfen. Kapitel 3 führt SOAP-Behandler ein und stellt die praktische Nutzung der SOAP-API vor. Dieser Abschnitt gibt anhand eines Simulationsbeispiels einen ersten Einblick in die SOAP-API. Die Applikation besteht nur aus einer einzigen Klasse namens `ch01.soap.DemoSoap`, simuliert aber das Senden einer SOAP-Nachricht als Anfrage *und* das Empfangen einer weiteren als Antwort:

Beispiel 1.10: Demonstration der SOAP-API von Java.

```
package ch01.soap;

import java.util.Date;
import java.util.Iterator;
import java.io.InputStream;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.Node;
import javax.xml.soap.Name;

public class DemoSoap {

    private static final String LocalName = ”TimeRequest”;
    private static final String Namespace = ”http://ch01/mysoap/”;
    private static final String NamespacePrefix = ”ms”;

    private ByteArrayOutputStream out;
    private ByteArrayInputStream in;

    public static void main(String[] args) {
        new DemoSoap().request();
    }

    private void request() {

        try {

            // Build a SOAP message to send to an output stream.
            SOAPMessage msg = create_soap_message();
```

```
            // Inject the appropriate information into the message.
            // In this case, only the (optional) message header is used
            // and the body is empty.
            SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
            SOAPHeader hdr = env.getHeader();

            // Add an element to the SOAP header.
            Name lookup_name = create_qname(msg);
            hdr.addHeaderElement(lookup_name).addTextNode("time_request");

            // Simulate sending the SOAP message to a remote system by
            // writing it to a ByteArrayOutputStream.
            out = new ByteArrayOutputStream();
            msg.writeTo(out);

            trace("The sent SOAP message:", msg);

            SOAPMessage response = process_request();
            extract_contents_and_print(response);

        }

        catch(SOAPException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }

    }

    private SOAPMessage process_request() {

        process_incoming_soap();
        coordinate_streams();
        return create_soap_message(in);

    }

    private void process_incoming_soap() {

        try {

            // Copy output stream to input stream to simulate
            // coordinated streams over a network connection.
            coordinate_streams();

            // Create the "received" SOAP message from the
            // input stream.
            SOAPMessage msg = create_soap_message(in);

            // Inspect the SOAP header for the keyword 'time_request'
            // and process the request if the keyword occurs.
            Name lookup_name = create_qname(msg);

            SOAPHeader header = msg.getSOAPHeader();
            Iterator it = header.getChildElements(lookup_name);
            Node next = (Node) it.next();
            String value = (next == null) ? "Error!" : next.getValue();

            // If SOAP message contains request for the time, create a
            // new SOAP message with the current time in the body.
            if (value.toLowerCase().contains("time_request")) {

                // Extract the body and add the current time as an element.
                String now = new Date().toString();
                SOAPBody body = msg.getSOAPBody();
                body.addBodyElement(lookup_name).addTextNode(now);
                msg.saveChanges();

                // Write to the output stream.
```

```
            msg.writeTo(out);
            trace("The received/processed SOAP message:", msg);
        }
    }

    catch(SOAPException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }

}

private void extract_contents_and_print(SOAPMessage msg) {

    try {

        SOAPBody body = msg.getSOAPBody();

        Name lookup_name = create_qname(msg);
        Iterator it = body.getChildElements(lookup_name);
        Node next = (Node) it.next();

        String value = (next == null) ? "Error!" : next.getValue();
        System.out.println("\n\nReturned from server: " + value);

    }

    catch(SOAPException e) { System.err.println(e); }

}

private SOAPMessage create_soap_message() {

    SOAPMessage msg = null;

    try {

        MessageFactory mf = MessageFactory.newInstance();
        msg = mf.createMessage();

    }

    catch(SOAPException e) { System.err.println(e); }
    return msg;

}

private SOAPMessage create_soap_message(InputStream in) {

    SOAPMessage msg = null;

    try {

        MessageFactory mf = MessageFactory.newInstance();
        msg = mf.createMessage(null, // ignore MIME headers
                               in);  // stream source

    }

    catch(SOAPException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
    return msg;

}

private Name create_qname(SOAPMessage msg) {

    Name name = null;

    try {

        SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
        name = env.createName(LocalName, NamespacePrefix, Namespace);
```

```
            }
            catch(SOAPException e) { System.err.println(e); }
            return name;
        }
        private void trace(String s, SOAPMessage m) {
            System.out.println("\n");
            System.out.println(s);
            try {
                m.writeTo(System.out);
            }
            catch(SOAPException e) { System.err.println(e); }
            catch(IOException e) { System.err.println(e); }
        }
        private void coordinate_streams() {
            in = new ByteArrayInputStream(out.toByteArray());
            out.reset();
        }
    }
```

[38] Es folgt eine Zusammenfassung der Funktionsweise des Programms mit Hervorhebung der Stellen, an denen SOAP-Benachrichtungen erzeugt oder verarbeitet werden. Die `request()`-Methode der Klasse `DemoSoap` erzeugt eine SOAP-Nachricht und fügt die Zeichenkette „`time_request`“ in den Headerabschnitt des SOAP-Envelopes ein. Der von den Kommentaren bereinigte Abschnitt des Quelltextes lautet:

```
SOAPMessage msg = create_soap_message();
SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
SOAPHeader hdr = env.getHeader();
Name lookup_name = create_qname(msg);
hdr.addHeaderElement(lookup_name).addTextNode("time_request");
```

Es gibt zwei grundlegende Möglichkeiten, um eine SOAP-Nachricht zu erzeugen. Die beiden folgenden Zeilen zeigen die einfachere Variante:

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
```

In der komplizierteren Variante bleibt die erste Zeile bestehen (das Erzeugen des `MessageFactory`-Objektes). Der Aufruf der `createMessage()`-Methode wird aber zu:

```
SOAPMessage msg = mf.createMessage(mime_headers, input_stream)
```

Das erste Argument der `createMessage()`-Methode ist ein „Container“ vom Typ `MimeHeaders`[1] mit den transport-layer headers (zum Beispiel den Schlüssel/Wert-Paaren eines HTTP-Headers). Das zweite Argument ist vom Typ `InputStream` und repräsentiert einen Eingabestrom, der die Bytes liefert, aus den die Nachricht zusammengesetzt wird (zum Beispiel der in einem `Socket`-Objekt gekapselte Eingabestrom).

[1] Anmerkung des Übersetzers: Der Typ `javax.xml.soap.MimeHeaders` arbeitet intern mit einem Objekt vom Typ `Vector`, ist also kein Container- oder Kollektionstyp im Sinne der „neue Containerbibliothek“, das heißt der Typen *`List`*, *`Set`*, *`Queue`* (Untertypen von *`Collection`*) und *`Map`*.

[39] Nach dem Erzeugen der SOAP-Nachricht wird über den SOAP-Envelope eine Referenz auf den SOAP-Header angefordert und ein XML-Textknoten mit dem Wert „`time_request`“ eingesetzt. Die entstehende SOAP-Nachricht lautet:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
      time_request
    </ms:TimeRequest>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body />
</SOAP-ENV:Envelope>
```

Im Augenblick besteht noch keine Notwendigkeit, sämtliche Details dieser SOAP-Nachricht zu untersuchen. Hier eine Zusammenfassung der wichtigsten Punkte: Eine SOAP-Nachricht hat stets einen SOAP-Body, der, wie in diesem Beispiel, aber leer sein darf. Eine SOAP-Nachricht kann einen SOAP-Header haben. In diesem Fall enthält der SOAP-Header den Text „`time_request`“. Nachrichteninhalte wie „`time_request`“ werden in der Regel im SOAP-Body platziert, während der SOAP-Header spezielle Informationen zur Verarbeitung beinhaltet, etwa Authentifizierungsdaten über den Benutzer. Das `DemoSoap`-Beispiel hat die Aufgabe, die Manipulation von SOAP-Header und SOAP-Body zu demonstrieren.

[40] Die `request()`-Methode übergibt die SOAP-Nachricht an ein `ByteArrayOutputStream`-Objekt, wodurch das Versenden der Nachricht über eine Netzwerkverbindung hinweg zu einem anderen Host simuliert wird. Die `request()`-Methode ruft ihrerseits die `process_request()`-Methode auf, die wiederum die verbleibenden Aufgaben an andere Methoden delegiert. Die Verarbeitung funktioniert wie folgt: Die empfangene SOAP-Nachricht wird aus einem `ByteArrayInputStream`-Objekt entgegengenommen, das den Eingabestrom auf der Empfängerseite simuliert. Dieser Eingabestrom enthält die gesendete SOAP-Nachricht. Die SOAP-Nachricht wird aus dem Eingabestrom rekonstruiert

```
SOAPMessage msg = null;

try {

    MessageFactory mf = MessageFactory.newInstance();
    msg = mf.createMessage(null, // ignore MIME headers
                           in);  // stream source (ByteArrayInputStream)

}
```

und anschließend weiterverarbeitet, um die Zeichenkette „`time_request`“ zu extrahieren. Die Extraktion geht folgendermaßen vor sich: Zunächst wird über die SOAP-Nachricht eine Referenz auf den SOAP-Header angefordert und ein Iterator über den Elementen mit dem Tagnamen

```
<ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
```

erzeugt. Im `DemoSoap`-Beispiel gibt es ein Element mit diesem Tagnamen und dieses Element sollte die Zeichenkette „`time_request`“ enthalten. Die Anweisungen zum Abfragen des Taginhaltes lauten:

```
SOAPHeader header = msg.getSOAPHeader();
Iterator it = header.getChildElements(lookup_name);
Node next = (Node) it.next();
String value = (next == null) ? "Error!" : next.getValue();
```

Enthält der SOAP-Header diese Zeichenkette, so wird über die SOAP-Nachricht eine Referenz auf den SOAP-Body angefordert und um ein Element ergänzt, welches das aktuelle Datum und die aktuelle Uhrzeit beinhaltet. Die geänderte SOAP-Nachricht wird anschließend als Antwort zurückgesendet. Der von den Kommentaren bereinigte Abschnitt des Quelltextes lautet:

```
if (value.toLowerCase().contains("time_request")) {

    String now = new Date().toString();
    SOAPBody body = msg.getSOAPBody();
    body.addBodyElement(lookup_name).addTextNode(now);
    msg.saveChanges();

    msg.writeTo(out);
    trace("The received/processed SOAP message:", msg);

}
```

Die ausgehende SOAP-Nachricht lautet bei mir:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
      time_request
    </ms:TimeRequest>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
      Mon Oct 27 14:45:53 CDT 2008
    </ms:TimeRequest>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Das `DemoSoap`-Beispiel vermittelt einen ersten Eindruck der SOAP-API von Java. Spätere Beispiele illustrieren die Verwendung der SOAP-API von Java auf Produktionsniveau.

1.9 Ein Beispiel mit aufwändigeren Datentypen: Der Teams-Dienst

[41] Die Operationen des *`TimeServer`*-Dienstes erwarten keine Argumente und geben Werte einfacher Typen zurück, das heißt eine Zeichenkette oder eine ganze Zahl. In diesem Abschnitt wird ein aufwändigeres Beispiel vorgestellt, dessen Details im nächsten Kapitel erläutert werden. Der `Teams`-Dienst im folgenden Beispiel unterscheidet sich in einigen wesentlichen Eigenschaften vom *`TimeServer`*-Dienst:

Beispiel 1.11: Der Teams-Dienst (Dokumentstil).

```
package ch01.team;

import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Teams {

    private TeamsUtility utils;

    public Teams() {

        utils = new TeamsUtility();
        utils.make_test_teams();

    }

    @WebMethod
    public Team getTeam(String name) { return utils.getTeam(name); }
```

```
    @WebMethod
    public List<Team> getTeams() { return utils.getTeams(); }
}
```

Der `Teams`-Webservice ist beispielsweise in einer einzigen Klasse implementiert, statt in einem separaten SEI und einer davon getrennten SIB, um auch diese Variante einmal zu zeigen. Ein wichtigerer Unterschied betrifft die Rückgabetypen der beiden Operationen des `Teams`-Dienstes. Die Operation `getTeam()` ist parametrisiert und gibt eine Referenz auf ein Objekt des selbstdefinierten Typs `Team` zurück, einer Liste von Referenzen des ebenfalls selbstdefinierten Typs `Player`. Die Operation `getTeams()` gibt eine Referenz vom Typ *`List<Team>`* zurück.

[42] Die Hilfsklasse `TeamsUtility` erzeugt die Daten. Im produktiven Betrieb könnte diese Hilfsklasse ein Team oder eine Liste von Teams aus einer Datenbank abfragen. `TeamsUtility` erzeugt die Teams und ihre Mitspieler beim Aufruf der `make_test_teams()`-Methode, damit das Beispiel einfach bleibt. Der Quelltext der Hilfsklasse `TeamsUtility` lautet (gekürzt):

```
package ch01.team;

import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

public class TeamsUtility {

    private Map<String, Team> team_map;

    public TeamsUtility() {
        team_map = new HashMap<String, Team>();
    }

    public Team getTeam(String name) { return team_map.get(name); }

    public List<Team> getTeams() {

        List<Team> list = new ArrayList<Team>();
        ...
        Set<String> keys = team_map.keySet();
        for (String key : keys)
            list.add(team_map.get(key));
        return list;

    }

    public void make_test_teams() {

        List<Team> teams = new ArrayList<Team>();
        ...
        Player chico = new Player("Leonard Marx", "Chico");
        Player groucho = new Player("Julius Marx", "Groucho");
        Player harpo = new Player("Adolph Marx", "Harpo");
        List<Player> mb = new ArrayList<Player>();
        mb.add(chico); mb.add(groucho); mb.add(harpo);
        Team marx_brothers = new Team("Marx Brothers", mb);
        teams.add(marx_brothers);

        store_teams(teams);

    }

    private void store_teams(List<Team> teams) {

        for (Team team : teams)
```

```
            team_map.put(team.getName(), team);
        }
    }
```

1.9.1 Inbetriebnahme des Dienstes und Schreiben eines Clients

[43] Rufen Sie sich ins Gedächtnis zurück, daß das SEI des *TimeServer*-Dienstes die folgende Annotation beinhaltet (Seite 5, Beispiel 1.1):

```
@SOAPBinding(style = Style.RPC)
```

Diese Annotation setzt voraus, daß der Dienst nur sehr einfache Datentypen wie Zeichenketten und ganze Zahlen verwendet. Der `Teams`-Dienst verwendet im Gegensatz dazu aufwändigere Datentypen, weshalb die Konstante `Style.RPC` („RPC-Stil“) durch den voreingestellten Wert `Style.DOCUMENT` („Dokumentstil“) ersetzt werden [should/sollte]. Der Dokumentstil erfordert mehr Konfiguration. Die entsprechenden Anweisungen sind unten angegeben, werden aber erst [Genauer im nächsten Kapitel] erläutert. Die folgenden Schritte sind notwendig, um den Webservice zu deployen und schnell einen Client zu schreiben:

- Die Quelltextdateien werden wie gewohnt übersetzt. Das Arbeitsverzeichnis ist das Verzeichnis, welches das Unterverzeichnis *ch01* enthält:

  ```
  % javac ch01/team/*.java
  ```

  Neben der mit `@WebService` annotierten Klasse `Teams` enthält das Verzeichnis *ch01/team* die vier Klassen `Team`, `Player`, `TeamsUtility` und `TeamsPublisher`:

  ```
  package ch01.team;

  public class Player {

      private String name;
      private String nickname;

      public Player() { }

      public Player(String name, String nickname) {
          setName(name);
          setNickname(nickname);
      }

      public void setName(String name) { this.name = name; }
      public String getName() { return name; }
      public void setNickname(String nickname) { this.nickname = nickname; }
      public String getNickname() { return nickname; }

  }
  // end of Player.java

  package ch01.team;

  import java.util.List;

  public class Team {

      private List<Player> players;
      private String name;

      public Team() { }

      public Team(String name, List<Player> players) {
          setName(name);
  ```

```
            setPlayers(players);
        }

        public void setName(String name) { this.name = name; }
        public String getName() { return name; }
        public void setPlayers(List<Player> players) { this.players = players; }
        public List<Player> getPlayers() { return players; }
        public void setRosterCount(int n) { } // no-op but needed for property
        public int getRosterCount() {
            return (players == null) ? 0 : players.size();
        }

    }
    // end of Team.java

    package ch01.team;

    import javax.xml.ws.Endpoint;

    class TeamsPublisher {

        public static void main(String[] args) {

            int port = 8888;
            String url = "http://localhost:" + port + "/teams";
            System.out.println("Publishing Teams on port " + port);
            Endpoint.publish(url, new Teams());

        }

    }
```

- Rufen Sie im Arbeitsverzeichnis das Hilfsprogramm *wsgen* auf, das zur SE 6 gehört:

  ```
  % wsgen -cp . ch01.team.Teams
  ```

  `wsgen` erzeugt verschiedene Artefakte, das heißt Java-Typen, welche die statische `Endpoint`-Methode `publish()` benötigt, um das WSDL-Dokument des Dienstes zu generieren. [Genauer: Kapitel 2] widmet sich diesen Artefakten und der Weise, in der sie zum WSDL-Dokument beitragen.

- Rufen Sie die `TeamsPublisher`-Applikation auf.

- Rufen Sie im Arbeitsverzeichnis das *wsimport-Kommando* auf, das ebenfalls zur SE 6 gehört:

  ```
  % wsimport -p teamsC -keep http://localhost:8888/teams?wsdl
  ```

  `wsimport` generiert verschiedene Klassen im Unterverzeichnis *teamsC* (der Schalter `-p` steht für Package). Diese Klassen erleichtern die Arbeit, einen Client für den Dienst zu schreiben.

Der vierte Schritt vereinfacht das Schreiben eines Clients:

```
import teamsC.TeamsService;
import teamsC.Teams;
import teamsC.Team;
import teamsC.Player;
import java.util.List;

class TeamClient {

    public static void main(String[] args) {

        TeamsService service = new TeamsService();
        Teams port = service.getTeamsPort();
        List<Team> teams = port.getTeams();
```

```
            for (Team team : teams) {
                System.out.println("Team name: " + team.getName() +
                                   " (roster count: " + team.getRosterCount() + ")");
                for (Player player : team.getPlayers())
                    System.out.println("  Player: " + player.getNickname());         }
        }
    }
```

Die Ausgabe beim Aufrufen des Clients lautet:

```
Team name: Abbott and Costello (roster count: 2)
  Player: Bud
  Player: Lou
Team name: Marx Brothers (roster count: 3)
  Player: Chico
  Player: Groucho
  Player: Harpo
Team name: Burns and Allen (roster count: 2)
  Player: George
  Player: Gracie
```

Dieses Beispiel deutet an, was mit einem kommerziellen SOAP-basierten Webservice möglich ist. Selbstdefinierte Typen wie `Player` und `Team` sowie beliebige Kollektionen davon, können einem Webservice als Argumente über- oder von dort zurückgegeben werden, solange gewisse Richtlinien eingehalten werden. Eine Richtlinie kommt in diesem Beispiel zum Tragen: Die JavaBean-Eigenschaften der Klassen `Player` und `Team` sind vom Typ `String` oder `int` und ein *`List`*-Container verfügt, wie jeder andere Typ unter dem Interface *`Collection`* über eine `toArray()`-Methode. Letztendlich reduziert sich ein Container vom Typ *`List<Team>`* auf ein Array von Elementen einfachen Typs, hier von `String`-Referenzen beziehungsweise `int`-Werten. Das nächste Kapitel behandelt die Einzelheiten, insbesondere in welcher Weise die Kommandos `wsgen` und `wsimport` das Schreiben von Diensten und Clients in Java vereinfachen.

1.10 Ein multithreadfähiger Endpoint-Publisher

[44] Bei den bisherigen Beispielen war der `Endpoint`-Publisher nur singlethreadfähig und konnte somit stets höchstens eine Anfrage verarbeiten. Der in Betrieb genommene Dienst mußte die Verarbeitung einer Anfrage beenden, bevor die Verarbeitung einer neuen Anfrage beginnen konnte. Blieb die Verarbeitung einer Anfrage „hängen“, konnte keine der folgenden Anfragen verarbeitet werden, bis die hängengebliebene Anfrage vollständig verarbeitet war.

[45] Im produktiven Betrieb muß der `Endpoint`-Publisher aber in der Lage sein, gleichzeitig eingehende Anfragen zu behandeln, so daß mehrere wartende Anfragen parallel verarbeitet werden können. Ist die unterliegende Plattform ein *symmetrisches Mehrprozessorsystem* (*SMP*), so können verschiedene Anfragen tatsächlich von separaten Prozessoren gleichzeitig verarbeitet werden. Bei einem Einprozessorsystem wird gleichzeitige Verarbeitung durch die zeitliche Aufteilung der Prozessorkapazität bewirkt. Dabei erhält jede Anfrage einen Anteil an den verfügbaren CPU-Zyklen, so daß sich stets mehrere Anfragen in unterschiedlichen Zuständen ihrer Verarbeitung befinden. Java bewerkstelligt die gleichzeitige Verarbeitung via Multithreading. Die JAX-WS unterstützt multithreadfähige `Endpoint`-Publisher, ohne den Programmierer zur Anwendung fehleranfälliger Konstruktionen wie synchronisierte Blöcke oder `wait()`/`notify()`-Aufrufrufe zu zwingen.

[46] Ein Objekt vom Typ `Endpoint` hat ein Feld vom Typ *`Executor`* (einen „Exekutor“), das über *Abfrage- und Änderungsmethoden* erreichbar ist. Ein Exekutor ist ein Objekt, das Aufgaben vom

Typ *Runnable* verabeitet, zum Beispiel Objekt des Java-Typs `Thread`. (Das Interface *Runnable* deklariert nur eine einzige Methode: `public void run()`.) Ein Exekutor ist eine Alternative zum `Thread`-Objekt, da der Exekutor abstrakte Eingriffsmöglichkeiten zum Starten und Steuern von Aufgaben bietet, die gleichzeitig verarbeitet werden sollen. Der erste Schritt, um einen `Endpoint`-Publisher multithreadfähigig zu machen, ist das Schreiben eines Exekutors:

```
package ch01.ts;

import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class MyThreadPool extends ThreadPoolExecutor {

    private static final int pool_size = 10;
    private boolean is_paused;
    private ReentrantLock pause_lock = new ReentrantLock();
    private Condition unpaused = pause_lock.newCondition();

    public MyThreadPool(){

        super(pool_size,        // core pool size
              pool_size,        // maximum pool size
              0L,               // keep-alive time for idle thread
              TimeUnit.SECONDS, // time unit for keep-alive setting
              new LinkedBlockingQueue<Runnable>(pool_size)); // work queue

    }

    // some overrides
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        pause_lock.lock();
        try {
            while (is_paused) unpaused.await();
        }
        catch (InterruptedException e) { t.interrupt(); }
        finally { pause_lock.unlock(); }
    }

    public void pause() {
        pause_lock.lock();
        try {
            is_paused = true;
        }
        finally { pause_lock.unlock(); }
    }

    public void resume() {
        pause_lock.lock();
        try {
            is_paused = false;
            unpaused.signalAll();
        }
        finally { pause_lock.unlock(); }
    }

}
```

Die Klasse `MyThreadPool` erzeugt einen Vorrat von zehn Threads, die hinter den Kulissen in ei-

ner Warteschlange fester Länge vorgehalten werden. Sind alle bevorrateten Threads in Betrieb, so muß die nächste zu verarbeitende Aufgabe warten, bis einer der beschäftigten Threads verfügbar wird. Alle solchen Steuerungsdetails werden automatisch abgehandelt. Die Klasse `MyThreadPool` überschreibt einige der vorhandenen Methoden ihrer Basisklasse `ThreadPoolExecutor` um dieses Verhalten zu bewirken.

[47] Ein Objekt der Klasse `ThreadPoolExecutor` kann nun verwendet werden, um einen multithreadfähigen `Endpoint`-Publisher zu schreiben. Das folgende Beispiel zeigt den überarbeiteten Publisher, der nun aus mehreren Methoden besteht, um die Arbeit zu verteilen:

```
package ch01.ts;

import javax.xml.ws.Endpoint;

class TimePublisherMT { // MT for multithreaded

    private Endpoint endpoint;

    public static void main(String[] args) {

        TimePublisherMT self = new TimePublisherMT();
        self.create_endpoint();
        self.configure_endpoint();
        self.publish();

    }

    private void create_endpoint() {
        endpoint = Endpoint.create(new TimeServerImpl());
    }

    private void configure_endpoint() {
        endpoint.setExecutor(new MyThreadPool());
    }
    private void publish() {

        int port = 8888;
        String url = "http://localhost:" + port + "/ts";
        endpoint.publish(url);
        System.out.println("Publishing TimeServer on port " + port);

    }
}
```

Ist die von `ThreadPoolExecutor` abgeleitete Klasse einmal implementiert, bleibt nur noch, das *`Executor`*-Feld des `Endpoint`-Publishers mit einer Referenz auf ein Objekt dieser Klasse zu bewerten. Die Details der Threadsteuerung dringen nicht in die Klasse `TimePublisherMT` ein.

[48] Der multithreadfähige `Endpoint`-Publisher eignet sich für leichtgewichtigen produktiven Betrieb, ist aber kein *Container für Dienste* im eigentlichen Sinne, also eine Applikation, die mehrere Webservices am selben Port deployen kann. Ein Webcontainer wie die Referenzimplementierung Tomcat eignet sich besser, um viele Webservices in Betrieb zu nehmen. Tomcat wird im Rahmen der späteren Beispiele in diesem Buch noch vorgestellt.

1.11 Ausblick

[49] Ein SOAP-basierter Webservice sollte für potentielle Clients einen Dienstkontrakt in Form eines WSDL-Dokumentes zur Verfügung stellen. Wir haben gesehen, wie ein Perl-, ein Ruby- und ein Java-Client dieses WSDL-Dokument anfordern und mit Hilfe der jeweiligen unterliegenden SOAP-

Bibliotheken auszuwerten. Das nächste Kapitel untersucht das WSDL-Dokument im Detail und zeigt auf, wie es genutzt werden kann, um clientseitige Artefakte zu generieren, das heißt Java-Klassen, welche die Entwicklung von Clients für Webservices vereinfachen. Die Java-Clients in Kapitel 2 werden, im Gegensatz zu unserem ersten Java-Client, nicht von Anfang an geschrieben, sondern unter Zuhilfenahme des *`wsimport`-Kommandos* programmiert, wie etwa die Klasse `TeamClient` Abschnitt 1.9. Kapitel 2 stellt darüberhinaus *JAX-B (Java Architecture for XML Binding)* vor, eine Sammlung von Java-Packages zur Abstimmung von Java- und XML-Datentypen. Das *`wsgen`-Kommando* generiert JAX-B-Artefakte, die bei dieser Koordination eine wesentliche Rolle spielen und wird daher ebenfalls behandelt.

Kapitel 2

WSDL-Dokumente

Inhaltsübersicht

2.1 Die Funktion des WSDL-Dokumentes

[0] Der Nutzen des WSDL-Dokumentes, des sogenannten *Dienstkontraktes* (*service contract*) eines SOAP-basierten Webservice', läßt sich am besten anhand von Beispielen zeigen. Der ursprüngliche

Java-Client (`ch01.ts.TimeClient`) des *`ch01.ts.TimeServer`*-Dienstes aus Abschnitt 1.5, ruft die statische `Service`-Methode *`create()`* mit zwei Argumenten auf: Einer URL, die den Endpunkt angibt, an dem der Dienst verfügbar ist sowie einem qualifizierten XML-Namen (einer Referenz auf ein Objekt der Klasse *`QName`*), der wiederum aus dem lokalen Namen des Dienstes, hier `TimeServerImplService`, und einem Namensraumbezeichner (*namespace identifier*) besteht, hier dem URI *http://ts.ch01/*. Die entsprechenden Zeilen aus dem Quelltext auf Seite 13 lauten (ohne Kommentare):

```
URL url = new URL("http://localhost:9876/ts?wsdl");
QName qname = new QName("http://ts.ch01/", "TimeServerImplService");
Service service = Service.create(url, qname);
```

Beachten Sie die *Umkehrung des Packagenames* der *Service Implementation Bean* (SIB, `ch01.ts.TimeServerImpl`): Der Packagename `ch01.ts` wird im URI zu `ts.ch01`! Dieses Detail ist wesentlich. Wird das erste Argument des `QName`-Konstruktors in `http://ch01.ts/` geändert, so wirft der Client `TimeClient` eine Ausnahme aus, aus der hervorgeht, daß das automatisch generierte WSDL-Dokument des Webservice' keinen Dienst mit diesem Namensraum-URI beschreibt. Das Ermitteln des Namensraum-URIs fällt dem Programmierer zu, in der Regel durch Nachlesen im WSDL-Dokument! Der Schrägstrich (`/`) am Ende des URI ist übrigens ebenfalls erforderlich. Das Argument `http://ts.ch01`, ohne abschließenden Schrägstrich, ruft dieselbe Ausnahme hervor wie `http://ch01.ts/` (*ch01* und *ts* in der „falschen" Reihenfolge).

[1] Für den Perl-Client gelten dieselben Bedingungen. Hier eine überarbeitete Version, die auf den Webservice zugreift, ohne dessen WSDL-Dokument anzufordern:

Beispiel 2.1: Überarbeiteter Perl-Client für den TimeServer-Dienst.

```
#!/usr/bin/perl -w

use SOAP::Lite;

my $endpoint = 'http://127.0.0.1:9876/ts'; # endpoint
my $uri      = 'http://ts.ch01/';          # namespace

my $client = SOAP::Lite->uri($uri)->proxy($endpoint);

my $response = $client->getTimeAsString()->result();
print $response, "\n";
$response = $client->getTimeAsElapsed()->result();
print $response, "\n";
```

Die Funktionalität dieses Perl-Clients ist äquivalent zur ursprünglichen Version in Abschnitt 1.3. Der geänderte Perl-Client muß allerdings den Namensraum-URI des Dienstes angeben: `http://ts.ch01/`. Nur dieser URI funktioniert, weil das per Java generierte WSDL-Dokument genau diesen URI nennt. Eigentlich sind zwei Informationen erforderlich, um den Webservice zu erreichen, nämlich der Endpunkt (URL) und der Namensraum (URI) des Dienstes. Der obige Perl-Client ist schwieriger zu programmieren, als die ursprüngliche Version. Die geänderte Variante setzt voraus, daß der Programmierer den Namensraum-URI des Dienstes sowie die URL des Endpunktes kennt. Der ursprüngliche Perl-Client umgeht dieses Problem durch Anfordern des WSDL-Dokumentes, welches den *URI* des Dienstes beinhaltet. Der ursprüngliche Perl-Client, der den URI aus dem erhaltenen *WSDL-Dokument* entnimmt, ist der einfachere Weg.

2.1.1 Generieren clientseitiger Artefakte aus einem WSDL-Dokument

[2] Das Java-Kommando `wsimport` vereinfacht das Schreiben eines Java-Clients für einen SOAP-basierten Webservice. Das `wsimport`-Kommando generiert, basierend auf dem WSDL-Dokument,

unterstützenden Quelltext für Clients (sogenannte *Artefakte*). Ohne Argumente aufgerufen, liefert das Kommando

```
% wsimport
```

eine kurze Erläuterung zu seiner Verwendungsweise. Das erste Anwendungsbeispiel für `wsimport` erzeugt clientseitige Artefakte für den *`TimeServer`*-Dienst. Nach dem Start des `Endpoint`-Publishers `TimeServerPublisher` generiert das Kommando

```
% wsimport -keep -p client http://localhost:9876/ts?wsdl
```

zwei *.java* und zwei *.class* Dateien im Verzeichnis *client*. Die URL gibt an, wo sich das WSDL-Dokument befindet und ist beim ursprünglichen Perl-, dem Ruby- und dem Java-Client identisch. Die Option `-p` gibt das Package an, dem die generierten Dateien zugeordnet werden sollen, hier `client`. Der Packagename ist frei wählbar und `wsimport` verwendet oder erzeugt ein Unterverzeichnis dieses Namens. Der Schalter *`-keep`* bewirkt, daß die *.java* Dateien nach dem Übersetzen erhalten bleiben, in diesem Fall zur Untersuchung. Die Option*`-p`* ist deshalb wichtig, weil `wsimport` eine Datei namens *TimeServer.class* erzeugt, also denselben Namen verwendet, wie für das ursprüngliche Service Endpoint Interface (SEI). Wird `wsimport` ohne Packageangabe aufgerufen, so wählt `wsimport` per Voreinstellung das Package, welches die Implementierung des Dienstes enthält, hier also `ch01.ts`. Die Option `-p` verhindert also, daß das bereits übersetzte *SEI* mit der von `wsimport` generierten Datei überschrieben wird. Soll das `wsimport`-Kommando mit einem lokalen WSDL-Dokumente arbeiten, zum Beispiel der Datei *ts.wsdl*, so lautet der Aufruf:

```
% wsimport -keep -p client ts.wsdl
```

Die beiden folgenden Beispiele zeigen zwei per `wsimport`-Kommando generierte *.java* Dateien, aus denen die Kommentare entfernt wurden:

Beispiel 2.2: Datei TimeServer.java (generiert per wsimport-Kommando).

```
package client;

import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name = "TimeServer", targetNamespace = "http://ts.ch01/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface TimeServer {

    @WebMethod
    @WebResult(partName = "return")
    public String getTimeAsString();

    @WebMethod
    @WebResult(partName = "return")
    public long getTimeAsElapsed();

}
```

Beispiel 2.3: Datei TimeServerImplService.java (generiert per wsimport-Kommando).

```
package client;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
```

```
@WebServiceClient(name = "TimeServerImplService",
                  targetNamespace = "http://ts.ch01/",
                  wsdlLocation = "http://localhost:9876/ts?wsdl")
public class TimeServerImplService extends Service {

    private final static URL TIMESERVERIMPLSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:9876/ts?wsdl");
        }
        catch (MalformedURLException e) {
            e.printStackTrace();
        }
        TIMESERVERIMPLSERVICE_WSDL_LOCATION = url;
    }

    public TimeServerImplService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public TimeServerImplService() {
        super(TIMESERVERIMPLSERVICE_WSDL_LOCATION,
            new QName("http://ts.ch01/", "TimeServerImplService"));
    }

    @WebEndpoint(name = "TimeServerImplPort")
    public TimeServer getTimeServerImplPort() {
    return (TimeServer) super.getPort(
        new QName("http://ts.ch01/", "TimeServerImplPort"), TimeServer.class);
    }

}
```

[3] Drei Aspekte dieser beiden generierten Dateien verdienen Aufmerksamkeit. Erstens: Das generierte Interface *client.TimeServer* deklariert dieselben Methoden wie das ursprüngliche SEI *ch01.ts.TimeServer* in Unterabschnitt 1.2.1. Diese Methoden sind die Operationen `getTimeAsString()` und `getTimeAsElapsed()` des *TimeServer*-Dienstes. Zweitens: Die generierte Klasse `client.TimeServerImplService` definiert einen argumentlosen Konstruktor, der das gleiche `Service`-Objekt erzeugt, wie der ursprüngliche Java-Client `ch01.ts.TimeClient` in Abschnitt 1.5. Drittens: Die generierte Klasse `TimeServerImplService` definiert die Methode `getTimeServerImplPort()`, die eine Referenz vom Typ des generierten Interfaces *TimeServer* zurückgibt. Dieses Objekt wiederum, unterstützt Aufrufe der beiden Operationen des *TimeServer*-Dienstes. Gemeinsam erleichtern die beiden generierten Typen *TimeServer* und `TimeServerImplService` das Schreiben eines Java-Clients für den *TimeServer*-Dienst. Das nächste Beispiel zeigt einen Java-Client, der die zuvor per `wsimport` generierten Artefakte (kurz „`wsimport`-Artefakte“) verwendet:

Beispiel 2.4: TimeClientWSDL nutzt die wsimport-Artefakte.

```
package client;

class TimeClientWSDL {
    public static void main(String[] args) {

        // The TimeServerImplService class is the Java type bound to
        // the service section of the WSDL document.
        TimeServerImplService service = new TimeServerImplService();

        // The TimeServer interface is the Java type bound to
        // the portType section of the WSDL document.
```

```
        TimeServer eif = service.getTimeServerImplPort();

        // Invoke the methods.
        System.out.println(eif.getTimeAsString());
        System.out.println(eif.getTimeAsElapsed());

    }

}
```

[4] Der Client in Beispiel 2.4 ist mit dem ursprünglichen Client `ch01.ts.TimeClient` in Abschnitt 1.5 funktional äquivalent, aber viel einfacher zu schreiben. Insbesondere sind fehleranfällige, aber entscheidende Details wie die richtigen Argumente des *`QName`*)-Konstruktors und der Endpunkt des Dienstes in der generierten Klasse `client.TimeServerImplService` versteckt. Die in diesem Unterabschnitt beschriebene Vorgehensweise funktioniert generell beim Schreiben von Clients mit Hilfe WSDL-basierter Artefakte wie *`client.TimeServer`* oder `client.TimeServerImplService`. Zusammenfassung:

- Zuerst wird mit Hilfe eines der beiden Konstruktoren der per `wsimport` generierten Klasse, in diesem Beispiel `client.TimeServerImplService`, ein `Service`-Objekt erzeugt. Ein argumentloser Konstruktor eignet sich wegen seiner Unkompliziertheit am besten. Der zweiargumentige Konstruktor kommt zum Einsatz, wenn sich der Namensraum (URI) oder der Endpunkt (URL) des Dienstes geändert hat. In diesem Fall ist es allerdings ratsam, die aus dem WSDL-Dokument erzeugten *.java* Dateien durch einen erneuten Aufruf von `wsimport` neu zu generieren.

- Rufen Sie die `get...Port()`-Methode des erzeugten `Service`-Objektes auf, in diesem Beispiel die Methode `getTimeServerImplPort()`. Die Methode gibt eine Referenz auf ein Objekt zurück, welches die Operationen des Webservice' *kapselt*, hier `getTimeAsString()` und `getTimeAsElapsed()`, deklariert im ursprünglichen SEI.

2.1.2 Die Annotation @WebResult

[5] Beide Methoden, des per `wsimport` aus dem WSDL-Dokument generierten Interface' *`client.TimeServer`* tragen die Annotation `@WebResult`. Das nächste Beispiel zeigt anhand einer überarbeiteten Version des SEI *`TimeServer`* (ohne Kommentare), wie diese Annotation wirkt:

Beispiel 2.5: Eine aufwändiger annotierte Version des TimeServer-Dienstes.

```
package ch01.ts;  // time server

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
@SOAPBinding(style = Style.RPC) // more on this later
public interface TimeServer {

    @WebMethod
    @WebResult(partName = "time_response")
    String getTimeAsString();

    @WebMethod
    @WebResult(partName = "time_response")
    long getTimeAsElapsed();
```

```
}
```

Die Annotation `@WebResult` tritt bei beiden Operationen des Dienstes auf. Der `<message>`-Abschnitt des resultierenden WSDL-Dokumentes reflektiert diese Änderung:

```
<message name="getTimeAsString"></message>
<message name="getTimeAsStringResponse">
  <part name="time_response" type="xsd:string"></part>
</message>
<message name="getTimeAsElapsed"></message>
<message name="getTimeAsElapsedResponse">
  <part name="time_response" type="xsd:long"></part>
</message>
```

Beachten Sie, daß die `name`-Attribute der `<part>`-Elemente, im Gegensatz zum ursprünglichen WSDL-Dokument in Unterabschnitt 1.2.3, nicht mit `return`, sondern mit `time_response` bewertet ist. Auch der SOAP-Envelope in der Antwort des Webservice' gibt diese Veränderung wieder:

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <ans:getTimeAsStringResponse xmlns:ans="http://ts.ch01/">
      <time_response>
        Thu Mar 27 21:20:09 CDT 2008
      </time_response>
    </ans:getTimeAsStringResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Das `<return>`-Tag in der ursprünglichen SOAP-Antwort wird hier durch das `<time_response>`-Tag ersetzt. Wäre nur die Operation `getTimeAsString()` mit `@WebResult` annotiert, so würde die SOAP-Antwort für diese Operation das `<time_response>`-Tag verwenden, für die Operation `getTimeAsElapsed()` dagegen das `<return>`-Tag.

[6] Die Quintessenz besteht darin, daß es verschiedene Annotationen gibt, mit denen die Gestalt des generierten WSDL-Dokumentes beeinflußt werden kann. Diese Annotationen werden in kleinen Dosierungen eingeführt. Die Verbreitung von Annotation läßt sich am einfachsten dadurch eindämmen, daß nur solche Annotationen verwendet werden, die eine wichtige Aufgabe haben. Gleichgültig ob der Attributwert `time_response`/das `<time_response>`-Tag oder der Attributwert `return`/das `<return>`-Tag im WSDL-Dokument beziehungsweise SOAP-Envelope der Antwort verwendet werden, funktioniert der *`TimeServer`*-Dienst immer gleich. In der Regel ist die Annotation `@WebResult` in einem von einem menschlichen Programmierer geschriebenen Quelltext also nicht erforderlich.

[7] Es gibt zahlreiche kommerzielle Dienste, aus deren WSDL-Dokumenten Java-Artefakte generiert werden können. In Abschnitt 2.3 wird ein Client für den E-Commerce-Dienst von Amazon entwickelt. Dieses Beispiel setzt allerdings eine genauere Untersuchung der Struktur von WSDL-Dokumenten voraus. Die ermüdenden Details werden dabei auf ein Minimum beschränkt. Der nächste Abschnitt behandelt die Grundzüge der Struktur von WSDL-Dokumenten bis hin zu der inoffiziellen, aber gängigen Unterscheidung zwischen der *verpackten* beziehungsweise der *unverpackten Variante* des SOAP-Bodys.

2.2 Die Struktur des WSDL-Dokumentes

[8] Abstrakt gesehen, beschreibt ein WSDL-Dokument einen Vertrag zwischen einem Webservice und seinen Clients. Dieser Vertrag enthält wesentliche Informationen wie den Endpunkt und die Operationen des Dienstes sowie die für diese Operationen erforderlichen Datentypen. Der Vertrag deutet insbesondere, über die Beschreibung der zwischen Webservice und Client ausgetauschten Nachrichten, das unterliegende Nachrichtenaustauschmuster an, zum Beispiel Request/Response oder Solicit/Response. Das oberste (äußerste) Element eines WSDL-Dokumentes, das sogenannte *Wurzel- oder Dokumentelement*, heißt `<definitions>`. Die Definitionen sind im WSDL-Dokument in die folgenden Abschnitte eingeteilt:

- Der optionale *`<types>`-Abschnitt* beinhaltet Definitionen von Datentypen in einem Typsystem wie XML-Schema[1]. Ein spezielles Dokument zur Definition von Datentypen ist die *XML Schema Definition* (XSD). Der `<types>`-Abschnitt beinhaltet, verweist auf oder importiert ein XML-Schema. Ist der `<types>`-Abschnitt leer, wie etwa beim *`ch01.ts.TimeServer`*-Dienst in Unterabschnitt 1.2.3, so verwendet der Dienst nur in XML-Schema vordefinierte einfache Datentypen wie `xsd:string` oder `xsd:long`.

  Obwohl Version 2.0 der WSDL-Spezifikation Alternativen zu XML-Schema gestattet (siehe *http://www.w3.org/TR/wsdl20-altschemalangs*), ist XML-Schema bei WSDL-Dokumenten sowohl Standard- als auch dominantes Typsystem. Dementsprechend wird bei den folgenden Beispielen XML-Schema verwendet, sofern nicht anders angegeben.

- Der *`<message>`-Abschnitt* definiert die Nachrichten, die den Dienst implementieren. Die Nachrichten basieren entweder auf den im unmittelbar vorangegangenen `<types>`-Abschnitt definierten Datentypen oder, falls dieser leer ist, auf den verfügbaren Standardtypen. Außerdem zeigt die Reihenfolge der Nachrichten das Nachrichtenaustauschmuster an. Die Richtungen „Eingabe“ und „Ausgabe“ der Nachrichten beziehen sich, wie bereits auf Seite 8 beschrieben, auf die Perspektive des Dienstes: Eine Eingabenachricht ist zum Dienst hin gerichtet, eine Ausgabenachricht vom Dienst fort. Dementsprechend zeigt die Reihenfolge Eingabe/Ausgabe das Request/Response-Muster an, während die Reihenfolge Ausgabe/Eingabe auf das Solicit/Response-Muster hindeutet. Der *`ch01.ts.TimeServer`*-Dienst hat vier Nachrichten: je eine Anfrage und eine Antwort für jede der beiden Operationen `getTimeAsString()` und `getTimeAsElapsed()`. Die Anordnung der Eingabenachricht vor der Ausgabenachricht in beiden Paaren zeigt an, daß die Operationen des Dienstes dem Request/Response-Muster gehorchen.

- Der *`<portType>`-Abschnitt* beschreibt den Dienst als eine Menge benannter Operationen, wobei jede Operation auf einer oder mehreren Nachrichten aufbaut. Beachten Sie, daß die Operationen nach Methoden benannt sind, die mit der Annotation `@WebMethod` gekennzeichnet sind. Dieser Punkt wird [Unterabschnitt 2.2.2] detailliert diskutiert. Der `<portType>`-Abschnitt eines Webservice' ähnelt der Definition eines Interface' bei Java, in dem der Dienst in abstrakter Form dargestellt wird, das heißt ohne Implementierungsdetails.

- Im *`<binding>`-Abschnitt* gehen die Definitionen des WSDL-Dokumentes von der abstrakten in eine konkrete Form über. Ein `<binding>`-Element (eine *WSDL-Bindung*) ähnelt der Implementierung eines Interface' bei Java (eines `<portType>`-Abschnitts). Eine WSDL-Bindung beinhaltet, wie die implementierende Klasse bei Java, wesentliche konkrete Details über den Dienst. Der `<binding>`-Abschnitt ist der komplizierteste Abschnitt des WSDL-Dokumentes,

[1] Anmerkung des Übersetzers: Der Begriff „XML-Schema“ bezeichnet in der deutschen Übersetzung sowohl die Sprache als auch das im `<types>`-Abschnitt enthaltene, referenzierte oder importierte Dokument beziehungsweise eine Datei mit der Endung *.xsd*.

da hier die folgenden, unter `<portType>` abstrakt definierten Implementierungsdetails aufgeführt werden müssen:

- Das zum Senden und Empfangen der unterliegenden SOAP-Nachrichten verwendete Transportprotokoll. Als Protokoll in der Anwendungsschicht, das heißt als Protokoll zum Transport der SOAP-Nachrichten, die den Dienst implementieren, können entweder HTTP oder *SMTP* (Simple Mail Transfer Protocol) verwendet werden. HTTP ist mit Abstand die häufigste Wahl. Das WSDL-Dokument des *`ch01.ts.TimeServer`*-Dienstes enthält das folgende Tag:

  ```
  <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http">
  ```

  Der Wert des *`transport`-Attributes* zeigt an, daß die SOAP-Nachrichten dieses Dienstes über HTTP gesendet und empfangen werden („*SOAP over HTTP*“).

- Der sogenannte *Stil* des Dienstes ist bereits in Beispiel 1.1 (Seite 5) in Gestalt des Wertes des *`style`-Attributes* aufgetreten. Ein Dienst hat entweder den RPC-Stil (Attributwert `rpc`) oder den Dokumentstil (`document`). Der Dokumentstil ist voreingestellt. Dadurch klärt sich, warum das SEI des *`ch01.ts.TimeServer`*-Dienstes die folgende Annotation trägt:

  ```
  @SOAPBinding(style = Style.RPC)
  ```

  Diese Annotation bewirkt, daß das `style`-Attribut des per Java generierten WSDL-Dokumentes mit `rpc` bewertet wird. Der Unterschied zwischen dem RPC- und dem Dokumentstil wird in Unterabschnitt 2.2.2 erklärt.

- Die für die SOAP-Nachrichten verwendete Zeichenkodierung (*character encoding* oder nur *encoding*). Es gibt zwei Varianten: `literal` und `encoded`, die am Ende von Unterabschnitt 2.2.2 erläutert werden.

- Der *`<service>`-Abschnitt* führt einen oder mehrere Endpunkte auf, an denen die Funktionalität des Dienstes, das heißt die Summe seiner Operationen, zur Verfügung steht. Aus technischer Sicht besteht der `<service>`-Abschnitt aus einem oder mehreren `<port>`-Elementen, die jeweils ein `<portType>`- (Interface) und ein entsprechendes `<binding>`-Element (Implementierung) enthalten. Der Begriff *Port* stammt aus dem Gebiet der verteilten Systeme. Eine über eine bestimmte Netzwerkadresse (zum Beispiel 127.0.0.1) verfügbare Applikation, ist für lokale oder entfernte Clients über einen bestimmten Port erreichbar. Beispielsweise können Clients den *`ch01.ts.TimeServer`*-Dienst über Port 9876 erreichen (siehe `main()`-Methode des `Endpoint`-Publishers `ch01.ts.TimeServerPublisher` in Unterabschnitt 1.2.2).

Das Verzwickte am `<binding>`-Abschnitt sind unter anderem die möglichen Kombinationen der Attribute `style` und `use`. Der folgende Unterabschnitt betrachtet die Beziehungen zwischen diesen Attributen genauer.

2.2.1 WSDL-Bindungen

[9] Das `style`-Attribut des `<binding>`-Elementes kann mit `rpc` („RPC-Stil“) oder `document` („Dokumentstil“) bewertet werden. Die Voreinstellung ist `document`. Das `use`-Attribut des `<Body>`-Elementes kann die Werte `literal` und `encoded` annehmen. Die Voreinstellung ist `literal`. Daraus ergeben sich theoretisch die vier Kombinationsmöglichkeiten in Tabelle 2.1.

[10] Nur zwei der vier Kombinationsmöglichkeiten in Tabelle 2.1 kommen bei heutigen SOAP-basierten Webservices tatsächlich vor: `document`/`literal` und `rpc`/`literal`. Der Attributwert `en-`

style-Attribut	rpc-Attribut
`document`	`literal`
`document`	`encoded`
`rpc`	`literal`
`rpc`	`encoded`

Tabelle 2.1: Mögliche Kombinationen der Werte der Attribute `style` und `use`.

`coded` ist zwar gültig, aber nicht mit den *WS-I-Richtlinien* (Web Services Interoperability, *http://www.ws-i.org*) verträglich, um wenigstens einen Grund zu nennen. Die WS-I-Initiative hat sich die Aufgabe gestellt, Softwarearchitekten und -entwicklern dabei zu helfen, Webservices zu schreiben, die nahtlos über unterschiedliche Plattformen und Programmiersprachen hinweg zusammenzuarbeiten.

[11] Ehe weitere Details zur Sprache kommen, ist es nützlich, ein Beispiel für ein WSDL-Dokument eines Dienstes im Dokumentstil zur Verfügung zu haben, das sich mit dem WSDL-Dokument des *`ch01.ts.TimeServer`*-Dienstes (RPC-Stil) vergleichen läßt. Zunächst wird das `style`-Attribut erläutert, `use` folgt am Ende von Unterabschnitt 2.2.2.

[12] Der *`ch01.ts.TimeServer`*-Dienst aus dem ersten Kapitel läßt sich in drei einfachen Schritten in einen Dienst im Dokumentstil umwandeln. Die neue Version im Dokumentstil wird dem Package `ch02.tsd` zugeordnet. Der erste Schritt besteht darin, für die Version im Dokumentstil ein neues Verzeichnis *ch02/tsd* anzulegen, den Inhalt des ursprünglichen Verzeichnisses *ch01/ts* nach *ch02/tsd* zu kopieren und die `package`-Anweisungen der Dateien in *ch02/tsd* zu korrigieren. Der zweite Schritt besteht darin, die folgende Zeile im Quelltext *ch02/tsd/TimeServer.java* des SEI auszukommentieren, bevor die Datei erneut übersetzt wird:

```
@SOAPBinding(style = Style.RPC)
```

Das Auskommentieren dieser Annotation bewirkt, daß der Dienst den voreingestellten Dokumentstil annimmt. Der dritte Schritt besteht darin, im Arbeitsverzeichnis das folgende Kommando aufzurufen:

```
% wsgen -keep -cp . ch02.tsd.TimeServerImpl
```

Vor dem Aufruf des `wsgen`-Kommandos müssen das geänderte SEI und die zugehörige SIB neu übersetzt werden, damit der Dokumentstil wirksam wird. Das `wsgen`-Kommando generiert anschließend vier *.java* und vier *.class* Dateien im Unterverzeichnis *ch02/tsd/jaxws*. Diese Dateien beinhalten die Datentypen, die ein Dienst im Dokumentstil benötigt, um das WSDL-Dokument automatisch erzeugen zu können. Unter den generierten Dateien sind beispielsweise Quelltext- und übersetzte Versionen der Klassen `GetTimeAsElapsed` (Anfragetyp) und `GetTimeAsElapsedResponse` (Antworttyp). Beide Typen dienen erwartungsgemäß Anfragen und Antworten an die Operation `getTimeAsElapsed()` des Dienstes. Das Kommando `wsgen` generiert entsprechende Typen für die Operation `getTimeAsString()`. Das Kommando `wsgen` wird in Abschnitt 2.4 untersucht.

[13] Der neue *`ch02.tsd.TimeServer`*-Dienst läßt sich wie in Unterabschnitt 1.2.2 mit Hilfe des `Endpoint`-Publishers `ch02.tsd.TimeServerPublisher` in Betrieb nehmen. Das oben angekündigte WSDL-Vergleichsdokument kann über die URL *http://localhost:7777/tsd?wsdl* angefordert werden. Die beiden verschiedenen WSDL-Dokumente werden nun verwendet, um die Unterschiede zwischen dem Dokument- und dem RPC-Stil im Detail zu erklären. Es gibt im Vergleich mit dem dem ursprünglichen Interface *`ch01.ts.TimeServer`* noch eine Änderung, die sich im WSDL-Dokument niederschlägt: Die Packagezuordnung ändert sich von `ch01.ts` in `ch02.tsd`. Diese Änderung äußert sich in einem für das gesamte WSDL-Dokument gültigen Namensraum.

2.2.2 Wesentliche Eigenschaften des Dokumentstils

[14] Beim Dokumentstil bestehen die Nachrichten des SOAP-basierten Dienstes aus ganzen *XML-Dokumenten*, zum Beispiel der Produktliste eines Unternehmens oder einer Kundenbestellung einiger Produkte aus dieser Liste. Beim RPC-Stil bestehen die SOAP-Nachrichten dagegen bei Anfragen aus Parametern und bei Antworten aus Rückgabewerten. Die folgenden Zeilen stammen aus dem WSDL-Dokument des ursprünglichen *ch01.ts.TimeServer*-Dienstes (RPC-Stil):

```
<types></types>

<message name="getTimeAsString"></message>
<message name="getTimeAsStringResponse">
  <part name="time_response" type="xsd:string"></part>
</message>
<message name="getTimeAsElapsed"></message>
<message name="getTimeAsElapsedResponse">
  <part name="time_response" type="xsd:long"></part>
</message>
```

Der `<types>`-Abschnitt ist leer, da der Dienst nur die einfachen vordefinierten Typen *xsd:string* und *xsd:long* als Rückgabetypen verwendet. Diese Typen brauchen im `<types>`-Abschnitt eines WSDL-Dokumentes nicht deklariert zu werden. Die `name`-Attribute der `<message>`-Elemente dokumentieren die Beziehung zu den entsprechenden, mit *@WebMethod* annotierten Methoden, hier *getTimeAsString()* und *getTimeAsElapsed()*. Die `<message>`-Elemente der Antworten haben ein `<part>`-Unterelement, das den Datentyp des Rückgabewertes angibt. Da die Anfragenachrichten keine Argumente erwarten, tritt bei den entsprechenden `<message>`-Elementen kein `<part>`-Unterelement auf.

[15] Im Gegensatz dazu, nun der entsprechende Abschnitt des WSDL-Dokumentes des *ch02.tsd.TimeServer*-Dienstes (Dokumentstil):

```
<types>
  <xsd:schema>
    <xsd:import
        schemaLocation="http://localhost:7777/tsd?xsd=1"
        namespace="http://tsd.ch02/">
    </xsd:import>
  </xsd:schema>
</types>

<message name="getTimeAsString">
  <part element="tns:getTimeAsString" name="parameters"></part>
</message>
<message name="getTimeAsStringResponse">
  <part element="tns:getTimeAsStringResponse" name="parameters"></part>
</message>
<message name="getTimeAsElapsed">
  <part element="tns:getTimeAsElapsed" name="parameters"></part>
</message>
<message name="getTimeAsElapsedResponse">
  <part element="tns:getTimeAsElapsedResponse" name="parameters"></part>
</message>
```

Der `<types>`-Abschnitt enthält nun ein `<import>`-Element für das zugehörige XML-Schema. Die URL des XML-Schemas lautet *http://localhost:7777/tsd?xsd=1*. Das folgende Beispiel zeigt den Quelltext dieses XML-Schemas (*chapter2/validate/ts.xsd*):

Beispiel 2.6: XML-Schema des ch02.tsd.TimeServer-Dienstes (Dokumentstil).

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:tns="http://ts.ch02/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ts.ch02/" version="1.0">
  <xs:element
      name="getTimeAsElapsed"
      type="tns:getTimeAsElapsed" />
  <xs:element
      name="getTimeAsElapsedResponse"
      type="tns:getTimeAsElapsedResponse" />
  <xs:element
      name="getTimeAsString"
      type="tns:getTimeAsString" />
  <xs:element
      name="getTimeAsStringResponse"
      type="tns:getTimeAsStringResponse" />
  <xs:complexType name="getTimeAsString" />
  <xs:complexType name="getTimeAsStringResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="getTimeAsElapsed" />
  <xs:complexType name="getTimeAsElapsedResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:long"></xs:element />
  </xs:complexType>
</xs:schema>
```

Das XML-Schema definiert vier komplexe Typen, deren Namen (zum Beispiel `getTimeAsElapsedResponse`) mit den Namen von Nachrichten im `<message>`-Abschnitt übereinstimmen. Beim *RPC-Stil* verweisen die `name`-Attribute der `<message>`-Elemente auf die Namen der mit `@WebMethod` annotierten Methoden. Der Dokumentstil implementiert eine zusätzliche Komplexitätsstufe, in dem die `<message>`-Elemente zusätzlich auch die im `<types>`-Abschnitt des WSDL-Dokumentes definierten Typen referenzieren.

[16] Das Request/Response-Muster für Webservices ist sowohl im Dokument- als auch im RPC-Stil möglich, auch wenn der RPC-Stil dieses Muster offenbar hervorhebt. Beim RPC-Stil werden die Nachrichten benannt, aber nicht explizit typisiert. Beim Dokumentstil werden die Nachrichten dagegen explizit in einem XML-Schema typisiert.

[17] Der Dokumentstil ist eine sinnvolle Voreinstellung. Der Dokumentstil unterstützt Dienste mit aufwändigen, explizit definierten Datentypen, da die benötigten Typen in einem XML-Schema innerhalb des WSDL-Dokumentes definiert werden können. Der Dokumentstil unterstützt außerdem sämtliche Nachrichtenaustauschmuster, insbesondere Request/Response. Die beim *`TimeServer`*-Dienst ausgetauschten SOAP-Nachrichten sind im RPC- (`ch01.ts`) und im Dokumentstil (`ch02.tsd`) identisch. Aus der Architekturperspektive, ist der Dokumentstil die einfachere Variante, da der SOAP-Body ein abgeschlossenes, genau definiertes Dokument ist. Der RPC-Stil benötigt `<message>`-Elemente mit den Namen der zugehörigen Operationen (bei Java die mit `@WebMethod` annotierten Methoden) mit Parametern in `<part>`-Unterelementen. Aus der Entwicklerperspektive ist dagegen der RPC-Stil die einfachere Variante, weil das *`wsgen`-Kommando* nicht aufgerufen werden muß, um Java-Typen zu generieren, die den im XML-Schema definierten Typen entsprechen. (Das aktuelle Metro-Release erzeugt die `wsgen`-Artefakte automatisch, siehe Kasten auf Seite 67.)

[18] Schließlich bestimmt das `use`-Attribut die Zeichenkodierung beziehungsweise -dekodierung der Typen des Dienstes. Das WSDL-Dokument legt fest, wie die Typen der Implementierungssprache (zum Beispiel Java) in die WSDL-kompatiblen Typen (Voreinstellung ist XML-Schema) serialisiert werden. Auf der Seite des Clients müssen die WSDL-kompatiblen Typen in die Typen der dortigen Implementierungssprache deserialisiert werden. Die Einstellung:

```
use = 'literal'
```

bedeutet, *that the service's type definitions literally follow the WSDL document's XML Schema*. Die Einstellung

```
use = 'encoded'
```

bedeutet dagegen, *that the service's type definitions come from encoding rules, typischerweise den Kodierungsregeln von Version 1.1 der SOAP-Spezifikation*. Wie bereits erwähnt, sind die Kombinationen `document/encoded` und `rpc/encoded` nicht kompatibel mit der WS-I. Die Wahl beschränkt sich somit auf `document/literal` sowie `rpc/literal`.

2.2.3 Validierung von SOAP-Anfragenachrichten bezüglich des mit dem WSDL-Dokument verknüpften XML-Schemas

[19] Beim Vergleich des RPC-Stils mit dem Dokumentstil verdient noch ein weiterer Punkt Aufmerksamkeit. Auf der Empfängerseite kann eine SOAP-Nachricht im Dokumentstil einfach bezüglich des mit dem WSDL-Dokument verknüpften XML-Schemas validiert werden. Beim *RPC-Stil* ist die Validierung dagegen komplizierter, da kein solches XML-Schema vorhanden ist. Das folgende Beispiel zeigt ein kleines Java-Programm, welches ein beliebiges XML-Dokument gegen ein beliebiges XML-Schema (*.xsd* Datei) validiert:

Beispiel 2.7: Ein Hilfsprogramm zur Validierung von XML-Dokumenten.

```
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Schema;
import javax.xml.XMLConstants;
import javax.xml.validation.Validator;

class ValidateXML {

    public static void main(String[] args) {
        if (args.length != 2) {
            String msg = "\nUsage: java ValidateXML XMLfile XSDfile";
            System.err.println(msg);
            return;
        }

        try {

            // Read and validate the XML Schema (XSD document).
            final String schema_uri = XMLConstants.W3C_XML_SCHEMA_NS_URI;
            SchemaFactory factory = SchemaFactory.newInstance(schema_uri);
            Schema schema = factory.newSchema(new StreamSource(args[1]));

            // Validate the XML document against the XML Schema.
            Validator val = schema.newValidator();
            val.validate(new StreamSource(args[0]));

        }

        // Return on any validation error.
        catch(Exception e) {
```

```
                System.err.println(e);
                return;
            }
            System.out.println(args[0] + " validated against " + args[1]);
        }
    }
```

Der Inhalt des SOAP-Bodys aus der Antwortnachricht des *ch02.tsd.TimeServer*-Dienstes (Dokumentstil) lautet bei mir:

```
<ns2:getTimeAsElapsedResponse xmlns:ns2="http://tsd.ch02/">
    <return>1208229395922</return>
</ns2:getTimeAsElapsedResponse>
```

Der Körper wurde durch Einsetzen der Namensraumdeklaration `xmlns:ns2="http://tsd.ch02/"` geringfügig nachbearbeitet, welche die Abkürzung `ns2` als Alias oder Stellvertreter des Namensraum-URIs `http://tsd.ch02/` angibt. Dieser URI kommt zwar in der SOAP-Nachricht vor, allerdings bereits im Element `<Envelope>` und nicht erst in dessen Unterelement `<ns1:getTimeAsElapsedResponse>`. Sie finden das komplette XML-Schema in Beispiel 2.6 auf Seite 40 beziehungsweise in der Datei *ts.xsd*. Der Inhalt des SOAP-Bodys der Antwort ist in der lokalen Datei *msg.xml* gespeichert, um die Verarbeitung zu erleichtern. Das Kommando:

```
% java ValidateXML msg.xml ts.xsd
```

validiert den SOAP-Body der Antwort bezüglich des XML-Schemas. Wird der Inhalt des `<return>`-Elementes beispielsweise in `foo bar` geändert, so liefert der Validierungsversuch die Fehlermeldung:

```
'foo bar' is not a valid value for 'integer'
```

Die Fehlermeldung zeigt an, daß die SOAP-Nachricht bezüglich des XML-Schemas des gewählten WSDL-Dokumentes nicht gültig ist.

2.2.4 Die verpackte/unverpackte Variante des Dokumentstils

[20] Im Basic-Profile der WS-I (*http://www.ws-i.org/Profiles/BasicProfile-1.1.html*) ist der Dokumenstil Standard. Der Dokumentstil unterstützt über das XML-Schema im `<types>`-Abschnitt des WSDL-Dokumentes die explizite und genaue Definition der Typen in den SOAP-Nachrichten des Dienstes. Der Dokumentstil fördert somit die Interoperabilität von Webservices, da der Client die für den Dienst benötigten Datentypen und die genaue Struktur der Dokumente in den SOAP-Bodys der Nachrichten ermitteln kann. Dennoch hat auch der RPC-Stil seinen Reiz, da die Namen der Operationen des Dienstes direkt mit der Implementierung zusammenhängen, bei Java zum Beispiel vermittels der mit `@WebMethod` annotierten Methoden. Beispielsweise hat der *ch01.ts.TimeServer*-Dienst die mit *@WebMethod* annotierte Methode *getTimeAsString()*. Dieser Methode entsprechen im WSDL-Dokument die `<message>`-Elemente mit den Attributen `name="getTimeAsString"` und `name="getTimeAsStringResponse"`. Der RPC-Stil erleichtert dem Programmierer die Arbeit.

[21] Die Motivation der zwar inoffiziellen aber weit verbreiteten „verpackten Variante“ des Dokumentstils (kurz „Verpackungsvariante“) besteht darin, einem Dienst im Dokumentstil das Verhalten und die Erscheinungsform eines Dienstes im RPC-Stil zu verleihen. Die Verpackungsvariante versucht, die Vorzüge des Dokument- und des RPC-Stils miteinander zu kombinieren.

[22] Das folgende Beispiel zeigt einen SOAP-Envelope, der einen SOAP-Body mit unverpacktem Inhalt umschließt, das übernächste Beispiel einen SOAP-Body mit verpacktem Inhalt:

Beispiel 2.8: Dokumentstil, unverpackte Variante.

```
<?xml version="1.0" ?>
<!-- Unwrapped document style -->
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <num1 xmlns:ans="http://ts.ch01/">27</num1>
    <num2 xmlns:ans="http://ts.ch01/">94</num2>
  </soapenv:Body>
</soapenv:Envelope>
```

Beispiel 2.9: Dokumentstil, verpackte Variante.

```
<?xml version="1.0" ?>
<!-- Wrapped document style -->
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <addNums xmlns:ans="http://ts.ch01/">
      <num1>27</num1>
      <num2>94</num2>
    </addNums>
  </soapenv:Body>
</soapenv:Envelope>
```

Der SOAP-Body der unverpackten Anfrage hat die beiden Unterelemente `<num1>` und `<num2>`, welche die zu addierenden Zahlen enthalten. Der Name der Operation, welche die Addition ausführt und die Summe als Antwort zurücksendet, ist *nicht* im SOAP-Body enthalten. Im Gegensatz dazu enthält der SOAP-Body der verpackten Anfrage ein einziges `<addNums>`-Element, das den Namen der angeforderten Operation des Dienstes angibt sowie zwei Unterelemente, die je einen Summanden beinhalten. Die verpackte Variante gibt die Operation des Dienstes explizit an. Die Argumente der Operation sind in intuitiver Weise geschachtelt, das heißt als *Unterelemente* des „Operationselementes“ `<addNums>`.

[23] Die Richtlinien für die Verpackungsvariante sind einfach. Hier eine Zusammenfassung:

- Das `<Body>`-Element des *SOAP-Envelopes* beinhaltet höchstens ein einziges Unterelement, das soviele Unterelemente als benötigt umschließen darf. Erwartet eine Operation eines Dienstes Argumente und gibt einen Wert zurück, so treten Argumente und Rückgabewert nicht als selbständige Unterelemente des SOAP-Bodys auf, sondern als Unterelemente eines „Zwischenelementes“ (der Verpackung). Beispiel 2.9 demonstriert diese Richtlinie anhand von `<addNums>`, dem einzigen Unterelement des SOAP-Bodys. Das `<addNums>`-Element beinhaltet wiederum die Unterelemente `<num1>` und `<num2>`.

- Die Beziehung zwischen dem mit dem WSDL-Dokument verknüpften XML-Schema und dem einzigen Unterelement von `<Body>` ist eindeutig festgelegt. Die Version des *`TimeServer`*-Dienstes im Dokumentstil (*`ch02.tsd.TimeServer`*) dient zur Veranschaulichung. Das XML-Schema aus Beispiel 2.6 enthält vier `<complexType>`-Elemente, die jeweils einen Typ definieren, etwa die komplexen Typen `getTimeAsString` und `getTimeAsStringResponse`. Diese Definitionen stehen in der unteren Hälfte des XML-Schemas. In der oberen Hälfte befinden sich Elementdefinitionen mit je einem `name`- und einem `type`-Attribut. Die Werte der `name`-Attribute der `<complexType>`-Elemente treten als Werte der `type`-Attribute der `<element>`-Elemente auf. Beispielsweise ist das `<element>`-Element mit dem Namen `getTimeAsString` mit dem gleichnamigen komplexen Typ verknüpft. Der folgende Ausschnitt aus dem XML-

Schema zeigt den Zusammenhang:

```
<xs:element name="getTimeAsString" type="tns:getTimeAsString" />
<xs:element
    name="getTimeAsStringResponse"
    type="tns:getTimeAsStringResponse" />
...
<xs:complexType name="getTimeAsString" />
<xs:complexType name="getTimeAsStringResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0" />
    </xs:sequence>
</xs:complexType>
```

Ein `<complexType>`-Element ist entweder leer, zum Beispiel bei `name="getTimeAsString"`, oder enthält ein `<sequence>`-Element, zum Beispiel bei `name="getTimeAsStringResponse"` (das `<complexType>`-Element enthält ein `<sequence>`-Unterelement, welches wiederum ein Unterelement enthält). Das `<sequence>`-Element enthält typisierte Argumente beziehungsweise Rückgabewerte. Das *`ch02.tsd.TimeServer`*-Beispiel ist recht einfach, da bei Anfragen keine Argumente und bei Antworten nur ein Rückgabewert vorkommt. Dennoch zeigt der obige Ausschnitt aus dem XML-Schema die Struktur des allgemeinen Falls. Hätte beispielsweise `getTimeAsStringResponse` mehr als einen Rückgabewert, so würde jeder davon als Unterelement von `<sequence>` notiert werden. Beachten Sie außerdem, daß jedes Element im obigen Ausschnitt (genau genommen sogar im gesamten XML-Schema, siehe Beispiel 2.6) benannt und typisiert ist, also ein `name`- und ein `type`-Attribut besitzt.

- Die im XML-Schema definierten Elemente dienen als Verpackungselemente für den Inhalt des SOAP-Bodys. Das folgende Beispiel zeigt einen SOAP-Envelope des *`ch02.tsd.TimeServer`*-Dienstes:

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.or g/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <ans:getTimeAsElapsedResponse xmlns:ans="http://ts.ch01/">
      <return>1205030105192</return>
    </ans:getTimeAsElapsedResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

  Ein Dienst im RPC-Stil erzeugt einen ebensolchen SOAP-Body (dies ist der Sinn der Verpackungsvariante des Dokumentstils). Der Unterschied zwischen der verpackten Variante des Dokumentstils und dem RPC-Stil besteht darin, daß die Verpackungsvariante im Gegensatz zum RPC-Stil explizite Typ- und Formatinformationen in einem XML-Schema im `<types>`-Abschnitt des WSDL-Dokumentes beinhaltet.

- Das „Verpackungselement“ in der Anfrage hat denselben Namen wie die Operation des Dienstes, zum Beispiel `<addNums>` in Beispiel 2.9. Das Verpackungselement in der Antwort hat den Namen des entsprechenden Elementes aus der Anfrage, erweitert um die Endung `Response`, zum Beispiel `<addNumsResponse>`.

- Der `<portType>`-Abschnitt des WSDL-Dokumentes deklariert benannte Operationen, mit *typisierten* Nachrichten. Beispielsweise hat die Eingabenachricht der Operation `getTimeAsString` den Typ `tns:getTimeAsString`, einen der vier komplexen Typen im XML-Schema des WSDL-Dokumentes. Der `<portType>`-Abschnitt des *`ch02.tsd.TimeServer`*-Dienstes (Doku-

mentstil) lautet:

```
<portType name="TimeServerImpl">
  <operation name="getTimeAsString">
    <input message="tns:getTimeAsString"></input>
    <output message="tns:getTimeAsStringResponse"></output>
  </operation>
  <operation name="getTimeAsElapsed">
    <input message="tns:getTimeAsElapsed"></input>
    <output message="tns:getTimeAsElapsedResponse"></output>
  </operation>
</portType>
```

Die verpackte Variante des Dokumentstils ist trotz ihres inoffiziellen Status vorherrschend. Java-Webservices unterstützen diese Variante. Ein in Java geschriebener SOAP-basierter Webservice operiert per Voreinstellung in der Verpackungsvariante des Dokumentstils mit literaler Kodierung.

[24] Diese Exkursion zu den Details des `<binding>`-Abschnitts in WSDL-Dokumenten zahlt sich im nächsten Beispiel aus. Dort werden mit Hilfe des `wsimport`-Kommandos clientseitige Artefakte für den E-Commerce-Dienst von Amazon erzeugt. Der Unterschied zwischen der verpackten und der unverpackten Variante des Dokumentstils erleichtert das Verständnis dieser clientseitigen Artefakte.

2.3 Der E-Commerce-Dienst von Amazon (SOAP-basierter Dienst)

[25] Der E-Commerce-Dienst von Amazon heißt heute offiziell *Amazon Associates Web Service* und ist sowohl in einer SOAP-basierten Version als auch im *REST*-Stil verfügbar. Der Dienst ist zwar kostenlos, aber Amazon verlangt aber eine Registrierung unter der Adresse *http://affiliate-program.amazon.com/gp/associates/join*. Für die Beispiele in diesem Abschnitt ist ein *Amazon-Zugangsschlüssel* erforderlich (im Gegensatz zum *geheimen Schlüssel* beim Erzeugen eines Schlüsselpaares (asymmetrische Verschlüsselung, siehe Unterabschnitt 5.2.2).

[26] Der E-Commerce-Dienst von Amazon repliziert die interaktive Bedienung der Internetseite von Amazon (*http://www.amazon.com*) und unterstützt beispielsweise die Suche nach Artikeln, das Bieten auf Artikel bei Auktionen, das Einstellen von Artikeln in Auktionen, das Anlegen eines Einkaufkorbes, das Aufnehmen von Artikeln in den Korb und so weiter. Die drei Clients in diesem Abschnitt implementieren die Suche nach Artikeln.

[27] Die beiden folgenden Abschnitte stellen je einen Java-Client für den E-Commerce-Dienst von Amazon vor. Beide Clients bauen auf Java-Artefakten auf, die mit Hilfe des in Abschnitt 2.1.1 beschriebenen *`wsimport`-Kommandos* generiert werden. Der Unterschied zwischen den beiden Clients verfeinert die Abgrenzung der verpackten von der unverpackten Variante des Dokumentstils.

2.3.1 Ein Client in der verpackten Variante des Dokumentstils

[28] Die clientseitigen Java-Artefakte werden mit dem folgenden Kommado generiert:

```
% wsimport -keep -p awsClient \
http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl
```

Die Option *`-p awsClient`* des `wsimport`-Kommandos legt ein Package (und somit ein Unterzeichnis) mit dem Namen *awsClient* an.

[29] Der Quelltext des ersten Clients `AmazonClientW` liegt im Arbeitsverzeichnis, das heißt in dem Verzeichnis, welches das Unterverzeichnis *awsClient* enthält. Der Client `AmazonClientW`[2], sucht nach Büchern über Quantengravitation:

Beispiel 2.10: Ein Client für den E-Commerce-Dienst von Amazon (verpackte Variante).

```
import AwsHandlerResolver;
import awsClient.AWSECommerceService;
import awsClient.AWSECommerceServicePortType;
import awsClient.Item;
import awsClient.ItemSearch;
import awsClient.ItemSearchRequest;
import awsClient.Items;
import awsClient.OperationRequest;
import awsClient.SearchResultsMap;

import javax.xml.ws.Holder;
import java.util.List;
import java.util.ArrayList;

class AmazonClientW { // W is for Wrapped style

    public static void main(String[] args) {

        if (args.length != 2) {
          System.err.println("Usage: java AmazonClientW <access key> <secret key>");
          return;
        }

        final String access_key = args[0];
        final String secret_key = args[1];

        // Construct a service object to get the port object.
        AWSECommerceService service = new AWSECommerceService();
        service.setHandlerResolver(new AwsHandlerResolver(secret_key));
        AWSECommerceServicePortType port = service.getAWSECommerceServicePort();

        // Construct an empty request object and then add details.
        ItemSearchRequest request = new ItemSearchRequest();
        request.setSearchIndex("Books");
        request.setKeywords("quantum gravity");

        ItemSearch search = new ItemSearch();
        search.getRequest().add(request);
        search.setAWSAccessKeyId(access_key);
```

[2] Anmerkung des Übersetzers: Das Beispiel in der englischen Originalausgabe funktioniert nicht mehr, da Amazon den Zugangsmodus zu seinen Webservices geändert hat. Der Fehler wurde im September 2009 von Dominik Gruntz gemeldet (siehe Errata-Liste, *http://oreilly.com/catalog/errata.csp?isbn=9780596521134*). Neben dem öffentlichen Schlüssel muß nun auch der geheime Schlüssel übergeben werden, das Programm `AmazonClientW` also mit zwei Argumenten aufgerufen werden. Die deutsche Ausgabe enthält den korrigierten Quelltext des Clients `AmazonClientW`. Zum Übersetzen des Clients sind die Klasse `AwsHandlerResolver` und das Archiv *commons-codec-1.3.jar* erforderlich, die unter der Adresse *http://developer.amazonwebservices.com/connect/servlet/JiveServlet/download/9-35999-14-3146-2680/awshandlerresolver.java* beziehungsweise *http://developer.amazonwebservices.com/connect/servlet/JiveServlet/download/9-35999-143146-2681/commons-codec-1.3.jar* heruntergeladen werden können. (Beachten Sie, daß der Dateiname *awshandlerresolver.java* nach dem Herunterladen in *AwsHandlerResolver.java* geändert werden muß.) Nach dem Übersetzen von *AwsHandlerResolver.java* und der korrigierten Version von *AmazonClientW.java* lautet der Aufruf

```
java AmazonClientW AKIAIEON54C4XHQTHIPQ UXn6cbchk7y2ykcaUzq+SPMKxog6+EzlfcSOazIM
```

Die Schlüssel sind willkürlich gewählt und gehören zu keinem existierenden Konto bei Amazon. Die Liste der gefundenen Bücher auf Seite 48 wurde ebenfalls aktualisiert.

```
        Holder<OperationRequest> operation_request = null;
        Holder<List<Items>> items = new Holder<List<Items>>();

        port.itemSearch(search.getMarketplaceDomain(),
                        search.getAWSAccessKeyId(),
                        search.getSubscriptionId(),
                        search.getAssociateTag(),
                        search.getXMLEscaping(),
                        search.getValidate(),
                        search.getShared(),
                        search.getRequest(),
                        operation_request,
                        items);

        // Unpack the response to print the book titles.
        Items retval = items.value.get(0);       // first and only Items element
        List<Item> item_list = retval.getItem(); // list of Item subelements
        for (Item item : item_list)              // each Item in the list
            System.out.println(item.getItemAttributes().getTitle());

    }

}
```

[30] Der Quelltext wird wie gewohnt übersetzt und aufgerufen, wobei Sie allerdings Ihren öffentlichen und Ihren privaten Zugangsschlüssel auf der Kommandozeile als Argumente übergeben müssen (siehe Fußnote 2 auf Seite 47). Die Ausgabe der Suche nach Büchern zum Thema Quantengravitation lautet bei mir:

```
Three Roads to Quantum Gravity
Quantum Gravity (Cambridge Monographs on Mathematical Physics)
Keeping It Real (Quantum Gravity, Book 1)
Introduction to Quantum Effects in Gravity
Approaches to Quantum Gravity: Toward a New Understanding of Space, Time and Matter
Reinventing Gravity: A Physicist Goes Beyond Einstein
New Paths Towards Quantum Gravity (Lecture Notes in Physics)
Euclidean Quantum Gravity
Quantum Field Theory on Curved Spacetimes: Concepts and Mathematical Foundations
(Lecture Notes in Physics)
Quantum Gravity (International Series of Monographs on Physics)
```

Die Klasse `AmazonClientW` ist alles andere als intuitiv. Der Quelltext bedient sich recht obskurer Typen wie etwa `javax.xml.ws.Holder`. Die `itemSearch()`-Methode, welche die eigentliche Suche ausführt, erwartet zehn Argumente. Das letzte Argument, hier `items`, verweist auf die Antwort des Dienstes. Diese Antwort muß erst „ausgepackt“ werden, um an die Liste der über die Suchanfrage zurückerhaltenen Bücher zu kommen. Der nächste Client (`AmazonClientU`) ist erheblich einfacher. Vor der Betrachtung des einfacheren Clients ist es aber lehrreich, zu untersuchen, warum der erste Client so kompliziert ist.

[31] Die Attribute `style` und `use` im `<binding>`-Abschnitt des WSDL-Dokumentes des E-Commerce-Dienstes von Amazon sind mit ihren Standardwerten `document` (Dokumentstil) beziehungsweise `literal` (literale Kodierung) belegt. Wie der folgende Auszug aus dem XML-Schema dokumentiert, ist außerdem die Verpackungsvariante im Spiel:

```
<xs:element name="ItemSearch">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="MarketplaceDomain" type="xs:string" minOccurs="0" />
      <xs:element name="AWSAccessKeyId"    type="xs:string" minOccurs="0" />
```

```
        <xs:element name="SubscriptionId"    type="xs:string" minOccurs="0" />
        <xs:element name="AssociateTag"      type="xs:string" minOccurs="0" />
        <xs:element name="XMLEscaping"       type="xs:string" minOccurs="0" />
        <xs:element name="Validate"          type="xs:string" minOccurs="0" />
        <xs:element name="Shared"            type="tns:ItemSearchRequest"
            minOccurs="0" />
        <xs:element name="Request"           type="tns:ItemSearchRequest"
            minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Dieser Auszug definiert das Element `<ItemSearch>`, welches die Rolle des Verpackungselementes beim Inhalt des SOAP-Bodys in der Anfrage übernimmt. Der folgende Absatz stammt aus dem `<message>`-Abschnitt des WSDL-Dokumentes und zeigt, daß die Anfragenachricht den oben definierten Typ hat:

```
<message name="ItemSearchRequestMsg">
   <part name="body" element="tns:ItemSearch" />
</message>
```

Der Verpackungstyp der Antwort des Dienstes auf eine Anfrage vom Typ `<ItemSearch>` ist ebenfalls im XML-Schema definiert und lautet:

```
<xs:element name="ItemSearchResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="tns:OperationRequest" minOccurs="0" />
      <xs:element ref="tns:Items" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Die Auswirkungen dieser Typdefinitionen im WSDL-Dokument zeigen sich in der SOAP-Anfragenachricht des Clients beziehungsweise in der SOAP-Antwortnachricht des Servers. Das folgende Beispiel zeigt eine SOAP-Anfrage:

Beispiel 2.11: SOAP-Anfrage an den E-Commerce-Dienst von Amazon.

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://webservices.amazon.com/AWSECommerceService/2008-03-03">
  <soapenv:Body>
    <ns1:ItemSearch>
      <ns1:AWSAccessKeyId>...</ns1:AWSAccessKeyId>
      <ns1:Request>
        <ns1:Keywords>quantum gravity</ns1:Keywords>
        <ns1:SearchIndex>Books</ns1:SearchIndex>
      </ns1:Request>
    </ns1:ItemSearch>
  </soapenv:Body>
</soapenv:Envelope>
```

`<ItemSearch>` ist das einzige Unterelement des SOAP-Bodys und umschließt zwei Unterelemente namens `<ns1:AWSAccessKeyId>` und `<ns1:Request>`. Die Unterelemente von `<ns1:Request>` beinhalten die Suchzeichenkette (hier „quantum gravity“) sowie die Suchkategorie (hier „Books“).

[32] Nun ein Auszug aus der SOAP-Antwort auf die obige Anfrage:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ItemSearchResponse
        xmlns="http://webservices.amazon.com/AWSECommerceService/2008-03-03">
      <OperationRequest>
        <HTTPHeaders>
          <Header Name="UserAgent" Value="Java/1.6.0"></Header>
        </HTTPHeaders>
        <RequestId>0040N1YEKV0CRCT2B5PR</RequestId>
        <Arguments>
          <Argument Name="Service" Value="AWSECommerceService"></Argument>
        </Arguments>
        <RequestProcessingTime>0.0566580295562744</RequestProcessingTime>
      </OperationRequest>
      <Items>
        <Request>
          <IsValid>True</IsValid>
          <ItemSearchRequest>
            <Keywords>quantum gravity</Keywords>
            <SearchIndex>Books</SearchIndex>
          </ItemSearchRequest>
        </Request>
        <TotalResults>207</TotalResults>
        <TotalPages>21</TotalPages>
        <Item>
          <ASIN>061891868X</ASIN>
          <DetailPageURL>http://www.amazon.com/gp/redirect.html...
          </DetailPageURL>
          <ItemAttributes>
            <Author>Lee Smolin</Author>
            <Manufacturer>Mariner Books</Manufacturer>
            <ProductGroup>Book</ProductGroup>
            <Title>The Trouble With Physics</Title>
          </ItemAttributes>
        </Item>
        ...
      </Items>
    </ItemSearchResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Der SOAP-Body beinhaltet nun ein einziges `<ItemSearchResponse>`-Element, dessen Typ im WSDL-Dokument des Dienstes definiert ist. Dieses Element enthält verschiedene Unterelemente. Das interessanteste davon ist `<Items>`. Das `<Items>`-Element enthält mehrere `<Item>`-Elemente, je eines für jedes Buch über Quantengravitation. Der Auszug zeigt nur ein `<Item>`-Element. Das genügt, um die Struktur des Antwortdokumentes zu erkennen. Die letzten Anweisungen in der Klasse `AmazonClientW` spiegeln die Struktur der SOAP-Antwort wider: Zuerst wird eine Referenz auf das `<Items>`-Element gespeichert und anschließend eine Liste von `<Item>`-Unterelementen, die jeweils den Buchtitel als Attribut mitführen.

[33] Wir betrachten nun eines der aus dem WSDL-Dokument E-Commerce-Dienstes von Amazon erzeugten `wsimport`-Artefakte, insbesondere die Annotationen der `itemSearch()`-Methode mit ihren zehn Argumenten. Die Methode ist im Interface *`AWSECommerceServicePortType`* deklariert, welches zu jeder Operation des E-Commerce-Dienstes eine einzige, mit `@WebMethod` annotierte Methode deklariert. Das folgende Beispiel zeigt die Deklaration der `itemSearch()`-Methode:

Beispiel 2.12: Auszug aus den wsimport-Artefakten des E-Commerce-Dienstes von Amazon (Deklaration der itemSearch()-Methode).

```
@WebMethod(operationName = "ItemSearch", action = "http://soap.amazon.com")

@RequestWrapper(localName = "ItemSearch",
  targetNamespace = "http://webservices.amazon.com/AWSECommerceService/2008-03-03",
  className = "awsClient.ItemSearch")

@ResponseWrapper(localName = "ItemSearchResponse",
  targetNamespace = "http://webservices.amazon.com/AWSECommerceService/2008-03-03",
  className = "awsClient.ItemSearchResponse")

public void itemSearch(

  @WebParam(name = "MarketplaceDomain",
  targetNamespace = "http://webservices.amazon.com/AWSECommerceService/2008-03-03")
  String marketplaceDomain,

  @WebParam(name = "AWSAccessKeyId",
  targetNamespace = "http://webservices.amazon.com/AWSECommerceService/2008-03-03")
  String awsAccessKeyId,
  ...
  ItemSearchRequest shared,

  @WebParam(name = "Request",
  targetNamespace = "http://webservices.amazon.com/AWSECommerceService/2008-03-03")
  List<ItemSearchRequest> request,

  @WebParam(name = "OperationRequest",
  targetNamespace = "http://webservices.amazon.com/AWSECommerceService/2008-03-03",
  mode = WebParam.Mode.OUT)
  Holder<OperationRequest> operationRequest,

  @WebParam(name = "Items",
  targetNamespace = "http://webservices.amazon.com/AWSECommerceService/2008-03-03",
  mode = WebParam.Mode.OUT)
  Holder<List<Items>> items);
```

[34] Die beiden letzten Parameter der `itemSearch()`-Methode, `operationRequest` und `items`, sind als `WebParam.Mode.OUT` bezeichnet, um sie als Rückgabewerte des E-Commerce-Dienstes an den Client auszuweisen. Ein `OUT`-Parameter wird in einem `Holder`-Objekt zurückgegeben. Die `itemSearch()`-Methode reflektiert somit die im XML-Schema des WSDL-Dokumentes definierten Typen. Der im XML-Schema definierte Typ `<itemSearch>` (der Verpackungstyp in der Anfrage) hat acht Unterelemente, darunter `<MarketplaceDomain>`, `<AWSAccessKeyId>` und `<ItemSearchRequest>`. Jedes dieser Unterelemente kommt als Parameter der `itemSearch()`-Methode vor. Der im XML-Schema definierte Typ `<itemSearchResponse>` hat zwei Unterelemente namens `<OperationRequest>` und `<Items>`, die den letzten beiden Parametern (den `OUT`-Parametern) der `itemSearch()`-Methode entsprechen. Die Schwierigkeit für den Programmierer besteht gerade darin, daß die `itemSearch()`-Methode angesichts ihrer zehn Parameter kompliziert aufzurufen ist, vor allem, weil die beiden letzten Parameter die Rückgabewerte beinhalten.

[35] Warum bewirkt die Verpackungsvariante einen so komplizierten Client? Die Verpackungsvariante hat das Ziel, einem Dienst im Dokumentstil das Verhalten und die Erscheinungsform eines

Diensts im RPC-Stil zu verleihen, ohne die Vorzüge des Dokumentstils aufzugeben. Der Dokumentstil mit Verpackung benötigt ein Verpackungselement, typischerweise mit dem Namen der Operation des Dienstes, hier `ItemSearch`, mit einem typisierten Unterelement für jeden Parameter der Operation. Die `itemSearch()`-Operation des E-Commerce-Dienstes von Amazon hat acht Eingabe- beziehungsweise Anfrageparameter und zwei Ausgabe- beziehungsweise Antwortparameter, darunter den entscheidenden Antwortparameter `Items`. Die Elemente, welche die Parameter repräsentieren, sind als Unterelemente von `<sequence>` definiert, das heißt jeder Parameter steht an einer festen Position. Der Antwortparameter `Items` muß beispielsweise als letzter in der Liste der zehn Parameter stehen, da der `@ResponseWrapper`-Abschnitt nach dem `@RequestWrapper`-Abschnitt kommt und `Items` der letzte Eintrag unter `@ResponseWrapper` ist. Das Fazit lautet, daß die Verpackungsvariante des Dokumentstils einen recht komplizierten Aufruf der `itemSearch()`-Methode bewirkt. Ohne einen Workaround ist die Verpackungsvariante durchaus in der Lage, den Programmierer zu verwirren. Der nächste Unterabschnitt stellt einen solchen Workaround vor.

2.3.2 Ein Client in der unverpackten Variante des Dokumentstils

[36] Der auf den `wsimport`-Artefakten aufbauende Client in der unverpackten Variante des Dokumentstils ist einfacher als der Client in der verpackten Variante im vorigen Unterabschnitt. Die Artefakte für die unverpackten Variante werden aber aus demselben WSDL-Dokument generiert, wie die Artefakte für den komplizierten Client in der verpackten Variante. Die unterliegenden SOAP-Nachrichten haben sowohl beim Client in der verpackten als auch bei der unverpackten Variante ein und dieselbe Struktur. Die Clients unterscheiden sich dagegen deutlich voneinander. Der einfachere Client ist viel leichter zu schreiben und zu verstehen. Der Aufruf der `itemSearch()`-Methode beim einfacheren Client lautet:

```
ItemSearchResponse response = port.itemSearch(item_search);
```

Die `itemSearch()`-Methode erwartet nun nur ein Argument und gibt nur einen Wert zurück, welcher der Referenzvariablen `response` zugewiesen wird. Im Gegensatz zum komplizierten Client im vorigen Unterabschnitt verwendet der einfache Client kein `Holder`-Objekt, um den Rückgabewert zu verpacken (eine exotische und schwierige Konstruktion). Der Aufruf der `itemSearch()`-Methode wirkt vertraut und entspricht einem Methodenaufruf in einer selbständigen Applikation. Das folgende Beispiel zeigt den kompletten vereinfachten Client:

Beispiel 2.13: Ein Client für den E-Commerce-Dienst von Amazon (unverpackte Variante).

```
import awsClient2.AWSECommerceService;
import awsClient2.AWSECommerceServicePortType;
import awsClient2.ItemSearchRequest;
import awsClient2.ItemSearchResponse;
import awsClient2.ItemSearch;
import awsClient2.Items;
import awsClient2.Item;
import java.util.List;

class AmazonClientU { // U is for Unwrapped style

    public static void main(String[] args) {

        // Usage
        if (args.length != 1) {
            System.err.println("Usage: java AmazonClientW <access key<");
            return;
        }
```

```
        final String access_key = args[0];

        // Create service and get portType reference.
        AWSECommerceService service = new AWSECommerceService();
        AWSECommerceServicePortType port = service.getAWSECommerceServicePort();

        // Create request.
        ItemSearchRequest request = new ItemSearchRequest();

        // Add details to request.
        request.setSearchIndex("Books");
        request.setKeywords("quantum gravity");
        ItemSearch item_search= new ItemSearch();
        item_search.setAWSAccessKeyId(access_key);
        item_search.getRequest().add(request);

        // Invoke service operation and get response.
        ItemSearchResponse response = port.itemSearch(item_search);

        List<Items> item_list = response.getItems();
        for (Items next : item_list)
            for (Item item : next.getItem())
                System.out.println(item.getItemAttributes().getTitle());

    }
}
```

[37] Das Erzeugen der *wsimport*-Artefakte für den vereinfachten Client ist ironischerweise schwieriger als beim komplizierten Client. Das Kommando ist zur besseren Lesbarkeit auf drei Zeilen verteilt:

```
% wsimport -keep -p awsClient2
  http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl
  -b custom.xml .
```

Die Option *-b* am Ende der Zeile erwartet ein applikationsspezifisches `<jaxws:bindings>`-Dokument, hier die Datei *custom.xml*, um Standardeinstellungen des `wsimport`-Kommandos zu überschreiben, in diesem Fall die Voreinstellung `true` des `WrapperStyle`. Das folgende Beispiel zeigt das `<jaxws:bindings>`-Dokument mit der geänderten Bindungseinstellung:

Beispiel 2.14: Anpassen einer Bindungseinstellung beim wsimport-Kommando.

```
<jaxws:bindings
    wsdlLocation =
      "http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
  <jaxws:enableWrapperStyle>false</jaxws:enableWrapperStyle>
</jaxws:bindings>
```

Die [Auswirkung(en)/Singular oder Plural] des applikationsspezifischen Bindungsdokumentes zeigen sich in den generierten Artefakten. Das Artefakt für das Interface *AWSECommerceServicePortType* wird zu (gekürzt):

```
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)

public interface AWSECommerceServicePortType {

    ...

    @WebMethod(operationName = "ItemSearch", action = "http://soap.amazon.com")

    @WebResult(name = "ItemSearchResponse",
      targetNamespace =
        "http://webservices.amazon.com/AWSECommerceService/2008-04-07",
      partName = "body")
```

```
public ItemSearchResponse itemSearch(
    @WebParam(name = "ItemSearch",
      targetNamespace =
        "http://webservices.amazon.com/AWSECommerceService/2008-04-07",
      partName = "body")
    ItemSearch body);
```

Das per `wsimport` generierte Interface *`AWSECommerceServicePortType`* trägt nun die Annotation `@SOAPBinding.ParameterStyle.BARE`, wobei die Alternativen `BARE` und `WRAPPED` zur Verfügung stehen. Bei Java Webservices heißt das entsprechende Attribut `parameterStyle`, da sich der Kontrast zwischen der verpackten und der unverpackten Variante des Dokumentstils in der Darstellung der Parameter im SOAP-Body niederschlägt. In der unverpackten Variante werden die Parameter unverhüllt aufgeführt, das heißt als Sequenz direkter Unterelemente des SOAP-Bodys. In der verpackten Variante werden die Parameter im SOAP-Body in ein „Zwischenelement“ mit dem Namen der Operation des Dienstes eingebettet. Dieses Zwischenelement ist das einzige direkte Unterelement des SOAP-Bodys.

[38] Überraschenderweise bleibt die Struktur der unterliegenden SOAP-Nachrichten sowohl der Anfrage als auch der Antwort unverändert. Beispielsweise ist die Anfragenachricht des vereinfachten Clients `AmazonClientU` hinsichtlich ihrer Struktur identisch mit der Anfragenachricht des komplizierteren Clients `AmazonClientW`. Das folgende Beispiel zeigt den Inhalt des SOAP-Bodys einer Anfrage des vereinfachten Clients:

```
<soapenv:Body>
  <ns1:ItemSearch>
    <ns1:AWSAccessKeyId>...</ns1:AWSAccessKeyId>
    <ns1:Request>
      <ns1:Keywords>quantum gravity</ns1:Keywords>
      <ns1:SearchIndex>Books</ns1:SearchIndex>
    </ns1:Request>
  </ns1:ItemSearch>
</soapenv:Body>
```

Der SOAP-Body enthält ein einziges Verpackungselement (`<ns1:ItemSearch>`) mit Unterelementen für den Zugangsschlüssel und die Anfrageparameter. Der komplizierte Client in der Verpackungsvariante erzeugt Anfragen mit identischer Struktur (siehe Beispiel 2.11, Seite 49).

[39] Der wesentliche Unterschied liegt natürlich im clientseitigen Java-Quelltext. Der vereinfachte Client in der unverpackten Variante ruft die `itemSearch()`-Methode mit einem Argument vom Typ `ItemSearch` auf und erwartet einen einzelnen Rückgabewert vom Typ `ItemSearchResponse`. Das `parameterStyle`-Attribut mit dem Wert `BARE` verhindert somit den komplizierten Aufruf von `itemSearch()` mit zehn Argumenten, von denen acht an Unterelemente in `@RequestWrapper` und zwei an Unterelemente in `@ResponseWrapper` gebunden sind.

[40] Das Beispiel des E-Commerce-Dienstes von Amazon zeigt, daß die Verpackungsvariante des Dokumentstils, trotz ihrer Vorteile, das Schreiben eines Clients verkomplizieren kann. In diesem Beispiel ist der einfache unverpackte Client, dessen `parameterStyle`-Attribut mit `BARE` bewertet ist, ein Workaround.

[41] Das WSDL-Dokument des E-Commerce-Dienstes von Amazon könnte vereinfacht werden, um das Schreiben von Clients zu erleichtern. Es ist bei kommerziell verfügbaren SOAP-basierten Webservices nicht ungewöhnlich, komplizierte WSDL-Dokumente vorzufinden, die komplizierte Clients nach sich ziehen.

2.3.3 Vor- und Nachteile des RPC- und des Dokumentstils

[42] Java Webservices unterstützen nach wie vor sowohl den RPC- als auch den Dokumentstil. Standardstil ist der Dokumentstil. Bei beiden Stilen wird im Hinblick auf die Kompatibilität mit dem Basic-Profile der WS-I nur die literale Kodierung unterstützt. Die Frage ob der RPC- oder Dokumentstil gewählt werden sollte, wird häufig als *Entscheidungsfreiheit* angepriesen. Der Dokumentstil, speziell in der Verpackungsvariante, gewinnt in der Wahrnehmung dennoch rapide an Boden. Dieser Unterabschnitt stellt einige Vor- und Nachteile der beiden Stile zusammen.

[43] Die Vor- und Nachteile müssen, wie bei jeder solche Diskussion, kritisch betrachtet werden, insbesondere dann, wenn sich eine Eigenschaft oder Fähigkeit sowohl zum Vor- als auch zum Nachteil auswirken kann. Ein häufiger Kritikpunkt gegen den RPC-Stil lautet, daß dieser Stil dem Dienst das Nachrichtenaustauschmuster Request/Response überstülpt. Diesem Einwand zum Trotz dominiert das Request/Response-Muster bei SOAP-basierten Diensten in der Realität und es gibt offensichtlich Situationen, in denen dieses Muster benötigt wird (etwa die Gültigkeitsprüfung einer Kreditkarte vor der Bezahlung).

[44] Einige Vorteile der RPC-Stils sind:

- Das automatisch generierte WSDL-Dokument ist relativ kurz und einfach, da es keinen `<types>`-Abschnitt enthält.
- Die `<message>`-Elemente des WSDL-Dokumentes beinhalten die Namen der unterliegenden Operationen des Webservice'. Bei Java sind dies die mit `@WebMethod` annotierten Methoden. Das WSDL-Dokument folgt bezüglich der Operationen daher der Richtlinie „What you see is what you get".
- Der Durchsatz an Nachrichten ist höher, da die Nachrichten keine Informationen über ~~Typkodierung~~ transportieren.

[45] Einige Nachteile des RPC-Stils sind:

- Ein WSDL-Dokument mit leerem `<types>`-Abschnitt hat kein XML-Schema zur Verfügung, bezüglich dessen der Inhalt des SOAP-Bodys einer Nachricht validiert werden kann.
- Ein Dienst im RPC-Stil kann keine beliebig ausgestalteten Datentypen verwenden, da kein XML-Schema zur Verfügung steht, in dem derartige Typen definiert werden können und ist daher auf relativ einfache Typen wie ganze Zahlen, Zeichenketten, Datumswerte und Arrays dieser Typen beschränkt.
- Der RPC-Stil begünstigt durch seine offenkundige Beziehung zum Nachrichtenaustauschmuster Request/Response eine *starke Kopplung* zwischen Dienst und Client. Der Java-Client `ch01.ts.TimeClient` beispielsweise, *blockiert* beim Aufruf der Methode

  ```
  port.getTimeAsString()
  ```

  solange, bis entweder der Dienst antwortet oder eine Ausnahme ausgeworfen wird. Dieselbe Eigenschaft wird auch durch die Sprechweise ausgedrückt, dem RPC-Stil wohne synchrones Aufrufverhalten inne (im Gegensatz zu asynchronem Verhalten). Der folgende Unterabschnitt stellt einen Workaround vor, der die Art und Weise veranschaulicht, in der Java Webservices nichtblockierende Clients im Request/Response-Muster unterstützen.
- In Java geschriebene Dienste im RPC-Stil sind unter Umständen in anderen Frameworks nicht nutzbar und untergraben dadurch die Interoperabilität. Außerdem ist die langfristige Unterstützung in der Webservice-Community und durch die WS-I-Gruppe nicht zweifelsfrei geklärt.

[46] Einige Vorteile des Dokumentstils sind:

- Der Inhalt des SOAP-Bodys einer Nachricht kann bezüglich des XML-Schemas im `<types>`-Abschnitt des WSDL-Dokumentes validiert werden.
- Ein Dienst im Dokumentstil kann beliebig ausgestaltete Datentypen verwenden, da die Sprache XML-Schema nicht nur einfache Datentypen wie ganze Zahlen, Zeichenketten und Datumswerte unterstützt, sondern auch beliebig konstruierte komplexe Typen.
- Der Inhalt des SOAP-Bodys kann sehr flexibel strukturiert werden, solange sich die Struktur in einem XML-Schema definieren läßt.
- Die Verpackungsvariante des Dokumentstils gestattet, einen wesentlichen Vorteil des RPC-Stils zu nutzen, nämlich den Inhalt des SOAP-Bodys nach der angeforderten Operation des Dienstes zu benennen, ohne die Nachteile des RPC-Stils in Kauf nehmen zu müssen.

[47] Einige Nachteile des Dokumentstils sind:

- In der unverpackten Variante transportiert die SOAP-Nachricht den Namen der Operation des Dienstes *nicht*, wodurch die Verteilung der Nachrichten an die entsprechenden Stellen innerhalb der Applikation verkompliziert wird.
- Die Verpackungsvariante schafft eine zusätzliche Komplexitätsebene, insbesondere in der API. Wie das Beispiel `AmazonClientW` des E-Commerce-Dienstes von Amazon zeigt, kann das Schreiben eines Clients für einen Dienst in der Verpackungsvariante des Dokumentstils eine Herausforderung sein.
- Die Verpackungsvariante unterstützt keine überladenen Operationen, da das *Verpackungselement* direkt im SOAP-Body den Namen der Operation haben muß. Effektiv kann zu jedem Wert eines `name`-Attributes eines `<element>`-Elementes nur eine Operation existieren.

2.3.4 Ein asynchroner Client

[48] Wie unter den Nachteilen des RPC-Stils im vorigen Unterabschnitt bereits beschrieben, blockiert der ursprüngliche Client `ch01.ts.TimeClient` bei Aufrufen an den *`ch01.ts.TimeServer`*-Dienst. Beispielsweise blockiert der Aufruf

```
port.getTimeAsString()
```

den Client solange, bis entweder der Dienst antwortet oder eine Ausnahme hervorgerufen wird. Der Aufruf der `getTimeAsString()`-Methode wird daher als *blockierender* (*synchroner*) Aufruf bezeichnet. Java Webservices unterstützen auch *nichtblockierende* (*asynchrone*) Aufrufe beziehungsweise Clients.

[49] Das folgende Beispiel zeigt einen Client, der einen asynchronen Aufruf einer Operation des E-Commerce-Dienstes veranlaßt:

Beispiel 2.15: Ein asynchroner Client für den E-Commerce-Dienst von Amazon.

```
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import awsClient3.AWSECommerceService;
import awsClient3.AWSECommerceServicePortType;
import awsClient3.ItemSearchRequest;
import awsClient3.ItemSearchResponse;
import awsClient3.ItemSearch;
import awsClient3.Items;
```

```
import awsClient3.Item;

import java.util.List;
import java.util.concurrent.ExecutionException;

class AmazonAsyncClient {

    public static void main(String[] args) {

        // Usage
        if (args.length != 1) {
            System.err.println("Usage: java AmazonAsyncClient <access key>");
            return;
        }

        final String access_key = args[0];

        // Create service and get portType reference.
        AWSECommerceService service = new AWSECommerceService();
        AWSECommerceServicePortType port = service.getAWSECommerceServicePort();

        // Create request.
        ItemSearchRequest request = new ItemSearchRequest();

        // Add details to request.
        request.setSearchIndex("Books");
        request.setKeywords("quantum gravity");
        ItemSearch item_search= new ItemSearch();
        item_search.setAWSAccessKeyId(access_key);
        item_search.getRequest().add(request);

        port.itemSearchAsync(item_search, new MyHandler());

        // In this case, just sleep to give the search process time.
        // In a production application, other useful tasks could be
        // performed and the application could run indefinitely.
        try {
            Thread.sleep(400);
        }
        catch(InterruptedException e) { System.err.println(e); }
    }

    // The handler class implements handleResponse, which executes
    // if and when there's a response.
    static class MyHandler implements AsyncHandler<ItemSearchResponse> {
        public void handleResponse(Response<ItemSearchResponse> future) {
            try {
                ItemSearchResponse response = future.get();
                List<Items> item_list = response.getItems();
                for (Items next : item_list)
                    for (Item item : next.getItem())
                        System.out.println(item.getItemAttributes().getTitle());
            }
            catch(InterruptedException e) { System.err.println(e); }
            catch(ExecutionException e) { System.err.println(e); }
        }
    }

}
```

Der nichtblockierende Client des E-Commerce-Dienstes basiert auf `wsimport`-Artefakten. Auch diesesmal sind applikationsspezifische Bindungseinstellungen im Spiel:

```
<jaxws:bindings
```

```
        wsdlLocation=
          "http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl"
        xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
      <jaxws:enableWrapperStyle>false</jaxws:enableWrapperStyle>
      <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
    </jaxws:bindings>
```

Das Attribut `enableAsyncMapping` wird mit `true` bewertet.

[50] Es gibt verschiedene Möglichkeiten den nichtblockierenden Methodenaufruf zu veranlassen. Die obige Version der `itemSearchAsync()`-Methode erwartet zwei Argumente: Das erste Argument ist vom Typ `ItemSearch` und das zweite vom Typ *`javax.xml.ws.AsyncHandler`*. Das Interface *`AsyncHandler`* deklariert eine einzige Methode namens `handleResponse()`. Der Aufruf lautet:

```
port.itemSearchAsync(item_search, new MyHandler());
```

Sendet der E-Commerce-Dienst eine Antwort, so wird die `handleResponse()`-Methode der statischen inneren Klasse `MyHandler` von einem separaten Thread verarbeitet und gibt die Bücherliste aus.

[51] Das Interface *`awsClient3.AWSECommerceServicePortType`* (`wsimport`-Artefakt) deklariert noch eine einargumentige Version der `itemSearchAsync()`-Methode. Das Argument ist wiederum vom Typ `ItemSearch`. Auch bei dieser Version kehrt der Aufruf zurück, wenn der E-Commerce-Dienst eine Antwort sendet, die anschließend wie bei den E-Commerce-Clients in den Unterabschnitten 2.3.1 und 2.3.2 verarbeitet wird. Eine Applikation könnte einen separaten Thread starten, um die folgenden Anweisungen zu verarbeiten:

```
try {
    ItemSearchResponse response = res.get();
    List<Items> item_list = response.getItems();
    for (Items next : item_list)
        for (Item item : next.getItem())
            System.out.println(item.getItemAttributes().getTitle());
}
catch(InterruptedException e) { System.err.println(e); }
catch(ExecutionException e) { System.err.println(e); }
```

Java Webservices sind flexibel in der Unterstützung sowohl nichtblockierender als auch der voreingestellten blockierenden Clients. Letztendlich bestimmt die Logik innerhalb der Applikation, welcher Clienttyp paßt.

2.4 Das wsgen-Kommando und die JAX-B-Artefakte

[52] Jeder Dienst im Dokumentstil, gleichgültig ob in der verpackten oder der unverpackten Variante, benötigt die Artefakte, welche das `wsgen`-Kommando erzeugt. Es ist an der Zeit, dieses Kommando unter die Lupe zu nehmen. Ein einfaches Experiment verdeutlicht die Funktion des `wsgen`-Kommandos. Zunächst muß die Zeile:

```
@SOAPBinding(style = Style.RPC)
```

im SEI *`ch01.ts.TimeServer`* auskommentiert oder entfernt werden. Ohne diese Annotation erhält der Webservice den Dokumentstil anstelle des RPC-Stils. Der Versuch, den Dienst nach dem Neuübersetzen des geänderten SEIs mit dem folgenden Kommando in Betrieb zu nehmen:

```
% java ch01.ts.TimeServerPublisher
```

endet mit der Fehlermeldung:

```
Exception in thread "main" com.sun.xml.internal.ws.model.RuntimeModelerException:
runtime modeler error: Wrapper class ch01.ts.jaxws.GetTimeAsString is not found.
```

Die Fehlermeldung ist undeutlich, da sie nur das unmittelbare Problem angibt, nicht aber die eigentliche Ursache. Das unmittelbare Problem besteht darin, daß der `Endpoint`-Publisher `ch01.ts.TimeServerPublisher` die Klasse `ch01.ts.jaxws.GetTimeAsString` nicht finden kann. Das Package `ch01.ts.jaxws`, welches diese Klasse enthält, ist zu diesem Zeitpunkt noch nicht vorhanden. Der `TimeServerPublisher` kann das WSDL-Dokument nicht generieren, da hierfür Java-Klassen wie `GetTimeAsString` benötigt werden. Das `wsgen`-Kommando erzeugt die zum Generieren des WSDL-Dokumentes erforderlichen Klassen, die sogenannten *`wsgen`-Artefakte*. Der Aufruf

```
% wsgen -keep -cp . ch01.ts.TimeServerImpl
```

liefert die benötigten Artefakte und, falls erforderlich, auch das Package `ch01.ts.jaxws`, welches diese Artefakte beinhaltet. Das *`TimeServer`*-Beispiel umfaßt insgesamt vier Nachrichten, nämlich die Anfrage-/Antwortnachrichten für die Operationen `getTimeAsString()` beziehungsweise `getTimeAsElapsed()`. Das `wsgen`-Kommando generiert eine Java-Klasse, also einen Java-Datentyp, für jede dieser vier Nachrichten. Der `TimeServerPublisher` verwendet diese Java-Klassen, um das WSDL-Dokument eines Diensts im Dokumentstil zu erzeugen. Jeder Java-Datentyp ist also an einen im XML-Schema definierten Typ gebunden, der als Typ einer der vier Nachrichten des Dienstes verwendet wird. Anders ausgedrückt, hat ein Webservice im Dokumentstil *typisierte* Nachrichten. Die `wsgen`-Artefakte sind diejenigen Java-Datentypen, von denen die im XML-Schema definierten Typen für die Nachrichten abgeleitet werden.

[53] Ein SOAP-basierter Webservice, gleichgültig ob im Dokument- oder RPC-Stil, sollte interoperabel sein. Ein in *einer* Sprache geschriebener Client sollte also in der Lage sein, nahtlos mit einem Dienst zusammenzuarbeiten, der in einer *anderen* Sprache geschrieben ist, auch wenn Unterschiede hinsichtlich der Datentypen der beteiligen Sprachen bestehen. Ein gemeinsames Typsystem wie XML-Schema ist daher der Schlüssel zur Interoperabilität. Der Dokumentstil erweitert das Konzept der Typisierung auf die Nachrichten des Dienstes. Typisierte Nachrichten wiederum, erfordern die explizite Bindung von Java- und XML-Schematypen.

[54] Ein Dienst im Dokumentstil, gleichsam in der verpackten oder unverpackten Variante, hat stets ein WSDL-Dokument dessen `<types>`-Abschnitt ein XML-Schema beinhaltet. Die Wortwahl „beinhaltet“ ist bewußt unscharf. Der `<types>`-Abschnitt kann ein XML-Schema auch importieren oder auf ein XML-Schema verweisen. Die Typen im XML-Schema eines WSDL-Dokumentes sind an die Typen in der Sprache des Dienstes beziehungsweise Clients gebunden, zum Beispiel Java. Das `wsgen`-Kommando, bereits in Unterabschnitt 2.2.1 eingeführt, um den *`ch01.ts.TimeServer`*-Dienst vom *RPC-Stil* in den Dokumentstil zu ändern, generiert Java-Typen, die an Typen in einem XML-Schema gebunden sind. Hinter den Kulissen verwendet `wsgen` die Packages der Java Architecture for XML Binding (JAX-B). Kurz, JAX-B unterstützt die Konvertierung zwischen Java- und im XML-Typen.

2.4.1 Ein Beispiel für Bindungen über JAX-B

[55] Bevor wir uns den `wsgen`-Artefakten zuwenden, wollen wir uns ein Beispiel für JAX-B ansehen, um ein Gefühl für die Funktionsweise des `wsgen`-Kommandos zu bekommen. Das folgende Beispiel zeigt die selbstgeschriebene Java-Klasse `Person`, das nächste Beispiel die ebenfalls selbstgeschriebene Java-Klasse `Skier`. Jede der beiden Klassen trägt eine einzelne Annotation, die mit der Bindung zwischen Java und XML zu tun hat. Die Klasse `Person` trägt die Annotation `@XmlType`, während `Skier` mit *`@XmlRootElement`* annotiert ist. Auf der Designebene ist ein Skiläufer (`Skier`) eine Person (`Person`), so daß die Klasse `Skier` ein `Person`-Feld besitzt:

Beispiel 2.16: Die Klasse Person des JAX-B-Beispiels.

```
import javax.xml.bind.annotation.XmlType;

@XmlType
public class Person  {

    // fields
    private String name;
    private int    age;
    private String gender;

    // constructors
    public Person() { }

    public Person(String name, int age, String gender){

        setName(name);
        setAge(age);
        setGender(gender);

    }

    // properties: name, age, gender
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age;  }
    public void setAge(int age) { this.age = age; }

    public String getGender() { return gender; }
    public void setGender(String gender) { this.gender = gender; }

}
```

Beispiel 2.17: Die Klasse Person des JAX-B-Beispiels.

```
import javax.xml.bind.annotation.XmlRootElement;
import java.util.Collection;

@XmlRootElement
public class Skier  {

    // fields
    private Person person;
    private String national_team;
    private Collection major_achievements;

    // constructors
    public Skier() { }
    public Skier (Person person,
                  String national_team,
                  Collection<String> major_achievements) {

        setPerson(person);
        setNationalTeam(national_team);
        setMajorAchievements(major_achievements);

    }

    // properties
    public Person getPerson() { return person; }
    public void setPerson (Person person) { this.person = person; }

    public String getNationalTeam() { return national_team; }
    public void setNationalTeam(String national_team) {
        this.national_team = national_team;
```

```
    }

    public Collection getMajorAchievements() { return major_achievements; }
    public void setMajorAchievements(Collection major_achievements) {
        this.major_achievements = major_achievements;
    }

}
```

[56] Die Annotationen *@XmlType* und *@XmlRootElement* steuern das sogenannte Marshalling von `Person`- beziehungsweise `Skier`-Objekten. *Marshalling* ist der Vorgang, durch den ein im Arbeitsspeicher befindliches Objekt (zum Beispiel vom Typ `Skier`) in ein XML-Dokument umgewandelt wird. Die umgewandelte Version kann beispielsweise über ein Netzwerk hinweg versendet und auf der Empfängerseite durch *Unmarshalling* wieder rekonstruiert werden kann. Im Allgemeinen ist das Marshalling/Unmarshalling-Konzept sehr ähnlich, wenn nicht gar identisch mit dem Konzept der *Serialisierung*/*Deserialisierung*. Ich verwende die Begriffe Marshalling/Unmarshalling und Serialisierung/Deserialisierung synonym. JAX-B unterstützt das Marshalling im Speicher befindlicher Objekte in XML-Dokumente und das Unmarshalling von XML-Dokumenten in Objekte im Arbeitspeicher.

[57] Die Annotation `@XmlType`, in diesem Beispiel bei der Klasse `Person`, gibt an, daß JAX-B aus dem Java-Typ eine Typdefinition in XML-Schemanotation erzeugen soll. Die Annotation `@XmlRootElement`, hier bei der Klasse `Skier`, gibt an, daß JAX-B aus dem Java-Typ das *Wurzel- oder auch Dokumentelement* erzeugen soll (das äußerste Element des XML-Dokumentes). Dementsprechend repräsentiert das Wurzelelement des generierten XML-Dokumentes in diesem Beispiel einen Skiläufer und besitzt ein Unterelement, welches eine Person darstellt.

[58] Die folgende Hilfsklasse `Marshal` zeigt Marshalling und Unmarshalling:

Beispiel 2.18: Marshalling-Hilfsprogramm für das JAX-B-Beispiel.

```
import java.io.File;
import java.io.OutputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.IOException;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.JAXBException;
import java.util.List;
import java.util.ArrayList;

class Marshal {

    private static final String file_name = "bd.mar";

    public static void main(String[] args) {

        new Marshal().run_example();

    }

    private void run_example() {

        try {

            JAXBContext ctx = JAXBContext.newInstance(Skier.class);
            Marshaller m = ctx.createMarshaller();
            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

            // Marshal a Skier object: 1st to stdout, 2nd to file
```

```
            Skier skier = createSkier();
            m.marshal(skier, System.out);

            FileOutputStream out = new FileOutputStream(file_name);
            m.marshal(skier, out);
            out.close();

            // Unmarshal as proof of concept
            Unmarshaller u = ctx.createUnmarshaller();
            Skier bd_clone = (Skier) u.unmarshal(new File(file_name));
            System.out.println();
            m.marshal(bd_clone, System.out);

        }

        catch(JAXBException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }

    }

    private Skier createSkier() {

        Person bd = new Person("Bjoern Daehlie", 41, "Male");
        List<String> list = new ArrayList<String>();
        list.add("12 Olympic Medals");
        list.add("9 World Championships");
        list.add("Winningest Winter Olympian");
        list.add("Greatest Nordic Skier");
        return new Skier(bd, "Norway", list);

    }

}
```

Die Hilfsklasse erzeugt ein `Skier`-Objekt mit einem gekapselten, das heißt von einem entsprechend typisierten Feld referenzierten `Person`-Objekt und bewertet die Eigenschaften beider Objekte. Eine wesentliche Eigenschaft des Marshallings besteht darin, daß der Zustand des serialisierten Objektes im XML-Dokument gespeichert wird. Der Zustand eines Objektes ist die Gesamtheit der Werte seiner dynamischen (nicht-statischen) Felder. Im Falle eines `Skier`-Objektes müssen beim Marshalling sowohl die Informationen über die wichtigsten sportlichen Leistungen des Skiläufers, als auch personenspezifische Informationen wie Name und Alter erfaßt werden. Die Annotation `@XmlRootElement` vor der Definition der Klasse `Skier` wirkt sich folgendermaßen auf das Marshalling aus: Das `Skier`-Objekt wird in ein XML-Dokument mit dem Wurzelelement `<skier>` umgewandelt. Die Annotation `@XmlType` vor der Definition der Klasse `Person` steuert das Marshalling so: Das Wurzelelement `<skier>` des zuvor angelegten XML-Dokumentes erhält ein `<person>`-Unterelement, welches wiederum Unterelemente für die Eigenschaften `name`, `age` und `gender` des serialisierten `Person`-Objektes hat.

[59] Der Marshallingmechanismus von JAX-B folgt per Voreinstellung den Standardnamenskonventionen von Java und JavaBeans. Beispielsweise wird der Klasse `Skier` das Element `<skier>` und der Klasse `Person` das Element `<person>` zugeordnet. Sowohl für das `Skier`- als auch für das gekapselte `Person`-Objekt werden die Abfragemethoden gemäß der JavaBeans-Konvention (zum Beispiel `getNationalTeam()` bei `Skier` und `getAge()` bei `Person`) aufgerufen, um die Elemente des XML-Dokumentes mit den entsprechenden Zustandsinformationen über den Skiläufer zu bewerten.

[60] Die Namenskonventionen für JavaBeans beim Marshalling und Unmarshalling können mit Hilfe von Annotationen überschrieben werden. Das folgende `wsgen`-Artefakt hat Abfrage- und Änderungsmethoden, welche die JavaBeans-Namenskonventionen *nicht* befolgen. Beachten Sie die hervorgehobene Annotation:

```
...
@XmlRootElement(name = "getTimeAsElapsedResponse", namespace = "http://ts.ch02/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getTimeAsElapsedResponse", namespace = "http://ts.ch02/")
public class GetTimeAsElapsedResponse {
    @XmlElement(name = "return", namespace = "")
    private long _return;
    public long get_return() { return this._return; }
    public void set_return(long _return) { this._return = _return; }
}
```

Die *@XmlAccessorType*-Annotation gibt an, daß das Feld mit dem Namen `_return` beim Marshalling/Unmarshalling anders behandelt wird, als eine Eigenschaft, mit *Abfrage*- und *Änderungsmethoden*, die den Namenskonventionen für JavaBeans gehorchen.

[61] Die voreingestellte Namenskonvention kann mit Hilfe von Annotationsattributen überschrieben werden. Wird die `@XmlRootElement`-Annotation der Klasse `Skier` folgendermaßen geändert

```
@XmlRootElement(name = "NordicSkier")
```

so lautet der Anfang des resultierenden XML-Dokumentes:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<NordicSkier>
```

Nachdem das `Skier`-Objekt erzeugt ist, serialisiert das Hilfsprogramm `Marshal` das Objekt in die Standardausgabe und für das anschließende Unmarshalling zugleich in eine lokale Datei namens *bd.mar*. Das folgende XML-Dokument ergibt sich per Marshalling für den legendären norwegischen Skiläufer Bjørn Dælie:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<skier>
  <majorAchievements
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xsi:type="xs:string">
    12 Olympic Medals
  </majorAchievements>
  <majorAchievements
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xsi:type="xs:string">
    9 World Championships
  </majorAchievements>
  <majorAchievements
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xsi:type="xs:string">
    Winningest Winter Olympian
  </majorAchievements>
  <majorAchievements
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xsi:type="xs:string">
    Greatest Nordic Skier
  </majorAchievements>
  <nationalTeam>Norway</nationalTeam>
  <person>
    <age>41</age>
```

```
      <gender>Male</gender>
      <name>Bjoern Daehlie</name>
    </person>
  </skier>
```

Das Unmarshalling erzeugt aus dem XML-Dokument ein `Skier`-Objekt mit gekapseltem `Person`-Objekt. Der Unmarshallingmechanismus von JAX-B setzt voraus, daß jede Klasse einen öffentlichen argumentlosen Konstruktor hat, der während der Objekterzeugung aufgerufen wird. Nachdem das Objekt erzeugt ist, werden die entsprechenden Änderungsmethoden aufgerufen (zum Beispiel `setNationalTeam()` bei `Skier` und `setAge()` bei `Person`), um den Zustand des serialisierten `Skier`-Objektes wiederherzustellen. Marshalling- und Unmarshalling sind auf der Quelltextebene bemerkenswert klar und einfach.

2.4.2 Marshalling und wsgen-Artefakte

[62] Das Kommando `wsgen` und Marshalling können nun miteinander verknüpft werden. Die Änderung des ursprünglichen *`ch01.ts.TimeServer`*-Dienstes (RPC-Stil) in den *`ch02.tsd.TimeServer`* (Dokumentstil) in Unterabschnitt 2.2.1 ging in drei Schritten vor sich. Der erste Schritt bestand darin, für die Version im Dokumentstil ein neues Verzeichnis *ch02/tsd* anzulegen, den Inhalt des ursprünglichen Verzeichnisses *ch01/ts* nach *ch02/tsd* zu kopieren und die `package`-Anweisungen der Dateien in *ch02/tsd* anzupassen. Der zweite Schritt bestand darin, die Annotation

```
@SOAPBinding(style = Style.RPC)
```

aus dem SEI *`ch02.tsd.TimeServer`* auszukommentieren. Im dritten Schritt wurde das Kommando `wsgen` im Arbeitsverzeichnis mit der Klasse `ch02.tsd.TimeServerImpl` als Argument aufgerufen:

```
% wsgen -keep -cp . ch02.tsd.TimeServerImpl
```

Der Aufruf des `wsgen`-Kommandos hinterließ zwei *.java* und zwei *.class* Dateien im automatisch angelegten Unterverzeichnis *ch02/tsd/jaxws*. Das folgende Beispiel zeigt eines dieser generierten `wsgen`-Artefakte, genauer die Klasse `GetTimeAsElapsedResponse`, wobei die Kommentare entfernt wurden:

Beispiel 2.19: Ein wsgen-Artefakt (ohne Kommentare).

```
package ch02.tsd.jaxws;

import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "getTimeAsElapsedResponse", namespace = "http://tsd.ch02/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getTimeAsElapsedResponse", namespace = "http://tsd.ch02/")
public class GetTimeAsElapsedResponse {

    @XmlElement(name = "return", namespace = "")
    private long _return;
    public long get_return() { return this._return; }
    public void set_return(long _return) { this._return = _return; }

}
```

Die Annotation `@XmlType` bei diese Klasse ist von besonderem Interesse. Das `name`-Attribut hat den Wert `getTimeAsElapsedResponse`, also den Namen der vom Webservice an seinen Client zurückgesendeten SOAP-Nachricht beim Aufruf der Operation `getTimeAsElapsed()`. Die Annotation

@XmlType bedeutet, daß derartige SOAP-Antwortnachrichten *typisiert* sind, das heißt einem XML-Schema gehorchen. Bei einem Webservice im Dokumentstil sind die SOAP-Nachrichten typisiert.

[63] Das nächste Beispiel zeigt eine modifizierte Version der Klasse `Marshal` namens `MarshalGTER`. Der geänderte Name soll andeuten, daß die überarbeitete Klasse mit einem `wsgen`-Artefakt namens `GetTimeAsElapsedResponse` arbeitet:

Beispiel 2.20: Veranschaulichung der Verbindung zwischen wsgen und JAX-B.

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.JAXBException;

import ch02.tsd.jaxws.GetTimeAsElapsedResponse;

class MarshalGTER {

    private static final String file_name = "gter.mar";

    public static void main(String[] args) {
        new MarshalGTER().run_example();
    }

    private void run_example() {

        try {

            JAXBContext ctx =
                JAXBContext.newInstance(GetTimeAsElapsedResponse.class);
            Marshaller m = ctx.createMarshaller();
            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

            GetTimeAsElapsedResponse tr = new GetTimeAsElapsedResponse();
            tr.set_return(new java.util.Date().getTime());

            m.marshal(tr, System.out);

        }

        catch(JAXBException e) { System.err.println(e); }

    }

}
```

Ein Aufruf bei mir ergab das folgende XML-Dokument:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:getTimeAsElapsedResponse xmlns:ns2="http://tsd.ch02/">
    <return>1209174518855</return>
</ns2:getTimeAsElapsedResponse>
```

Hier, zum Vergleich, die Antwortnachricht des *ch02.tsd.TimeServer*-Dienstes (Dokumentstil):

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://tsd.ch02/">
  <soapenv:Body>
    <ns1:getTimeAsElapsedResponse>
      <return>1209181713849</return>
    </ns1:getTimeAsElapsedResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Es gibt einige nebensächliche Unterschiede zwischen dem SOAP-Body der Nachricht und der Ausgabe der Klasse `MarshalGTER`. Beispielsweise lautet der Namensraumpräfix in der SOAP-Nachricht `ns1`, im obigen XML-Dokument dagegen `ns2`. Beachten Sie aber, daß der wichtige Namensraum-URI in beiden Fällen identisch ist: `http://tsd.ch02/`. Bei der SOAP-Nachricht ist der Namensraumpräfix im `<Envelope>`-Element anstelle des `<getTimeAsElapsedResponse>`-Elementes definiert. Die `wsgen`-Artefakte, etwa die Klasse `GetTimeAsElapsedResponse`, sind die annotierten Typen, welche die unterliegenden SOAP-Bibliotheken verwenden, um Java-Objekte in XML-Dokumente/Elemente zu serialisieren oder ein solches XML-Dokument per Unmarshalling in Java-Objekte entsprechenden Typs zu deserialisieren.

2.4.3 Bindung der primitiven Typen von Java an die vordefinierten Typen von XML-Schema

[64] Die *primitiven Typen* von Java wie *`int`* und *`byte`* werden an äquivalente, in der Sprache XML-Schema definierte Typen „gebunden“ (auf diese abgebildet/miteinander verknüpft), hier *`xsd:int`* beziehungsweise *`xsd:byte`*. Der Java-Typ *`java.lang.String`* wird an *`xsd:string`*, und *`java.util.Calendar`* an *`xsd:date`*, *`xsd:time`* sowie *`xsd:dateTime`* gebunden. Der vordefinierte Typ *`xsd:decimal`* ist an *`java.math.BigDecimal`* gebunden. Nicht alle Bindungen sind offensichtlich und führen zum gleichnamigen Java-*Typ*. Beispielsweise paßt `int` zu mehreren in XML-Schema vordefinierten Typen, etwa `xsd:int` und `xsd:unsignedShort`. Die scheinbare Unstimmigkeit zwischen dem Java-Typ `int` und `xsd:unsignedShort` hat die folgende Ursache: Java verfügt technisch über keinen vorzeichenlosen ganzzahligen Typ. (Sie können natürlich einwenden, daß der *`char`-Typ* von Java eigentlich ein vorzeichenloser 16-Bit langer, ganzzahliger Typ ist.) Der Maximalwert des 16-Bit langen Java-Typs `short` ist 32767, während der Maximalwert von `namespace:unsignedShort` 65535 lautet, also im Wertebereich des 32-Bit langen Java-Typs `int` liegt.

[65] *JAX-B* gestattet die Bindung beliebig ausgestalteter Java-Typen an XML-Typen (die Klasse `Skier` vermittelt einen Eindruck dieser Möglichkeiten). Objekte derartiger Klassen können per Marshalling in XML-Dokumente/Elemente umgewandelt werden, welche sich schließlich per Unmarshalling wieder in Java-Typen oder Typen einer anderen Programmiersprache konvertieren lassen.

[66] Nun kann die Interaktion zwischen dem `wsgen`-Kommando und den JAX-B-Bibliotheken bei Java-Webservices zusammengefaßt werden: Das WSDL-Dokument eines Java-Webservice' im Dokumentstil (im Gegensatz zum RPC-Stil) hat einen nicht-leeren `<types>`-Abschnitt. Dieser Abschnitt definiert mit den Ausdrucksmitteln der Sprache XML-Schema die für diesen Dienst erforderlichen XML-Typen. Das `wsgen`-Kommando generiert aus der SIB Java-Klassen, die den im XML-Schema definierten Typen entsprechen. Diese `wsgen`-Artefakte stehen den unterliegenden Bibliotheken für Java-Webservices zur Verfügung (insbesondere den JAX-B-Bibliotheken), um im Arbeitsspeicher befindliche Objekte von Java-Klassen per Marshalling in XML-Dokumente/Elemente der im XML-Schema definierten Typen zu konvertieren. In der umgekehrten Richtung werden XML-Dokumente/Elemente per Unmarshalling in Objekte im Speicher umgewandelt, wobei die Objekte von Java-Klassen oder vergleichbaren Typen in einer anderen Programmiersprache stammen können. Das `wsgen`-Kommando erzeugt also die Artefakte, welche die Interoperabilität Java-basierter Webservices gewährleisten. Die JAX-B-Bibliotheken liefern die Unterstützung für die Verknüpfung von Typdefinitionen zwischen Java und XML-Schema. Das `wsgen`-Kommando kann zumeist benutzt werden, ohne sich um die erzeugten Artefakte kümmern zu müssen. JAX-B bleibt zumeist unsichtbare Infrastruktur.

Automatischer wsgen-Aufruf bei Metro

Das SEI eines Dienstes enthält alle benötigten Informationen zum Erzeugen der `wsgen`-Artefakte. Das SEI kennzeichnet die Operationen des Dienstes mit der Annotation `@WebMethod`. Aus den deklarierten Methoden gehen die Argument- und Rückgabetypen hervor.

Beim aktuellen *Metro*-Release generiert der `Endpoint`-Publisher die `wsgen`-Artefakte automatisch, sofern es der Programmierer nicht selbst tut. Liegt der Dienst `test.WS1Pub` in übersetzter Form vor, so nimmt das Kommando

```
% java -cp .:$METRO_HOME/lib/jaxws-rt.jar test.WS1Pub
```

diesen Dienst in Betrieb, obwohl zuvor keine `wsgen`-Artefakte erzeugt wurden. (Bei Windows ersetzen Sie `$METRO_HOME` durch `%METRO_HOME%`.) Die Ausgabe lautet:

```
com.sun.xml.ws.model.RuntimeModeler getRequestWrapperClass
INFO: Dynamically creating request wrapper Class test.jaxws.Op
com.sun.xml.ws.model.WrapperBeanGenerator createBeanImage
INFO:
@XmlRootElement(name=op, namespace=http://test/)
@XmlType(name=op, namespace=http://test/)
public class test.jaxws.Op {
    @XmlRootElement(name=arg0, namespace=)
    public I arg0
}
com.sun.xml.ws.model.RuntimeModeler getResponseWrapperClass
INFO: Dynamically creating response wrapper bean Class
      test.jaxws.OpResponse
com.sun.xml.ws.model.WrapperBeanGenerator createBeanImage
INFO:
@XmlRootElement(name=opResponse, namespace=http://test/)
@XmlType(name=opResponse, namespace=http://test/)
public class test.jaxws.OpResponse {
    @XmlRootElement(name=return, namespace=)
    public I _return
}
```

Diese praktische Eigenschaft des Metro-Release wird rechtzeitig ihren Weg in die Java Standard Edition finden, so daß sich das explizite Aufrufen des `wsgen`-Kommandos bei Diensten im Dokumentstil vermeiden läßt. Es ist dennoch nützlich, zu verstehen, welche Aufgabe die per `wsgen`-Kommando erzeugten JAX-B-Artefakte bei Java-basierten Webservices haben.

2.4.4 Generieren eines WSDL-Dokumentes per wsgen-Kommando

[67] Das Kommando `wsgen` gestattet darüberhinaus das Erzeugen des WSDL-Dokumentes. Beispielsweise generiert der Aufruf

```
% wsgen -cp ".." -wsdl ch01.ts.TimeServerImpl
```

ein WSDL-Dokument für den ursprünglichen *`TimeServer`*-Dienst aus dem ersten Kapitel. Der *`TimeServer`*-Dienst hat den RPC-Stil und nicht den Dokumentstil, wie das WSDL-Dokument ausweist. Zwischen dem per `wsgen` erzeugten und dem zur Laufzeit nach Inbetriebnahme des Dienstes erhaltenen WSDL-Dokument besteht ein wesentlicher Unterschied: Das per `wsgen` generierte WSDL-Dokument enthält den Endpunkt des Dienstes *nicht*, da die URL von der konkreten Inbetriebnahme des Dienstes abhängt. Der relevante Abschnitt des per `wsgen` erzeugten WSDL-Dokumentes ist:

```
<service name="TimeServerImplService">
  <port name="TimeServerImplPort" binding="tns:TimeServerImplPortBinding">
```

```
      <soap:address location="REPLACE_WITH_ACTUAL_URL" />
    </port>
  </service>
```

Abgesehen von diesem Unterschied beinhalten beide WSDL-Dokumente denselben Dienstkontrakt.

[68] Per `wsgen`-Kommando direkt erzeugte WSDL-Dokumente sind bei den späteren Beispielen zum Thema Sicherheit nützlich. Wird ein Webservice abgesichert, so auch sein WSDL-Dokument. Der erste Schritt zum Schreiben eines `wsimport`-gestützten Clients, um einen Dienst zu testen, ist der Zugriff auf das WSDL-Dokument. Das `wsgen`-Kommando ist eine einfache Möglichkeit, das WSDL-Dokument zu beschaffen.

2.5 Ergänzende WSDL-Aspekte

[69] Dieser Abschnitt rundet das Kapitel mit einem Blick auf auf zwei weitere Aspekte von WSDL-Dokumenten ab. Der erste betrifft die Frage, ob zuerst der Quelltext eines Dienstes, insbesondere das SEI und die SIB, oder aber das WSDL-Dokument geschrieben werden sollte. Soll, mit anderen Worten, das WSDL-Dokument automatisch aus der Implementierung des Dienstes generiert werden oder soll es entworfen und geschrieben werden, bevor der Quelltext des Dienstes entwickelt wird? Diese Frage ist als die Debatte „Code First versus Contract First“ bekannt geworden. Der andere Aspekt betrifft die begrenzte Menge an Informationen, die ein WSDL-Dokument potentiellen Clients des Dienstes zur Verfügung stellt.

2.5.1 Die Ansätze „Code First“ und „Contract First“ im Vergleich

[70] Der Gegenstand dieser Debatte läßt sich in Form einer Frage ausdrücken: Soll der Quelltext des Dienstes verwendet werden, um das WSDL-Dokument automatisch zu generieren oder soll ein unabhängig entworfenes und geschriebenes WSDL-Dokument verwendet werden, um die Entwicklung des Quelltextes des Dienstes zu leiten? Alle bisherigen Beispiele folgen aufgrund seines offensichtlichen Vorteils dem „Code First“-Ansatz, nämlich seiner Einfachheit. Dennoch birgt dieses Vorgehensweise einige offenkundige Nachteile und sogar Gefahren, zum Beispiel:

- Ändert sich der Dienst, so auch sein WSDL-Dokument. Das WSDL-Dokument verliert ein Stück seiner Attraktivität hinsichtlich des Erzeugens clientseitiger Artefakte, da der Vorgang immer wieder erneut durchgeführt werden muß. Eines der wichtigsten Prinzipien der Softwareentwicklung lautet, daß eine einmal veröffentlichte Schnittstelle als unveränderlich betrachtet werden sollte, damit Implementierungen, die sich auf diese Schnittstelle beziehen, nicht umgeschrieben werden müssen. Der „Code First“-Ansatz kann dieses Prinzip gefährden.

- Der „Code First“-Ansatz führt in der Regel zu einem Dienstkontrakt, der, falls überhaupt, nur wenige Vorkehrungen zur Behandlung komplizierter aber häufiger Probleme bei verteilten Systemen berücksichtigt, zum Beispiel einen teilweisen Ausfall des Dienstes. Das liegt daran, daß der Dienst wie eine eigenständige Applikation entwickelt wird, nicht aber als Komponente einer verteilten Anwendung mit Clients auf anderen Hosts.

- Ist die Implementierung des Dienstes kompliziert oder gar chaotisch, so schlagen sich diese Eigenschaften auch im WSDL-Dokument nieder. Das Verständnis des WSDL-Dokumentes und das Erzeugen clientseitiger Artefakte kann dadurch erschwert werden. Kurz, der „Code First“-Ansatz ist offensichtlich nicht clientorientiert.

- Der „Code First“-Ansatz widerspricht dem Leitmotiv der *Sprachtransparenz* SOAP-basierter Webservices. Wird das WSDL-Dokument zuerst entwickelt, so bleibt die Implementierungssprache dagegen offen.

Die Liste ließe sich noch erweitern. Mein Ziel ist indes nicht, die Auseinandersetzung zu befeuern, sondern warnend darauf hinzuweisen, daß sie nicht ungerechtfertigt ist. Das Hauptproblem der „Contract First“-Befürworter ist die reale Praxis der Softwareentwicklung. Ein Programmierer unter dem Druck von Terminen und Erwartungen an eine entwickelte Applikation hat wahrscheinlich keine Zeit, geschweige denn den Drang, die Feinheiten der WSDL zu bezwingen und ein WSDL-Dokument zu schreiben, das die Entwicklung des Dienstes in geeigneter Weise anleitet.

[71] Selbst das generierte WSDL-Dokument des einfachen *`ch02.tsd.TimeServer`*-Dienstes (Dokumentstil) kann noch verbessert werden. Die folgenden Zeilen zeigen einen Auszug aus dessen XML-Schema:

```
<xs:complexType name="getTimeAsStringResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Beachten Sie, daß das Element für den Rückgabewert die Mindestanzahl von Vorkommen auf 0 anstelle von 1 setzt. Das ist sinnvoll, da ein Java-basierter Webservice `null` als gültigen Wert des Typs `String` zurückgegeben kann. Das WSDL-Dokument läßt sich durch die Forderung straffen, daß nur von `null` verschiedene `String`-Referenzen zurückgegeben werden dürfen, in dem die 0 durch eine 1 ersetzt wird. (Alternativ kann das Attribut `minOccurs="0"` auch fortgelassen werden, da für `minOccurs` und `maxOccurs` jeweils der Wert 1 voreingestellt ist.)

2.5.2 Ein Beispiel für „Contract First“

[72] Das `wsimport`-Kommando läßt sich zur Unterstützung des „Contract First“-Ansatzes nutzen. Das folgende Beispiel zeigt einen in C# geschriebenen Webservice zur Umwandlung von Temperaturangaben zwischen den Einheiten Celsius (C) und Fahrenheit (F):

Beispiel 2.21: Ein SOAP-basierter Dienst in C#.

```
using System;
using System.Linq;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Linq;

[WebService(Namespace = "http://tempConvertURI.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService {

    public Service () { }

    [WebMethod]
    public double c2f(double t) { return 32.0 + (t * 9.0 / 5.0); }

    [WebMethod]
    public double f2c(double t) { return (5.0 / 9.0) * (t - 32.0); }

}
```

[73] Das nächste Beispiel zeigt das WSDL-Dokument dieses Dienstes:

Beispiel 2.22: Das WSDL-Dokument des Dienstes in Beispiel 2.21.

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:tns="http://tempConvertURI.org/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    targetNamespace="http://tempConvertURI.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema
        elementFormDefault="qualified"
        targetNamespace="http://tempConvertURI.org/">
      <s:element name="c2f">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="t" type="s:double" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="c2fResponse">
        <s:complexType>
          <s:sequence>
            <s:element
                minOccurs="1" maxOccurs="1" name="c2fResult" type="s:double" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="f2c">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="t" type="s:double" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="f2cResponse">
        <s:complexType>
          <s:sequence>
            <s:element
                minOccurs="1" maxOccurs="1" name="f2cResult" type="s:double" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="c2fSoapIn">
    <wsdl:part name="parameters" element="tns:c2f" />
  </wsdl:message>
  <wsdl:message name="c2fSoapOut">
    <wsdl:part name="parameters" element="tns:c2fResponse" />
  </wsdl:message>
  <wsdl:message name="f2cSoapIn">
    <wsdl:part name="parameters" element="tns:f2c" />
```

```
</wsdl:message>
<wsdl:message name="f2cSoapOut">
  <wsdl:part name="parameters" element="tns:f2cResponse" />
</wsdl:message>
<wsdl:portType name="ServiceSoap">
  <wsdl:operation name="c2f">
    <wsdl:input message="tns:c2fSoapIn" />
    <wsdl:output message="tns:c2fSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="f2c">
    <wsdl:input message="tns:f2cSoapIn" />
    <wsdl:output message="tns:f2cSoapOut" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="ServiceSoap" type="tns:ServiceSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="c2f">
    <soap:operation
        soapAction="http://tempConvertURI.org/c2f" style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="f2c">
    <soap:operation
        soapAction="http://tempConvertURI.org/f2c" style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="ServiceSoap12" type="tns:ServiceSoap">
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="c2f">
    <soap12:operation
        soapAction="http://tempConvertURI.org/c2f" style="document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="f2c">
    <soap12:operation
        soapAction="http://tempConvertURI.org/f2c" style="document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
```

```
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Service">
    <wsdl:port name="ServiceSoap" binding="tns:ServiceSoap">
      <soap:address location="http://localhost:1443/TempConvert/Service.asmx" />
    </wsdl:port>
    <wsdl:port name="ServiceSoap12" binding="tns:ServiceSoap12">
      <soap12:address location="http://localhost:1443/TempConvert/Service.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Dieses WSDL-Dokument hat einige Eigenschaften, deren Betrachtung sich lohnt. Der hervorgehobene `<wsdl:service>`-Abschnitt enthält zwei `<wsdl:port>`-Elemente („Ports“), je eines für Version 1.1 beziehungsweise 1.2 des SOAP-Protokolls. Dennoch haben die `location`-Attribute beider `<wsdl:port>`-Elemente denselben Wert, das heißt beide Ports stellen dieselbe Implementierung zur Verfügung. Der `<wsdl:binding>`-Abschnitt (die Implementierung) liefert die Erklärung: Zwei `<wsdl:port>`-Elemente, die jeweils ein `<wsdl:portType>`- und ein `<wsdl:binding>`-Element referenzieren, haben verschiedene Möglichkeiten, um sich voneinander zu unterscheiden. Beispielsweise könnten sich (im obigen Fall nicht) die `<soap:binding>`/`<soap12:binding>`-Elemente durch die Werte ihrer `transport`-Attribute unterscheiden, etwa indem SOAP-Nachrichten einmal über HTTP und einmal über SMTP versendet werden. Die `<soap:binding>`/`<soap12:binding>`-Elemente können sich aber auch, wie in diesem Beispiel, durch die Version des SOAP-Protokolls unterscheiden. Das erste Element beschreibt eine Bindung bezüglich der SOAP-Version 1.1, das zweite eine Bindung bezüglich Version 1.2. Der Dienst in Beispiel 2.21 ist aber so einfach, daß sich keine Unterschiede infolge der verschiedenen SOAP-Versionen zeigen. Die beiden `<soap:binding>`/`<soap12:binding>`-Elemente und somit auch die beiden `<wsdl:port>`-Elemente sind in diesem Fall identisch.

[74] Das `wsimport`-Kommando liefert, auf das obige WSDL-Dokument angewendet, die gewohnten Artefakte zur Unterstützung zum Schreiben eines Clients:

```
% wsimport -p tcClient -extension \
    http://localhost:1443/TempConvert/Service.asmx?wsdl
```

Der Schalter *-extension* wird verwendet, da das WSDL-Dokument eine Bindungseinstellung bezüglich Version 1.2 des SOAP-Protokolls enthält. Mit den generierten Artefakten ergibt sich der folgende einfache Client:

```
import tcClient.Service;
import tcClient.ServiceSoap; // port

// A sample Java client against a C# web service
class ClientDotNet {

    public static void main(String[] args) {

        Service service = new Service();
        // There's also a getServiceSoap12 for the SOAP 1.2 binding
        ServiceSoap port = service.getServiceSoap();

        double temp = 98.7;
        System.out.println(port.c2F(temp)); // 209.65999450683594
        System.out.println(port.f2C(temp)); //  37.05555386013455

    }

}
```

Dieser Abschnitt soll aber demonstrieren, wie ein und dasselbe WSDL-Dokument genutzt werden kann, um das Schreiben einer Java-Version *des Dienstes selbst* zu unterstützen.

[75] Das WSDL-Dokument des Dienstes wurde zwar automatisch generiert, die Herkunft des WSDL-Dokumentes ist hier aber unerheblich. Das WSDL-Dokument könnte auch manuell beliebig angepaßt oder von Anfang an neu geschrieben werden. Das Wesentliche ist die Sprachtransparenz des WSDL-Dokumentes, so daß der darin beschriebene Dienst also auch in Java anstelle von C# geschrieben werden kann. Das WSDL-Dokument in diesem Beispiel stammt bewußt *nicht* von einem in Java geschriebenen Dienst.

[76] Einige Details dieses WSDL-Dokumentes sind für eine Java-Implementierung des darin beschriebenen Dienstes nicht sinnvoll. Beispielsweise endet der Wert des `location`-Attributes mit `Service.asmx`, also dem Dateinamen der C#-Implementierung. Dieses Detail könnte im WSDL-Dokument geändert werden. Ein anderes Beispiel ist der Port des Dienstes: Das `name`-Attribut des `<port>`-Elementes könnte statt `ServiceSoap` den geeigneteren Wert `TempConvert` bekommen. In diesem Beispiel bleibt das WSDL-Dokument aber unverändert. Die Änderungen werden stattdessen in den Quelltext der `wsimport`-Artefakte eingesetzt.

[77] Befindet sich das WSDL-Dokument in einer lokalen Datei namens *tempc.wsdl*, so erzeugt das Kommando:

```
% wsimport -keep -p  ch02.tc tempc.wsdl
```

die `wsimport`-Artefakte im Unterverzeichnis *ch02/tc*. Dabei ist das Java-Äquivalent des `<portType>`-Abschnitts von Interesse, nämlich die Datei *ServiceSoap.java*. Hier ein Ausschnitt:

```
@WebService(name = "ServiceSoap", targetNamespace = "http://tempConvertURI.org/")
public interface ServiceSoap {

    @WebMethod(operationName = "c2f", action = "http://tempConvertURI.org/c2f")
    @WebResult(name = "c2fResult", targetNamespace = "http://tempConvertURI.org/")
    @RequestWrapper(localName = "c2f",
      targetNamespace = "http://tempConvertURI.org/",
      className = "ch02.tc.C2F")
    @ResponseWrapper(localName = "c2fResponse",
      targetNamespace = "http://tempConvertURI.org/",
      className = "ch02.tc.C2FResponse")
    public double c2F(

        @WebParam(name = "t", targetNamespace = "http://tempConvertURI.org/")
        double t);
...
```

Die Datei *ServiceSoap.java* beinhaltet das SEI, läßt sich aber mühelos in die SIB umwandeln, indem das Schlüsselwort `interface` in `class` und die Methodendeklarationen (hier ist nur `c2F()` angegeben) in Methodendefinitionen geändert werden. Im optionalen dritten Schritt kann der Name `ServiceSoap` durch den passenderen Namen `TempConvert` ersetzt werden. Der Quelltext der SIB lautet nach den Änderungen am SEI:

```
package ch02.tc;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(name = "TempConvert", targetNamespace = "http://tempConvertURI.org/")
```

```
public class TempConvert {

    @WebMethod(operationName = "c2f", action = "http://tempConvertURI.org/c2f")
    @WebResult(name = "c2fResult", targetNamespace = "http://tempConvertURI.org/")
    @RequestWrapper(localName = "c2f",
      targetNamespace = "http://tempConvertURI.org/",
      className = "ch02.tc.C2F")
    @ResponseWrapper(localName = "c2fResponse",
      targetNamespace = "http://tempConvertURI.org/",
      className = "ch02.tc.C2FResponse")
    public double c2F(

        @WebParam(name = "t", targetNamespace = "http://tempConvertURI.org/")
        double t) { return 32.0 + (t * 9.0 / 5.0); }

    @WebMethod(operationName = "f2c", action = "http://tempConvertURI.org/f2c")
    @WebResult(name = "f2cResult", targetNamespace = "http://tempConvertURI.org/")
    @RequestWrapper(localName = "f2c",
      targetNamespace = "http://tempConvertURI.org/",
      className = "ch02.tc.F2C")
    @ResponseWrapper(localName = "f2cResponse",
      targetNamespace = "http://tempConvertURI.org/",
      className = "ch02.tc.F2CResponse")
    public double f2C(

        @WebParam(name = "t", targetNamespace = "http://tempConvertURI.org/")
        double t) { return (5.0 / 9.0) * (t - 32.0); }

}
```

Nach der Inbetriebnahme des Dienstes mit Hilfe der statischen `Endpoint`-Methode `publish()`, können die clientseitigen `wsimport`-Artefakte generiert werden:

```
% wsimport -keep -p clientTC http://localhost:5599/tc?wsdl
```

Damit läßt sich ein Java-Client schreiben:

```
import javax.xml.ws.Service;
import clientTC.TempConvertService;
import clientTC.TempConvert;

class ClientTC {

    public static void main(String args[]) throws Exception {

        TempConvertService service = new TempConvertService();
        TempConvert port = service.getTempConvertPort();
        double d1 = -40.1, d2 = -39.4;
        System.out.printf("f2C(%f) = %f\n", d1, port.f2C(d1));
        System.out.printf("c2F(%f) = %f\n", d2, port.c2F(d2));

   }

}
```

Die Ausgabe dieses Clients lautet:

```
f2C(-40.100000) = -40.055556
c2F(-39.400000) = -38.920000
```

Das Beispiel zeigt, daß die `wsimport`-Artefakte sowohl zur Entwicklung eines Clients für den Dienst als auch zum Schreiben des Dienstes selbst in Java verwendet werden können. Ist ein WSDL-Dokument verfügbar, so rückt eine Java-Implementierung des Dienstes in greifbare Nähe.

2.5.3 Ein Beispiel für „Code-First, Contract-Aware“

[78] Java-Webservices unterstützen auch einen „Code First, Contract Aware“-Ansatz. Java-Webservices begünstigen „Code First“ durch das einfache Erzeugen des WSDL-Dokumentes. Nach der Inbetriebnahme des Dienstes wird das WSDL-Dokument automatisch generiert und steht potentiellen Clients zur Verfügung. Darüberhinaus bietet Java dem Programmierer die Möglichkeit, bei kritischen Abschnitten mit Hilfe von Annotationen auf die Gestaltung des generierten WSDL-Dokumentes beziehungsweise der daraus erzeugten Artefakte einzuwirken. Das nächste Beispiel zeigt eine dritte Variante des `TimeServer`-Dienstes (`ch02.tsa.TimeServer`). Der Webservice ist diesesmal in einer einzigen Datei namens *TimeServer.java* implementiert und mit verschiedenen Annotationen versehen, um deren Auswirkungen auf das automatisch generierte WSDL-Dokument und die SOAP-Nachrichten beobachten zu können:

Beispiel 2.23: Eine Variante des TimeService-Dienstes nach dem „Code First, Contract Aware“-Ansatz.

```
package ch02.tsa;  // 'a' for 'annotation'

import java.util.Date;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.Oneway;
import javax.jws.WebParam;
import javax.jws.WebParam.Mode;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;
import javax.jws.soap.SOAPBinding.ParameterStyle;

@WebService(name            = "AnnotatedTimeServer",
            serviceName     = "RevisedTimeServer",
            targetNamespace = "http://ch02.tsa")
@SOAPBinding(style          = SOAPBinding.Style.DOCUMENT,
             use            = SOAPBinding.Use.LITERAL,
             parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
public class TimeServer {

    @WebMethod(operationName   = "time_string")
    @WebResult(name            = "ts_out",
               targetNamespace = "http://ch02.tsa")
    public String getTimeAsString(

        @WebParam(name            = "client_message",
                  targetNamespace = "http://ch02.tsa",
                  mode            = WebParam.Mode.IN)
        String msg) {
            return msg + " at " + new Date().toString();
    }

    @WebMethod (operationName = "time_elapsed")
    public long getTimeAsElapsed() { return new Date().getTime(); }

    @WebMethod
    @Oneway
    public void acceptInput(String msg) { System.out.println(msg); }

}
```

Wir beginnen mit der Annotation `@WebService`, die in dieser Version des *`TimeServer`*-Beispiels Attribute hat. Nach der Inbetriebnahme ist der Dienst an Port 9876 erreichbar. Das Kommando

```
% wsimport -keep -p clientA http://localhost:9876/tsa?wsdl
```

generiert die üblichen Artefakte im Unterverzeichnis *clientA*. Das Attribut

```
serviceName = RevisedTimeServer
```

der `@WebService`-Annotation ~~causes the service artifact to be the class named~~ `RevisedTimeServer` und das Attribut

```
name = AnnotatedTimeServer
```

~~causes the portType artifact to be the interface named~~ *`AnnotatedTimeServer`*. Der Java-Client für diesen geänderten Dienst veranschaulicht diese Punkte:

```
import clientA.RevisedTimeServer;
import clientA.AnnotatedTimeServer;

class TimeClientA {

    public static void main(String[] args) {
        RevisedTimeServer ts = new RevisedTimeServer();
        AnnotatedTimeServer ats = ts.getAnnotatedTimeServerPort();

        System.out.println(ats.timeString("Hi, world!"));
        System.out.println(ats.timeElapsed());
        ats.acceptInput("Hello, world!");

    }

}
```

Das `targetNamespace`-Attribut hat den Wert `http://ch02.tsa` (ohne abschließenden Schrägstrich). Diese Notation ~~bewirkt~~ ~~bewährleistet~~, daß *dieser* Namensraum-URI im WSDL-Dokument verwendet wird, das heißt `ch02.tsa` wird nicht, wie beim ursprünglichen *`ch01.ts.TimeServer`*-Dienst, in `tsa.ch02` invertiert. Die Attribute der *`@SOAPBinding`-Annotation* haben ihre Standardwerte und sind lediglich deshalb vorhanden, um die Standardwerte einmal zu zeigen.

[79] Unter den drei mit `@WebMethod` annotierten Methoden, trägt *`getTimeAsString()`* mit `@WebMethod`, `@WebResult` und `@WebParam` die meisten Annotationen. Die Operationsnamen der Methoden `getTimeAsString()` und `getTimeAsElapsed()` sind explizit als `time_string` und `time_elapsed` definiert. Der folgende Auszug aus dem `<message>`-Abschnitt des WSDL-Dokumentes gibt diese Einstellungen wieder. Beachten Sie auch, daß die `name`-Attribute der `<message>`-Elemente die Operationsnamen reflektieren, wie in der Voreinstellung der Verpackungsvariante des Dokumentstils:

```
<message name="time_string">
  <part element="tns:time_string" name="parameters"></part>
</message>
<message name="time_stringResponse">
  <part element="tns:time_stringResponse" name="parameters"></part>
</message>
<message name="time_elapsed">
  <part element="tns:time_elapsed" name="parameters"></part>
</message>
<message name="time_elapsedResponse">
  <part element="tns:time_elapsedResponse" name="parameters"></part>
</message>
```

Die Annotation `@WebResult` der Methode `getTimeAsString()` wirkt sich nicht auf das WSDL-Dokument, sondern auf die SOAP-Antwortnachricht aus, genauer auf das Element `<ns1:ts_out>`

im verpackten Inhalt des SOAP-Bodys:

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://ch02.tsa">
  <soapenv:Body>
    <ns1:time_stringResponse>
      <ns1:ts_out>Hi, world! at Thu Oct 23 22:24:59 CDT 2008</ns1:ts_out>
    </ns1:time_stringResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Die mit `@WebMethod` annotierte Methode `acceptInput()` trägt außerdem die Annotation `@Oneway`, die voraussetzt, daß die Methode den Rückgabetyp `void` hat. Infolgedessen kann der Client eine Anfrage über die parametrisierte `acceptInput()`-Operation senden, erhält aber keine Antwort. Das folgende WSDL-Element definiert die Operation, so daß sie nur eine Eingabe- aber keine Ausgabenachricht hat:

```
<operation name="acceptInput">
   <input message="tns:acceptInput"></input>
</operation>
```

Dieses Beispiel zeigt einige Möglichkeiten des „Code First, Contract Aware"-Ansatzes bei Java-Webservices. Ist der Autor des Dienstes bereit, Standards zu akzeptieren, so sind Annotationen wie `@WebService` und `@WebMethod` klar und einfach zu gebrauchen. Ist eine feiner aufgelöste Kontrolle über das WSDL-Dokument und die SOAP-Nachrichten erforderlich, stehen entsprechende zusätzliche Annotationen zur Verfügung.

2.5.4 Grenzen des WSDL-Dokumentes

[80] Als Beschreibungen von Webservices, sollten WSDL-Dokumente zur Veröffentlichung geeignet und auffindbar sein. Eine *UDDI*-Registratur (Universal Description, Discovery and Integration) ist eine Möglichkeit zur Veröffentlichung von WSDL-Dokumenten, so daß potentielle Clients die Dokumente finden und letztendlich die beschriebenen Dienste in Anspruch nehmen können. UDDI unterstützt keinen Dienstbeschreibungtyp direkt, darunter WSDL, sondern verwendet ein eigenes Typsystem, das WSDL-Dokumente einschließt. Nach den UDDI-Begrifflichkeiten ist ein WSDL-Dokument im Wesentlichen ein zweiteiliges Dokument. Der erste Teil umfaßt die Abschnitte `<types>` bis `<binding>`, und wird als UDDI-*Dienstschnittstelle* bezeichnet. Der zweite Teil besteht aus allen Importdirektiven sowie dem `<service>`-Abschnitt und wird als UDDI-*Dienstimplementierung* bezeichnet. In der WSDL sind Schnittstelle und Implementierung des Dienstes zwei Teile ein und desselben Dokumentes. Bei UDDI stehen beide Teile in separaten Dokumenten.

[81] Auch nachdem ein WSDL-Dokument lokalisiert worden ist, unter Umständen per UDDI, bleiben wesentliche Fragen hinsichtlich des in diesem WSDL-Dokument beschriebenen Dienstes bestehen. Das WSDL-Dokument beschreibt zum Beispiel nicht die Semantik des Dienstes, in einfacheren Worten also, welche Aufgabe der Dienst erfüllt. Ein WSDL-Dokument beschreibt präzise, was als die Aufrufsyntax des Diensts bezeichnet werden könnte: Die Namen der Operationen des Dienstes, das erwartete MEP (zum Beispiel das Request/Response-Pattern), Anzahl, Reihenfolge und Typen der Argumente/Rückgabewerte sowie eventuelle Statuswerte der einzelnen Operationen. Das WSDL-Dokument des E-Commerce-Dienstes von Amazon hat etwa 4000 Zeilen und beinhaltet alle diese Informationen für jede der zahlreichen Operationen. Dennoch enthält das WSDL-Dokument selbst keinerlei Informationen darüber, wie der Dienst anzuwenden ist. Dies herauszufinden bleibt dem

Programmierer überlassen, der, voraussichtlich aufgrund seiner Erfahrung mit der Website von Amazon, anhand der Operationsnamen des Dienstes wie `itemSearch()`, `sellerLookup()` und `cartCreate()` erkennt, daß der E-Commerce-Dienst die aus den Browsersitzungen auf der Website von Amazon vertraute Funktionalität reproduziert. Amazon bietet Ergänzungsmaterial in Form von Dokumentation, Tutorials und Bibliotheken mit Beispielen an, um die sematischen Informationen zu liefern, die das WSDL-Dokument nicht enthält. Das W3C unterhält unter der Rubrik WSDL-S (für „Semantik“) Initiativen *in web semantics*. Weiterführende Informationen über WSDL-S finden Sie unter der Adresse *http://www.w3.org/Submission/WSDL-S*. *As of now*, ist ein WSDL-Dokument typischerweise nur zu gebrauchen, wenn der Client-Programmierer bereits weiß, welche Aufgabe der Dienst hat.

2.6 Ausblick

[82] SOAP-basierte Java-Webservices haben zwei Ebenen. Die Applikationsebene, bestehend aus dem Dienst selbst und allen Java-basierten Clients, verbirgt typischer- und passenderweise die SOAP-Nachrichten. Die Behandlerebene besteht aus client- beziehungsweise serverseitigen Nachrichtenbehandlern, welche die SOAP-Nachrichten ver- und bearbeiten können. Gelegentlich ist der direkte Zugriff auf die SOAP-Nachrichten nützlich, wie das nächste Kapitel zeigt. Das nächste Kapitel betrachtet außerdem, wie SOAP-basierte Dienste voluminöse byteorientierte Datenmengen effizient als „Nutzlast“ transportieren können.

Kapitel 3

Verarbeitung von SOAP-Nachrichten

Inhaltsübersicht

3.1 Behandler

[0] SOAP-Nachrichten wurden bis jetzt nur unter dem Aspekt betrachtet, wie ein SOAP-basierter Webservice „hinter den Kulissen“ funktioniert. Das Ziel der Webservicetechnologie bei Java ist schließlich die Entwicklung und der Betrieb von Webservices, weitestgehend ohne Beachtung der Infrastruktur. Das Abhören der Nachrichten eines Dienstes auf der Übertragungsebene, um die Vorgänge „unter der Haube“ zu verstehen, ist eine Sache. Das Verarbeiten von SOAP-Nachrichten mit dem Ziel, die Applikationslogik zu unterstützen, ist eine ganz andere.

[1] Hin und wieder muß ein Webservice oder Client einen SOAP-Envelope (ein `<Envelope>`-Element) verarbeiten. Ein Client könnte zum Beispiel aus Sicherheitsgründen Berechtigungsnachweise in den SOAP-Header (das `<Header>`-Element) einer Nachricht einsetzen und der Webservice auf der Empfängerseite diese Berechtigungsnachweise aus den SOAP-Headern der eingetroffenen Nachrichten abfragen und auswerten. Das Hauptthema dieses Kapitels ist Anlegen, Abfragen und sonstiges Verarbeiten von Informationen in SOAP-Nachrichten. Das Ziel dieses Kapitels besteht darin, Verfahren zur direkten Be- und Verarbeitung der Nachrichten bei SOAP-basierten Diensten zu veranschaulichen. Darüberhinaus behandelt dieses Kapitel SOAP-Anhänge, vor allem im Hinblick auf voluminöse byteorientierte Nutzdaten.

3.1.1 Die Protokollversionen 1.1 und 1.2

[2] Es gibt zur Zeit die beiden Versionen 1.1 und 1.2 des SOAP-Protokolls. Das W3C erörtert die Notwendigkeit einer Version 1.3. Erfreulicherweise wirken sich die Unterschiede zwischen Version 1.1 und Version 1.2 nur geringfügig auf die Entwicklung von Webservices im Allgemeinen und auf das Programmieren mit der JAX-WS-Bibliothek im Besonderen aus. Es gibt aber Ausnahmefälle. Beispielsweise unterscheidet sich die Struktur des SOAP-Headers zwischen beiden Versionen, wodurch sich Auswirkungen auf die Programmierung ergeben können. Abbildung 3.1 zeigt den generellen Aufbau einer SOAP-Nachricht, gültig für beide Protokollversionen.

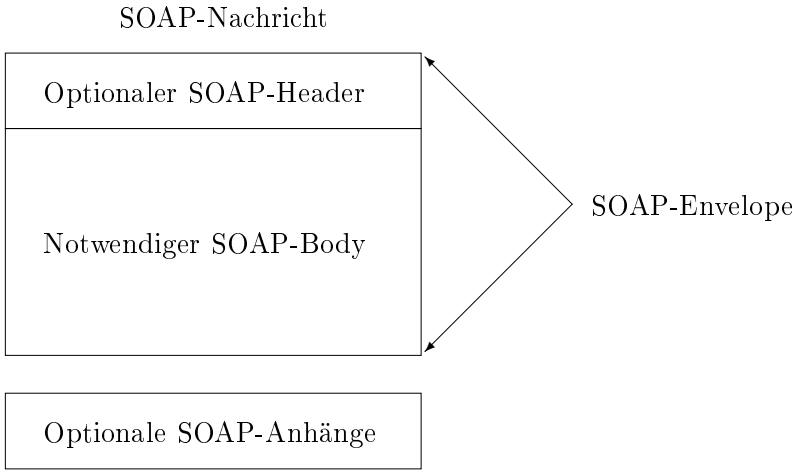

Abbildung 3.1: Aufbau einer SOAP-Nachricht.

[3] SOAP-Nachrichten können bei beiden Versionen mit Hilfe von *Behandlern* be- und verarbeitet werden. Diese Behandler sind von Ihnen selbst geschriebene Klassen, die sogenannte *Rückrufmethoden* (*callbacks*) definieren, das heißt Methoden, die von den Laufzeitbibliotheken für Webservices aufgerufen werden, so daß eine Applikation Zugriff auf den unterliegenden SOAP-Nachrichtenverkehr erhält. Auf der Applikationsebene, bestehend aus Webservices und ihren Clients, bleiben die SOAP-Nachrichten verborgen. Auf der Behandlerebene dagegen, ist die SOAP-Nachricht zugänglich, so daß der Programmierer die ein/ausgehenden Nachrichten verarbeiten kann.

[4] Auch auf der Behandlerebene belaufen sich die Unterschiede zwischen den Protokollversionen 1.1 und 1.2 zumeist auf Feinheiten, die in der Regel unbeachtet bleiben können. Beispielsweise erlaubt Version 1.1 die Platzierung von Elementen *nach* dem SOAP-Body (dem `<Body>`-Element). Im Gegensatz dazu ist der SOAP-Body unter Version 1.2 das letzte Unterelement des SOAP-Envelopes. Es sind allerdings auch unter Version 1.1 nur künstliche Beispiele, in denen der SOAP-Envelope Unterelemente im „Niemandsland“ zwischen dem Ende des SOAP-Bodys und dem Ende des SOAP-Envelopes enthält. Version 1.1 bindet den Transport der SOAP-Nachrichten an HTTP („SOAP

over HTTP“), während Version 1.2 auch SMTP als Transportprotokoll unterstützt („SOAP over SMTP“). Version 1.1 der SOAP-Spezifikation ist ein einziges Dokument, Version 1.2 erstreckt sich dagegen über drei Dokumente. JAX-WS verwendet, wie die meisten Frameworks, Version 1.1 des SOAP-Protokolls als Voreinstellung und stellt Version 1.2 optional zur Verfügung, wie eine Reihe von Beispielen in diesem Kapitel zeigen. Ab Version 1.2 ist die Abkürzung SOAP nicht länger ein Akronym!

3.1.2 Die messaging architecture: Sender, Mittler und Empfänger

[5] Eine SOAP-Nachricht wird in eine Richtung vom Sender zum Empfänger hin übertragen. Das fundamentale Nachrichtenaustauschmuster für SOAP ist also unidirektional. SOAP-basierte Applikationen wie Webservices können aus der unidirektionalen Nachrichtenübertragung aufwendiger gestaltete Konversationsmuster kombinieren. Das Request/Response-Muster SOAP-basierter Dienste beschreibt eine kurze Konversation, die mit der Anfrage beginnt und mit der Antwort endet. Nachrichtenaustauschmuster wie Request/Response und Solicit/Response können miteinander kombiniert werden, um bei Bedarf weiterreichende Konversationsmuster zu implementieren.

[6] Obwohl eine SOAP-Nachricht für einen endgültigen Empfänger bestimmt ist, gestattet die messaging architecture von SOAP sogenannte *SOAP-Mittler* (*intermediaries*), das heißt nichtterminale Empfänger oder *Knoten* entlang des Weges vom Sender zum eigentlichen Empfänger. Ein SOAP-Mittler (kurz „Mittler“) kann eingehende SOAP-Nachrichten untersuchen und sogar verändern, bevor er sie wieder auf den Weg zu ihrem endgültigen Empfänger schickt. Abbildung 3.2 zeigt einen Sender, zwei Mittler und den endgültigen Empfänger.

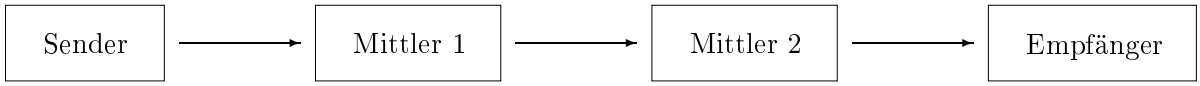

Abbildung 3.2: Sender, Mittler und endgültiger Empfänger.

[7] Ein SOAP-Envelope besteht aus einem verpflichtenden SOAP-Body und einem optionalen SOAP-Header. Der SOAP-Body darf leer sein. Ein Mittler sollte stets nur die Elemente im SOAP-Header untersuchen und verarbeiten, nicht aber den SOAP-Body, der die vom Sender für den Empfänger bestimmten Nutzdaten transportiert. Der SOAP-Header ist dagegen für den Transport aller vom Empfänger oder den Mittlern benötigten Metainformationen vorgesehen, zum Beispiel die digitale Signatur des Senders als Identitätsnachweis oder einen Zeitstempel, der das „Verfallsdatum“ der Information im SOAP-Body der Nachricht angibt. Die Elemente im optionalen Header werden im SOAP-Jargon als *Headerblöcke* (*header blocks*) bezeichnet. Das erste Beispiel zeigt eine SOAP-Nachricht mit Header gemäß Protokollversion 1.1. Der Header enthält einen Headerblock (`<uuid>`-Element):

Beispiel 3.1: Ein SOAP-Header.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
    <uuid xmlns="http://ch03.fib"
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next">
        ca12fd33-16e1-4a95-b17e-3ef6744babdc
    </uuid>
  </S:Header>
  <S:Body>
    <ns2:countRabbits xmlns:ns2="http://ch03.fib">
      <arg0>45</arg0>
```

```
      </ns2:countRabbits>
    </S:Body>
  </S:Envelope>
```

Der `<uuid>`-Headerblock enthält einen UUID-Wert (Universally Unique Identifier), eine als Zeichenkette formatierte 128 Bit lange Zahl aus hexadezimalen Ziffern. Eine UUID ist ein mit hoher Wahrscheinlichkeit eindeutiger Identifikator, in diesem Beispiel der Identifikator der Anfrage an den `RabbitCounter`-Dienst, der in den [Unterabschnitten 3.1.4 und 3.1.6] vorgestellt wird. Das `RabbitCounter`-Beispiel setzt voraus, daß jede Anfrage an den Dienst eine eigene UUID hat, die zusammen mit weiteren Informationen zur späteren Untersuchung protokolliert wird.

[8] Das `<uuid>`-Element hat ein `SOAP-ENV:actor`-Attribut, dessen Wert mit `next` endet. Der `next`-Akteur ist der nächste Empfänger auf dem Weg der Nachricht vom Sender bis zum endgültigen Empfänger, das heißt jeder Mittler und der endgültige Empfänger treten in der Rolle des `next`-Akteurs auf. Ein `next`-Akteur hat die Aufgabe, die Headerblöcke in einer der Applikation dienlichen Weise zu inspizieren. Kann ein Mittler einen Headerblock nicht verarbeiten, so sollte der Knoten einen SOAP-Fault (Fehlermeldung) auszuwerfen. In Version 1.2 des SOAP-Protokolls kann dieses Verhalten explizit gefordert werden, indem das Attribut `mustUnderstand` eines Headerblocks mit `true` bewertet wird. In diesem Fall ist der Knoten *verpflichtet*, einen SOAP-Fault auszuwerfen, wenn der Headerblock nicht verarbeitet werden kann.

[9] Die SOAP-Spezifikation erklärt nicht explizit, wie ein intermediärer oder terminaler Knoten einen Headerblock verarbeiten soll, da die Verarbeitungsweise applikationsspezifisch und keine Anforderung von SOAP selbst ist. Der nächste Knoten könnte beispielsweise mit Hilfe eines von der Applikation festgelegten Algorithmus' wie SHA-1 (Secure Hash Algorithm 1) prüfen, ob der UUID-Wert ein vorgegebenes Format hat und andernfalls einen SOAP-Fault auswerfen, um die Abweichung anzuzeigen.

[10] Zunächst steht die Frage im Vordergrund, wie ein Header wie in Beispiel 3.1 erzeugt werden kann. Java-Webservices generieren per Voreinstellung SOAP-Anfragen ohne Header. Unterabschnitt 3.1.5 zeigt, wie sich die SOAP-Nachricht in Beispiel 3.1 erzeugen läßt.

3.1.3 Das JAX-WS-Behandlerframework

[11] Die JAX-WS stellt ein *Behandlerframework*, zur Verfügung, mit dessen Unterstützung Applikationen ein- und ausgehende SOAP-Nachrichten untersuchen und verarbeiten können. Zwei Schritte sind erforderlich, um einen Behandler bei diesem Framework zu registrieren:

- *Schritt 1*: Schreiben einer Behandlerklasse, die das Interface *`javax.xml.ws.handler.Handler`* implementiert. Die JAX-WS-Bibliotheken definieren zwei Interfaces, die *`Handler`* erweitern, nämlich *`javax.xml.ws.handler.LogicalHandler`* und *`javax.xml.ws.handler.soap.SOAPHandler`*. Seinem Namen entsprechend ist *`LogicalHandler`* protokollneutral, während *`SOAPHandler`* SOAP-spezifisch ist. Der Zugriff eines Behandlers vom Typ *`LogicalHandler`* beschränkt sich auf die Nutzdaten im SOAP-Body. Ein Behandler vom Typ *`SOAPHandler`* hat dagegen Zugriff auf die gesamte SOAP-Nachricht, inklusive des optionalen Headers und sämtlicher optionalen Anhänge. Eine Klasse, die *`LogicalHandler`* oder *`SOAPHandler`* implementiert, muß stets drei Methoden definieren. Die Methode `handleMessage()` gewährt dem Programmier Zugriff auf die [underlying/unterliegende/eigentliche] Nachricht. Die beiden anderen gemeinsamen Methoden sind `handleFault()` und `close()`. Das Interface *`SOAPHandler`* verlangt die Implementierung einer vierten Methode namens `getHeaders()`. Diese Methoden lassen sich am besten anhand von Beispielen erklären.

- *Schritt 2*: Einordnen der Behandlerklasse in eine Behandlerkette, typischerweise mit Hilfe

einer Konfigurationsdatei. Behandler können aber auch programmatisch verwaltet werden. Ein Beispiel folgt in [Unterabschnitt 3.1.6 beziehungsweise 3.1.7].

[12] Nach dem Registrieren im Behandlerframework funktioniert ein selbstgeschriebener Behandler wie eine Abfangvorrichtung mit Zugriff auf jede ein- oder ausgehende Nachricht. Beim Nachrichtenaustauschmuster Request/Response hat ein clientseitiger Behandler zum Beispiel Zugriff auf die ausgehende Nachricht, nachdem sie erzeugt wurde, aber bevor sie an den Webservice gesendet wird. Derselbe clientseitige Behandler hat auch Zugriff auf die eingehende Antwort des Dienstes. Ein serviceseitiger Behandler hat bei diesem Nachrichtenaustauschmuster Zugriff auf die eingehende Anfrage und die ausgehende Antwort nachdem die Antwort erzeugt wurde.

[13] Das JAX-WS-Behandlerframework unterstützt somit das Entwurfsmuster *Chain-of-Responsibility*, das Servlet-Programmierern im Zusammenhang mit Filtern begegnet. Die Idee besteht darin, die Zuständigkeit unter verschiedenen Behandlern aufzuteilen, damit die Gesamtanwendung hochmodular und daher pflegeleicht ist.

3.1.4 Der RabbitCounter-Dienst

[14] Nun können wir uns anhand eines Beispiels den Details nähern. Im Rahmen dieses Beispiels werden Konstrukte einführt, die für spätere Beispiele nützlich sind. Hier eine Zusammenfassung der Eigenschaften des Beispiels:

- Der `RabbitCounter`-Dienst hat eine Operation namens `countRabbits()`, die ein ganzzahliges Argument erwartet und einen ganzzahligen Rückgabewert liefert, nämlich die Fibonaccizahl des Argumentes. Fibonaccizahlen treten in der Botanik, den Ingenieurwissenschaften, der Informatik, der Ästhetik, dem Finanzmarkt und eben auch in der Kaninchenzucht auf. Die Operation `countRabbits()` hat einen Parameter, um SOAP-Faults demonstrieren zu können. In diesem Beispiel wird ein SOAP-Fault ausgeworfen, wenn `countRabbits()` mit einem negativen ganzzahligen Argument aufgerufen wird. Das Beispiel liefert somit einen ersten Einblick in die Bedeutung und Verwendung von SOAP-Faults.

- Vom `RabbitCounter`-Dienst, seinen Clients sowie sämtlichen Mittlern zwischen Client und Dienst wird erwartet, die SOAP-Header der Nachrichten zu verarbeiten. Der Client legt in jeder gesendeten Anfrage einen Headerblock an und alle Mittler sowie der endgültige Empfänger prüfen die darin enthaltene Information. Falls erforderlich, wird ein SOAP-Fault ausgeworfen. In den späteren Beispielen dienen Headerblocks zum Transport von Berechtigungsnachweisen, wie etwa digitalen Signaturen. Im Augenblick gilt das primäre Interesse dem Anlegen und der Verarbeitung von Headerblocks.

- Java-Webservices haben zwei unterschiedliche Möglichkeiten, um SOAP-Faults auszuwerfen. Das `RabbitCounter`-Beispiel zeigt beide Alternativen. Der einfachere Weg besteht darin, einen Ausnahmetyp von der Klasse `Exception` abzuleiten (zum Beispiel `FibException`) und die Ausnahme bei Bedarf aus einer mit `@WebMethod` annotierten Methode auszuwerfen. JAX-WS bildet die Java-Ausnahme automatisch auf einen SOAP-Fault ab. Der andere, aufwendigere Weg besteht darin, den SOAP-Fault über einen Behandler auszuwerfen. In diesem Fall wird eine `javax.xml.ws.soap.SOAPFaultException` erzeugt und ausgeworfen.

Die Einzelheiten werden in den folgenden Unterabschnitten ausgearbeitet.

3.1.5 Einsetzen eines Headerblocks in den SOAP-Header

[15] Der Headerblock in Beispiel 3.1 stammt aus dem folgenden clientseitigen Behandler:

Beispiel 3.2: Einsetzen eines SOAP-Headerblock per Behandler.

```
package fibC;

import java.util.UUID;
import java.util.Set;
import java.util.logging.Logger;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPHeaderElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPConstants;
import java.io.IOException;

public class UUIDHandler implements SOAPHandler<SOAPMessageContext> {

    private static final String LoggerName = "ClientSideLogger";
    private Logger logger;
    private final boolean log_p = true; // set to false to turn off

    public UUIDHandler() {
        logger = Logger.getLogger(LoggerName);
    }

    public boolean handleMessage(SOAPMessageContext ctx) {

        if (log_p) logger.info("handleMessage");

        // Is this an outbound message, i.e., a request?
        Boolean request_p = (Boolean)
            ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

        // Manipulate the SOAP only if it's a request
        if (request_p) {

            // Generate a UUID and a timestamp to place in the message header.
            UUID uuid = UUID.randomUUID();

            try {

                SOAPMessage msg = ctx.getMessage();
                SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
                SOAPHeader hdr = env.getHeader();

                // Ensure that the SOAP message has a header.
                if (hdr == null) hdr = env.addHeader();

                QName qname = new QName("http://ch03.fib", "uuid");
                SOAPHeaderElement helem = hdr.addHeaderElement(qname);

                helem.setActor(SOAPConstants.URI_SOAP_ACTOR_NEXT); // default
                helem.addTextNode(uuid.toString());
                msg.saveChanges();

                // For tracking, write to standard output.
                msg.writeTo(System.out);

            }

            catch(SOAPException e) { System.err.println(e); }
            catch(IOException e) { System.err.println(e); }
```

```
            }

            return true; // continue down the chain

        }

        public boolean handleFault(SOAPMessageContext ctx) {
            if (log_p) logger.info("handleFault");
            try {
                ctx.getMessage().writeTo(System.out);
            }
            catch(SOAPException e) { System.err.println(e); }
            catch(IOException e) { System.err.println(e); }
            return true;
        }

        public Set<QName> getHeaders() {
            if (log_p) logger.info("getHeaders");
            return null;
        }

        public void close(MessageContext messageContext) {
            if (log_p) logger.info("close");
        }

    }
```

[16] Da die Klasse `UUIDHandler` einen clientseitigen Behandler definiert und das Nachrichtenaustauschmuster Request/Response verwendet wird, ruft das Framework die `handleMessage()`-Methode erst auf, *nachdem* die SOAP-Anfrage erzeugt wurde, aber *bevor* die Anfrage an den Dienst gesendet wird. Stellen Sie sich einen SOAP-basierten Dienst `TestService` mit einem Client namens `TestClient` vor. Der Client verwendet einen Behandler vom Typ `UUIDHandler`. Hier eine Zusammenfassung der Geschehnisse zwischen dem `TestService`, dem `TestClient` und der Behandlerklasse `UUIDHandler`:

- Veranlaßt der `TestClient` eine Anfrage an den `TestService`, so erzeugt die clientseitige JAX-WS-Bibliothek eine SOAP-Nachricht, die diese Anfrage repräsentiert.
- Nachdem die SOAP-Nachricht erzeugt wurde, aber bevor sie gesendet wird, werden die Rückrufmethoden der Klasse `UUIDHandler` aufgerufen. Die `handleMessage()`-Methode hat Zugriff auf die gesamte SOAP-Nachricht, da die Behandlerklasse `UUIDHandler` das Interface *`SOAPHandler`* statt *`LogicalHandler`* implementiert, welches lediglich die Nutzdaten der Nachricht erreicht. Die Rückrufmethode `handleMessage()` setzt einen UUID-Wert in den Header der SOAP-Nachricht ein.
- Hat der Behandler vom Typ `UUIDHandler` seine Arbeit erledigt, so wird die SOAP-Nachricht mit dem eingefügten Headerblock auf ihren Weg zum endgültigen Empfänger geschickt, hier dem `TestService`-Dienst.

[17] Noch einige zusätzliche Details zum gerade beschriebenen Vorgang: Derselbe Behandler vom Typ `UUIDHandler` hat ebenfalls Zugriff auf die beim Client eingehende Nachricht (Antwort). Die `handleMessage()`-Methode hat ein Argument vom Typ *`javax.xml.ws.handler.soap.SOAPMessageContext`*, welches den Zugriff auf die unterliegende SOAP-Nachricht gestattet. Die Rückrufmethode `handleMessage()` prüft dementsprechend zuerst, ob die SOAP-Nachricht eine Anfrage, aus der Perspektive des Clients ist, also eine ausgehende Nachricht. Ist die Nachricht eine Anfrage, so erzeugt der Behandler einen UUID-Wert, der in einen Headerblock eingesetzt wird. Der SOAP-Header ist bei beiden Protokollversionen 1.1 und 1.2 optional. Der Behandler prüft daher, ob die bereits erzeugte SOAP-Nachricht einen Header hat und legt andernfalls einen Header im SOAP-Envelope an. Dann wird im SOAP-Header ein Objekt vom Typ *`SOAPHeaderElement`* angelegt, der

den Headerblock programmatisch repräsentiert und das `actor`-Attribut mit Hilfe der entsprechenden Änderungsmethode bewertet:

```
SOAPHeaderElement helem = hdr.addHeaderElement(qname);
helem.setActor(SOAPConstants.URI_SOAP_ACTOR_NEXT); // the default
```

Der UUID-Wert wird als XML-Textknoten im Header angelegt und alle Änderungen an der SOAP-Nachricht werden gesichert. Die ausgehende SOAP-Nachricht reflektiert die vorgenommenen Änderungen. Der SOAP-Header lautet nach Ausführung des Behandlers und Sichern der Ergänzungen:

```
<S:Header>
    <uuid xmlns="http://ch03.fib"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next">
      ca12fd33-16e1-4a95-b17e-3ef6744babdc
    </uuid>
</S:Header>
```

Der `UUIDHandler`-Behandler in Beispiel 3.2 ist der einzige Behandler in der für den Client `FibClient` (Seite 88) wirksamen *Behandlerkette* (Aneinanderreihung von Behandlern). Es können weitere Behandler in der Kette positioniert werden. Die Behandlerkette kann zwar programmatisch festgelegt werden, aber die Deklaration in einer Konfigurationsdatei ist sauberer. Das folgende Beispiel zeigt die Konfigurationsdatei *handler-chain.xml*, wobei Sie aber einen beliebigen Namen wählen können:

Beispiel 3.3: Konfigurationsdatei *handler-chain.xml* für den UUIDHandler.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<javaee:handler-chains
    xmlns:javaee="http://java.sun.com/xml/ns/javaee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <javaee:handler-chain>
    <javaee:handler>
      <javaee:handler-class>fibC.UUIDHandler</javaee:handler-class>
    </javaee:handler>
  </javaee:handler-chain>
</javaee:handler-chains>
```

Weitere Behandler, beliebige Kombinationen aus Implementierungen der Interfaces *`LogicalHandler`* und *`SOAPHandler`*, würden vor oder nach der Deklaration von `fibC.UUIDHandler` (vollqualifizierter Name der Klasse aus Beispiel 3.2) angeordnet werden. Die Behandlermethoden eines Typs, zum Beispiel die `handleMessage()`-Methoden der *`SOAPHandler`*-Implementierungen, werden in der deklarierten Reihenfolge von oben nach unten aufgerufen, solange keine Behandlermethode die Verarbeitung der Kette durch die Rückgabe des Wertes `false` unterbricht. Die Methoden `handleMessage()` und `handleFault()` geben je einen booleschen Wert zurück. Der Rückgabewert `true` bedeutet, daß die Verarbeitung der Nachricht mit dem nächsten Behandler in der Kette fortgesetzt wird, während die Verarbeitung bei `false` beendet wird. Beendet ein Behandler die Verarbeitung einer ausgehenden Nachricht, also einer clientseitigen Anfrage oder einer serviceseitigen Antwort, so wird die Nachricht nicht gesendet.

[18] Die Reihenfolge der Deklaration der Behandler in der Konfigurationsdatei von oben nach unten bestimmt die Reihenfolge, in der die Behandlermethoden eines Typs (zum Beispiel *`SOAPHandler`*) verarbeitet werden. Die Reihenfolge der Aufrufe zur Laufzeit kann von der Reihenfolge in der Konfigurationsdatei abweichen. Begründung:

- Bei einer ausgehenden Nachricht, zum Beispiel einer clientseitigen Anfrage beim Request/Response-Muster, werden die *`LogicalHandler`*-Versionen der Methoden `handleMessage()` und `handleFault()` *vor* den *`SOAPHandler`*-Versionen aufgerufen.

- Bei einer eingehenden Nachricht werden die *`SOAPHandler`*-Versionen der Methoden `handleMessage()` und `handleFault()` *vor* den *`LogicalHandler`*-Versionen aufgerufen.

Deklariert eine Konfigurationsdatei die SOAP-Behandler (SH) und die logischen Behandler (LH) beispielsweise in der folgenden Reihenfolge:

```
SH1
LH1
SH2
SH3
LH2
```

so werden die Behandler, trotz ihrer Anordnung in der Konfigurationsdatei, bei einer *ausgehenden* Nachricht in der folgenden Reihenfolge aufgerufen:

```
LH1
LH2
SH1
SH2
SH3
```

Bei einer *eingehenden* Nachricht lautet die Reihenfolge der Behandleraufrufe dagegen:

```
SH1
SH2
SH3
LH1
LH2
```

Abbildung 3.3 zeigt die Reihenfolge der Verarbeitung der logischen Behandler und SOAP-Behandler zur Laufzeit. Diese Laufzeitreihenfolge ist sinnvoll, da ein Behandler vom Typ *`LogicalHandler`* lediglich Zugriff auf den SOAP-Body der Nachricht hat, während ein Behandler vom Typ *`SOAPHandler`* die gesamte SOAP-Nachricht erreichen kann. Bei einer ausgehenden Nachricht sollten die logischen Behandler in der Lage sein, die Nutzdaten, also den SOAP-Body, zu verarbeiten, bevor der SOAP-Behandler die gesamte Nachricht verarbeitet. Braucht eine Applikation keine SOAP-Header oder -Anhänge zu verarbeiten, so genügt ein Behandler vom Typ *`LogicalHandler`*.

[19] Die Konfigurationsdatei, hier *handler-chain.xml*, kann an einer beliebigen Stelle im Klassenpfad deponiert werden. Bei diesem Beispiel liegt die Datei im Verzeichnis *fibC*, dem benannten Package der `wsimport`-Artefakte. Der übersetzte Client *FibClient.class* liegt in dem Verzeichnis,

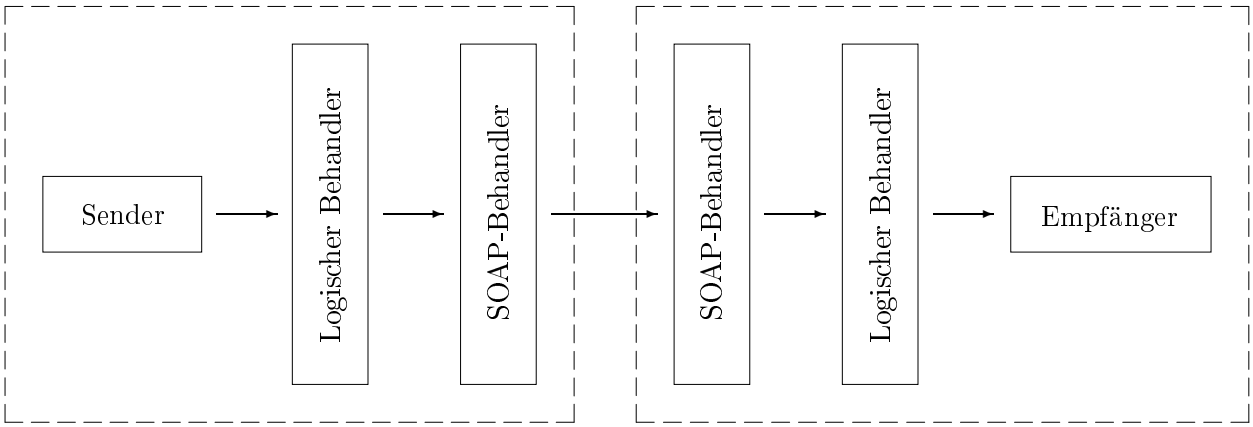

Abbildung 3.3: Verarbeitungsreihenfolge der logischen Behandler und SOAP-Behandler zur Laufzeit.

welches das Unterverzeichnis *fibC* enthält. Die Begründung für dieses Layout erfolgt im nächsten [Abschnitt/Unterabschnitt]. Der Quelltext der Klasse `FibClient` lautet:

```
import fibC.RabbitCounterService;
import fibC.RabbitCounter;

class FibClient {
    public static void main(String[] args) {
        RabbitCounterService service = new RabbitCounterService();
        RabbitCounter port = service.getRabbitCounterPort();
        try {
            int n = -45;
            System.out.println("fib(" + n + ") = " + port.countRabbits(n));
        }
        catch(Exception e) { System.err.println(e); }
    }
}
```

Der Aufruf der `countRabbits()`-Methode des Dienstes steht in einem `try`-Block, da diese Methode beim Aufruf mit einem negativen Argument (wie hier der Fall) einen SOAP-Fault auswirft. Davon abgesehen, ist der Aufbau des Clients mittlerweile vertraut.

[20] Die Behandlerklasse `UUIDHandler` wird außerdem genutzt, um die ausgehende SOAP-Nachricht auf der Übertragungsebene abzuhören und über die Standardausgabe auszugeben. SOAP-Behandler sind somit bei Java eine Alternative zu Kommandos wie `tcpmon` und `tcpdump`.

3.1.6 Deklaratives Registrieren eines clientseitigen Behandlers

[21] Der Client `FibClient` sendet Anfragen an den `RabbitCounter`-Dienst:

Beispiel 3.4: Der RabbitCounter-Dienst.

```
package ch03.fib;

import java.util.Map;
import java.util.HashMap;
import java.util.Collections;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;
import javax.jws.soap.SOAPBinding.ParameterStyle;

@WebService(targetNamespace = "http://ch03.fib")
@SOAPBinding(style          = SOAPBinding.Style.DOCUMENT,
             use            = SOAPBinding.Use.LITERAL,
             parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
public class RabbitCounter {

    // stores previously computed values
    private Map<Integer, Integer> cache =
        Collections.synchronizedMap(new HashMap<Integer, Integer>());

    @WebMethod
    public int countRabbits(int n) throws FibException {

        // Throw a fault if n is negative.
        if (n < 0) throw new FibException("Neg. arg. not allowed.", n + " < 0");

        // Easy cases.
```

```
        if (n < 2) return n;

        // Return cached values if present.
        if (cache.containsKey(n)) return cache.get(n);
        if (cache.containsKey(n - 1) &&
            cache.containsKey(n - 2)) {
            cache.put(n, cache.get(n - 1) + cache.get(n - 2));
            return cache.get(n);
        }

        // Otherwise, compute from scratch, cache, and return.
        int fib = 1, prev = 0;
        for (int i = 2; i <= n; i++) {
            int temp = fib;
            fib += prev;
            prev = temp;
        }
        cache.put(n, fib); // cache value for later lookup
        return fib;
    }
}
```

Der Dienst hat nur eine einzige Operation: `countRabbits()`. Die Fibonaccizahlen können zwar per Rekursion berechnet werden, aber diese Vorgehensweise ist nicht effizient. Die rekursive Definition der Fibonaccizahlen

$$\texttt{fib(n)} = \begin{cases} \text{nicht definiert}, & \text{falls } \texttt{n} < 0, \\ \texttt{n}, & \text{falls } 0 \leq \texttt{n} < 2, \\ \texttt{fib(n-1)} + \texttt{fib(n-2)}, & \text{sonst.} \end{cases}$$

läßt sich zwar mühelos in eine rekursive Methode übertragen:

```
int fib(int n) {
    if (n < 0) throw new RuntimeException("Undefined for negative values.");
    if (n < 2) return n;                    // base case
    else return fib(n - 1) + fib(n - 2);    // recursive calls
}
```

aber die rekursive Implementierung wiederholt Rechenschritte. Beispielsweise berechnet die rekursive Methode beim Aufruf `fib(5)` den Wert von `fib(2)` dreimal neu. Der Basisfall der rekursiven Definition ist `fib(1)`. Ist das Argument `n` größer als 1, so berechnet `fib(n)` den Basisfall `fib(n-1)`-mal, ein für große `n` sehr ineffizientes Verfahren. Der `RabbitCounter`-Dienst mutet zwar etwas wunderlich an, die Fibonaccizahlen haben allerdings durchaus allgemeinverständliche Anwendungen. Angenommen, die Schrittlänge eines Menschen betrage einen Meter und er sei in der Lage zwei Meter weit zu springen. Wie viele Möglichkeiten gibt es, um eine Strecke von hundert Metern zurückzulegen? Die Antwort lautet `fib(100)`: 3'314'859'971 Möglichkeiten.

[22] Die `countRabbits()`-Operation des `RabbitCounter`-Dienstes berechnet beim Ermitteln von `fib(n)` keine Werte doppelt, sondern verwendet einen Zwischenspeicher für bereits ermittelte Werte, so daß eine erneute Berechnung durch Nachschlagen vermieden werden kann.

[23] Unser Interesse gilt eigentlich den clientseitigen Behandlern, welche die zum `RabbitCounterService`-Dienst gesendeten Anfragen beziehungsweise die von dort zurückgesendeten Antworten abfangen. Der nächste Schritt ist die Angabe, wo sich die Konfigurationsdatei für die Behandlerkette befindet. Das `wsimport`-Artefakt `fibC.RabbitCounterService` ist eine naheliegende Komponente für einen Verweis auf die Konfigurationsdatei. Die Annotation `@HandlerChain` gibt an, wo sich die Konfigurationsdatei befindet:

```
import javax.jws.HandlerChain;
@WebServiceClient(name = "RabbitCounterService",
                  targetNamespace = "http://ch03.fib",
                  wsdlLocation = "http://localhost:8888/fib?wsdl")
@HandlerChain(file = "handler-chain.xml")
public class RabbitCounterService extends Service {
```

Die Konfigurationsdatei kann überall im Klassenpfad liegen. Es ist somit bequem, sie in dem Verzeichnis *fibC* zu deponieren, das die `wsimport`-Artefakte beinhaltet.

3.1.7 Programmatisches Registrieren eines clientseitigen Behandlers

[24] Das deklarative Registrieren von Behandlern per Konfigurationsdatei ist die bevorzugte aber nicht die einzige Möglichkeit. Die Konfigurationsdatei wird bevorzugt, da der client- beziehungsweise serviceseitige Quelltext dabei relativ „sauber“ bleibt. Behandler liegen am Rand der Applikationslogik, im Gegensatz zum Zentrum und lassen sich daher in der Regel am besten durch Metadatendateien wie *handler-chain.xml* verwalten. Es ist aber nicht schwierig, einen Behandler programmatisch zu registrieren.

[25] Das folgende Beispiel zeigt eine überarbeitete Version des Clients für den `RabbitCounter`-Dienst:

Beispiel 3.5: Client mit programmatisch registriertem Behandler.

```
import fibC.RabbitCounterService;
import fibC.RabbitCounter;
import fibC.UUIDHandler;
import fibC.TestHandler;

import java.util.List;
import java.util.ArrayList;
import javax.xml.ws.handler.Handler;
import javax.xml.ws.handler.HandlerResolver;
import javax.xml.ws.handler.PortInfo;

class FibClientHR {
    public static void main(String[] args) {
        RabbitCounterService service = new RabbitCounterService();
        service.setHandlerResolver(new ClientHandlerResolver());
        RabbitCounter port = service.getRabbitCounterPort();

        try {
            int n = 27;
            System.out.printf("fib(%d) = %d\n", n, port.countRabbits(n));
        }
        catch(Exception e) { System.err.println(e); }
    }
}

class ClientHandlerResolver implements HandlerResolver {
    public List<Handler> getHandlerChain(PortInfo port_info) {
        List<Handler> hchain = new ArrayList<Handler>();
        hchain.add(new UUIDHandler());
        hchain.add(new TestHandler()); // for illustration only
        return hchain;
    }
}
```

Die Datei *FibClientHR.java* enthält zwei Klassen: `FibClientHR` und `ClientHandlerResolver`. Das Interface *`javax.xml.ws.handler.HandlerResolver`* deklariert nur ein einzige Methode namens

`getHandlerChain()`, die von der Laufzeitbibliothek aufgerufen wird, um eine Liste der registrierten Behandler anzufordern. `FibClientHR` ruft die `setHandlerResolver()`-Methode des von der lokalen Variablen `service` referenzierten `RabbitCounterService`-Objektes auf:

```
service.setHandlerResolver(new ClientHandlerResolver());
```

Die Konfigurationsdatei *handler-chain.xml* spielt nun keine Rolle mehr beim Registrieren der Behandler. Statt dessen übernimmt der Quelltext diese Aufgabe. Das Beispiel registriert einen zweiten Behandler vom Typ `TestHandler`, um die intuitive Reihenfolge der Behandler zu veranschaulichen. Der `UUIDHandler`-Behandler steht zuerst in der Behandlerkette und wird zuerst aufgerufen. Die Klasse `TestHandler` gibt übrigens einfach die SOAP-Nachricht zu Beoachtungszwecken über die Standardausgabe aus.

3.1.8 Auswerfen eines SOAP-Faults

[26] Der `RabbitCounter`-Dienst verfügt über einen applikationsspezifischen Ausnahmetyp. Die mit `@WebMethod` annotierten Methode `countRabbits()` wirft eine Ausnahme vom Typ `FibException` aus, wenn sie mit einem negativen Argument aufgerufen wird:

```
@WebMethod
public int countRabbits(int n) throws FibException {
    // Throw a fault if n is negative.
    if (n < 0) throw new FibException("Negative args not allowed.", n + " < 0");
```

Die Implementierung der Ausnahmeklasse `FibException` ist:

```
package ch03.fib;

public class FibException extends Exception {
    private String details;
    public FibException(String reason, String details) {
        super(reason);
        this.details = details;
    }
    public String getFaultInfo() { return details; }
}
```

Wird in der `countRabbits()`-Methode eine Ausnahme vom Typ `FibException` ausgeworfen, so enthält die an den Client zurückgesendete Antwort des Dienstes eine Fehlermeldung anstelle einer gewöhnlichen Ausgabe. Beispielsweise lautet die Antwort auf eine Anfrage mit dem Argument -999:

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope">
  <S:Header />
  <S:Body>
    <ns3:Fault
        xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ns3="http://www.w3.org/2003/05/soap-envelope">
      <ns3:Code><ns3:Value>ns3:Receiver</ns3:Value></ns3:Code>
      <ns3:Reason>
        <ns3:Text xml:lang="en">Negative args not allowed.</ns3:Text>
      </ns3:Reason>
      <ns3:Detail>
        <ns2:FibException xmlns:ns2="http://ch03.fib">
          <faultInfo>-999 < 0</faultInfo>
          <message>Negative args not allowed.</message>
        </ns2:FibException>
```

```
        </ns3:Detail>
        ...
      </ns3:Fault>
    </S:Body>
  </S:Envelope>
```

Die Fehlermeldung beinhaltet die Ursache (`<Reason>`-Element), die dem Konstruktor als erstes Argument übergeben wurde:

```
new FibException("Negative args not allowed.", n + " < 0");
```

sowie eine genaue Information zur Ursache (`<Detail>`-Element) als zweites Argument. Die Fehlermeldung in der Antwort ist ausführlich und detailliert. Der hervorgehobene Ausschnitt zeigt das Wesentliche.

[27] Nach der Inbetriebnahme des Dienstes geht aus dem generierten WSDL-Dokument hervor, daß der Dienst einen SOAP-Fault auswerfen kann. Das WSDL-Dokument enthält ein `<message>`-Element für die Implementierung der Ausnahme. Der `<portType>`-Abschnitt enthält neben den üblichen `<input>`- und `<output>`-Elementen ein `<Fault>`-Element. Die folgenden Zeilen zeigen den betreffenden Ausschnitt aus dem WSDL-Dokument:

```
<message name="FibException">
  <part name="fault" element="tns:FibException" />
</message>
<portType name="RabbitCounter">
  <operation name="countRabbits">
    <input message="tns:countRabbits" />
    <output message="tns:countRabbitsResponse" />
    <fault message="tns:FibException" name="FibException" />
  </operation>
</portType>
```

Das Auswerfen eines SOAP-Faults aus einer mit `@WebMethod` annotierten Methode ist einfach. Die von `Exception` abgeleitete Klasse, die den SOAP-Fault implementiert, sollte über einen zweiargumentigen Konstruktor verfügen. Das erste Argument gibt die Ursache des Fehlers an (hier ein negatives Argument), das zweite Argument zusätzliche Details über die Ursache. Die Ausnahmeklasse sollte eine Methode namens `getFaultInfo()` definieren, welche die Detailinformation über den aufgetretenen Fehler zurückgibt. Die Fehlermeldung des SOAP-Faults umfaßt Ursache und Details.

3.1.9 Ein logischer Behandler verbessert die Robustheit des Clients

[28] Der `RabbitCounter`-Dienst wirft einen SOAP-Fault aus, wenn die Methode `countRabbits()` mit einem negativen Argument aufgerufen wird. Ein clientseitiger logischer Behandler (*`javax.xml.ws.handler.LogicalHandler`*) könnte die ausgehende Anfrage abfangen, das Argument der `countRabbits()`-Methode überprüfen und ein negatives Argument abändern. Die Klasse `ArgHandler` im folgenden Beispiel hat genau diese Aufgabe:

Beispiel 3.6: Ein logischer Behandler erhöht die Robustheit des Clients.

```
package fibC;

import javax.xml.ws.LogicalMessage;
import javax.xml.ws.handler.LogicalHandler;
import javax.xml.ws.handler.LogicalMessageContext;
import javax.xml.ws.handler.MessageContext;
import java.util.logging.Logger;
```

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.JAXBException;

public class ArgHandler implements LogicalHandler<LogicalMessageContext> {

    private static final String LoggerName = "ArgLogger";
    private Logger logger;
    private final boolean log_p = true; // set to false to turn off

    public ArgHandler() {
        logger = Logger.getLogger(LoggerName);
    }

    // If outgoing message argument is negative, make non-negative.
    public boolean handleMessage(LogicalMessageContext ctx) {
        Boolean outbound_p = (Boolean)
            ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        if (outbound_p) {
            if (log_p) logger.info("ArgHandler.handleMessage");
            LogicalMessage msg = ctx.getMessage();

            try {
                JAXBContext jaxb_ctx = JAXBContext.newInstance("fibC");
                Object payload = msg.getPayload(jaxb_ctx);
                if (payload instanceof JAXBElement) {
                    Object obj = ((JAXBElement) payload).getValue();
                    CountRabbits obj_cr = (CountRabbits) obj;
                    int n = obj_cr.getArg0();        // current value
                    if (n < 0) {                     // negative argument?
                        obj_cr.setArg0(Math.abs(n)); // make non-negative

                        // Update the message.
                        ((JAXBElement) payload).setValue(obj_cr);
                        msg.setPayload(payload, jaxb_ctx);
                    }
                }
            }
            catch(JAXBException e) { }
        }
        return true;
    }
    public boolean handleFault(LogicalMessageContext ctx) { return true; }
    public void close(MessageContext ctx) { }
}
```

Die Klasse `ArgHandler` stützt sich auf ein Hilfsobjekt vom Typ `javax.xml.bind.JAXBContext`, um die Nutzdaten aus der Nachricht zu extrahieren. Diese befinden sich im SOAP-Body der ausgehenden Nachricht, gekapselt in einem Objekt vom Typ *`LogicalMessage`*. Das Interface *`LogicalMessage`* deklariert zwei Versionen der `getPayload()`-Methode. Die argumentlose Version von `getPayload()` gibt eine Referenz vom Typ *`javax.xml.transform.Source`* zurück, dessen Inhalt durch Unmarshalling in ein Java-Objekt umgewandelt werden kann. Die einargumentige Version der `getPayload()`-Methode gibt eine Referenz vom Typ `Object` zurück und erwartet ein Argument vom Typ `JAXBContext`, welches entweder eine Klasse oder, wie hier, den Namen eines oder mehrerer Packages repräsentiert. (Das Package *`fibC`* beinhaltet die `wsimport`-Artefakte.) Die Nutzdaten der Nachricht werden in ein Objekt vom Typ `JAXBElement` umgewandelt, welches ein `CountRabbits`-Objekt in XML-Darstellung repräsentiert. Die Klasse `CountRabbits` ist eines der `wsimport`-Artefakte zur Unterstützung des Clients. Insbesondere ist `fibC.CountRabbits` der Java-Typ, der zu einer SOAP-Anfrage für die Operation `countRabbits()` des Dienstes gehört.

[29] Die Klasse `CountRabbits` definiert zwei Methoden `getArg0()` und `setArg0()`, die Zugriff auf das Argument der Operation `countRabbits()` gestatten. Ist dieses Argument eine negative Zahl, so erzeugt der logische Behandler mit Hilfe der statischen `Math`-Methode `abs()` einen nicht-negativen Wert, aktualisiert das `JAXBElement`-Objekt und schließlich den SOAP-Body der Nachricht über einen Aufruf der Methode `setPayload()`. Dieser Behandler macht den Client robuster, indem SOAP-Nachrichten abgefangen werden, die den Dienst andernfalls veranlassen würden, einen SOAP-Fault auszuwerfen.

[30] Die Konfigurationsdatei *handler-chain.xml* wird aktualisiert, um einen zweiten Behandler in der Kette zu deklarieren:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<javaee:handler-chains
    xmlns:javaee="http://java.sun.com/xml/ns/javaee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <javaee:handler-chain>
    <javaee:protocol-bindings>##SOAP12_HTTP</javaee:protocol-bindings>
    <javaee:handler>
      <javaee:handler-class>fibC.UUIDHandler</javaee:handler-class>
    </javaee:handler>
    <javaee:handler>
      <javaee:handler-class>fibC.ArgHandler</javaee:handler-class>
    </javaee:handler>
  </javaee:handler-chain>
</javaee:handler-chains>
```

Die neue Behandlerklasse `ArgHandler` ist absichtlich *nach* `UUIDHandler` angeordnet, um zu betonen, daß die Methoden `handleMessage()` und `handleFault()` eines logischen Behandlers, unabhängig von der Deklarationsreihenfolge, bei ausgehenden Nachrichten stets *vor* den entsprechenden Methoden eines SOAP-Behandlers aufgerufen werden. Bei eingehenden Nachrichten werden die SOAP-Behandlermethoden vor den entsprechenden logischen Behandlermethoden aufgerufen.

[31] Der `RabbitCounter`-Dienst kann sich natürlich nicht auf den clientseitigen logischen Behandler verlassen, prüft nach wie vor auf negative Argumente und wirft gegebenenfalls einen SOAP-Fault aus. Natürlich könnte auch der Webservice selbst negative Argumente in positive umwandeln, aber der gegenwärtige Entwurf des Dienstes gestattet die Beobachtung, auf welche Weise JAX-WS-Behandler auf der Clientseite nützlich sein können.

3.1.10 Registrieren eines serviceseitigen Behandlers

[32] Die clientseitige Behandlerklasse `UUIDHandler` setzt einen Headerblock ein, dessen `actor`-Attribut mit `next` bewertet ist. Diese Einstellung bedeutet, daß jeder Mittler auf dem Weg der Nachricht zwischen Sender und Empfänger den Headerblock verarbeiten und gegebenenfalls einen SOAP-Fault auswerfen soll. Das Auswerfen eines SOAP-Faults ist angemessen, wenn der Mittler den Headerblock nicht in der für die Applikation erforderlichen Weise verarbeiten kann. Die SOAP-Spezifikation erklärt nicht im Detail, welche Art von Verarbeitung eine Applikation von Mittlern und endgültigen Empfängern verlangen kann, sondern gestattet den Sendern lediglich, anzuzeigen, daß die applikationspezifische Verarbeitung von Headerblöcken entlang des Weges zum Empfänger erwartet wird. Bei dieser messaging architecture zählt auch der endgültige Empfänger als `next`-Akteur und unterliegt ebenfalls der Erwartung, den Headerblock zu verarbeiten und erforderlichenfalls einen SOAP-Fault auszuwerfen. Wir benötigen zum Abrunden des Beispiels also einen serviceseitigen Behandler, der den clientseitig in die ausgehende Nachricht eingesetzten Headerblock verarbeitet. Der serviceseitige Behandler zeigt, wie ein Mittler beziehungsweise der endgültige

Empfänger einen Headerblock verarbeiten könnte. Die spezifische Logik der Verarbeitung ist weniger wichtig als die Verarbeitung selbst. Mit der Platzierung client- und serviceseitiger Behandler führt das Beispiel die Grundzüge des JAX-WS-Behandlerframeworks sowohl auf der Sender- als auch auf der Empfängerseite vor.

[33] Die Klasse `ch03.fib.UUIDValidator` im folgenden Beispiel prüft eine beim Dienst eingehende Nachricht. Die Klasse braucht Zugriff auf die gesamte Nachricht, nicht nur deren SOAP-Body und muß daher das Interface *`SOAPHandler`* anstelle von *`LogicalHandler`* implementieren:

Beispiel 3.7: Serviceseitiger Behandler zur Prüfer von Anfragen.

```
package ch03.fib;

import java.util.UUID;
import java.util.Set;
import java.util.Iterator;
import java.util.Locale;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPConstants;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPHeaderElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.Node;
import javax.xml.ws.soap.SOAPFaultException;
import javax.xml.soap.SOAPFault;
import java.io.IOException;

public class UUIDValidator implements SOAPHandler<SOAPMessageContext> {

    private static final boolean trace = false; // make true to see message

    public boolean handleMessage(SOAPMessageContext ctx) {

        Boolean response_p = (Boolean)
            ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

        // Handle the SOAP only if it's incoming.
        if (!response_p) {

            try {

                SOAPMessage msg = ctx.getMessage();
                SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
                SOAPHeader hdr = env.getHeader();

                // Ensure that the SOAP message has a header.
                if (hdr == null)
                    generateSOAPFault(msg, "No message header.");

                // Get UUID value from header block if it's there.
                Iterator it =
                    hdr.extractHeaderElements(SOAPConstants.URI_SOAP_ACTOR_NEXT);
                if (it == null || !it.hasNext())
                    generateSOAPFault(msg, "No header block for next actor.");
                Node next = (Node) it.next();
                String value = (next == null) ? null : next.getValue();
                if (value == null)
```

```
                    generateSOAPFault(msg, "No UUID in header block.");

                // Reconstruct a UUID object to check some properties.
                UUID uuid = UUID.fromString(value.trim());
                if (uuid.variant() != UUIDvariant ||
                    uuid.version() != UUIDversion)
                    generateSOAPFault(msg, "Bad UUID variant or version.");

                if (trace) msg.writeTo(System.out);

            }

            catch(SOAPException e) { System.err.println(e); }
            catch(IOException e) { System.err.println(e); }
        }

        return true; // continue down the chain
    }

    public boolean handleFault(SOAPMessageContext ctx) {
        return true;
    }
    public Set<QName> getHeaders() { return null; }
    public void close(MessageContext messageContext) { }

    private void generateSOAPFault(SOAPMessage msg, String reason) {

        try {

            SOAPBody body = msg.getSOAPPart().getEnvelope().getBody();
            SOAPFault fault = body.addFault();
            fault.setFaultString(reason);

            // wrapper for a SOAP 1.1 or SOAP 1.2 fault
            throw new SOAPFaultException(fault);

        }

        catch(SOAPException e) { }

    }

    private static final int UUIDvariant = 2; // layout
    private static final int UUIDversion = 4; // version

}
```

Die Klasse `UUIDValidator` prüft zuerst, ob die eingehende Nachricht überhaupt einen Header besitzt und, falls ein Header existiert, ob der erwartete Headerblock vorhanden ist, im vorliegenden Fall ein XML-Element dessen `actor`-Attribut mit der SOAP-Rolle `next` bewertet ist:

```
if (hdr == null) // hdr refers to the SOAPHeader
    generateSOAPFault(msg, "No message header.");

Iterator it = hdr.extractHeaderElements(SOAPConstants.URI_SOAP_ACTOR_NEXT);
if (it == null || !it.hasNext())
    generateSOAPFault(msg, "No header block for next actor.");
Node next = (Node) it.next();
String value = (next == null) ? null : next.getValue();
if (value == null)
    generateSOAPFault(msg, "No UUID in header block.");
```

Verläuft die erste Prüfung erfolgreich, so fährt `UUIDValidator` fort und kontrolliert, ob der UUID-Wert die erwarteten Eigenschaften hat. In diesem Beispiel werden die UUID-Versionsnummer und die UUID-Variante geprüft, die den Wert 4 beziehungsweise 2 haben sollte (die UUID-Variante

steuert die Formatierung des UUID-Wertes als Zeichenkette):

```
UUID uuid = UUID.fromString(value.trim());
if (uuid.variant() != UUIDvariant || uuid.version() != UUIDversion)
    generateSOAPFault(msg, "Bad variant or version.");
```

Scheitert eine dieser Prüfungen, so wirft die Klasse `UUIDValidator` eine Ausnahme vom Typ `javax.xml.ws.soap.SOAPFaultException` aus, einem Wrappertyp für SOAP-Faults in den Protokollversionen 1.1 und 1.2. Der Wrappertyp gestattet dem Programmierer, die Unterschiede zwischen beiden SOAP-Versionen unbeachtet zu lassen. Die Methode `generateSOAPFault()` kümmert sich um das Erzeugen und Auswerfen der Ausnahmen:

```
private void generateSOAPFault(SOAPMessage msg, String reason) {
    try {
        SOAPBody body = msg.getSOAPPart().getEnvelope().getBody();
        SOAPFault fault = body.addFault();
        fault.setFaultString(reason);
        throw new SOAPFaultException(fault);
    }
    catch(SOAPException e) { }
}
```

Die Behandlerklasse `UUIDValidator` trägt die Annotation `@HandlerChain` und wird mit Hilfe einer Konfigurationsdatei registriert. Die beiden folgenden Zeilen dokumentieren die erforderlichen Änderungen in der SIB RabbitCounter oder RabbitCounterService:

```
@HandlerChain(file = "handler-chain.xml")
public class RabbitCounter {
```

Die Dateien *handler-chain.xml* und *UUIDValidator.class* liegen im Unterverzeichnis *ch03/fib* des Arbeitsverzeichnisses.

[34] Die Behandlerklasse `UUIDValidator` läßt sich auf verschiedene Arten testen. Beispielsweise könnte die Zeile

```
helem.addTextNode(uuid.toString());
```

in der Klasse `UUIDHandler` auf Seite 84 auskommentiert werden, so daß kein UUID-Wert in den Header eingesetzt wird. In diesem Fall wird die folgende Fehlermeldung erzeugt und von der `handleFault()`-Methode der Klasse `UUIDHandler` abgefangen:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header />
  <S:Body>
    <ns2:Fault
        xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:ns3="http://www.w3.org/2003/05/soap-envelope">
      <faultcode>ns2:Server</faultcode>
      <faultstring>No UUID in header block.</faultstring>
      ...
      <message>No UUID in header block.</message>
      ...
    </ns2:Fault>
  </S:Body>
</S:Envelope>
```

Die vollständige Fehlermeldung ist umfangreich und beinhaltet den aktuellen Zustand des Aufrufstapels.

[35] Ein etwas geistreicherer Test besteht darin, einen UUID-Wert zu wählen, der sich von den erwarteten Eigenschaften unterscheidet. Zu diesem Zweck können Sie die Zeile:

```
UUID uuid = UUID.randomUUID();
```

im Quelltext der Klasse `UUIDHandler` durch die beiden folgenden Zeilen ersetzen:

```
uuid = new UUID(new java.util.Random().nextLong(),   // lower 64 bits
                new java.util.Random().nextLong());  // upper 64 bits
```

Der zweiargumentige Konstruktor der Klasse `UUID` erzeugt ein `UUID`-Objekt mit anderen Eigenschaften als dem von der `randomUUID()`-Methode erzeugten Objekt. Die Behandlerklasse `UUIDValidator` erkennt diesen Unterschied. Die Fehlermeldung lautet in diesem Fall:

```
Bad UUID variant or version.
```

3.1.11 Zusammenfassung der Behandlermethoden

[36] Das Interface *`javax.xml.ws.handler.Handler`* deklariert drei Methoden: `handleMessage()`, `handleFault()` und `close()`. Das Interface *`SOAPHandler`* erweitert *`Handler`* um die Methode `getHeaders()`, die eine Referenz vom Typ *`Set`* auf ein Containerobjekt zurückgibt, welches wiederum Referenzen auf `QName`-Objekte beinhaltet, die SOAP-Headerblöcke repräsentieren. Die `getHeaders()`-Methode kann verwendet werden, um Sicherheitsinformationen in einen SOAP-Header einzusetzen, siehe [Beispiel auf Seite ?? (Kapitel 5)]. In den Behandlerklassen `UUIDHandler` und `UUIDValidator` gibt die `getHeaders()`-Methode einfach den Wert `null` zurück (Seite 85/96). Die anderen drei Methoden haben je einen Parameter vom Typ *`MessageContext`*[1], der den Zugriff auf die eigentliche SOAP-Nachricht gestattet.

[37] Hier zur Veranschaulichung der Ablaufsteuerung die Konfigurationsdatei *handler-chain.xml* der clientseitigen Behandler, die beim Aufruf des Clients `FibClient` ausgeführt werden:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<javaee:handler-chains
    xmlns:javaee="http://java.sun.com/xml/ns/javaee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <javaee:handler-chain>
    <javaee:handler>
      <javaee:handler-class>fibC.TestHandler</javaee:handler-class>
    </javaee:handler>
    <javaee:handler>
      <javaee:handler-class>fibC.UUIDHandler</javaee:handler-class>
    </javaee:handler>
    <javaee:handler>
      <javaee:handler-class>fibC.ArgHandler</javaee:handler-class>
    </javaee:handler>
  </javaee:handler-chain>
</javaee:handler-chains>
```

Die Behandlerklassen `TestHandler` und `UUIDHandler` implementieren das Interface *`SOAPHandler`*, die Behandlerklasse `ArgHandler` dagegen *`LogicalHandler`*. Denken Sie daran, daß bei ausgehenden Nachrichten wie Anfragen vom Client `FibClient` die logischen Behandler vor den SOAP-Behandlern aufgerufen werden. In diesem Beispiel lautet die Reihenfolge der Aufrufe:

[1] Anmerkung des Übersetzers: Genau genommen deklariert das Interface *`javax.xml.ws.handler.Handler<C extends MessageContext>`* für die `close()`-Methode einen Parameter vom Typ *`MessageContext`* und für die beiden Methoden `handleMessage()` und `handleFault()` je einen Parameter des von *`MessageContext`* abgeleiteten Platzhaltertyps `C`.

1. Die `getHeaders()`-Methoden der Klassen `TestHandler` und `UUIDHandler` (beides SOAP-Behandler) werden zuerst aufgerufen.
2. Die `handleMessage()`-Methode der Klasse `ArgHandler` (logischer Behandler) wird unter allen Methoden dieses Namens zuerst aufgeufen.
3. Die `handleMessage()`-Methode der Klasse `TestHandler` wird das nächstes aufgerufen, da die Klasse `TestHandler` als erste in der Konfigurationsdatei deklariert ist.
4. Die `handleMessage()`-Methode der Klasse `UUIDHandler` wird das nächstes aufgerufen.
5. Die `close()`-Methode der Klasse `UUIDHandler` wird aufgerufen, um zu signalisieren, daß der Behandler `UUIDHandler` die Verarbeitung beendet hat.
6. Die `close()`-Methode der Klasse `TestHandler` wird aufgerufen, um zu signalisieren, daß der Behandler `TestHandler` die Verarbeitung beendet hat.
7. Die `close()`-Methode der Klasse `ArgHandler` wird aufgerufen. Damit ist die Verarbeitung der Behandlerkette für die ausgehende Nachricht abgeschlossen.

Das JAX-WS-Behandlerframework bietet dem Programmierer Gelegenheiten zum Eingriff in die Nachrichtenverarbeitung, so daß ein- und ausgehende Nachrichten abgefangen und in einer der Applikation dienlichen Weise verarbeitet werden können. Die Rückrufmethode `handleMessage()` ist besonders praktisch, da sie den Zugriff entweder auf die gesamte SOAP-Nachricht (beim Typ *`SOAPHandler`*) oder selektiv auf die Nutzdaten (beim Typ *`LogicalHandler`*) gewährt.

3.2 Anpassung des RabbitCounter-Dienstes an SOAP-Version 1.2

[38] Die Anpassung der Klasse `RabbitCounter` von SOAP-Version 1.1 nach Version 1.2 erfordert nur wenige Schritte. Der wesentliche Schritt ist das Einsetzen der Annotation `@BindingType` in die SIB:

```
import javax.xml.ws.BindingType;
@BindingType(value="http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/")
public class RabbitCounter {
```

Die JAX-WS definiert zwar die Konstante

```
javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING
```

für die Standardbindung bei SOAP 1.2, aber der *`Endpoint`-Publisher* kann mit der Standardbindung kein WSDL-Dokument generieren. Der Workaround ist der obige nichtstandardisierte Bindungswert. Bei der Inbetriebnahme des Dienstes, gibt der `Endpoint`-Publisher die folgende unkritische Warnung aus:

```
com.sun.xml.internal.ws.server.EndpointFactory generateWSDL
WARNING: Generating non-standard WSDL for the specified binding
```

Das `wsgen`-Kommando wird wie zuvor verwendet, während `wsimport` mit dem Schalter `-extension` aufgerufen wird, der anzeigt, daß die clientseitigen Artefakte aus einem vom Standard abweichenden WSDL-Dokument generiert werden:

```
% wsimport -keep -extension -p fibC2 http://localhost:8888/fib?wsdl
```

Die Klasse `UUIDHandler` läßt sich ebenfalls ändern, um eine Eigenschaft von SOAP 1.2 zu nutzen, nämlich das Headerblockattribut `mustUnderstand`. Die neue Zeile wird direkt nach dem Aufruf der `setActor()`-Methode in der `UUIDHandler`-Version zu SOAP 1.1 eingesetzt:

```
helem.setActor(SOAPConstants.URI_SOAP_ACTOR_NEXT);
helem.setMustUnderstand(true);  // SOAP 1.2
```

Die resultierende Anfrage lautet (der eingesetzte Header ist hervorgehoben):

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope">
  <S:Header>
    <uuid
        xmlns="http://ch03.fib"
        xmlns:env="http://www.w3.org/2003/05/soap-envelope"
        env:mustUnderstand="true"
        env:role="http://schemas.xmlsoap.org/soap/actor/next">
      b50064b5-24e8-42bc-9716-10537c86bbd8
    </uuid>
  </S:Header>
  <S:Body>
    <ns2:countRabbits xmlns:ns2="http://ch03.fib">
      <arg0>-45</arg0>
    </ns2:countRabbits>
  </S:Body>
</S:Envelope>
```

Das *mustUnderstand-Attribut* verlangt ausdrücklich, daß jeder Mittler und der endgültige Empfänger den Headerblock in einer der Applikation dienlichen Weise verarbeiten.

[39] Der `<binding>`-Abschnitt des vom `Endpoint`-Publisher generierten WSDL-Dokumentes reflektiert die Änderung von SOAP 1.1 zu SOAP 1.2:

```
<binding name="RabbitCounterPortBinding" type="tns:RabbitCounter">
  <soap12:binding
    transport="http://www.w3.org/2003/05/soap/bindings/HTTP/"
    style="document" />
  <operation name="countRabbits">
    <soap12:operation soapAction="" />
    <input>
      <soap12:body use="literal" />
    </input>
    <output>
      <soap12:body use="literal" />
    </output>
    <fault name="FibException">
      <soap12:fault name="FibException" use="literal" />
    </fault>
  </operation>
</binding>
```

Angesichts der relativ nachrangigen Unterschiede zwischen SOAP 1.1 und SOAP 1.2 und dem Status von SOAP 1.1 als *de facto*-Standard, ist es sinnvoll, bei Version 1.1 zu bleiben, sofern keine zwingenden Gründe bestehen, um SOAP 1.2 zu verwenden. Letztendlich ist SOAP 1.2 abwärts kompatibel mit SOAP 1.1.

3.3 Zugriff auf Transportheader über den Nachrichtenkontext

[40] Dieser Abschnitt betrachtet die Zusammenarbeit zwischen Dienst- und Transportebene. Der Schwerpunkt liegt auf dem *Nachrichtenkontext*, der durch das Interface `javax.xml.ws.handler.MessageContext` repräsentiert wird. Der Zugriff auf den Nachrichtenkontext erfolgt in der Regel

über Behandler mit den von *MessageContext* abgeleiteten Interfaces *SOAPMessageContext* und *LogicalMessageContext* als Parametertypen entsprechender Methoden, zum Beispiel der Rückrufmethode `handleMessage()`, sowohl bei SOAP- als auch bei logischen Behandlern.

[41] Das Konzept des *Kontextes* kommt bei modernen Programmiersprachen und Bibliotheken vor, darunter auch Java. Servlets haben das Interface *javax.servlet.ServletContext*, EJBs (Enterprise JavaBeans) *javax.ejb.EJBContext* (mit entsprechenden abgeleiteten Typen wie *javax.ejb.SessionContext*) und Webservices *javax.xml.ws.WebServiceContext*. Aus der Architekturperspektive betrachtet, gewährt der Kontext einem Objekt (einem Servlet, einer EJB oder einem Webservice) Zugriff auf den unterliegenden *Container* (Servletcontainer, EJB-Container oder Webservicecontainer). Der Container wiederum, implementiert „unter der Haube“ die Unterstützung dieser Objekte. Aus der Perspektive des Programmierers ist der Kontext ein Container vom Typ *Map<String,Object>* dessen Schlüssel Zeichenketten sind und auf beliebige Objekte verweisen.

[42] Die Applikationsebene eines Dienstes, bestehend aus SEI und SIB, setzt den unterliegenden Nachrichtenkontext in der Regel als gegebene, unsichtbare Infrastruktur voraus. Auf der Behandlerebene ist der Nachrichtenkontext als Parameter der Rückrufmethoden verfügbar, so daß der SOAP- oder logische Behandler die Nachrichten oder Nutzdaten erreichen kann. Dieser Abschnitt behandelt die eher ungewöhnliche Situation des Zugriffs auf den Nachrichtenkontext *außerhalb eines Behandlers*, das heißt in einer Hauptkomponente der Applikation, nämlich der *SIB* und deren *Clients*.

[43] SOAP-Nachrichten werden vorwiegend über HTTP versendet. Damit erhebt sich die Frage, in welchem Umfang die HTTP-Infrastruktur durch den Nachrichtenkontext eines Java-basierten Webservice zugänglich ist. Dieselbe Frage stellt sich auch für alternative Transportprotokolle wie SMTP oder JMS.

[44] In einer Behandlerklasse oder SIB gewährt Java über das Interface *MessageContext* Zugriff auf HTTP-Nachrichten. In einem Java-basierten Client gewährt Java ebenfalls Zugriff auf HTTP, hier allerdings über das Interface *javax.xml.ws.BindingProvider* und die Anfrage-/Antwortkontexte, die als *BindingProvider*-Eigenschaften zur Verfügung stehen. Die Beispiele zeigen den Zugriff auf die transportierten Nachrichten auf der Applikations- im Gegensatz zur Behandlerebene.

3.3.1 Der Echo-Dienst

[45] Der `Echo`-Dienst im Beispiel dieses Abschnitts gibt lediglich die Textmeldung des Clients an den Client zurück. Der Dienst ist bewußt einfach gestaltet, da der Fokus auf der Transportebene liegt. Das gesamte Beispiel verwendet HTTP als Transportprotokoll für die SOAP-Nachrichten („SOAP over HTTP“):

Beispiel 3.8: Ein Dienst mit Zugriff auf den Nachrichtenkontext.

```
package ch03.mctx;

import java.util.Map;
import java.util.Set;
import javax.annotation.Resource;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.jws.HandlerChain;

/**
  * A minimalist service to explore the MessageContext.
```

```
  * The operation takes a string and echoes it together
  * with transport information back to the client.
  */
@WebService
@HandlerChain(file = "handler-chain.xml")
public class Echo {

    // Enable 'dependency injection' on web service context
    @Resource
    WebServiceContext ws_ctx;

    @WebMethod
    public String echo(String from_client) {
        MessageContext ctx = ws_ctx.getMessageContext();
        Map req_headers = (Map) ctx.get(MessageContext.HTTP_REQUEST_HEADERS);
        MapDump.dump_map((Map) ctx, """);
        String response = "Echoing your message: " + from_client;
        return response;
    }

}
```

Die Klasse `Echo` hat ein Feld namens `ws_ctx` vom Typ *`WebServiceContext`*, das mit `@Resource` annotiert ist. Diese Annotation wird verwendet, um *Dependency Injection* anzufordern (ein Konzept aus der Aspektorientierten Programmierung, kurz AOP). Wie jedes nicht initialisierte Feld erhält `ws_ctx` *zunächst* den Standardwert `null`. Dennoch wird in der ersten Zeile der `echo()`-Methode die `getMessageContext()`-Methode des Objektes aufgerufen, das `ws_ctx` nun auf geheimnisvolle Weise referenziert. Selbstverständlich ist keine Zauberei im Spiel. Der Webservicecontainer erzeugt ein Objekt vom Typ *`WebServiceContext`*, setzt es in die Applikation ein und sorgt dafür, daß das `ws_ctx`-Feld auf dieses Objekt verweist. Dieses Objekt dient zum Anfordern einer Referenz auf den Nachrichtenkontext, der wiederum verwendet wird, um eine Referenz auf einen Container vom Typ *`Map`* mit den Transportheadern (typischerweise HTTP-Header) abzufragen. Die `dump_map()`-Methode der folgenden Hilfsklasse `MapDump`:

```
package ch03.mctx;

import java.util.Map;
import java.util.Set;

public class MapDump {

    public static void dump_map(Map map, String indent) {
        Set keys = map.keySet();
        for (Object key : keys) {
            System.out.println(indent + key + " ==> " + map.get(key));
            if (map.get(key) instanceof Map)
                dump_map((Map) map.get(key), indent += "   ");
        }
    }

}
```

wird aufgerufen, um die HTTP-Header auszugeben. Das nächste Beispiel zeigt den betreffenden Teil der Ausgabe einer Anfrage an den `Echo`-Dienst:

Beispiel 3.9: Ausgabe eines HTTP-Nachrichtenkontextes.

```
javax.xml.ws.wsdl.port ==> {http://mctx.ch03/}EchoPort
javax.xml.ws.soap.http.soapaction.uri ==> "echo"
com.sun.xml.internal.ws.server.OneWayOperation ==> null
javax.xml.ws.http.request.pathinfo ==> null
```

```
com.sun.xml.internal.ws.api.message.packet.outbound.transport.headers ==>
  com.sun.net.httpserver.Headers@0
    com.sun.xml.internal.ws.client.handle ==> null
    javax.xml.ws.wsdl.service ==> {http://mctx.ch03/}EchoService
    javax.xml.ws.reference.parameters ==> []
    javax.xml.ws.http.request.headers ==>
      sun.net.httpserver.UnmodifiableHeaders@2c47bd03
      Host ==> [localhost:9797]
      Content-type ==> [text/xml; charset=utf-8]
      Accept-encoding ==> [gzip]
      Content-length ==> [198]
      Connection ==> [keep-alive]
      Greeting ==> [Hello, world!]
      User-agent ==> [Java/1.6.0_06]
      Accept ==> [text/xml, multipart/related, text/html, image/gif,
                  image/jpeg, *; q=.2, */*; q=.2]
      Soapaction ==> [”echo”]
      ...
javax.xml.ws.http.request.method ==> POST
```

Die hervorgehobenen Zeilen sind die HTTP-Anfrageheader des Aufrufs der Operation `echo()`. Die clientseitig unterliegenden Bibliotheken setzen einige standardisierte Schlüssel/Wert-Paare in den Headerabschnitt der HTTP-Anfrage ein, zum Beispiel:

```
Host ==> [localhost:9797]
Content-type ==> [text/xml; charset=utf-8]
```

Bei HTTP-Headern sind Schlüssel und Wert durch einen Doppelpunkt voneinander getrennt. Bei Java wird zur Klarheit das Pfeilsymbol `==>` verwendet. Zwei der Einträge in diesem Beispiel stammen vom Client des `Echo`-Dienstes:

```
Accept-encoding ==> [gzip]
Greeting ==> [Hello, world!]
```

Der erste Eintrag ist der HTTP-Standardheader `Accept-encoding` mit dem Wert `gzip` und zeigt an, daß der Client bereit ist, im Körper der HTTP-Antwort einen `gzip`-komprimierten SOAP-Envelope entgegenzunehmen. Der zweite Eintrag verwendet den frei erfundenen Schlüssel `Greeting`. HTTP 1.1, die aktuelle Protokollversion, gestattet beliebige Schlüssel/Wert-Paare als HTTP-Header.

[46] Ein weiterer HTTP-Header verdient besondere Aufmerksamkeit:

```
Soapaction ==> [”echo”]
```

Bei einem Java-Client wäre die leere Zeichenkette der Standardwert des `Soapaction`-Attributes. Die Ausgabe `echo` kommt dadurch zustande, daß der Client diesen Wert in den Headerabschnitt der HTTP-Anfrage einsetzt (siehe Quelltext der Klasse `EchoClient`).

[47] Die JAX-WS-Laufzeitbibliothek verarbeitet die zur Transportschicht gehörigen Header, die vom Client oder Dienst eingesetzt werden. Der folgende Client `EchoClient` setzt den Schlüssel `Accept-Encoding` (mit großem „E“ bei `Encoding`) ein, während die JAX-WS-Laufzeitbibliothek die Schreibweise in `Accept-encoding` (mit kleinem „e“) ändert. Nun der Quelltext des Clients:

Beispiel 3.10: Ein Client mit Zugriff auf den Nachrichtenkontext.

```
import java.util.Map;
import java.util.Set;
import java.util.List;
import java.util.Collections;
import java.util.HashMap;
import javax.xml.ws.BindingProvider;
```

```
import javax.xml.ws.handler.MessageContext;
import echoC.EchoService;
import echoC.Echo;

class EchoClient {

    public static void main(String[] args) {

        EchoService service = new EchoService();
        Echo port = service.getEchoPort();

        Map<String, Object> req_ctx =
            ((BindingProvider) port).getRequestContext();

        // Sample invocation:
        //
        // % java EchoClient http://localhost:9797 echo
        //
        // 1st command-line argument ends with service location port number
        // 2nd command-line argument is the service operation
        if (args.length >= 2) {

            // Endpoint address becomes: http://localhost:9797/echo
            req_ctx.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                        args[0] + "/" + args[1]);

            // SOAP action becomes: echo
            req_ctx.put(BindingProvider.SOAPACTION_URI_PROPERTY, args[1]);

        }

        // Add some application-specific HTTP headers
        Map<String, List<String>> my_header =
            new HashMap<String, List<String>>();
        my_header.put("Accept-Encoding", Collections.singletonList("gzip"));
        my_header.put("Greeting", Collections.singletonList("Hello, world!"));

        // Insert customized headers into HTTP message headers
        req_ctx.put(MessageContext.HTTP_REQUEST_HEADERS, my_header);

        dump_map(req_ctx, "");
        System.out.println("\n\nRequest above, response below\n\n");

        // Invoke service operation to generate an HTTP response.
        String response = port.echo("Have a nice day :)");

        Map<String, Object> res_ctx =
            ((BindingProvider) port).getResponseContext();
        dump_map(res_ctx, "");

        Object response_code =
            res_ctx.get(MessageContext.HTTP_RESPONSE_CODE);

    }

    private static void dump_map(Map map, String indent) {
        Set keys = map.keySet();
        for (Object key : keys) {
            System.out.println(indent + key + " ==> " + map.get(key));
            if (map.get(key) instanceof Map)
                dump_map((Map) map.get(key), indent += "   ");
        }
    }
}
```

Die Ausgabe in Beispiel 3.9 stammt vom folgenden Aufruf des Clients

```
% java EchoClient http://localhost:9797/ echo
```

Die beiden auf der Kommandozeile übergebenen Argumente werden in der Klasse `EchoClient` zur Adresse des Dienstendpunktes verknüpft. Das zweite Kommandozeilenargument, der Name der Operation `echo()`, wird als Wert der Eigenschaft `SOAPACTION_URI_PROPERTY` im Headerabschnitt der HTTP-Anfrage angelegt.

[48] Die Klasse `EchoClient` erhält über das von der lokalen Variablen `port` referenzierte `Echo`-Objekt Zugriff auf den Headerabschnitt der HTTP-Anfrage. Die Referenz in `port` wird zuvor in den Typ *`BindingProvider`* umgewandelt, um die Methode `getRequestContext()` aufrufen zu können. Diese Methode gibt eine Referenz auf einen Container vom Typ *`Map`* zurück, der zwei Einträge beinhaltet. Erstens, wird die Eigenschaft `SOAPACTION_URI_PROPERTY` mit der Zeichenkette `echo` bewertet. Zweitens, wird die Eigenschaft `ENDPOINT_ADDRESS_PROPERTY` mit der URL *http://localhost:9797/echo* verknüpft, um zu demonstrieren, wie sich die Endpunktadresse des Dienstes im Client dynamisch zusammensetzen läßt.

[49] Die lokale Variable `my_header` verweist auf einen leeren Container vom Typ *`Map`*, der `String`-Referenzen als Schlüssel und `Object`-Referenzen als Werte akzeptiert. Die tatsächlichen Werte sind Referenzen vom Typ *`List<String>`*. Die statische `singletonList()`-Methode der Hilfsklasse `Collections` gibt eine Referenz dieses Typs zurück. Die zusätzlichen Schlüssel/Wert-Paare werden in den Headerabschnitt der HTTP-Anfrage eingesetzt. Die Operation `echo()` des Dienstes wird vom Client zwar nur einmal aufgerufen, aber jeder Aufruf bewirkt eine HTTP-Anfrage mit den zusätzlichen Headern.

[50] Gegen Ende wird die Referenz in `port` nochmals in den Typ *`BindingProvider`* umgewandelt, um die Methode `getResponseContext()` aufrufen zu können, die eine Referenz auf den Kontext der HTTP-Antwort zurückgibt (wiederum ein Containerobjekt vom Typ *`Map`*). Das Interface *`MessageContext`* deklariert zahlreiche Konstanten, darunter `HTTP_RESPONSE_CODE`, die nützlich sind, um Informationen aus dem Headerabschnitt der HTTP-Antwort abzufragen.

[51] Der `Echo`-Dienst verwendet den folgenden SOAP-Behandler. Es ist bemerkenswert, daß Dienst, Client und SOAP-Behandler Zugriff auf den Nachrichtenkontext haben, auch wenn die Syntax für den Zugriff auf diesen Kontext unter den dreien geringfügig variiert.

Beispiel 3.11: Ein serviceseitiger Behandler mit Zugriff auf den Nachrichtenkontext.

```
package ch03.mctx;

import java.util.Map;
import java.util.Set;
import java.util.Locale;
import javax.xml.namespace.QName;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class EchoHandler implements SOAPHandler<SOAPMessageContext> {

    public boolean handleMessage(SOAPMessageContext ctx) {

        // Is this an inbound message, i.e., a request?
        Boolean response_p = (Boolean)
            ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

        // Manipulate the SOAP only if it's incoming.
        if (!response_p) MapDump.dump_map((Map) ctx, "");
        return true; // continue down the chain
```

```
        }

        public boolean handleFault(SOAPMessageContext ctx) { return true; }
        public Set<QName> getHeaders() { return null; }
        public void close(MessageContext messageContext) { }
    }
```

Wir nehmen den Zugriff auf die Transportebene aus der Applikationsebene, insbesondere auf HTTP- und äquivalente Header, in einigen späteren Beispielen wieder auf. HTTP-Header sind beispielsweise eine Stelle zur Aufbewahrung von Berechtigungsnachweisen bei Diensten, welche die Authentifizierung ihrer Clients verlangen.

3.4 Webservices und Transport byteorientierter Daten

[52] Die SOAP-Nachrichten der bisherigen Beispiele enthielten *zeichenorientierte Daten*, die bei Bedarf in dienstspezifische Typen umgewandelt wurden. Die Typumwandlung geschieht größtenteils automatisch und innerhalb der JAX-WS-Infrastruktur ohne Eingriff der eigentlichen Applikation. Die folgenden Zeilen zeigen exemplarisch den SOAP-Body einer Anfrage zum Aufruf der Operation `countRabbits()`. Das Argument „45“ steht als zeichenorientierter Wert in der Nachricht:

```
<S:Body>
  <ns2:countRabbits xmlns:ns2="http://ch03.fib">
    <arg0>45</arg0>
  </ns2:countRabbits>
</S:Body>
```

wird aber automatisch nach `int` konvertiert, so daß die `countRabbits()`-Methode des Dienstes:

```
@WebMethod
public int countRabbits(int n) throws FibException {
```

die *Fibonaccizahl* des ganzzahligen Argumentes `n` bestimmen und zurückgeben kann. Weder der Client `FibClient` noch der `RabbitCounter`-Dienst führen eine explizite Typumwandlung durch.

[53] Auf der Behandlerebene dagegen, finden einige explizite Typumwandlungen statt. Beispielsweise führen die clientseitige Behandlerklasse `UUIDHandler` und die serviceseitige Behandlerklasse `UUIDValidator` explizite, wenn auch einfache Typumwandlungen aus. Die Klasse `UUIDHandler` wandelt das `UUID`-Objekt in ein `String`-Objekt um:

```
helem.addTextNode(uuid.toString());
```

Die Klasse `UUIDValidator` nimmt die Umwandlung in entgegengesetzter Richtung vor:

```
UUID uuid = UUID.fromString(value.trim());
```

Die clientseitige Behandlerklasse `ArgHandler` verrichtet im Hinblick auf Typumwandlungen die meiste Arbeit:

```
JAXBContext jaxb_ctx = JAXBContext.newInstance("fibC");
Object payload = msg.getPayload(jaxb_ctx);
if (payload instanceof JAXBElement) {
    Object obj = ((JAXBElement) payload).getValue();
    CountRabbits obj_cr = (CountRabbits) obj;
    int n = obj_cr.getArg0();        // current value
    if (n < 0) {                     // negative argument?
        obj_cr.setArg0(Math.abs(n)); // make non-negative

        // Update the message.
```

```
            ((JAXBElement) payload).setValue(obj_cr);
            msg.setPayload(payload, jaxb_ctx);
        }
    }
```

`ArgHandler` stützt sich auf JAX-B, um aus einem XML-Element ein `CountRabbits`-Objekt zu bekommen. Der logische Behandler ruft die Methode `getArg0()` und bei Bedarf auch `setArg0()` auf, um sicherzustellen, daß das Argument der `countRabbits()`-Methode nicht negativ ist. Schließlich konvertiert die Methode `setPayload()` das `CountRabbits`-Objekt wieder zurück in ein XML-Element.

[54] Typumwandlungen treten angesichts der Frage in den Vordergrund, wie byteorientierte Daten, zum Beispiel Bilder oder Filme, Operationen von Webservices als Argument übergeben oder als deren Rückgabewert verwendet werden können. SOAP-basierte Webservices sind keineswegs auf zeichenorientierte Daten beschränkt, die Kombination mit byteorientierten Daten führt aber zu wichtigen Effizienzüberlegungen.

[55] Im Allgemeinen werden zwei Möglichkeiten in Betracht gezogen, um byteorientierte Daten mit einem SOAP-basierten Webservice zu übertragen:

- Byteorientierten Daten können nach einem Verfahren wie Base64 kodiert und anschließend als Nutzdaten im SOAP-Body übertragen werden. Eine Operation, die ein Bild an den Client zurücksendet, könnte zum Beispiel den Rückgabetyp `java.awt.Image` verwenden, einen Java-Wrappertyp für Graphiken. Die Graphikbytes würden in diesem Fall kodiert und im SOAP-Body einer Nachricht übertragen werden. Der Nachteil besteht darin, daß Base64 und ähnliche Kodierungsverfahren das Volumen der Nutzlast, verglichen mit dem unkodierten Zustand, um etwa ein Drittel erhöhen. Kurz, das Kodieren byteorientierter Daten nach einem Verfahren wie Base64 bläht die Datenmenge auf.

- Byteorientierten Daten können in Form eines oder mehrerer Anhänge einer SOAP-Nachricht transportiert werden. Eine SOAP-Nachricht besteht aus einem SOAP-Envelope, der einen optionalen SOAP-Header und einen verpflichtenden, eventuell aber leeren SOAP-Body beinhaltet. Darüberhinaus kann eine SOAP-Nachricht Anhänge haben, die Daten beliebiger MIME-Typen beinhalten, darunter Multimediatypen wie `audio/x-wav`, `video/mpeg` und `image/jpeg`. JAX-B liefert die erforderlichen Bindungen (Verknüpfungen) zwischen MIME- und Java-Typ: Die MIME-Typen `image/*` werden an `java.awt.Image` und die übrigen Multimediatypen an `javax.activation.DataHandler` gebunden.

Anhänge haben gegenüber der Kodierung den Vorzug, daß die Volumenvergrößerung vermieden wird, da die Bytes unkodiert vom Sender an den Empfänger übertragen werden. Der Nachteil besteht darin, daß der Empfänger die „rohen“ Bytes interpretieren muß, zum Beispiel durch Rückwärtsumwandlung in Multimediatypen für Bilder und Musik.

3.4.1 Drei Alternativen zur Realisierung von SOAP-Anhängen

[56] Es gibt grundsätzlich drei Alternativen zur Realisierung von SOAP-Anhängen: SwA (SOAP with Attachments), die ursprüngliche SOAP-Spezifikation für Anhänge, DIME (Direct Internet Message Encapsulation), ein leichtgewichtiges, mittlerweile aber veraltetes Kodierungsverfahren für Anhänge und *MTOM* (SOAP Message Transmission Optimization Mechanism), basierend auf *XOP* (XML-binary Optimized Packaging). Die JAX-WS hat eine DIME-Erweiterung, deren Aufgabe hauptsächlich in der Zusammenarbeit mit Microsoft-Clients besteht. Bis zum Erscheinen von Microsoft Office 2003 konnte ein in VBA (Visual Basic for Applications) geschriebener Webservice-Client nur mit

DIME-Anhängen umgehen, nicht aber mit MTOM-Anhängen. Der SwA-Ansatz hat diverse Nachteile. Es ist beispielsweise schwierig, SwA mit einem Dienst im Dokumentstil zu kombinieren, heute der Regelfall. Außerdem wird SwA von Frameworks wie .Net nicht unterstützt. MTOM trägt das Anerkennungssiegel des W3C, genießt weitverbreitete Unterstützung und ist daher ein effizientes, modernes, interoperables Verfahren, um byteorientierte Daten über SOAP-basierte Webservices zu transportieren. Bevor wir uns MTOM zuwenden, wollen wir kurz die Base64-Kodierung byteorientierter Daten als Alternative für Nutzdaten mit geringem Volumen betrachten.

3.4.2 Base64-Kodierung für kleine Datenmengen

[57] Der `SkiImageService`-Dienst im folgenden Beispiel hat zwei Operationen: `getImage()` erwartet die Angabe eines Skistils und gibt ein Bild mit einem oder mehreren Skiläufern zurück, während `getImages()` eine Liste der verfügbaren Bilder liefert:

Beispiel 3.12: Der SkiImageService-Dienst liefert Bilder von Skistilen.

```
package ch03.image;

import javax.jws.WebService;
import javax.jws.WebMethod;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.awt.Image;
import java.io.FileInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.imageio.stream.ImageInputStream;
import javax.imageio.ImageReader;
import javax.jws.HandlerChain;

@WebService(serviceName = "SkiImageService")
@HandlerChain(file = "handler-chain.xml") // for message tracking
public class SkiImageService {

    // Returns one image given the image's name.
    @WebMethod
    public Image getImage(String name) { return createImage(name);  }

    // Returns a list of all available images.
    @WebMethod
    public List<Image> getImages() { return createImageList(); }

    public SkiImageService() {
        photos = new HashMap<String, String>();
        photos.put("nordic", "nordic.jpg");
        photos.put("alpine", "alpine.jpg");
        photos.put("telemk", "telemk.jpg");
        default_key = "nordic";
    }

    // Create a named image from the raw bytes.
     private Image createImage(String name) {
        byte[] bytes = getRawBytes(name);
```

```
        ByteArrayInputStream in = new ByteArrayInputStream(bytes);
        Iterator iterators = ImageIO.getImageReadersByFormatName("jpeg");
        ImageReader iterator = (ImageReader) iterators.next();
        try {
            ImageInputStream iis = ImageIO.createImageInputStream(in);
            iterator.setInput(iis, true);
            return iterator.read(0);
        }
        catch(IOException e) {
            System.err.println(e);
            return null;
        }
    }

    // Create a list of all available images.
    private List<Image> createImageList() {
        List<Image> list = new ArrayList<Image>();
        Set<String> key_set = photos.keySet();
        for (String key : key_set) {
            Image image = createImage(key);
            if (image != null) list.add(image);
        }
        return list;
    }

    // Read the bytes from the file for one image.
    private byte[] getRawBytes(String name) {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        try {
            String cwd = System.getProperty ("user.dir");
            String sep = System.getProperty ("file.separator");
            String base_name = cwd + sep + "jpegs" + sep;
            String file_name = base_name + name + ".jpg";
            FileInputStream in = new FileInputStream(file_name);

            // Send default image if there's none with this name.
            if (in == null) in = new FileInputStream(base_name + "nordic.jpg");
            byte[] buffer = new byte[2048];
            int n = 0;
            while ((n = in.read(buffer)) != -1)
                out.write(buffer, 0, n); // append to ByteArrayOutputStream
            in.close();
        }
        catch(IOException e) { System.err.println(e); }
        return out.toByteArray();
    }

    private static final String[] names = {
        "nordic.jpg", "tele.jpg", "alpine.jpg" };
    private Map<String, String> photos;
    private String default_key;

}
```

Der Quelltext des Dienstes besteht zum größten Teil aus Hilfsmethoden, welche die Bytes des Bildes aus einer lokalen Datei einlesen und daraus ein `Image`-Objekt erzeugen. Die Dateien befinden sich im Unterverzeichnis *jpegs* des Arbeitsverzeichnisses. Die beiden Operationen des Dienstes, `getImage()` und `getImages()`, sind unkompliziert. Die Operation `getImage()` gibt eine Referenz auf ein `Image`-Objekt zurück, `getImages()` eine Referenz auf einen Container vom Typ *`List<Image>`*. An

dieser Stelle ist von Interesse, daß die Rückgabetypen abstrakt und keine Arrays von Bytes sind.

[58] Der `<binding>`-Abschnitt des WSDL-Dokumentes:

```
<binding name="SkiImageServicePortBinding" type="tns:SkiImageService">
  <soap:binding
      transport="http://schemas.xmlsoap.org/soap/http"
      style="document"></soap:binding>
  <operation name="getImage">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
  <operation name="getImages">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
</binding>
```

zeigt, daß der Dienst Dokumentstil mit literaler Zeichenkodierung hat. Zwei Abschnitte aus dem zugehörigen XML-Schema liefern weitere Informationen:

```
<xs:complexType name="getImagesResponse">
  <xs:sequence>
    <xs:element
        name="return" type="xs:base64Binary"
        minOccurs="0" maxOccurs="unbounded"></xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="getImageResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:base64Binary" minOccurs="0">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Das XML-Schema weist aus, daß der Dokumentstil in seiner verpackten Variante vorliegt. Eines der Verpackungselemente ist `<getImageResponse>`. Der Inhalt dieses Verpackungselementes ist erwartungsgemäß der in der Sprache XML-Schema vordefinierte Typ `base64Binary`.

[59] Hier der SOAP-Envelope aus der Antwort auf eine Anfrage nach einem der Bilder:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getImageResponse xmlns:ns2="http://image.ch03/">
      <return>iVBORw0KGgoAAAANSUhEUgAAAZAAAAEsCAIAAABi1X...</return>
    </ns2:getImageResponse>
  </S:Body>
</S:Envelope>
```

Das gesamte Bild beziehungsweise die Liste der Bilder wird Base64-kodiert im SOAP-Body des SOAP-Envelopes zurückgegeben.

[60] Nun der Quelltext des Clients `SkiImageClient`:

```
import skiC.SkiImageService_Service;
import skiC.SkiImageService;
import java.util.List;

class SkiImageClient {

    public static void main(String[] args) {

        // wsimport-generated artifacts
        SkiImageService_Service service = new SkiImageService_Service();
        SkiImageService port = service.getSkiImageServicePort();

        // Note the return types: byte[] and List<byte[]>
        byte[] image = port.getImage("nordic");
        List<byte[]> images = port.getImages();

        /* Transform the received bytes in some useful way :) */

    }

}
```

Der Client stützt sich auf `wsimport`-Artefakte, um die Operationen des Dienstes aufzurufen. Der Client bekommt allerdings entweder ein Array von Bytes oder eine Liste solcher Arrays, da der in XML-Schema vordefinierte Typ `base64Binary` auf den Java-Typ `byte[]` abgebildet wird. Der Client erhält die Base64-Kodierung der Bilder als Arrays von Bytes und muß diese Kodierung anschließend in Bilder zurückumwandeln. Diese Vorgehensweise ist lästig.

[61] Die Abhilfe besteht darin, das WSDL-Dokument so zu ändern, daß sich der Dienst den Clients gegenüber komfortabler verhält. Nach dem Sichern des WSDL-Dokumentes und seines XML-Schemas in lokalen Dateien (zum Beispiel *ski.wsdl* und *ski.xsd*) können die folgenden Änderungen vorgenommen werden:

- Ändern Sie das XML-Schema. Insbesondere sollten Sie die beiden hervorgehobenen Änderungen vornehmen:

```
<xs:complexType name="getImagesResponse">
  <xs:sequence>
    <xs:element
        name="return" type="xs:base64Binary"
        minOccurs="0" maxOccurs="unbounded"
        xmime:expectedContentTypes="image/jpeg"
        xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="getImageResponse">
  <xs:sequence>
    <xs:element
        name="return" type="xs:base64Binary"
        minOccurs="0"
        xmime:expectedContentTypes="image/jpeg"
        xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Das Attribut `expectedContentTypes` ist mit dem MIME-Typ `image/jpeg` bewertet, so daß das `wsimport`-Kommando Versionen der Operationen `getImage()` und `getImages()` erzeugen kann, welche die Java-Typen `Image` beziehungsweise *`List<Image>`* zurückgeben.

- Ändern Sie die Annotation `@WebService` in der Klasse `SkiImageService` (Angabe des neuen Pfades zum WSDL-Dokument):

```
@WebService(serviceName = "SkiImageService",
            wsdlLocation = "ch03/image/ski.wsdl")
```

- Das WSDL-Dokument muß ebenfalls geändert werden (Angabe des neuen Pfades zum XML-Schema):

```
<types>
  <xsd:schema>
    <xsd:import
        namespace="http://image.ch03/"
        schemaLocation="ski.xsd">
    </xsd:import>
  </xsd:schema>
</types>
```

- Rufen Sie im Arbeitsverzeichnis (das Verzeichnis, welches das Unterverzeichnis *ch03* enthält), das `wsimport`-Kommando auf, um die Artefakte neu zu generieren:

```
% wsimport -keep -p skiC2 ch03/image/ski.wsdl
```

Nach diesen Änderungen arbeitet der Client mit den Java-Typen `Image` und *`List<Image>`* anstelle der Bytes aus der Base64-Kodierung der Bilder. Folgende Änderungen sind beim Client erforderlich:

```
import skiC2.SkiImageService_Service;
import skiC2.SkiImageService;
import java.awt.Image;
import java.util.List;

class SkiImageClient2 {

    public static void main(String[] args) {

        SkiImageService_Service service = new SkiImageService_Service();
        SkiImageService port = service.getSkiImageServicePort();

        Image image = port.getImage("telemk");
        List<Image> images = port.getImages();

        /* Process the images in some appropriate way. */

    }

}
```

Der geänderte `SkiImageService` und die aktualisierten `wsimport`-Artefakte gestatten den Clients, abstrakte Typen wie `java.awt.Image` anstelle „roher“ Bytes zu empfangen. Diese Änderungen vermögen allerdings nicht, die Volumenvergößerung durch die Base64-Kodierung abzustellen. Der nächste Unterabschnitt zeigt ein Beispiel, das die Base64-Kodierung vollständig umgeht, in dem es mit „rohen“ anstelle von kodierten Bytes arbeitet.

3.4.3 MTOM für große Datenmengen

[62] In diesem Unterabschnitt wird der `SkiImageService`-Dienst an die Übertragung byteorientierter Daten per MTOM angepaßt. Hierfür sind einige Änderungen erforderlich. Die einzelnen Änderungen

sind aber geringfügig.

[63] Das Attribut `expectedContentTypes` im XML-Schema tritt zweimal auf. Die Änderungen sind hervorgehoben:

```
<xs:complexType name="getImagesResponse">
  <xs:sequence>
    <xs:element
        name="return" type="xs:base64Binary"
        minOccurs="0" maxOccurs="unbounded"
        xmime:expectedContentTypes="application/octet-stream"
        xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="getImageResponse">
  <xs:sequence>
    <xs:element
        name="return" type="xs:base64Binary"
        minOccurs="0"
        xmime:expectedContentTypes="application/octet-stream"
        xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Der MIME-Typ `application/octet-stream` captures the optimization that recommends MTOM: Die Bytes der Graphik werden unkodiert an den Client des Dienstes übertragen. Die Volumenausdehnung durch Base64 oder vergleichbare Kodierungsverfahren wird dadurch vermieden.

[64] Die Klasse `SkiImageService` wird zur Verdeutlichung mit einer Annotation ausgezeichnet, die anzeigt, daß MTOM ins Spiel kommen könnte:

```
import javax.xml.ws.soap.SOAPBinding;

@WebService(serviceName = "SkiImageService")
// This binding value is enabled by default but put here for emphasis.
@BindingType(value = SOAPBinding.SOAP11HTTP_MTOM_BINDING)
@HandlerChain(file = "handler-chain.xml") // for message tracking
public class SkiImageService {
```

Dieser Bindungswert ist voreingestellt, die `@BindingType`-Annotation kann also fortgelassen werden und ist hier zur Hervorhebung angegeben.

[65] In der überarbeiteten Version des `Endpoint`-Publishers ist die Arbeit der Übersichtlichkeit halber auf mehrere Methoden verteilt:

```
package ch03.image;

import javax.xml.ws.Endpoint;
import javax.xml.ws.soap.SOAPBinding;

public class SkiImagePublisher {

    private Endpoint endpoint;

    public static void main(String[] args) {
        SkiImagePublisher me = new SkiImagePublisher();
        me.create_endpoint();
        me.configure_endpoint();
        me.publish();
```

```
    }

    private void create_endpoint() {
        endpoint = Endpoint.create(new SkiImageService());
    }

    private void configure_endpoint() {
        SOAPBinding binding = (SOAPBinding) endpoint.getBinding();
        binding.setMTOMEnabled(true);
    }

    private void publish() {
        int port = 9999;
        String url = "http://localhost:" + port + "/ski";
        endpoint.publish(url);
        System.out.println(url);
    }

}
```

Der Endpunkt des Dienstes aktiviert MTOM für Antworten an den Client. Der Dienst verwendet noch immer einen SOAP-Behandler, um die Antwortnachricht über die Standardausgabe auszugeben. Diese Ausgabe ist allerdings irreführend, da sie die Bilder als Base64-kodierte Werte im SOAP-Body anzeigt. Die MTOM-Optimierung geschieht nämlich erst *nach* der Verarbeitung des Behandlers. Ein Kommando wie `tcpdump` liefert ein genaueres Bild des tatsächlichen Geschehens.

[66] Die letzte Änderung betrifft den Client, mittlerweile `SkiImageClient3`, nachdem das `wsimport`-Kommando erneut aufgerufen wurde, um die Artefakte im Verzeichnis *skiC3* zu generieren:

```
import skiC3.SkiImageService_Service;
import skiC3.SkiImageService;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.soap.SOAPBinding;
import java.util.List;
import java.io.IOException;
import javax.activation.DataHandler;

class SkiImageClient3 {

    public static void main(String[] args) {

        SkiImageService_Service service = new SkiImageService_Service();
        SkiImageService port = service.getSkiImageServicePort();

        DataHandler image = port.getImage("nordic");
        List<DataHandler> images = port.getImages();
        dump(image);
        for (DataHandler hd : images)
            dump(hd);

    }

    private static void dump(DataHandler dh) {

        System.out.println();

        try {

            System.out.println("MIME type: " + dh.getContentType());
            System.out.println("Content:   " + dh.getContent());

        }

        catch(IOException e) { System.err.println(e); }
```

```
        }
    }
```

Die Operationen des Dienstes geben nun eine Referenz vom Typ `DataHandler` beziehungsweise *`List<DataHandler>`* zurück. Diese Änderung schlägt sich in den von `wsimport` erzeugten JAX-B-Artefakten nieder. Hier zum Beispiel ein Ausschnitt aus dem Artefakt `GetImageResponse` mit einer `@XmlMimeType`-Annotation, die sowohl das geänderte XML-Schema des Dienstes reflektiert, als auch anzeigt, daß der Dienst nun eine Referenz vom Typ `DataHandler` zurückgibt:

```
@XmlElement(name = "return")
@XmlMimeType("application/octet-stream")
protected DataHandler _return;
```

Nun hat der Client die Aufgabe, die entsprechenden Objekte aus dem optimierten, vom Dienst erhaltenen Bytestrom zu rekonstruieren. Der Kompromiß ist offensichtlich: MTOM optimiert die Übertragung durch Vermeiden der Volumenausdehnung, aber der Nachrichtenempfänger ist gezwungen, „rohe“ Bytes zu verarbeiten. Die Ausgabe der `dump()`-Methode lautet bei mir:

```
MIME type: application/octet-stream
Content:   java.io.ByteArrayInputStream@210b5b

MIME type: application/octet-stream
Content:   java.io.ByteArrayInputStream@170888e

MIME type: application/octet-stream
Content:   java.io.ByteArrayInputStream@11563ff
...
```

Der Eingabestrom gestattet den Zugriff auf die [underlying/ unterliegenden/eigentlichen] Bytes aus denen die `Image`-Objekte rekonstruiert werden können. Die `createImage()`-Methode des Dienstes könnte clientseitig zu diesem Zweck verwendet werden.

[67] Hat auch der Client größere Mengen byteorientierter Daten an den Webservice zu übertragen, so kann auch clientseitig MTOM aktiviert werden. Der folgende geänderte Client `SkiImageClient2` zeigt die Vorgehensweise mit Hilfe der *`BindingProvider`*-Methode `setMTOMEnable()`:

```
SkiImageService port = service.getSkiImageServicePort();

// Enable MTOM for client transmissions
BindingProvider bp = (BindingProvider) port;
SOAPBinding binding = (SOAPBinding) bp.getBinding();
binding.setMTOMEnabled(true);
```

Auch MTOM verursacht Unkosten, die der Empfänger bezahlt, indem er die „rohen“ Bytes aus dem DataHandler-Eingabestrom verarbeiten muß. Dennoch macht der Effizienzgewinn bei großen Datenmengen die Unkosten mühelos wett. Wie das Beispiel zeigt, ist MTOM sowohl client- als auch serviceseitig relativ leicht zu aktivieren.

3.5 Ausblick

[68] SOAP-basierte Webservices sind in vieler Hinsicht attraktiv. Sie basieren auf industriestandardisierten, anbieterunabhängigen Protokollen wie HTTP, XML und SOAP selbst. SOAP-basierte Webservices repräsentieren einen sprachneutralen Ansatz zur Entwicklung und zum Deployment verteilter Softwaresysteme. Die WSDL und damit verbundene Werkzeuge wie das `wsimport`-Kommando erleichtern das Schreiben von Clients für Webservices und sogar des Dienst selbst in der bevorzugten Sprache. SOAP, als ein die Sprache nicht wahrnehmendes Nachrichtensystem und XML-Schema, als

sprachneutrales Typsystem fördern die Interoperabilität von Diensten und die Standardisierung von Schnittstellen.

[69] Andererseits sind SOAP-basierte Dienste kompliziert, vor allem, wenn der Entwickler, aus welchem Grund auch immer, in die Tiefe der Infrastruktur vorstoßen muß. Selbst auf der Applikationsebene sind die APIs für SOAP-basierte Webservices ziemlich umfangreich geworden. Zahlreiche Standardisierungsorganisationen sind an SOAP und SOAP-basierten Webservices beteiligt, darunter das W3C, *OASIS* (Organization for the Advancement of Structured Information Standards), die *IETF* (Internet Engineering Task Force) und die *WS-I* (Web Services Interoperability). Spezifizierungsinitiativen sind auf zahlreichen und so unterschiedlichen Gebieten wie Interoperabilität, Geschäftsabläufe, Präsentation, Sicherheit, Metadaten, Ausfallsicherheit, Resourcen, Datentransfer, XML, Management, Transaktionen und SOAP selbst aktiv. Jedes Gebiet zerfällt in mehrere Teilgebiete. Das Gebiet Sicherheit hat beispielsweise zehn Teilgebiete, Interoperabilität sechs, Metadaten und Datentransfer je neun.

[70] Man hört nicht selten die Beschwerde, SOAP- und SOAP-basierte Webservices seien „zu Tode entwickelt“ worden. Das JAX-WS-Framework reflektiert diese Komplexität durch seine Fülle von Annotationen und Werkzeugen. Diese Komplexität erklärt, wenigstens zum Teil, die gegenwärtige Beliebtheit der REST-basierten oder RESTful („gemütlichen“) Ansätze für Webservices. Das nächste Kapitel konzentiert sich auf Dienste im REST-Stil.

Kapitel 4

Dienste im REST-Stil

Inhaltsübersicht

4.1 Resourcen und ihre Repräsentationen

[1] *Roy Fielding* (*http://roy.gbiv.com*) hat das Akronym REST in seiner Doktorarbeit geprägt. Kapitel 5 der Dissertation beschreibt die Leitsätze dessen, was unter der Bezeichnung „Dienst im

REST-Stil“ beziehungsweise RESTful Webservice bekannt geworden ist. Fielding hat einen beeindruckenden Lebenslauf. Er ist unter anderem einer der Hauptautoren der HTTP-Spezifikation sowie Mitgründer der ASF (Apache Software Foundation).

[2] Die Ansätze REST und SOAP sind recht verschieden voneinander. SOAP ist ein Nachrichtenübertragungsprotokoll, REST dagegen ein Softwarearchitekturstil für verteilte Hypermediasysteme, sprich Systeme, in denen Text, Graphiken, Audiodateien und andere Medien über ein Netzwerk verteilt und über *Hyperlinks* miteinander verbunden sind. Das World Wide Web (WWW) ist das offensichtliche Beispiel eines solchen Systems. Da dieses Buch von Webservices handelt, ist das verteilte Hypermediasystem WWW der Gegenstand unseres Interesses. Im WWW ist HTTP sowohl Transportprotokoll als auch Nachrichtenübertragungssytem, da HTTP-Anfragen und -Antworten Nachrichten sind. Die Nutzdaten von HTTP-Nachrichten können mit Hilfe des MIME-Typsystems typisiert werden. HTTP liefert in jeder Antwort einen Statuswerte, der die Quelle der Anfrage über Erfolg oder Scheitern der Anfrage und im letzteren Fall auch über die Ursache informiert.

[3] Die Abkürzung REST steht für „Representational State Transfer“ und bedarf der Erläuterung, da die zentrale Abstraktion bei REST, nämlich die *Resource*, nicht in der Expansion des Akronyms vorkommt. Eine *Resource* im Sinne von REST ist alles, was einen URI besitzt, das heißt einen Bezeichner dessen Format bestimmten Regeln gehorcht. Diese Formatierungsanforderungen bewirken die Einheitlichkeit der URIs. Die Expansion der Abkürzung URI lautet „Uniform *Resource* Identifier“. Die Begriffe „*URI*“ und „Resource“ sind also miteinander verflochten.

[4] In der Praxis ist eine Resource ein Stück Information, das von einem Hyperlink referenziert wird. Hyperlinks verwenden URIs zur Angabe der Verknüpfung. Beispiele für Resourcen sind ebenso zahlreich wie irreführend, indem sie scheinbar andeuten, daß Resourcen außer ihrer Identifizierbarkeit durch ihre URIs noch eine andere Gemeinsamkeit haben müssen. Aber das Bruttosozialprodukt von Litauen im Jahre 2001 ist ebenso eine Resource, wie das Modern Jazz Quartett. Ernie Banks Baseballerfolge zählen ebenso als Resource, wie ein Algorithmus zur Berechnung des maximalen Flusses. Das Konzept der Resource ist bemerkenswert breit, gleichzeitig aber auch beeindruckend einfach und präzise.

[5] Resourcen sind, als webbasierte Informationsstücke, solange nutzlos, bis sie wenigstens eine Repräsentation haben. Im WWW sind Repräsentationen MIME-typisiert. Der häufigste Repräsentationstyp für Resourcen ist wahrscheinlich nach wie vor `text/html`, wobei Resourcen heute tendenziell mehrere Repräsentationen und Repräsentationstypen haben. Es gibt beispielsweise verschiedene miteinander verbundene HTML-Seiten, die das Modern Jazz Quartet repräsentieren, die Resource hat aber auch audio- und audiovisuelle Repräsentationsformen.

[6] Resourcen sind zustandsbehaftet. Ernie Banks Erfolge im Baseball haben sich während seiner Karriere bei den Chicago Cubs zwischen 1953 und 1971 verändert und gipfelten 1977 mit seiner Aufnahme in die Baseball Hall of Fame. Eine nützliche Repräsentation einer Resource muß einen Zustand erfassen. Die aktuellen HTML-Seiten über Ernie Banks bei *http://www.baseball-reference.com* müssen alle seine Leistungen in der Major League darstellen, von seinem Anfangsjahr 1953 bis zu seiner Aufnahme in die Hall of Fame.

[7] Bei einer REST-Anfrage an eine Resource bleibt die Resource selbst auf dem Host, der den Dienst anbietet. Die Quelle der Anfrage erhält bei einer erfolgreich verarbeiteten Anfrage typischerweise eine *Repräsentation* der Resource zurück. Es ist diese Repräsentation, die vom Dienst- an den Clienthost übertragen wird. Ein REST-Client veranlaßt eine Anfrage bezüglich einer Resource, zum Beispiel eine Anfrage, die Resource zu lesen. Wird diese Anfrage erfolgreich verarbeitet, so wird eine typisierte Repräsentation der Resource (zum Beispiel `text/html`) vom Server, auf dem die Resource liegt, an den Client gesendet, der die Anfrage veranlaßt hat. Die Repräsentation ist nur dann a good one, wenn sie den Zustand der Resource in einer zweckmäßigen Weise angibt“.

[8] Webservices im REST-Stil verlangen nicht nur die Repräsentation von Resourcen, sondern auch clientseitig aufgerufene Operationen bezüglich dieser Resourcen. Im Mittelpunkt des REST-Ansatzes steht die Erkenntnis, daß HTTP, obwohl der Name das Wort „Transport“ enthält, eine Programmierschnittstelle (API) und nicht einfach nur ein Transportprotokoll ist. HTTP hat bekannte Schlüsselworte, die offiziell als „HTTP-Methoden“ bezeichnet werden. Tabelle 4.1 zeigt vier dieser Schlüsselworte, die den bekannten CRUD-Operationen (`create`, `read`, `update`, `delete`) entsprechen. HTTP beachtet Groß-/Kleinschreibung zwar nicht, aber die Namen der HTTP-Methoden werden traditionell groß geschrieben. Es gibt noch drei weitere HTTP-Methoden. Die `HEAD`-Methode ist eine Variante der `GET`-Methode und fordert nur die HTTP-Header an, die bei einer `GET`-Anfrage gesendet werden würden. Die beiden verbleibenden HTTP-Methoden sind `TRACE` und `OPTIONS`.

HTTP-Methode	Bedeutung im Sinne von CRUD
`POST`	Anlegen (`create`) einer neuen Resource aus den Anfragedaten.
`GET`	Lesen (`read`) einer Resource.
`PUT`	Aktualisieren (`update`) einer Resource aus den Anfragedaten.
`DELETE`	Löschen (`delete`) einer Resource.

Tabelle 4.1: HTTP-Schlüsselworte (HTTP-Methoden) und CRUD-Operationen (`create`, `read`, `update`, `delete`).

[9] Abbildung 4.1 zeigt eine Beispielresource mit identifierendem URI, einen REST-Client und einige typisierte Repräsentationen, die als Antworten auf HTTP-Anfragen an die Resource gesendet wurden. Jede HTTP-Anfrage beinhaltet den Namen einer HTTP-Methode, um die bezüglich der Resource auszuführende CRUD-Operationen zu benennen. Eine gute Repräsentation zeichnet sich genau dadurch aus, daß sie zu der angeforderten Operation paßt und den Zustand der Resource in einer zweckmäßigen Weise erfaßt. Eine `GET`-Anfrage kann hier die Geschichte meines Hackerlebens entweder als HTML-Seite oder als zusammengefaßten Kurzfilm anfordern. Der Film fällt hinsichtlich der Zustandsbeschreibung der Resource durch, da er nur die wichtigsten Fehler in der Karriere meines Bruders anstelle meiner eigenen Fehler zeigt. Eine typische HTML-Repräsentation einer Resource enthält Hyperlinks auf andere Resource, die wiederum Gegenstand von HTTP-Anfragen mit entsprechenden HTTP-Methoden sein können.

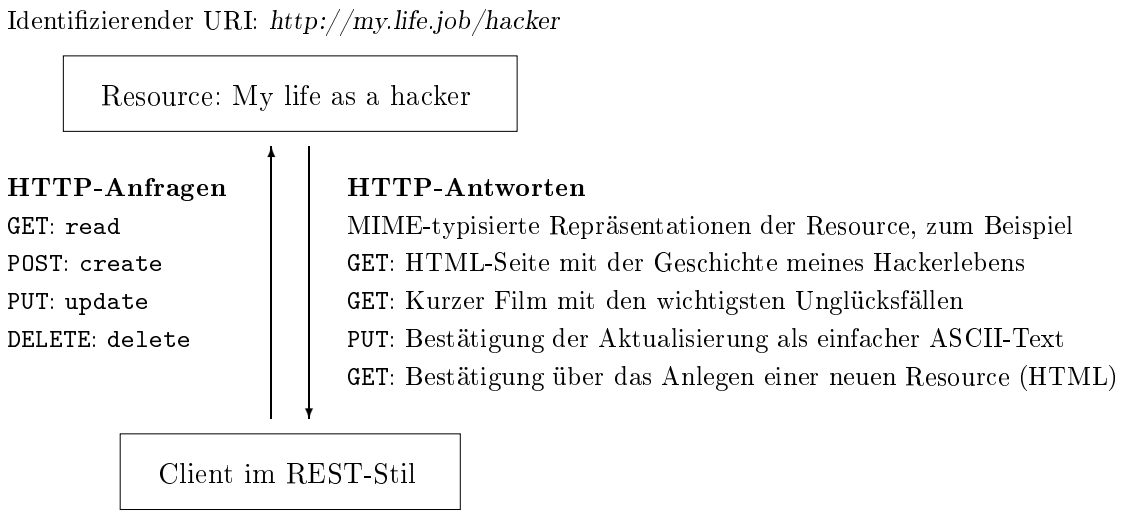

Abbildung 4.1: Ausschnitt aus einem REST-System.

[10] HTTP-Antworten beinhalten standardisierte Statuswerte, etwa `404` (`Not Found`), um zu signalisieren, daß die angeforderte Resource nicht gefunden oder `200` (`OK`), um anzuzeigen, daß die Anfrage erfolgreich verarbeitet wurde. Kurz, HTTP unterstützt Schlüsselworte und MIME-Typen für Anfragen und Statuswerte (und MIME-Typen) für Antworten.

[11] Moderne Webbrowser veranlassen nur `GET`- und `POST`-Anfragen. Viele Applikationen behandeln diese beiden Anfragetypen synonym. Die abstrakte Klasse `javax.servlet.http.HttpServlet` zum Beispiel, definiert Rückrufmethoden wie `doGet()` und `doPost()`, um `GET`- beziehungsweise `POST`-Anfragen zu verarbeiten. Jede Rückrufmethode hat dieselben Parametertypen *`javax.servlet.http.HttpServletRequest`* (die Schlüssel/Wert-Paare aus der Anfrage) und *`javax.servlet.http.HttpServletResponse`* (eine typisierte Antwort an die Quelle der Anfrage). Häufig rufen beide Rückrufmethoden dieselben Anweisungen auf (zum Beispiel in dem eine Methode die andere aufruft) und heben dadurch die ursprüngliche Unterscheidung zwischen `read` und `update` im HTTP-Kontext auf. Ein Leitprinzip des REST-Stils lautet, die ursprüngliche Wirkung der HTTP-Methoden zu respektieren. Insbesondere ist jede `GET`-Anfrage *idempotent*, das heißt frei von Seiteneffekten, da `GET` eine `read`-Operation ist (im Gegensatz zu einer `create`-, `update`- oder `delete`-Operation). Eine `GET`-Anfrage wird als *sichere* `GET`-Anfrage bezeichnet, wenn sie eine `read`-Operation ohne Seiteneffekte ist.

[12] Der REST-Ansatz setzt keineswegs voraus, daß die Resourcen oder die zur Herstellung von Repräsentationen dieser Resourcen erforderliche Verarbeitung einfach ist. Ein Dienst im REST-Stil kann genau so ausgetüftelt und kompliziert sein, wie ein SOAP-basierter Webservice. Der REST-Stil versucht, die Dinge zu vereinfachen, indem er übernimmt, was HTTP durch das MIME-Typsystem bereits anbietet: Eingebaute CRUD-Operationen, einheitlich identifizierbare Resourcen und typisierte Repräsentationen, die den Zustand einer Resource erfassen. Als Entwurfsphilosophie versucht REST, die Komplexität der Applikation an den Endpunkten zu isolieren, das heißt bei Client und Dienst. Auf der Seite des Dienstes kann viel Logik und Rechenaufwand erforderlich sein, um die Resourcen zu unterhalten und zweckmäßige Repräsentationen zu generieren, zum Beispiel große und komplex formatierte XML-Dokumente. Auf der Clientseite kann beträchtlicher Aufwand an XML-Verarbeitung notwendig sein, um die gewünschte Information aus den vom Dienst an den Client übertragenen XML-Repräsentationen zu entnehmen. Der REST-Ansatz hält die Komplexität allerdings von der Transportebene fern, da die Repräsentation einer Resource im Körper einer HTTP-Antwort an den Client übertragen wird. Im Gegensatz dazu verkompliziert ein SOAP-basierter Dienst die Transportebene zwangsläufig, da eine SOAP-Nachricht im Körper einer Transportnachricht verschachtelt wird, zum Beispiel in einer HTTP- oder SMTP-Nachricht. SOAP benötigt Nachrichten in Nachrichten, REST nicht.[1]

4.1.1 HTTP-Methoden und die Undurchsichtigkeit von URIs

[13] Ein URI „*undurchsichtig*“, das heißt zwischen den URIs

```
http://bedrock/citizens/fred
```

und

```
http://bedrock/citizens
```

besteht keine Vererbungsbeziehung, obwohl Fred Einwohner von Bedrock ist. Beide URIs sind nicht mehr als zwei verschiedene, voneinander unabhängige Bezeichner. Gute URIs sind selbstständlich suggestiv hinsichtlich dessen, was sie identifizieren. Die Kernaussage lautet: URIs haben keine intrinsische hierarchische Struktur. URIs können und sollen interpretiert werden, aber diese Interpre-

[1] Eine sorgfältige Darstellung von Diensten im REST-Stil finden Sie in Leonard Richardsons and Sam Rubys Buch *RESTful Web Services* (O'Reilly).

tationen werden den URIs aufgeprägt, statt ihnen innezuwohnen. Die URI-Syntax ähnelt zwar der Syntax zur Navigation in einem hierarchischen Dateisystem, aber diese Ähnlichkeit ist irreführend. Ein URI ist ein undurchlässiger Bezeichner, ein logischer Name und bezeichnet genau eine Resource.

[14] Bei Diensten im REST-Stil funktionieren URIs wie Substantive und HTTP-Methoden wie Verben, die Operationen bezüglich der durch diese Substantive bezeichneten Resource bestimmen. Hier die HTTP-Kopfzeile einer Anfrage an den *`ch01.ts.TimeServer`*-Dienst aus Kapitel 1:

```
POST http://127.0.0.1:9876/ts HTTP/ 1.1
```

Die HTTP-Methode wird zuerst angegeben, danach der URI und schließlich die vom Client verwendete HTTP-Version. Der URI ist natürlich eine URL, die den Webservice lokalisiert. Tabelle 4.2 zeigt anhand vereinfachter URIs die beabsichtigte Wirkung einiger Kombinationen von HTTP-Methoden und URIs.

HTTP-Methode und URI	Beabsichtige Wirkung
`POST emps`	Anlegen eines neuen Angestellten aus den Anfragedaten.
`GET emps`	Anfordern einer Liste aller Angestellten.
`GET emps?id=27`	Anfordern des Angestellten mit der Kennnummer 27.
`PUT emps`	Aktualisieren der Angestellenliste aus den Anfragedaten.
`DELETE emps`	Löschen der Angestelltenliste.
`DELETE emps?id=27`	Löschen des Angestellten mit der Kennummer 27.

Tabelle 4.2: Beispiele für Kombinationen von HTTP-Methoden und URIs mit der beabsichtigten Wirkung.

[15] Diese Kombinationen sind knapp, präzise und einheitlich. Sie zeigen, daß die REST-Konventionen einfache und klare Ausdrücke dahingehend ermöglichen, welche Operation bezüglich welcher Resource ausgeführt werden soll. Die Methoden `PUT` und `POST` treten bei Anfragen mit Körpern auf. Die Anfragedaten werden im Körper der HTTP-Nachricht befördert. Die Methoden `GET` und `DELETE` treten bei Anfragen ohne Körper auf, bei denen die Anfragedaten im Query-String transportiert werden.

[16] Dienste im REST-Stil sind Turing-vollständig, das heißt äquivalent zu jedem computational system, insbesondere also einem System, das aus SOAP-basierten Webservices aufgebaut ist. Die Entscheidung, ob eine bestimmte Applikation im REST-Stil entworfen wird, hängt wie immer von den praktischen Anforderungen ab. Dieser erste Abschnitt hat REST aus der Vogelperspektive betrachtet. Es ist nun an der Zeit, mit Beispielen zu den Details vorzudringen.

4.2 Die Annotation @WebServiceProvider ersetzt @WebService

[17] Die Annotation `@WebService` gibt an, daß die zwischen einem Dienst und seinen Clients ausgetauschten Nachrichten SOAP-Envelopes sind. Die Annotation `@WebServiceProvider` gibt an, daß die ausgetauschten Nachrichten XML-Dokumente irgendeines Typs sind, sogenanntes „*rohes XML*“. Selbstverständlich könnte ein mit `@WebServiceProvider` annotierter Dienst SOAP-Dokumente selbst verarbeiten und generieren, aber diese Vorgehensweise wird nicht empfohlen. ([Ein späteres Beispiel] zeigt sie dennoch, siehe [Seite ??]) Die Verwendung der `@WebService`-Annotation ist der nächstliegende Weg, um einen SOAP-basierten Webservice anzubieten.

[18] Die Antwort eines Dienstes im REST-Stil nach dem Request/Response-Muster beinhaltet „rohes XML“, während die beim Dienst eingehende Anfrage unter Umständen überhaupt kein XML enthält. Eine `GET`-Anfrage hat keinen Körper, so daß Anfrageargumente als Attribute (Kollektion von Schlüssel/Wert-Paaren) im *Query-String* übertragen werden. Ein Beispiel:

```
http://www.onlineparlor.com/bets?horse=bigbrown&jockey=kent&amount=25
```

Das Fragezeichen (`?`) markiert den Anfang des Query-Strings. Die Attribute sind durch Kaufmanns-Und (`&`) voneinander getrennte Schlüssel/Wert-Paare. Die Reihenfolge der Attribute im Query-String ist beliebig. Das `jockey`-Attribut könnte als erstes Attribut im Query-String auftreten, ohne die Wirkung der Anfrage zu verändern. Eine `POST`-Anfrage hat dagegen einen Körper, der ein beliebiges XML-Dokument anstelle eines SOAP-Envelopes enthalten kann.

[19] Ein mit `@WebServiceProvider` annotierter Dienst implementiert das Interface *`javax.xml.ws.Provider`*, welches die `invoke()`-Methode deklariert:

```
public Source invoke(Source request)
```

Die Methode erwartet eine Referenz vom Typ *`Source`* (das referenzierte Objekt repräsentiert Bytes, zum Beispiel die Bytes in einem XML-Dokument, das die Anfrage an den Dienst darstellt) und gibt eine Referenz desselben Typs zurück (die Bytes in der XML-Antwort). Trifft eine Anfrage ein, so leitet die Infrastruktur die Anfrage an die `invoke()`-Methode weiter, welche die Anfrage in einer zweckdienlichen Weise verarbeitet. Diese Punkte werden nun durch Beispiele veranschaulicht.

4.3 Der Teams-Dienst (REST-Stil)

[20] Das erste Beispiel für einen Dienst im REST-Stil ist eine überarbeitete Version des SOAP-basierten `ch01.team.Teams`-Dienstes aus Abschnitt 1.9. Die Teams sind Komikertruppen wie die Marx Brothers. Der Dienst reagiert zunächst nur auf `GET`-Anfragen, wird aber noch ausgebaut, um auch die übrigen HTTP-Methoden der CRUD-Operationen zu unterstützen.

4.3.1 Die Annotation @WebServiceProvider

[22] Das folgende Beispiel zeigt den Quelltext der Anfangsversion des `RestfulTeams`-Dienstes:

Beispiel 4.1: Der RestfulTeams-Dienst.

```
package ch04.team;

import javax.xml.ws.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.annotation.Resource;
import javax.xml.ws.BindingType;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPException;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.http.HTTPBinding;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.util.Collections;
import java.util.Map;
import java.util.HashMap;
```

```
import java.util.List;
import java.util.ArrayList;
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.beans.XMLEncoder;
import java.beans.XMLDecoder;

// The class below is a WebServiceProvider rather than the more usual
// SOAP-based WebService. The service implements the generic Provider
// interface rather than a customized SEI with designated @WebMethods.
@WebServiceProvider

// There are two ServiceModes: PAYLOAD, the default, signals that the service
// wants access only to the underlying message payload (e.g., the
// body of an HTTP POST request); MESSAGE signals that the service wants
// access to entire message (e.g., the HTTP headers and body).
@ServiceMode(value = javax.xml.ws.Service.Mode.MESSAGE)

// The HTTP_BINDING as opposed, for instance, to a SOAP binding.
@BindingType(value = HTTPBinding.HTTP_BINDING)
public class RestfulTeams implements Provider<Source> {

    @Resource
    protected WebServiceContext ws_ctx;

    private Map<String, Team> team_map; // for easy lookups
    private List<Team> teams;           // serialized/deserialized
    private byte[] team_bytes;          // from the persistence file

    private static final String file_name = "teams.ser";

    public RestfulTeams() {
        read_teams_from_file(); // read the raw bytes from teams.ser
        deserialize();          // deserialize to a List<Team>
    }

    // This method handles incoming requests and generates the response.
    public Source invoke(Source request) {

        if (ws_ctx == null) throw new RuntimeException("DI failed on ws_ctx.");

        // Grab the message context and extract the request verb.
        MessageContext msg_ctx = ws_ctx.getMessageContext();
        String http_verb = (String)
            msg_ctx.get(MessageContext.HTTP_REQUEST_METHOD);
        http_verb = http_verb.trim().toUpperCase();

        // Act on the verb. To begin, only GET requests accepted.
        if (http_verb.equals("GET")) return doGet(msg_ctx);
        else throw new HTTPException(405); // method not allowed

    }

    private Source doGet(MessageContext msg_ctx) {

        // Parse the query string.
        String query_string = (String) msg_ctx.get(MessageContext.QUERY_STRING);

        // Get all teams.
        if (query_string == null)
            return new StreamSource(new ByteArrayInputStream(team_bytes));
        // Get a named team.
        else {
```

```
        String name = get_value_from_qs("name", query_string);

        // Check if named team exists.
        Team team = team_map.get(name);
        if (team == null) throw new HTTPException(404); // not found
        // Otherwise, generate XML and return.
        ByteArrayInputStream stream = encode_to_stream(team);
        return new StreamSource(stream);
    }

}

private ByteArrayInputStream encode_to_stream(Object obj) {

    // Serialize object to XML and return
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    XMLEncoder enc = new XMLEncoder(stream);
    enc.writeObject(obj);
    enc.close();
    return new ByteArrayInputStream(stream.toByteArray());

}

private String get_value_from_qs(String key, String qs) {

    String[] parts = qs.split("=");
    // Check if query string has form: name=<team name>
    if (!parts[0].equalsIgnoreCase(key))
        throw new HTTPException(400); // bad request
    return parts[1].trim();

}

private void read_teams_from_file() {

    try {
        String cwd = System.getProperty ("user.dir");
        String sep = System.getProperty ("file.separator");
        String path = get_file_path();
        int len = (int) new File(path).length();
        team_bytes = new byte[len];
        new FileInputStream(path).read(team_bytes);
    }
    catch(IOException e) { System.err.println(e); }

}

private void deserialize() {

    // Deserialize the bytes into a list of teams
    XMLDecoder dec = new XMLDecoder(new ByteArrayInputStream(team_bytes));
    teams = (List<Team>) dec.readObject();

    // Create a map for quick lookups of teams.
    team_map = Collections.synchronizedMap(new HashMap<String, Team>());
    for (Team team : teams) team_map.put(team.getName(), team);

}

private String get_file_path() {

    String cwd = System.getProperty ("user.dir");
    String sep = System.getProperty ("file.separator");
    return cwd + sep + "ch04" + sep + "team" + sep + file_name;

}
```

```
}
```

Die Annotationen zeigen den Umstieg von einem SOAP-basierten Webservice auf einen Dienst im REST-Stil an. Die wichtigsten Annotation ist `@WebServiceProvider`, anstelle von `@WebService`. Die erste bei beiden folgenden Annotationen überschreibt den voreingestellten Wert `PAYLOAD` durch `MESSAGE`:

```
@ServiceMode(value = javax.xml.ws.Service.Mode.MESSAGE)
@BindingType(value = HTTPBinding.HTTP_BINDING)
```

Die `@ServiceMode`-Annotation ist nur zur Vorführung vorhanden. Der `RestfulTeams`-Dienst würde mit dem Standardwert ebenso gut funktionieren. Die zweite Annotation gibt an, daß der Dienst „rohes XML“ über HTTP versendet, statt SOAP über HTTP.

[23] Der `RestfulTeams`-Dienst arbeitet mit „rohem XML“ anstelle von SOAP-Dokumenten. Die Komikertruppen sind auf der lokalen Festplatte in der Datei *teams.ser* gespeichert. Die Datei enthält ein mit Hilfe der Klasse `java.beans.XMLEncoder` generiertes XML-Dokument. Hier ein Ausschnitt aus der Datei *teams.ser*:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<java version=”1.6.0_06” class=”java.beans.XMLDecoder”>
  <object class=”java.util.ArrayList”>
    <void method=”add”>
      <object class=”ch04.team.Team”>
        <void property=”name”>
          <string>BurnsAndAllen</string>
        </void>
        <void property=”players”>
          <object class=”java.util.ArrayList”>
            <void method=”add”>
              <object class=”ch04.team.Player”>
                <void property=”name”>
                  <string>George Burns</string>
                </void>
                <void property=”nickname”>
                  <string>George</string>
                </void>
              </object>
            </void>
            ...
</java>
```

Das gespeicherte XML-Dokument wird mit Hilfe eines `java.beans.XMLDecoder`-Objektes in einen Container vom Typ *`List<Team>`* zurückverwandelt („deserialisiert“). Der komfortableren Bedienung halber, verfügt die Klasse `RestfulTeams` auch über einen Container vom Typ *`Map<String,Team>`*, damit einzelne Komikertruppen auch über ihren Namen abgefragt werden können. Die Implementierung der `deserialize()`-Methode ist:

```
private void deserialize() {

    // Deserialize the bytes into a list of teams
    XMLDecoder dec = new XMLDecoder(new ByteArrayInputStream(team_bytes));
    teams = (List<Team>) dec.readObject();

    // Create a map for quick lookups of teams.
    team_map = Collections.synchronizedMap(new HashMap<String, Team>());
    for (Team team : teams) team_map.put(team.getName(), team);

}
```

Der `RestfulTeams`-Dienst wird mit Hilfe eines `Endpoint`-Publishers aus der JAX-WS-Bibliothek in Betrieb genommen. Die Vorgehensweise ist mittlerweile vertraut und funktioniert wie bei den SOAP-basierten Diensten:

```
package ch04.team;

import javax.xml.ws.Endpoint;

class TeamsPublisher {
    public static void main(String[] args) {
        int port = 8888;
        String url = "http://localhost:" + port + "/teams";
        System.out.println("Publishing Teams restfully on port " + port);
        Endpoint.publish(url, new RestfulTeams());
    }
}
```

Unter den vier HTTP-Methoden, die den CRUD-Operationen entsprechen, hat nur `GET` keinen Seiteneffekt auf die Resource, hier die Liste klassischer Komikertruppen. Im Augenblick besteht somit keine Notwendigkeit, eine geänderte Liste in der Datei *teams.ser* zu speichern („serialisieren").

[24] Die JAX-WS-Laufzeitbibliothek gibt Anfragen an die `invoke()`-Methode des `RestfulTeams`-Dienstes weiter:

```
public Source invoke(Source request) {

    if (ws_ctx == null)
        throw new RuntimeException("Injection failed on ws_ctx.");

    // Grab the message context and extract the request verb.
    MessageContext msg_ctx = ws_ctx.getMessageContext();
    String http_verb = (String) msg_ctx.get(MessageContext.HTTP_REQUEST_METHOD);
    http_verb = http_verb.trim().toUpperCase();

    // Act on the verb. For now, only GET requests accepted.
    if (http_verb.equals("GET")) return doGet(msg_ctx);
    else throw new HTTPException(405); // method not allowed

}
```

Die `invoke()`-Methode fragt über den Nachrichtenkontext (*`MessageContext`*) die HTTP-Methode ab und ruft anschließend die entsprechende Methode auf, um die Anfrage zu verarbeiten (an dieser Stelle nur `doGet()`). Nennt die Anfrage eine andere HTTP-Methode als `GET`, so wird eine Ausnahme vom Typ `HTTPException` mit dem Statuswert `405` (`Method Not Allowed`) ausgeworfen, um anzuzeigen, daß diese HTTP-Methode nicht unterstützt wird. Tabelle 4.3 zeigt eine Auswahl der zahlreichen HTTP-Statuswerte.

HTTP-Statuswert	Offizielle Begründung	Bedeutung
200	OK	Anfrage wurde erfolgreich verarbeitet.
400	Bad Request	Anfrage malformed.
403	Forbidden	Anfrage wurde zurückgewiesen.
404	Not Found	Resource wurde nicht gefunden.
405	Method Not Allowed	Methode wird nicht unterstützt.
415	Unsupported Media Type	Inhaltstyp nicht erkannt.
500	Internal Server Error	Anfrageverarbeitung gescheitert.

Tabelle 4.3: Beispiele für HTTP-Statuswerte.

[25] Im Allgemeinen haben Statuswerte im Bereich 100 bis 199 informatorischen Charakter, 200 bis 299 melden erfolgreiche Verarbeitung, 300 bis 399 betreffen Umleitung (*redirection*), 400 bis 499 sind Client- und 500 bis 599 Serverfehler.

[26] Der Dienst behandelt zwei Arten von `GET`-Anfragen (später auch zwei Arten von `DELETE`-Anfragen). Eine `GET`-Anfrage ohne Query-String wird als Anfrage nach der Liste aller gespeicherten Komikertruppen interpretiert und mit einer Kopie des XML-Dokumentes in der Datei *teams.ser* beantwortet. Bei einer `GET`-Anfrage mit Query-String der Form `?name=<team name>`, zum Beispiel `?name=MarxBrothers`, fragt die `doGet()`-Methode das angeforderte Team ab und [kodiert/serialisiert] es mit Hilfe eines `XMLEncoder`-Objektes in der Methode `encode_to_stream()` als XML-Dokument. Die Implementierung der `doGet()`-Methode lautet:

```
if (query_string == null) // get all teams

    // Respond with list of all teams

    return new StreamSource(new ByteArrayInputStream(team_bytes));
else { // get the named team

    String name = get_name_from_qs(query_string);

    // Check if named team exists.
    Team team = team_map.get(name);
    if (team == null) throw new HTTPException(404); // not found

    // Respond with named team.
    ByteArrayInputStream stream = encode_to_stream(team);
    return new StreamSource(stream);

}
```

Die Klasse `javax.xml.transform.stream.StreamSource` repräsentiert eine Quelle von Bytes, die aus dem XML-Dokument stammen und dem anfragenden Client zur Verfügung gestellt werden. Bei einer Anfrage für die Marx Brothers, gibt `doGet()` das folgende XML-Dokument als Bytestrom zurück (gekürzt):

```
<java version="1.6.0_06" class="java.beans.XMLDecoder">
  <object class="ch04.team.Team">
    <void property="name">
      <string>MarxBrothers</string>
    </void>
    <void property="players">
      <object class="java.util.ArrayList">
        <void method="add">
          <object class="ch04.team.Player">
            <void property="name">
              <string>Leonard Marx</string>
            </void>
            <void property="nickname">
              <string>Chico</string>
            ...
```

4.3.2 Sprachtransparenz bei Diensten im REST-Stil

[27] Der erste Client des `RestfulTeams`-Dienstes ist zum Nachweis der Sprachtransparenz nicht in Java, sondern in Perl geschrieben. Der Client *teamsClientREST.pl* (hier gekürzt) sendet eine `GET`-Anfrage und verarbeitet die Antwort in einigen einfachen Schritten (die ungekürzte Fassung aus der Quelltextdistribution enthält neben weiteren `GET`-Anfragen auch `POST`-, `PUT`- und `DELETE`-Anfragen):

```
#!/usr/bin/perl

use strict;
use LWP;
use XML::XPath;

# Create the user agent.
my $ua = LWP::UserAgent->new;

my $base_uri = 'http://localhost:8888/teams';

# GET teams?name=MarxBrothers
my $request = $base_uri . '?name=MarxBrothers';
send_GET($request);

sub send_GET {
    my ($uri, $qs_flag) = @_;

    # Send the request and get the response.
    my $req = HTTP::Request->new(GET => $uri);
    my $res = $ua->request($req);

    # Check for errors.
    if ($res->is_success) {
        parse_GET($res->content, $qs_flag); # Process raw XML on success
    }
    else {
        print $res->status_line, "\n";      # Print error code on failure
    }
}

# Print raw XML and the elements of interest.
sub parse_GET {
    my ($raw_xml) = @_;
    print "\nThe raw XML response is:\n$raw_xml\n;;;\n";

    # For all teams, extract and print out their names and members
    my $xp = XML::XPath->new(xml => $raw_xml);
    foreach my $node ($xp->find('//object/void/string')->get_nodelist) {
        print $node->string_value, "\n";
    }
}
```

[28] Der Perl-Client veranlaßt eine `GET`-Anfrage an den URI *http://localhost:8888/teams*, den Endpunkt des per `Endpoint`-Publisher in Betrieb genommenen Dienstes. Wird die Anfrage erfolgreich verarbeitet, so gibt der Dienst eine XML-Repräsentation der Komikertruppen zurück, hier die mit Hilfe der `XMLEncoder`-Methode `writeObject()` generierte XML-Datei. Der Perl-Client gibt das „rohe XML“ aus und parst die Eingabedatei mit Hilfe eines XPath-Ausdrucks, um die Namen der Komikertruppen, die Namen der Mitglieder und ihre Spitznamen zu bekommen. Im produktiven Betrieb wäre die XML-Verarbeitung sorgfältiger konstruiert, die grundsätzliche Logik des Clients aber gleich: Veranlassen einer Anfrage an den Dienst und Verarbeiten der Antwort in einer zweckdienlichen Weise. Die Ausgabe des Clients lautet (gekürzt):

```
<java version="1.6.0_06" class="java.beans.XMLDecoder">
  <object class="java.util.ArrayList">
    <void method="add">
      <object class="ch04.team.Team">
        <void property="name">
          <string>BurnsAndAllen</string>
        </void>
        <void property="players">
```

```
            <object class="java.util.ArrayList">
              <void method="add">
                <object class="ch04.team.Player">
                  <void property="name">
                    <string>George Burns</string>
                  </void>
                  <void property="nickname">
                    <string>George</string>
                  </void>
                </object>
              </void>
              <void method="add">
                <object class="ch04.team.Player">
                  <void property="name">
                    <string>Gracie Allen</string>
                  </void>
                  <void property="nickname">
                    <string>Gracie</string>
                  </void>
                </object>
              </void>
            </object>
          </void>
        </object>
      </void>
      ...
</java>
;;;
BurnsAndAllen
George Burns
George
Gracie Allen
Gracie
AbbottAndCostello
William Abbott
Bud
Louis Cristillo
Lou
MarxBrothers
Leonard Marx
Chico
Julius Marx
Groucho
Adolph Marx
Harpo
```

Die Ausgabe unter den drei Semikola (`;;;`) besteht aus den abgefragten Komikertruppennamen, zusammen mit den Namen und Spitznamen der Mitglieder.

[29] Nun ein Java-Client für den `RestfulTeams`-Dienst:

```
import java.util.Arrays;
import java.net.URL;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URLEncoder;
import java.io.IOException;
```

```
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.ByteArrayInputStream;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.ParserConfigurationException;

class TeamsClient {

    private static final String endpoint = "http://localhost:8888/teams";

    public static void main(String[] args) {
        new TeamsClient().send_requests();
    }

    private void send_requests() {
        try {
            // GET requests
            HttpURLConnection conn = get_connection(endpoint, "GET");
            conn.connect();
            print_and_parse(conn, true);

            conn = get_connection(endpoint + "?name=MarxBrothers", "GET");
            conn.connect();
            print_and_parse(conn, false);
        }
        catch(IOException e) { System.err.println(e); }
        catch(NullPointerException e) { System.err.println(e); }
    }

    private HttpURLConnection get_connection(String url_string, String verb) {
        HttpURLConnection conn = null;
        try {
            URL url = new URL(url_string);
            conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod(verb);
        }
        catch(MalformedURLException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }
        return conn;
    }

    private void print_and_parse(HttpURLConnection conn, boolean parse) {
        try {
            String xml = "";
            BufferedReader reader =
                new BufferedReader(new InputStreamReader(conn.getInputStream()));
            String next = null;
            while ((next = reader.readLine()) != null)
                xml += next;
            System.out.println("The raw XML:\n" + xml);

            if (parse) {
                SAXParser parser =SAXParserFactory.newInstance().newSAXParser();
                parser.parse(new ByteArrayInputStream(xml.getBytes()),
                             new SaxParserHandler());
            }
```

```
            }
            catch(IOException e) { System.err.println(e); }
            catch(ParserConfigurationException e) { System.err.println(e); }
            catch(SAXException e) { System.err.println(e); }
        }

        static class SaxParserHandler extends DefaultHandler {
            char[] buffer = new char[1024];
            int n = 0;

            public void startElement(String uri, String lname,
                                     String qname, Attributes attributes) {
                clear_buffer();
            }

            public void characters(char[] data, int start, int length) {
                System.arraycopy(data, start, buffer, 0, length);
                n += length;
            }

            public void endElement(String uri, String lname, String qname) {
                if (Character.isUpperCase(buffer[0]))
                    System.out.println(new String(buffer));
                clear_buffer();
            }

            private void clear_buffer() {
                Arrays.fill(buffer, '\0');
                n = 0;
            }
        }
    }
```

Der Java-Client veranlaßt zwei `GET`-Anfragen und verwendet einen SAX-Parser (Simple API for XML), um die zurückerhaltene XML-Datei auszuwerten. Java bietet ein Sortiment von Hilfsmitteln zur XML-Verarbeitung an und die Beispiele stellen einige davon vor. Ein SAX-Parser ist ein strombasierter und ereignisgetriebener Parser. Er erhält einen Bytestrom und ruft Rückrufmethoden auf (etwa `startElement()` und `characters()`, siehe innere Klasse `SaxParserHandler` oben), um bestimmte Ereignisse zu verarbeiten, in diesem Fall das Vorkommen eines XML-Starttags oder das Vorkommen von Zeichendaten zwischen Start- und Endtag.

4.3.3 Zusammenfassung der charakteristischen Eigenschaften des REST-Stils

[30] Der `RestfulTeams`-Dienst zeigt bereits im jetzigen Stadium einige charakteristische Eigenschaften von Diensten im REST-Stil, läßt aber auch eine dieser Eigenschaften unbeachtet. Hier eine Zusammenfassung der Eigenschaften des Beispiels im jetzigen Umfang:

- Die Kombination aus einer HTTP-Methode und einem URI wie *http://.../teams* in einer Anfrage beschreibt eine CRUD-Operation bezüglich einer Ressource, hier eine Anfrage, um die verfügbaren Informationen über Komikertruppen zu lesen.
- Der `RestfulTeams`-Dienst verwendet HTTP-Statuswerte wie `404` (`Not Found`) oder `405` (`Method Not Allowed`), um fehlerhafte Anfragen zu beantworten.
- Bei einer gültigen Anfrage antwortet der Dienst mit einer XML-Repräsentation, die den Zustand der angeforderten Resource beinhaltet. Der Dienst reagiert im Augenblick nur auf `GET`-Anfragen. Die übrige CRUD-Funktionalität wird im folgenden Unterabschnitt ergänzt.

- Der Dienst nutzt die Vorteile von MIME-Typen nicht. Ein Client veranlaßt eine Anfrage entweder nach einer benannten Komikertruppe oder nach einer Liste aller Komikertruppen, gibt aber keinen bevorzugten Repräsentationstyp an (zum Beispiel `text/plain` im Gegensatz zu `text/xml` oder `text/html`). Ein späteres Beispiel zeigt typisierte Anfragen und Antworten.
- Die Implementierung eines Dienstes im REST-Stil ist nicht in derselben Weise eingeschränkt, wie ein SOAP-basierter Webservice, da es keinen formalen Dienstkontrakt gibt. Die Implementierung ist zwar flexibler, aber natürlich auch gleichermaßen aus dem Steigreif.

Im folgenden Unterabschnitt wird der `RestfulTeams`-Dienst erweitert, so daß auch `POST`-, `PUT`- und `DELETE`-Anfragen veranlaßt werden können.

4.3.4 Implementierung der restlichen CRUD-Operationen

[31] Die verbleibenden drei CRUD-Operationen, `create` (`POST`), `update` (`PUT`) und `delete` (`DELETE`), haben Seiteneffekte. Dadurch ergibt sich die Notwendigkeit, daß der `RestfulTeams`-Dienst die Datenstruktur im Arbeitsspeicher (die Komikertruppen- und die Mitgliederliste) und den dauerhaften Speicher (die lokale Datei *teams.ser*) aktualisieren muß. Der Dienst folgt einer eher eifrigen als trägen Strategie zur Aktualisierung von *teams.ser*. Die Datei wird nach jeder erfolgreich verarbeiteten `POST`-, `PUT`- oder `DELETE`-Anfrage aktualisiert. Im produktiven Betrieb könnte eine trägere, dafür aber effizientere Lösung gewählt werden.

[32] Die Implementierung der `invoke()`-Methode in der Klasse `RestfulTeams` ändert sich nur geringfügig, um die neuen Anfragemöglichkeiten zu integrieren. Die Änderungen sind:

```
MessageContext msg_ctx = ws_ctx.getMessageContext();
String http_verb = (String) msg_ctx.get(MessageContext.HTTP_REQUEST_METHOD);
http_verb = http_verb.trim().toUpperCase();

// Act on the verb.
if      (http_verb.equals("GET"))    return doGet(msg_ctx);
else if (http_verb.equals("DELETE")) return doDelete(msg_ctx);
else if (http_verb.equals("POST"))   return doPost(msg_ctx);
else if (http_verb.equals("PUT"))    return doPut(msg_ctx);
else throw new HTTPException(405);   // method not allowed
```

Die `doPost()`-Methode erwartet, daß die Anfrage ein XML-Dokument mit den Daten der neu anzulegenden Komikertruppe enthält, zum Beispiel:

```
<create_team>
  <name>SmothersBrothers</name>
  <player>
    <name>Thomas</name>
    <nickname>Tom</nickname>
  </player>
  <player>
    <name>Richard</name>
    <nickname>Dickie</nickname>
  </player>
</create_team>
```

Natürlich könnte ein XML-Schema an die Clients verteilt werden, das diese Struktur präzise beschreibt. In diesem Beispiel validiert `doPost()` das über die Anfrage erhaltene XML-Dokument nicht gegen ein Schema, sondern parst es, um die benötigten Informationen zu entnehmen, etwa den Namen der Komikerruppe und die Namen der Mitglieder. Ist eine benötigte Information nicht

vorhanden, wird der Statuswert `500` (`Internal Server Error`) beziehungsweise `400` (`Bad Request`) an den Client zurückgegeben. Die Implementierung der hinzugefügten Methode `doPost()` lautet:

```
private Source doPost(MessageContext msg_ctx) {

    Map<String, List> request = (Map<String, List>)
        msg_ctx.get(MessageContext.HTTP_REQUEST_HEADERS);

    List<String> cargo = request.get(post_put_key);
    if (cargo == null) throw new HTTPException(400); // bad request

    String xml = "";
    for (String next : cargo) xml += next.trim();
    ByteArrayInputStream xml_stream =
        new ByteArrayInputStream(xml.getBytes());
    String team_name = null;

    try {

        // Set up the XPath object to search for the XML elements.
        DOMResult dom = new DOMResult();
        Transformer trans =
            TransformerFactory.newInstance().newTransformer();
        trans.transform(new StreamSource(xml_stream), dom);
        URI ns_URI = new URI("create_team");

        XPathFactory xpf = XPathFactory.newInstance();
        XPath xp = xpf.newXPath();
        xp.setNamespaceContext(new NSResolver("", ns_URI.toString()));

        team_name = xp.evaluate("/create_team/name", dom.getNode());
        List<Player> team_players = new ArrayList<Player>();
        NodeList players = (NodeList)
            xp.evaluate("player", dom.getNode(), XPathConstants.NODESET);

        for (int i = 1; i <= players.getLength(); i++) {
            String name = xp.evaluate("name", dom.getNode());
            String nickname = xp.evaluate("nickname", dom.getNode());
            Player player = new Player(name, nickname);
            team_players.add(player);

        }

        // Add new team to the in-memory map and save List to file.
        Team t = new Team(team_name, team_players);
        team_map.put(team_name, t);
        teams.add(t);
        serialize();

    }

    catch(URISyntaxException e) { throw new HTTPException(500); }
    catch(TransformerConfigurationException e) { throw new HTTPException(500); }
    catch(TransformerException e) { throw new HTTPException(500); }
    catch(XPathExpressionException e) { throw new HTTPException(400); }

    // Send a confirmation to requester.
    return response_to_client("Team " + team_name + " created.");

}
```

4.3.5 Die Java API for XML Processing (JAX-P)

[33] Die `doPost()`-Methode des `RestfulTeams`-Dienstes stützt sich auf Interfaces und Klassen aus dem Package `javax.xml.transform`, Teil der Java API for XML Processing (JAX-P), um das in der Anfrage enthaltene XML-Dokument zu parsen. Die JAX-P wurde entworfen, um die XML-Verarbeitung zu vereinfachen und kommt den Anforderungen an Dienste im REST-Stil entgegen. Die beiden wichtigsten JAX-P-Bausteine des `RestfulTeams`-Dienstes sind die Klasse `javax.xml.transform.dom.DOMResult` und das Interface *`javax.xml.xpath.XPath`*. Beim Java-Client `TeamsClient` in Unterabschnitt 4.3.2 wurde ein SAX-Parser verwendet, um die Komikertruppenliste zu verarbeiten, die der Dienst bei einer erfolgreichen `GET`-Anfrage ohne Query-String zurückgibt. Ein SAX-Parser ist strombasiert und ruft von Ihnen selbst geschriebene Rückrufmethoden auf, um verschiedene, beim Parsen auftretende Ereignisse zu verarbeiten, beispielsweise das Vorkommen eines Starttags. Ein DOM-Parser (Document Object Model) dagegen, arbeitet baumbasiert, das heißt der Parser konstruiert eine Baumdarstellung eines wohlgeformten XML-Dokumentes. Sie können nun eine standardisierte API verwenden, um beispielsweise die Baumstruktur nach gewünschten Elementen zu durchsuchen. Die JAX-P-Bibliothek verwendet den, im Kontext der Extensible Stylesheet Language Transformation (XSLT) gebräuchlichen Begriff „transformieren“ beziehungsweise „Transformation“, um die Umwandlung einer XML-Quellstruktur (zum Beispiel das durch die Anfrage erhaltene „Stück XML“) in eine XML-Zielstruktur (zum Beispiel einen *DOM-Baum*) zu beschreiben. Die entsprechende Anweisung in der `doPost()`-Methode lautet:

```
trans.transform(new StreamSource(xml_stream), dom);
```

Die Referenzvariable `xml_stream` verweist auf ein Objekt vom Typ `ByteArrayInputStream`, das die vom Client gesendeten Bytes repräsentiert, `dom` verweist auf ein Objekt vom Typ `DOMResult`. Es gibt verschiedene Möglichkeiten, einen DOM-Baum zu verarbeiten. Im vorliegenden Fall wird ein Objekt vom Typ *`XPath`* (repräsentiert einen „XPath-Ausdruck“) verwendet, um nach relativ einfachen Mustern zu suchen. Die Anweisung

```
NodeList players = (NodeList)
    xp.evaluate("player", dom.getNode(), XPathConstants.NODESET);
```

fordert beispielsweise eine Liste der `<player>`-Elemente des DOM-Baums (eine „Knotenliste“) an. Die Anweisungen:

```
String name = xp.evaluate("name", dom.getNode());
String nickname = xp.evaluate("nickname", dom.getNode());
```

extrahieren den Namen und Spitznamen des Spielers aus dem DOM-Baum.

[34] Die `doPost()`-Methode respektiert die HTTP-Methode, deren Namen sie hat. Nachdem der Name der neuen Komikertruppe aus dem XML-Dokument in der Anfrage entnommen wurde, führt die Methode eine Prüfung durch:

```
team_name = xp.evaluate("/create_team/name", dom.getNode());
if (team_map.containsKey(team_name)) throw new HTTPException(400); // bad request
```

Die Prüfung stellt fest, ob bereits eine Komikertruppe mit diesem Namen existiert. Eine `POST`-Anfrage ist eine `create`-Operation. Eine bereits vorhandene Komikertruppe kann nicht angelegt, sondern muß über eine `PUT`-Anfrage aktualisiert werden.

[35] Sind die benötigten Informationen über die neue Komikertruppe aus dem XML-Dokument in der Anfrage extrahiert, so werden der *`Map<String,Team>`*- und der *`List<Team>`*-Container aktualisiert, um eine erfolgreich verarbeitete `create`-Operation abzubilden. Die Komikertruppenliste wird in die Form der dauerhaft vorhandenen Datei überführt.

[36] Die beiden übrigen CRUD-Operationen `update` und `delete` werden durch die Methoden `doPut()` und `doDelete()` implementiert. Der `RestfulTeams`-Dienst verlangt, daß eine `DELETE`-Anfrage per Query-String eine bestimmte Komikertruppe angibt. Das Löschen aller Komikertruppen auf einmal ist nicht erlaubt. Im Augenblick kann eine `PUT`-Anfrage nur den Namen einer Komikertruppe ändern. Die Funktionalität könnte aber mühelos erweitert werden, um die Änderungen der Mitglieder- und Spitznamen ebenfalls zu ermöglichen. Die Implementierungen der Methoden `doPut()` und `doDelete()` sind:

```
private Source doDelete(MessageContext msg_ctx) {
    String query_string = (String) msg_ctx.get(MessageContext.QUERY_STRING);

    // Disallow the deletion of all teams at once.
    if (query_string == null) throw new HTTPException(403); // illegal operation
    else {
        String name = get_value_from_qs("name", query_string);
        if (!team_map.containsKey(name)) throw new HTTPException(404);

        // Remove team from Map and List, serialize to file.
        Team team = team_map.get(name);
        teams.remove(team);
        team_map.remove(name);
        serialize();

        // Send response.
        return response_to_client(name + " deleted.");
    }
}

private Source doPut(MessageContext msg_ctx) {
    // Parse the query string.
    String query_string = (String) msg_ctx.get(MessageContext.QUERY_STRING);
    String name = null;
    String new_name = null;

    // Get all teams.
    if (query_string == null) throw new HTTPException(403); // illegal operation
    // Get a named team.
    else {
        // Split query string into name= and new_name= sections
        String[] parts = query_string.split("&");
        if (parts[0] == null || parts[1] == null) throw new HTTPException(403);

        name = get_value_from_qs("name", parts[0]);
        new_name = get_value_from_qs("new_name", parts[1]);
        if (name == null || new_name == null) throw new HTTPException(403);

        Team team = team_map.get(name);
        if (team == null) throw new HTTPException(404);
        team.setName(new_name);
        team_map.put(new_name, team);
        serialize();
    }

    // Send a confirmation to requester.
    return response_to_client("Team " + name + " changed to " + new_name);
}
```

Die Methoden sind einander ähnlich und die Logik wurde soweit wie möglich vereinfacht, um die Aufmerksamkeit auf die charakteristischen Merkmale eines Dienstes im REST-Stil zu richten. Hier der vollständige Quelltext des Dienstes:

```
package ch04.team;

import javax.xml.ws.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.annotation.Resource;
import javax.xml.ws.BindingType;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPException;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.http.HTTPBinding;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.util.Collections;
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;
import java.beans.XMLEncoder;
import java.beans.XMLDecoder;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import java.net.URI;
import java.net.URISyntaxException;
import org.w3c.dom.NodeList;

// The class below is a WebServiceProvider rather than
// the more usual SOAP-based WebService. As a result, the
// service implements the generic Provider interface rather
// than a customized SEI with designated @WebMethods.
@WebServiceProvider

// There are two ServiceModes: PAYLOAD, the default, signals that the service
// wants access only to the underlying message payload (e.g., the
// body of an HTTP POST request); MESSAGE signals that the service wants
// access to entire message (e.g., the HTTP headers and body). In this
// case, the MESSAGE mode lets us check on the request verb.
@ServiceMode(value = javax.xml.ws.Service.Mode.MESSAGE)

// The HTTP_BINDING as opposed, for instance, to a SOAP binding.
@BindingType(value = HTTPBinding.HTTP_BINDING)

// The generic, low-level Provider interface is an alternative
// to the SEI (service endpoint interface) of a SOAP-based
// web service. A Source is a source of the bytes. The invoke
// method expects a source and returns one.
```

```
public class RestfulTeams implements Provider<Source> {

    @Resource
    protected WebServiceContext ws_ctx;

    private Map<String, Team> team_map; // for easy lookups
    private List<Team> teams;           // serialized/deserialized
    private byte[] team_bytes;         // from the persistence file

    private static final String file_name = "teams.ser";
    private static final String post_put_key = "Cargo";

    public RestfulTeams() {
        read_teams_from_file();
        deserialize();
    }

    // Implementation of the Provider interface method: this
    // method handles incoming requests and generates the
    // outgoing response.
    public Source invoke(Source request) {

        if (ws_ctx == null)
            throw new RuntimeException("Injection failed on ws_ctx.");

        if (request == null) System.out.println("null request");
        else System.out.println("non-null request");

        // Grab the message context and extract the request verb.
        MessageContext msg_ctx = ws_ctx.getMessageContext();
        String http_verb = (String)
            msg_ctx.get(MessageContext.HTTP_REQUEST_METHOD);
        http_verb = http_verb.trim().toUpperCase();

        // Act on the verb.
        if      (http_verb.equals("GET"))    return doGet(msg_ctx);
        else if (http_verb.equals("DELETE")) return doDelete(msg_ctx);
        else if (http_verb.equals("POST"))   return doPost(msg_ctx);
        else if (http_verb.equals("PUT"))    return doPut(msg_ctx);
        else throw new HTTPException(405);   // bad verb exception

    }

    private Source doGet(MessageContext msg_ctx) {

        // Parse the query string.
        String query_string = (String)
            msg_ctx.get(MessageContext.QUERY_STRING);

        // Get all teams.
        if (query_string == null)
            return new StreamSource(new ByteArrayInputStream(team_bytes));
        // Get a named team.
        else {
            String name = get_value_from_qs("name", query_string);

            // Check if named team exists.
            Team team = team_map.get(name);
            if (team == null) throw new HTTPException(404); // not found

            // Otherwise, generate XML and return.
            ByteArrayInputStream stream = encode_to_stream(team);
            return new StreamSource(stream);
        }
```

```
}

private Source doPost(MessageContext msg_ctx) {

    Map<String, List> request = (Map<String, List>)
        msg_ctx.get(MessageContext.HTTP_REQUEST_HEADERS);

    List<String> cargo = request.get(post_put_key);
    if (cargo == null) throw new HTTPException(400); // bad request

    String xml = "";
    for (String next : cargo) xml += next.trim();
    ByteArrayInputStream xml_stream = new ByteArrayInputStream(xml.getBytes());
    String team_name = null;

    try {
        // Set up the XPath object to search for the XML elements.
        DOMResult dom = new DOMResult();
        Transformer trans =
            TransformerFactory.newInstance().newTransformer();
        trans.transform(new StreamSource(xml_stream), dom);
        URI ns_URI = new URI("create_team");

        XPathFactory xpf = XPathFactory.newInstance();
        XPath xp = xpf.newXPath();
        xp.setNamespaceContext(new NSResolver("", ns_URI.toString()));

        team_name = xp.evaluate("/create_team/name", dom.getNode());

        if (team_map.containsKey(team_name))
            throw new HTTPException(400); // bad request

        List<Player> team_players = new ArrayList<Player>();

        NodeList players = (NodeList)
            xp.evaluate("player",
                        dom.getNode(),
                        XPathConstants.NODESET);

        for (int i = 1; i <= players.getLength(); i++) {
            String name = xp.evaluate("name", dom.getNode());
            String nickname = xp.evaluate("nickname", dom.getNode());
            Player player = new Player(name, nickname);
            team_players.add(player);
        }
        // Add new team to the in-memory map and save List to file.
        Team t = new Team(team_name, team_players);
        team_map.put(team_name, t);
        teams.add(t);
        serialize();
    }
    catch(URISyntaxException e) {
        throw new HTTPException(500);   // internal server error
    }
    catch(TransformerConfigurationException e) {
        throw new HTTPException(500);   // internal server error
    }
    catch(TransformerException e) {
        throw new HTTPException(500);   // internal server error
    }
    catch(XPathExpressionException e) {
        throw new HTTPException(400);   // bad request
    }
```

```
        // Send a confirmation to requester.
        return response_to_client("Team " + team_name + " created.");
    }

    private Source doPut(MessageContext msg_ctx) {

        // Parse the query string.
        String query_string = (String) msg_ctx.get(MessageContext.QUERY_STRING);
        String name = null;
        String new_name = null;

        // Get all teams.
        if (query_string == null)
            throw new HTTPException(403); // illegal operation
        // Get a named team.
        else {
            // Split query string into name= and new_name= sections
            String[] parts = query_string.split("&");
            if (parts[0] == null || parts[1] == null)
                throw new HTTPException(403);

            name = get_value_from_qs("name", parts[0]);
            new_name = get_value_from_qs("new_name", parts[1]);
            if (name == null || new_name == null)
                throw new HTTPException(403);

            Team team = team_map.get(name);
            if (team == null) throw new HTTPException(404);
            team.setName(new_name);
            team_map.put(new_name, team);
            serialize();
        }

        // Send a confirmation to requester.
        return response_to_client("Team " + name + " changed to " + new_name);

    }

    private Source doDelete(MessageContext msg_ctx) {

        String query_string = (String)
            msg_ctx.get(MessageContext.QUERY_STRING);

        // Disallow the deletion of all teams at once.
        if (query_string == null)
            throw new HTTPException(403);     // illegal operation
        else {
            String name = get_value_from_qs("name", query_string);
            if (!team_map.containsKey(name))
                throw new HTTPException(404); // not found

            // Remove team from Map and List, serialize to file.
            Team team = team_map.get(name);
            teams.remove(team);
            team_map.remove(name);
            serialize();

            // Send response.
            return response_to_client(name + " deleted.");
        }

    }

    private StreamSource response_to_client(String msg) {
```

```
        HttpResponse response = new HttpResponse();
        response.setResponse(msg);
        ByteArrayInputStream stream = encode_to_stream(response);
        return new StreamSource(stream);

    }

    private ByteArrayInputStream encode_to_stream(Object obj) {

        // Serialize object to XML and return
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        XMLEncoder enc = new XMLEncoder(stream);
        enc.writeObject(obj);
        enc.close();
        return new ByteArrayInputStream(stream.toByteArray());

    }

    private String get_value_from_qs(String key, String qs) {

        String[] parts = qs.split("=");

        // Check if query string has form: name=<team name>
        if (!parts[0].equalsIgnoreCase(key))
            throw new HTTPException(400); // bad request
        return parts[1].trim();

    }

    private void read_teams_from_file() {

        try {
            String cwd = System.getProperty ("user.dir");
            String sep = System.getProperty ("file.separator");
            String path = get_file_path();
            int len = (int) new File(path).length();
            team_bytes = new byte[len];
            new FileInputStream(path).read(team_bytes);
        }
        catch(IOException e) { System.err.println(e); }

    }

    private void deserialize() {

        // Deserialize the bytes into a list of teams
        XMLDecoder dec =
            new XMLDecoder(new ByteArrayInputStream(team_bytes));
        teams = (List<Team>) dec.readObject();

        // Create a map for quick lookups of teams.
        team_map = Collections.synchronizedMap(new HashMap<String, Team>());
        for (Team team : teams)
            team_map.put(team.getName(), team);

    }

    private void serialize() {

        try {
            String path = get_file_path();
            BufferedOutputStream out =
                new BufferedOutputStream(new FileOutputStream(path));
            XMLEncoder enc = new XMLEncoder(out);
            enc.writeObject(teams);
            enc.close();
```

```
            out.close();
        }
        catch(IOException e) { System.err.println(e); }
    }

    private String get_file_path() {
        String cwd = System.getProperty ("user.dir");
        String sep = System.getProperty ("file.separator");
        return cwd + sep + "ch04" + sep + "team" + sep + file_name;
    }
}
```

Der folgende überarbeitete Perl-Client testet den Dienst mittels einer Reihe von Anfragen:

```
#!/usr/bin/perl

use strict;
use LWP;
use XML::XPath;
use Encode;
use constant true   =>  1;
use constant false  =>  0;

# Create the user agent.
my $ua = LWP::UserAgent->new;

my $base_uri = 'http://localhost:8888/teams';

# GET teams
send_GET($base_uri, false); # false means no query string

# GET teams?name=MarxBrothers
send_GET($base_uri . '?name=MarxBrothers', true);

$base_uri = $base_uri;
send_POST($base_uri);

# Check that POST worked
send_GET($base_uri . '?name=SmothersBrothers', true);
send_DELETE($base_uri . '?name=SmothersBrothers');

# Recreate the Smothers Brothers as a check.
send_POST($base_uri);

# Change name and check.
send_PUT($base_uri . '?name=SmothersBrothers&new_name=SmuthersBrothers');
send_GET($base_uri . '?name=SmuthersBrothers', true);

sub send_GET {
    my ($uri, $qs_flag) = @_;

    # Send the request and get the response.
    my $req = HTTP::Request->new(GET => $uri);
    my $res = $ua->request($req);

    # Check for errors.
    if ($res->is_success) {
        parse_GET($res->content, $qs_flag); # Process raw XML on success
    }
    else {
        print $res->status_line, "\n";      # Print error code on failure
    }
}
```

```
sub send_POST {
    my ($uri) = @_;

    my $xml = <<EOS;
      <create_team>
         <name>SmothersBrothers</name>
         <player>
           <name>Thomas</name>
           <nickname>Tom</nickname>
         </player>
         <player>
           <name>Richard</name>
           <nickname>Dickie</nickname>
         </player>
      </create_team>
EOS
    # Send request and capture response.
    my $bytes = encode('iso-8859-1', $xml); # encoding is Latin-1
    my $req = HTTP::Request->new(POST => $uri, ['Cargo' => $bytes]);
    my $res = $ua->request($req);

    # Check for errors.
    if ($res->is_success) {
        parse_SIMPLE("POST", $res->content); # Process raw XML on success
    }
    else {
        print $res->status_line, "\n";        # Print error code on failure
    }
}

sub send_DELETE {
    my $uri = shift;

    # Send the request and get the response.
    my $req = HTTP::Request->new(DELETE => $uri);
    my $res = $ua->request($req);

    # Check for errors.
    if ($res->is_success) {
        parse_SIMPLE("DELETE", $res->content);   # Process raw XML on success
    }
    else {
        print $res->status_line, "\n"; # Print error code on failure
    }
}

sub send_PUT {
    my $uri = shift;

    # Send the request and get the response.
    my $req = HTTP::Request->new(PUT => $uri);
    my $res = $ua->request($req);

    # Check for errors.
    if ($res->is_success) {
        parse_SIMPLE("PUT", $res->content);   # Process raw XML on success
    }
    else {
        print $res->status_line, "\n"; # Print error code on failure
    }
}
```

```
sub parse_SIMPLE {
    my $verb = shift;
    my $raw_xml = shift;
    print "\nResponse on $verb: \n$raw_xml;;;\n";
}

sub parse_GET {
    my ($raw_xml) = @_;
    print "\nThe raw XML response is:\n$raw_xml\n;;;\n";

    # For all teams, extract and print out their names and members
    my $xp = XML::XPath->new(xml => $raw_xml);
    foreach my $node ($xp->find('//object/void/string')->get_nodelist) {
        print $node->string_value, "\n";
    }
}
```

4.4 Die komplementären Interfaces Provider und Dispatch

[37] Die Clients des `RestfulTeams`-Dienstes in den Unterabschnitten 4.3.2 und 4.3.5 übertragen Anfragedaten, wie den Namen einer Komikertruppe, im Query-String (zum Beispiel bei `GET`-Anfragen) sowie mittels optionaler HTTP-Header (zum Beispiel bei `POST`-Anfragen) an den Dienst. `GET`- und `DELETE`-Anfragen haben keinen Körper, im Gegensatz zu `POST`- und `PUT`-Anfragen. Die Clients des `RestfulTeams`-Dienstes senden ausschließlich körperlose Anfragen. Selbst bei `POST`- und `PUT`-Anfragen werden die Informationen über die neue oder zu aktualisierende Komikertruppe im Headerabschnitt und nicht im Körper der HTTP-Nachricht transportiert.

[38] Die Entwicklung des `RestfulTeams`-Dienstes in Abschnitt 4.3 dokumentiert die Flexibilität des REST-Stils. Die nächste Version in diesem Abschnitt zeigt, wie der Körper einer `POST`-Anfrage mit Hilfe des Interfaces *`javax.xml.ws.Dispatch`* used/ausgewertet/geschrieben werden kann. *`Dispatch`* ist das clientseitige Äquivalent des serviceseitigen Interfaces *`javax.xml.ws.Provider`*. Die erste Fassung des `RestfulTeams`-Dienstes hat bereits gezeigt, daß *`Provider`* auf der Seite des Dienstes auch ohne clientseitige Implementierung von *`Dispatch`* genutzt werden kann. Das Beispiel in Unterabschnitt 4.4.3 dokumentiert, daß eine *`Dispatch`*-Implementierung auf der Seite des Clients unabhängig von der Implementierung des Dienstes verwendet werden kann. Dennoch sind die Interfaces *`Dispatch`* und *`Provider`* ein natürliches Paar.

[39] Ein Dienst im REST-Stil, der das Interface *`Provider`* implementiert, verfügt über die Methode:

```
public Source invoke(Source request)
```

Ein Client, der das Interface *`Dispatch`* implementiert wird gelegentlich auch als *dynamischer Stellvertreter des Dienstes* bezeichnet und verfügt über eine clientseitige Implementierung dieser Methode. Das Interface *`javax.xml.transform.Source`* repräsentiert eine „XML-Quelle“ (ein XML-Dokument), die sich als Eingabe einer XSLT-Transformation eignet. Die Transformation ist durch ein Objekt der abstrakten Klasse `javax.xml.transform.Transformer` vertreten und liefert ein Objekt vom Typ *`javax.xml.transform.Result`*, das typischerweise wiederum ein XML-Dokument ist. Die Beziehung zwischen *`Dispatch`* und *`Provider`* unterstützt einen natürlichen Austausch von XML-Dokumenten zwischen Client und Server:

- Der Client ruft die *`Dispatch`*-Methode `invoke()` auf und übergibt ein XML-Dokument in Form eines Objektes vom Typ *`Source`*. Ist für die Anfrage kein XML-Dokument erforderlich, so kann anstelle des *`Source`*-Argumentes `null` übergeben werden.

- Die serviceseitige Laufzeitbibliothek übergibt die Anfrage an die *Provider*-Methode `invoke()`, deren *Source*-Argument dem clientseitigen Argument entspricht.
- Der Dienst transformiert das erhaltene XML-Dokument in ein Objekt vom Typ *Result*, zum Beispiel einen DOM-Baum, verarbeitet dieses Objekt in einer der Applikation dienlichen Weise und gibt ein XML-Dokument in Form eines Objektes vom Typ *Source* an den Client zurück. Ist keine Antwort erforderlich, so kann anstelle des *Source*-Argumentes `null` übergeben werden.
- Die *Dispatch*-Methode `invoke()` gibt eine Referenz vom Typ *Source* auf ein vom Dienst erhaltenes Objekt zurück, welches der Client in ein entsprechendes Objekt vom Typ *Result* umwandelt und nach Bedarf verarbeitet.

Die Tatsache, daß die *Provider*-Methode `invoke()` und die *Dispatch*-Methode `invoke()` identische Signaturen haben, hebt die natürliche Passung hervor.

4.4.1 Der RabbitCounterProvider-Dienst (REST-Stil)

[40] Der `RabbitCounterProvider`-Dienst ist das Äquivalent des SOAP-basierten `RabbitCounter`-Dienstes aus Unterabschnitt 3.1.6 (Beispiel 3.4) und akzeptiert `POST`-, `GET`- und `DELETE`-Anfragen. Eine `POST`-Anfrage ist eine `create`-Operation (CRUD-Operation) und erzeugt eine Liste von Fibonaccizahlen, die der Dienst für spätere `read`- oder `delete`-Operationen zwischenspeichert. `POST`-Anfragen werden von der `doPost()`-Methode beantwortet, die ein Argument vom Typ *Source* erwartet. Das Argument repräsentiert ein XML-Dokument, zum Beispiel:

```
<fib:request xmlns:fib = 'urn:fib'>[1, 2, 3, 4]</fib:request>
```

Das XML-Dokument stellt eine Liste ganzer Zahlen dar, deren Fibonacciwerte berechnet werden sollen. Die Methoden `doGet()` und `doDelete()` behandeln `GET`- beziehungsweise `POST`-Anfragen, die beide keinen Körper besitzen, so daß die Methoden keinen Parameter vom Typ *Source* haben. Alle drei Methoden geben eine Referenz vom Typ *Source* zurück, der auf eine Bestätigung im XML-Format verweist. Die obige `POST`-Anfrage beispielsweise, gibt das folgende XML-Dokument zurück:

```
<fib:response xmlns:fib = 'urn:fib'>POSTed[1, 1, 2, 3]</fib:response>
```

Die beiden anderen Methoden geben ebenfalls operationsbezogene Bestätigungen zurück.

[41] Hier der Quelltext der Klasse `RabbitCounterProvider`:

```
package ch04.dispatch;

import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
import java.util.Collection;
import javax.xml.ws.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.annotation.Resource;
import javax.xml.ws.BindingType;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPException;
import javax.xml.ws.WebServiceProvider;
```

```
import javax.xml.ws.http.HTTPBinding;
import java.io.ByteArrayInputStream;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;

// The RabbitCounter service implemented as REST style rather than SOAP based.
@WebServiceProvider
@BindingType(value = HTTPBinding.HTTP_BINDING)
public class RabbitCounterProvider implements Provider<Source> {

    @Resource
    protected WebServiceContext ws_ctx;

    // stores previously computed values
    private Map<Integer, Integer> cache =
        Collections.synchronizedMap(new HashMap<Integer, Integer>());

    private final String xml_start = "<fib:response xmlns:fib = 'urn:fib'>";
    private final String xml_stop = "</fib:response>";
    private final String uri = "urn:fib";

    public Source invoke(Source request) {

        // Filter on the HTTP request verb
        if (ws_ctx == null)
            throw new RuntimeException("DI failed on ws_ctx.");

        // Grab the message context and extract the request verb.
        MessageContext msg_ctx = ws_ctx.getMessageContext();
        String http_verb =
            (String) msg_ctx.get(MessageContext.HTTP_REQUEST_METHOD);
        http_verb = http_verb.trim().toUpperCase();

        // Act on the verb.
        if      (http_verb.equals("GET"))    return doGet();
        else if (http_verb.equals("DELETE")) return doDelete();
        else if (http_verb.equals("POST"))   return doPost(request);
        else throw new HTTPException(405);   // bad verb exception

    }

    private Source doPost(Source request) {

        if (request == null)
            throw new HTTPException(400); // bad request

        String nums = extract_request(request);
        // Extract the integers from a string such as: "[1, 2, 3]"
        nums = nums.replace('[', '\0');
        nums = nums.replace(']', '\0');
        String[] parts = nums.split(",");
        List<Integer> list = new ArrayList<Integer>();
        for (String next : parts) {
            int n = Integer.parseInt(next.trim());
            cache.put(n, countRabbits(n));
            list.add(cache.get(n));
```

```
        }
        String xml = xml_start + "POSTed: " + list.toString() + xml_stop;
        return make_stream_source(xml);
    }

    private Source doGet() {
        Collection<Integer> list = cache.values();
        String xml = xml_start + "GET: " + list.toString() + xml_stop;
        return make_stream_source(xml);
    }

    private Source doDelete() {
        cache.clear();
        String xml = xml_start + "DELETE: Map cleared." + xml_stop;
        return make_stream_source(xml);
    }

    private String extract_request(Source request) {

        String request_string = null;
        try {
            DOMResult dom_result = new DOMResult();
            Transformer trans = TransformerFactory.newInstance().newTransformer();
            trans.transform(request, dom_result);
            XPathFactory xpf = XPathFactory.newInstance();
            XPath xp = xpf.newXPath();
            xp.setNamespaceContext(new NSResolver("fib", uri));
            request_string = xp.evaluate("/fib:request", dom_result.getNode());
        }
        catch(TransformerConfigurationException e) { System.err.println(e); }
        catch(TransformerException e) { System.err.println(e); }
        catch(XPathExpressionException e) { System.err.println(e); }

        return request_string;

    }

    private StreamSource make_stream_source(String msg) {
        System.out.println(msg);
        ByteArrayInputStream stream = new ByteArrayInputStream(msg.getBytes());
        return new StreamSource(stream);
    }

    private int countRabbits(int n) {

        if (n < 0)
            throw new HTTPException(403); // forbidden

        // Easy cases.
        if (n < 2)
            return n;

        // Return cached values if present.
        if (cache.containsKey(n)) return cache.get(n);
        if (cache.containsKey(n - 1) && cache.containsKey(n - 2)) {
            cache.put(n, cache.get(n - 1) + cache.get(n - 2));
            return cache.get(n);
        }

        // Otherwise, compute from scratch, cache, and return.
        int fib = 1, prev = 0;
        for (int i = 2; i <= n; i++) {
            int temp = fib;
```

```
                fib += prev;
                prev = temp;
            }
            cache.put(n, fib);
            return fib;
        }

    }
```

Die vier Zeilen

```
XPathFactory xpf = XPathFactory.newInstance();
XPath xp = xpf.newXPath();
xp.setNamespaceContext(new NSResolver("fib", uri));
request_string = xp.evaluate("/fib:request", dom_result.getNode());
```

verdienen nähere Betrachtung, da die Klasse `NSResolver` auch im `RestfulTeams`-Dienst auftritt. Die `evaluate()`-Methode erwartet zwei Argumente: Einen XPath-Ausdruck, hier `/fib:request` und den `DOMResult`-Knoten, der die gewünschten Zeichen zwischen dem Starttag `<fib:request>` und dem gehörigen Endtag `</fib:request>` enthält. Das `fib` bei `fib:request` ist der Stellvertreter eines Namensraum-URIs, hier `urn:fib`. Das vollständige Starttag im XML-Dokument der Anfrage ist:

```
<fib:request xmlns:fib = 'urn:fib'>
```

Die Klasse `ch04.dispatch.NSResolver`, „NS“ ist die Abkürzung für „Namensraum“ (*namespace*), stellt die Abbildungen von `fib` auf `urn:fib` und umgekehrt zur Verfügung:

```
package ch04.dispatch;

import java.util.Collections;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
import javax.xml.namespace.NamespaceContext;

public class NSResolver implements NamespaceContext {

    private Map<String, String> prefix2uri;
    private Map<String, String> uri2prefix;

    public NSResolver() {

        if (prefix2uri == null) prefix2uri =
            Collections.synchronizedMap(new HashMap<String, String>());
        if (uri2prefix == null) uri2prefix =
            Collections.synchronizedMap(new HashMap<String, String>());
    }

    public NSResolver(String prefix, String uri) {
        this();
        prefix2uri.put(prefix, uri);
        uri2prefix.put(uri, prefix);
    }

    public String getNamespaceURI(String prefix) {
        return prefix2uri.get(prefix);
    }

    public String getPrefix(String uri) {
        return uri2prefix.get(uri);
    }
```

```
        public Iterator getPrefixes(String uri) {
            return uri2prefix.keySet().iterator();
        }

    }
```

Die Klasse `NSResolver` stellt den Namensraumkontext (eine Implementierung des Interfaces *`javax.xml.namespace.NamespaceContext`*) für die XPath-Suche zur Verfügung, verknüpft also einen Namensraum-URI und seinen Stellvertreter. Die korrekte Funktionsweise der Applikation setzt voraus, daß Client und Dienst denselben Namensraum-URI verwenden, hier den einfach strukturierten URI `urn:fib`.

4.4.2 Mehr über das Dispatch-Interface

[42] Der folgende, auf einer Implementierung des *`Dispatch`*-Interfaces basierende Client des `RabbitCounterProvider`-Dienstes im REST-Stil, erinnert an den Client der SOAP-basierten Version aus Unterabschnitt 3.1.5 oder 3.1.7. Der Client erzeugt identifizierende `QName`-Objekte für einen Dienst und einen Port, ein Objekt des Dienstes und adds a port und einen mit dem Port verbundenen dynamischen Stellvertreter des Dienstes:

```
QName service_name = new QName("rcService", ns_URI.toString()); // uri is urn:fib
QName port = new QName("rcPort", ns_URI.toString());
String endpoint = "http://localhost:9876/fib";

// Now create a service proxy or dispatcher.
Service service = Service.create(service_name);
service.addPort(port, HTTPBinding.HTTP_BINDING, endpoint);
Dispatch<Source> dispatch =
    service.createDispatch(port, Source.class, Service.Mode.PAYLOAD);
```

This client-side dispatch object can dispatch XML documents as requests to the service as XML Source instances. Ein XML-Dokument wird durch einen Aufruf der `invoke()`-Methode an den Dienst übergeben. Die folgenden Zeilen bereiten ein XML-Dokument als Körper einer `POST`-Anfrage vor:

```
String xml_start = "<fib:request xmlns:fib = 'urn:fib'>";
String xml_end = "</fib:request>";
List<Integer> nums = new ArrayList<Integer>();
for (int i = 0; i < 12; i++) nums.add(i + 1);
String xml = xml_start + nums.toString() + xml_end;
```

Anschließend wird das XML-Dokument für die Anfrage in einem Objekt vom Typ `javax.xml.transform.stream.StreamSource` verpackt und per `invoke()` an den Dienst gesendet:

```
StreamSource source = null;
if (data != null) source = make_stream_source(data.toString()); // data = XML doc
Source result = dispatch.invoke(source);
display_result(result, uri); // do an XPath search of the resturned XML
```

Die Operationen `GET` und `DELETE` brauchen kein XML-Dokument. Daher ist das *`Source`*-Argument in beiden Fällen `null`. Die folgenden Zeilen dokumentieren die an den Dienst gesendeten Anfragen und die von dort erhaltenen Antworten:

```
Request: <fib:request xmlns:fib = 'urn:fib'>
           [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
         </fib:request>
POSTed: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

```
Request: null
GET: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]

Request: null
DELETE: Map cleared.

Request: null
GET: []

Request: <fib:request xmlns:fib = 'urn:fib'>
            [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,...,20, 21, 22, 23, 24]
         </fib:request>
POSTed: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,..., 10946, 17711, 28657, 46368]

Request: null
GET: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,..., 10946, 6765, 28657, 17711, 46368]
```

Schließlich noch der gesamte Quelltext der Klasse `DispatchClient`:

```
import java.net.URI;
import java.net.URISyntaxException;
import java.io.ByteArrayInputStream;
import java.util.Map;
import java.util.List;
import java.util.ArrayList;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.Dispatch;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.ws.handler.MessageContext;
import org.w3c.dom.NodeList;
import ch04.dispatch.NSResolver;

class DispatchClient {

    public static void main(String[] args) throws Exception {
        new DispatchClient().setup_and_test();
    }

    private void setup_and_test() {

        // Create identifying names for service and port.
        URI ns_URI = null;
        try {
            ns_URI = new URI("urn:fib");
        }
        catch(URISyntaxException e) { System.err.println(e); }

        QName service_name = new QName("rcService", ns_URI.toString());
        QName port = new QName("rcPort", ns_URI.toString());
```

```
        String endpoint = "http://localhost:9876/fib";

        // Now create a service proxy or dispatcher.
        Service service = Service.create(service_name);
        service.addPort(port, HTTPBinding.HTTP_BINDING, endpoint);
        Dispatch<Source> dispatch =
            service.createDispatch(port, Source.class, Service.Mode.PAYLOAD);

        // Send some requests.
        String xml_start = "<fib:request xmlns:fib = 'urn:fib'>";
        String xml_end = "</fib:request>";

        // To begin, a POST to create some Fibonacci numbers.
        List<Integer> nums = new ArrayList<Integer>();
        for (int i = 0; i < 12; i++) nums.add(i + 1);
        String xml = xml_start + nums.toString() + xml_end;
        invoke(dispatch, "POST", ns_URI.toString(), xml);

        // GET request to test whether the POST worked.
        invoke(dispatch, "GET", ns_URI.toString(), null);

        // DELETE request to remove the list
        invoke(dispatch, "DELETE", ns_URI.toString(), null);

        // GET to test whether the DELETE worked.
        invoke(dispatch, "GET", ns_URI.toString(), null);

        // POST to repopulate and a final GET to confirm
        nums = new ArrayList<Integer>();
        for (int i = 0; i < 24; i++) nums.add(i + 1);
        xml = xml_start + nums.toString() + xml_end;
        invoke(dispatch, "POST", ns_URI.toString(), xml);
        invoke(dispatch, "GET", ns_URI.toString(), null);

    }

    private void invoke(Dispatch<Source> dispatch,
                        String verb,
                        String uri,
                        Object data) {

        Map<String, Object> request_context = dispatch.getRequestContext();
        request_context.put(MessageContext.HTTP_REQUEST_METHOD, verb);

        System.out.println("Request: " + data);

        // Invoke
        StreamSource source = null;
        if (data != null) source = make_stream_source(data.toString());
        Source result = dispatch.invoke(source);
        display_result(result, uri);

    }

    private void display_result(Source result, String uri) {

        DOMResult dom_result = new DOMResult();

        try {

            Transformer trans = TransformerFactory.newInstance().newTransformer();
            trans.transform(result, dom_result);

            XPathFactory xpf = XPathFactory.newInstance();
            XPath xp = xpf.newXPath();
            xp.setNamespaceContext(new NSResolver("fib", uri));
```

```
            String result_string =
                xp.evaluate("/fib:response", dom_result.getNode());
            System.out.println(result_string);
        }
        catch(TransformerConfigurationException e) { System.err.println(e); }
        catch(TransformerException e) { System.err.println(e); }
        catch(XPathExpressionException e) { System.err.println(e); }
    }
    private StreamSource make_stream_source(String msg) {
        ByteArrayInputStream stream = new ByteArrayInputStream(msg.getBytes());
        return new StreamSource(stream);
    }
}
```

4.4.3 Ein Dispatch-Client für einen SOAP-basierten Dienst

[43] Der *Dispatch*-Client ist in dem Sinne flexibel, daß er sowohl Anfragen an einen Webservice im REST-Stil, als auch an einen SOAP-basierten Dienst veranlassen kann. Dieser Unterabschnitt zeigt, wie ein SOAP-basierter Dienst angesprochen werden kann, als sei er ein Dienst im REST-Stil. Diese Verwendungsweise des *Dispatch*-Interfaces unterstreicht, daß SOAP-basierte Webservices mit HTTP als Transportprotokoll (die meisten Dienste verwenden diese Kombination) ein Spezialfall der Dienste im REST-Stil sind. Die SOAP-Bibliotheken ersparen dem Programmierer die direkte XML-Verarbeitung entweder auf der Dienst- oder auf der Clientseite. Behandler sind eine Ausnahme von dieser Regel.

[44] Die folgende Klasse `DispatchClientTS` verwendet einen dynamischen Stellvertreter, um eine Anfrage an den SOAP-basierten *ch01.ts.TimeServer*-Dienst aus Abschnitt 1.2 zu senden. Der Dienst unterstützt zwei Operationen: Eine liefert das aktuelle Datum und die aktuelle Uhrzeit in menschenlesbarer Form zurück, die andere in Form der verstrichenen Millisekunden seit der Unix-Epoche. Nun der Quelltext der Klasse `DispatchClientTS`:

```
import java.util.Map;
import java.net.URI;
import java.net.URISyntaxException;
import java.io.ByteArrayInputStream;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.Dispatch;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.ws.handler.MessageContext;
import ch04.dispatch.NSResolver;
```

```
// Dispatch client against the SOAP-based TimeServer service
class DispatchClientTS {

    public static void main(String[] args) throws Exception {
        new DispatchClientTS().send_and_receive_SOAP();
    }

    private void send_and_receive_SOAP() {

        // Create identifying names for service and port.
        URI ns_URI = null;
        try {
            ns_URI = new URI("http://ts.ch01/");      // from WSDL
        }
        catch(URISyntaxException e) { System.err.println(e); }

        QName service_name = new QName("tns", ns_URI.toString());
        QName port = new QName("tsPort", ns_URI.toString());
        String endpoint = "http://localhost:9876/ts"; // from WSDL

        // Now create a service proxy or dispatcher.
        Service service = Service.create(service_name);
        service.addPort(port, HTTPBinding.HTTP_BINDING, endpoint);
        Dispatch<Source> dispatch =
            service.createDispatch(port, Source.class, Service.Mode.PAYLOAD);

        // Send a request.
        String soap_request =
            "<?xml version='1.0' encoding='UTF-8'?> " +
            "<soap:Envelope " +
            "soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' " +
            "xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' " +
            "xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/' " +
            "xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' " +
            "xmlns:tns='http://ts.ch01/' " +
            "xmlns:xsd='http://www.w3.org/2001/XMLSchema'> " +
            "<soap:Body>" +
            "<tns:getTimeAsElapsed xsi:nil='true' />" +
            "</soap:Body>" +
            "</soap:Envelope>";

        Map<String, Object> request_context = dispatch.getRequestContext();
        request_context.put(MessageContext.HTTP_REQUEST_METHOD, "POST");
        StreamSource source = make_stream_source(soap_request);
        Source result = dispatch.invoke(source);
        display_result(result, ns_URI.toString());

    }

    private void display_result(Source result, String uri) {

        DOMResult dom_result = new DOMResult();

        try {

            Transformer trans = TransformerFactory.newInstance().newTransformer();
            trans.transform(result, dom_result);

            XPathFactory xpf = XPathFactory.newInstance();
            XPath xp = xpf.newXPath();
            xp.setNamespaceContext(new NSResolver("tns", uri));
            // In original version, "//time_result" instead
            String result_string = xp.evaluate("//return", dom_result.getNode());
            System.out.println(result_string);
```

```
            }
            catch(TransformerConfigurationException e) { System.err.println(e); }
            catch(TransformerException e) { System.err.println(e); }
            catch(XPathExpressionException e) { System.err.println(e); }
        }
        private StreamSource make_stream_source(String msg) {
            ByteArrayInputStream stream = new ByteArrayInputStream(msg.getBytes());
            return new StreamSource(stream);
        }
    }
```

Das SOAP-Dokument ist als eine lange Zeichenkette hartkodiert. Die restliche Konfiguration ist nicht schwierig. Nachdem Erzeugen eines `Service`-Objektes und dem Anlegen eines Ports mit dem Endpunkt des *`TimeServer`*-Dienstes, wird ein Stellvertreterobjekt mit dem Betriebsmodus *`Service.Mode.PAYLOAD`* erzeugt, so daß das SOAP-Dokument der Anfrage als XML-Quelle im Körper der HTTP-Anfrage an den Dienst gesendet wird. Der SOAP-basierte Dienst antwortet mit einem SOAP-Envelope, der mit Hilfe eines Objektes vom Typ *`XPath`* nach dem ganzzahligen Wert durchsucht wird, der die verstrichenen Millisekunden angibt. Ein Aufruf der der Operation `getTimeAsElapsed()` über den REST-Client ergab bei mir 1'214'514'573'623 Millisekunden (die Striche wurden zur besseren Lesbarkeit nachträglich ergänzt).

4.5 Implementierung eines Dienstes im REST-Stil als HttpServlet

[45] Dieser Abschnitt beginnt mit einer kurzen Zusammenfassung zum Thema „Servlets". Die Verwendung von Servlets zum Anbieten von Diensten im REST-Stil steht dabei im Vordergrund. Die abstrakte Klasse `HttpServlet` ist von der ebenfalls abstrakten Klasse `GenericServlet` abgeleitet, die wiederum das Interface *`Servlet`* implementiert. Alle drei Typen liegen in den Packages `javax.servlet` beziehungsweise `javax.servlet.http`, die nicht zur SE 6 gehören. Das Interface *`Servlet`* deklariert fünf Methoden, unter denen die `service()`-Methode am wichtigsten ist und vom Webcontainer bei jeder Anfrage an ein Servlet aufgerufen wird. Die `service()`-Methode hat zwei Parameter, je einen vom Typ *`ServletRequest`* beziehungsweise *`ServletResponse`*. Der *`ServletRequest`*-Parameter referenziert ein Objekt, das Ähnlichkeit mit einem Container vom Typ *`Map`* hat und die vom Client erfaßten Anfragedaten beinhaltet. Der *`ServletResponse`*-Parameter referenziert ein Objekt, das eine Netzwerkverbindung zum Client zurück bereithält. Die Klasse `GenericServlet` implementiert die in *`Servlet`* deklarierten Methoden in einer bezüglich des Transportprotokolls neutralen Weise, während die Unterklasse `HttpServlet` HTTP-spezifische Implementierungen dieser Methoden definiert. Dementsprechend, haben die Parameter der `service()`-Methode der Klasse `HttpServlet` die Typen *`HttpServletRequest`* beziehungsweise *`HttpServletResponse`*. Die Klasse `HttpServlet` bewerkstelligt außerdem um die Weitergabe der Anfragen an die verarbeitende Einheit innerhalb der Anwendung (*request dispatching*): Die `service()`-Methode leitet eingehende `GET`-Anfragen an die `doGet()`-Methode weiter, `POST`-Anfragen an die `doPost()`-Methode und so weiter. Abbildung 6.2 zeigt einen Servletcontainer mit einigen Servlets.

[46] Die `doXXX()`-Methoden der Klasse `HttpServlet` sind funktionslose, genaugenommen rudimentär implementierte (das heißt „nicht voll ausgebildete") Methoden, die Sie je nach Bedarf in einer abgeleiteten Klasse überschreiben müssen. Ist beispielsweise die Klasse `MyServlet` von `HttpServlet` abgeleitet, überschreibt aber nur die `doGet()`-Methode, nicht aber `doPost()`, so ist `doPost()` bei `MyServlet`-Objekten eine rudimentär implementierte Methode.

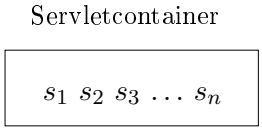

Abbildung 4.2: Servletcontainer mit einigen Servlets.

[47] Servlets vom Typ `HttpServlet` sind aus zwei Gründen eine natürliche und komfortable Möglichkeit zur Implementierung eines Dienstes im REST-Stil. Ein solches Servlet verfügt über Methoden wie `doGet()` und `doDelete()`, die mit den HTTP-Methoden zusammenpassen und bei Bedarf vom Webcontainer als Rückrufmethoden aufgerufen werden. Jede `doXXX()`-Methode hat zwei Parameter der Typen *`HttpServletRequest`* beziehungsweise *`HttpServletResponse`*, so daß ein einheitliches Muster zur Verarbeitung von Anfragen gegeben ist: Die vom Client gesendeten Daten werden im Methodenkörper über den *`HttpServletRequest`*-Parameter abgefragt und in der erforderlichen Weise verarbeitet. Anschließend wird über den mit dem *`HttpServletResponse`*-Parameter verbundenen Ausgabestrom eine Antwort an den Client zurückgesendet.

4.5.1 Das RabbitCounterServlet

[48] Die folgende Klasse `RabbitCounterServlet` ist eine servletbasierte Version im REST-Stil des SOAP-basierten `RabbitCounter`-Dienstes aus Unterabschnitt 3.1.6. Die Logik ist bewußt einfach, um die Aufmerksamkeit auf die Frage zu richten, warum das Servlet eine so attraktive Implementierungsvariante für Dienste im REST-Stil ist:

```
package ch04.rc;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.http.HTTPException;
import java.util.Collections;
import java.util.Map;
import java.util.HashMap;
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;
import java.io.IOException;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.OutputStream;
import java.beans.XMLEncoder;

public class RabbitCounterServlet extends HttpServlet {

    private Map<Integer, Integer> cache;

    // Executed when servlet is first loaded into container.
    public void init() {
        cache = Collections.synchronizedMap(new HashMap<Integer, Integer>());
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) {

        String num = request.getParameter("num");

        // If no query string, assume client wants the full list
```

```
        if (num == null) {
            Collection<Integer> fibs = cache.values();
            send_typed_response(request, response, fibs);
        }
        else {
            try {
                Integer key = Integer.parseInt(num.trim());
                Integer fib = cache.get(key);
                if (fib == null) fib = -1;
                send_typed_response(request, response, fib);
            }
            catch(NumberFormatException e) {
                send_typed_response(request, response, -1);
            }
        }

    }

    public void doPost(HttpServletRequest request, HttpServletResponse response) {

        String nums = request.getParameter("nums");

        if (nums == null)
            throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);

        // Extract the integers from a string such as: "[1, 2, 3]"
        nums = nums.replace('[', '\0');
        nums = nums.replace(']', '\0');
        String[] parts = nums.split(", ");
        List<Integer> list = new ArrayList<Integer>();
        for (String next : parts) {
            int n = Integer.parseInt(next.trim());
            cache.put(n, countRabbits(n));
            list.add(cache.get(n));
        }
        send_typed_response(request, response, list + " added.");

    }

    public void doDelete(HttpServletRequest request, HttpServletResponse response) {

        String key = request.getParameter("num");

        // Only one Fibonacci number may be deleted at a time.
        if (key == null)
            throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);
        try {
            int n = Integer.parseInt(key.trim());
            cache.remove(n);
            send_typed_response(request, response, n + " deleted.");
        }
        catch(NumberFormatException e) {
            throw new HTTPException(HttpServletResponse.SC_BAD_REQUEST);
        }

    }

    public void doPut(HttpServletRequest req, HttpServletResponse res) {
        throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
    }

    public void doInfo(HttpServletRequest req, HttpServletResponse res) {
        throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
```

```
    }

    public void doHead(HttpServletRequest req, HttpServletResponse res) {
        throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
    }

    public void doOptions(HttpServletRequest req, HttpServletResponse res) {
        throw new HTTPException(HttpServletResponse.SC_METHOD_NOT_ALLOWED);
    }

    private void send_typed_response(HttpServletRequest request,
                                     HttpServletResponse response,
                                     Object data) {

        String desired_type = request.getHeader("accept");

        // If client requests plain text or HTML, send it; else XML.
        if (desired_type.contains("text/plain"))
            send_plain(response, data);
        else if (desired_type.contains("text/html"))
            send_html(response, data);
        else
            send_xml(response, data);

    }

    // For simplicity, the data are stringified and then XML encoded.
    private void send_xml(HttpServletResponse response, Object data) {
        try {
            XMLEncoder enc = new XMLEncoder(response.getOutputStream());
            enc.writeObject(data.toString());
            enc.close();
        }
        catch(IOException e) {
            throw new HTTPException(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        }
    }

    private void send_html(HttpServletResponse response, Object data) {
        String html_start =
            "<html><head><title>send_html response</title></head><body><div>";
        String html_end = "</div></body></html>";
        String html_doc = html_start + data.toString() + html_end;
        send_plain(response, html_doc);
    }

    private void send_plain(HttpServletResponse response, Object data) {
        try {
            OutputStream out = response.getOutputStream();
            out.write(data.toString().getBytes());
            out.flush();
        }
        catch(IOException e) {
            throw new HTTPException(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        }
    }

    private int countRabbits(int n) {
        if (n < 0) throw new HTTPException(403);

        // Easy cases.
        if (n < 2)
            return n;
```

```
        // Return cached value if present.
        if (cache.containsKey(n))
            return cache.get(n);
        if (cache.containsKey(n - 1) && cache.containsKey(n - 2)) {
            cache.put(n, cache.get(n - 1) + cache.get(n - 2));
            return cache.get(n);
        }
        // Otherwise, compute from scratch, cache, and return.
        int fib = 1, prev = 0;
        for (int i = 2; i <= n; i++) {
            int temp = fib;
            fib += prev;
            prev = temp;
        }
        cache.put(n, fib);
        return fib;
    }
}
```

Übersetzen und Deployment eines Servlets

Die SE 6 reicht nicht aus, um die Klasse `RabbitCounterServlet` zu übersetzen. Apache Tomcat, typischerweise abgekürzt zu „Tomcat“, ist die Referenzimplementierung für Servletcontainer und steht unter der Adresse *http://tomcat.apache.org* zum kostenlosen Herunterladen zur Verfügung. Die aktuelle Version ist 6.x. Der Vereinfachung halber bezeichne `$CATALINA_HOME` das Installationsverzeichnis. Direkt unterhalb von `$CATALINA_HOME` sind drei Unterverzeichnisse von Interesse:

- *$CATALINA_HOME/bin*: Dieses Unterverzeichnis enthält die Unix-Shellskripte *startup.sh* und *shutdown.sh* (für Windows mit der Endung *.bat*). Tomcat wird gestattet, indem Sie das *startup.sh* Skript aufrufen. Öffnen Sie ein Browserfenster mit der URL *http://localhost:8080*, um zu testen, ob Tomcat gestartet wurde.

- *$CATALINA_HOME/logs*: Tomcat pflegt verschiedene Protokolldateien, die nützlich sind, um zu kontrollieren, ob das Deployment eines Servlets erfolgreich ausgeführt wurde.

- *$CATALINA_HOME/webapps*: Servlets werden in Form von WAR-Dateien (Web Archive) deployt. Diese WAR-Dateien sind JAR-Dateien mit der Endung *.war*. Kopieren Sie die WAR-Datei eines Servlets in dieses Verzeichnis, um das Servlet zu deployen.

Der Quelltext der Klasse `RabbitCounterServlet` liegt im Package `ch04.rc`. Es folgen die erforderlichen Schritte, um das Servlet zu übersetzen, zu verpacken und zu deployen. Das Kommando zum Übersetzen wird im Arbeitsverzeichnis aufgerufen (das Verzeichnis, welches das Unterverzeichnis *ch04* enthält):

- Das Übersetzen des Servlets setzt mehrere Packages voraus, die von Tomcat zur Verfügung gestellt werden. Rufen Sie im Arbeitsverzeichnis das folgende Kommando auf:

  ```
  % javac -cp .:$TOMCAT_HOME/lib/servlet-api.jar ch04/rc/*.java
  ```

  Die Umgebungsvariable lautet bei unixartigen Systemen `$CATALINA_HOME` und bei Windows `%TOMCAT_HOME%`. Bei unixartigen Systemen werden die Komponenten des Klassenpfades durch Doppelpunkte (`:`) getrennt, bei Windows per Semikolon (`;`).

- Tomcat erwartet *.class* Dateien, wie *RabbitCounterServlet.class*, innerhalb einer deployten WAR-Datei in der Verzeichnisstruktur unterhalb von *WEB-INF/classes*. Legen Sie im Arbeits-

verzeichnis das Unterverzeichnis *WEB-INF/classes/ch04/rc* an und kopieren Sie das *übersetzte* Servlet in dieses Unterverzeichnis.

- Nahezu jedes Servlet im produktiven Betrieb hat eine Konfigurationsdatei namens *web.xml* (Deploymentdeskriptor, kurz DD), obwohl diese Datei technisch nicht mehr benötigt wird. Der Deploymentdeskriptor zu diesem Beispiel ist:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
      version="2.4">
  <display-name>RabbitCounter Servlet</display-name>
    <servlet>
      <servlet-name>rcounter</servlet-name>
      <servlet-class>
        ch04.rc.RabbitCounterServlet
      </servlet-class>
      <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
      <servlet-name>rcounter</servlet-name>
      <url-pattern>/fib</url-pattern>
    </servlet-mapping>
</web-app>
```

  Beachten Sie das `<url-pattern>`-Element, welches das URL-Muster `/fib` deklariert. Angenommen, das Servlet befindet sich in der Datei *rc.war*. Dann lautet die URL für das Servlet *http://localhost:8080/rc/fib*. Die `<load-on-start-up>`-Einstellung besagt, daß Tomcat die Servletklasse nicht schon beim Deployment laden, sondern auf die erste Anfrage warten soll.

- Das Servlet ist nun fertig, um verpackt und deployt zu werden. Das Kommando

```
% jar cvf rc.war WEB-INF
```

  im Arbeitsverzeichnis erzeugt eine WAR-Datei, die den Deploymentdeskriptor und das übersetzte Servlet enthält. Deployen Sie die WAR-Datei anschließend, in dem Sie sie in das Verzeichnis *$CATALINA_HOME/webapps* kopieren. Sie können in den Protokolldateien nachsehen, ob das Deployment erfolgreich verlaufen ist oder in einem Browserfenster versuchen, die URL *http://localhost:8080/rc/fib* zu erreichen.

---

[49] Die Klasse `RabbitCounterServlet` überschreibt die `init()`-Methode, die der Servletcontainer aufruft, wenn das Servlet erstmals geladen wird. Die Methode erzeugt den Container, der die infolge von `POST`-Anfragen errechneten Fibonaccizahlen zwischenspeichert. Außerdem werden die HTTP-Methoden `GET` und `DELETE` unterstützt. Eine `GET`-Anfrage ohne Query-String liefert alle verfügbaren Zahlen, während eine `GET`-Anfrage mit Query-String als Anfrage nach einer spezifischen Fibonaccizahl interpretiert wird. Der Dienst gestattet stets nur das Löschen einer einzelnen Fibonaccizahl. Eine `DELETE`-Anfrage muß also einen Query-String haben, der die zu löschende Zahl angibt. Die verbleibende CRUD-Operation `update` ist nicht implementiert. Die `doPut()`-Methode wirft, wie die übrigen rudimentär implementierten `doXXX()`-Methoden, eine Ausnahme vom Typ `HTTPException` mit dem Statuswert `405` aus:

```
HttpServletResponse.SC_METHOD_NOT_ALLOWED
```

Es gibt ähnliche Konstanten für die übrigen HTTP-Statuswerte.

4.5.2 Anfragen nach Antworten eines bestimmten MIME-Typs

[50] Die Klasse `RabbitCounterServlet` unterscheidet sich dadurch vom `RabbitCounterProvider`-Dienst in Unterabschnitt 4.4.1, daß sie als Servlet und nicht als Klasse mit der Annotation `@WebServiceProvider` implementiert ist. `RabbitCounterServlet` unterscheidet sich in noch einer weiteren Hinsicht vom ersten Beispiel: Es akzeptiert Anfragen die eine Antwort eines bestimmten MIME-Typs verlangen. Ein Client für das Servlet:

```
import java.util.List;
import java.util.ArrayList;
import java.net.URL;
import java.net.HttpURLConnection;
import java.net.URLEncoder;
import java.net.MalformedURLException;
import java.net.URLEncoder;
import java.io.IOException;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;

class ClientRC {

    private static final String url = "http://localhost:8080/rc/fib";

    public static void main(String[] args) {
        new ClientRC().send_requests();
    }

    private void send_requests() {

        try {

            HttpURLConnection conn = null;

            // POST request to create some Fibonacci numbers.
            List<Integer> nums = new ArrayList<Integer>();
            for (int i = 1; i < 15; i++) nums.add(i);
            String payload = URLEncoder.encode("nums", "UTF-8") + "=" +
                URLEncoder.encode(nums.toString(), "UTF-8");

            // Send the request
            conn = get_connection(url, "POST");
            conn.setRequestProperty("accept", "text/xml");
            DataOutputStream out = new DataOutputStream(conn.getOutputStream());
            out.writeBytes(payload);
            out.flush();
            get_response(conn);

            // GET to test whether POST worked
            conn = get_connection(url, "GET");
            conn.addRequestProperty("accept", "text/xml");
            conn.connect();
            get_response(conn);

            conn = get_connection(url + "?num=12", "GET");
            conn.addRequestProperty("accept", "text/plain");
            conn.connect();
            get_response(conn);
```

```
            // DELETE request
            conn = get_connection(url + "?num=12", "DELETE");
            conn.addRequestProperty("accept", "text/xml");
            conn.connect();
            get_response(conn);

            // GET request to test whether DELETE worked
            conn = get_connection(url + "?num=12", "GET");
            conn.addRequestProperty("accept", "text/html");
            conn.connect();
            get_response(conn);
        }

        catch(IOException e) { System.err.println(e); }
        catch(NullPointerException e) { System.err.println(e); }

    }

    private HttpURLConnection get_connection(String url_string, String verb) {

        HttpURLConnection conn = null;

        try {

            URL url = new URL(url_string);
            conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod(verb);
            conn.setDoInput(true);
            conn.setDoOutput(true);

        }

        catch(MalformedURLException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }
        return conn;

    }

    private void get_response(HttpURLConnection conn) {

        try {

            String xml = "";
            BufferedReader reader =
                new BufferedReader(new InputStreamReader(conn.getInputStream()));
            String next = null;
            while ((next = reader.readLine()) != null)
                xml += next;
            System.out.println("The response:\n" + xml);

        }

        catch(IOException e) { System.err.println(e); }

    }

}
```

Der Client sendet `POST`-, `GET`- und `DELETE`-Anfragen, die jeweils den gewünschten MIME-Typ der Antwort nennen. Beispielsweise bewirkt die Anweisung

```
conn.setRequestProperty("accept", "text/xml");
```

in der `POST`-Anfrage, daß die Antwort ein XML-Dokument enthält. Der Wert des `Accept`-Headers ist nicht auf einen einzelnen MIME-Typ beschränkt. Die Anweisung könnte auch lauten:

```
conn.setRequestProperty("accept", "text/xml, text/xml, application/soap");
```

Die aufgezählten Typen können sogar mit Prioritäten und Gewichten bezeichnet werden, die der Dienst in Betracht zieht. Dieses Beispiel hält sich aber an einzelne MIME-Typen wie `text/html`.

[51] Das `RabbitCounterServlet` kann Antworten mit den MIME-Typen `text/html`, `text/html` und `text/plain` senden. Hier die Ausgabe des Clients, zur besseren Lesbarkeit ein wenig umformatiert und mit Kommentaren versehen:

```
The response: // from the initial POST request to create some Fibonacci numbers
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_06" class="java.beans.XMLDecoder">
<string>
   [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377] added.
</string>
</java>

The response: // from a GET request to confirm that the POST worked
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_06" class="java.beans.XMLDecoder">
<string>[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]</string>
</java>

The response: // from a GET request with text/plain as the desired type
144

The response: // from a DELETE request
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_06" class="java.beans.XMLDecoder">
<string>12 deleted.</string>
</java>

The response: // from a GET to confirm the DELETE with HTML as the desired type
<html><head><title>send_html response</title></head>
<body><div>-1</div></body>
</html>
```

Der Rückgabewert -1 in der letzten Antwort zeigt an, daß die Fibonaccizahl für den Wert 12 nicht verfügbar ist. Die XML- und HTML-Formate sind zwar schlicht, zeigen aber, wie Dienste im REST-Stil typisierte Antworten erzeugen können, die den in der Anfrage festgelegten Anforderungen genügen.

4.6 Java-Clients für existierende Dienste im REST-Stil

[52] Es gibt zahlreiche Webservices im REST-Stil von bekannten Anbietern wie Yahoo!, Amazon und eBay, obwohl die Debatte um die Frage weitergeht, was einen echten Dienst im REST-Stil ausmacht. Dieser Abschnitt stellt Clients zu einigen dieser kommerziellen Dienste im REST-Stil vor.

4.6.1 Der Nachrichtendienst von Yahoo!

[53] Das erste Beispiel ist ein Client für einen Yahoo!-Dienst im REST-Stil, der die aktuellen Neuigkeiten zu einem bestimmten Thema zusammenfaßt. Die Anfrage verwendet die HTTP-Methode `GET` mit einem Query-String:

```
import java.net.URI;
import java.util.Map;
import javax.xml.namespace.QName;
```

```
import javax.xml.ws.Service;
import javax.xml.ws.Dispatch;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.transform.Source;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.ws.handler.MessageContext;
import org.w3c.dom.NodeList;
import yahoo.NSResolver;

// A client against the Yahoo! RESTful news summary service.
class YahooClient {

    public static void main(String[] args) throws Exception {

        if (args.length < 1) {
            System.err.println("YahooClient <your AppID>");
            return;
        }
        String app_id = "appid=" + args[0];

        // Create a name for a service port.
        URI ns_URI = new URI("urn:yahoo:yn");
        QName serviceName = new QName("yahoo", ns_URI.toString());
        QName portName = new QName("yahoo_port", ns_URI.toString());

        // Now create a service proxy
        Service s = Service.create(serviceName);

        String qs = app_id + "&type=all&results=10&" +
                    "sort=date&language=en&query=quantum mechanics";

        // Endpoint address
        URI address = new URI("http",                   // HTTP scheme
                              null,                     // user info
                              "api.search.yahoo.com",   // host
                              80,                       // port
                              "/NewsSearchService/V1/newsSearch", // path
                              qs,                       // query string
                              null);                    // fragment

        // Add the appropriate port
        s.addPort(portName, HTTPBinding.HTTP_BINDING, address.toString());

        // From the service, generate a Dispatcher
        Dispatch<Source> d =
            s.createDispatch(portName, Source.class, Service.Mode.PAYLOAD);
        Map<String, Object> request_context = d.getRequestContext();
        request_context.put(MessageContext.HTTP_REQUEST_METHOD, "GET");

        // Invoke
        Source result = d.invoke(null);
        DOMResult dom_result = new DOMResult();
        Transformer trans = TransformerFactory.newInstance().newTransformer();
        trans.transform(result, dom_result);

        XPathFactory xpf = XPathFactory.newInstance();
        XPath xp = xpf.newXPath();
        xp.setNamespaceContext(new NSResolver("yn", ns_URI.toString()));
```

```
        NodeList resultList = (NodeList)
            xp.evaluate("/yn:ResultSet/yn:Result",
                        dom_result.getNode(),
                        XPathConstants.NODESET);

        int len = resultList.getLength();
        for (int i = 1; i <= len; i++) {
            String title =
                xp.evaluate("/yn:ResultSet/yn:Result(" + i + ")/yn:Title",
                            dom_result.getNode());
            String click =
                xp.evaluate("/yn:ResultSet/yn:Result(" + i + ")/yn:ClickUrl",
                            dom_result.getNode());
            System.out.printf("(%d) %s (%s)\n", i, title, click);
        }
    }
}
```

Dieser Client erwartet die „Application-ID“ des Benutzers als Kommandozeilenargument. Der News-Dienst ist kostenlos, verlangt aber dieses Identifizierungsmerkmal. (Registrieren Sie sich unter der Adresse *http://www.yahoo.com*.) In diesem Beispiel fragt der Client maximal zehn Ergebnisse zum Thema „Quantengravitation“ ab. Hier ein Teil der „rohen“ XML-Struktur, die der Dienst zurückgibt:

```
<?xml version="1.0" encoding="UTF-8"?>
<ResultSet
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="urn:yahoo:yn"
    xsi:schemaLocation="urn:yahoo:yn
    http://api.search.yahoo.com/NewsSearchService/V1/NewsSearchResponse.xsd"
    totalResultsAvailable="9" totalResultsReturned="9"
    firstResultPosition="1">
  <Result>
    <Title>Cosmic Log: Private space age turns 4</Title>
      <Summary>
        Science editor Alan Boyle's Weblog: Four years after the first
        private-sector spaceship crossed the 62-mile mark, some space-age
        dreams have been slow to rise while others have paid off.
      </Summary>
      <Url>
        http://cosmiclog.msnbc.msn.com/archive/2008/06/20/1158681.aspx
      </Url>
      <ClickUrl>
        http://cosmiclog.msnbc.msn.com/archive/2008/06/20/1158681.aspx
      </ClickUrl>
      <NewsSource>MSNBC</NewsSource>
      <NewsSourceUrl>http://www.msnbc.msn.com/</NewsSourceUrl>
      <Language>en</Language>
      <PublishDate>1213998337</PublishDate>
      <ModificationDate>1213998338</ModificationDate>
  </Result>
  ...
</ResultSet>
```

Nun die geparste Ausgabe des Clients `YahooClient`:

```
(1) Cosmic Log: Private space age turns 4 (http://cosmiclog.msnbc.msn.com/...
(2) Neutrino experiment shortcuts from novel to real world...
(3) There Will Be No Armageddon (http://www.spacedaily.com/reports/...
(4) TSX Venture Exchange Closing Summary for June 19, 2008 (http://biz.yahoo.com...
```

```
(5) Silver Shorts Reported (http://news.goldseek.com/GoldSeeker/1213848000.php)
(6) There will be no Armageddon (http://en.rian.ru/analysis/20080618/...
(7) New Lunar Prototype Vehicles Tested (Gallery)...
(8) World's Largest Quantum Bell Test Spans Three Swiss Towns...
(9) Creating science (http://www.michigandaily.com/news/2008/06/16/...
```

Der Client verwendet einen dynamischen Stellvertreter, um die Anfrage zu veranlassen sowie einen XPath-Ausdruck, um den DOM-Baum nach bestimmten Elementen zu durchsuchen. Die obige Ausgabe beinhaltet auch den Inhalt der Elemente `<Summary>` und `<ClickURL>` aus der „rohen“ XML-Struktur. Da Quantengravitation keine aktuelles Nachrichtenthema ist, liefert die Anfrage nach zehn Ergebnissen nur neun Einträge.

[54] Das Yahoo!-Beispiel veranschaulicht, daß Clients von Diensten im REST-Stil die Last der Verarbeitung des Antwortdokumentes, typischerweise XML, in einer für die Applikation dienlichen Weise tragen. Es gibt zwar im Allgemeinen ein XML-Schema, das den Aufbau des „rohen“ XML-Struktur beschreibt, aber keinen Dienstkontrakt wie das WSDL-Dokument bei SOAP-basierten Webservices.

4.6.2 Der E-Commerce-Dienst von Amazon (REST-Stil)

[55] Yahoo! stellt nur Webservices im REST-Stil zur Verfügung, aber Amazon bietet seine Dienste in zwei Varianten an, einmal SOAP-basiert und einmal im REST-Stil. Der folgende Client `AmazonClientREST` sendet eine `read`-Anfrage zu Büchern über die Fibonaccizahlen an den E-Commerce-Dienst von Amazon. Der Client verwendet ein Objekt vom Typ *`Dispatch`* sowie eine `GET`-Anfrage mit Query-String, der die Details der Anfrage enthält:

```
import java.util.Map;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.Dispatch;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.ws.handler.MessageContext;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import ch04.dispatch.NSResolver;

class AmazonClientREST {

    private final static String uri =
        "http://webservices.amazon.com/AWSECommerceService/2005-03-23";

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.err.println("Usage: AmazonClientREST <access key>");
            return;
        }
```

```
        new AmazonClientREST().item_search(args[0].trim());
    }

    private void item_search(String access_key) {

        QName service_name = new QName("awsREST", uri);
        QName port = new QName("awsPort", uri);

        String base_url = "http://ecs.amazonaws.com/onca/xml";
        String qs = "?Service=AWSECommerceService&" +
            "Version=2005-03-23&" +
            "Operation=ItemSearch&" +
            "ContentType=text%2Fxml&" +
            "AWSAccessKeyId=" +  access_key + "&" +
            "SearchIndex=Books&" +
            "Keywords=Fibonacci";
        String endpoint = base_url + qs;

        // Now create a service proxy dispatcher.
        Service service = Service.create(service_name);
        service.addPort(port, HTTPBinding.HTTP_BINDING, endpoint);
        Dispatch<Source> dispatch =
            service.createDispatch(port, Source.class, Service.Mode.PAYLOAD);

        // Set HTTP verb.
        Map<String, Object> request_context = dispatch.getRequestContext();
        request_context.put(MessageContext.HTTP_REQUEST_METHOD, "GET");

        Source result = dispatch.invoke(null); // null payload: GET request
        display_result(result);

    }

    private void display_result(Source result) {

        DOMResult dom_result = new DOMResult();

        try {
            Transformer trans =
                TransformerFactory.newInstance().newTransformer();
            trans.transform(result, dom_result);
            XPathFactory xpf = XPathFactory.newInstance();
            XPath xp = xpf.newXPath();
            xp.setNamespaceContext(new NSResolver("aws", uri));

            NodeList authors = (NodeList)
                xp.evaluate("//aws:ItemAttributes/aws:Author",
                            dom_result.getNode(),
                            XPathConstants.NODESET);

            NodeList titles = (NodeList)
                xp.evaluate("//aws:ItemAttributes/aws:Title",
                            dom_result.getNode(),
                            XPathConstants.NODESET);

            int len = authors.getLength();
            for (int i = 0; i < len; i++) {
                Node author = authors.item(i);
                Node title  = titles.item(i);
                if (author != null && title != null) {
                    String a_name = author.getFirstChild().getNodeValue();
                    String t_name = title.getFirstChild().getNodeValue();
                    System.out.printf("%s: %s\n", a_name, t_name);
                }
```

```
                }
            }
            catch(TransformerConfigurationException e) { System.err.println(e); }
            catch(TransformerException e) { System.err.println(e); }
            catch(XPathExpressionException e) { System.err.println(e); }
        }
    }
```

Das mit der Antwort versendete XML-Dokument ist nun eine „rohe“ XML-Struktur anstelle eines SOAP-Envelopes. Dennoch gehorcht diese XML-Struktur demselben XML-Schema, das auch dem WSDL-Dokument für die SOAP-basierte Version des Dienstes unterliegt. Der einzige Unterschied besteht darin, daß der SOAP-basierte Dienst die XML-Struktur in einem SOAP-Envelope verpackt. In beiden Fällen wird die XML-Struktur als Nutzlast der HTTP-Antwort transportiert. Hier ein Teil XML-Struktur:

```
<?xml version="1.0" encoding="UTF-8"?>
<ItemSearchResponse
  xmlns="http://webservices.amazon.com/AWSECommerceService/2005-03-23">
  ...
  <ItemSearchRequest>
    <Keywords>Fibonacci</Keywords>
    <SearchIndex>Books</SearchIndex>
  </ItemSearchRequest>
  ...
  <TotalResults>177</TotalResults>
  <TotalPages>18</TotalPages>
  ...
  <Items>
    <Item>
      <ItemAttributes>
        <Author>Carolyn Boroden</Author>
        <Manufacturer>McGraw-Hill</Manufacturer>
        <ProductGroup>Book</ProductGroup>
        <Title>Fibonacci Trading: How to Master Time and Price Advantage</Title>
      </ItemAttributes>
    </Item>
    ...
  </Items>
</ItemSearchResponse>
```

Die Klasse `AmazonClientREST` parst die „rohe“ XML-Struktur zur Abwechslung ein wenig anders, als bei den früheren Beispielen. Insbesondere verwendet der Client XPath, um separate Liste von Autoren und Buchtiteln zu bekommen:

```
NodeList authors = (NodeList)
    xp.evaluate("//aws:ItemAttributes/aws:Author",
                dom_result.getNode(), XPathConstants.NODESET);
NodeList titles = (NodeList)
    xp.evaluate("//aws:ItemAttributes/aws:Title",
                dom_result.getNode(), XPathConstants.NODESET);
```

und extrahiert Autor und Titel anschließend in einer Schleife aus dem DOM-Baum:

```
int len = authors.getLength();
for (int i = 0; i < len; i++) {
    Node author = authors.item(i);
    Node title  = titles.item(i);
    if (author != null && title != null) {
```

```
            String a_name = author.getFirstChild().getNodeValue();
            String t_name = title.getFirstChild().getNodeValue();
            System.out.printf("%s: %s\n", a_name, t_name);
        }
    }
```

Die Ausgabe lautet bei mir:

```
Carolyn Boroden: Fibonacci Trading: How to Master Time and Price Advantage
Kimberly Elam: Geometry of Design: Studies in Proportion and Composition
Alfred S. Posamentier: The Fabulous Fibonacci Numbers
Ingmar Lehmann: Math for Mystics: From the Fibonacci sequence to Luna's Labyrinth...
Renna Shesso: Breakthrough Strategies for Predicting any Market...
Jeff Greenblatt: Wild Fibonacci: Nature's Secret Code Revealed
Joy N. Hulme: Fibonacci Analysis (Bloomberg Market Essentials: Technical Analysis)
Constance Brown: Fibonacci Fun: Fascinating Activities With Intriguing Numbers
Trudi Hammel Garland: Fibonacci Applications and Strategies for Traders
Robert Fischer: New Frontiers in Fibonacci Trading: Charting Techniques,...
```

Der Amazon Simple Storage Service, bekannt unter dem Kürzel S3, ist ein kostenpflichtiger Dienst und ebenfalls sowohl über einen SOAP-basierten Client als auch einen Client im REST-Stil erreichbar. Der Dienst gestattet seinen Nutzern individuelle Datenobjekte bis zu 5GB pro Stück zu speichern oder abzuholen. S3 wird häufig als gutes Beispiel für einen nützlichen Webservice mit sehr einfacher Schnittstelle zitiert.

4.6.3 Der Tumblr-Dienst im REST-Stil

[56] Der Tumblr-Dienst ist wohl am meisten durch den Begriff „tumblelog“ beziehungsweise „tlog“ bekannt, einer Variante des traditionellen Blogs, die kurze Texte zusammen mit Multimediaelementen wie Photos, Musik oder Filmen hervorhebt. Tumblelogs sind oft künstlerisch ambitioniert. Der Dienst ist kostenlos, der uneingeschränkte Zugriff auf die REST-Operationen setzt allerdings ein Benutzerkonto voraus, das Sie unter der Adresse *http://www.tumblr.com* einrichten können.

[57] Die folgende Klasse `TumblrClient` stützt sich auf die abstrakte Klasse `HttpURLConnection`, um `GET`- und `POST`-Anfragen an den Tumblr-Dienst im REST-Stil zu senden. `HttpURLConnection` ist hier eine bessere Wahl als eine Implementierung des Interfaces *`Dispatch`* (dynamischer Stellvertreter), da eine `POST`-Anfrage an den Tumblr-Dienst kein XML-Dokument, sondern gewöhnliche Schlüssel/Wert-Paare enthält. Nun der Quelltext der Klasse `TumblrClient`:

```
import java.net.URL;
import java.net.HttpURLConnection;
import java.net.URLEncoder;
import java.net.MalformedURLException;
import java.net.URLEncoder;
import java.io.IOException;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.dom.DOMResult;
import javax.xml.xpath.XPathFactory;
import javax.xml.xpath.XPath;
```

```
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import java.io.ByteArrayInputStream;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;

class TumblrClient {

    public static void main(String[] args) {

        if (args.length < 2) {
            System.err.println("Usage: TumblrClient <email> <passwd>");
            return;

        }

        new TumblrClient().tumble(args[0], args[1]);

    }

    private void tumble(String email, String password) {

        try {

            HttpURLConnection conn = null;

            // GET request.
            String url = "http://mgk-cdm.tumblr.com/api/read";
            conn = get_connection(url, "GET");
            conn.setRequestProperty("accept", "text/xml");
            conn.connect();
            String xml = get_response(conn);
            if (xml.length() > 0) {
                System.out.println("Raw XML:\n" + xml);
                parse(xml, "\nSki photo captions:", "//photo-caption");
            }

            // POST request
            url = "http://www.tumblr.com/api/write";
            conn = get_connection(url, "POST");
            String title = "Summer thoughts up north";
            String body = "Craigieburn Ski Area, NZ";
            String payload =
                URLEncoder.encode("email", "UTF-8") + "=" +
                URLEncoder.encode(email, "UTF-8") + "&" +
                URLEncoder.encode("password", "UTF-8") + "=" +
                URLEncoder.encode(password, "UTF-8") + "&" +
                URLEncoder.encode("type", "UTF-8") + "=" +
                URLEncoder.encode("regular", "UTF-8") + "&" +
                URLEncoder.encode("title", "UTF-8") + "=" +
                URLEncoder.encode(title, "UTF-8") + "&" +
                URLEncoder.encode("body", "UTF-8") + "=" +
                URLEncoder.encode(body, "UTF-8");
            DataOutputStream out = new DataOutputStream(conn.getOutputStream());
            out.writeBytes(payload);
            out.flush();
            String response = get_response(conn);
            System.out.println("Confirmation code: " + response);

        }

        catch(IOException e) { System.err.println(e); }
        catch(NullPointerException e) { System.err.println(e); }
```

```
}
private HttpURLConnection get_connection(String url_s, String verb) {
    HttpURLConnection conn = null;
    try {
        URL url = new URL(url_s);
        conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod(verb);
        conn.setDoInput(true);
        conn.setDoOutput(true);
    }
    catch(MalformedURLException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
    return conn;
}
private String get_response(HttpURLConnection conn) {
    String xml = "";
    try {
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(conn.getInputStream()));
        String next = null;
        while ((next = reader.readLine()) != null) xml += next;
    }
    catch(IOException e) { System.err.println(e); }
    return xml;
}
private void parse(String xml, String msg, String pattern) {
    StreamSource source =
        new StreamSource(new ByteArrayInputStream(xml.getBytes()));
    DOMResult dom_result = new DOMResult();
    System.out.println(msg);
    try {
        Transformer trans =
            TransformerFactory.newInstance().newTransformer();
        trans.transform(source, dom_result);
        XPathFactory xpf = XPathFactory.newInstance();
        XPath xp = xpf.newXPath();
        NodeList list = (NodeList)
            xp.evaluate(pattern, dom_result.getNode(),
                        XPathConstants.NODESET);
        int len = list.getLength();
        for (int i = 0; i < len; i++) {
            Node node = list.item(i);
            if (node != null)
                System.out.println(node.getFirstChild().getNodeValue());
        }
    }
    catch(TransformerConfigurationException e) { System.err.println(e); }
```

```
            catch(TransformerException e) { System.err.println(e); }
            catch(XPathExpressionException e) { System.err.println(e); }
        }
    }
```

Die URL der `GET`-Anfrage ist:

```
http://mgk-cdm.tumblr.com/api/read
```

Dies ist die URL meines Tumblr-Benutzerkontos mit der Erweiterung `api/read`. Die Anfrage gibt alle meine öffentlichen (das heißt ungesicherten) Beiträge zurück. Hier ein Teil der „rohen“ XML-Struktur aus der Antwort:

```
<?xml version="1.0" encoding="UTF-8"?>
<tumblr version="1.0">
  <tumblelog name="mgk-cdm" timezone="US/Eastern" title="Untitled" />
  <posts start="0" total="5">
    <post id="40130991" url="http://mgk-cdm.tumblr.com/post/40130991"
          type="photo" date-gmt="2008-06-28 03:09:29 GMT"
          date="Fri, 27 Jun 2008 23:09:29" unix-timestamp="1214622569">
      <photo-caption>
        Trying the new skis, working better than I am.
      </photo-caption>
      <photo-url max-width="500">
        http://media.tumblr.com/awK1GiaTRar6p46p6Xy13mBH_500.jpg
      </photo-url>
    </post>
    ...
    <post id="40006745" url="http://mgk-cdm.tumblr.com/post/40006745"
          type="regular" date-gmt="2008-06-27 04:12:53 GMT"
          date="Fri, 27 Jun 2008 00:12:53" unix-timestamp="1214539973">
      <regular-title>Weathering the weather</regular-title>
      <regular-body>
        miserable, need to get fully wet or not at all
      </regular-body>
    </post>
    ...
    <post id="40006638" url="http://mgk-cdm.tumblr.com/post/40006638"
          type="regular" date-gmt="2008-06-27 04:11:34 GMT"
          date="Fri, 27 Jun 2008 00:11:34" unix-timestamp="1214539894">
      <regular-title>tumblr. API</regular-title>
      <regular-body>Very restful</regular-body>
    </post>
  </posts>
</tumblr>
```

Die „rohe“ XML-Struktur hat einen sehr einfachen Aufbau und verzichtet sogar auf Namensräume. Die Klasse `TumblrClient` verwendet einen XPath-Ausdruck, um eine Liste der Beschreibungen der Photos zu extrahieren:

```
Ski photo captions:
Trying the new skis, working better than I am.
Very tough day on the trails; deep snow, too deep for skating.
Long haul up, fun going down.
```

Der Client sendet eine `POST`-Anfrage, die meinen Tumblr-Beiträgen einen neuen Eintrag hinzufügt. Die geänderte URL lautet nun *http://www.tumblr.com* (Tumblr-Hauptseite) mit der Erweiterung

/api/write. Die Nutzdaten der `POST`-Anfrage müssen meine E-Mailadresse und mein Paßwort enthalten:

```
String payload =
   URLEncoder.encode("email", "UTF-8") + "=" +
   URLEncoder.encode(email, "UTF-8") + "&" +
   URLEncoder.encode("password", "UTF-8") + "=" +
   URLEncoder.encode(password, "UTF-8") + "&" +
   URLEncoder.encode("type", "UTF-8") + "=" +
   URLEncoder.encode("regular", "UTF-8") + "&" +
   URLEncoder.encode("title", "UTF-8") + "=" +
   URLEncoder.encode(title, "UTF-8") + "&" +
   URLEncoder.encode("body", "UTF-8") + "=" +
   URLEncoder.encode(body, "UTF-8");
DataOutputStream out = new DataOutputStream(conn.getOutputStream());
out.writeBytes(payload);
out.flush();
```

Die Dokumentation der Tumblr-API besteht aus nur einer einzigen Seite. Die API unterstützt die CRUD-Operationen `read` und `create` durch die Endungen */api/read* beziehungsweise */api/write*. Eine `read`-Operation wird, wie üblich, durch eine `GET`- und eine `create`-Operation durch eine `POST`-Anfrage bewerkstelligt. Der Tumblr-Dienst unterstützt noch einige weitere Endungen. Beispielsweise bewirkt die *Endung /api/read/json*, daß die Antwort in JSON- (JavaScript Object Notation), statt im XML-Format gesendet wird. Eine `POST`-Anfrage an den Tumblr-Dienst kann zum Hochladen von Bildern, Audio- und Videodateien zusammen mit Texten verwendet werden. Multimediadateien können entweder als unkodierte Bytes oder als URL-kodierte Nutzdaten im Körper der `POST`-Anfrage übergeben werden.

[58] Die Einfachheit der Tumblr-API begünstigt den Aufbau graphischer Schnittstellen und Plugins, die wiederum Tumblr gestatten, einfach mit anderen Websites wie Facebook zu interagieren. Die Tumblr-API ist eine gutes Beispiel dafür, wie viel man mit so wenig erreichen kann.

4.7 Die Web Application Description Language (WADL)

[59] Das WSDL-Dokument SOAP-basierter Webservices ist ein Segen für die Programmierer, da dieser Dienstkontrakt verwendet werden kann, um clientseitige Artefakte und sogar ein serviceseitiges Interface zu generieren. Dienste im REST-Stil haben kein offizielles oder zumindest allgemein akzeptiertes Äquivalent zum WSDL-Dokument, obwohl Anstrengungen in diese Richtung unternommen werden. Einer dieser Versuche ist die WADL-Initiative (*https://wadl.dev.java.net*). Die Abkürzung WADL steht für Web Application Description Language.

[60] Die WADL-Distribution beinhaltet das `wadl2java`-Kommando, eine Bibliothek benötigter JAR-Dateien und ein Beispieldokument namens *YahooSearch.wadl*. Die Distribution enthält außerdem Ant-, Maven- und Kommandozeilenskripte. Wir beginnen mit einem Yahoo!-Client, der die `wadl2java`-Artefakte verwendet:

```
import com.yahoo.search.ResultSet;
import com.yahoo.search.ObjectFactory;
import com.yahoo.search.Endpoint;
import com.yahoo.search.Endpoint.NewsSearch;
import com.yahoo.search.Type;
import com.yahoo.search.Result;
import com.yahoo.search.Sort;
import com.yahoo.search.ImageType;
```

```
import com.yahoo.search.Output;
import com.yahoo.search.Error;
import com.yahoo.search.SearchErrorException;
import javax.xml.bind.JAXBException;
import java.io.IOException;
import java.util.List;

class YahooWADL {

    public static void main(String[] args) {

        if (args.length < 1) {
            System.err.println("Usage: YahooWADL <app id>");
            return;
        }
        String app_id = args[0];

        try {
            NewsSearch service = new NewsSearch();
            String query = "neutrino";

            ResultSet result_set = service.getAsResultSet(app_id, query);
            List<Result> list = result_set.getResultList();
            int i = 1;
            for (Result next : list) {
                String title = next.getTitle();
                String click = next.getClickUrl();
                System.out.printf("(%d) %s %s\n", i++, title, click);
            }
        }

        catch(JAXBException e) { System.err.println(e); }
        catch(SearchErrorException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }

    }
}
```

Der Quelltext ist sauberer als meine ursprüngliche Klasse `YahooClient`. Die `wadl2java`-Artefakte verbergen die XML-Verarbeitung und einige andere Details, etwa die Formatierung eines passenden Query-Strings für eine `GET`-Anfrage an den News-Dienst von Yahoo!. Der Client `YahooClient` liefert bei einer Anfrage nach Artikeln, die das Suchwort „neutrino“ enthalten, bei mir die folgende Ausgabe:

```
(1) Congress to the rescue for Fermi jobs http://www.dailyherald.com/story/...
(2) AIP FYI #69: Senate FY 2009 National Science Foundation Funding Bill...
(3) Linked by Thom Holwerda on Wed 12th Sep 2007 11:51 UTC...
(4) The World's Nine Largest Science Projects http://science.slashdot.org/...
(5) Funding bill may block Fermi layoffs http://www.suntimes.com/business/...
(6) In print http://www.sciencenews.org/view/generic/id/33654/title/For_Kids...
(7) Recent Original Stories http://www.osnews.com/thread?284017
(8) Antares : un télescope pointé vers le sol qui utilise la terre comme filtre...
(9) Software addresses quality of hands-free car phone audio...
(10) Planetary science: Tunguska at 100 http://www.nature.com/news/2008/...
```

[61] Nun das WADL-Dokument, das zum Erzeugen der clientseitigen Artefakte verwendet wurde:

```
<?xml version="1.0"?>
<!--
The contents of this file are subject to the terms of the Common Development
and Distribution License (the "License"). You may not use this file except
in compliance with the License. You can obtain a copy of the license at
     http://www.opensource.org/licenses/cddl1.php
```

```
See the License for the specific language governing permissions and limitations
under the License.
-->
<application xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:yn="urn:yahoo:yn"
             xmlns:ya="urn:yahoo:api"
             xmlns:html="http://www.w3.org/1999/xhtml"
             xmlns="http://research.sun.com/wadl/2006/10">
  <grammars>
    <include href="NewsSearchResponse.xsd" />
    <include href="NewsSearchError.xsd" />
  </grammars>

  <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
    <resource path="newsSearch">
      <doc xml:lang="en" title="Yahoo News Search Service">
        The <html:i>Yahoo News Search</html:i> service provides online
        searching of news stories from around the world.
      </doc>
      <param name="appid" type="xsd:string" required="true" style="query">
        <doc>The application ID. See
          <html:a href="http://developer.yahoo.com/faq/index.html#appid">
            Application IDs
          </html:a> for more information.
        </doc>
      </param>
      <method href="#search" />
    </resource>
  </resources>

  <method name="GET" id="search">
    <doc xml:lang="en" title="Search news stories by keyword" />
      <request>
        <param name="query" type="xsd:string" required="true" style="query">
          <doc xml:lang="en" title="Space separated keywords to search for" />
        </param>
        <param name="type" type="xsd:string" default="all" style="query">
          <doc xml:lang="en" title="Keyword matching" />
          <option value="all">
            <doc>All query terms.</doc>
          </option>
          <option value="any">
            <doc>Any query terms.</doc>
          </option>
          <option value="phrase">
            <doc>Query terms as a phrase.</doc>
          </option>
        </param>
        <param name="results" type="xsd:int" default="10" style="query">
          <doc xml:lang="en" title="Number of results" />
        </param>
        <param name="start" type="xsd:int" default="1" style="query">
          <doc xml:lang="en" title="Index of first result" />
        </param>
        <param name="sort" type="xsd:string" default="rank" style="query">
          <doc xml:lang="en" title="Sort by date or rank" />
          <option value="rank" />
          <option value="date" />
```

```
          </param>
          <param name="language" type="xsd:string" style="query">
            <doc xml:lang="en" title="Language filter, omit for any language" />
          </param>
          <param name="output" type="xsd:string" default="xml" style="query">
            <doc>
              The format for the output. If <html:em>json</html:em> is
              requested, the results will be returned in
              <html:a href="http://developer.yahoo.com/common/json.html">
              JSON</html:a> format. If <html:em>php</html:em> is requested,
              the results will be returned in
              <html:a href="http://developer.yahoo.com/common/phpserial.html">
              Serialized PHP</html:a> format.
            </doc>
            <option value="xml" />
            <option value="json" />
            <option value="php" />
          </param>
          <param name="callback" type="xsd:string" style="query">
            <doc>
              The name of the callback function to wrap around the JSON
              data. The following characters are allowed: A-Z a-z 0-9 .
              [] and _. If output=json has not been requested, this
              parameter has no effect. More information on the callback
              can be found in the
              <html:a href="http://developer.yahoo.com/common/json.html
              #callbackparam">Yahoo! Developer Network JSON
              Documentation</html:a>.
            </doc>
          </param>
        </request>
        <response>
          <representation mediaType="application/xml" element="yn:ResultSet">
            <doc xml:lang="en" title="A list of news items matching the query" />
          </representation>
          <fault id="SearchError" status="400"
                 mediaType="application/xml" element="ya:Error" />
      </response>
    </method>
  </application>
```

Das WADL-Dokument beginnt mit Verweisen auf zwei XML-Schemata. Eines ist die Grammatik für Fehlerdokumente, das andere die Grammatik für normale Antwortdokumente des Dienstes. Daran schließt sich eine Liste verfügbarer Resourcen an, in diesem Fall nur der Suchdienst für Nachrichten. Der `<methods>`-Abschnitt beschreibt die HTTP-Methoden, das heißt die vom Dienst angebotenen CRUD-Operationen. Beim Nachrichtendienst von Yahoo! sind nur `GET`-Anfragen erlaubt. Die übrigen Abschnitte enthalten Details zu den Anfrage- und Antwortparametern. Das WADL-Dokument beinhaltet, wie ein WSDL-Dokument, auch Bezüge auf in der Sprache XML-Schema vordefinierte Typen.

[62] Ein Aufruf des `wadl2java`-Kommandos auf der Datei *YahooSearch.wadl* erzeugt elf *.java* Dateien, von denen *Endpoint.java* für das clientseitige Programmieren am wichtigsten ist. Die Klasse `Endpoint` beinhaltet die statische Klasse `NewsSearch`, die Hilfsmethoden wie `getAsResultSet()` zur Verfügung stellt. Hier nochmals der Hauptabschnitt der Klasse `YahooWADL` von Seite 171f. Die WADL-Artefakte ermöglichen kürzeren und deutlicheren Quelltext:

```
NewsSearch service = new NewsSearch();
```

```
String query = "neutrino";

ResultSet result_set = service.getAsResultSet(app_id, query);
List<Result> list = result_set.getResultList();
int i = 1;
for (Result next : list) {
    String title = next.getTitle();
    String click = next.getClickUrl();
    System.out.printf("(%d) %s %s\n", i++, title, click);
}
```

Der Suchbegriff, hier in der Referenzvariablen `query` gespeichert, ist eine Liste durch Leerzeichen voneinander getrennter Wörter. Das Objekt der statischen inneren Klasse `NewsSearch` hat Eigenschaften, um die Sortierreihenfolge, die maximale Trefferanzahl und ähnliche Parameter einzustellen. Die generierten Artefakte erleichtern die Programmierarbeit.

[63] WADL hat zwar Interesse und Diskussionen angeregt, bleibt aber zunächst eine auf Java beschränkte Initiative. Die entscheidende Frage lautet, ob sich Dienste im REST-Stil in einem solchen Ausmaß standardisieren lassen, daß es Kommandos wie `wadl2java` und `java2wadl` mit den inzwischen für SOAP-basierte Dienste verfügbaren Werkzeugen aufnehmen können. Die Kritik, SOAP-basierte Dienste seien „zu Tode entwickelt", ist gerechtfertigt. Der Vorwurf, Dienste im REST-Stil seien unterentwickelt, ist allerdings ebenso gerechtfertigt.

Probleme bei `wadl2java`-Artefakten

Das `wadl2java`-Kommando generiert unter anderem die Dateien *Error.java* und *ObjectFactory.java*. In ~~[each/beiden|allen]~~ Dateien ~~[should/sollte|muß]~~ jedes Vorkommen von `urn:yahoo:api` durch `urn:yahoo:yn` ersetzt werden. In Version 1.0 der Distribution gab es zwei Vorkommen bei ~~[each/beiden|allen]~~ *.java* Dateien. Ohne diese Änderung wird eine JAX-B-Ausnahme ausgeworfen, wenn die von der Yahoo!-Suche zurückerhaltenen Ergebnisse in Java-Objekte umgewandelt werden. Diese Änderungen könnten im XML-Schema und dem WADL-Dokument vorgenommen, welche das `wadl2java`-Kommando verwendet.

4.8 JAX-RS und die Referenzimplementierung Jersey

[64] Das Jersey-Projekt ist das Zentrum der aktuellen Java API for RESTful Web Services (JAX-RS). Jersey-Applikationen können über bekannte Container wie *Tomcat* und *GlassFish* deployt werden. Jersey stellt aber auch den leichtgewichtigen Container *Grizzly* zur Verfügung, der sich hervorragend zum Erlernen des Frameworks anbietet. Jersey arbeitet gut mit Maven zusammen. Ein deployter Jersey-Dienst generiert automatisch ein WADL-Dokument, das anschließend über eine `GET`-Abfrage angefordert werden kann. Die Adresse *https://jersey.dev.java.net* ist ein guter Einstiegspunkt in Jersey.

[65] Ein Jersey-Dienst befolgt die REST-Prinzipien, akzeptiert also die gewöhnlichen Anfragen für CRUD-Operationen, die über die HTTP-Methoden `GET`, `POST`, `DELETE` und `PUT` spezifiziert werden. Anfragen richten sich an Jersey-Resourcen. Eine Jersey-Resource ist ein POJO (Plain Old Java Object). Die folgende Klasse `MsgResource` veranschaulicht diese Eigenschaften:

```
package msg.resources;

import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.FormParam;
```

```
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.DELETE;
import java.beans.XMLEncoder;
import java.io.ByteArrayOutputStream;

// This is the base path, which can be extended at the method level.
@Path("/")
public class MsgResource {

    private static String msg = "Hello, world!";

    @GET
    @Produces("text/plain")
    public String read() {
        return msg + "\n";
    }

    @GET
    @Produces("text/plain")
    @Path("{extra}")
    public String personalized_read(@PathParam("extra") String cus) {
        return this.msg + ": " + cus + "\n";
    }

    @POST
    @Produces("text/xml")
    public String create(@FormParam("msg") String new_msg ) {
        this.msg = new_msg;
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        XMLEncoder enc = new XMLEncoder(stream);
        enc.writeObject(new_msg);
        enc.close();
        return new String(stream.toByteArray()) + "\n";
    }

    @DELETE
    @Produces("text/plain")
    public String delete() {
        this.msg = null;
        return "Message deleted.\n";
    }
}
```

Die Klasse `MsgResource` trägt intuitive Annotationen, darunter solche für die HTTP-Methoden und MIME-Typen in Antworten. Die `@Path`-Annotation unmittelbar vor der Zeile `public class MsgResource` wird verwendet, um die Resource von einer bestimmten Basis-URL zu entkoppeln. Beispiele für Basis-URLs sind *http://foo.bar.org:1234* und *http://localhost:9876*. Die Annotationen `@GET`, `@POST` und `@DELETE` ordnen jeder Operation des Dienstes eine HTTP-Methode zu. Die Annotation `@Produces` gibt den MIME-Typ der Antwort an, hier `text/plain` für die Operationen `GET` und `DELETE` beziehungsweise `text/xml` bei `POST`. Jede mit `@Produces` annotierte Methode ist dafür verantwortlich, den deklarierten Antworttyp zu liefern.

[66] Die Klasse `MsgResource` kann zusammen mit den benötigten JAR-Dateien für Jersey in einer WAR-Datei verpackt und in einem Servletcontainer wie Tomcat deployt werden. Es gibt allerdings einen schnellen Handgriff, mit dem sich Jersey-Dienst auch programmatisch in Betrieb nehmen läßt:

```
import com.sun.jersey.api.container.grizzly.GrizzlyWebContainerFactory;
import java.util.Map;
```

```
import java.util.HashMap;

class JerseyPublisher {

    public static void main(String[] args) {

        final String base_url = "http://localhost:9876/";
        final Map<String, String> config = new HashMap<String, String>();

        config.put("com.sun.jersey.config.property.packages",
                   "msg.resources"); // package with resource classes

        System.out.println("Grizzly starting on port 9876.\n" +
                           "Kill with Control-C.\n");
        try {
            GrizzlyWebContainerFactory.create(base_url, config);
        }
        catch(Exception e) { System.err.println(e); }
    }

}
```

Die Grizzly-Konfiguration erfordert Informationen über das Package, das die für die Clients erreichbaren Resourcen beinhaltet, hier `msg.resources`. In diesem Beispiel enthält das Package nur die einzelne Klasse `MsgResource`, könnte aber auch mehrere Resourcen beinhalten. Bei jeder eingehenden Anfrage geht Grizzly die verfügbaren Resourcen durch, um zu ermitteln, welche Methode die Anfrage behandeln soll, das heißt es findet Routing statt. Eine `POST`-Anfrage wird beispielsweise nur an eine mit `@POST` annotierte Methode delegiert.

[67] Das Übersetzen und Ausführen der Resource und der Publisherklasse setzt einige JAR-Dateien des Jersey-Projektes im Klassenpfad voraus. Here is the list of five under the current release:

```
asm-3.1.jar
grizzly-servlet-webserver-1.8.3.jar
jersey-core-0.9-ea.jar
jersey-server-0.9-ea.jar
jsr311-api-0.9.jar
```

Diese und weitere JAR-Dateien für die Maven-zentrierte Version von Jersey sind auf der Jersey-Homepage (*https://jersey.dev.java.net*) zum Herunterladen verfügbar.

[68] Nach dem Starten des `JerseyPublishers` ist die Resource per Webbrowser oder über ein Hilfsprogramm wie das `curl`-Kommando erreichbar. Ein Aufruf des `curl`-Kommandos:

```
% curl http://localhost:9876/
```

veranlaßt eine `GET`-Anfrage an den Dienst, der die mit `@GET` annotierten `read()`-Methode aufruft. Die Antwort besteht aus der voreingestellten Meldung:

```
Hello, world!
```

Im Gegensatz dazu veranlaßt das `curl`-Kommando

```
% curl -d msg='Goodbye, cruel world!' http://localhost:9876/echo/fred
```

eine `POST`-Anfrage an den Dienst, die den Aufruf der mit `@POST` annotierten `create()`-Methode bewirkt. (REST-Puristen werden einwenden, daß eine `PUT`-Anfrage angemessener wäre, da die `create()`-Methode offensichtlich eine vorhandene Resource aktualisiert, statt nur eine Meldung zu erzeugen.) Der `String`-Parameter der `create()`-Methode ist mit der Annotation `@FormParam` bezeichnet, wodurch die per `POST`-Anfrage übertragenen Daten als Argument an die Methode übergeben werden. Das Attribut der `@FormParam`-Annotation (hier `msg`) muß nicht namentlich mit dem

Methodenparameter (`new_msg`) übereinstimmen. Da die `@Produces`-Annotation bei der `create()`-Methode den MIME-Typ `text/xml` für die Antwort deklariert, ist die Ausgabe ein kleines XML-Dokument:

```
<?xml version="1.0" encoding="UTF-8"?>

<java version="1.6.0_06" class="java.beans.XMLDecoder">
  <string>Goodbye, cruel world!</string>
</java>
```

Die `create()`-Methode erzeugt diese Antwort mit Hilfe der Klasse `XMLEncoder`.

[69] Neben der `read()`-Methode ist noch eine weitere Methode mit `@GET` annotiert, nämlich `personalized_read()`. Die Methode trägt außerdem die Annotation `@Path("{extra}")`. Die Anfrage

```
% curl http://localhost:9876/bye
```

bewirkt einen Aufruf dieser Methode mit dem Argument `bye`. Die geschweiften Klammern um `extra` bedeuten, daß `extra` nur ein Platzhalter ist, aber kein Literal. Eine `@Path`-Annotation auf Methodenebene wird an die `@Path`-Annotation auf Klassenebene angehängt. In diesem Beispiel enthält die `@Path`-Annotation auf Klassenebene nur das Zeichen `/`.

[70] Der Grizzly-Publisher generiert automatisch ein WADL-Dokument, das über die folgende URL angefordert werden kann:

```
http://localhost:9876/application.wadl
```

Hier das automatisch generierte WADL-Dokument für `MsgResource`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<application xmlns="http://research.sun.com/wadl/2006/10">
  <doc xmlns:jersey="http://jersey.dev.java.net/"
       jersey:generatedBy="Jersey: 0.9-ea 08/22/2008 04:48 PM" />
  <resources base="http://localhost:9876/">
    <resource path="/">
      <method name="DELETE" id="delete">
        <response>
          <representation mediaType="text/plain" />
        </response>
      </method>
      <method name="GET" id="read">
        <response>
          <representation mediaType="text/plain" />
        </response>
      </method>
      <method name="POST" id="create">
        <request>
          <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 type="xs:string" name="msg" />
        </request>
        <response>
          <representation mediaType="text/xml" />
        </response>
      </method>
      <resource path="{extra}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
               type="xs:string" style="template" name="extra" />
        <method name="GET" id="personalized_read">
          <response>
```

```
            <representation mediaType="text/plain" />
          </response>
        </method>
      </resource>
    </resource>
  </resources>
</application>
```

Das WADL-Dokument drückt aus, daß die Klasse `MsgResource` zwei `GET`-Operationen, eine `POST`- und eine `DELETE`-Operation unterstützt. Das WADL-Dokument beschreibt außerdem den MIME-Typ der mit der Antwort ausgelieferten Repräsentation der Resource für jede Operation. Dieses WADL-Dokument kann als Eingabe für das `wadl2java`-Kommando verwendet werden.

[71] Jersey ist ein angemessen leichtgewichtiges Framework, das den Geist des REST-Stils von Webservices anerkennt. Der Grizzly-Publisher ist durch das automatisch generierte WADL-Dokument, welches den betriebenen Dienst beschreibt, für die Entwicklung attraktiv. Der Übergang zum produktiven Betrieb in einem Webcontainer, ob separat oder in einen Applikationsserver eingebettet, ist mühelos, da Jersey-Resourcen schlichte annotierte POJOs sind. Die gesamte JAX-RS (JSR-311) besteht aus nur drei Packages mit etwa fünfzig Interfaces und Klassen.

[72] Im Augenblick sind JAX-WS und JAX-RS noch separate Frameworks. Es wäre aber nicht überraschend, wenn beide Frameworks in Zukunft eins würden.

4.9 Das Restlet-Framework

[73] Einige Webframeworks beziehen REST ein, wohl aber keines so bestimmt wie Rails mit seinem Typ `ActiveResource`, der eine Resource im REST-Sinne implementiert. Rails betont außerdem REST-artiges Routing durch CRUD-Operationen in Form der standardisierten HTTP-Methoden. Grails ist ein in Groovy geschriebenes Imitat von Rails, wobei Groovy wiederum ein Imitat von Ruby ist und Zugriff auf die Packages der Java SE hat. Apache Sling (*http://incubator.apache.org-/sling/site/index.html*) ist ein Java-basiertes Webframework mit REST-Orientierung.

[74] Das Restlet-Framework (*http://www.restlet.org*) folgt dem REST-Architekturstil und bezieht Inspiration von anderen leichtgewichtigen aber mächtigen Frameworks wie *Netkernel* (*http://www-.1060.org*) und Rails. Wie der Name andeutet, ist das Restlet eine Alternative zum traditionellen Java-Servlet im REST-Stil. Das Restlet-Framework hat eine Client- und eine Dienst-API. Das Framework ist gut entworfen, relativ geradlinig, professionell implementiert und gut dokumentiert. Es arbeitet gut mit bereits vorhandenen Technologien zusammen. Ein Restlet kann beispielsweise in einem Servletcontainer wie Tomcat oder *Jetty* deployt werden. Die Restlet-Distribution unterstützt die Integration des Spring-Frameworks und wird mit der Simple-HTTP-Engine (*http://www.simpleframework.org*) ausgeliefert, die sich in Java-Applikationen einbetten läßt. Das Restletbeispiel in diesem Abschnitt wird mit Hilfe der Simple-HTTP-Engine in Betrieb genommen.

[75] Die Klasse `FibRestlet` nimmt das Beispiel mit den Fibonaccizahlen nochmals auf. Der mit Hilfe der Simple-HTTP-Engine in Betrieb genommene Dienst veranschaulicht wichtige Konstrukte eines Restlets:

```
package ch04.restlet;

import java.util.Collections;
import java.util.Map;
import java.util.HashMap;
import java.util.Collection;
import java.util.List;
```

```
import java.util.ArrayList;
import org.restlet.Component;
import org.restlet.Restlet;
import org.restlet.data.Form;
import org.restlet.data.MediaType;
import org.restlet.data.Method;
import org.restlet.data.Parameter;
import org.restlet.data.Protocol;
import org.restlet.data.Request;
import org.restlet.data.Response;
import org.restlet.data.Status;

public class FibRestlet {

    private Map<Integer, Integer> cache =
        Collections.synchronizedMap(new HashMap<Integer, Integer>());
    private final String xml_start = "<fib:response xmlns:fib = 'urn:fib'>";
    private final String xml_stop = "</fib:response>";

    public static void main(String[] args) {
        new FibRestlet().publish_service();
    }

    private void publish_service() {

        try {

            // Create a component to deploy as a service.
            Component component = new Component();

            // Add an HTTP server to connect clients to the component.
            // In this case, the Simple HTTP engine is the server.
            component.getServers().add(Protocol.HTTP, 7777);

            // Attach a handler to handle client requests. (Note the
            // similarity of the handle method to an HttpServlet
            // method such as doGet or doPost.)
            Restlet handler = new Restlet(component.getContext()) {
                @Override
                public void handle(Request req, Response res) {

                    Method http_verb = req.getMethod();

                    if (http_verb.equals(Method.GET)) {
                        String xml = to_xml();
                        res.setStatus(Status.SUCCESS_OK);
                        res.setEntity(xml, MediaType.APPLICATION_XML);
                    }
                    else if (http_verb.equals(Method.POST)) {
                        // The HTTP form contains key/value pairs.
                        Form form = req.getEntityAsForm();
                        String nums = form.getFirstValue("nums");
                        if (nums != null) {
                            // nums should be a list in the form: "[1, 2, 3]"
                            nums = nums.replace('[', '\0');
                            nums = nums.replace(']', '\0');
                            String[] parts = nums.split(",");
                            List<Integer> list = new ArrayList<Integer>();
                            for (String next : parts) {
                                int n = Integer.parseInt(next.trim());
                                cache.put(n, countRabbits(n));
                                list.add(cache.get(n));
```

```
                        }
                        String xml =
                          xml_start + "POSTed: " + list.toString() + xml_stop;
                        res.setStatus(Status.SUCCESS_OK);
                        res.setEntity(xml, MediaType.APPLICATION_XML);
                    }
                }
                else if (http_verb.equals(Method.DELETE)) {
                    cache.clear(); // remove the resource
                    String xml =
                        xml_start + "Resource deleted" + xml_stop;
                    res.setStatus(Status.SUCCESS_OK);
                    res.setEntity(xml, MediaType.APPLICATION_XML);
                }
                else // only GET, POST, and DELETE supported
                    res.setStatus(Status.SERVER_ERROR_NOT_IMPLEMENTED);
            }};

        // Publish the component as a service and start the service.
        System.out.println("FibRestlet at: http://localhost:7777/fib");
        component.getDefaultHost().attach("/fib", handler);
        component.start();

    }

    catch (Exception e) { System.err.println(e); }

}

private String to_xml() {
    Collection<Integer> list = cache.values();
    return xml_start + "GET: " + list.toString() + xml_stop;
}

private int countRabbits(int n) {
    n = Math.abs(n); // eliminate possibility of a negative argument

    // Easy cases.
    if (n < 2)
        return n;

    // Return cached values if present.
    if (cache.containsKey(n))
        return cache.get(n);
    if (cache.containsKey(n - 1) && cache.containsKey(n - 2)) {
       cache.put(n, cache.get(n - 1) + cache.get(n - 2));
       return cache.get(n);
    }

    // Otherwise, compute from scratch, cache, and return.
    int fib = 1, prev = 0;
    for (int i = 2; i <= n; i++) {
        int temp = fib;
        fib += prev;
        prev = temp;
    }
    cache.put(n, fib);
    return fib;
}
}
```

Das Restlet-Framework stellt einfach zu verwendende Java-Wrapperklassen wie `Method`, `Request`,

`Response`, `Form`, `Status` und `MediaType` für HTTP- und MIME-Konstrukte zur Verfügung. Das Framework unterstützt virtuelle Hosts für Applikationen auf kommerziellem Niveau.

[76] Die Restlest-Distribution beinhaltet ein Unterverzeichnis *$RESTLEST_HOME/lib*, welches die verschiedenen JAR-Dateien für das Restlet-Framework selbst und die Interoperabilität mit Tomcat, Jetty, Spring, Simple-HTTP-Engine und so weiter enthält. Der Restlet-Dienst in diesem Abschnitt setzt beispielsweise voraus, daß der Klassenpfad die Archivdateien *com.noelios.restlet.jar*, *org.restlet.jar* und *org.simpleframework.jar* enthält.

[77] Ein Restlet-Client könnte natürlich unter Anwendung einer Standardklasse wie `HttpURLConnection` geschrieben werden. Der folgende Client führt die clientseitige Restlet-API vor, eine API, die unabhängig von der serviceseitigen API verwendet werden kann:

```
import org.restlet.Client;
import org.restlet.data.Form;
import org.restlet.data.Method;
import org.restlet.data.Protocol;
import org.restlet.data.Request;
import org.restlet.data.Response;
import java.util.List;
import java.util.ArrayList;
import java.io.IOException;

class RestletClient {

    public static void main(String[] args) {
        new RestletClient().send_requests();
    }

    private void send_requests() {

        try {

            // Setup the request.
            Request request = new Request();
            request.setResourceRef("http://localhost:7777/fib");

            // To begin, a POST to create some service data.
            List<Integer> nums = new ArrayList<Integer>();
            for (int i = 0; i < 12; i++) nums.add(i);

            Form http_form = new Form();
            http_form.add("nums", nums.toString());
            request.setMethod(Method.POST);
            request.setEntity(http_form.getWebRepresentation());

            // Generate a client and make the call.
            Client client = new Client(Protocol.HTTP);

            // POST request
            Response response = get_response(client, request);
            dump(response);

            // GET request to confirm POST
            request.setMethod(Method.GET);
            request.setEntity(null);
            response = get_response(client, request);
            dump(response);

            // DELETE request
            request.setMethod(Method.DELETE);
            request.setEntity(null);
            response = get_response(client, request);
```

```
            dump(response);

            // GET request to confirm DELETE
            request.setMethod(Method.GET);
            request.setEntity(null);
            response = get_response(client, request);
            dump(response);

        }

        catch(Exception e) { System.err.println(e); }

    }

    private Response get_response(Client client, Request request) {
        return client.handle(request);
    }

    private void dump(Response response) {
        try {
            if (response.getStatus().isSuccess())
                response.getEntity().write(System.out);
            else
                System.err.println(response.getStatus().getDescription());
        }
        catch(IOException e) { System.err.println(e); }
    }
}
```

Die clientseitige API ist bemerkenswert klar. In diesem Beispiel veranlaßt der Client `POST`-, `DELETE`- und `GET`-Anfragen, um zu bestätigen, daß die `create`- und `delete`-Operationen des Dienstes erfolgreich verarbeitet wurden.

[78] Das Restlet-Framework ist schnell gelernt. Sein Reiz besteht hauptsächlich in der REST-Orientierung, die in eine leichtgewichtige aber mächtige Softwareumgebung zum Entwickeln und Nutzen von Webservices im REST-Stil mündet. Die wichtigste Frage lautet, ob das Restlet-Framework auf dem Markt und in den Köpfen ausreichend an Boden gewinnt, um zur Standardumgebung für Java-basierte Dienste im REST-Stil zu werden. Jersey trägt das Anerkennungssiegel des JSR, wodurch das Framework klar im Vorteil ist.

4.10 Ausblick

[79] Webservices, gleichsam ob SOAP-basiert oder im REST-Stil, sind oft sicherheitsbedürftig. Der Begriff „Sicherheit“ ist breit und undeutlich. Das nächste Kapitel klärt die Begriffe und untersucht die zur Absicherung von Webservices verfügbaren Verfahren. Der Schwerpunkt liegt auf der Authentifizierung und Autorisierung der Benutzer, gegenseitiger Authentifizierung sowie auf der Verbeziehungsweise Entschlüsselung von Nachrichten.

Kapitel 5

Sicherheit

Inhaltsübersicht

5.1 Sicherheitskonzepte bei Webservices im Überblick

[0] Die Sicherheitskonzepte für Webservices sind ein breit gefächertes Gebiet und können nicht auf einmal erschlossen werden. Das Gebiet ist so umfangreich, daß es in kleinere, besser handhabbare Stücke unterteilt werden muß. Hier eine Übersicht, wie sich dieses und das nächste Kapitel dem Thema nähern:

- *Sicherheit auf der Übertragungsebene*: Sicherheit beginnt auf der Transport- oder Übertragungsebene, das heißt bei den grundlegenden Protokollen, welche die Kommunikation zwischen einem Webservice, gleichsam SOAP-basiert oder im REST-Stil und seinen Clients steuern. Sicherheit auf dieser Ebene hat typischerweise drei Ausprägungen: Erstens brauchen Client und Dienst auf der Übertragungsebene die Gewißheit, daß jeder mit dem jeweils anderen kommuniziert und nicht etwa mit einem Schwindler. Zweitens müssen die von einer zur anderen Seite versendeten Daten stark genug verschlüsselt sein, so daß ein Lauscher nicht in der Lage ist, die Daten zu entschüsseln und somit Zugriff auf die eventuell darin enthaltenen Geheimnisse erhält. Drittens braucht jede Seite die Gewißheit, daß die empfangene Nachricht identisch mit der gesendeten Nachricht ist. Kapitel 5 deckt die Grundlagen der Sicherheit auf der Übertragungsebene anhand von Beispielen ab.
- *Benutzerauthentifizierung und -autorisierung*: Webservices bieten Clients Zugriff auf Resourcen. Ist eine Resource abgesichert, so muß der Client, um Zugriff zu erhalten, einen entsprechenden Berechtigungsnachweis beibringen. Der Berechtigungsnachweis wird in der Regel in einem zweiphasigen Ablauf vorgelegt und verifiziert. Im ersten Schritt übergibt der Client (Benutzer) eine identifizierende Information, zum Beispiel einen Benutzernamen, zusammen mit einem Berechtigungsnachweis, zum Beispiel einem Paßwort. Wird der Berechtigungsnachweis *nicht* akzeptiert, so wird der Zugriff auf die angeforderte Resource verweigert. Diese erste Phase wird als *Benutzerauthentifizierung* (*user authentication*) bezeichnet. Im optionalen zweiten Schritt werden die Zugriffsberechtigungen des authentifizierten Benutzers ermittelt. Beispielsweise könnten die Benutzer eines Stock-Picking-Dienstes zwar jeweils einen Benutzernamen und ein Paßwort erhalten, vom Dienst aber in Kategorien eingeordnet werden, etwa Privat- und Geschäftskunden oder Standard- und Premiumkunden. Der Zugriff auf bestimmte Resourcen kann Geschäftskunden vorbehalten sein. Diese zweite Phase wird als *Benutzerautorisierung* (*user authorization*) oder *Rollenautorisierung* (*role authorization*) bezeichnet. Kapitel 5 behandelt diese *rollenbasierte Sicherheit*. Kapitel 6 baut auf dieser Einführung auf.
- *Web Services Security(Web Services Security)*: WSS ist eine Sammlung von Protokollen, die beschreiben, wie verschiedene Sicherheitsebenen in der Nachrichtenübertragung bei SOAP-basierten Diensten implementiert werden können. WSS beschreibt beispielsweise, wie sich digitale Signaturen und Informationen über die gewählte Verschlüsselung in den SOAP-Header einsetzen lassen. Denken Sie daran, daß SOAP-basierte Dienste entwurfsbedingt neutral bezüglich des Transportprotokolls sind. Dementsprechend sorgt WSS für umfassende Sicherheit zwischen dem einen und dem anderen Ende, unabhängig vom unterliegenden Transportprotokoll. Kapitel 5 stellt WSS anhand eines per `Endpoint`-Publisher in Betrieb genommenen Dienstes vor. Kapitel 6 baut auf dieser Einführung auf, in dem WSS bei Applikationsservern wie GlassFish untersucht wird.

5.2 Sicherheit auf der Übertragungsebene

[1] Ein kostenpflichtiger Dienst, wie der Amazon Simple Storage Service (S3), muß den Client einer Anfrage authentifizieren, das Speichern und Abholen von Daten ermöglichen und dafür sorgen, daß der Dienst nur von zahlenden Kunden genutzt werden kann, wobei jeder Kunde nur Zugriff auf seine eigenen Daten hat. Die Version des S3-Dienstes im REST-Stil authentifiziert den Client einer Anfrage per HMAC (Keyed-Hash Message Authentication Code). Die Authentifizierung funktioniert zusammengefaßt wie folgt:

- Aus einigen Stücken der Anfrage wird eine neue Zeichenkette zusammengesetzt. Diese Zeichenkette, die sogenannte „Eingabenachricht“ (*input message*) ist eines von zwei Argumenten eines Hashverfahrens.

- Das zweite Argument ist der eindeutige geheime Schlüssel, der jedem zahlenden Kunden von Amazon ausgestellt wird. Der generierte Hashwert, auch als *Message-Digest* bezeichnet, hat stets eine feste Länge, unabhängig von der Länge der Eingabenachricht (Abbildung 5.1). Der S3-Dienst verwendet SHA-1 (Secure Hash Algorithm 1) als Hashverfahren. SHA-1 liefert für jede Eingabenachricht einer Länge kleiner als 2^{64} Bit einen 160 Bit langen Digest. Amazon bezeichnet diesen Hashwert als „Signatur", da seine Verwendungsweise derjenigen einer digitalen Signatur ähnelt. Genau genommen, ist eine digitale Signatur aber ein *verschlüsselter Message-Digest*. Die Signatur im Sinne von Amazon, ist dagegen *nicht verschlüsselt*.
- Diese Signatur wird als Wert des `Authorization`-Headers in die Anfrage eingesetzt.
- Beim Empfang der Anfrage validiert der S3-Dienst zuerst die Signatur und verarbeitet die Anfrage erst, wenn die Validierung erfolgreich war.

[2] Welcher Mechanismus schützt eine Anfrage an den S3-Dienst davor, daß sie abgefangen und der Digest nach dem `Authorization`-Header gestohlen wird? Der S3-Dienst setzt voraus, daß die Anfrage über eine per HTTPS (Hypertext Transfer Protocol Secure) gesicherte Verbindung übertragen wird. *HTTPS* entspricht HTTP mit einer zusätzlichen Sicherheitsschicht. Die ursprüngliche Entwicklung und Implementierung dieser Sicherheitsschicht stammt von Netscape und heißt *SSL* (Secure Sockets Layer). Die *IETF* (Internet Engineering Task Force) hat SSL übernommen und in *TLS* (Transport Layer Security) umbenannt. SSL und TLS unterscheiden sich zwar in der Versionsnummerierung und einigen technischen Details, sind aber im Grunde genommen identisch. Daher werden die Abkürzungen SSL, TLS und SSL/TLS häufig synonym verwendet.

[3] Java verfügt über verschiedene Packages zur Unterstützung von SSL/TLS im Allgemeinen und HTTPS im Besonderen. Beispielsweise gehört *JSSE* (Java Secure Socket Extension) seit Version 1.4 zu Java. An dieser Stelle ist von Interesse, daß die höheren Sicherheitsebenen, wie die Benutzerauthentifizierung, in der Regel die durch HTTPS gewährleistete Sicherheit auf der Übertragungsebene voraussetzen. Dementsprechend beginnt die Diskussion der Sicherheitskonzepte bei Webservices mit HTTPS.

Abbildung 5.1: Erzeugen eines Message-Digests.

5.2.1 Gegenseitige Authentifizierung, Geheimhaltung und Fälschungssicherheit

[4] HTTPS ist unter den sicheren HTTP-Versionen am weitesten verbreitet. HTTPS erweitert die Transportfunktionalität von HTTP um die drei grundlegenden, in der Kapiteleinleitung auf Seite 186 bereits beschriebenen Sicherheitskonzepte: Authentifizierung von Client und Dienst, Geheimhaltung beziehungsweise Datenschutz sowie Fälschungssicherheit. In Abbildung 5.1 sendet Alice eine geheime Nachricht an Bob. Eve kann die Nachricht abhören und sogar versuchen, Alice und Bob hinters Licht zu führen, so daß beide glauben, mit dem jeweils anderen zu kommunizieren, während beide statt dessen in Wirklichkeit mit Eve verkehren. Eine solche Sicherheitsverletzung wird als *MITM* (Man-in-the-middle Attack) bezeichnet. Folgende drei Sicherheitskonzepte müssen implementiert sein, damit Alice und Bob sicher kommunizieren können:

- *Gegenseitige Authentifizierung* (*peer authentication*): Bob muß sich bei Alice authentifizieren („seine Identität belegen"), damit Alice weiß, wer sich auf der Empfängerseite befindet, bevor sie eine geheime Nachricht sendet. Alice muß sich ebenso bei Bob authentifizieren, damit Bob sicher sein kann, daß die geheime Nachricht von Alice stammt und nicht von einem Schwindler

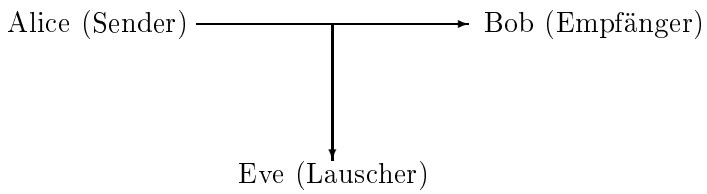

Abbildung 5.2: Übertragung einer geheimen Nachricht von Alice an Bob. Eve fängt die Nachricht ab.

wie Eve. Dieser Schritt wird als *gegenseitige Authentifizierung* oder auch *gegenseitige Aufforderung* (*mutual authentication*) bezeichnet.

- *Geheimhaltung/Datenschutz* (*confidentiality*): Nachdem sich Alice und Bob gegenseitig authentifiziert haben, muß Alice die geheime Nachricht so verschlüsseln, daß sie nur von Bob entschlüsselt werden kann. Selbst wenn Eve die verschlüsselte Nachricht abfängt, sollte sie nicht in der Lage sein, die Nachricht zu lesen, da die Entschlüsselung enorme Rechenleistung oder unglaubliches Glück erfordert.
- *Fälschungssicherheit* (*data integrity*): Die von Alice gesendete Nachricht muß identisch mit der Nachricht sein, die Bob empfängt und andernfalls eine Fehlermeldung angezeigt werden. Die empfangene Nachricht kann sich aus verschiedenen Gründen von der gesendeten unterscheiden, zum Beispiel aufgrund von Störungen während der Übertragung oder einer absichtlichen Manipulation durch Eve. Jeder Unterschied zwischen den gesendeten und der empfangenen Nachricht muß erkannt werden.

Diese Sicherheitskonzepte können auf verschiedenen Wegen implementiert werden. Vor der Untersuchung der Implementierung dieser Konzepte bei HTTPS ist es nützlich, die Verschlüsselung (*encryption*) und Entschlüsselung (*decryption*) von Daten zu betrachten.

5.2.2 Symmetrische und asymmetrische Verschlüsselung

[5] Moderne Ansätze zur Datenverschlüsselung beschreiben entweder ein symmetrisches oder ein asymmetrisches Verfahren. Sowohl die symmetrischen als auch die asymmetrischen Verfahren erwarten stets zwei „Argumente“, nämlich die zu verschlüsselnden Daten und einen Schlüssel (Abbildung 5.3). Die verschlüsselten Daten werden gleichbedeutend als *chiffrierte Daten* (*cipher bits*) bezeichnet. Beim Verschlüsseln von Textdaten wird die Eingabe als *Klartext* (*plaintext*) und die Ausgabe als *Chiffretext* (*ciphertext*) bezeichnet. Auch die Entschlüsselungsverfahren, sowohl symmetrisch als auch asymmetrisch, erwarten zwei „Argumente“, nämlich die verschlüsselten Daten und einen „Entschlüsselungsschlüssel“. Die Entschlüsselung liefert die ursprünglichen unverschlüsselten Daten. Bei der symmetrischen Verschlüsselung wird ein und derselbe Schlüssel, auch als *geheimer Schlüssel* (*secret key*) bezeichnet, zur Verschlüsselung und zur Entschlüsselung verwendet (Abbildung 5.4). Die symmetrische Verschlüsselung hat zwar den Vorteil relativ hoher Geschwindigkeit, dafür aber den Nachteil des sogenannten *Schlüsselverteilungsproblems* (*key distribution problem*), das heißt der

Abbildung 5.3: Grundsätzlicher Vorgang der asymmetrischen Verschlüsselung/Entschlüsselung.

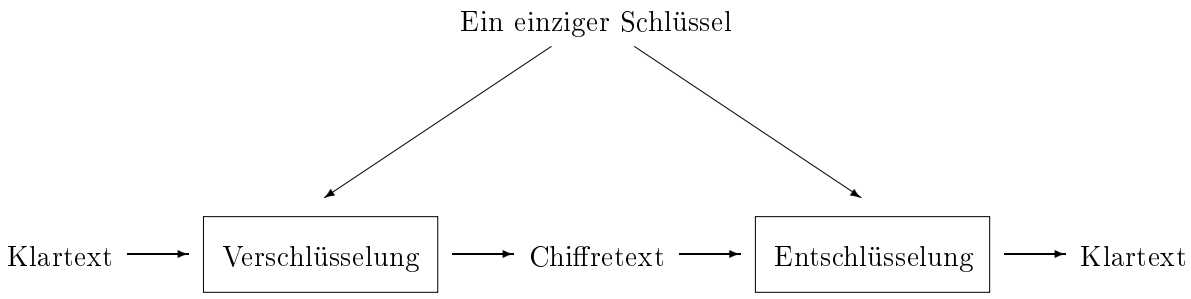

Abbildung 5.4: Symmetrische Verschlüsselung/Entschlüsselung mit einem einzigen Schlüssel.

sicheren Übergabe des geheimen Schlüssels an Sender und Empfänger.

[6] Der asymmetrischen Verschlüsselung liegt ein *Schlüsselpaar* (*key pair*) zugrunde, bestehend aus einem *geheimen Schlüssel* (*secret key*) und einem *öffentlichen Schlüssel* (*public key*). Der geheime Schlüssel darf, wie sein Name andeutet, nicht weitergegeben, sondern muß vom Erzeuger des Schlüsselpaars stets sicher verwahrt werden. Der öffentliche Schlüssel kann dagegen bedenkenlos verteilt werden. Mit dem öffentlichen Schlüssel verschlüsselte Daten können nur mit dem geheimen Schlüssel entschlüsselt werden und umgekehrt (Abbildung 5.5). Die asymmetrische Verschlüsselung löst zwar das *Schlüsselverteilungsproblem*, aber die asymmetrische Verschlüsselung und Entschlüsselung sind *erheblich langsamer* als die entsprechenden symmetrischen Vorgänge.

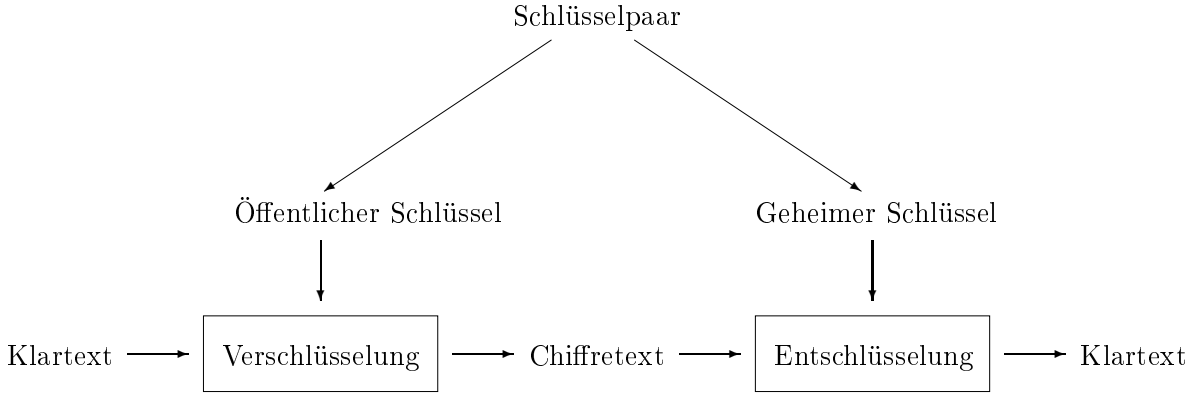

Abbildung 5.5: Asymmetrische Verschlüsselung und Entschlüsselung mit einem Schlüsselpaar, bestehend aus einem öffentlichen und einem geheimen Schlüssel.

[7] Die asymmetrische Verschlüsselung löst das Geheimhaltungsproblem bei Alice und Bob. Wenn Alice ihre Nachricht mit Bobs öffentlichem Schlüssel verschlüsselt und Bob das einzige Exemplar des geheimen Schüssels seines Schlüsselpaares besitzt, kann nur Bob die Nachricht entschlüsseln. Selbst wenn Eve Alice' Nachricht abfängt, reicht Bobs öffentlicher Schlüssel nicht aus, um die Nachricht zu entschlüsseln.

5.2.3 Implementierung von Authentifizierung, Geheimhaltung und Fälschungssicherheit bei HTTPS

[8] Die Fälschungssicherheit, das letzte der drei Sicherheitskonzepte, ist am einfachsten zu bewerkstelligen. Eine über HTTPS versendete Nachricht beinhaltet einen Digest, den der Empfänger

erneut berechnet. Unterscheiden sich der gesendete und der vom Empfänger neu berechnete Digest, so wurde die Nachricht während der Übertragung versehentlich oder absichtlich verändert. Auch eine Änderung des gesendeten Digests selbst, während der Übertragung, zählt als Verletzung der Fälschungssicherheit.

[9] HTTPS erwirkt die gegenseitige Authentifizierung durch Austauschen *digitaler Zertifikate*. Häufig verlangt nur der Client die Authentifizierung des Servers. Stellen Sie sich eine typische Webapplikation vor, in der ein Kunde eine Bestellung für den Inhalt seines Einkaufskorbes abschließt, in dem er seine Kreditkartennummer an den Händler übermittelt. Hier eine Zusammenfassung der typischen Vorgänge, wenn der clientseitige Webbrowser und der Webserver den Aufbau einer HTTPS-Verbindung aushandeln:

- Der Webbrowser des Kunden fordert den Webserver des Anbieters auf, sich zu authentifizieren. Der Server antwortet, in dem er eines oder mehrere digitale Zertifikate an den Webbrowser sendet.
- Der Webbrowser prüft das digitale Zertifikat des Webservers anhand seines *Truststores* (einer Liste von digitalen Zertifikaten, denen der Webbrowser vertraut). Die Prüfung des erhaltenen Zertifikates durch den Webbrowser verläuft aus praktischen Gründen typischerweise indirekt. Angenommen, der Webbrowser erhält ein Zertifikat von Amazon, hat aber kein solches Zertifikat in seiner Liste. Ferner angenommen, daß das Zertifikat von Amazon eine garantierte Unterschrift von VeriSign trägt, einer bekannten Zertifizierungsstelle (*certificate authority*). Enthält die Liste des Webbrowsers ein Zertifikat von VeriSign, so kann der Webbrowser anhand dieses Zertifikates die digitale Signatur von VeriSign auf dem Zertifikat von Amazon prüfen. Die Zertifikateliste des Webbrowsers ist dessen Vorrat von Zertifikaten, anhand derer erhaltene Zertifikate geprüft werden. Kann der Webbrowser ein erhaltenes Zertifikat mit keinem Zertifikat aus seiner Liste vergleichen, so fragt der Webbrowser typischerweise den Benutzer, ob er diesem Zertifikat nur diesesmal oder immer wieder vertrauen soll. Wählt der Benutzer die Option „immer wieder“, so ergänzt der Webbrowser seine Liste um das neue Zertifikat.
- Der Webserver verlangt typischerweise keine Authentifizierung des Webbrowsers. Die Webapplikation ist an der Kreditkartennummer des Kunden interessiert, nicht an der Identität seines Webbrowsers.

[10] Die typischerweise einseitige Authentifizierungsaufforderung, wobei der Client die Authentifizierung des Servers verlangt, aber nicht umgekehrt, zeigt sich in der Konfigurationsdatei *server.xml* von Tomcat. Der Eintrag für HTTPS lautet:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS" />
```

Das `clientAuth`-Attribut ist mit `false` bewertet, d.h. Tomcat fordert keine Authentifizierung des Clients an. Das Standardverhalten des Clients ist dagegen, die Authentifizierung des Servers zu verlangen.

[11] HTTPS verläßt sich bei der Authentifizierung und Geheimhaltung auf digitale Zertifikate. Diese sind durch die Verwendung asymmetrischer Kryptosysteme weit verbreitet, gerade weil die Verteilung geheimer Schlüssel auf viele Benutzer so schwierig ist. Die folgende Liste faßt zusammen, wie Authentifizierung und Geheimhaltung bei HTTPS ineinander greifen. Der Vorgang wird gelegentlich als *Handschlag* zwischen Client und Server bezeichnet, der zum Aufbau einer sicheren Netzwerkverbindung führt. Der Client kann hier ein Webbrowser oder eine Applikation sein, die sich wie der Client eines Webservice' verhält. Ein Einfachheit halber faßt der Begriff „Webserver“ hier sowohl eigenständige Webapplikationsserver wie Tomcat, als auch `Endpoint`-Publisher von Webservices und

voll ausgestattete Java-Applikationsserver wie BEA WebLogic, GlassFish, JBoss oder WebSphere zusammen:

- Der Client fordert den Webserver auf, sich zu authentifizieren. Der Webserver sendet ein oder mehrere digitale Zertifikate zur Authentifizierung. Moderne digitale Zertifikate gehorchen in der Regel dem X.509-Format. Die aktuelle Versionsnummer des X.509-Standards ist v3.
- Ein X.509-Zertifikat ist ein *digitales Zertifikat*, welches als *Identitätsnachweis* dient, da es mit dem öffentlichen Schlüssel des Schlüsselpaares einer Identität verschlüsselt wurde, zum Beispiel einer Person (etwa Alice) oder einer Organisation (etwa Bobs Arbeitgeber). Das Zertifikat enthält die digitale Signatur einer Zertifizierungsstelle wie VeriSign. Zertifikate können aber zu Versuchszwecken auch selbst unterschrieben werden. Durch das Unterschreiben eins digitalen Zertifikates bestätigt die Zertifizierungsstelle das Zertifikat und verifiziert, daß der öffentliche Schlüssel des Zertifikats an eine bestimmte Identität gebunden ist. Beispielsweise unterschreibt VeriSign Alice' Zertifikat und verifiziert dadurch, daß der öffentliche Schlüssel dieses Zertifikats zu Alices Schlüsselpaar gehört.
- Der Client kann entscheiden, ob er das beziehungsweise die digitalen Zertifikate des Servers akzeptiert, in dem er sie mit seiner Liste vergleicht.
- Der Server kann die Authentifizierung des Clients verlangen.
- Nach Abschluß der Zertifizierungsphase erzeugt der Client einen geheimen Schlüssel. Der Vorgang beginnt damit, daß der Client einen *vorläufigen Schlüssel* erzeugt, eine Zeichenkette, die gemeinsam mit dem Server verwendet wird. Der vorläufige Schlüssel wird von Client und Server verwendet, um den beidseitig identischen *eigentlichen geheimen Schlüssel* zu generieren, welcher zur Verschlüsselung und Entschlüsselung des Verkehrs zwischen Client und Server verwendet wird. Dabei erhebt sich die Frage, wie der vorläufige Schlüssel sicher vom Client zum Server transportiert wird.
- Der Client verschlüsselt den 48 Bit langen vorläufigen Schlüssel mit dem öffentlichen Schlüssel des Servers, der über das während der gegenseitigen Authentifizierung heruntergeladene digitale Zertifikat des Servers erhältlich ist. Das verschlüsselte vorläufige Schlüssel wird dem Server übergeben, der ihn wieder entschlüsselt. Ist nach wie vor alles in Ordnung, so bestätigt jede Seite, daß der verschlüsselte Verkehr beginnt. Das Schlüsselpaar aus dem öffentlichen und dem privaten Schlüssel ist somit wesentlich für die Lösung des Verteilungsproblems bei geheimen Schlüsseln.
- Sowohl der Client als auch der Server können jederzeit darauf bestehen, den gesamten Vorgang von neuem zu beginnen. Hat zum Beispiel entweder Alice oder Bob den Verdacht, daß Eve nichts Gutes im Schilde führt, kann sowohl Alice als auch Bob den Handschlagvorgang von vorne beginnen.

[12] Ein (gemeinsamer) geheimer Schlüssel wird aus verschiedenen Gründen zur Verschlüsselung und Entschlüsselung verwendet. Erstens ist symmetrische Verschlüsselung vergleichweise schnell. Zweitens verfügt der Server nicht über den öffentlichen Schlüssel des Clients, wenn der Server keine Authentifizierung anfordert, kann also Nachrichten an den Client nicht verschlüsseln. Der Server kann Nachrichten an den Client nicht mit seinem geheimen Schlüssel verschlüsseln, da eine solche Nachricht von jedem Empfänger der den öffentlichen Schlüssel des Servers zur Verfügung hat (zum Beispiel Eve) entschlüsselt werden kann. Schließlich ist Verschlüsselung und Entschlüsselung mit zwei separaten Schlüsseln von Natur aus kniffliger als mit nur einem Schlüssel.

[13] Die Herausforderung besteht darin, den vorläufigen Schlüssel sicher vom Client zum Server zu übertragen. Der öffentliche Schlüssel des Servers, der dem Client durch das digitale Zertifikat des Servers nach der gegenseitigen Authentifizierung zur Verfügung steht, erfüllt diese Aufgabe perfekt.

Der eigentliche geheime Schlüssel wird erst erzeugt, nachdem sich Client und Server über die zu verwendende Cipher-Suite beziehungsweise kryptographischen Algorithmen geeinigt haben. Eine Cipher-Suite enthält einen key-pair algorithm und einen Hashalgorithmus.

[14] Digitale Zertifikate spielen zwar eine dominante Rolle in gegenseitigen Authentifizierungssituationen, sind aber nicht die einzige Lösung. Beispielsweise implementiert SRP (Secure Remote Password) die gegenseitigen Authentifizierung ohne digitale Zertifikate. Sie finden weitere Informationen zu SRP unter der Adresse *http://srp.stanford.edu*.

5.2.4 Die Klasse HttpsURLConnection

[15] Es ist an der Zeit, diese architektonischen Skizzen durch ein Beispiel zu untermauern. Die abstrakte Klasse `HttpsURLConnection` unterstützt HTTPS-Verbindungen. Die folgende Applikation `SunClient` stützt sich auf ein Objekt vom Typ `HttpsURLConnection`, um eine `GET`-Anfrage über HTTPS an die Java-Homepage *https://java.sun.com:443* zu veranlassen. Beachten Sie die Portnummer 443 in der URL, den Standardport für HTTPS-Verbindungen:

```
import java.net.URL;
import javax.net.ssl.HttpsURLConnection;
import java.net.MalformedURLException;
import java.security.cert.Certificate;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

// Send a GET request over HTTPS to the Java home site and
// display information about the security artifacts that come back.
class SunClient {

    private static final String url_s = "https://java.sun.com:443";

    public static void main(String[] args) {
        new SunClient().do_it();
    }

    private void do_it() {
        try {
            URL url = new URL(url_s);
            HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
            conn.setDoInput(true);
            conn.setRequestMethod("GET");
            conn.connect();
            dump_features(conn);
        }
        catch(MalformedURLException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }
    }
    private void dump_features(HttpsURLConnection conn) {
        try {
            print("Status code:  " + conn.getResponseCode());
            print("Cipher suite: " + conn.getCipherSuite());
            Certificate[] certs = conn.getServerCertificates();
            for (Certificate cert : certs) {
                print("\tCert. type: " + cert.getType());
                print("\tHash code:  " + cert.hashCode());
                print("\tAlgorithm:  " + cert.getPublicKey().getAlgorithm());
                print("\tFormat:     " + cert.getPublicKey().getFormat());
```

```
                print("");
            }
        }
        catch(Exception e) { System.err.println(e); }
    }
    private void print(String s) { System.out.println(s); }
}
```

Die Ausgabe eines Aufrufes lautet bei mir:

```
Status code:  200
Cipher suite: SSL_RSA_WITH_RC4_128_MD5
        Cert. type: X.509
        Hash code:  23073427
        Algorithm:  RSA
        Format:     X.509

        Cert. type: X.509
        Hash code:  32560810
        Algorithm:  RSA
        Format:     X.509

        Cert. type: X.509
        Hash code:  8222443
        Algorithm:  RSA
        Format:     X.509
```

[16] Der Statuswert zeigt an, daß die `GET`-Anfrage erfolgreich verarbeitet wurde. Der Bezeichner der Cipher-Suite läßt sich wie folgt erklären:

- *RSA*: Gibt den verwendeten *Verschlüsselungsalgorithmus* an, benannt nach den ehemaligen MIT-Professoren Rivest, Shamir und Adleman, die das Verfahren entworfen haben. RSA ist der am häufigsten verwendete Verschlüsselungsalgorithmus und dient zur Verschlüsselung des vorläufigen Schlüssels, der vom Client zum Server gesendet wird.

- *RC4_128*: Der zur Verschlüsselung des Verkehrs zwischen Client und Server verwendete *Stromverschlüsselungsalgorithmus (stream cipher algorithm)* mit einer Schlüsselänge von 128 Bit. Der Buchstabe „R“ steht für Rivest von RSA, der Buchstabe „C“ für Cipher (Chiffrierung). (Gelegentlich wird „RC“ auch als Abkürzung für „*Rons Code*“ gedeutet, da Rivests Vorname Ron lautet.) RC4 ist der am häufigsten verwendete Stromverschlüsselungsalgorithmus. RC4_128 verschlüsselt den Datenverkehr nach Abschluß des Handschlags.

- *MD5*: Der identifizierende 128 Bit lange Hashwert des Zertifikates, auch als dessen *Fingerabdruck* bezeichnet, wird mit dem Message-Digest Algorithm 5 generiert. MD5 liefert, was offiziell als kryptographische Hashfunktion bezeichnet wird. Rivest hat MD5 als Verbesserung von MD4 geplant. MD5 hat zwar keine schwerwiegenden Fehler, verliert aber Schritt für Schritt Boden an Alternativen wie die Familie der SHA-Algorithmen (Secure Hash Algorithm).

Der Server von Sun hat während der gegenseitigen Authentifizierungsphase drei digitale Zertifikate gesendet. Jedes davon ein per RSA-Algorithmus generiertes X.509-Zertifikat. Jeder der unterschiedlichen MD5-Digests ist 128 Bit lang. Das Format der X.509-Zertifikate gehorcht verständlicherweise der X.509-Spezifikation.

[17] Die drei an die `SunClient`-Applikation gesendeten digitalen X.509-Zertifikate können anhand der Zertifikateliste des Clients geprüft werden (*$JAVA_HOME/jre/lib/security/cacerts*). Während der Entwicklungsphase ist es unter Umständen sinnvoll, die Prüfung abzuschalten, zum Beispiel weil die Liste keine passenden Zertifikate beinhaltet.

[18] Die folgende `SunTrustingClient`-Applikation ist eine überarbeitete Version der Klasse `SunClient` mit abgeschalteter Prüfung.

```
import java.net.URL;
import java.security.SecureRandom;
import java.security.cert.X509Certificate;
import javax.net.ssl.SSLContext;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;
import java.net.MalformedURLException;
import java.security.cert.Certificate;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

class SunTrustingClient {
    private static final String url_s = "https://java.sun.com:443";

    // Send a GET request and print the response status code.
    public static void main(String[] args) {
        new SunTrustingClient().do_it();
    }
    private void do_it() {
        try {
            // Configure HttpsURLConnection so that it doesn't check certificates.
            SSLContext ssl_ctx = SSLContext.getInstance("SSL");
            TrustManager[] trust_mgr = get_trust_mgr();
            ssl_ctx.init(null,                // key manager
                         trust_mgr,           // trust manager
                         new SecureRandom()); // random number generator
            HttpsURLConnection.setDefaultSSLSocketFactory(ssl_ctx.getSocketFactory());
            URL url = new URL(url_s);
            HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
            conn.setDoInput(true);
            conn.setRequestMethod("GET");
            conn.connect();
            dump_features(conn);
        }
        catch(MalformedURLException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }
        catch(Exception e) { System.err.println(e); }
    }
    private TrustManager[] get_trust_mgr() {
        // No exceptions thrown in the three overridden methods.
        TrustManager[] certs = new TrustManager[] {
            new X509TrustManager() {
                public X509Certificate[] getAcceptedIssuers() { return null; }
                public void checkClientTrusted(X509Certificate[] c, String t) { }
                public void checkServerTrusted(X509Certificate[] c, String t) { }
            }
        };
        return certs;
    }

    private void dump_features(HttpsURLConnection conn) {
        try {
            print("Status code:  " + conn.getResponseCode());
            print("Cipher suite: " + conn.getCipherSuite());
```

```
                Certificate[] certs = conn.getServerCertificates();
                for (Certificate cert : certs) {
                    print("\tCert. type: " + cert.getType());
                    print("\tHash code:  " + cert.hashCode());
                    print("\tAlgorithm:  " + cert.getPublicKey().getAlgorithm());
                    print("\tFormat:     " + cert.getPublicKey().getFormat());
                    print("");
                }
            }
            catch(Exception e) { System.err.println(e); }
        }

        private void print(String s) {
            System.out.println(s);
        }
    }
```

Die geänderte Version verwendet eine eigene Zertifikateliste (*TrustManager*), um die Voreinstellung zu überschreiben. Ein trust manager validiert Zertifikate. Die Ausgaben von `SunTrustingClient` und `SunClient` sind im wesentlich gleich, da auch die neue Version drei Zertifikate vom Sun-Server erhält.

5.3 Absichern des RabbitCounter-Dienstes

[19] Die statische `Endpoint`-Methode `publish()` unterstützt keine HTTPS-Verbindungen. Die SE 6 beinhaltet aber die abstrakte Klasse `com.sun.net.httpserver.HttpServer` und die davon abgeleitete ebenfalls abstrakte Klasse `com.sun.net.httpserver.HttpsServer`, mit deren Hilfe sich ein Dienst unter HTTPS in Betrieb nehmen läßt. (Die abstrakte Klasse `Endpoint` stützt sich hinter den Kulissen ebenfalls auf `HttpServer`.) Die Klasse `HttpsServer` kann an sich verwendet werden, um einen Dienst im REST-Stil wie den `RabbitCounter`-Dienst in Betrieb zu nehmen. Hier ist die überarbeitete Version der Publisherklasse für den `RabbitCounter`-Dienst, die HTTPS-Verbindungsanfragen akzeptiert:

```
package ch05.rc;

import java.net.InetSocketAddress;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLParameters;
import javax.net.ssl.SSLEngine;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;
import java.security.cert.X509Certificate;
import java.security.SecureRandom;
import java.security.KeyStore;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.TrustManagerFactory;
import java.io.FileInputStream;
import javax.xml.ws.http.HTTPException;
import java.io.OutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.util.Collections;
import java.util.Map;
import java.util.HashMap;
import java.util.Collection;
```

```
import java.util.List;
import java.util.ArrayList;
import com.sun.net.httpserver.HttpContext;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpsServer;
import com.sun.net.httpserver.HttpsConfigurator;
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpsParameters;

public class RabbitCounterPublisher {
    private Map<Integer, Integer> fibs;

    public RabbitCounterPublisher() {
        fibs = Collections.synchronizedMap(new HashMap<Integer, Integer>());
    }

    public static void main(String[] args) {
        new RabbitCounterPublisher().publish();
    }

    public Map<Integer, Integer> getMap() { return fibs; }

    private void publish() {
        int port = 9876;
        String ip = "https://localhost:";
        String path = "/fib";
        String url_string = ip + port + path;

        HttpsServer server = get_https_server(ip, port, path);
        HttpContext http_ctx = server.createContext(path);

        System.out.println("Publishing RabbitCounter at " + url_string);
        if (server != null) server.start();
        else System.err.println("Failed to start server. Exiting.");
    }

    private HttpsServer get_https_server(String ip, int port, String path) {
        HttpsServer server = null;
        try {
            InetSocketAddress inet = new InetSocketAddress(port);
            // 2nd arg = max number of client requests to queue
            server = HttpsServer.create(inet, 5);

            SSLContext ssl_ctx = SSLContext.getInstance("TLS");
            // password for keystore
            char[] password = "qubits".toCharArray();
            KeyStore ks = KeyStore.getInstance("JKS");
            FileInputStream fis = new FileInputStream("rc.keystore");
            ks.load(fis, password);

            KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
            kmf.init(ks, password);
            TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
            tmf.init(ks);
            ssl_ctx.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

            // Create SSL engine and configure HTTPS to use it.
            final SSLEngine eng = ssl_ctx.createSSLEngine();
            server.setHttpsConfigurator(new HttpsConfigurator(ssl_ctx) {
                    public void configure(HttpsParameters parms) {
                        parms.setCipherSuites(eng.getEnabledCipherSuites());
                        parms.setProtocols(eng.getEnabledProtocols());
                    }
```

```
                });
            server.setExecutor(null); // use default

            HttpContext http_context =
                server.createContext(path, new MyHttpHandler(this));
        }
        catch(Exception e) { System.err.println(e); }
        return server;
    }
}

// The handle method is called on a particular request context,
// in this case on any request to the server that ends with /fib.
class MyHttpHandler implements HttpHandler {
    private RabbitCounterPublisher pub;
    public MyHttpHandler(RabbitCounterPublisher pub) { this.pub = pub; }

    public void handle(HttpExchange ex) {
        String verb = ex.getRequestMethod().toUpperCase();
        if (verb.equals("GET"))         doGet(ex);
        else if (verb.equals("POST"))   doPost(ex);
        else if (verb.equals("DELETE")) doDelete(ex);
        else throw new HTTPException(405);
    }

    private void respond_to_client(HttpExchange ex, String res) {
        try {
            ex.sendResponseHeaders(200, 0); // 0 means: arbitrarily many bytes
            OutputStream out = ex.getResponseBody();
            out.write(res.getBytes());
            out.flush();
            ex.close(); // closes all streams
        }
        catch(IOException e) { System.err.println(e); }
    }
    private void doGet(HttpExchange ex) {
        Map<Integer, Integer> fibs = pub.getMap();
        Collection<Integer> list = fibs.values();
        respond_to_client(ex, list.toString());
    }

    private void doPost(HttpExchange ex) {
        Map<Integer, Integer> fibs = pub.getMap();
        fibs.clear(); // clear to create a new map
        try {
            InputStream in = ex.getRequestBody();
            byte[] raw_bytes = new byte[4096];
            in.read(raw_bytes);
            String nums = new String(raw_bytes);
            nums = nums.replace('[', '\0');
            nums = nums.replace(']', '\0');
            String[] parts = nums.split(",");
            List<Integer> list = new ArrayList<Integer>();
            for (String next : parts) {
                int n = Integer.parseInt(next.trim());
                fibs.put(n, countRabbits(n));
                list.add(fibs.get(n));
            }
            Collection<Integer> coll = fibs.values();
            String res = "POSTed: " + coll.toString();
```

```
                respond_to_client(ex, res);
            }
            catch(IOException e) { }
        }
        private void doDelete(HttpExchange ex) {
            Map<Integer, Integer> fibs = pub.getMap();
            fibs.clear();
            respond_to_client(ex, "All entries deleted.");
        }
        private int countRabbits(int n) {
            n = Math.abs(n);
            if (n < 2) return n; // easy cases

            Map<Integer, Integer> fibs = pub.getMap();
            // Return cached values if present.
            if (fibs.containsKey(n)) return fibs.get(n);
            if (fibs.containsKey(n - 1) &&
                fibs.containsKey(n - 2)) {
                fibs.put(n, fibs.get(n - 1) + fibs.get(n - 2));
                return fibs.get(n);
            }

            // Otherwise, compute from scratch, cache, and return.
            int fib = 1, prev = 0;
            for (int i = 2; i <= n; i++) {
                int temp = fib;
                fib += prev;
                prev = temp;
            }
            fibs.put(n, fib);
            return fib;
        }
    }
```

[20] Das Objekt vom Typ `HttpsServer` ist schnell erzeugt:

```
HttpsServer server = null;
try {
    InetSocketAddress inet = new InetSocketAddress(port);
    server = HttpsServer.create(inet, 5);
```

Das zweite Argument der `create()`-Methode, hier 5, ist die Höchstanzahl von Anfragen, die in eine Warteschlange eingereiht werden. Der Wert 0 bedeutet, daß die Systemvoreinstellung verwendet werden soll.

[21] Die Anweisungen zur Konfiguration eines Objektes vom Typ `HttpsServer` sind kniffliger als diejenigen, um das Objekt zu erzeugen. Die Konfiguration betrifft eine Reihe wesentlicher Eigenschaften, die sich um den „SSL-Kontext“ (das `SSLContext`-Objekt) drehen, der mit einem spezifischen Protokoll wie TLS verknüpft ist. In diesem Beispiel dient der SSL-Kontext zwei Konfigurationsschritten. Der erste Konfigurationsschritt betrifft den sogenannten *Keystore* und den *Truststore*. Der serverseitige Keystore verwaltet die digitalen Zertifikate des Servers, die im Rahmen der gegenseitigen Authentifizierung an den Client übergeben werden können. Der clientseitige Keystore beinhaltet die digitalen Zertifikate des Clients für denselben Zweck. Es gibt keinen voreingestellten Keystore. Das folgende Kommando erzeugt die Keystoredatei *rc.keystore* (der Dateiname ist frei wählbar):

```
% keytool -genkey -keyalg RSA -keystore rc.keystore
```

Das Kommando erzeugt ein einzelnes selbstunterschriebenes X.509-Zertifikat und speichert es in der Datei *rc.keystore*. Das `keytool`-Kommando gehört zur SE 6 und fragt eine Reihe von Daten ab, die zum Ausstellen des X.509-Zertifikates verwendet werden, darunter ein Paßwort, hier `qubits`. Zur Laufzeit wird der Inhalt des Keystores in die Applikation geladen, damit während des TLS-Handschlages jedes digitale Zertifikat an den Client gesendet werden kann.

[22] Der zweite Konfigurationsschritt betrifft den Truststore, das heißt die Datei der digitalen Zertifikate, denen die Applikation vertraut. (Der voreingestellte Truststore ist *$JAVA_HOME/jre/lib/-security/cacerts*.) Nach der Konfiguration von Key- und Truststore übergibt die Anweisung

```
ssl_ctx.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
```

diese Einstellungen an den SSL-Kontext. Das dritte Argument, hier `null`, gibt an, daß die Standardzufallsgeneratorklasse `java.security.SecureRandom` verwendet werden soll. Die Konfiguration von Key- und Truststore läßt sich alternativ mit Hilfe des Kommandozeilenschalters `-D` oder über das Bewerten von Systemeigenschaften im Quelltext vornehmen.

[23] Im letzte Schritt der Konfiguration wird der SSL-Kontext verwendet, um ein Objekt vom Typ `SSLEngine` zu erzeugen, welches wiederum die verfügbaren Protokolle und Cipher-Suites zum Aufbau einer sicheren Verbindung bereitstellt. In diesem Beispiel werden die Systemvoreinstellungen verwendet und keine zusätzlichen Protokolle oder Cipher-Suites zur Standardkombination hinzugenommen.

[24] Der letzte Schritt vor dem Starten des Servers ist der Aufbau eines HTTP-Kontextes (Objekt vom Typ `HttpContext`), bestehend aus einem Pfad (hier `/fibs`) und einem HTTP-Behandler (Objekt vom Typ ***`HttpHandler`***), dessen `handle()`-Methode bei jeder Anfrage an die Resource bei `/fibs` aufgerufen wird. Der Behandler wird mit dem HTTPS-Server verknüpft. Der HTTPS-Server delegiert die Anfrageverarbeitung effektiv an die spezifizierte `handle()`-Methode. Ein sorgfältiger ausgearbeiteter HTTPS-Server könnte mehrere HTTP-Kontexte mit separaten Behandlern haben.

[25] Hier ist die Implementierung der im Interface ***`HttpHandler`*** deklarierten `handle()`-Methode in der Klasse `MyHttpHandler`:

```
public void handle(HttpExchange ex) {
     String verb = ex.getRequestMethod().toUpperCase();
     if (verb.equals("GET"))         doGet(ex);
     else if (verb.equals("POST"))   doPost(ex);
     else if (verb.equals("DELETE")) doDelete(ex);
     else throw new HTTPException(405); // bad verb
}
```

Der Argumenttyp, die abstrakte Klasse `HttpExchange`, definiert Methoden für den Zugriff auf einen Eingabestrom, der zum Einlesen clientseitiger `POST`-Anfragen verwendet werden kann sowie auf einen Ausgabestrom, der zum Erzeugen des Körpers einer Antwort an den Client verwendet werden kann. `HttpExchange` definiert fernerhin Methoden, um den Statuswert der Antwort zu setzen und weitere Aufgaben. Die `respond_to_client()`-Methode, zum Beispiel, behandelt `GET`-, `POST`- und `DELETE`-Anfragen:

```
private void respond_to_client(HttpExchange ex, String res) {
    try {
        ex.sendResponseHeaders(200, 0); // 0 means: arbitrarily many bytes
        OutputStream out = ex.getResponseBody();
        out.write(res.getBytes());
        out.flush();
        ex.close(); // closes all streams
    }
```

```
        catch(IOException e) { System.err.println(e); }
    }
```

Der HTTPS-Client für die HTTPS-Version des `RabbitCounter`-Dienstes unterscheidet sich nur geringfügig von den zuvor besprochenen Klassen `SunClient` und `SunTrustingClient`. Wiederum wird ein Objekt vom Typ `HttpsURLConnection` gewählt und der Einfachheit halber so konfiguriert, daß die vom Server erhaltenen digitalen Zertifikate nicht geprüft werden. Eine Änderung betrifft die anonyme Implementierung des Interface' *`HostnameVerifier`*, welche dem Client die Entscheidung gestattet, ob eine HTTPS-Verbindung zu einem bestimmten Server aufgebaut werden soll. Die entsprechenden Zeilen lauten (die Referenzvariable `conn` verweist auf das Objekt vom Typ `HttpsURLConnection`):

```
conn.setHostnameVerifier(new HostnameVerifier() {
          public boolean verify(String host, SSLSession sess) {
              if (host.equals("localhost")) return true;
              else return false;
          }});
```

Die `verify()`-Methode kann je nach dem implementiert werden, welche Verbindungslogik zur Applikation paßt. In diesem Fall sind HTTPS-Verbindungen zu einem Server erlaubt, der auf `localhost` läuft. Bei Verbindungsversuchen zu anderen Servern wird eine Ausnahme ausgeworfen. Nun zum Quelltext des Clients, der zum Testen eine `POST`-Anfrage an den Server sendet:

```
import java.util.List;
import java.util.ArrayList;
import java.net.URL;
import java.security.SecureRandom;
import java.security.cert.X509Certificate;
import javax.net.ssl.SSLContext;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;
import java.net.HttpURLConnection;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;
import java.net.MalformedURLException;
import java.security.cert.Certificate;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

class SecureClientRC {

    private static final String url_s = "https://localhost:9876/fib";

    public static void main(String[] args) {
        new SecureClientRC().do_it();
    }

    private void do_it() {

        try {

            // Create a context that doesn't check certificates.
            SSLContext ssl_ctx = SSLContext.getInstance("TLS");
            TrustManager[] trust_mgr = get_trust_mgr();
            ssl_ctx.init(null,                // key manager
                         trust_mgr,           // trust manager
```

```
                    new SecureRandom()); // random number generator
        HttpsURLConnection.setDefaultSSLSocketFactory(ssl_ctx.getSocketFactory());

        URL url = new URL(url_s);
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();

        // Guard against "bad hostname" errors during handshake.
        conn.setHostnameVerifier(new HostnameVerifier() {
                public boolean verify(String host, SSLSession sess) {
                    if (host.equals("localhost")) return true;
                    else return false;
                }
            });

        // Test request
        List<Integer> nums = new ArrayList<Integer>();
        nums.add(3); nums.add(5); nums.add(7);

        conn.setDoInput(true);
        conn.setDoOutput(true);
        conn.setRequestMethod("POST");
        conn.connect();
        OutputStream out = conn.getOutputStream();
        out.write(nums.toString().getBytes());

        byte[] buffer = new byte[4096];
        InputStream in = conn.getInputStream();
        in.read(buffer);
        System.out.println(new String(buffer));
        dump_features(conn);
        conn.disconnect();
    }

    catch(MalformedURLException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
    catch(Exception e) { System.err.println(e); }

}

private TrustManager[] get_trust_mgr() {
    TrustManager[] certs = new TrustManager[] {
        new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() { return null; }
            public void
                checkClientTrusted(X509Certificate[] certs, String t) { }
            public void
                checkServerTrusted(X509Certificate[] certs, String t) { }
        }
    };
    return certs;
}

private void dump_features(HttpsURLConnection conn) {
    try {
        print("Status code:  " + conn.getResponseCode());
        print("Cipher suite: " + conn.getCipherSuite());
        Certificate[] certs = conn.getServerCertificates();
        for (Certificate cert : certs) {
            print("\tCert. type: " + cert.getType());
            print("\tHash code:  " + cert.hashCode());
            print("\tAlgorithm:  " + cert.getPublicKey().getAlgorithm());
            print("\tFormat:     " + cert.getPublicKey().getFormat());
```

```
                    print("");
                }
            }
            catch(Exception e) { System.err.println(e); }
        }

        private void print(String s) { System.out.println(s); }
    }
```

Die Ausgabe lautet:

```
POSTed: [2, 5, 13]
Status code:  200
Cipher suite: SSL_RSA_WITH_RC4_128_MD5
        Cert. type: X.509
        Hash code:  1992213
        Algorithm:  RSA
        Format:     X.509
```

Der Keystore des Servers, die Datei *rc.keystore*, enthält nur ein einziges selbstunterschriebenes X.509-Zertifikat. Die Cipher-Suite ist mit beiden Sun-Beispielen identisch. Das `keytool`-Kommando gestattet die Angabe anderer Cipher-Suites.

[26] As proof of concept hier ein Perl-Client für den `RabbitCounter`-Dienst, der nach dem Aufruf des Java-Clients mit dem obigen Ergebnis aufgerufen wurde:

```
#!/usr/bin/perl -w

use Net::SSLeay qw(get_https);
use strict;

my ($type, $start_line, $misc, $extra) = get_https('localhost', 9876, '/fib');
print "Type/value:  $type\n";
print "Start line:  $start_line\n";
print "Misc:        $misc => $extra\n";
```

Der Perl-Client prüft die Zertifikate des Servers per Voreinstellung nicht. Die Prüfung kann jedoch mühelos zugeschaltet werden. Die Subroutine `get_https()` gibt eine Liste mit vier Einträgen zurück.

[27] Die Ausgabe lautet:

```
Type/value:  a [2, 5, 13]

Start line:  HTTP/1.1 200 OK
Misc:        TRANSFER-ENCODING => chunked
```

Das Zeichen `a` in der ersten Zeile gibt an, daß der Dienst ein Array zurückgibt.

5.3.1 Adding User Authentication

[28] Zunächst eine kurze Zusammenfassung der zuvor eingeführten Sprechweisen. Die Benutzerauthentifizierung ist der erste von zwei Schritten eines Verfahrens, das unter der Bezeichnung *rollenbasierte Sicherheit* bekannt ist. Beim ersten Schritt legt der Benutzer einen Berechtigungsnachweis vor, zum Beispiel ein Passwort, um den Status eines *authentifizierten Subjektes* zu erlangen. Auch anspruchsvollere Berechtigungsnachweise wie Chipkarten oder biometrische Daten (beispielsweise Fingerabdruck oder Augenhintergrund) können zur Benutzerauthentifizierung verwendet werden.

[29] Der zweite Schritt, die Autorisierung, ist optional und schließt sich in der Regel nur an, wenn der erste Schritt erfolgreich war. In dieser Phase werden die Rollen bestimmt, für die ein authentifiziertes Subjekt autorisiert ist. Autorisierungsrollen können nach Bedarf angepaßt werden. Unixbasierte

Betriebssysteme unterscheiden beispielsweise zwischen Benutzern und dem Administrator. Beide Autorisierungsrollen bestimmen den Umfang des Zugriffs auf Systemresourcen und welche privilegierten Tätigkeiten eine Rolle begleiten. Ein IT-Unternehmen könnte Autorisierungsrollen wie „Programmierer“, „erfahrener Datenbankadministrator“, „Systemanalyst“ und so weiter verwenden. In großen Unternehmen dienen unter Umständen digitale Zertifikate zur Bestimmung von Autorisierungsrollen. Ein auf Benutzerrollen basierendes Sicherheitssystem könnte beispielsweise von Fred Feuerstein die Angabe von Benutzername und Paßwort verlangen, um zu einem authentifizierten Subjekt zu werden und anhand einer Datenbank ein Zertifikat nachschlagen, welches Fred als Kranführer autorisiert.

5.3.2 HTTP-BASIC-Authentifizierung

[30] Mit einigen geringfügigen Änderungen ist der `RabbitCounter`-Dienst ist der Lage, die `BASIC`-Authentifizierung von HTTP zu unterstützen. JAX-WS bietet zwei komfortable Konstanten zur Auswertung von Benutzername und Paßwort. Alle Änderungen finden lokal in der anonymen Implementierung des Interface' *`HttpHandler`* statt. Die `handle()`-Methode ruft nun die Methode `authenticate()` auf, die eine `HTTPException` mit dem Statuswert `401` auswirft, wenn die Authentifizierung scheitert und hat darüberhinaus keine andere Aufgabe:

Beispiel 5.1: HTTP-BASIC-Authentifizierung beim RabbitCounter-Dienst.

```
public void handle(HttpExchange ex) {
    authenticate(ex);
    String verb = ex.getRequestMethod().toUpperCase();
    if (verb.equals("GET"))         doGet(ex);
    else if (verb.equals("POST"))   doPost(ex);
    else if (verb.equals("DELETE")) doDelete(ex);
    else throw new HTTPException(405);
}

private void authenticate(HttpExchange ex) {
    // Extract the header entries.
    Headers headers = ex.getRequestHeaders();
    List<String> ulist = headers.get(BindingProvider.USERNAME_PROPERTY);
    List<String> plist = headers.get(BindingProvider.PASSWORD_PROPERTY);

    // Extract username/password from the two singleton lists.
    String username = ulist.get(0);
    String password = plist.get(0);
    if (!username.equals("fred") || !password.equals("rockbed"))
        throw new HTTPException(401); // authentication error
}
```

Im produktiven Betrieb würden Benutzername und Paßwort mit einer Datenbank verglichen werden. Die Änderung des Quelltextes erfordert außerdem zwei zusätzliche `import`-Anweisungen, nämlich für das Interface *`javax.xml.ws.BindingProvider`* und die Klasse `com.sun.net.httpserver.Headers`.

[31] Die clientseitigen Änderungen sind ebenfalls miminal. Benutzername und Paßwort werden der HTTP-Anfrage hinzugefügt, bevor die `connect()`-Methode des Objektes vom Typ `HttpsURLConnection` aufgerufen wird. In diesem Abschnitt des Quelltextes verweist die Referenzvariable `conn` auf das Objekt vom Typ `HttpsURLConnection`:

```
conn.addRequestProperty(BindingProvider.USERNAME_PROPERTY, "fred");
conn.addRequestProperty(BindingProvider.PASSWORD_PROPERTY, "rockbed");
```

Wesentlich ist, daß die Authentifizierungsinformation mit Hilfe der Sicherheitmechanismen auf der Übertragungsebene an den `RabbitCounter`-Dienst gesendet werden, insbesondere verschlüsselt.

5.4 Containergestützte Sicherheit für Webservices

[32] Die jüngste Version des `RabbitCounter`-Dienstes implementiert Benutzerauthentifizierung, allerdings nicht in maßgeblicher Weise. Ein Webservicecontainer, der nicht nur Benutzerauthentifizierung, sondern auch Sicherheit auf der Übertragungsebene bietet, wäre eine bessere Lösung. Tomcat, die Referenzimplementierung des Java-Webcontainers, bietet beides. Abschnitt 4.5 hat gezeigt, wie sich ein Webservice im REST-Stil mittels Tomcat in Form eines Servlet deployen läßt. Tomcat kann auch SOAP-basierte Webservices in Betrieb nehmen. Tomcat kann sowohl einen Webservice als auch einen Webservice-Provider in Betrieb nehmen.

[33] Das Beispiel zur Veranschaulichung wie Tomcat containergesteuerte Sicherheit liefert, wird in zwei Schritten aufgebaut. Im ersten Schritt wird ein SOAP-basierter Dienst per Tomcat in Betrieb genommen und im zweiten Schritt die Sicherheit hinzugefügt. [Ein späteres Beispiel] zeigt einen sicheren Webservice-Provider mit Hilfe von Tomcat.

5.4.1 Deployment eines Webservice' mit Tomcat

[34] Der SOAP-basierte Dienst ist in gewohnter Weise strukturiert. Zunächst der Quelltext des SEI:

```
package ch05.tc;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public interface TempConvert {
    @WebMethod float c2f(float c);
    @WebMethod float f2c(float f);
}
```

Der Quelltext der entsprechenden SIB:

```
package ch05.tc;
import javax.jws.WebService;

@WebService(endpointInterface = "ch05.tc.TempConvert")
public class TempConvertImpl implements TempConvert {
    public float c2f(float t) { return 32.0f + (t * 9.0f / 5.0f); }
    public float f2c(float t) { return (5.0f / 9.0f) * (t - 32.0f); }
}
```

Das Deployment unter Tomcat setzt einen DD voraus, die Datei *web.xml*. Diese Konfigurationsdatei informiert den Servletcontainer darüber, wie Anfragen an den Dienst behandelt werden sollen. Der DD deklariert insbesondere zwei spezielle Klassen, die zu Version 2.1 des JAX-WS gehören: `WSServlet` und `WSServletContextListener`. Der folgende Abschnitt des DDs:

```
<servlet>
    <servlet-name>TempConvertWS</servlet-name>
    <servlet-class>
        com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
</servlet>
<servlet-mapping>
```

```
        <servlet-name>TempConvertWS</servlet-name>
        <url-pattern>/tc</url-pattern>
    </servlet-mapping>
```

delegiert Anfragen, deren URL mit dem Pfad `/tc` endet, an ein `WSServlet`-Objekt, welches wiederum in der JAX-WS-Laufzeitbibliothek verankert ist. Tomcat stellt die Klasse `WSServlet` zur Verfügung.

[35] Der folgende Abschnitt des DDs:

```
    <listener>
        <listener-class>
            com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>
```

deklariert die Behandlerklasse `WSServletContextListener`, die eine zweite Sun-spezifische Konfigurationsdatei namens *sun-jaxws.xml* einliest. Die Datei *sun-jaxws.xml* liefert den Endpunkt des Webservice' durch Verknüpfung des `WSServlet`-Objektes mit der SIB. Die Datei *sun-jaxws.xml* enthält folgende Zeilen:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
        <endpoint
            name="TempConvertWS"
            implementation="ch05.tc.TempConvertImpl"
            url-pattern="/tc" />
    </endpoints>
```

`TempConvertWS` ist der Name des Servlets `WSServlet` im DD. Wie die Syntax andeutet, kann die Konfigurationsdatei mehrere Endpunkte deklarieren, spezifiziert aber in diesem Fall nur einen.

[36] In der zum Deployment verwendete WAR-Datei liegen die Dateien *web.xml* und *sun-jaxws.xml* im Verzeichnis *WEB-INF*. Zur Übersicht nun einmal der vollständige DD:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                                 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"2
             version="2.4">
        <listener>
            <listener-class>
                com.sun.xml.ws.transport.http.servlet.WSServletContextListener
            </listener-class>
       </listener>
       <servlet>
           <servlet-name>TempConvertWS</servlet-name>
           <servlet-class>
               com.sun.xml.ws.transport.http.servlet.WSServlet
           </servlet-class>
       </servlet>
       <servlet-mapping>
           <servlet-name>TempConvertWS</servlet-name>
           <url-pattern>/tc</url-pattern>
       </servlet-mapping>
    </web-app>
```

Der Dienst wird wie eine gewöhnliche Webapplikation unter Tomcat deployt. Exemplare der übersetzten SEI und SIB müssen im Verzeichnis *WEB-INF/classes/ch05/tc* deponiert sein, die beiden Konfigurationsdateien selbst im Verzeichnis *WEB-INF*. Beachten Sie, daß der Dienst per Voreinstellung im Dokumentstil ist und literale Kodierung verwendet. Somit können Sie das `wsgen`-Kommando aufrufen:

```
% wsgen -cp . WEB-INF.classes.ch05.tc.TempConvertImpl
```

um die benötigten JAX-B-Artefakte zu generieren.

[37] Die WAR-Datei, deren Name frei gewählt werden kann, wird nun mit dem folgenden Kommando erzeugt:

```
% jar cvf tempc.war WEB-INF
```

und zum Deployment in das Verzeichnis *$TOMCAT_HOME/webapps* kopiert. Erfolgreiches Deployment läßt sich anhand der Protokolldateien von Tomcat oder durch den Versuch bestätigen, in einem Browserfenster die URL *http://localhost:8080/tempc/tc?wsdl* aufzurufen.

[38] Das `wsgen`-Kommando liefert nun a anhand des WSDL-Dokumentes die clientseitigen Artefakte:

```
% wsimport -keep -p tcClient http://localhost:8080/tempc/tc?wsdl
```

Hier ein Client, basierend auf den `wsimport`-Artefakten:

```
import tcClient.TempConvertImplService;
import tcClient.TempConvert;

class ClientTC {
    public static void main(String args[]) throws Exception {

        TempConvertImplService service = new TempConvertImplService();
        TempConvert port = service.getTempConvertImplPort();

        System.out.println("f2c(-40.1) ==> " + port.f2C(-40.1f));
        System.out.println("c2f(-40.1) ==> " + port.c2F(-40.1f));
        System.out.println("f2c(+98.7) ==> " + port.f2C(+98.7f));
    }
}
```

Die Ausgabe lautet:

```
f2c(-40.1) ==> -40.055557
c2f(-40.1) ==> -40.18
f2c(+98.7) ==> 37.055557
```

5.4.2 Sicherung eines Webservice' bei Tomcat

[39] Der *`TempConvert`*-Dienst bedarf keinerlei Änderung. Dies ist einer der offensichtlichen Vorteile des Tomcat-Containers, statt die Sicherheit in der Applikation selbst zu implementieren.

[40] Der erste Schritt ist das Zuschalten des *Tomcat-Konnektors* für SSL/TLS. Ein Konnektor ist ein Endpunkt für Anfragen von Clients. Der Abschnitt

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" maxThreads="150"
           scheme="https" secure="true" clientAuth="false" sslProtocol="TLS" />
```

in *$TOMCAT_HOME/conf/server.xml*, der Hauptkonfigurationsdatei von Tomcat, muß gegebenenfalls einkommentiert (die Kommentarzeichen entfernt) werden. War der Abschnitt auskommentiert, so muß Tomcat neu gestartet werden, damit die Änderung wirksam wird. Tomcat erwartet per Voreinstellung HTTPS-Anfragen an Port 8443. Die Portadresse kann aber geändert werden.

[41] Sendet ein Client eine HTTPS-Anfrage an Tomcat, so benötigt Tomcat im Rahmen der gegenseitigen Authentifizierung ein digitales Zertifikat. Das folgende Kommando erzeugt ein selbstunterschriebenes Zertifikat:

```
% keytool -genkey -alias tomcat -keyalg RSA
```

Das `keytool`-Kommando fragt eine Reihe von Daten ab, die zum Ausstellen des X.509-Zertifikates benötigte werden, unter anderem ein Paßwort, die mit dem speziellen, von Tomcat erwarteten Wert `changeit` versehen werden sollten. Das Zertifikat wird per Voreinstellung in der Datei *.keystore* im Homeverzeichnis des Benutzers deponiert. Die Option `-keystore` gestattet aber die Wahl eines beliebigen Dateinamens für den Keystore.

[42] Beim lokalen System muß die Keystoredatei im Homeverzeichnis des Benutzers liegen, den Tomcat startet. Die Konfigurationsdatei *server.xml* gestattet ebenfalls die Deklaration des Pfades zu einer Keystoredatei und dies scheint die sicherste Lösung zu sein. Hier der geänderte `<Connector>`-Abschnitt für die HTTPS-Konfiguration bei mir:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" maxThreads="150"
           scheme="https" secure="true" clientAuth="false" sslProtocol="TLS"
           keystoreFile="/home/mkalin/.keystore" />
```

Tomcat kann nun neu gestartet und durch Eingabe der URL *https://localhost:8443* in einem Browserfenster getestet werden. Da der Browser das neue, selbstunterschriebene Zertifikat des Servers noch nicht in seinem eigenen Truststore hat, wird er sich erkundigen, ob das Zertifikat akzeptiert werden und in diesem Fall, ob es einmalig oder permant akzeptiert werden soll. Im letzten Fall wird das Zertifikat in den Truststore des Browsers aufgenommen.

[43] Auf der Clientseite ist eine leichte Modifikation der Klasse `ClientTC` erforderlich. Dieser Client stützt sich auf die per `wsimport` generierten Artefakte, insbesondere die Klasse `tcClient.TempConvertImplService`, welche die Endpunkt-URL auf den Wert *http://localhost:8080/tempc/tc?wsdl* setzt. Es gibt keinen Grund, um den Endpunkt zu ändern, da das WSDL-Dokument nach wie vor unter dieser URL zur Verfügung steht. Dennoch muß der Client die Verbindung mit der URL *https://localhost:8443* anstelle von *http://localhost:8080* aufbauen. Die Anpassung läßt sich im Quelltext des Clients mit Hilfe des Interface' *`BindingProvider`* bewerkstelligen. Hier der überarbeitete Client:

```
import tcClient.TempConvertImplService;
import tcClient.TempConvert;
import javax.xml.ws.BindingProvider;
import java.util.Map;

class ClientTCSecure {

    private static final String endpoint = "https://localhost:8443/tempc/tc";

    public static void main(String args[]) {
        TempConvertImplService service = new TempConvertImplService();
        TempConvert port = service.getTempConvertImplPort();

        Map<String, Object> req_ctx = ((BindingProvider) port).getRequestContext();
        req_ctx.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpoint);

        System.out.println("f2c(-40.1) ==> " + port.f2C(-40.1f));
        System.out.println("c2f(-40.1) ==> " + port.c2F(-40.1f));
        System.out.println("f2c(+98.7) ==> " + port.f2C(+98.7f));
    }
}
```

Tomcat fordert per Voreinstellung keine Authentifizierungs des Clients an, aber das Standardverhalten kann sich ändern. Der Client kann sich jedenfalls nicht darauf verlassen, nicht zur Authentifizierung aufgefordert zu werden und muß somit über einen Keystore mit identifizierendem digitalen

Zertifikat verfügen. Der Client benötigt ferner einen Truststore mit dem das von Tomcat erhaltene Zertifikat verglichen werden kann. Keystore und Truststore dienen zwar unterschiedlichen Zwecken, aber jedes von beiden ist eine Datei mit Zertifikaten im selben Format. Zur Vereinfachung erfüllt der Keystore von Tomcat drei Aufgaben, in dem er zugleich als Keystore und Truststore des Clients dient. Die Sicherheitsarchitektur ist dabei der Gegenstand des Interesses, nicht die einzelnen Trust- und Keystores. Das folgende Kommando dient zum Aufrufen des Client. Die `-D`-Optionen übergeben die Informationen über die Trust- und Keystores:

```
% java -Djavax.net.ssl.trustStore=/home/mkalin/.keystore
       -Djavax.net.ssl.trustStorePassword=changeit
       -Djavax.net.ssl.keyStore=/home/mkalin/.keystore
       -Djavax.net.ssl.keyStorePassword=changeit ClientTC
```

Das Kommando ist kompliziert genug, um Ant oder ein entsprechendes Skript zu verwenden. Die Ausgabe ist natürlich identisch mit der vorigen.

5.4.3 Applikationsgesteuerte Authentifizierung

[44] Der nächste Schritt zur Absicherung des *`TempConvert`*-Dienstes ist die Hinzunahme von Authentifizierung und Autorisierung. Diese Ergänzungen können auf der Applikationsebene, das heißt im Quelltext des Webservice' selbst oder auf der Ebene des Containers zugeschaltet werden. Dabei werden gemeinsame Resourcen wie eine Datenbank von Benutzernamen und Paßwörtern verwendet. Die containergesteuerte Authentifizierung hat den offensichtlichen Reiz, daß sich Tomcat um alle Sicherheitsangelegenheiten kümmert, so daß Sie sich im Webservice auf die Applikationslogik konzentrieren können. Die containergesteuerte Authentifizierung führt zu saubererem Quelltext, in dem sie die AOP-Praxis befolgt, die Sicherheitsangelegenheiten dem Container zu überlassen, statt den verschiedenen Applikationen die der Container enthält.

[45] Die Authentifizierung auf Applikationsebene bringt nur wenige Anweisungen mit sich, verunreinigt aber den Quelltext durch die Kombination von Sicherheitsaspekten und der eigentlichen Geschäftslogik. Beispiel 5.1 auf Seite 203 implementiert den wohl einfachsten Ansatz auf Applikationsebene, nämlich die Übergabe von Benutzername und Paßwort als Teil des Anfragekontextes. Hier ist der überarbeitete Quelltext des Clients `ClientTC`:

```
import tcClient.TempConvertImplService;
import tcClient.TempConvert;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.MessageContext;
import java.util.Map;
import java.util.HashMap;
import java.util.Collections;
import java.util.List;

class ClientTC {

    private static final String endpoint = "https://localhost:8443/tempc/tc";

    public static void main(String args[]) {
        TempConvertImplService service = new TempConvertImplService();
        TempConvert port = service.getTempConvertImplPort();

        Map<String, Object> req_ctx = ((BindingProvider) port).getRequestContext();
        req_ctx.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpoint);

        // Place the username/password in the HTTP request headers,
        // which a non-Java client can do as well.
        Map<String, List<String>> hdr = new HashMap<String, List<String>>();
```

```
        hdr.put("Username", Collections.singletonList("fred"));
        hdr.put("Password", Collections.singletonList("rockbed"));
        req_ctx.put(MessageContext.HTTP_REQUEST_HEADERS, hdr);

        // Invoke service methods.
        System.out.println("f2c(-40.1) ==> " + port.f2C(-40.1f));
        System.out.println("c2f(-40.1) ==> " + port.c2F(-40.1f));
        System.out.println("f2c(+98.7) ==> " + port.f2C(+98.7f));
    }
}
```

Die überarbeitete Klasse `ClientTC` platziert Benutzername und Paßwort nun in den HTTP-Headern, um zu betonen, daß dieser Ansatz nicht Java-spezifisch ist. Benutzername und Paßwort sind zur Klarheit hartkodiert, obwohl sie voraussichtlich als Kommandozeilenargumente übergeben werden würden.

[46] Die Änderung an der Klasse `TempConvertImpl` sind relativ geringfügig. Die zusätzliche `authenticated()`-Methode prüft Benutzername und Paßwort. Im produktiven Betrieb würden die Daten wahrscheinlich mit Werten in einer Datenbank verglichen werden:

```
package ch05.tc;

import javax.jws.WebService;
import javax.annotation.Resource;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPException;
import java.util.Map;
import java.util.List;

@WebService(endpointInterface = "ch05.tc.TempConvert")
public class TempConvertImpl implements TempConvert {
    @Resource
    WebServiceContext ws_ctx;

    public float c2f(float t) {
        if (authenticated()) return 32.0f + (t * 9.0f / 5.0f);
        else throw new HTTPException(401); // authorization error
    }

    public float f2c(float t) {
        if (authenticated()) return (5.0f / 9.0f) * (t - 32.0f);
        else throw new HTTPException(401); // authorization error
    }

    private boolean authenticated() {
        MessageContext mctx = ws_ctx.getMessageContext();
        Map http_headers = (Map) mctx.get(MessageContext.HTTP_REQUEST_HEADERS);
        List ulist = (List) http_headers.get("Username");
        List plist = (List) http_headers.get("Password");

        // proof of concept authentication
        if (ulist.contains("fred") && plist.contains("rockbed")) return true;
        else return false;
    }
}
```

Der Nachteil der Authentifizierung im Quelltext der Applikation ist offensichtlich. Die Operationen des Webservice', hier `f2c()` und `c2f()`, sind hier eine Mischung aus Applikationslogik und Sicherheitsverarbeitung, in diesem Beispiel allerdings nur geringfügig. Die `authenticated()`-Methode benötigt Zugriff auf den Nachrichtenkontext, das heißt die Art systemnaher Verarbeitung, die am

besten Behandler überlassen bleibt. Das Beispiel ist klein genug, um die Vermischung in Grenzen zu halten, deutet aber an, wie kompliziert die Kombination von Applikations- und Sicherheitslogik bei einem realistischen Webservice werden kann. Darüberhinaus skaliert dieser Ansatz schlecht. Benötigt der Benutzer Fred Zugriff auf andere authentifizierungspflichtige Dienste, so wiederholt sich die hier gezeigte Kombination aus Applikations- und Sicherheitslogik bei jedem dieser Dienste. Eine attraktive Alternative zur Vorgehensweise in diesem Beispiel ist es, die Authentifizierung dem Container zu überlassen.

5.4.4 Containergesteuerte Authentifizierung und Autorisierung

[47] Tomcat bietet containergesteuerte Authentifizierung und Autorisierung. Das Konzept des *Administrationsbereichs* (*realm*) spielt beim Tomcat-Ansatz eine zentrale Rolle. Ein Administrationsbereich ist eine Gruppe von Resourcen, darunter HTML-Seiten und Webservices, mit vorgesehener Authentifizierungs- und Autorisierungseinrichtung. Nach den Tomcat-Dokumentation ähnelt ein Administrationsbereich einer Unix-Gruppe, im Hinblick auf die Zugriffsberechtigungen. Der Administrationsbereich ist ein Organisationswerkzeug, welches die Zuweisung einer einzigen Richtlinie zur Zugriffskontrolle an eine Gruppe von Resourcen gestattet.

[48] Tomcat liefert fünf Standardplugins, deren Name das Wort *Realm* enthält, um die Kommunikation zwischen Container und der Authentifizierungs-/Autorisierungseinheit zu steuern. Die fünf Plugins sind, jeweils mit einer kurzen Beschreibung:

- *JDBCRealm*: Die Authentifizierungsinformationen sind in einer relationalen Datenbank gespeichert, die über den JDBC-Standardtreiber zugänglich ist.
- *DataSourceRealm*: Die Authentifizierungsinformationen sind wiederum in einer relationalen Datenbank gespeichert, und über ein Objekt vom JDBC-Typ `DataSource` erreichbar, welches über den JNDI-Dienst (Java Naming and Directory Interface) zur Verfügung steht.
- *JNDIRealm*: Die Authentifizierungsinformationen sind in einem *LDAP-basierten* (Lightweight Directory Access Protocol) Verzeichnisdienst gespeichert, der über einen JNDI-Anbieter verfügbar ist.
- *MemoryRealm*: Die Authentifizierungsinformationen sind wird beim Starten des Containers aus der Datei *$TOMCAT_HOME/conf/tomcat-users.xml* eingelesen. Dies ist die einfachste Variante und die Voreinstellung.
- *JAASRealm*: Die Authentifizierungsinformationen sind über einen *JAAS*-Anbieter (Java Authentication and Authorization Service) verfügbar, der wiederum Bestandteil von Java-Applikationsservern wie BEA WebLogic, GlassFish und WebSphere ist.

Bei allen diesen Varianten ist der Tomcat-Container und nicht die Applikation Anbieter der Sicherheitsmechanismen.

5.4.5 Konfiguration der containergesteuerten Sicherheit bei Tomcat

[49] Der Sicherheitsansatz bei Tomcat ist *deklarativ* statt *programmatisch*, das heißt die Details über den Administrationsbereich werden in einer Konfigurationsdatei und nicht im Quelltext eingestellt. Die Konfigurationsdatei ist der DD *web.xml* in der deployten WAR-Datei.

[50] Dies ist der DD für die nächste Version `TempConvert`-Dienstes, der das Tomcat-Plugin MemoryRealm verwendet:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" version="2.4">
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>TempConvertWS</servlet-name>
    <servlet-class>
      com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
  </servlet>

  <security-role>
    <description>The Only Secure Role</description>
    <role-name>bigshot</role-name>
  </security-role>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Users-Roles Security</web-resource-name>
      <url-pattern>/tcauth</url-pattern>
    </web-resource-collection>

    <auth-constraint>
      <role-name>bigshot</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>

  <servlet-mapping>
    <servlet-name>TempConvertWS</servlet-name>
    <url-pattern>/tcauth</url-pattern>
  </servlet-mapping>
</web-app>
```

Vier Punkte in dieser Version des DDs sind wichtig:

- Die gesicherten Resourcen werden mit Hilfe des `<web-resource-collection>`-Elementes deklariert. Die Gruppe umfaßt in diesem Fall alle Resourcen, die über den Pfad */tcauth* erreichbar sind, den Pfad zum *`TempConvert`*-Dienst in der deployten WAR-Datei. Die Sicherheit schließt somit die beiden Operationen `f2c()` und `c2f()` des Dienstes ein. Dieser Pfad beinhaltet auch das WSDL-Dokument, dessen URL mit *tcauth?wsdl* endet.

- Der Zugriff auf Resourcen unter dem Pfad */tcauth* ist auf authentifizierte Benutzer der Rolle `bigshot` beschränkt. Will Fred zum Beispiel die Methode `f2c()` aufrufen, so muß er einen gültigen Benutzernamen mit Paßwort haben und autorisiert sein, die Rolle `bigshot` anzunehmen.

- Der HTTP-Authentifizierungstyp ist `BASIC` im Gegensatz zu den drei anderen Standardau-

thentifizierungsverfahren: `DIGEST`, `FORM` und `CLIENT-CERT`. Die Authentifizierungstypen werden in Kürze erklärt. Der Begriff „Autorisierung“ umfaßt hier sowohl die Authentifizierung als auch die Rollenautorisierung.

- Die Transportgarantie ist `CONFIDENTAL` und deckt somit die gegenseitige Authentifizierung, Verschlüsselung und Fälschungssicherheit ab. Ein Zugriffsversuch auf eine HTTP-URL wie *http://localhost:8080/tc/tcauth* wird von Tomcat an die HTTPS-URL *https://localhost:8443/tc/tcauth* umgeleitet. (Die Ziel-URL der Umleitung wird in *$TOMCAT_HOME/conf/server.xml* deklariert.)

Die Konfiguration des DDs gestattet bei Bedarf Einstellungen pro HTTP-Methode. Beispielsweise könnten starke Sicherheitseinstellungen für `POST`-Anfragen eingerichtet werden, während bei `GET`-Anfragen überhaupt keine Sicherheit notwendig ist.

[51] Der Webservice ist schnell implementiert:

```
package ch05.tcauth;

import javax.jws.WebService;

@WebService(endpointInterface = "ch05.tcauth.TempConvertAuth")
public class TempConvertAuthImpl implements TempConvertAuth {

    public float c2f(float t) { return 32.0f + (t * 9.0f / 5.0f); }
    public float f2c(float t) { return (5.0f / 9.0f) * (t - 32.0f); }

}
```

Der Client für diesen Dienst ist genauso schnell entwickelt:

```
import tcauthClient.TempConvertAuthImplService;
import tcauthClient.TempConvertAuth;
import javax.xml.ws.BindingProvider;

class ClientAuth {

    public static void main(String args[]) {

        TempConvertAuthImplService service = new TempConvertAuthImplService();
        TempConvertAuth port = service.getTempConvertAuthImplPort();
        BindingProvider prov = (BindingProvider) port;

        prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "fred");
        prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "rockbed");

        System.out.println("f2c(-40.1) ==> " + port.f2C(-40.1f));
        System.out.println("c2f(-40.1) ==> " + port.c2F(-40.1f));
        System.out.println("f2c(+98.7) ==> " + port.f2C(+98.7f));

    }

}
```

Es gibt einen kniffligen Aspekt im Zusammenhang mit dem Client, nämlich den Aufruf des `wsimport`-Kommandos, um die Artefakte im Package `tcauthClient` zu erzeugen. Das Problem besteht darin, daß das WSDL-Dokument des Dienstes ebenfalls gesichert ist, der Zugriff also Authentifizierung voraussetzt. Es gibt Workarounds. Eine Möglichkeit ist, das WSDL-Dokument lokal durch Aufrufen des `wsgen`-Kommandos auf der SIB zu erzeugen. Einee andere Möglichkeit ist, das WSDL-Dokument über eine ungesicherte Version des Dienstes zu beziehen. Anschließend werden das lokal gesicherte WSDL-Dokument und sein XML-Schema dem `wsimport`-Kommando übergeben, um die Artefakte zu generieren.

[52] Die Client-Applikation `ClientAuth` verwendet die *`BindingProvider`*-Kostanten als Schlüssel

für Benutzername und Paßwort. Tomcat erwartet die folgenden beiden Zeichenketten als Schlüssel zum Nachschlagen von Benutzername und Paßwort:

```
javax.xml.ws.security.auth.username
javax.xml.ws.security.auth.password
```

Dies sind die Werte der *BindingProvider*-Konstanten *USERNAME_PROPERTY* beziehungsweise *PASSWORD_PROPERTY*.

[53] Trifft eine Anfrage nach einer gesicherten Resource bei Tomcat ein, so „weiß“ Tomcat anhand des DDs aus der WAR-Datei, daß die Quelle der Anfrage authentifiziert und autorisiert werden muß. Tomcat prüft dann die übergebenen Berechtigungsnachweise anhand der Vergleichsdaten im voreingestellten MemoryRealm, welches die Benutzernamen, Paßwörter und Autorisierungsrollen aus der Datei *tomcat-users.xml* zur Verfügung hat. Hier die Datei *tomcat-users.xml* zu diesem Beispiel:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat" />
  <role rolename="bigshot" />
  <user username="tomcat" password="tomcat" roles="tomcat" />
  <user username="fred" password="rockbed" roles="bigshot"" />
</tomcat-users>
```

5.4.6 Verwendung eines Digests anstelle des Paßwortes

[54] Neben dem Authentifizierungstyp `BASIC` unterstützt HTTP 1.1 auch die Typen `DIGEST`, `FORM` und `CLIENT-CERT`. Der Authentifizierungstyp `FORM` ist für browserbasierte Webapplikationen am besten geeignet. Die Quintessenz besteht darin, daß die Webapplikation selbst und nicht der Webbrowser das Eingabeformular für Benutzername und Paßwort liefert. Der Authentifizierungstyp `CLIENT-CERT` authentifiziert den Client natürlich über ein digitales Zertifikat, stößt aber auf beträchtliche praktische Probleme. Stellen Sie zum Beispiel vor, daß ein Client ein gültiges Zertifikat bezüglich eines Hosts besitzt, aber unerwartet Zugriff auf den gesicherten Webauftritt auf einem anderen Host benötigt, der dieses Exemplar des Zertifikates nicht akzeptiert.

[55] Tomcat unterstützt auch den Authentifizierungstyp `DIGEST`. Ein vorsichtiger Benutzer wird, angesichts einer Kopie seines Paßwortes auf einem entfernten Host, in diesem Fall der Host auf dem der gesicherte Webservice läuft, verständlicherweise besorgt sein. Ist der entfernte Host bloß gestellt, so auch das Paßwort des Benutzers. Ein aus dem Paßwort generierter Digest vermeidet dieses Problem, solange eine Einweg-Hashfunktion zugrundeliegt, das heißt eine Funktion deren Werte mühelos berechnet werden können, die aber schwer umzukehren ist. Sind ein Digest und der Algorithmus zu seiner Berechnung gegeben (zum Beispiel MD5 oder SHA-1), so ist die Aufgabe, das ursprüngliche Paßwort aus dem Digest herzuleiten, rechnerisch sehr schwierig zu bewältigen.

[56] Das Umschalten von `BASIC` auf `DIGEST` ist nicht schwierig. Der Eintrag

```
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

im DD muß folgendermaßen geändert werden:

```
<login-config>
  <auth-method>DIGEST</auth-method>
</login-config>
```

Eine analoge Änderung ist in der Datei *tomcat-users.xml* eines gesicherten Webservice' mit dem MemoryRealm-Plugin erforderlich. Es gibt aber eine leichtere Vorgehensweise, die sich wie folgt zusammenfassen läßt:

- Tomcat bringt im Verzeichnis *$TOMCAT_HOME/bin* das `digest.sh`-Kommando mit, welches das Erzeugen von Digests mit Standardalgorithmen wie MD5 und SHA-1 gestattet. Der folgende Aufruf erzeugt einen Digest aus Freds Paßwort `rockbed`:

  ```
  % digest.sh -a SHA rockbed
  ```

  Das entsprechende Kommando für Windows heißt `digest.bat`. Das Kommando liefert einen 20-Byte langen Digest, hier:

  ```
  4b177c8995e6b0fa796581ac191f256545f0b8c5
  ```

- Der aus dem Paßwort generierte Digest ersetzt das Paßwort in der Datei *tomcat-users.xml*:

  ```
  <?xml version='1.0' encoding='utf-8'?>
  <tomcat-users>
    <role rolename="tomcat" />
    <role rolename="bigshot" />
    <user username="tomcat" password="tomcat" roles="tomcat" />
    <user username="fred"
      password="4b177c8995e6b0fa796581ac191f256545f0b8c5" roles="bigshot" />
  </tomcat-users>
  ```

- Ein Client wie Fred muß nun denselben Digest erzeugen. Tomcat stellt hierfür die Methode `RealmBase.Digest()` zur Verfügung, die auch das `digest.sh`-Kommando verwendet. Hier der überarbeitete Client:

  ```
  import tcauthClient.TempConvertAuthImplService;
  import tcauthClient.TempConvertAuth;
  import javax.xml.ws.BindingProvider;
  import org.apache.catalina.realm.RealmBase;

  // Revised to send a digested password.
  class ClientAuth {

      public static void main(String args[]) {

          TempConvertAuthImplService service = new TempConvertAuthImplService();
          TempConvertAuth port = service.getTempConvertAuthImplPort();
          BindingProvider prov = (BindingProvider) port;

          String digest = RealmBase.Digest("rockbed", // password
                                           "SHA",     // digest algorithm
                                           null);     // default char. encoding

          prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "fred");
          prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, digest);

          System.out.println("f2c(-40.1) ==> " + port.f2C(-40.1f));
          System.out.println("c2f(-40.1) ==> " + port.c2F(-40.1f));
          System.out.println("f2c(+98.7) ==> " + port.f2C(+98.7f));

      }

  }
  ```

  Der überarbeitete Client muß mit den beiden JAR-Bibliotheken *bin/tomcat-juli.jar* und *lib/catalina.jar* aufgerufen werden. Der Compiler begnügt sich mit *catalina.jar*.

Die Übertragung des Digest anstelle des eigentlichen Paßwortes erfordert nur wenig zusätzliche

Arbeit.

5.4.7 Ein sicherer Webservice-Provider

[57] Tomcat kann sowohl Webservices als auch Webservice-Provider deployen. Hier ist eine weitere Version des *TempConvert*-Dienstes, diesesmal im REST-Stil:

```
package ch05.authrest;

import javax.xml.ws.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.annotation.Resource;
import javax.xml.ws.BindingType;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPException;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.http.HTTPBinding;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.beans.XMLEncoder;
import java.util.List;
import java.util.ArrayList;

@WebServiceProvider
@BindingType(value = HTTPBinding.HTTP_BINDING)
public class TempConvertR implements Provider<Source> {
    @Resource
    protected WebServiceContext ws_ctx;

    public Source invoke(Source request) {
        // Grab the message context and extract the request verb.
        MessageContext msg_ctx = ws_ctx.getMessageContext();
        String http_verb = (String)
            msg_ctx.get(MessageContext.HTTP_REQUEST_METHOD);
        http_verb = http_verb.trim().toUpperCase();

        if (http_verb.equals("GET")) return doGet(msg_ctx);
        else throw new HTTPException(405); // bad verb exception
    }
    private Source doGet(MessageContext msg_ctx) {
        String query_string = (String) msg_ctx.get(MessageContext.QUERY_STRING);
        if (query_string == null) throw new HTTPException(400); // bad request

        String temp = get_value_from_qs("temp", query_string);
        if (temp == null) throw new HTTPException(400); // bad request

        List<String> converts = new ArrayList<String>();
        try {
            float f = Float.parseFloat(temp.trim());
            float f2c = f2c(f);
            float c2f = c2f(f);
            converts.add(f2c + "C");
            converts.add(c2f + "F");
        }
        catch (NumberFormatException e) { throw new HTTPException(400); }

        // Generate XML and return.
        ByteArrayInputStream stream = encode_to_stream(converts);
```

```
        return new StreamSource(stream);
    }

    private String get_value_from_qs(String key, String qs) {
        String[] parts = qs.split("=");
        if (!parts[0].equalsIgnoreCase(key))
            throw new HTTPException(400); // bad request
        return parts[1].trim();
    }

    private ByteArrayInputStream encode_to_stream(Object obj) {
       // Serialize object to XML and return
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        XMLEncoder enc = new XMLEncoder(stream);
        enc.writeObject(obj);
        enc.close();
        return new ByteArrayInputStream(stream.toByteArray());
    }
    private float c2f(float t) { return 32.0f + (t * 9.0f / 5.0f); }
    private float f2c(float t) { return (5.0f / 9.0f) * (t - 32.0f); }
}
```

Nichts im Quelltext weist darauf hin, daß Authentifizierung und Autorisierung im Spiel sind, da diese sicherheitsbezogenen Aufgaben wiederum an den Tomcat-Container delegiert wurden. Der Quelltext implementiert nur Applikationslogik. Die Deploymentdateien *web.xml* und *sun-jaxws.xml* werden in keiner bedeutenden Weise modifiziert.

[58] Der überarbeitete Client für den Webservice-Provider folgt demselben Ansatz wie der ursprüngliche Client für den Webservice, das heißt Benutzername und der aus dem Paßwort generierte Digest werden unter den von Tomcat erwarteten Schlüsseln in den Anfragekontext eingesetzt:

```
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.Dispatch;
import javax.xml.ws.http.HTTPBinding;
import org.xml.sax.InputSource;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathFactory;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.BindingProvider;
import org.apache.catalina.realm.RealmBase;

class DispatchClientTC {
    public static void main(String[] args) throws Exception {
        QName service_name = new QName("TempConvert");
        QName port_name = new QName("TempConvertPort");
        String endpoint = "https://localhost:8443/tempcR/authRest";

        // Now create a service proxy or dispatcher.
        Service service = Service.create(service_name);
        service.addPort(port_name, HTTPBinding.HTTP_BINDING, endpoint);
        Dispatch<Source> dispatch =
            service.createDispatch(port_name, Source.class, Service.Mode.PAYLOAD);

        String digest = RealmBase.Digest("rockbed", // password
                                         "SHA",     // digest algorithm
                                         null);     // default encoding
```

```
            dispatch.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "fred");
            dispatch.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, digest);
            dispatch.getRequestContext().put(MessageContext.HTTP_REQUEST_METHOD, "GET");
            dispatch.getRequestContext().put(MessageContext.QUERY_STRING, "temp=-40.1");

            StreamSource result = (StreamSource) dispatch.invoke(null);
            InputSource source = new InputSource(result.getInputStream());
            String expression = "//object";
            XPath xpath = XPathFactory.newInstance().newXPath();
            String list = xpath.evaluate(expression, source);
            System.out.println(list);
        }
    }
```

Die Ausgabe lautet:

```
-40.055557C
-40.18F
```

5.5 Web Services Security (WSS)

[59] WS-Security (WSS) ist eine Familie von Spezifikationen [(Abbildung ??)] mit dem Ziel, die Sicherheit auf der Übertragungsebene zu erweitern, in dem ein einheitliches, hinsichtlich des Transportprotokoll neutrales Punkt-zu-Punkt-Framework für höhere Sicherheitsanforderungen wie Authentifizierung und Autorisierung geschaffen wird.

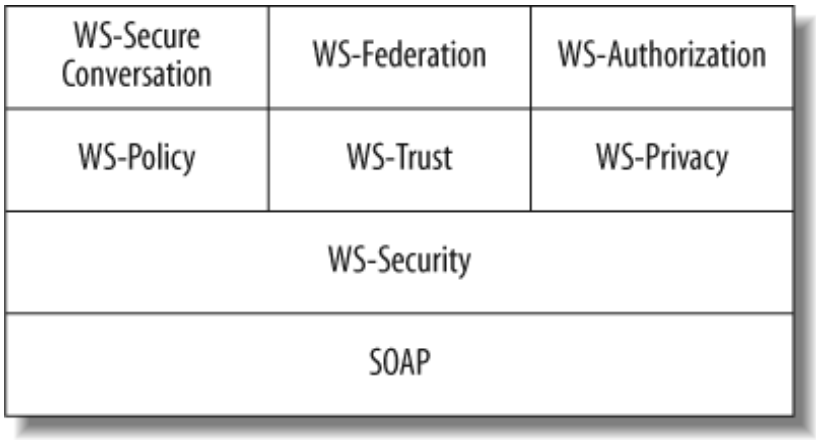

Abbildung 5.6: ...

[60] Die Blöcke über WS-Security in Abbildung **??** lassen wie folgt in Kürze erläutern. Die erste Schicht besteht aus WS-Policy, WS-Trust und WS-Privacy, die zweite Schicht aus WS-SecureConversation, WS-Federation und WS-Authorization und baut auf der ersten Schicht auf. Die Architektur ist somit modular, allerdings auch kompliziert. Es folgt eine kurze Beschreibung jeder Spezifikation, beginnend mit der ersten Lage:

- *WS-Policy*: Die Spezifikation beschreibt allgemeine Sicherheitskonzepte, -einschränkungen und -richtlinien. Eine Zusicherung im Sinne der WS-Policy könnte beispielsweise vorschreiben, daß eine Nachricht ein Sicherheitsmerkmal oder die Verwendung eines bestimmten Verschlüsselungsalgorithmus' verlangt.

- *WS-Trust*: Die Spezifikation beschreibt primär, wie Sicherheitsmerkmale herausgegeben, erneuert und geprüft werden können. Im Allgemeinen behandelt die Spezifikation broker trust relationships, die [im einem späteren Beispiel] veranschaulicht werden.
- *WS-Privacy*: Die Spezifikation erklärt, wie Dienste Datenschutzrichtlinien festlegen und erzwingen können. Ferner beschreibt die Spezifikation, wie ein Dienst feststellen kann, ob die Quelle einer Anfrage beabsichtigt, eine solche Richtlinie zu befolgen.
- *WS-SecureConversation*: Die Spezifikation beschreibt sichere Webservicekommunikation über verschiedene sites und somit über unterschiedliche Sicherheitskontexte und Trustdomains hinweg. Die Spezifikation konzentriert sich darauf, wie ein Sicherheitskontext erzeugt wird und wie Sicherheitsschlüssel erhalten und ausgetauscht werden.
- *WS-Federation*: Die Spezifikation widmet sich der Herausforderung der Verwaltung von Sicherheitsidentitäten über verschiedene Plattformen und Organisationen hinweg. Der Kern dieser Herausforderung besteht in der Frage, wie eine einzelne authentifizierte Identität (zum Beispiel Alice' Sicherheitsidentität) in einer heterogenen Sicherheitsumgebung gepflegt werden kann.
- *WS-Authorization*: Die Spezifikation deckt die Verwaltung von Autorisierunginformationen wie Sicherheitsmerkmalen und unterliegenden Richtlinien für die Zugriffserlaubnis auf gesicherte Resourcen ab.

WS-Security wird häufig einer Art von gebündelter Sicherheit/Verbundsicherheit (*federated security*) im weiteren Sinne zugerechnet, deren Ziel darin besteht, die Logik des Webservice' klar von höheren Sicherheitsangelegenheiten, vor allem Authentifizierung und Autorisierung, zu trennen, that challenge die Entwicklung von Webservices. Die Trennung der Dinge ist dazu gedacht, die Zusammenarbeit über Computersysteme und trust-Administrationsbereiche hinweng zu erleichtern.

[61] SOAP-basierte Webservices sind neutral im Hinblick auf das Transportprotokoll. Dementsprechend können sich SOAP-basierte Dienste nicht einfach auf die zuverlässigen Transportbedingungen unter HTTP und HTTPS verlassen, obwohl die meisten SOAP-Nachrichten über HTTP transportiert werden. Die Protokolle HTTP und HTTPS beruhen auf TCP/IP (Transmission Control Protocol/Internet Protocol), welches zuverlässigen Nachrichtenaustausch unterstützt. Die WS-ReliableMessaging-Spezifikation konzentriert sich auf die Aufgabe, SOAP-basierte Dienste über einer unzuverlässigen Infrastruktur zu betreiben.

[62] Ein SOAP-basierter Dienst kann sich nicht auf die Unterstützung von Authentifizierung und Autorisierung verlassen, die ein Webcontainer wie Tomcat oder ein Applikationsserver wie BEA WebLogic, JBoss, GlassFish oder WebSphere eventuell anbietet. Die WS-Security Spezifikationen beschreiben daher höhere Sicherheitsaspekte als Teil von SOAP selbst, statt als Teil der einem SOAP-basierten Webservice unterliegenden Infrastruktur. Die Ziele der WS-Security werden häufig zusammenfassend mit dem Begriff „Punkt-zu-Punkt-Sicherheit“ beschrieben, das heißt die Sicherheitsangelegenheiten nicht an die Übertragungsebene delegiert, sondern mittels einer entsprechenden Sicherheits-API direkt behandelt werden. Ein Framework für Punkt-zu-Punkt-Sicherheit muß die Situation berücksichtigen, daß eine Nachricht Mittler durchläuft, welche die Nachricht eventuell verarbeiten müssen, bevor sie den endgültigen Empfänger erreicht. Dementsprechend konzentiert sich die Punkt-zu-Punkt-Sicherheit auf den Inhalt der Nachricht und nicht auf the underlying transport.

5.5.1 Sichern eines Webservice mit WSS bei Deployment per Endpoint

[63] Dies ist der Quelltext eines einfachen Dienstes, der mit den Mitteln der WS-Security gesichert wird:

```
package ch05.wss;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.ws.WebServiceContext;
import javax.annotation.Resource;

@WebService
public class Echo {

    @Resource
    WebServiceContext ws_ctx;

    @WebMethod
    public String echo(String msg) {
        return "Echoing: " + msg;

    }

}
```

Nichts im Quelltext weist auf WS-Security hin. Die Publisher-Klasse liefert den ersten Hinweis:

```
package ch05.wss;

import javax.xml.ws.Endpoint;
import javax.xml.ws.Binding;
import javax.xml.ws.soap.SOAPBinding;
import java.util.List;
import java.util.LinkedList;
import javax.xml.ws.handler.Handler;

public class EchoPublisher {

    public static void main(String[] args) {

        Endpoint endpoint = Endpoint.create(new Echo());
        Binding binding = endpoint.getBinding();
        List<Handler> hchain = new LinkedList<Handler>();
        hchain.add(new EchoSecurityHandler());
        binding.setHandlerChain(hchain);
        endpoint.publish("http://localhost:7777/echo");
        System.out.println("http://localhost:7777/echo");

    }

}
```

Beachten Sie, daß es einen programmatisch hinzugefügten Behandler gibt. Hier ist der Quelltext der Behandlerklasse:

```
package ch05.wss;

import java.util.Set;
import java.util.HashSet;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import java.io.FileInputStream;
import java.io.File;
import com.sun.xml.wss.ProcessingContext;
import com.sun.xml.wss.SubjectAccessor;
import com.sun.xml.wss.XWSSProcessorFactory;
```

```
import com.sun.xml.wss.XWSSProcessor;
import com.sun.xml.wss.XWSSecurityException;

public class EchoSecurityHandler implements SOAPHandler<SOAPMessageContext> {

    private XWSSProcessor xwss_processor = null;
    private boolean trace_p;

    public EchoSecurityHandler() {
        XWSSProcessorFactory fact = null;
        try {
            fact = XWSSProcessorFactory.newInstance();
        }
        catch(XWSSecurityException e) { throw new RuntimeException(e); }

        FileInputStream config = null;
        try {
            config =  new FileInputStream(new File("META-INF/server.xml"));
            xwss_processor =
                fact.createProcessorForSecurityConfiguration(config, new Verifier());
            config.close();
        }
        catch (Exception e) { throw new RuntimeException(e); }
        trace_p = true; // set to true to enable message dumps
    }

    public Set<QName> getHeaders() {
        String uri = "http://docs.oasis-open.org/wss/2004/01/" +
            "oasis-200401-wss-wssecurity-secext-1.0.xsd";
        QName security_hdr = new QName(uri, "Security", "wsse");
        HashSet<QName> headers = new HashSet<QName>();
        headers.add(security_hdr);
        return headers;
    }

    public boolean handleMessage(SOAPMessageContext msg_ctx) {
        Boolean outbound_p = (Boolean)
            msg_ctx.get (MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        SOAPMessage msg = msg_ctx.getMessage();

        if (!outbound_p.booleanValue()) {
            // Validate the message.
            try {
              ProcessingContext p_ctx = xwss_processor.createProcessingContext(msg);
              p_ctx.setSOAPMessage(msg);
              SOAPMessage verified_msg = xwss_processor.verifyInboundMessage(p_ctx);
              msg_ctx.setMessage(verified_msg);

              System.out.println(SubjectAccessor.getRequesterSubject(p_ctx));
              if (trace_p) dump_msg("Incoming message:", verified_msg);
            }
            catch (XWSSecurityException e) { throw new RuntimeException(e); }
            catch(Exception e) { throw new RuntimeException(e); }
        }
        return true;
    }

    public boolean handleFault(SOAPMessageContext msg_ctx) { return true; }
    public void close(MessageContext msg_ctx) { }

    private void dump_msg(String msg, SOAPMessage soap_msg) {
        try {
```

```
                System.out.println(msg);
                soap_msg.writeTo(System.out);
                System.out.println();
            }
            catch(Exception e) { throw new RuntimeException(e); }
        }
    }
```

Zwei Abschnitt der Behandlerklasse `EchoSecurityHandler` sind von besonderem Interesse. Die erste Rückrufmethode, `getHeaders()`, wird von der Laufzeitbibliothek vor der Rückrufmethode `handleMessage()` aufgerufen. Die `getHeaders()`-Methode generiert einen `Security`-Header nach dem OASIS-Standard (Organization for the Advancement of Structured Information Standards), insbesondere nach den WSS-Standards (Web Services Security). Der Sicherheitsprozessor validiert den `Security`-Header.

[64] Der zweite interessante Abschnitt ist natürlich die Rückrufmethode `handleMessage()`, welche den größten Teil der Arbeit verrichtet. Die eintreffende SOAP-Nachricht (das heißt die Anfrage des Clients) wird durch Authenfizierung des Clients durch Benutzername und Paßwort verifiziert. Die Details folgen in Kürze. Verläuft die Verifizierung erfolgreich, so wird die verifizierte SOAP-Nachricht zur neuen Anfragenachricht. Scheitert die Verifizierung, so wird eine Ausnahme vom Typ `XWSSecurityException` als SOAP-Fehler ausgeworfen. Der entsprechende Abschnitt des Quelltextes ist:

```
try {
    ProcessingContext p_ctx =  xwss_processor.createProcessingContext(msg);
    p_ctx.setSOAPMessage(msg);
    SOAPMessage verified_msg = xwss_processor.verifyInboundMessage(p_ctx);
    msg_ctx.setMessage(verified_msg);

    System.out.println(SubjectAccessor.getRequesterSubject(p_ctx));
    if (trace_p) dump_msg("Incoming message:", verified_msg);
}
catch (XWSSecurityException e) { throw new RuntimeException(e); }
```

Bei erfolgreicher Verifizierung lautet die Ausgabe:

```
Subject:
   Principal: CN=fred
   Public Credential: fred
```

Das authentifizierte Subject ist Fred mit einem Principal, der eine spezifische Identität bezüglich der Benutzer/Rollen-Sicherheit ist. (Die Abkürzung `CN` steht für „Common Name“.) Freds Name ist der öffentliche Berechtigungsnachweis, während sein Paßwort geheim bleibt.

Inbetriebnahme eines WSS-basierten Dienstes per Endpoint.publish()

Webservices die von WS-Security Gebrauch machen, benötigen Packages, die gegenwärtig nicht mit der SE 6 ausgeliefert werden. Es ist einfacher, solche Webservices mit einer IDE wie NetBeans zu entwickeln und zu konfigurieren und den Dienst mit Hilfe eines Applikationsservers wie GlassFish zu deployen, dem Ansatz des nächsten Kapitels.

Dieser Abschnitt zeigt, daß ein WSS-basierter Dienst über die statische `Endpoint`-Methode `publish()` in Betrieb genommen werden kann. Hier die Schritte zur Konfiguration/Inbetriebnahme:

- Die JAR-Bibliothek *xws-security-3.0.jar* sollte heruntergeladen werden, da die darin enthaltenen Packages zur Zeit nicht Teil der SE 6 sind. Eine komfortable Adresse zum Herunterladen ist *http://fisheye5.cenqua.com/browse/xwss/repo/com.sun.xml.wss*. Die JAR-Bibliothek

kann der Einfachheit halber im Verzeichnis *$METRO_HOME/lib* deponiert werden.

- Beim Übersetzen und Aufrufen sollten zwei JAR-Bibliotheken im Klassenpfad stehen: *jaxws-tools.jar* und *xws-security-3.0.jar*.
- Die Konfigurationsdateien eines WSS-basierten Webservice' in einem Unterverzeichnis namens *WEB-INF* des Arbeitsverzeichnisses, das heißt des Verzeichnisses, in dem der Publisher des Dienstes und der Client aufgerufen werden. In diesem Fall ist das Arbeitsverzeichnis das Verzeichnis, welches das Unterverzeichnis *ch05* enthält. In diesem Beispiel werden zwei Konfigurationsdateien verwendet: *server.xml* und *client.xml*. Beide Dateien sind identisch, um das Beispiel so einfach wie möglich zu halten. Im produktiven Betrieb unterschieden sie sich selbstverständlich voneinander.

Nachdem der Webservice deployt ist, können per `wsimport`-Kommando die üblichen clientseitige Artefakte generiert werden. Die Metro-Laufzeitbibliothek schaltet sich automatisch ein, um die `wsgen`-Artefakte zu generieren, das heißt `wsgen` muß nicht manuell aufgerufen werden.

Das Beispiel zeigt die klare Trennung von Sicherheit und Applikationslogik. Alle Anweisungen im Zusammenhang mit WS-Security befinden sich in client- und serverseitigen Behandlerklassen.

---

[65] Die Behandlerklasse `EchoSecurityHandler` hat einen argumentlosen Konstruktor, der einen Sicherheitsprozessor nach den Informationen aus der Konfigurationsdatei erzeugt, hier *server.xml*. Die Implementierung des Konstruktors lautet:

```
public EchoSecurityHandler() {
    XWSSProcessorFactory fact = null;
    try {
        fact = XWSSProcessorFactory.newInstance();
    }
    catch(XWSSecurityException e) { throw new RuntimeException(e); }

    FileInputStream config = null;
    try {
        config =  new FileInputStream(new File("META-INF/server.xml"));
        xwss_processor =
        fact.createProcessorForSecurityConfiguration(config, new Verifier());
        config.close();
    }
    catch (Exception e) { throw new RuntimeException(e); }
    trace_p = true; // set to true to enable message dumps
}
```

Das `Verifier`-Objekt in der hervorgehobenen Zeile führt die eigentliche Prüfung durch. Die Konfigurationdatei ist sehr einfach:

```
<!-- Copyright 2004 Sun Microsystems, Inc. All rights reserved.
    SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms. -->
<xwss:SecurityConfiguration
    xmlns:xwss="http://java.sun.com/xml/ns/xwss/config"
    dumpMessages="true" >
  <xwss:RequireUsernameToken passwordDigestRequired="false" />
</xwss:SecurityConfiguration>
```

und wird für gewöhnlich im Verzeichnis *META-INF* deponiert. Bei diesem Beispiel schreibt die Konfigurationsdatei vor, daß Nachrichten an und vom Webservice zur Überprüfung ausgegeben werden sollen und daß das Paßwort anstelle des daraus generierten Digest in der Anfragenachricht akzeptiert

werden soll. Das `Verifier`-Objekt kümmert sich um die systemnahen Details der Authentifizierung der Anfrage durch Prüfung des Benutzernamens und Paßwortes:

```
package ch05.wss;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import com.sun.xml.wss.impl.callback.PasswordCallback;
import com.sun.xml.wss.impl.callback.PasswordValidationCallback;
import com.sun.xml.wss.impl.callback.UsernameCallback;

// Verifier handles service-side callbacks for password validation.
public class Verifier implements CallbackHandler {
    // Username/password hard-coded for simplicity and clarity.
    private static final String _username = "fred";
    private static final String _password = "rockbed";

    // For password validation, set the validator to the inner class below.
    public void handle(Callback[] callbacks) throws UnsupportedCallbackException
{
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof PasswordValidationCallback) {
               PasswordValidationCallback cb=(PasswordValidationCallback)callbacks[i];
               if (cb.getRequest() instanceof
                   PasswordValidationCallback.PlainTextPasswordRequest)
                   cb.setValidator(new PlainTextPasswordVerifier());
            }
            else
               throw new UnsupportedCallbackException(null, "Not needed");
        }
    }

    // Encapsulated validate method verifies the username/password.
    private class PlainTextPasswordVerifier
                      implements PasswordValidationCallback.PasswordValidator {
        public boolean validate(PasswordValidationCallback.Request req)
            throws PasswordValidationCallback.PasswordValidationException {

            PasswordValidationCallback.PlainTextPasswordRequest plain_pwd =
                (PasswordValidationCallback.PlainTextPasswordRequest) req;
            if (_username.equals(plain_pwd.getUsername()) &&
                _password.equals(plain_pwd.getPassword())) {
                return true;
            }
            else
                return false;
        }
    }
}
```

Die Authentifizierung ist erfolgreich mit dem Benutzernamen `fred` und dem Paßwort `rockbed` und scheitert in allen anderen Fällen. Der im Konstruktor der Klasse `EchoSecurityHandler` erzeugte Sicherheitsprozessor ist dafür zuständig, die `handle()`-Methode des `Verifier`-Objektes aufzurufen. Benutzername und Paßwort sind hartkodiert, um das Beispiel einfach zu halten und werden nicht aus einer Datenbank abgefragt. Das Paßwort ist ferner im Klartext angegeben.

[66] Eine clientseitige Implementierung des Interface' ***`SOAPHandler`*** generiert die vom ***`Echo`***-Dienst erwarteten WSS-Artefakte:

```
package ch05.wss;

import java.util.Set;
import java.util.HashSet;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import java.io.FileInputStream;
import java.io.File;
import com.sun.xml.wss.ProcessingContext;
import com.sun.xml.wss.SubjectAccessor;
import com.sun.xml.wss.XWSSProcessorFactory;
import com.sun.xml.wss.XWSSProcessor;
import com.sun.xml.wss.XWSSecurityException;

public class ClientHandler implements SOAPHandler<SOAPMessageContext> {
    private XWSSProcessor xwss_processor;
    private boolean trace_p;

    public ClientHandler() {
        XWSSProcessorFactory fact = null;
        try {
            fact = XWSSProcessorFactory.newInstance();
        }
        catch(XWSSecurityException e) { throw new RuntimeException(e); }

        // Read client configuration file and configure security.
        try {
            FileInputStream config =
                new FileInputStream(new File("META-INF/client.xml"));
            xwss_processor =
                fact.createProcessorForSecurityConfiguration(config, new Prompter());
            config.close();
        }
        catch (Exception e) { throw new RuntimeException(e); }
        trace_p = true; // set to true to enable message dumps
    }

    // Add a security header block
    public Set<QName> getHeaders() {
        String uri = "http://docs.oasis-open.org/wss/2004/01/" +
            "oasis-200401-wss-wssecurity-secext-1.0.xsd";
        QName security_hdr = new QName(uri, "Security", "wsse");
        HashSet<QName> headers = new HashSet<QName>();
        headers.add(security_hdr);
        return headers;
    }

    public boolean handleMessage(SOAPMessageContext msg_ctx) {
        Boolean outbound_p = (Boolean)
            msg_ctx.get (MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        SOAPMessage msg = msg_ctx.getMessage();

        if (outbound_p.booleanValue()) {
            // Create a message that can be validated.
            ProcessingContext p_ctx = null;
            try {
                p_ctx = xwss_processor.createProcessingContext(msg);
                p_ctx.setSOAPMessage(msg);
```

```
                SOAPMessage secure_msg = xwss_processor.secureOutboundMessage(p_ctx);
                msg_ctx.setMessage(secure_msg);

                if (trace_p) dump_msg("Outgoing message:", secure_msg);
            }
            catch (XWSSecurityException e) { throw new RuntimeException(e); }
        }
        return true;
    }

    public boolean handleFault(SOAPMessageContext msg_ctx) { return true; }
    public void close(MessageContext msg_ctx) { }

    private void dump_msg(String msg, SOAPMessage soap_msg) {
        try {
            System.out.println(msg);
            soap_msg.writeTo(System.out);
            System.out.println();
        }
        catch(Exception e) { throw new RuntimeException(e); }
    }
}
```

5.5.2 Die Klassen Prompter und Verifier

[67] Auf der Seite des Dienstes verwendet der Sicherheitsprozessor ein `Verifier`-Objekt, um die Anfrage zu validieren. Auf der Seite des Clients verwendet der Sicherheitsprozessor ein `Prompter`-Objekt, um Benutzername und Paßwort abzufragen, welche anschließend in die ausgehende SOAP-Nachricht eingesetzt werden. Der Sicherheitsprozessor auf der Seite des Dienstes generiert eine geprüfte SOAP-Nachricht, wenn die Authentifizierung erfolgreich verläuft. Der Sicherheitsprozessor auf der Seite des Clients erzeugt eine gesicherte SOAP-Nachricht, wenn das `Prompter`-Objekt korrekt arbeitet. Der Quelltext der Klasse `Prompter` lautet:

```
package ch05.wss;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import com.sun.xml.wss.impl.callback.PasswordCallback;
import com.sun.xml.wss.impl.callback.PasswordValidationCallback;
import com.sun.xml.wss.impl.callback.UsernameCallback;
import java.io.BufferedReader;
import java.io.InputStreamReader;

// Prompter handles client-side callbacks, in this case
// to prompt for and read username/password.
public class Prompter implements CallbackHandler {
    // Read username or password from standard input.
    private String readLine() {
        String line = null;
        try {
            line = new BufferedReader(new InputStreamReader(System.in)).readLine();
        }
        catch(IOException e) { }
        return line;
    }
```

```
        // Prompt for and read the username and the password.
        public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
            for (int i = 0; i < callbacks.length; i++) {
                if (callbacks[i] instanceof UsernameCallback) {
                    UsernameCallback cb = (UsernameCallback) callbacks[i];
                    System.out.print("Username: ");
                    String username = readLine();
                    if (username != null) cb.setUsername(username);
                }
                else if (callbacks[i] instanceof PasswordCallback) {
                    PasswordCallback cb = (PasswordCallback) callbacks[i];
                    System.out.print("Password: ");
                    String password = readLine();
                    if (password != null) cb.setPassword(password);
                }
            }
        }
    }
```

Der Sicherheitsprozessor interagiert mit dem `Prompter`-Objekt über zwei Objekte vom Typ `PasswordCallback` beziehungsweise `UsernameCallback`, welche Benutzername und Paßwort des Clients abfragen, lesen und speichern.

5.5.3 Der sichere SOAP-Envelope

[68] Der clientseitige Sicherheitsprozessor generiert eine SOAP-Nachricht mit allen WSS-Informationen im SOAP-Header. Der SOAP-Body ist von einem ebensolchen in einer ungesicherten SOAP-Nachricht nicht zu unterscheiden. Der Client sendet die folgende SOAP-Nachricht:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
    <wsse:Security
        xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
        oasis-200401-wss-wssecurity-secext-1.0.xsd"
        S:mustUnderstand="1">
      <wsse:UsernameToken
          xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
          oasis-200401-wss-wssecurity-utility-1.0.xsd"
          wsu:Id="XWSSGID-1216851209528949783553">
        <wsse:Username>fred</wsse:Username>
        <wsse:Password
            Type="http://docs.oasis-open.org/wss/2004/01/
                oasis-200401-wss-username-token-profile-1.0#PasswordText">
           rockbed
        </wsse:Password>
        <wsse:Nonce
            EncodingType="http://docs.oasis-open.org/wss/2004/01/
                oasis-200401-wss-soap-message-security-1.0#Base64Binary">
          Vyg90XUn/rl2F4m6lSFIZCoU
        </wsse:Nonce>
        <wsu:Created>2008-07-23T22:13:33.001Z</wsu:Created>
      </wsse:UsernameToken>
    </wsse:Security>
  </S:Header>
  <S:Body>
```

```
      <ns2:echo xmlns:ns2="http://wss.ch05/">
        <arg0>Hello, world!</arg0>
      </ns2:echo>
    </S:Body>
  </S:Envelope>
```

Der SOAP-Header beinhaltet zusätzlich zu Benutzername und Paßwort eine sogenannte Nonce,[1] ein zufällig gewähltes, statistisch eindeutiges Merkmal, welches in die Nachricht eingesetzt wird. Die Nonce wird zum Schutz gegen Diebstahl der Nachricht und Replay-Angriffe hinzugefügt, wobei ein ungesicherter Berechtigungsnachweis, etwa ein Paßwort, erneut übertragen wird, um eine unautorisierte Operation aufrufen zu können. Fängt beispielsweise Eve Alice' Paßwort aus einer SOAP-Nachricht ab, die Geld von Alice' Konto auf ein anderes Konto überweist, so könnte Eve die Überweisung zu einem späteren Zeitpunkt wiederholen und mit Hilfe des gestohlenen Paßwortes Alice' Geld auf Eves Konto überweisen. Verlangt der Empfänger der Nachricht nicht nur die üblichen Berechtigungsnachweise wie Benutzername und Paßwort, sondern auch eine Nonce mit bestimmten geheimen Attributen, so würde Eve mehr Informationen als nur das gestohlende Paßwort brauchen. Eve müßte auch die Nonce replizieren, was rechnerisch sehr schwierig zu bewältigen ist.

[69] Die SOAP-Nachricht des *Echo*-Dienstes an den Client hat keinerlei WSS-Artefakte:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header />
  <S:Body>
    <ns2:echoResponse xmlns:ns2="http://wss.ch05/">
      <return>Echoing: Hello, world!</return>
    </ns2:echoResponse>
  </S:Body>
</S:Envelope>
```

5.5.4 Zusammenfassung des WSS-Beispiels

[70] Das erste Beispiel für WS-Security stellt die API vor, hat aber den offensichtlichen Nachteil, daß Benutzername und Paßwort des Clients zusammen mit der Nonce über eine unsichere Verbindung übertragen werden. Der Grund besteht darin, daß der `Endpoint`-Publisher keine HTTPS-Verbindungen unterstützt. Ein schnelle Lösung ist der zuvor gezeigte HTTPS-Server.

[71] Das Beispiel erfüllt dennoch seine Aufgabe, zu veranschaulichen, auf welche Weise WS-Security Authentifizierung und Autorisierung unterstützt, ohne die Unterstützung durch ein Transportprotokoll wie HTTP oder einen Container wie Tomcat in Anspruch zu nehmen. Das Beispiel zeigt ferner, daß WS-Security eine klare Trennung der Sicherheits- von der Webservicelogik unterstützt. Das nächste Kapitel dringt tiefer in die Details der WS-Security ein und liefert ein Beispiel auf Produktionsniveau.

5.6 Ausblick

[72] Bis jetzt wurden alle Beispiele für Webservices entweder über die statische `Endpoint`-Methode `publish()`, einen HTTPS-Server oder Tomcat deployt. Diese Vorgehensweise ohne viel Aufhebens hat den Vorzug, daß der Fokus auf der Webservice-API von Java beruht. Im produktiven Betrieb wäre ein leichtgewichtiger Webcontainer wie Tomcat oder ein schwergewichtiger Applikationsserver

[1] Anmerkung des Übersetzers: In der Kryptographie bezeichnet der Begriff „Nonce“ (Abkürzung für: „used only once“ oder „number used once“) eine Zahlen- oder Buchstabenkombinationen, die ad hoc gewählt und nach einmaliger Verwendung verworfen wird. Aus *http://de.wikipedia.org*, Artikel „Nonce“.

wie BEA Weblogic, GlassFish, JBoss oder WebSphere die wahrscheinliche Wahl zum Deployment SOAP-basierter Webservices oder Dienste im REST-Stil. Das nächste Kapitel betrachtet den Applikationsserver GlassFish, der Metro einschließt und die Referenzimplementierung eines Containers für Java-Webservices ist.

Kapitel 6

Java-Webservices in Java-Applikationsservern

Inhaltsübersicht

6.1 Die Komponenten eines Java-Applikationsservers

[0] In den vorangegangenen Kapiteln wurden Webservices, gleichsam SOAP-basiert oder im REST-Stil, mit Hilfe der statischen `Endpoint`-Methode `publish()` oder des Webcontainers Tomcat in Betrieb genommen. Dieses Kapitel zeigt das Deployment von Webservices mit Hilfe eines Java-Applikationsservers (JAS), der tragenden Säule der Java Enterprise Edition. Die aktuelle Version ist EE 5 und beinhaltet EJB 3.0. Wir beginnen mit einer Übersicht über die in einem JAS gebündelten Softwarekomponenten:

- *Webcontainer*: Ein Webcontainer dient zum Deployment von Servlets und Webservices. Eine traditionelle Java-Webapplikation ist ein Konglomerat aus statischen HTML-Seiten, Servlets, Servleterzeugern wie JSP-Seiten (JavaServer Pages) und ~~JSF-Skripten~~ (JavaServer Faces), unterstützende JavaBeans für JSP-Seiten und ~~JSF-Skripte~~ und Hilfsklassen. Die *Referenzimplementierung* für Container ist *Tomcat*. Tomcat kann, wie andere Webcontainer, in einen Applikationserver eingebettet werden. Webkomponenten werden in Form von WAR-Dateien im Webcontainer deployt, die typischerweise die Standardkonfigurationsdatei *web.xml* (DD) und eventuell anbieterspezifische Konfigurationsdateien wie zum Beispiel *sun-jaxws.xml* enthalten. Der Webcontainer verläßt sich auf einen Servlet-Interzeptor (bei Tomcat ein Objekt der Klasse `WSServlet`), der zwischen dem Client und der SIB des Dienstes mittelt.

- *Nachrichtenorientierte Middleware*: Die nachrichtenorientierte Middleware unterstützt den JMS (Java Message Service), der die Teilstreckenverfahren unter dem Begriff *Nachrichtenübertragung* (*messaging*) zusammenfaßt. JMS unterstützt sowohl synchrone als auch asynchrone Methoden sowie zwei Ablageverfahren für Nachrichten: Zum einen, eine Art Forum/schwarzes Brett, wobei eine Leseoperation nicht automatisch zum Löschen der gesendeten Nachricht führt. Zum anderen, eine Warteschlange (*FIFO*-Liste, First In, First Out), wobei eine Leseoperation per Voreinstellung das Löschen der gelesenen Nachricht bewirkt. Beim JMS veröffentlicht ein *Publisher* Nachrichten zu einem Thema und ein Sender überträgt die Nachricht in eine Warteschlange. Ein *Abonnent/Bezieher* des Themas oder ein *Empfänger* an der Warteschlange nimmt solche Nachrichten entweder synchron durch eine blockierende Leseoperation oder asynchron durch den JMS-Benachrichtigungsmechanismus entgegen. JMS-Themen implementieren das Publisher/Subscriber-Nachrichtenaustauschmuster, JMS-Queues dagegen das Punkt-zu-Punkt-Modell.

- *Enterprise JavaBean (EJB) Container*: Ein EJB-Container enthält ~~EJB-Instanzen~~, die einem von drei Typen angehören: Sitzungs-, Entitäts- oder ~~message-driven~~-EJB. Sitzungs- und traditionelle EJBs beruhen auf *RMI (Remote Method Invocation)*, während ~~message-driven~~ EJBs auf JMS aufbauen.

  Die ~~message-driven~~ EJB ist eine Implementierung des Interface' `MessageListener` als EJB. Ein Ereignisbehandler erhält eine Benachrichtigung, wenn eine neue Nachricht in einem Forum oder einer Warteschlange eingeht, bei welcher der Behandler registriert ist.

  Eine Sitzungs-EJB implementiert typischerweise die Geschäftslogik einer Enterprise-Applikation und arbeitet je nach Bedarf mit anderen Komponenten zusammen, entweder lokalen (zum Beispiel anderen EJBs im selben Container) oder entfernten Komponenten (zum Beispiel Clients auf einem anderen Host. Wie der Name andeutet, dient eine Sitzungs-EJB entwurfsbedingt dem Unterhalt einer Clientsitzung. Eine Sitzungs-EJB ist entweder zustandslos oder zustandsbehaftet. Eine zustandslose Sitzungs-EJB ist effektiv eine Anzahl paarweise voneinander unabhängiger Objektmethoden, welche nur auf den als Argument übergebenen Daten operieren sollten. Der EJB-Container setzt voraus, daß eine zustandslose Sitzungs-EJB keine Zustandsinformationen in Objektfeldern vorhält. Stellen Sie zum Beispiel eine Sitzungs-EJB vor, die zwei Methoden `m1()` und `m2()` kapselt. Würde diese EJB als zustandlos deployt, so würde der EJB-Container annehmen, daß ein Client `C` die `m1()`-Methode einer EJB-Instanz und die `m2()`-Methode einer anderen EJB-Instanz aufrufen könnte, da beide Methoden auf keinen gemeinsamen Zustand zugreifen. Würde dieselbe EJB dagegen als zustandbehaftet deployt, so müßte der EJB-Container gewährleisten, daß die Aufrufe der Methoden `m1()` und `m2()` durch den Client `C` auf derselben EJB-Instanz ausgeführt werden, da beide Methoden voraussichtlich gemeinsame Zustandsinformationen teilen. Wie diese Zusammenfassung schließen läßt, verwaltet ein EJB-Container automatisch einen Vorrat von EJB-Instanzen aller EJB-Typen. Ein SOAP-basierter Webservice kann durch Markieren mit der Annotation `@Stateless` als

zustandslose Sitzungs-EJB implementiert werden.

Vor der Version 5 der Java Enterprise Edition waren Entitäts-EJBs die bevorzugte Lösung, um eine Enterprise-Applikation mit einem Zwischenspeicher für Datenbankobjekte wie beispielsweise Tabellenzeilen auszustatten. Entitäts-EJBs waren die Persistenzkonstruktion, die ORM-Fähigkeiten (Object-Relational Mapping) für Applikationen verfügbar machte. Eine traditionelle Entitäts-EJB konnte entweder über BMP (Bean Managed Persistence) oder per CMP (Container-Managed Persistence) deployt werden. Die Frage war, ob der Programmierer oder der EJB-Container für die Kohärenz („Stimmigkeit") zwischen Datenquelle (zum Beispiel einer Tabellezeile) und EJB-Instanz verantwort sei. Bei den ersten EJB-Containern konnte das Argument vorgebracht werden, BMP sei effizienter. Die EJB-Container wurden schnell bis an den Punkt verbessert, an dem die Entscheidung für CMP offensichtlich war. Tatsächlich zeichnete sich CMP als Hauptansporn für und Nutzen durch den Einsatz traditioneller Sitzungs-EJBs ab. Eine über CMP deployte EJB mußte auch sich auch per CMT (Container Managed Transactions) deployen lassen.

Vor EJB 3.0 waren EJBs im Allgemeinen und Entitäts-EJBs im Besonderen schwer zu programmieren und zu konfigurieren. Die Entwicklung war knifflig und die Konfiguration erforderte ein langes, kompliziertes XML-Dokument, liebevoll als DD (Deploymentdeskriptor) bezeichnet. All dies änderte sich mit der EE 5, welche die Einsatzmöglichkeiten der ursprünglichen Entitäts-EJB auf POJO-Klassen mit der Annotation `@Entity` ausweitete. Durch die `@Entity`-Annotation kamen Java-Programmierer in den Genuß, die Vorteile der traditionellen Entitäts-EJB zu nutzen, ohne sich mit der Konfiguration und Programmierung dieser Art von EJB auseinandersetzen zu müssen. Die Annotation `@Entity` liegt im Zentrum der *JPA* (Java Persistence API), welche Eigenschaften und Fähigkeiten verwandter Technologien wie Hibernate, Oracles TopLink, Java Data Objects (JDO) und traditionelle Entitäts-EJBs integriert. The `@Entity` ist heute die bevorzugte Vorgehensweise im Umgang mit Persistenz. An `@Entity` kann im Gegensatz zur Entitäts-EJB sowohl in der Java Standard Edition, als auch in der Java Enterprise Edition verwendet werden.

EJBs sind im Gegensatz zu Servlets threadsicher, da der EJB-Container für die Threadsynchronisierung verantwortlich ist. (Wie bei Servlets wird jede Anfrage an eine EJB mit Hilfe eines separaten Threads verarbeitet.) EJBs eignen sich selbst bei traditionellen browserbasierten Webapplikationen als Unterstützung für Servlets. Ein Servlet könnte beispielsweise eine Anfrage an eine Sitzungs-EJB veranlassen, die wiederum Objekte verschiedener mit `@Entity` annotierter Klassen als persistente Datenquellen nutzt ([Abbildung ??]).

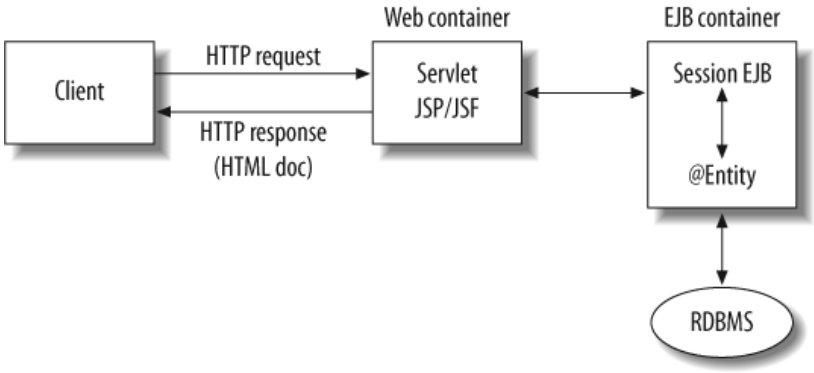

Abbildung 6.1: ...

- *JNDI-Anbieter*: Ein JNDI-Anbieter (Java Naming and Directory Interface) unterhält wenigstens einen Namensdienst, der Namen (zum Beispiel den deployten Namen einer EJB oder eine

Nachrichtenwarteschlange) an Entitäten (in diesem Fall die deployte EJB oder Warteschlange) bindet. Unterstützt der JNDI-Anbieter auch Verzeichnisdienste, ist der Name einer Entität an die Entität und ihre Attribute gebunden. In the simple case pflegt ein JNDI-Anbieter eine hierarchische Datenbank von Name/Entität-Paaren. Jede in einen Java-Applikationsserver gestellte Komponente wird automatisch beim Namensdienst registriert und steht potentiellen Client anschließend zum Nachschlagen zur Verfügung. In der Version 5 der Java Enterprise Edition ist der JNDI-Anbieter weitestgehend unsichtbare Infrastruktur.

- *Sicherheitsanbieter*: Der Sicherheitsanbieter bietet high-level Sicherheitsdienste für die in den Containern deployten Komponenten an, darunter natürlich auch Authentifizierung und Autorisierung. Der Sicherheitsanbieter muß JAAS-kompatibel (Java Authentication and Authorization Service) sein. Der Sicherheitsanbieter hat heute typischerweise Plugins für weitere Anbieter wie *LDAP* (Lightweight Directory Access Protocol). Die Sicherheit ist gewöhnlich im Applikationsserver integriert. Die vom Webcontainer angebotene containergesteuerte Sicherheit ist beispielsweise in die voreingestellte JAAS-Sicherheit integriert, die der EJB-Container anbietet.

- *Relational Database Management System*: Ein Applikationsserver beinhaltet in der Regel ein RDBMS (Relational Database Management System), welches als dauerhafter Speicher für Objekte mit `@Entity` annotierter Klassen voreingestellt ist und programmatisch oder direkt von anderen Komponenten wie Webservices und Servlets erreicht werden kann. Ein moderner Applikationsserver gestattet das mühelose Anschließen eines anderen Datenbanksystems, wenn das angebotene System nicht das bevorzugte ist. Der in diesem Kapitel vorgestellte Applikationsserver GlassFish wird mit der Java-DB ausgeliefert, der von Sun unterstützten Version des Apache Derby RDBMS.

- client container: Der Clientcontainer besteht aus Softwarebibliotheken, die ein Client benötigt, um mit deployten Komponenten wie Nachrichtenthemen oder EJBs zu interagieren und Dienste wie JNDI und Sicherheit zu nutzen.

[1] Auf der Web-Ebene unterstützt ein Java Applikationsserver sowohl traditionelle interaktive Webauftritte als auch Webservices. Die *Model-View-Controller* (*MVC*) Architekur ist bei modernen interaktiven Webauftritten gängig. Die Modellkomponente (*model*) pflegt die statischen Informationen des Auftritts und ist für dauerhafte Speicherung zuständig. Die Präsentationskomponente (*view*) liefert eine entsprechende Darstellung der Modellkomponente. Die Steuerungskomponente (*controller*) ist ein request endpoint, der die Geschäftslogik implementiert, welche Modell- und Präsentationskomponente koordiniert. In der Enterprise Edition und selbst in der Standard Edition ist eine mit `@Entity` annotierte Klasse ein natürlicher Weg, um eine Modellkomponente mit dauerhafter Speicherung zu implementieren. Bei interaktiven Webapplikationen können JSP-Seiten oder JSF-Skripte eine HTML-Ansicht einer Modellkomponente erzeugen und entweder ein Servlet oder eine Sitzungs-EJB ist eine natürliche Vorgehensweise, um eine Steuerungskomponente zu implementieren. Bei Java-basierten in einem Applikationserver wie GlassFish deployten Webservices sind `@Entity`-Klassen ebenfalls ein natürlicher Weg, um Modellkomponenten zu implementieren. Der Webservice ist die Steuerungskomponente, welche die Geschäftslogik in mit `@WebMethod` annotierten Methoden darlegt und erforderlichenfalls mit Modellkomponenten interagiert. Der Webservice kann als WAR-Datei deployt werden, was auf ein servletbasiertes Deployment hinausläuft, oder als EAR-Datei (Enterprise Archive), das heißt als EJB-basiertes Deployment. Die Unterscheidung zwischen servletbasiertem und EJB-basiertem Deployment wird [später (Seite)] nochmals diskutiert.

[2] Infolge der Bündelung so vieler Eigenschaften, Fähigkeiten und Dienste ist ein Applikationsserver in natürlicher Weise ein kompliziertes Stück Software. Eine unter Java-Programmierern diskutierte Frage lautet, ob der Nutzen eines Applikationsservers die Komplexität seiner Verwendung aufwiegt. Diese Komplexität ist zu einem großen Teil die Folge der unzähligen APIs, welche in ei-

nem Applikationsserver zusammenspielen. Eine Applikation, die Servlets, JSP-Seiten, JSF-Skripte, Nachrichtübermittelung und EJBs beinhaltet, bedient alleine fünf verschiedene APIs. Die Integration von Webkomponenten wie Servlets und EJBs ist nach wie vor keine triviale Aufgabe. Diese Sachlage trägt zu den jüngeren Anstrengungen der Anbieter bei, eine nahtlose Integration von EE-Komponenten anzubieten, was aller Voraussicht nach zu einem leichter gewichtigen, programmiererfreundlicheren Framework für Applikation im Rahmen der EE führen würde. JBoss Seam ist ein Beispiel für diese Bemühungen. Dennoch soll darauf hingewiesen werden, daß die Version 5 der Java Enterprise Edition erheblich leichter zu benutzen ist, als ihr Vorgänger, die Java Enterprise Edition 1.4. Die EE bewegt sich definitiv in der Schneise, welche leichtere Frameworks wie Spring schlagen.

[3] Der Applikationsserver *GlassFish* ist die quelloffene *Referenzimplementierung*. (Genauer ist ein bestimmter Snapshot von GlassFish die Referenzimplementiertung.) Die aktuelle Produktivversion kann unter der Adresse *https://glassfish.dev.java.net* heruntergeladen werden. Diese Version ist auch im Download der NetBeans-IDE enthalten, siehe *http://www.netbeans.org*. Die Beispiele in diesem Kapitel werden mittels GlassFish deployt. Einige Beispiele werden mit Hilfe von NetBeans entwickelt oder nachbearbeitet. Keines der Beispiel verlangt allerdings eine bestimmte IDE, inbesondere also kein NetBeans.

Grundlagen zum Gebrauch des GlassFish-Applikationsservers

Die Umgebungsvariable `$AS_HOME` verweise auf das Installationsverzeichnis von GlassFish. Das Unterverzeichnis *bin* enthält ein Shellskript namens `asadmin`, welches zum Starten des GlassFish Applikationservers dient:

```
% asadmin start-domain domain1
```

Für Windows verwenden Sie das Skript `asadmin.bat`. Der Applikationsserver gibt beim Starten die Ports aus, an denen er auf Anfragen wartet. Beispielsweise können die Ports 8080 oder 8081 als HTTP-Ports angegeben sein. Um zu bestätigen, daß GlassFish gestartet ist, öffnen Sie ein Browserfenster mit der URL *http://localhost:8080*, vorausgesetzt, daß 8080 in der Liste der HTTP-Ports vorkommt. Die Begrüßungsseite mit Verweisen auf die GlassFish-Dokumentation und ähnliches sollte angezeigt werden. Das Kommando:

```
% asadmin stop-domain domain1
```

beendet das Deployment von `domain1`, während das Kommando

```
% asadmin stop-appserv
```

den gesamten Applikationsserver anhält.

Die URL *http://localhost:4848* führt zur Administratorkonsole mit zahlreichen Hilfsmitteln zur Beobachtung und Verwaltung deployter Komponenten, darunter Webservices. Der Standardbenutzername für die Administratorkonsole ist `admin`, das Standardpaßwort `adminadmin`. Die Administratorkonsole ist hilfreich beim Testen neu deployter Webservices. Zu jeder mit `@WebMethod` annotierten Methode, die über HTTP aufgerufen werden kann, erzeugt GlassFish eine interaktive HTML-Seite über welche die Methode mit Argumenten getestet werden kann. SOAP-Anfrage- und Antwortnachrichten werden angezeigt. Das WSDL- und sonstige XML-Dokumente stehen ebenfalls zur Einsichtnahme zur Verfügung.

Das Deployment eines Webservice' ist ein Kinderspiel. Angenommen ein Webservice mit der SIB-Klasse `Hi` sei in einer WAR-Datei namens *hello.war* verpackt. (Die Verpackungsdetails sind wie bei Tomcat, werden aber anläßlich des ersten Beispiels auf [Seite ??] nochmals zusammengestellt.) Um

die Applikation zu deployen, kopieren Sie die WAR-Datei in das Verzeichnis *$AS_HOME/domains-/domain1/autodeploy*. Verläuft das Deployment erfolgreich, so erscheint eine zweite Datei namens *hello.war_deployed* in a second or so in the autodeploy directory. Scheitert das Deployment, so erscheint statt dessen die leere Datei *hello.war_deployFailed*. Die Protokolldatei *$AS_HOME/domains/domain1/logs/server.log* erteilt Auskunft über die Details des gescheiterten Versuchs. Bei erfolgreichem Deployment kann das WSDL-Dokument unter der Adresse *http://localhost:8080/hello/HiService?wsdl* im Browserfenster betrachtet werden. Der Standardname des Dienstes, `HiService`, befolgt die JAX-WS-Konvention, die Endung `Service` an den Namen der SIB (hier `Hi`) anzufügen.

Das Java-DB Datenbanksystem läßt sich mit dem folgenden Kommando starten:

```
% asadmin start-database
```

Das Kommando startet das RDBMS Derby, welches unabhängig vom Applikationsserver läuft. Die Datenbank wird mit dem folgenden Kommando gestoppt:

```
% asadmin stop-database
```

6.2 Deployment von Webservices und Webservice-Providern

[4] Der SOAP-basierte `Teams`-Dienst aus Abschnitt 1.9 hat vier Klassen: `Player`, `Team`, `TeamsUtility` und `Teams` (die SIB). In diesem Beispiel liegen die vier Dateien im Verzeichnis *ch06/team* und gehören zum Package `ch06.team` Zur Wiederholung hier nochmals die ursprüngliche SIB, diesesmal aber im neuen Package:

```
package ch06.team;

import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Teams {

    private TeamsUtility utils;
    public Teams() { utils = new TeamsUtility(); }

    @WebMethod
    public Team getTeam(String name) { return utils.getTeam(name); }
    @WebMethod
    public List<Team> getTeams() { return utils.getTeams(); }

}
```

Nach dem Übersetzen werden die *.class* Dateien in das Verzeichnis *ch06/team/WEB-INF/classes-/ch06/team* kopiert, da Tomcat, der Webcontainer in GlassFish, die übersetzten Klassen unterhalb des Verzeichnisses *WEB-INF/classes* erwartet. Die WAR-Datei wird mit dem folgenden Kommando erzeugt:

```
% jar cvf team.war WEB-INF
```

und anschließend im Verzeichnis *$AS_HOME/domains/domain1/autodeploy* deponiert. Obwohl der `Teams`-Dienst im Dokumentstil geschrieben ist, müssen die von einem solchen Dienst benötigten JAX-B-Artefakte nicht manuell per `wsgen` generiert werden. GlassFish wird mit dem aktuellen Metro-Release ausgeliefert, welches diese Artefakte automatisch generiert.

[5] Die clientseitigen `wsimport`-Artefakte werden in gewohnter Weise erzeugt:

```
% wsimport -keep -p clientC http://localhost:8081/team/TeamsService?wsdl
```

Der folgende Client verwendet die `wsimport`-Artefakte:

```
import teamsC.TeamsService;
import teamsC.Teams;
import teamsC.Team;
import teamsC.Player;
import java.util.List;

class TeamsClient {
    public static void main(String[] args) {
        TeamsService service = new TeamsService();
        Teams port = service.getTeamsPort();

        List<Team> teams = port.getTeams();
        for (Team team : teams) {
            System.out.println("Team name: " + team.getName() +
                               " (roster count: " + team.getRosterCount() + ")");
            for (Player player : team.getPlayers())
                System.out.println("  Player: " + player.getNickname());
        }
    }
}
```

Die Ausgabe lautet:

```
Team name: Abbott and Costello (roster count: 2)
  Player: Bud
  Player: Lou
Team name: Marx Brothers (roster count: 3)
  Player: Chico
  Player: Groucho
  Player: Harpo
Team name: Burns and Allen (roster count: 2)
  Player: George
  Player: Gracie
```

Neben dem WSDL-Dokument generiert GlassFish die Deployment-Artefakte, darunter zwei Dokumente namens *webservices.xml* und *sun-web.xml*. Diese Dokumente können über die Administratorkonsole betrachtet werden.

6.2.1 Deployment von Webservice-Providern

[6] Das Deployment eines Webservice-Providers ist etwas aufwändiger, da die WAR-Datei die Dateien *web.xml* und *sun-jaxws.xml* beinhalten muß. Zunächst der Quelltext eines Dienstes im REST-Stil, welcher Temperaturen konvertiert und *Fibonaccizahlen* berechnet:

```
package ch06.rest;

import javax.xml.ws.Provider;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.Source;
import javax.xml.transform.Result;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamSource;
```

```
import javax.xml.transform.stream.StreamResult;
import javax.annotation.Resource;
import javax.xml.ws.BindingType;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPException;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.http.HTTPBinding;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.StringWriter;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;

@WebServiceProvider
@ServiceMode(value = javax.xml.ws.Service.Mode.MESSAGE)
@BindingType(value = HTTPBinding.HTTP_BINDING)
public class RestfulProviderD implements Provider<Source> {
    @Resource
    protected WebServiceContext ws_context;
    protected Document document; // DOM tree
    public Source invoke(Source request) {
        try {
            if (ws_context == null) throw new RuntimeException("No ws_context.");
            MessageContext msg_context = ws_context.getMessageContext();
            // Check the HTTP request verb. In this case, only POST is supported.
            String http_verb = (String)
                msg_context.get(MessageContext.HTTP_REQUEST_METHOD);
            if (!http_verb.toUpperCase().trim().equals("POST"))
                throw new HTTPException(405); // bad verb exception
            build_document(request);
            String operation = extract_node("operation").trim();
            String operand = extract_node("operand").trim();
            if (operation.equals("fib"))      return fib_response(operand);
            else if (operation.equals("c2f")) return c2f_response(operand);
            else if (operation.equals("f2c")) return (f2c_response(operand));
            throw new HTTPException(404); // client error
        }
        catch(Exception e) { throw new HTTPException(500); }
    }
    // Build a DOM tree from the XML source for later lookups.
    private void build_document(Source request) {
        try {
            Transformer transformer =
                TransformerFactory.newInstance().newTransformer();
            this.document = DocumentBuilderFactory.newInstance()
                .newDocumentBuilder().newDocument();
            Result result = new DOMResult(this.document);
            transformer.transform(request, result);
        }
        catch(Exception e) { this.document = null; }
    }

    // Extract a node's value from the DOM tree given the node's tag name.
```

```
    private String extract_node(String tag_name) {
        try {
            NodeList nodes = this.document.getElementsByTagName(tag_name);
            Node node = nodes.item(0);
            return node.getFirstChild().getNodeValue().trim();
        }
        catch(Exception e) { return null; }
    }

    // Prepare a response Source in which obj refers to the return value.
    private Source prepare_source(Object obj) {
        String xml =
            "<uri:restfulProvider xmlns:uri = 'http://foo.bar.baz'>" +
            "<return>" + obj + "</return>" +
            "</uri:restfulProvider>";
        return new StreamSource(new ByteArrayInputStream(xml.getBytes()));
    }

    private Source fib_response(String num) {
        try {
            int n = Integer.parseInt(num.trim());
            int fib = 1;
            int prev = 0;

            for (int i = 2; i <= n; ++i) {
                int temp = fib;
                fib += prev;
                prev = temp;
            }
            return prepare_source(fib);
        }
        catch(Exception e) { throw new HTTPException(500); }
    }

    private Source c2f_response(String num) {
        try {
            float c = Float.parseFloat(num.trim());
            float f = 32.0f + (c * 9.0f / 5.0f);
            return prepare_source(f);
        }
        catch(Exception e) { throw new HTTPException(500); }
    }

    // Compute f2c(c)
    private Source f2c_response(String num) {
        try {
            float f = Float.parseFloat(num.trim());
            float c = (5.0f / 9.0f) * (f - 32.0f);
            return prepare_source(c);
        }
        catch(Exception e) { throw new HTTPException(500); }
    }
}
```

[7] Würden die übersetzten Klassen im Verzeichnis *WEB-INF/classes/ch06/rest*, wie im obigen Beispiel, ohne die XML-Dokumente *web.xml* und *sun-jaxws.xml* in eine WAR-Datei eingesetzt und deployt, so würde eine Anfrage an den Dienst mit dem Statuswert `404` (`Not Found`) beantwortet werden. Der Webservice-Provider muß zusammen mit den beiden XML-Dokumenten deployt werden. Zunächst *web.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>RestfulProvider Service</display-name>
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <display-name>RestfulProviderD</display-name>
    <servlet-name>RestfulProviderD</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RestfulProviderD</servlet-name>
    <url-pattern>/restful/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Nun *sun-jaxws.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  <endpoint name="RestfulProviderD"
    implementation="ch06.rest.RestfulProviderD"
    binding="http://www.w3.org/2004/08/wsdl/http"
    url-pattern='/restful/*' />
</endpoints>
```

Hinsichtlich der Struktur sind dies im wesentlichen dieselben XML-Konfigurationsdateien, wie beim Deployment von SOAP-basierten Webservices beziehungsweise Webservices im REST-Stil über Tomcat. GlassFish erkennt einen Webservice automatisch, sogar ohne Konfigurationsdatei, nicht aber einen Webservice-Provider. Somit müssen die Konfigurationsdateien im zweiten Fall vorhanden sein.

[8] Nun zum Client für diesen Dienst. Zur Abwechselung parst der Client einen DOM-Baum, diesesmal allerdings ohne XPath:

```
import javax.xml.ws.Service;
import javax.xml.namespace.QName;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.Dispatch;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.ws.handler.MessageContext;
import java.net.URL;
import java.util.Map;
import java.io.StringReader;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import org.w3c.dom.Document;
```

```
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;

class DispatchClient {
    private static String xml =
        ”<?xml version = ’1.0’ encoding = ’UTF-8’ ?>” + ”\n” +
        ”<uri:RequestDocument xmlns:uri = ’urn:RequestDocumentNS’>” + ”\n” +
        ”  <operation>f2c</operation>” + ”\n” +
        ”  <operand>-40</operand>” + ”\n” +
        ”</uri:RequestDocument>” + ”\n”;

    public static void main(String[] args) throws Exception {
        QName qname = new QName(”””, ”””);
        String url_string = ”http://127.0.0.1:8080/restfulD/restful/”;
        URL url = new URL(url_string);

        // Create the service and add a port
        Service service = Service.create(qname);
        service.addPort(qname, HTTPBinding.HTTP_BINDING, url_string);

        Dispatch<Source> dispatcher = service.createDispatch(qname,
                                                Source.class,
                                                javax.xml.ws.Service.Mode.MESSAGE);
        Map<String, Object> rc = dispatcher.getRequestContext();
        rc.put(MessageContext.HTTP_REQUEST_METHOD, ”POST”);
        Source result = dispatcher.invoke(new StreamSource(new StringReader(xml)));
        parse_response(result);
    }

    private static void parse_response(Source res) throws Exception {
        Transformer transformer = TransformerFactory.newInstance().newTransformer();
        ByteArrayOutputStream bao = new ByteArrayOutputStream();
        StreamResult sr = new StreamResult(bao);
        transformer.transform(res, sr);
        ByteArrayInputStream bai = new ByteArrayInputStream(bao.toByteArray());
        DocumentBuilder db = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document root = db.parse(bai);
        NodeList nodes = root.getElementsByTagName(”return”);
        Node node = nodes.item(0); // should be only one <return> element
        System.out.println(”Request document:\n” + xml);
        System.out.println(”Return value: ” + node.getFirstChild().getNodeValue());
    }
}
```

Die Ausgabe lautet:

```
Request document:
<?xml version = ’1.0’ encoding = ’UTF-8’ ?>
<uri:RequestDocument xmlns:uri = ’urn:RequestDocumentNS’>
  <operation>f2c</operation>
  <operand>-40</operand>
</uri:RequestDocument>

Return value: -40.0
```

Das Deployment eines Webservice’ und eines Webservice-Providers unterscheiden sich bei GlassFish geringfügig, obwohl beide Vorgänge mit Hilfe der statischen `Endpoint`-Methode `publish()` beziehungsweise beim eigenständigen Tomcat-Container identisch sind. Der nächste Abschnitt veranschaulicht, wie ein deployter Webservice aus einer JSP-Seite oder einem JSF-Skript aufgerufen

wird.

6.3 Bedienung eines Webservice' über eine interaktive HTML-Seite

[9] Clients von Webservices sind in der Regel nicht interaktiv und insbesondere keine Webbrowser. Dieser Abschnitt zeigt as proof of concept, wie ein browserbasierter Client eines SOAP-basierten Webservice' funktionieren könnte. Der Webservice ist einfach und vertraut:

```
package ch06.tc;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class TempConvert {

    @WebMethod
    public float c2f(float t) { return 32.0F + (t * 9.0F / 5.0F); }

    @WebMethod
    public float f2c(float t) { return (5.0F / 9.0F) * (t - 32.0F); }

}
```

Nachdem die übersetzte Klasse `TempConvert` im Verzeichnis *ch06/tc/WEB-INF/classes/ch06/tc* deponiert ist, wird die WAR-Datei erzeugt und in das Deploymentverzeichnis *$AS_HOME/domains/domain1/autodeploy* kopiert.

[10] Der nächste Schritt ist das Schreiben einer interaktiven Webapplikation, die in diesem Fall aus einer HTML-Seite, zwei kleinen JSP-Seiten und einer einfachen *web.xml*-Datei besteht. Zunächst das HTML-Formular in dem der Benutzer die zu umzurechnende Temperatur eingibt:

```
<html><body>
  <form method = 'post' action = 'temp_convert.jsp'>
    Temperature to convert: <input type = 'text' name = 'temperature'><br /><hr />
    <input type = 'submit' value = ' Click to submit ' />
  </form>
</body></html>
```

Die folgende JSP-Seite ruft den `TempConvert`-Dienst auf, um einen Temperaturwert umzurechnen:

```
<%@ page errorPage = 'error.jsp' %>
<%@ page import = 'client.TempConvert' %>
<%@ page import = 'client.TempConvertService' %>
<html><body>
<%! private float f2c, c2f, temp; %>
<%
  String temp_str = request.getParameter("temperature");
  if (temp_str != null) temp = Float.parseFloat(temp_str.trim());

  TempConvertService service = new TempConvertService();
  TempConvert port = service.getTempConvertPort();
  f2c = port.f2C(temp);
  c2f = port.c2F(temp);
%>
<p><%= this.temp %>F = <%= this.f2c %>C</p>
<p><%= this.temp %>C = <%= this.c2f %>F</p>
<a href = 'index.html'>Try another</a>
</body></html>
```

Die beiden wichtigen Klassen `TempConvert` und `TempConvertService` sind `wsimport`-generierte Artefakte im Unterverzeichnis *WEB-INF/classes/client* des Verzeichnisses, welches die HTML-Seite und die beiden JSP-Seiten enthält. Der Vollständigkeit halber, nun die Fehlerseite *error.jsp*:

```
<%@ page isErrorPage = ”true” %>
<html>
  <% response.setStatus(400); %>
  <body>
    <h2><%= exception.toString() %></h2>
    <p>Bad data: please try again.</p>
    <p><a href = ”index.html”>Return to home page</a></p>
  </body>
</html>
```

Der DD ist ebenfalls kurz und bündig:

```
<?xml version = ’1.0’ encoding = ’ISO-8859-1’?>
<web-app xmlns = ’http://java.sun.com/xml/ns/javaee’
    xmlns:xsi = ’http://www.w3.org/2001/XMLSchema-instance’
    xsi:schemaLocation = ’http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd’
    version = ’2.5’>
  <description>JSP frontend to TempConvert service</description>
  <error-page>
    <exception-type>java.lang.NumberFormatException</exception-type>
    <location>/error.jsp</location>
  </error-page>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

[11] Die HTML-Seite und die JSP-Seiten werden zusammen mit den übersetzten Klasse und *web.xml* zu einer WAR-Datei verpackt:

```
% jar cvf tcJSP.war index.html *.jsp WEB-INF
```

und im Deploymentverzeichnis des GlassFish-Applikationservers deponiert. Zum Testen öffnen Sie ein Browserfenster mit der URL *http://localhost:8081/tcJSP*.

6.4 Ein Webservice als EJB

[12] Der EJB-Container ist durch die Lösung von Problemen wie Threadsicherheit, das Bevorraten von Objekten und transaction-guarded persistence programmiererfreundlich. Dieser Abschnitt zeigt, wie ein Webservice Nutzen aus diesen Vorteilen gewinnen kann, in dem er als zustandslose Sitzungs-EJB implementiert wird. Das Beispiel geht in zwei Schritten vor sich. Der erste Schritt betrifft die Details des Deployments eines Webservice’ als zustandslose Sitzungs-EJB. Der zweite Schritt ergänzt die Datenbankpersistenz durch Einsetzen einer @Entity in die Applikation. Das Beispiel ist einfach angelegt, damit die EJB- und @Entity-Details hervortreten.

6.4.1 Implementierung als zustandlose Sitzungs-EJB

[13] Zuerst die SEI des `FibEJB`-Dienstes, der als zustandslose Sitzungs-EJB implementiert wird:

```
package ch06.ejb;

import java.util.List;
```

```
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;

@Stateless
@WebService
public interface Fib {
    @WebMethod int fib(int n);
    @WebMethod List getFibs();
}
```

Die wichtigste Änderung ist das Vorkommen der `@Stateless`-Annotation. Die Annotation tritt in der SIB nochmals auf:

```
package ch06.ejb;

import java.util.List;
import java.util.ArrayList;
import javax.ejb.Stateless;
import javax.jws.WebService;

@Stateless
@WebService(endpointInterface = "ch06.ejb.Fib")
public class FibEJB implements Fib {
    public int fib(int n) {
        int fib = 1, prev = 0;
        for (int i = 2; i <= n; i++) {
            int temp = fib;
            fib += prev;
            prev = temp;
        }
        return fib;
    }
    public List getFibs() { return new ArrayList(); } // for now, empty list
}
```

Die implementierende Klasse besteht, im Sinne der zustandslosen Sitzungs-EJB, aus in sich geschlossenen Methoden, deren Funktionsweise nicht vom Inhalt dynamischer Felder anhängt. Die erste Version der `getFibs()`-Methode gibt eine Referenz auf ein leeres Containerobjekt vom Typ *`List`* zurück. Die nächste Version liefert die Zeilen einer Datenbanktabelle.

[14] Eine EJB-Implementierung wird anders verpackt, als eine Standard-WAR-Datei. Zunächst werden alle benötigten Klassen in einer JAR-Datei mit frei wählbarem Namen verpackt. Das folgende `jar`-Kommando erzeugt diese JAR-Datei:

```
% jar cvf rc.jar ch06/ejb/*.class
```

Das Archiv enthält keine Konfigurationsdatei. Folglich erzeugt GlassFish eine solche Datei automatisch. Diese JAR-Datei wird anschließend in einer EAR-Datei (Enterprise Archive) verpackt;

```
% jar cvf fib.ear rc.jar
```

Die traditionelle Endung für EJBs ist *.ear*. GlassFish erwartet, daß eine deployte EAR-Datei wenigstens eine EJB oder mit der `@Entity`-Annotation versehene POJO-Klasse enthält. Eine EAR-Datei auf Produktionsniveau beinhaltet in der Regel mehrere EJBs. Eine EAR-Datei enthält stets eine JAR-Datei pro EJB und kann auch WAR-Dateien enthalten. Eine EAR-Datei bildet eine einzelne Enterprise-Applikation und die einzelnen JAR-Dateien beinhalten die verschiedenen Komponenten der Applikation. In diesem Beispiel gibt es nur eine Komponente, nämlich die zustandslose Sitzungs-EJB in ihrer eigenen JAR-Datei.

[15] Nachdem eine EAR-Datei in der gewohnten Weise in das Deploymentverzeichnis *domains-/domain1/autodeploy* kopiert wurde, gestattet die Administratorkonsole, Einsicht in die deployte Applikation zu nehmen, welche im Abschnitt „Webservices“ der Konsole angezeigt wird. GlassFish generiert automatisch verschiedene Deploymentartefakte, darunter das WSDL-Dokument:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net.
     RI's version is JAX-WS RI 2.1.3.1-hudson-417-SNAPSHOT. -->
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net.
     RI's version is JAX-WS RI 2.1.3.1-hudson-417-SNAPSHOT. -->

<definitions
    xmlns:wsu=
      "http://docs.oasis-open.org/wss/2004/01/
        oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://ejb.ch06/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://ejb.ch06/"
    name="FibEJBService">

  <ns1:Policy
      xmlns:ns1="http://www.w3.org/ns/ws-policy"
      wsu:Id="FibEJBPortBinding_getFibs_WSAT_Policy">
    <ns1:ExactlyOne>
      <ns1:All>
        <ns2:ATAlwaysCapability
            xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/10/wsat">
        </ns2:ATAlwaysCapability>
        <ns3:ATAssertion
            xmlns:ns4="http://schemas.xmlsoap.org/ws/2002/12/policy"
            xmlns:ns3="http://schemas.xmlsoap.org/ws/2004/10/wsat"
            ns1:Optional="true"
            ns4:Optional="true">
        </ns3:ATAssertion>
      </ns1:All>
    </ns1:ExactlyOne>
  </ns1:Policy>

  <ns5:Policy
      xmlns:ns5="http://www.w3.org/ns/ws-policy"
      wsu:Id="FibEJBPortBinding_fib_WSAT_Policy">
    <ns5:ExactlyOne>
      <ns5:All>
        <ns6:ATAlwaysCapability
            xmlns:ns6="http://schemas.xmlsoap.org/ws/2004/10/wsat">
        </ns6:ATAlwaysCapability>
        <ns7:ATAssertion
            xmlns:ns8="http://schemas.xmlsoap.org/ws/2002/12/policy"
            xmlns:ns7="http://schemas.xmlsoap.org/ws/2004/10/wsat"
            ns5:Optional="true" ns8:Optional="true">
        </ns7:ATAssertion>
      </ns5:All>
    </ns5:ExactlyOne>
  </ns5:Policy>

  <types>
    <xsd:schema>
```

```
    <xsd:import
        namespace="http://ejb.ch06/"
        schemaLocation="http://localhost:8080/FibEJBService/FibEJB?xsd=1">
    </xsd:import>
  </xsd:schema>
</types>

<message name="fib">
  <part name="parameters" element="tns:fib"></part>
</message>
<message name="fibResponse">
  <part name="parameters" element="tns:fibResponse"></part>
</message>
<message name="getFibs">
  <part name="parameters" element="tns:getFibs"></part>
</message>
<message name="getFibsResponse">
  <part name="parameters" element="tns:getFibsResponse"></part>
</message>

<portType name="Fib">
  <operation name="fib">
      <ns9:PolicyReference
          xmlns:ns9="http://www.w3.org/ns/ws-policy"
          URI="#FibEJBPortBinding_fib_WSAT_Policy">
      </ns9:PolicyReference>
    <input message="tns:fib"></input>
    <output message="tns:fibResponse"></output>
  </operation>
  <operation name="getFibs">
    <ns10:PolicyReference
        xmlns:ns10="http://www.w3.org/ns/ws-policy"
        URI="#FibEJBPortBinding_getFibs_WSAT_Policy">
    </ns10:PolicyReference>
    <input message="tns:getFibs"></input>
    <output message="tns:getFibsResponse"></output>
  </operation>
</portType>

<binding name="FibEJBPortBinding" type="tns:Fib">
  <soap:binding
      transport="http://schemas.xmlsoap.org/soap/http"
      style="document">
  </soap:binding>
  <operation name="fib">
    <ns11:PolicyReference xmlns:ns11="http://www.w3.org/ns/ws-policy"
        URI="#FibEJBPortBinding_fib_WSAT_Policy">
    </ns11:PolicyReference>
    <soap:operation soapAction=""></soap:operation>

    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
  <operation name="getFibs">
    <ns12:PolicyReference xmlns:ns12="http://www.w3.org/ns/ws-policy"
```

```
            URI="#FibEJBPortBinding_getFibs_WSAT_Policy">
        </ns12:PolicyReference>
        <soap:operation soapAction=""></soap:operation>
        <input>
          <soap:body use="literal"></soap:body>
        </input>
        <output>
          <soap:body use="literal"></soap:body>
        </output>
      </operation>
    </binding>

    <service name="FibEJBService">
      <port name="FibEJBPort" binding="tns:FibEJBPortBinding">
        <soap:address location="http://localhost:8080/FibEJBService/FibEJB">
        </soap:address>
      </port>
    </service>
  </definitions>
```

Dieses WSDL-Dokument unterscheidet sich von den vorausgegangenen durch die beiden `<policy>`-Abschnitte, einem XML-Dialekt zur Beschreibung der ~~capabilities~~ des Webservice' und der Anforderungen an den Client. GlassFish beinhaltet das aktuelle Metro-Release, welches WS-Policy und damit verwandte WS-* Initiativen unterstützt, die die Interoperabilität von Webservices fördern: WSIT im Metro-Jargon. Die Richtlinienvorlage wird zu Beginn des WSDL-Dokumentes deklariert und vom restlichen WSDL-Dokument aus referenziert. Alle Richtlinienvorlagen sind allerdings als lediglich optional gekennzeichnet. Die eigentliche Idee besteht darin, daß ein Webservice und seine potentiellen Clients Richtlinien zur Sicherheit und anderen dienstbezogenen Anforderungen ausdrücken können sollten. Es gibt beispielsweise einen Satz von Richtlinien über zuverlässige Nachrichtenübermittlung, der zur WS-ReliableMessaging-Spezifikation gehört. Unter dieser Spezifikation könnte ein WSDL-Dokument bekannt machen, daß jede Nachricht genau einmal gesendet wird, folglich also niemals dupliziert werden darf. Stößt ein Mittler entlang des Pfades zwischen Sender und Empfänger beim Ausliefern einer Nachricht auf ein Problem, ist ein entsprechendes Fehlerobjekt auszuwerfen und die Nachricht *nicht* noch einmal zu senden. Eine andere Richtlinie gemäß WS-ReliableMessaging schreibt vor, daß Nachrichten in der gesendeten Reihenfolge empfangen werden, eine Eigenschaft, welche die TCP-Infrastruktur unter HTTP/HTTPS bereits garantiert. Da SOAP-basierte Webservices aber entwurfsbedingt neutral bezüglich des Transportprotokolls sind, gibt es eine Spezifikation über zuverlässige Nachrichtübermittlung, diverse über Sicherheit und so weiter.

6.4.2 Die Endpunkt-URL eines EJB-basierten Dienstes

[16] Die Endpunkt-URL des EJB-basierten Webservice' weicht ebenfalls von den bisherigen WAR-Beispielen ab. Der Name der EAR-Datei tritt beispielsweise nicht im Pfad auf, wie der Name einer WAR-Datei. Der Pfadabschnitt der URL besteht aus zwei Teilen: Erstens, der Name der SIB mit der Endung *Service*, in diesem Beispiel *FibEJBService*. Zweitens, der Name der SIB, hier also *FibEJB*. Die URL im folgenden Perl-Client zeigt die Kombination:

```
#!/usr/bin/perl -w
use SOAP::Lite;
use strict;
my $url = 'http://localhost:8081/FibEJBService/FibEJB?wsdl';
my $service = SOAP::Lite->service($url);
print $service->fib(7), "\n";
```

Dieses Beispiel verwendet zur Bequemlichkeit und Einfachheit nur Standardwerte. Die EJB wird beispielsweise unter dem Namen *FibEJBService* und dem Pfad */FibEJBService/FibEJB* deployt. Die Voreinstellungen können über Annotationsattribute oder Einträge im Deploymentdeskriptor überschrieben werden. Immerhin gibt es voreingestellte Werte.

[17] Der nächste Schritt ist die Aufnahme einer Datenbanktabelle als dauerhafter Speicher sowie einer mit `@Entity` annotierten Klasse, um die Interaktion zwischen Webservice und Datenbank zu automatisieren. Dieser Schritt erfordert nur moderate Ergänzungen am Dienst.

6.4.3 Datenbankunterstützung per @Entity-Annotation

[18] Das SEI *`Fib`* bleibt unverändert, aber die SIB `FibEJB` wird folgendermaßen modifiziert:

```
package ch06.ejb;
import java.util.List;
import java.util.ArrayList;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.Query;

@Stateless
@WebService(endpointInterface = "ch06.ejb.Fib")
public class FibEJB implements Fib {

    @PersistenceContext(unitName = "FibServicePU")
    private EntityManager em;

    public int fib(int n) {
        // Computed already? If so, return.
        FibNum fn = em.find(FibNum.class, n); // read from database
        if (fn != null) return fn.getF();

        int f = compute_fib(Math.abs(n));
        fn = new FibNum();
        fn.setN(n);
        fn.setF(f);
        em.persist(fn);                       // write to database
        return f;
    }

    public List getFibs() {
        Query query = em.createNativeQuery("select * from FibNum");
        // fib_nums is a list of pairs: N and Fibonacci(N)
        List fib_nums = query.getResultList(); // read from database
        List results = new ArrayList();
        for (Object next : fib_nums) {
            List list = (List) next;
            for (Object n : list) results.add(n);
        }
        return results; // N, fib(N), K, fib(K),...
    }

    private int compute_fib(int n) {
        int fib = 1, prev = 0;
        for (int i = 2; i <= n; i++) {
            int temp = fib;
            fib += prev;
```

```
                prev = temp;
            }
            return fib;
        }
    }
```

Sendet ein Client eine Anfrage nach einer bestimmten Fibonaccizahl, so führt der Dienst eine Suchanfrage (`find()`-Methode) an die Datenbank aus. Ist der angeforderte Wert nicht in der Datenbank, so wird er berechnet und dort gespeichert. Fordert der Client eine Liste der bis jetzt berechneten Fibonaccizahlen an, so führt der Dienst eine Anfrage an die Datenbank aus, um die Liste zusammenzustellen und gibt sie anschließend zurück. Dieser Überblick wird nun mit Details ausgebaut.

[19] Die SIB `FibEJB` stützt sich auf dependency injection, um eine Referenz auf einen Entitätsmanager (*`EntityManager`*) zu bekommen, welcher die Objekte der mit `@Entity` annotierten Klassen in einem gegebenen Persistenzkontext (`PersistenceContext`) verwaltet. Die `@Entity`-Klasse ist in diesem Fall `FibNum`:

```
package ch06.ejb;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Column;
import java.io.Serializable;

// A FibNum is a pair: an integer N and its Fibonacci value.
@Entity
public class FibNum implements Serializable {
    private int n;
    private int f;

    public FibNum() { }

    @Id
    public int getN() { return n; }
    public void setN(int n) { this.n = n; }

    public int getF() { return f; }
    public void setF(int f) { this.f = f; }
}
```

Ein `FibNum`-Objekt hat zwei Felder, nämlich eines für das ganzzahlige Argument `n` und eines für die entsprechende Fibonaccizahl des Argumentes (`f`). In der zugehörigen Datenbanktabelle hat das ganzzahlige Argument `n` die Rolle des Primärschlüssels, das heißt as the `@Id`. Das Anlegen der Datenbanktabelle wird in Kürze erläutert. Zuvor sind einige Informationen über den Entitätsmanager an der Reihe.

[20] Der Persistenzkontext eines EJB-Containers unterstützt per Voreinstellung Transaktionen mit Hilfe eines *JTA*-Managers (Java Transaction API). Transaktionen sind somit automatisch gegeben, da vom Container angeboten und verwaltet. Wiederum per Voreinstellung erstreckt sich der Geltungsbereich einer Transaktion auf eine einzelne Lese- oder Schreiboperation in der Datenbank. Die `fib()`-Methode ruft die `find()`-Methode des Entitätsmanagers auf, um eine Fibonaccizahl (`FibNum`) aus der Datenbank abzufragen, wobei jede Fibonaccizahl eine Tabellenzeile ist. Fehlt eine bestimmte Fibonaccizahl in der Tabelle, so wird ein neues Objekt der `@Entity`-Klasse `FibNum` erzeugt und durch einen Aufruf der `persist()`-Methode des Entitätsmanagers dauerhaft in der Datenbank gespeichert.

[21] Die Methoden des Entitätsmanagers sind nicht von Natur aus threadsicher. Die Threadsicherheit dieser Methoden in diesem Beispiel kommt daher, daß der Entitätsmanager von einem Feld

in einer EJB referenziert wird und somit in den Genuß der vom EJB-Container gewährleisteten Threadsicherheit kommt.

6.4.4 Die Konfigurationsdatei persistence.xml

[22] Nichts im Quelltext des Webservice' oder den Hilfsklassen bezieht sich explizit auf die Datenbank oder die Tabelle darin. Diese Details sind in der Konfigurationsdatei *META-INF/persistence.xml* deklariert:

```
<persistence>
  <persistence-unit name="FibServicePU" transaction-type="JTA">
    <description>
      This unit manages Fibonacci number persistence.
    </description>
    <jta-data-source>jdbc/__default</jta-data-source>
    <properties>
      <!--Use the java2db feature -->
      <property name="toplink.ddl-generation" value="drop-and-create-tables" />
      <!-- Generate the sql specific to Derby database -->
      <property name="toplink.platform.class.name"
          value="oracle.toplink.essentials.platform.database.DerbyPlatform" />
    </properties>
    <class>ch06.ejb.FibNum</class>
  </persistence-unit>
</persistence>
```

Jede JAR-Datei mit einer `@Entity`-Klasse sollte ein *META-INF*-Verzeichnis mit einer Datei *persistence.xml* enthalten. Die hervorgehobenen Zeilen bedürfen der Erläuterung.

[23] *GlassFish* verfügt über mehrere JDBC-Resourcen, darunter diejenige hier verwendete mit dem JNDI-Namen `jdbc/__default`. Die Administratorkonsole bietet Funktionalität, um neue Datenbanken anzulegen. GlassFish wird mit dem RDBMS Apache Derby ausgeliefert, welches das TopLink-Kommando `java2db` beinhaltet, das ein Tabellenschema aus einer Java-Klasse erzeugen kann, hier `FibNum`. Denken Sie daran, daß die Klasse `FibNum` mit der Annotation `@Entity` versehen ist und zwei dynamische ganzzahlige Felder für eine Zahl (den Primärschlüssel) und die entsprechende Fibonaccizahl besitzt. Das `java2db`-Kommando erzeugt die entsprechende Datenbanktabelle mit zwei Spalten, eine für das Argument und eine für die Fibonaccizahl. Falls erwünscht können der Name der Tabelle und die Spaltennamen mit Hilfe der Annotationen `@Table` und `@Column` eingestellt werden.

[24] Die mit `@WebMethod` annotierte `fib()`-Methode der SIB `FibEJB` ruft die `find()`-Methode auf, deren Argument der Primärschlüssel für das gewünschte Zahlenpaar ist. Im Gegensatz dazu verwendet die `getFibs()`-Methode eine Anfrage, um alle Zeilen aus der Datenbank abzufragen. Die Zeilen werden als Liste von Zahlenpaaren zurückgegeben, das heißt in der Form $((1,1),(2,1),(3,2),\ldots)$. Die `getFibs()`-Methode verwendet eine native im Gegensatz zu einer JPQL-Anfrage (Java Persistence Query Language), um die Zeilen anzufordern und erzeugt eine einfache Liste ganzer Zahlen, die an die Quelle der Anfrage zurückgegeben wird. Die Liste hat das Format $(1,1,2,1,3,2,\ldots)$ mit Argument und Fibonaccizahl direkt nebeneinander.

[25] Hier ist ein Client für den `FibEJB`-Dienst:

```
import clientEJB.FibEJBService;
import clientEJB.Fib;
import java.util.List;

class ClientEJB {
    public static void main(String args[]) {
```

```
        FibEJBService service = new FibEJBService();
        Fib port = service.getFibEJBPort();

        final int n=8;
        for (int i=1; i<n; i++)
            System.out.println("Fib(" + i + ") == "+port.fib(i));

        List fibs = port.getFibs();
        for (Object next : fibs) System.out.println(next);
    }
}
```

Die Ausgabe lautet:

```
Fib(1) == 1
Fib(2) == 1
Fib(3) == 2
Fib(4) == 3
Fib(5) == 5
Fib(6) == 8
Fib(7) == 13
1
1
2
1
3
2
4
3
5
5
6
8
7
13
```

6.4.5 Der Deploymentdeskriptor der EJB

[26] Der `FibEJB`-Dienst ist als EJB implementiert und in einer EAR-Datei ohne Deploymentdeskriptor (DD) verpackt. Der DD ist ein XML-Dokument in der Datei *ejb-jar.xml*. In einer EAR-Datei liegt jede EJB in einer eigenen JAR-Datei, die einen optionalen DD für die EJB enthält. Hier ist der von GlassFish für `FibEJB` generierte DD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    metadata-complete="true" version="3.0"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <enterprise-beans>
    <session>
      <display-name>Fib</display-name>
      <ejb-name>Fib</ejb-name>
      <ejb-class>ch06.ejb.Fib</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
```

```
        <use-caller-identity />
      </security-identity>
    </session>
    <session>
      <display-name>FibEJB</display-name>
      <ejb-name>FibEJB</ejb-name>
      <business-local>ch06.ejb.Fib</business-local>
      <service-endpoint>ch06.ejb.Fib</service-endpoint>
      <ejb-class>ch06.ejb.FibEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <persistence-context-ref>
        <persistence-context-ref-name>ch06.ejb.FibEJB/em
        </persistence-context-ref-name>
        <persistence-unit-name>FibServicePU</persistence-unit-name>
        <persistence-context-type>Transaction</persistence-context-type>
        <injection-target>
          <injection-target-class>ch06.ejb.FibEJB</injection-target-class>
          <injection-target-name>em</injection-target-name>
        </injection-target>
      </persistence-context-ref>
      <security-identity>
        <use-caller-identity />
      </security-identity>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Der Deploymentdeskriptor enthält zwei `<session>`-Abschnitte, einen am Anfang für das SEI *`Fib`* und einen am Ende für die SIB `FibEJB`. Beide Abschnitte deklarieren, daß die Sitzungs-EJB als zustandlos und mit containergesteuerten Transaktionen deployt werden soll. Der Abschnitt in der SIB liefert zusätzliche Details über die persistence, darunter die Referenz `em`, die auf den in `FibEJB` verwendeten Entitätsmanager verweist. Der springende Punkt besteht darin, daß der Persistenzkontext transaktionsbasiert ist, wobei der EJB-Container für die Transaktionsverwaltung zuständig ist.

6.4.6 Servlet- und EJB-basierte Implementierungen von Webservices

[27] GlassFish gestattet das Deployment von Webservices entweder als WAR- oder als EAR-Datei. Im ersteren Fall charakterisiert GlassFish die Implementierung als *servletbasiert*, da die Anfragen an den Webservice von einem `WSServlet`-Objekt abgefangen und an den in der WAR-Datei verpackten Webservice übergeben werden, wie die Beispiele mit einem eigenständigen Tomcat-Container gezeigt haben. Im zweiteren Fall ist der Webservice eine zustandslose Sitzungs-EJB, die vom EJB-Container verwaltet wird. Bei der servletbasierten Implementierung behandelt der *Webcontainer* die Anfragen an den Webservice. Bei der EJB-basierten Implementierung behandelt der *EJB-Container* die Anfragen an den Webservice. Ein gegebener Webservice läßt sich also auf zwei unterschiedlichen Wegen implementieren. Damit stellt sich die Frage nach den Vor- und Nachteilen zwischen den beiden Implementierungsvarianten eines Webservice'.

[28] Der Hauptvorteil der EJB-basierten Implementierung ist, daß der EJB-Container mehr Dienste anbietet, als der Servletcontainer. Ein als EJB implementierter Webservice ist dadurch threadsicher, während die hinsichtlich ihrer Funktionalität identische Implementierung als POJO-Klasse nicht threadsicher ist. Die Operation `fib()` im `FibEJB`-Beispiel bewirkt eine Lese- und eventuell eine Schreiboperation in der Datenbank. Die Operation `getFibs()` bewirkt eine Leseoperation. Der

EJB-Container verpackt diese Datenbankoperationen in Transaktionen, which remain transparent. Der EJB-Container kümmert sich auch um die dauerhafte Speicherung, so daß die Applikation von dieser Aufgabe befreit ist. Bei einem servlet- oder `Endpoint`-basierten Dienst würde die Applikation selbst dafür verantwortlich sein.

[29] Vor EJB 3.0 konnte als starkes Argument angeführt werden, daß EJBs im Allgemeinen und selbst Sitzungs-EJBs einfach zu viel Arbeit für zu wenig Lohn seien. Die Stärke dieses Argumentes hat nachgelassen. Wie das `FibEJB`-Beispiel zeigt, verlangt die EJB-basierte Implementierung eines SOAP-basierten Dienstes lediglich eine zusätzliche Annotation (`@Stateless`) und das Verpacken eines EJB-basierten Dienstes in einer EAR-Datei ist nicht komplizierter als das Verpacken desselben Dienstes in einer WAR-Datei mit ihrer internen *WEB-INF/classes*-Struktur.

6.5 Java-Webservices und der Java Message Service

[30] GlassFish beinhaltet, wie jeder vollausgestattete Applikationsserver, einen JMS-Anbieter, Publish/Subscribe-artige (Form/schwarzes Brett) sowie *Punkt-zu-Punkt-artige (Warteschlange) Nachrichtenübermittlung* unterstützt. Themen und Warteschlangen sind dauerhafte Speichermedien. Eine Applikation kann zum Beispiel eine Nachricht an eine JMS-Warteschlange senden und eine andere Applikation diese Nachricht später aus der Warteschlange abholen. JMS spezifiziert allerdings nicht, wielange eine Implementierung braucht, um Nachrichten dauerhaft zu speichern. Dieser Abschnitt veranschaulicht die Grundlagen der Interaktion zwischen JAX-WS und JMS anhand zweier Webservices. Der `MsgSender`-Dienst hat eine `receive()`-Operation, die Nachrichten am Kopfende einer Warteschlange abholt ([Abbildung ??]).

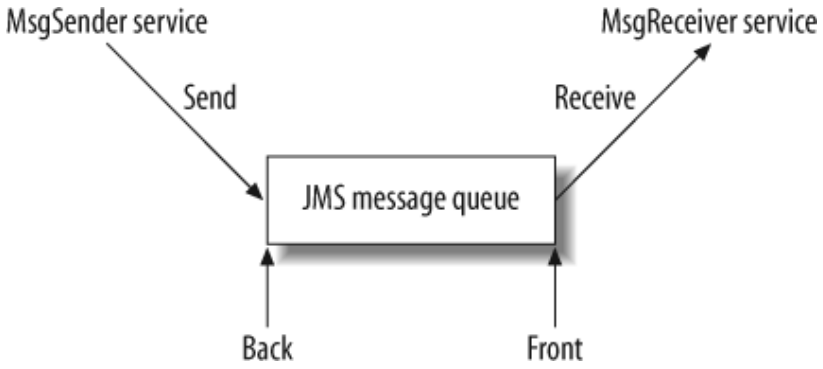

Abbildung 6.2: ...

[31] Hier ist der Quelltext der Klasse `MsgSender`:

```
package ch06.jms;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.Session;
import javax.jms.JMSException;
import javax.annotation.Resource;

// A web service that sends a message to a queue.
```

```
@WebService
public class MsgSender {

    // name and mappedName can differ; mappedName is the JNDI lookup name
    @Resource(name="qcf", mappedName="qcf")
    private QueueConnectionFactory qf;
    @Resource(name="jmsQ", mappedName="jmsQ")
    private Queue queue;

    private QueueConnection conn;

    @WebMethod
    public void send(String msg) {
        try {

            if (conn == null) conn = (QueueConnection) qf.createConnection();
            Session session =
                conn.createSession(false, // no transaction support
                                   Session.AUTO_ACKNOWLEDGE);

            // Wrap the string in a TextMessage for sending.
            TextMessage tmsg = session.createTextMessage(msg);
            session.createProducer(queue).send(tmsg);
            session.close();
        }

        catch(JMSException e) { throw new RuntimeException(e); }

    }

}
```

Einrichten von Nachrichtenthemen und Warteschlangen bei GlassFish

Das `asadmin`-Kommando im Verzeichnis *$AS_HOME/bin* kann verwendet werden, um festzustellen, ob der JMS-Anbieter konfiguriert ist und läuft. Der Aufruf lautet:

```
% asadmin jms-ping
```

Eine Statusmeldung wie

```
JMS Ping Status = RUNNING
Command jms-ping executed successfully.
```

dokumentiert, daß der JMS-Anbieter läuft. Der JMS-Anbieter sollte beim Start von GlassFish automatisch gestartet werden. GlassFish kann aber auch so konfiguriert werden, daß der JMS-Anbieter separat gestartet werden muß.

Die GlassFish-Administratorkonsole hat einen „Resources"-Reiter mit einem „JMS-Resources"-Unterreiter, der wiederum die Punkte „Connection Factories" und „Destination Resources" enthält. Letztere beiden sind Themen beziehungsweise Warteschlangen. Im folgenden Quelltext sind die Namen der Connection-Factory und der Warteschlange mit `@Resource` annotiert.

[32] Die Klasse `MsgSender` stützt sich auf dependency injection, zugeschaltet durch die Annotation `@Resource`, um je eine Referenz auf die in der Administratorkonsole erzeugte Connection-Factory und die Warteschlange zu erhalten. Das Senden einer Nachricht setzt eine Verbindung zum JMS-Anbieter, eine Sitzung mit dem Anbieter und einen message producer voraus, der die `send()`-Methode kapselt. JMS verfügt über mehrere Nachrichtentypen, darunter den hier gewählten Typ *`TextMessage`*. Nach dem Senden der Nachricht an die Warteschlange wird die Sitzung beendet,

während die Verbindung zum JMS-Anbieter für weitere Operationen auf der Warteschlange geöffnet bleibt.

[33] Der `MsgReceiver`-Dienst stützt sich auf dieselben Anweisungen für dependency injection. Die Konfiguration ist ebenfalls ähnlich, außer, daß der generische Typ *`Session`* durch *`QueueSession`* ersetzt wird. Das Interface *`QueueReceiver`* wird verwendet, um einen Teil der Vielfalt der JMS-API zu zeigen. Der Aufruf der im Interface *`QueueConnection`* deklarierten Methode `start()` beendet die Konfigurationsphase und gibt die Nachricht am Kopfende der Warteschlange zur Abholung frei, falls eine Nachricht existiert:

```
package ch06.jms;

import java.util.List;
import java.util.ArrayList;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.Session;
import javax.jms.Queue;
import javax.jms.QueueSession;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.QueueReceiver;
import javax.jms.JMSException;
import javax.annotation.Resource;

@WebService
public class MsgReceiver {

    // name and mappedName can differ
    @Resource(name="qcf", mappedName="qcf")
    private QueueConnectionFactory qf;
    @Resource(name="jmsQ", mappedName="jmsQ")
    private Queue queue;

    private QueueConnection conn;

    @WebMethod
    public String receive() {
        String cliche = null;
        try {
            if (conn == null) conn = qf.createQueueConnection();
            QueueSession session =
                conn.createQueueSession(false,
                                        Session.AUTO_ACKNOWLEDGE);
            QueueReceiver receiver = session.createReceiver(queue);
            conn.start();

            Message msg = receiver.receiveNoWait();
            if (msg != null && msg instanceof TextMessage ) {
                TextMessage tmsg = (TextMessage) msg;
                cliche = tmsg.getText().trim();
            }
        }
        catch(JMSException e) { throw new RuntimeException(e); }
        return cliche;
    }
}
```

[34] Es gibt drei JMS-Methoden zum Abholen einer Nachricht aus einer Warteschlange. Dieses Beispiel benutzt `receiveNoWait()`, die sich verhält, wie ihr Name ankündigt: Die Methode gibt die Nachricht am Kopfende der Warteschlange zurück, falls vorhanden und sofort `null`, wenn die Warteschlange leer ist. Die `receives()`-Methode ist überladen. Die argumentlose Version blockiert, bis eine Nachricht zur Abholung erscheint, während die einargumentige Version lediglich für `n>0` Millisekunden blockiert, wenn die Warteschlange leer ist.

[35] JMS definiert auch das Interface *`MessageListener`*, welches die Methode `onMessage()` deklariert. Ein Nachrichtenkonsument, zum Beispiel ein Objekt vom Typ *`QueueReceiver`*, hat eine `setMessageListener()`-Methode, die eine Referenz auf ein Objekt erwartet, dessen Klasse das Interface *`MessageListener`* implementiert. Die `onMessage()`-Methode verhält sich dann wie eine Rückrufmethode, die aufgerufen wird, wenn eine Nachricht zur Verarbeitung eintrifft.

[36] In der Java-Community besteht fortdauerndes Interesse an der Zusammenarbeit zwischen JAX-WS und JMS. Es gibt beispielsweise eine Drittinitiative, um „SOAP over JMS“ zu ermöglichen. Selbst ohne diese Initiative kann ein SOAP-Envelope, verpackt im Körper einer JMS-Nachricht oder eine JMS-Nachricht, verpackt im Körper einer SOAP-Nachricht, transportiert werden. Außerdem gibt es Packages für die Umwandlung zwischen JMS und SOAP. Das Beispiel in diesem Abschnitt betont die JMS-Funktionalität zum Speichern und Weitersenden und zeigt wie sie mit SOAP-basierten Webservice kombiniert werden können.

6.6 WS-Security bei GlassFish

[37–38] Der *`Echo`*-Dienst aus Kapitel 5 zeigt, daß sich WS-Security und die Inbetriebnahme von Webservices über die statische `Endpoint`-Methode `publish()` mit Hilfe der Metro-Packages kombinieren lassen. Die Nutzung von WS-Security ist bei GlassFish einfacher, eben weil das aktuelle Metro-Release Teil der GlassFish-Distribution ist. Dieser Abschnitt dokumentiert diesen Aspekt.

Automatisches Protokollieren des HTTP-Verkehrs bei GlassFish

Zu jeder Deploymentdomain von GlassFish, zum Beispiel *domain1*, gehört ein Unterverzeichnis *config*, das verschiedene Dateien enthält, darunter *domain.xml*. Diese Konfigurationsdatei hat einen Abschnitt mit Einstellungen für die Laufzeitumgebung (JVM). Die folgenden beiden Optionen bewirken eine automatische Ausgabe des HTTP-Verkehrs von SOAP-Nachrichten und ihren Sicherheitsartefakten:

```
<jvm-options>
  -Dcom.sun.xml.ws.transport.http.HttpAdapter.dump=true
</jvm-options>
<jvm-options>
  -Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true
</jvm-options>
```

Falls der Applikationsserver bereits läuft, ist ein Neustart erforderlich, damit die geänderte Konfiguration wirksam wird. Die Ausgabe befindet sich in der Protokolldatei *domains/domain1/logs-/server.log*.

Das erste Beispiel konzentriert sich auf gegenseitige Authentifizierung mit digitalen Zertifikaten.

6.6.1 Gegenseitige Authentifizierung über digitale Zertifikate

[39] In einer typischen browserbasierten Webapplikation fordert der Webbrowser, beim Versuch eine HTTPS-Verbindung aufzubauen, den Webserver auf, sich zu authentifizieren. Bereits in Unterabschnitt 5.2.3 wurde beschrieben, daß der Webserver typischerweise keine Authentifizierung des Clients verlangt. Das Standardverhalten von Tomcat sieht beispielsweise keine Clientauthentifizierung vor. GlassFish hat im Großen und Ganzen dasselbe Standardverhalten. Das voreingestellte Verhalten von GlassFish kann unter dem Reiter „Security“ in der Administratorkonsole geändert werden, so daß der in GlassFish eingebettete Tomcat-Webserver automatisch die Authentifizierung des Clients verlangt. In diesem Beispiel soll die Authentifizierung auf der Applikationsebene im Gegensatz zur Ebene des gesamten Applikationsservers erzwungen werden. In beiden Fällen kann der aufgeforderte Teilnehmer mit digitalen Zertifikaten antworten, die seine Identität belegen. Dieser Abschnitt zeigt die gegenseitige Authentifizierung mit Zertifikaten, einen Vorgang, der unter dem Akronym *MCS (Mutual Certificates Security)* zusammengefaßt ist.

[40] Erinnern Sie sich an die Situation, daß Alice eine sichere Nachricht an Bob senden möchte und sich beide auf digitale Zertifikate verlassen, um die Identität des jeweils anderen zu bestätigen. Alice's *Keystore* enthält ihre eigenen digitalen Zertifikate, die sie Bob senden kann. Alice's *Truststore* enthält dagegen eines oder mehrere digitale Zertifikate von Bob, denen Alice vertraut. Alice's Truststore kann zuvor verifizierte Zertifikate von Bob enthalten oder Zertifikate einer Zertifizierungsstelle, welche die Echtheit von Bob's Zertifikaten bestätigt. Zum bequemen Testen können Key- und Truststore ein und dieselbe Datei sein.

[41] Es gibt zwei Möglichkeiten, um eine MCS-Applikation zu implementieren. Sie können sich direkt auf das HTTPS-Protokoll festlegen, welches die gegenseitige Authentifizierung als Teil der Initialisierungsphase beim Aufbau einer sicheren Verbindung zwischen Client und Webservice beinhaltet. Auf der Clientseite kann der Authentifizierungstyp `CLIENT-CERT` anstelle von `BASIC` oder `DIGEST` verwendet werden, so daß der Client, wie der Server, ein digitales Zertifikat zur Bestätigung seiner Identität präsentiert. Sie können aber auch vom Transportprotokoll unabhängig bleiben und die gegenseitige Authentifizierung statt dessen über die WSIT-Unterstützung von Metro erwirken. (Der Buchstabe „I“ in WSIT steht für Interoperabilität, worunter auch die Neutralität hinsichtlich des Transportprotokolls fällt.)

6.6.1.1 Authentifizierung per HTTPS

[42] Der Quelltext des Webservice' ist im Hinblick auf die Sicherheit neutral, kann also sowohl unter HTTP als auch per HTTPS deployt werden:

```
package ch06.sslWS;

import javax.ejb.Stateless;
import javax.jws.WebService;

@Stateless
@WebService
public class EchoSSL {
    public String echo(String msg) { return "Echoing: " + msg; }
}
```

Der *`EchoSSL`*-Dienst wird zuerst einmal ohne Sicherheit deployt, um zu unterstreichen, daß die Sicherung keine Änderungen am Quelltext erfordert. Die `wsimport`-Artefakte können wie gewohnt aus der ungesicherten Version generiert werden beziehungsweise per `wsgen` aus der gesicherten. Auf jeden Fall erwartet GlassFish HTTP-Verbindungen an Port 8181.

[43] Die Sicherheit wird dem *EchoSSL*-Dienst mit Hilfe der GlassFish-spezifischen Konfigurationsdatei *sun-ejb-jar.xml* „übergestülpt“:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE
    sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//
    DTD Sun ONE Application Server 8.0 EJB 2.1//EN'
    'http://www.sun.com/software/sunone/appserver/
    dtds/sun-ejb-jar_2_1-0.dtd'>
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>EchoSSL</ejb-name>
      <webservice-endpoint>
        <port-component-name>EchoSSL</port-component-name>
        <login-config>
          <auth-method>CLIENT-CERT</auth-method>
          <realm>certificate</realm>
        </login-config>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </webservice-endpoint>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

Die Sicherheitseinstellungen in dieser Konfigurationsdatei stimmen im wesentlichen mit dem `<security-constraint>`-Abschnitt des Deploymentdeskriptors *web.xml*. Die beiden interessanten Elemente sind `<login-config>` und `<transport-guarantee>`. Sie deklarieren den Authentifizierungstyp `CLIENT-CERT` beziehungsweise die Transportgarantie `CONFIDENTAL`. Die Transportgarantie fordert Fälschungssicherheit beim Transport über die Verbindung hinweg an. Die Konfigurationsdatei *sun-ejb-jar.xml* gehört in die JAR-Datei, welche den übersetzten Webservice enthält, hier *echoSSL.jar*. Der Name der JAR-Datei ist frei wählbar, während *sun-ejb-jar.xml* im Verzeichnis *META-INF* innerhalb der JAR-Datei deponiert werden muß:

```
META-INF/sun-ejb-jar.xml
ch06/sslWS/EchoSSL.class
```

Die JAR-Datei mit *sun-ejb-jar.xml* und die zustandslose EJB werden wie gewohnt in einer EAR-Datei verpackt. Wie die früheren Beispiele gezeigt haben, braucht eine EAR-Datei nicht unbedingt eine Konfigurationsdatei. Die Standardkonfigurationsdatei heißt *application.xml*. Hier die Konfigurationsdatei zu diesem Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE
    application PUBLIC "-//Sun Micro., Inc.//DTD J2EE Application 1.3//EN"
    "http://java.sun.com/dtd/application_1_3.dtd">
<application>
  <display-name>EchoSSL</display-name>
  <module>
    <ejb>echoSSL.jar</ejb>
  </module>
</application>
```

Die Datei *META-INF/application.xml* ist ein Verzeichnis und listet jede JAR-Datei in der EAR-Datei als Modul der Gesamtapplikation auf. In diesem Fall besteht die Applikation aus der zustandslosen EJB in *echoSSL.jar*. Die erzeugte EAR-Datei wird wie gewohnt durch Kopieren in das Verzeichnis *domains/domain1/autodeploy* deployt.

[44] Auch der Quelltext des Clients liefert keinen Hinweis dahingehend, daß der Client eine HTTPS-Verbindung verwenden muß:

```
import clientSSL.EchoSSLService;
import clientSSL.EchoSSL;

class EchoSSLClient {
    public static void main(String[] args) {
        try {
            EchoSSLService service = new EchoSSLService();
            EchoSSL port = service.getEchoSSLPort();
            System.out.println(port.echo("Goodbye, cruel world!"));
        }
        catch(Exception e) { System.err.println(e); }
    }
}
```

Der Quelltext des Clients ist einfach. Seine Ausführung setzt aber Informationen über Key- und Truststore voraus. GlassFish bietet Standardkey- und -truststores für die Entwicklungsphase an. Die Key- und Truststores für *domain1* liegen im Verzeichnis *domains/domain1/config* und heißen *keystore.jks* beziehungsweise *cacerts.jks*. Der Keystore enthält ein selbstunterschriebenes Zertifikat. Im produktiven Betrieb würden natürlich von einer Zertifizierungsstelle unterschriebene Zertifikate verwendet werden. Der Client wird nur mit dem GlassFish-Keystore aufgerufen, um das Testen so einfach wie möglich zu machen. Hier das zum Aufruf benötigte Kommando mit Ausgabe:

```
% java -Djavax.net.ssl.trustStore=cacerts.jks
       -Djavax.net.ssl.trustStorePassword=changeit
       -Djavax.net.ssl.keyStore=keystore.jks
       -Djavax.net.ssl.keyStorePassword=changeit EchoSSLClient
Echoing: Goodbye, cruel world!
```

Die SOAP-Nachrichten unterscheiden sich *nicht* von den über HTTP versendeten Nachrichten:

```
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:echo xmlns:ns2="http://sslWS.ch06/">
      <arg0>Goodbye, cruel world!</arg0>
    </ns2:echo>
  </S:Body>
</S:Envelope>
```

Die Sicherheit zeigt sich nicht auf der SOAP-Ebene, da sie bereits in der Transportebene implementiert ist, das heißt durch HTTPS. Nur die Konfigurationsdateien enthalten einen Hinweis darauf, daß MCS im Spiel ist. Beim folgenden WSIT-Beispiel verändern sich die SOAP-Nachrichten dagegen erheblich, daß die Sicherheit auf der SOAP-Ebene stattfindet.

6.6.1.2 Authentifizierung per WSIT

[45] Der Client ist nun zur Abwechselung ein Servlet, welches eine Texteingabe von einem Webbrowser oder etwas ähnlichem erhält und die `echo()`-Methode des `EchoService`-Dienstes mit diesem Argument aufruft. Das Servlet antwortet dem Webbrowser mit dem Rückgabewert der `echo()`-Methode. HTTP kommt nur im Rahmen der Verbindung zwischen dem clientseitigen Webbrowser und dem Servletcontainer ins Spiel. Innerhalb des Servletcontainers tauschen der `EchoService`-Dienst und das `EchoClient`-Servlet SOAP-Nachrichten aus, aber *nicht* über HTTP ([Abbildung ??]).

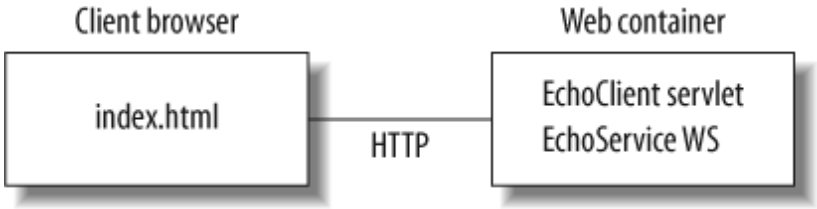

Abbildung 6.3: ...

[46] Die gegenseitige Authentifizierung per WSIT wird mit Hilfe von Konfigurationsdateien in die Applikation eingebracht. Diese Konfigurationsdateien sind kompliziert genug, um sie mit Hilfe eines Werkzeuges zu generieren. Die *NetBeans-IDE* eignet sich hierfür, da sie sowohl die Konfigurationsdateien als auch die entsprechenden Ant-Skripte für Übersetzung, Verpacken und Deployment liefert. Es soll nochmals darauf hingewiesen werden, daß diese und jede andere WSIT-Applikation auch ohne NetBeans entwickelt werden kann.

[47] Der Webservice ist nun servlet- und nicht EJB-basiert, obwohl diese Änderung selbst für die gegenseitige Authentifizierung keine Rolle spielt:

```
package ch06.mcs;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Echo {
    @WebMethod
    public String echo(String msg) { return "Echoing: " + msg; }
}
```

Der Client ist ein Servlet, welches sich auf die üblichen per `wsimport` erzeugten Artefakte stützt:

```
package ch06.mcs.client;

import java.io.PrintWriter;
import java.io.IOException;
import java.io.Closeable;
import javax.annotation.Resource;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;
import javax.xml.ws.soap.SOAPFaultException;

public class EchoClient extends HttpServlet {
    @WebServiceRef(wsdlLocation = "http://localhost:8080/echoMCS/EchoService?wsdl")
    public EchoService service;

    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        res.setContentType("text/html;charset=UTF-8");
        PrintWriter out = null;
        try {
            out = res.getWriter();
            out.println("<h3>EchoServlet: " + req.getContextPath() + "</h3>");

            // Get the port reference and invoke echo method.
            Echo port = service.getEchoPort();
            String result = port.echo(req.getParameter("msg"));
```

```
            // If there's no SOAP fault so far, authentication worked.
            out.println("<h3>Authentication OK</h3><br />");
            out.println(result);
            out.flush();
            ((Closeable) port).close(); // close connection to service
        }
        catch (SOAPFaultException e) {
            out.println("Authentication failure: " + e);
        }
        catch (Exception e) { out.println(e); }
        finally { out.close(); }
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res) {
        try {
            this.doGet(req, res); // shouldn't happen but just in case :)
        }
        catch(Exception e) { throw new RuntimeException("doPost"); }
    }
}
```

Der Servletcontainer beachtet die hervorgehobene Annotation `@WebServiceRef` und initialisiert die Referenzvariable `service` vom Typ `EchoService`. Der Client `EchoClient` schließt den Port nachdem die `echo()`-Operation des Webservice' aufgerufen und die Antwort an den Webbrowser zurückgesendet wurde, um dem Container anzuzeigen, daß bezüglich dieser Anfrage kein weiterer Kontakt mit dem Webservice erforderlich ist.

[48] Das `EchoClient`-Servlet wird jedesmal beim Abschicken des einfachen HTML-Formulars aufgerufen, in das der Client einen zu wiederholenden Text einträgt. Der Vollständigkeit halber hier das Formular:

```
<html>
  <head />
  <body>
    <h3>EchoService Client</h3>
    <form action = 'EchoServlet' method = 'GET'><br />
      <input type = 'text' name = 'msg' /><br />
      <p><input type = 'submit' value = 'Send message' /></p>
    </form>
  </body>
</html>
```

Bis jetzt gibt es noch keine Anzeichen dafür, daß gegenseitige Authentifizierung im Spiel ist. Die Konfigurationsdatei alleine praktiziert die gegenseitige Authentifizierung in die Applikation. Sowohl der `EchoService`-Dienst als auch das `EchoClient`-Servlet haben eine kleine *sun-web.xml*-Konfigurationsdatei, die ein Kontextwurzelverzeichnis für den Container deklariert, das heißt einen Namen für die deployte WAR-Datei. Die folgende *sun-web.xml*-Datei gilt für `EchoClient`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Application Server 9.0 Servlet 2.5//EN"
    "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
  <context-root>/echoClient</context-root>
</sun-web-app>
```

Die *sun-web.xml*-Dateien liegen im *WEB-INF*-Verzeichnis der deployten WAR-Dateien. Es gibt noch eine zweite Konfigurationsdatei.

[49] Der Name der Hauptkonfigurationsdatei für die gegenseitige Authentifizierung zwischen dem Webservice und seinen Clients hat den Präfix *wsit-*: *wsit-ch06.mcs.Echo.xml* für den Webservice und *wsit-client.xml* für den Client. Jede Datei hat die Struktur eines WSDL-Dokuments. (Die clientseitige Konfigurationsdatei *wsit-client.xml* importiert allerdings weitere Dokumente, die den größten Teil der Einstellungen enthalten.) Die serviceseitige Konfigurationsdatei liegt im Verzeichnis *WEB-INF*, die clientseitige im Verzeichnis *WEB-INF/classes/META-INF* . Hier ist die serviceseitige Konfigurationsdatei *wsit-ch06.mcs.Echo.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    name="EchoService"
    targetNamespace="http://mcs.ch06/"
    xmlns:tns="http://mcs.ch06/"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
      oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
    xmlns:sc="http://schemas.sun.com/2006/03/wss/server"
    xmlns:wspp="http://java.sun.com/xml/ns/wsit/policy" >

  <message name="echo" />
  <message name="echoResponse" />

  <portType name="Echo">
        <wsdl:operation name="echo">
      <wsdl:input message="tns:echo" />
      <wsdl:output message="tns:echoResponse" />
    </wsdl:operation>
  </portType>

  <binding name="EchoPortBinding" type="tns:Echo">
    <wsp:PolicyReference URI="#EchoPortBindingPolicy" />
    <wsdl:operation name="echo">
      <wsdl:input>
        <wsp:PolicyReference URI="#EchoPortBinding_echo_Input_Policy" />
      </wsdl:input>
      <wsdl:output>
        <wsp:PolicyReference URI="#EchoPortBinding_echo_Output_Policy" />
      </wsdl:output>
    </wsdl:operation>
  </binding>

  <service name="EchoService">
    <wsdl:port name="EchoPort" binding="tns:EchoPortBinding" />
  </service>

  <wsp:Policy wsu:Id="EchoPortBindingPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <wsaws:UsingAddressing
          xmlns:wsaws="http://www.w3.org/2006/05/addressing/wsdl" />
        <sp:SymmetricBinding>
          <wsp:Policy>
            <sp:ProtectionToken>
```

```
            <wsp:Policy>
              <sp:X509Token
                sp:IncludeToken="http://schemas.xmlsoap.org/ws/
                  2005/07/securitypolicy/IncludeToken/Never">
                <wsp:Policy>
                  <sp:WssX509V3Token10 />
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:ProtectionToken>
          <sp:Layout>
            <wsp:Policy>
              <sp:Strict />
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp />
          <sp:OnlySignEntireHeadersAndBody />
          <sp:AlgorithmSuite>
                      <wsp:Policy>
              <sp:Basic128 />
            </wsp:Policy>
          </sp:AlgorithmSuite>
        </wsp:Policy>
      </sp:SymmetricBinding>
      <sp:Wss11>
        <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier />
          <sp:MustSupportRefIssuerSerial />
          <sp:MustSupportRefThumbprint />
          <sp:MustSupportRefEncryptedKey />
        </wsp:Policy>
      </sp:Wss11>
      <sp:SignedSupportingTokens>
        <wsp:Policy>
          <sp:UsernameToken
            sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/
            securitypolicy/IncludeToken/AlwaysToRecipient">
            <wsp:Policy>
              <sp:WssUsernameToken10 />
            </wsp:Policy>
          </sp:UsernameToken>
        </wsp:Policy>
      </sp:SignedSupportingTokens>
      <sc:KeyStore wspp:visibility="private" alias="xws-security-server" />
    </wsp:All>
      </wsp:ExactlyOne>

</wsp:Policy>
<wsp:Policy wsu:Id="EchoPortBinding_echo_Input_Policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:EncryptedParts>
        <sp:Body />
      </sp:EncryptedParts>
      <sp:SignedParts>
        <sp:Body />
        <sp:Header Name="To"
            Namespace="http://www.w3.org/2005/08/addressing" />
```

```
            <sp:Header Name="From"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="FaultTo"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="ReplyTo"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="MessageID"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="RelatesTo"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="Action"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="AckRequested"
                Namespace="http://schemas.xmlsoap.org/ws/2005/02/rm" />
            <sp:Header Name="SequenceAcknowledgement"
                Namespace="http://schemas.xmlsoap.org/ws/2005/02/rm" />
            <sp:Header Name="Sequence"
                Namespace="http://schemas.xmlsoap.org/ws/2005/02/rm" />
          </sp:SignedParts>
        </wsp:All>
      </wsp:ExactlyOne>
    </wsp:Policy>

    <wsp:Policy wsu:Id="EchoPortBinding_echo_Output_Policy">
      <wsp:ExactlyOne>
        <wsp:All>
          <sp:EncryptedParts>
            <sp:Body />
          </sp:EncryptedParts>
          <sp:SignedParts>
            <sp:Body />
            <sp:Header Name="To"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="From"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="FaultTo"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="ReplyTo"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="MessageID"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="RelatesTo"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="Action"
                Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="AckRequested"
                Namespace="http://schemas.xmlsoap.org/ws/2005/02/rm" />
            <sp:Header Name="SequenceAcknowledgement"
                Namespace="http://schemas.xmlsoap.org/ws/2005/02/rm" />
            <sp:Header Name="Sequence"
                Namespace="http://schemas.xmlsoap.org/ws/2005/02/rm" />
          </sp:SignedParts>
        </wsp:All>
      </wsp:ExactlyOne>
    </wsp:Policy>

  </definitions>
```

Der Detailgrad ist überwältigend. Von Interesse ist hier, daß viele dieser Details zu Abschnitten des SOAP-Headers gehören, der verschlüsselt und digital unterschrieben sein muß. Auf jeden Fall wird ein so kompliziertes XML-Dokument wie dieses besser mit einem Werkzeug wie der Netbeans-IDE erzeugt, als von Hand geschrieben.

[50] Die clientseitige Konfigurationsdatei *wsit-client.xml* (im Verzeichnis *WEB-INF/classes/META-INF*) ist weniger kompliziert:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
      oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://mcs.ch06/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://mcs.ch06/"
    name="EchoService"
    xmlns:sc="http://schemas.sun.com/2006/03/wss/client"
    xmlns:wspp="http://java.sun.com/xml/ns/wsit/policy">
  <wsp:UsingPolicy></wsp:UsingPolicy>
  <types>
    <xsd:schema>
      <xsd:import
        namespace="http://mcs.ch06/"
        schemaLocation="http://localhost:8080/echoMCS/EchoService?xsd=1">
      </xsd:import>
    </xsd:schema>
  </types>
  <message name="echo">
    <part name="parameters" element="tns:echo"></part>
  </message>
  <message name="echoResponse">
    <part name="parameters" element="tns:echoResponse"></part>
  </message>
  <portType name="Echo">
    <operation name="echo">
      <input message="tns:echo"></input>
      <output message="tns:echoResponse"></output>
    </operation>
  </portType>
  <binding name="EchoPortBinding" type="tns:Echo">
    <wsp:PolicyReference URI="#EchoPortBindingPolicy" />
    <soap:binding
        transport="http://schemas.xmlsoap.org/soap/http"
        style="document"></soap:binding>
    <operation name="echo">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
  </binding>
```

```
  <service name="EchoService">
    <port name="EchoPort" binding="tns:EchoPortBinding">
      <soap:address location="http://localhost:8080/echoMCS/EchoService">
      </soap:address>
    </port>
  </service>
  <wsp:Policy wsu:Id="EchoPortBindingPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sc:CallbackHandlerConfiguration wspp:visibility="private">
          <sc:CallbackHandler default="wsitUser" name="usernameHandler" />
          <sc:CallbackHandler default="changeit" name="passwordHandler" />
        </sc:CallbackHandlerConfiguration>
        <sc:TrustStore wspp:visibility="private" peeralias="xws-security-server"
/>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
</definitions>
```

Der hervorgehobene Abschnitt gibt den Benutzernamen, hier `wsitUser` und das Paßwort für den Zugriff auf den dateibasierten Sicherheitsadministrationsbereich des Servers an. Das GlassFish-Kommando:

```
% asadmin list-file-users
```

liefert eine Liste aller autorisierten Benutzer. Die Ausgabe sollte `wsitUser` enthalten, damit die gegenseitige Authentifizierung funktioniert.

[51] Der servletbasierte `EchoService`-Dienst und sein Servletclient `EchoClient` werden in separate WAR-Dateien verpackt und wie gewohnt deployt. Zunächst die Ausgabe des WAR-Datei des Dienstes, *echoMCS.war* (die erste Spalte links gibt die Dateigröße in Bytes an):

```
   71 META-INF/MANIFEST.MF
 2488 WEB-INF/web.xml
 2506 WEB-INF/sun-web.xml
 6704 WEB-INF/wsit-ch06.mcs.Echo.xml
  558 WEB-INF/classes/ch06/mcs/Echo.class
```

Obwohl die Konfigurationsdatei für die gegenseitige Authentifizierung die Struktur eines WSDL-Dokumentes hat, generiert GlassFish das für Clients verfügbare WSDL-Dokument in der üblichen Weise. Das WSDL-Dokument stimmt im wesentlichen mit *wsit-ch06.mcs.Echo.xml* überein, aber die Dateien unterscheiden sich in einen geringfügigen Details.

[52] Nun die Ausgabe der WAR-Datei des Clients, *echoClient.war*:

```
   71 META-INF/MANIFEST.MF
  698 WEB-INF/web.xml
  305 WEB-INF/sun-web.xml
  745 WEB-INF/classes/ch06/mcs/client/Echo.class
 1583 WEB-INF/classes/ch06/mcs/client/ObjectFactory.class
  646 WEB-INF/classes/ch06/mcs/client/Echo_Type.class
  734 WEB-INF/classes/ch06/mcs/client/EchoResponse.class
  237 WEB-INF/classes/ch06/mcs/client/package-info.class
 1484 WEB-INF/classes/ch06/mcs/client/EchoService.class
 2042 WEB-INF/classes/ch06/mcs/client/EchoClient.class
  381 WEB-INF/classes/META-INF/wsit-client.xml
 2590 WEB-INF/classes/META-INF/EchoService.xml
```

Die WAR-Datei enthält die per `wsimport` generierten Klassen. Die Datei *wsit-client.xml* verrichtet etwas mehr Arbeit, als *EchoService.xml* zu importieren.

6.6.2 Der dramatische SOAP-Envelope

[53] Die gegenseitige Authentifizierung bewirkt vergleichsweise lange und komplizierte SOAP-Envelopes, eben weil die Sicherheit nun auf der Ebene der SOAP-Nachrichten statt auf der Übertragungsebene liegt. Der SOAP-Envelope beinhaltet bei gegenseitiger Authentifizierung Informationen über Verschlüsselungsalgorithmen, Message-Digest-Algorithmen, digitale Signaturverfahren, Sicherheitsmerkmale, Zeitstempel und so weiter. Alle diese Informationen stehen innerhalb eines SOAP-Headerblocks, dessen `mustUnderstand`-Attribut mit `true` bewertet ist. Alle Mittler und der endgültige Empfänger müssen entweder in der Lage sein, den Headerblock in einer zweckdienlichen Weise zu verarbeiten oder einen SOAP-Fehler auswerfen. Hier, um der dramatischen Wirkung willen, der SOAP-Envelope einer Anfrage des Clients `EchoClient` an den `EchoService`:

```
<?xml version="1.0" ?>
<S:Envelope
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
      oasis-200401-wss-wssecurity-secext-1.0.xsd"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
    oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:exc14n="http://www.w3.org/2001/10/xml-exc-c14n#">

  <S:Header>
    <To xmlns="http://www.w3.org/2005/08/addressing" wsu:Id="_5006">
      http://localhost:8080/echoMCS/EchoService
    </To>
    <Action xmlns="http://www.w3.org/2005/08/addressing"
      wsu:Id="_5005">http://mcs.ch06/Echo/echoRequest
    </Action>
    <ReplyTo xmlns="http://www.w3.org/2005/08/addressing" wsu:Id="_5004">
      <Address>http://www.w3.org/2005/08/addressing/anonymous</Address>
    </ReplyTo>
    <MessageID xmlns="http://www.w3.org/2005/08/addressing"
      wsu:Id="_5003">uuid:b85e5024-85c6-4482-b118-3d00d8ebff17
    </MessageID>
    <wsse:Security S:mustUnderstand="1">
      <wsu:Timestamp
        xmlns:ns10="http://www.w3.org/2003/05/soap-envelope"
        xmlns:ns11="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
        wsu:Id="_3">
        <wsu:Created>2008-08-14T02:06:32Z</wsu:Created>
        <wsu:Expires>2008-08-14T02:11:32Z</wsu:Expires>
      </wsu:Timestamp>
      <xenc:EncryptedKey
          xmlns:ns10="http://www.w3.org/2003/05/soap-envelope"
          xmlns:ns11="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
          Id="_5002">
        <xenc:EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p" />
          <ds:KeyInfo
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:type="keyInfo">
```

```
        <wsse:SecurityTokenReference>
          <wsse:KeyIdentifier
              ValueType="http://docs.oasis-open.org/wss/2004/
                01/oasis-200401-wss-x509-token-profile-1.0
                #X509SubjectKeyIdentifier"
              EncodingType="http://docs.oasis-open.org/wss/2004/
                01/oasis-200401-wss-soap-message-security-1.0
                #Base64Binary">
          dVE29ysyFW/iD1la3ddePzM6IWo=
        </wsse:KeyIdentifier>
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>UIz4WopejXJGmw2yg0M0pIf8hEomI7vI...</xenc:CipherValue>
      </xenc:CipherData>
  </xenc:EncryptedKey>
  <xenc:ReferenceList
    xmlns:ns17="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
    xmlns:ns16="http://www.w3.org/2003/05/soap-envelope"
    xmlns="">
    <xenc:DataReference URI="#_5008"></xenc:DataReference>
    <xenc:DataReference URI="#_5009"></xenc:DataReference>
  </xenc:ReferenceList>
  <xenc:EncryptedData
    xmlns:ns10="http://www.w3.org/2003/05/soap-envelope"
    xmlns:ns11="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
    Type="http://www.w3.org/2001/04/xmlenc#Element"
    Id="_5009">
    <xenc:EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
      <ds:KeyInfo
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:type="keyInfo">
      <wsse:SecurityTokenReference>
        <wsse:Reference
          ValueType="http://docs.oasis-open.org/wss/
            oasis-wss-soap-message-security-1.1
            #EncryptedKey"
            URI="#_5002" />
      </wsse:SecurityTokenReference>
    </ds:KeyInfo>
    <xenc:CipherData>
      <xenc:CipherValue>0GCRGwFKlLfnRYnQd...</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
  <ds:Signature
      xmlns:ns10="http://www.w3.org/2003/05/soap-envelope"
      xmlns:ns11="http://docs.oasis-open.org/ws-sx/
        ws-secureconversation/200512"
      Id="_1">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
        <exc14n:InclusiveNamespaces PrefixList="wsse S" />
      </ds:CanonicalizationMethod>
      <ds:SignatureMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1" />
```

```
<ds:Reference URI="#_5003">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <exc14n:InclusiveNamespaces PrefixList="S" />
    </ds:Transform>
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>NI9i+HGoWeYAsu8K1eOcmmSn+SY=</ds:DigestValue>
</ds:Reference>

<ds:Reference URI="#_5004">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <exc14n:InclusiveNamespaces PrefixList="S" />
    </ds:Transform>
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>5Ab1ebo4/FraGgck/A8iDx1J9+I=</ds:DigestValue>
</ds:Reference>

<ds:Reference URI="#_5005">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <exc14n:InclusiveNamespaces PrefixList="S" />
    </ds:Transform>
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>Qso/D/tFg2kzZnb0J7tOzqRW84M=</ds:DigestValue>
</ds:Reference>

<ds:Reference URI="#_5006">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <exc14n:InclusiveNamespaces PrefixList="S" />
    </ds:Transform>
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>DQsOAHfFqRDBiV4MqOLwbMRLXcc=</ds:DigestValue>
</ds:Reference>

<ds:Reference URI="#_5007">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <exc14n:InclusiveNamespaces PrefixList="S" />
    </ds:Transform>
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>9vpXDjjwI7bLNBAVe5n1jcpHou4=</ds:DigestValue>
</ds:Reference>

<ds:Reference URI="#_3">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <exc14n:InclusiveNamespaces PrefixList="wsu wsse S" />
    </ds:Transform>
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>9NwWEZNcMLby0fEQlrwbJ6fVGQA=</ds:DigestValue>
</ds:Reference>
```

```
          <ds:Reference URI="#uuid_e2da395f-b9bc-4d52-9cb8-57bafe97ac25">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                <exc14n:InclusiveNamespaces PrefixList="wsu wsse S" />
              </ds:Transform>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>nRSRynHPET8TPA4DvAR9iB6OG1E=</ds:DigestValue>
          </ds:Reference>

        </ds:SignedInfo>
        <ds:SignatureValue>QDgHtRo7NYLsmzKIPDd5RZ/a7hk=</ds:SignatureValue>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference
              wsu:Id="uuid_5b05ed00-1333-49c3-9f03-0225ea41d3da">
            <wsse:Reference
                ValueType="http://docs.oasis-open.org/wss/
                  oasis-wss-soap-message-security-1.1#EncryptedKey"
                URI="#_5002" />
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </S:Header>
  <S:Body wsu:Id="_5007">
    <xenc:EncryptedData
        xmlns:ns10="http://www.w3.org/2003/05/soap-envelope"
        xmlns:ns11="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
        Type="http://www.w3.org/2001/04/xmlenc#Content" Id="_5008">
      <xenc:EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
      <ds:KeyInfo
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:type="keyInfo">
        <wsse:SecurityTokenReference>
          <wsse:Reference
              ValueType="http://docs.oasis-open.org/wss
                  /oasis-wss-soap-message-security-1.1#EncryptedKey"
              URI="#_5002" />
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>
          p2TQL4JqgBCWh9Jiv6PWikJ0beMuNDvj1wH...
        </xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </S:Body>
</S:Envelope>
```

[54] Eine Zusammenfassung der Schritte hinter den Kulissen durch die diese SOAP-Anfrage entstanden ist, sollte helfen, das Dokument zu erklären:

- GlassFish beinhaltet einen Secure Token Service (STS), welcher im Zentrum der gegenseitigen Authentifizierung zwischen dem `EchoService`-Dienst und seinem Client `EchoClient` steht. Der STS nimmt Anfragen nach Sicherheitsmerkmalen entgegen. Wird die Anfrage anerkannt, so sendet der STS ein *SAML*-Merkmal (Security Assertion Markup Language) an den Client zurück. SAML ist ein XML-Dialekt für Sicherheitserklärungen. Der Client kann dieses

Merkmal verwenden, um sich bei einem Webservice zu authentifizieren. In diesem Beispiel verlangt das `EchoClient`-Servlet einen registrierten Benutzernamen (`wsitUser`), damit der GlassFish-STS ein SAML-Token ausgibt. Dieser erste Schritt stiftet Vertrauen zwischen dem Servletclient, dem STS und dem Webservice.

- Servletclient und Webservice tauschen X.509-Zertifikate in einem gegenseitigen Aufforderungs- oder Authentifizierungsvorgang aus. Denken Sie daran, daß diese Zertifikate eine digitale Unterschrift tragen, die typischerweise von einer Zertifizierungsstelle erteilt wird. Im Testbetrieb können die Zertifikate selbst unterschrieben werden. Das X.509-Zertifikat enthält auch den öffentlichen Schlüssel aus dem Schlüsselpaar des Besitzers. Die digitale Unterschrift hat die Aufgabe, die Identität des öffentlichen Schlüssels zu bestätigen. Dieser zweite Schritt gewährleistet sichere Kommunikation zwischen Client und Webservice, da jede Seite nun der Identität auf der Gegenseite vertraut.

- Die zwischen Client und Server ausgetauschten Daten stehen in SOAP-Envelopes, die zur Geheimhaltung selbstverständlich verschlüsselt werden. In der obigen SOAP-Anfrage sind die verschlüsselten Daten hervorgehoben und befinden sich zwischen `<xenc:CipherData>`-Tags.

Die Komplexität ist unter der Voraussetzung sinnvoll, daß sich WSIT im Hinblick auf die umfassende Sicherheit nicht auf die Übertragungsebene (inbesondere die Anwendung von HTTPS) verlassen kann. Statt dessen muß WSIT die SOAP-Nachrichten selbst verwenden, um Vertrauen, sichere Kommunikation und die Geheimhaltung der Daten zwischen Client und Server zu unterstützen. Dieses Beispiel drives home the point, da der Servletclient und der Webservice nicht über HTTPS kommunizieren.

6.7 Vorteile des Deployments über einen Java-Applikationsserver

[55] Die statische `Endpoint`-Methode `publish()` ist ein leichtgewichtiges Mittel, um Java-Webservices in Betrieb zu nehmen. Ein eigenständiger Webcontainer wie Tomcat ist ein mittelgewichtiges und ein Applikationsserver wie GlassFish ein schwergewichtiges Mittel zu diesem Zweck. Für alle Software gilt: *schwergewichtig* bedeutet *kompliziert*. Es ist daher eventuell nützlich, dieses Kapitel mit einer Rückschau zu beenden, unter welchen Umständen sich ein vollausgestatteter Applikationsserver, trotz der mit einer solchen Entscheidung einhergehenden Probleme empfiehlt. Hier ist eine kurze Zusammenfassung der Vorteile, die ein Applikationsserver beim Deployment von Webservices bietet, gleichsam SOAP-basiert oder im REST-Stil:

- Ein Webservice kann als zustandslose EJB implementiert werden. Der EJB-Container bietet Vorteile, insbesondere Threadsicherheit und containergesteuerte Transaktionen, die weder die abstrakte Klasse `Endpoint` noch ein Webcontainer in dieser Form zur Verfügung stellen. Ein Webservice auf Produktionsniveau benötigt voraussichtlich die Datenhaltung durch `@Entity`-Klassen. *`EntityManager`*-Methoden wie `find()` und `persist()` sind nicht von Natur aus threadsicher, bekommen diese Eigenschaft aber, wenn die Implementierung des Interface' von einem Feld einer EJB referenziert wird.

- Ein Java-Applikationsserver bietet typischerweise eine Webbrowserschnittstelle zur Einsichtnahme in das WSDL-Dokument und zum Testen jeder mit `@WebMethod` annotierten Methode. GlassFish bietet diese und weitere Entwicklungswerkzeuge in der Administratorkonsole.

- Ein Applikationsserver umfaßt ein RDBMS, bei GlassFish die Java DB. Andere Datenbanksysteme können bei Bedarf angeschlossen werden.

- Eine in einem Applikationsserver deployte Enterprise-Applikation kann eine beliebige Kombination von Komponenten enthalten, darunter interaktive servletbasierte Webapplikationen,

EJBs, Nachrichtenthemen und -warteschlangen sowie Ereignisbehandler, `@Entity`-Klassen und Webservices. Alle Komponenten in einer EAR-Datei teilen sich einen ENC (Enterprise Naming Context), wodurch JNDI-Anfragen vereinfacht werden.

- Applikationsserver bieten typischerweise umfangreiche administrative Unterstützung, insbesondere zum Protokollieren und zur Fehlersuche. Die Administratorkonsole von GlassFish ist ein Beispiel hierfür.
- GlassFish ist die Referenzimplementierung des Java-Applikationsservers und beinhaltet Metro mit aller begleitenden Unterstützung für die Interoperabilität von Webservices.
- Ein deployter Webservice kann, wie jede andere Enterprise-Komponente, Teil einer clustered Applikation sein.

6.8 Ausblick

[56] Der Streit über SOAP-basierte Webservices und Dienste im REST-Stil wird häufig angefeuert. Jeder Ansatz hat seine Vorzüge und repräsentiert ein nützliches Werkzeug im Werkzeugkasten des modernen Programmierers. Die meisten SOAP-basierten Dienste bauen auf HTTP oder HTTPS auf und sind daher ein Sonderfall des REST-Stils, wie [frühere Beispiele] gezeigt haben Es besteht keine Notwendigkeit, eine Trennlinie zwischen SOAP und REST zu ziehen. Das nächste und letzte Kapitel wirft einen kurzen Blick auf die jüngere Vergangenheit der verteilten Systeme, um eine deutlichere Sicht auf die Wahlmöglichkeiten in der Entwicklung von Webservices zu erhalten.

Kapitel 7

Beyond the Flame Wars

Inhaltsübersicht

[0] Die Auseinandersetzung um die Ansätze SOAP und REST in der Webservice-Community ist oft hitzig und gelegentlich unmäßig, da die Befürworter der einen Richtung deren Vorteile übermäßig anpreisen und gegen die Nachteile der anderen Richtung wettern. Worin der Reiz des SOAP/REST-Streits auch immer bestehen mag, können beide Ansätze friedlich und produktiv koexistieren. Es gibt keine harte Wahl, entweder der eine Ansatz oder der andere. Beide Ansätze haben ihren individuellen Reiz und jeder von beiden ist besser als die älteren Ansätze für verteilte Softwaresysteme. Ein kurzer Blick auf die Geschichte der verteilten Systeme ist eine Möglichkeit, sich Einblick in das Thema zu verschaffen.

7.1 Die Geschichte des Webservice-Konzeptes

[1] Webservices haben sich aus dem RPC-Mechanismus in DCE (Distributed Computing Environment) entwickelt, einem Framework für Softwareentwicklung aus den frühen 1990er Jahren. DCE enthielt ein verteiltes Dateisystem (DCE/DFS) und ein Authentifizierungssystem auf der Basis von Kerberos. Obwohl der Ursprung von DCE in der Unix-Welt liegt, lieferte Microsoft schnell eine eigene Implementierung namens *MSRPC* (Microsoft RPC), die wiederum als Infrastruktur für die Interprozeßkommunikation bei Windows diente. Die COM/OLE-Technologien und -Dienste (Component Object Model/Object Linking and Embedding) von Microsoft bauten auf dem DCE/RPC-Fundament auf. Die Frameworks für Systeme verteilter Objekte der ersten Generation, CORBA (Common Object Request Broker Architecture) und DCOM (Distributed Component Object Model) von Microsoft, sind im prozeduralen DCE/RPC-Framework verankert. Auch Java RMI leitet sich von DCE/RPC ab und die Methodenaufrufe in den ursprünglichen EJB-Typen, Sitzung und Entität, sind RMI-Aufrufe. Die Java Enterprise Edition und das .NET-Framework von Microsoft, die Frameworks für Systeme verteilter Objekte der zweiten Generation, leiten ihre Abstammung ebenfalls aus DCE/RPC her. Zahlreiche gängige Systemhilfsmittel (zum Beispiel der Samba Datei- und Druckdienst für Windows-Clients) bauen auf DCE/RPC auf.

[2] DCE/RPC hat die vertraute Client/Server-Architektur bei der ein Client eine Prozedur aufruft, die auf dem Server ausgeführt wird. Argumente können vom Client an den Server und Rückgabewerte vom Server an den Client übergeben werden. Das Framework ist im Prinzip sprachneutral, obwohl in der Praxis stark in Richtung C/C++ voreingenommen. DCE/RPC beinhaltet Kommandos zum Erzeugen client- und serverseitiger Artefakte (Stubs beziehungsweise Skeletons) sowie Softwarebibliotheken, welche die Transportdetails verbergen. Wir konzentrieren uns nun auf das IDL-Dokument (Interface Definition Language), welches als Dienstkontrakt und Eingabe für die Kommandos dient, die Artefakte zur Unterstützung von DCE/RPC-Aufrufen erzeugt.

7.1.1 Dienstkontrakte bei DCE/RPC

[3] Hier eine einfache IDL-Datei:

```
/* echo.idl */
[uuid(2d6ead46-05e3-11ca-7dd1-426909beabcd), version(1.0)]
interface echo
{
    const long int ECHO_SIZE = 512;
    void echo(
        [in] handle_t h,
        [in, string] idl_char from_client[],
        [out, string] idl_char from_server[ECHO_SIZE]
    );
}
```

Das durch die maschinell erzeugte UUID identifierte Interface deklariert eine einzelne Funktion mit drei Parametern. Die beiden `in`-Parameter werden beim Aufruf der entfernten Prozedur übergeben, der `out`-Parameter ist der Rückgabewert der Prozedur. Der erste Parameter, vom definierten Typ `handle_t` ist ein Pflichtparameter und verweist auf eine RPC-Bindingstruktur. Die `echo()`-Funktion could but does not return a value, da die wiederholte Zeichenkette statt dessen als `out`-Parameter zurückgegeben wird. Die IDL-Datei ähnelt einem WSDL-Dokument, obwohl sie offenbar kein XML-Dokument ist.

[4] Die IDL-Datei wird einem IDL-Compiler übergeben (bei Windows zum Beispiel dem `midl`-Kommando), um die entsprechenden client- beziehungsweise serverseitigen Artefakte zu generieren. Der folgende C-Client ruft die entfernte `echo()`-Funktion auf:

```
#include <stdio.h>   /* standard C header file */
#include <dce/rpc.h> /* DCE/RPC header file */

int main(int argc, char* argv[]) {
  /* DCE/RPC data types */
  rpc_ns_handle_t import_context;
  handle_t        binding_h;
  error_status_t  status;
  idl_char        reply[ECHO_SIZE + 1];

  char* msg = "Hello, world!";

  /* Set up RPC */
  rpc_ns_binding_import_begin(rpc_c_ns_syntax_default,
                              (unsigned_char_p_t) argv[1], /* server id*/
                              echo_v1_0_c_ifspec,
                              0,
                              &import_context,
                              &status);
```

```
    check_status_maybe_die(status, "import_begin"); /* code not shown */

    rpc_ns_binding_import_next(import_context,
                               &binding_h,
                               &status);

    check_status_maybe_die(status, "import_next");

    /* Make the remote call. */
    echo(binding_h, (idl_char*) msg,reply);
    printf("%s echoed from server: %s\n", msg, reply);

    return 0;

  }
```

Die Initialisierung zu Anfang ist ein wenig einschüchternd, der Aufruf dagegen einfach.

[5] Die IDL-Datei spielt eine zentrale Rolle in der ActiveX-Technologie von Microsoft. Ein ACticeX-Steuerelement ist im Grunde genommen eine DLL (Dynamic Link Library) mit einer eingebetteten typelib, die wiederum eine übersetzte IDL-Datei ist. Ein ActiveX-Steuerelement ist somit eine DLL, mit eigener RPC-Schnittstelle. Das Steuerelement kann also dynamisch in eine host application (zum Beispiel einen Browser) eingebettet werden, der die typelib konsultiert, um zu ermitteln, wie die in der DLL implementierten Funktionen aufgerufen werden müssen. Dies ist eine Vorschau dafür, wie ein Client eines SOAP-basierten Webservice' ein WSDL-Dokument auswertet.

[6] DCE/RPC ist ein wichtiger Schritt in die Richtung sprach-agnostischer Systeme, hat allerdings einige offensichtliche, if understandable Minuspunkte. DCE/RPC verwendet zum Beispiel einen proprietären Transportmechanismus, was bei einer bahnbrechenden Technologie verständlich ist, die HTTP um rund zehn Jahre vorausging. Ich will DCE/RPC nicht einschätzen, sondern die Ähnlichkeiten zwischen DCE/RPC und seinen Nachfolgern, den SOAP-basierten Webservices, zu beleuchten. XML-RPC ist der nächste SOAP-Vorgänger, den es zu betrachten gilt.

7.1.2 XML-RPC

[7] XML-RPC wurde in den späten 1990er Jahren von Dave Winer bei *UserLand Software* entwickelt und ist ein sehr leichtgewichtigtes RPC-System mit Unterstützung elementarer Datentypen (im Wesentlichen die eingebauten C-Typen, einem booleschen Typ und einem Typ für Datums- und Zeit-Werte) sowie einigen elementaren Kommandos. Die ursprüngliche Spezifikation umfaßt sieben Seiten. Die beiden Schlüsseleigenschaften sind XML-Marshalling/Unmarshalling, um Sprachunabhängigkeit zu erreichen und das Vertrauen auf HTTP (später auch SMTP) als Transportprotokoll. Der open-wire Meerkat-Dienst von O'Reilly ist eine XML-RPC-Applikation.

[8] XML-RPC gehorcht als RPC-Technologie dem Nachrichtenaustauschmuster Request/Response. Hier ist die XML-Anfrage eines Aufrufs der Fibonaccifunktion mit dem Argument 11, übergeben als 4-Byte langer ganzzahliger Wert:

```
<?xml version="1.0"?>
<methodCall>
    <methodName>ch07.fib</methodName>
    <params>
        <param>
            <value><i4>11</i4></value>
        </param>
    </params>
</methodCall>
```

XML-RPC ist absichtlich sehr leichtgewichtig und ohne Aufhebens. SOAP ist für einen XML-RPC-Fan XML-RPC nach einem ausgedehnten Gelage.

7.1.3 SOAP-Standardisierungsinitiativen

[9] Die wohl unwiderstehlichste Möglichkeit, um den Kontrast zwischen der Einfachheit von XML-RPC und der Komplexität von SOAP aufzuzeigen, ist die Aufzählung der Kategorien SOAP-bezogener Standardisierungsinitiativen und der Anzahl von Initiativen pro Kategorie. (Unter der Adresse *http://www.innoq.com/resources/ws-standards-poster* finden Sie ein amüsantes Poster dieser Standards.) Tabelle 7.1 zeigt diese Liste.

Kategorie	**Anzahl Initiativen**
Interoperabilität	10
Sicherheit	10
Metadaten	9
Nachrichtenübermittlung	9
Geschäftsvorgänge	7
Resource	7
Translation	7
XML	7
Management	4
SOAP	3
Presentation	1

Tabelle 7.1: Standardisierungsinitiativen für SOAP-basierte Webservices.

[10] Die Summe von 74 verschiedenen, aber miteinander verwandten Initiativen unterstützt die Erläuterung der Anschuldigung, SOAP-basierte Webservices seien „zu Tode entwickelt“ worden. Ein Teil der Standardisierungen, zum Beispiel in der Kategorie Geschäftsvorgänge, repräsentieren eine Anstrengung, Webservices nützlicher und praxisgerechter zu machen. Andere, zum Beispiel in den Kategorien Interoperabilität, Sicherheit, Metadaten und SOAP selbst, unterstreichen das Ziel der Unabhängigkeit von Sprache, Plattform und Transportprotokoll SOAP-basierter Dienste. Rufen Sie sich die Beispiele zur gegenseitigen Authentifizierung mit digitalen Zertifikaten aus Kapitel 7 ins Gedächtnis zurück. Die gegenseitige Authentifizierung erfordert unter HTTPS nur minimale Initialisierungen und die ausgetauschten SOAP-Envelopes transportieren keine wie auch immer geartete Sicherheitsinformation. Bei WSIT ist die Initialisierung dagegen kompliziert und die ausgetauschten SOAP-Envelopes transportieren relative große komplexe Headerblocks mit gekapselten Berechtigungsnachweisen und anderen sicherheitrelevanten Informationen. Die WSIT-Komplexität läßt sich allerdings vermeiden, in diesem Fall, in dem die Authentifizierung an HTTPS delegiert wird. Trotz ihrer Kompliziertheit sind selbst SOAP-basierte Dienste ein evolutionärer Vorstoß.

7.2 SOAP-basierte Dienste im Vergleich mit verteilten Objekten

[11] *RMI (Remote Method Invocation)*, (inklusive der darauf aufbauenden Sitzungs- und Entitäts-EJBs) und .NET-Remoting sind Beispiele für Systeme verteilter Objekte der zweiten Generation. Betrachten Sie, was ein RMI-Client braucht, um eine im Interface eines Dienstes deklarierte Methode aufzurufen:

```
package ch07.doa;  // distributed object architecture
```

```
import java.util.List;

public interface BenefitsService extends java.rmi.Remote {
    public List<EmpBenefits> getBenefits(Emp emp) throws RemoteException;
}
```

Der Client braucht einen RMI-Stub, ein Objekt einer Klasse, die das Interface *BenefitsService* implementiert. Die Stubklasse wird, per Serialisierung über den Socket der Client und Server verbindet, automatisch auf der Clientseite heruntergeladen. Die Stubklasse benötigt die vom Programmierer definierten Typen `Emp` und `EmpBenefits`, die wiederum andere, vom Programmierer definierte Typen enthalten können, zum Beispiel `Department`, `BusinessCertification` und `ClientAccount`:

```
public class Emp {
    private Department                  department;
    private List<BusinessCertification> certification;
    private List<ClientAccount>         accounts;
    ...
}
```

Die Java-Standardtypen wie *List* sind bereits auf der Clientseite vorhanden, da der Client nach Voraussetzung eine Java-Applikation ist. Die Herausforderung betrifft die vom Programmierer definierte Typen wie `Emp` und `EmpBenefits`, die zur Unterstützung des clientseitigen Aufrufs einer entfernt ausgeführten Methode benötigt werden, zum Beispiel:

```
List<EmpBenefits> fred_benefits = remote_object.getBenefits(fred);
```

Bei Java-RMI werden die vom Programmierer definierte Typen automatisch vom entfernten Server heruntergeladen. Das Herunterladen von einem entfernten Server ist dennoch von Natur aus kompliziert und das Format der *.class* Datei von Java ist natürlich proprietär. Die Kernaussage besteht darin, daß vieles vom Server zum Client heruntergeladen werden muß, bevor der Client eine entfernte Methode aufrufen kann. Die an den Dienstv (hier das `Emp`-Objekt) übertragenen Argumente werden clientseitig serialisiert und serviceseitig deserialisiert. Der Rückgabewert, hier ein Containerobjekt mit `EmpBenefits`-Objekten, wird auf der Dienstseite serialisiert und auf der Clientseite deserialisiert. Client und Dienst kommunizieren über byte- statt zeichenorientierte Ströme und die Struktur des binären Stroms ist Java-spezifisch.

[12] SOAP-basierte Webservice vereinfachen die Dinge. Client und Dienst tauschen zum Beispiel nun XML-Dokumente aus, um nur einen Punkt zu nennen, also zeichenorientierte Daten in Form des SOAP-Envelopes. (In speziellen Situationen können auch „rohe Bytes“ ausgetauscht werden.) Der ausgetauschte Text kann eingesehen, überprüft, transformiert, dauerhaft gespeichert und anderweitig verarbeitet werden, wofür leicht verfügbare und häufig kostenlose Hilfsmittel verwendet werden können. Beide Seiten, Client und Server, brauchen lediglich eine lokale Softwarebibliothek, welche sprachspezifische Typen, wie den Java-Typ `String` an *sprachneutrale* XML-Schematypen (oder vergleichbare Typen) bindet, in diesem Fall `xsd:string`. Mit diesen Bindungen können relative einfache Bibliotheksmodule die Serialisierung und Deserialisierung ausführen. Die Verarbeitung setzt zu beiden Seiten lokal verfügbare Bibliotheken und Hilfsmittel voraus. Die Komplexität läßt sich in den Endpunkten isolieren und von den Endpunkten muß nichts in die ausgetauschten SOAP-Nachrichten „durchsickern“.

[13] Wie bereits wiederholt erwähnt, besteht kein Zwang Client oder Dienst in derselben Sprache oder gar im selben Sprachtyp zu schreiben. Clients und Dienste können in objektorientierten, prozeduralen, funktionalen oder sonstigen Sprachtypen implementiert werden. Die Sprachen zu beiden Seiten der Verbindung können statisch typisiert (zum Beispiel Java und C#) oder dynamisch typisiert (zum Beispiel Perl und Ruby) sein, wobei es allerdings einfacher ist, ein WSDL-Dokument für einen Dienst zu generieren, der in einer statischen Sprache geschrieben ist. Ferner isoliert SOAP Client und Server von plattformspezifischen Details. Das Leben wird durch den Umstieg von verteilten

Objekten zu SOAP-basierten Diensten einfacher und entsprechend besser.

7.3 SOAP und REST in Harmonie

[14] Aus der Perspektive des Programmierers gibt es zwei hochgradig verschiedene Möglichkeiten, einen SOAP-basierten Webservice zu implementieren. Die erste, in der Praxis vorherrschende Möglichtkeit besteht darin, HTTP/HTTPS als Transportprotokoll zu wählen und dem Transportprotokoll soviel Arbeit wie möglich zu überlassen. Das Transportprotokoll kümmert sich in diesem Fall um Anforderungen wie zuverlässige Nachrichtenübermittlung, gegenseitige Authentizierung und Geheimhaltung der transportierten Daten. Die zweite Möglichkeit erfordert einen hohen Anteil an den WS*-Initiativen und verlangt dem Programmierer die Auseinandersetzung mit den zahlreichen APIs für transportneutrale Dienst ab.

[15] Ich bevorzuge den ersten Weg. Mit HTTP/HTTPS als Transportprotokoll sind SOAP-basierte Webservices im vorherrschenden Request/Response-Muster nur ein Spezialfall der Dienste im REST-Stil: Eine Anfrage ist ein per `POST` versendeter SOAP-Envelope und die Antwort des Dienstes ist ebenfalls ein SOAP-Envelope. Der Programmier kommt zumeist nicht mit den SOAP-Envelopes in Berührung, der XML-Inhalte auf der Seite des Clients nicht generieren und auf der Seite des Dienstes nicht parsen muß. Die Clientbeispiele in Kapitel 4 „Dienste im REST-Stil“ veranschaulichen diesen Aspekt. Bei jedem Client fällt der XML-Verarbeitung den größte Teil des Quelltextes zu. *Transparentes XML* klingt nur für denjenigen wie ein leeres Motto, der nicht direkt mit der Verarbeitung großen und komplizierter XML-Dokumente in Berührung gekommen ist.

[16] Aus der Perspektive hat der SOAP-Ansatz durch das WSDL-Dokument, den Dienstkontrakt, einen enormen Reiz, da es verwendet werden kann, um clientseitige Artefakte zu generieren, welche die Aufgabe einen Client zu entwickeln standardisieren. Der Hauptnachteil bei Webservices im REST-Stil im Allgemeinen ist das Fehlen eines einheitlichen Dienstkontraktes, der die Programmierung des Clients erleichtert. Eine XML-Schema ist eine Grammatik, kein Dienstkontrakt. Die Grammatik kann den Programmierer beim Parsen eines XML-Dokumentes natürlich leiten, um die benötigten Daten zu extrahieren, ist aber nur ein geringer Trost, wenn jeder Dienst im REST-Stil sein eigenes Antwortformat verwendet, ganz zu schweigen von seiner eigenen Aufrufsyntax. Der nächste wichtige Schritt in der REST-Community ist die Einigung auf ein Format wie zum Beispiel das WADL-Dokument als Dienstkontrakt. Stellen Sie sich den Vorschub vor, den der REST-Stil durch einen leichtgewichtigen Dienstkontrakt erfahren würde, der zum erzeugen clientseitiger Artefakte verwendet werden kann.

[17] Die Ansätze SOAP und REST präsentieren unterschiedliche Anregungen bieten dadurch aber auch unterschiedliche Möglichkeiten. Der SOAP-Ansatz verbirgt XML, während XML beim REST-Ansatz allgegenwärtig ist. Der SOAP-basierte Ansatz nutzt die Vorteile von HTTP nicht direkt, auch wenn der SOAP-Envelope bei SOAP-basierten Diensten fast immer über HTTP/HTTPS transportiert wird. Der REST-Ansatz nutzt offensichtlich, was HTTP/HTTPS bietet, tut bis jetzt allerdings wenig, um die Verarbeitung der XML-Nutzdaten zu erleichtern.

[18] Hoffen wir, daß sich ein einheitlicher Ansatz für Webservices im REST-Stil zeigt. Bis dahin möchte ich mit meiner unbedeutenden Meinung Webservices im REST-Stil unterstützen — und SOAP-basierte Dienst über HTTP/HTTPS.

Index

Über den Autor

Martin Kalin hat einen Doktortitel von der Northwestern University und ist Professor am College of Computing and Digital Media der DePaul University. Er ist Mitautor mehrerer Bücher über C und C++ und hat ein Buch über Java für Programmierer geschrieben. Kalin interessiert sich für kommerzielles Programmieren und hat große verteilte Systeme in den Bereichen process scheduling und product configuration mitentwickelt.

Kolophon

Das Tier auf dem Einband von *Java Web Services: Up and Running* ist ein Kormoran (Phalacrocorax carbo). Dieser Vogel ist ein häufiger Vertreter der Familie der Phalacrocoracidae, welche etwa vierzig Gattungen von Kormoranen umfaßt — große Seevögel mit hakenförmigem Schnabel, bunten Hälsen und steifen Schwanzfedern. Der Name „Kormoran“ ist vom lateinischen *corvus marinus* („Seerabe“) abgeleitet, aufgrund seines schwarzen Federkleides.

Ein ausgewachsener Kormoran wird bis zu einem Meter lang und hat eine Flügelspannweite bis zu zwei Meter. Er hat einen langen Hals, eine gelbe Kehle und einen weißen Fleck am Kinn. Der Kormoran hat eine bestimmte Art, während des Sitzens seine Flügel zu strecken, der unter Naturbeobachtern als Trocknen der Federn gedeutet wird, obwohl diese Erklärung diskutiert wird. Der Kormoran lebt hauptsächlich in der Nähe des atlantischen Ozeans, an der Westküste Europas sowie an der Ostküste Nordamerikas und insbesondere in den maritimen Zonen Kanadas. Er brütet in Felsenklippen und Bäumen, wobei er Nester aus dünnen Ästen und Seetang baut.

Der Komoran ist ein ausgezeichneter Fischer und kann bis in beträchtliche Tiefen tauchen. Chinesische und japanische Fischer dressieren Koromane mit einer Jahrhunderte alten Methode, bei der sie einen Strick um den Hals des Vogels legen, um ihn am Hinunterschlucken der Beute zu hindern und schicken ihn von den Booten aus ins Wasser. Die Kormorane fangen nun Fische in ihren Schnäbeln und kehren zum Boot zurück, wo die Fischer den Fang entgegennehmen. Obwohl einst erfolgreich, ist der Fischfang mit Kormoranen heute hauptsächlich eine Touristenattraktion.

Das Bild auf dem Einband stammt aus Cassells *Popular Natural History, Vol. III: Birds*. Der Zeichensatz auf dem Einband ist Adobe ITC Garamond. Der Zeichensatz des Textes ist Linotype Birka, die Überschriften sind in Adobe Myriad Condensed gesetzt und die Quelltexte in LucaFonts TheSansMonoCondensed.